

6.034 Notes: Section 1.1

Slide 1.1.1

This is a brief introduction to the content and organization of 6.034.

6.034 Artificial Intelligence

- Topics covered
- Prerequisites
- How the subject works
- Grading
- Collaboration Policy
- Check the course page often.

Tip • Spring 02 • 1 

Topics

The course covers three major topics:

- Search
 - Graph search
 - Constraint Satisfaction
 - Games
- Machine Learning
 - Nearest Neighbors
 - Decision Trees
 - Neural Networks
 - SVM
- Knowledge Representation & Inference
 - Propositional & First Order Logic
 - Rule-based systems
 - Natural Language

Tip • Spring 02 • 2 

Slide 1.1.2

These are the topics that we will cover during the semester. 6.034 is an introductory subject. Our goal is to give you a solid introduction to three key topics: search, knowledge representation and inference, and machine learning. We will introduce a variety of other different topics in AI, such as planning, robotics and natural language only in passing. Subsequent courses in AI cover those areas in more depth.

Slide 1.1.3

These are the formal and informal prerequisites for the subject.

6.001 is an essential prerequisite. In particular, we expect you to read and understand substantial Scheme programs and to make small modifications to the code. Remember, this is a subject in computer science. Programming is to CS as calculus is to physics and EE; it is the essential language for making the ideas concrete. Also, practice makes perfect and you should take every opportunity to practice programming. Scheme is the language that we can count on everyone having from 6.001, so we use it heavily. It is also highly suitable for many (though not all) of the topics covered in this subject. If you're going to study computer science, you should take mastering programming languages in stride.

We will assume that you know basic differential calculus of several variables and vector algebra, such as covered in 18.02. You will not be able to understand machine learning without this basic mathematical background.

Prerequisites

• 6.001

We will have regular assignments that expect you to be able to read and write Scheme. This is the only formal pre-requisite.

• 18.02

We will assume that you know what the chain rule is and what a dot product is, and a partial derivative, etc. If you have not taken 18.02, you should really wait to take the subject until you have.

Tip • Spring 02 • 3 

Course organization

- 2 x 1.5 hr classes (MW11-12:30)
- 1 recitation with TA
- On-line text + exercises
 - Recommended book (available at Quantum & Amazon): Russell & Norvig, AI: A Modern Approach 2nd edition
 - This book is **only** for supplementary reading; all of the course material is covered in the notes.
- On-line problem set
- 2 Design problems (3-5 page papers)
- 2 in-class quizzes (March 7, April 4)
- Final

lfp • Spring 02 • 4

Slide 1.1.4

The class meets as a whole twice a week for 1 1/2 hours. The lectures will introduce the basic material for the course. These on-line chapters are the textbook for the course; you are responsible for the material in the on-line text. Most weeks there will be a required on-line problem set.

We will schedule meetings with the TAs on Friday in groups of about 20 students. These recitation meetings will go over the on-line problem set problems. The meetings are required unless you've already completed all the on-line problems correctly by Friday.

The recommended book provides a wealth of additional materials and in-depth exploration of the topics we will cover. Although the book is **not** required, you might find it very helpful if you want to dig deeper into the material.

We will have two in-class quizzes (held in-class on the indicated dates) and a final.

Slide 1.1.5

The grading is broken down as shown here. The only aspect that requires some comment is the treatment of the on-line problems. I feel that you can't really learn the material without doing the problems. You may even be able to do well on the exams but you won't have a full understanding without working the problems. And so, the problems are required.

Since most (though not all) of the problems have "Check" buttons, you should be able to get full credit for them. Some of the problems don't have Check buttons and so you won't necessarily get 100% on every problem set. On the other hand, we don't expect you to get a score of 100%; 90% is sufficient for full credit. You won't get a better final grade because you got 100% on the problems. So, don't obsess about losing a point here and there. The objective of the problems and exercises is to help you learn the material and make sure that you understand what we view as key points.

Collaboration

- **Everything** you do for credit in this subject is supposed to be your own work; this includes on-line work.
- You can talk to other students (and TAs) about approaches to problems, but then you should sit down and do the problem yourself. This is not only the ethical way but also the only effective way of learning the material.

lfp • Spring 02 • 6

Slide 1.1.6

Don't hand in work that you did not do, even to the on-line system. Talking to other people to try to understand the material is fine, in fact, encouraged. Cutting and pasting someone else's answer is NOT fine under any circumstances. Not only is it unethical but you will fail to learn anything in the course.

If you feel so pressured that you are tempted to turn in someone else's work, you are probably trying to do too much. You should probably be taking fewer subjects or cutting back somewhere else. Speak to your advisor or a counseling Dean or come talk to one of us.

lfp • Spring 02 • 5

Grading

- 30% Final
- 30% Quizzes
- 25% On-line assignments + Recitation participation
- 15% Design problems
- The on-line exercises and problems are an essential component of the subject and are required. A 90% score on any on-line assignment gets full credit. There is no difference between 90% and 100%. An average score below 75% will lead to a grade of Incomplete in the subject.
- On-line work that is submitted late will receive half credit unless you have a valid reason and make an arrangement with your TA.

6.034 Notes: Section 1.2

Slide 1.2.1

This section provides you with a quick overview of the on-line system that we will be using in the class. If you've used the On-Line Tutor in 6.001, you should be familiar with the operation of this system in spite of the slightly different look.

6.034 Artificial Intelligence

- Intro to On-Line Interactive Text
- On-Line Problems & Exercises
- Suggestions/Issues

lip · Spring 02 · 1 

On-Line Interactive Text

- We will use this on-line format to introduce the detailed material for the course. We will also point you to additional material in the suggested textbooks.
- Each of the presentations comes in two forms:
 - Slides, Narration and Narration Text (on-line)
 - PDF of Slides and Narration Text (for printing)
- Associated with each presentation there will be a few exercises to drive home key points.
- Each week, there will also be a set of assigned problems (including programming ones) that will be done on-line.

lip · Spring 02 · 2 

Slide 1.2.2

These presentations will tend to focus on the nitty-gritty detail of the material and be a bit skimpy on motivation - we will do more of that in class. We have tried to provide a variety of ways of going through the material, either on-line or for printing; hopefully you will find one that suits you.

Note that each assigned chapter will have several presentations (sections) as well as some interactive exercise problems. You should do these correctly after going through the appropriate section - this should help make sure that you picked up on the key points of the section.

We also have more substantial interactive problems, including programming problems, in the week's problem set.

On-line problems and exercises

- Problems and exercises come in three forms:
 - Multiple Choice/True False
 - Short answer
 - Coding
- Short-answer and coding problems have a "Check" button that allows you to verify whether your answer is correct before final submission. Multiple-choice/True-False problems do not have a Check button.
- When you are done with a problem, you need to click the "Submit" button. This will show you the "official" answer. You need to submit every problem before the due date to get full credit. After submitting a problem, you cannot change your answers to that problem.
- If you submit every problem and exercise in a problem set and you score 90% or better, you should get a gold star for the problem set. If your score is 90% or better but you have no star, then you forgot to submit some problem – maybe the hours or feedback “problems”.

lip · Spring 02 · 3 

On-line problems and exercises

- Some problems have multiple parts. There is a single submit button for multi-part problems. Be careful that you do not submit a problem before you have done all the parts. It is useful to check your score before submitting (there is a button to get Scores on every page).
- If you want to see an on-line presentation while working on a problem, you can open a new page and navigate to the appropriate chapter. Modern browsers allow you to open a new page (Ctrl-N) directly. Note that you need to do this since the system does not allow you to login multiple times (a security feature).

lip · Spring 02 · 4 

Slide 1.2.4

Some problems are multi-part with each part being of one of the types discussed earlier. These problems will have only one Submit button; make sure that you do all of the parts before submitting.

On a separate point, it is often useful to have a page open to one of the presentations while working on a problem. To do this, use Ctrl-N in most modern browsers.

Slide 1.2.5

Some problems can be solved by guessing using the Check button. This is not recommended; if it comes up on an exam, you won't have a Check button handy.

Occasionally, the system will mark wrong an answer that you know is right. Please let us know right away so that we can fix it (although probably not at 3am). Consider it your humanitarian duty towards your fellow students. Don't waste your time trying to figure out what the system is looking for. Of course, before sending us e-mail, please make sure that you actually read the problem...

On-line problems and exercises

- Some on-line exercises and problems can be solved by guessing multiple times, but you probably won't learn anything that way.
- If you have trouble with the system recognizing an answer that you're sure is correct, please let us know right away via e-mail (rather than wasting time trying to guess what the system is looking for). We will give you credit for your answer (and update the legal answers so it works correctly for others).

tip • Spring 02 • 5



Preferences/Options

- You can set a number of options, e.g. whether to have the system verify each problem submission or whether it should play the sound with each slide, from the Options page.
- Note that you need to press Submit (at the bottom of the preferences page) for any change to take effect.

tip • Spring 02 • 6

Slide 1.2.6

Click on the Preferences button to see what options are available. Hopefully the defaults are set sensibly but the Preferences are generally there in response to requests from previous users.

Slide 1.2.7

We'd like to be able to address any issues that you have with the material, the organization of the course, or the on-line system. To that end, we have provided you with several mechanisms to try to get your feedback. Of course, you can always send email about any of this to me or to any of the TAs. There's an email link at the bottom of every page that reaches all of us. In addition, each chapter and problem set has a question that's explicitly geared to getting feedback on the current material.

Of course, you can always ask questions in class. But, we know that many of you will not ask a question in class if your life depended on it. So, let's see if the technology can help.

Questions and Suggestions

- We have added a question to each presentation and problem set asking for any issues that you would like to see discussed in class. Please use that to give us feedback so that we can make the class time maximally useful.
- Each on-line page has an e-mail link at the bottom. Please use that to ask any questions that you have on the presented material or on the problems or exercises, especially when the next class will not be for a while.
- If you have any feedback or suggestions about the course, you can use any of these methods to communicate them to us. There is nothing we can do about complaints at the end of the term; we may actually be able to deal with an issue if you let us know during the term.

tip • Spring 02 • 7



Scheme

- There will be coding problems throughout the term
- In many cases, it will be feasible to simply debug your answer on the on-line system using the Check button.
- In other case, you should develop your code on a stand-alone Scheme system, such as MIT Scheme, which hopefully you know from 6.001. The course homepage has a pointer to additional information on Scheme systems.
- We encourage you to install the most recent full release of MIT Scheme on your machine, or use it from the 6.034 locker on Athena. Do not rely on the 6.001 release.
- MIT Scheme can be found at:
<http://www.gnu.org/software/mit-scheme/>

tip • Spring 02 • 8

Slide 1.2.8

You will need to use Scheme in this subject. For short problems, you might be able to debug directly on the on-line system - although the underlying Scheme is not very good about error messages and interactive debugging is out of the question.

For any substantial programming, especially for the projects, you will need a stand-alone system. On Intel x86 machines (running Windows, Linux, FreeBSD), the MIT Scheme system works pretty well. However, you should install the most recent complete release of MIT Scheme - don't rely on the one from 6.001.

If for some reason you don't have regular access to an Intel x86 machine, there are a couple of other options mentioned on our Web page, but we make no guarantees.

Slide 1.2.9

If you're experiencing any technical problems, please send us e-mail with helpful information, such as the HTML source of the page showing the problem and a description of your environment.

If you are using an old Netscape and you get a blank page that doesn't mean there is no HTML there. What it means is that the Netscape HTML renderer has decided that the HTML on the page is not perfect and it's not worth rendering. Unfortunately, there's usually an informative error message lurking in there. So, if faced with a blank page, get the HTML source and, if there's anything there, send it along to us. It's likely to help us catch the bug.

We had someone in the end-of-term survey say "Halfway through the term the sound stopped working", but this person never told anyone who could do anything about it that they were having problems! I guess they assumed that it had stopped working for everyone and someone else would report it. Well it didn't fail for everyone and no one reported it. We might not be able to solve all your problems, given the immense variety of incompatible or broken software and hardware out there, but we definitely can't solve any of your problems if you don't tell us about them.

Crash?

- If your browser or machine crashes while running our on-line system, then most likely you've got a broken OS/Browser/JavaVM combination (they're out there).
- Our code already has patches to avoid the more common landmines but there are many more trained professionals in industry producing buggy code for your browser and OS than there are of us trying to patch around them.
- So, please report these problems but odds are we won't even be able to reproduce them. Luckily, these problems seem to be quite rare.
- Upgrading/Reinstalling the browser sometimes helps. Finding a new machine almost always helps ;-)
- Or, switch to using an HTML form for the code entry window (see Preferences). This most likely will work – the problem, when it occurs, is usually with Java .

tip • Spring 02 • 10

Slide 1.2.10

This is a bit of a 6.001 review. Your browser has interpreters for a few languages, specifically HTML, Javascript and Java. You learned about interpreters in 6.001. So, suppose I write a Scheme program for a Scheme interpreter and, when I run it, the interpreter crashes and burns, whose fault is this? Is it my fault for writing such a nasty Scheme program? Or, do we have a buggy interpreter on our hands? If you said it was the fault of the program, you should go back and review 6.001 and try again.

Similarly, if you feel that your browser (or your OS) crashing is due to the little piece of Java (or HTML or Javascript) code in our on-line system, try again. I can pretty much guarantee you that it is because you have some buggy combination of browser and Java Virtual Machine installed on your machine; there are plenty of these around. I am confident of this because, by design, I'm not supposed to be able to write Java (or HTML or Javascript) code that crashes your browser (or, heaven forbid, your machine). Note that I am not promising that all our code will work correctly (I wish!). But crashing your browser or your machine should be beyond our power.

Unfortunately, it means that these problems are also likely to be beyond our power to fix. We'd like to know when they happen (which seems to be mercifully rarely), but it's not clear that we will be able to help since we most likely won't even be able to reproduce it. It's likely something about your

particular setup. So, you'll need to change the setup. In any case, do let us know if you experience problems.

Bug Reports

- For technical problems, use the email address at the foot of any of the pages to let us know. Please include the HTML source of the page (Right Click and View Source) and let us know your environment (Linux/Windows, Mozilla/Netscape/IE or whatever).
- If you get a blank page in an old Netscape, still do View Source and send that (old Netscapes are finicky about rendering ill-formed HTML). Better still, get a modern browser.
- On Macs, the Java applet for entering code seems to work best on Safari.

tip • Spring 02 • 9



6.034 Notes: Section 2.1

Slide 2.1.1

Search plays a key role in many parts of AI. These algorithms provide the conceptual backbone of almost every approach to the systematic exploration of alternatives.

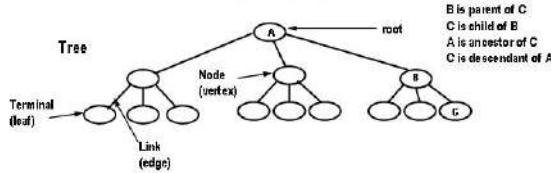
We will start with some background, terminology and basic implementation strategies and then cover four classes of search algorithms, which differ along two dimensions: First, is the difference between **uninformed** (also known as **blind**) search and then **informed** (also known as **heuristic**) searches. Informed searches have access to task-specific information that can be used to make the search process more efficient. The other difference is between **any path** searches and **optimal** searches. Optimal searches are looking for the best possible path while any-path searches will just settle for finding some solution.

6.034 Artificial Intelligence

- **Big idea:** Search allows exploring alternatives
- **Background**
- **Uninformed vs Informed**
- **Any Path vs Optimal Path**
- **Implementation and Performance**

10 / 50 12 / 1

Trees and Graphs



Slide 2.1.2

The search methods we will be dealing with are defined on trees and graphs, so we need to fix on some terminology for these structures:

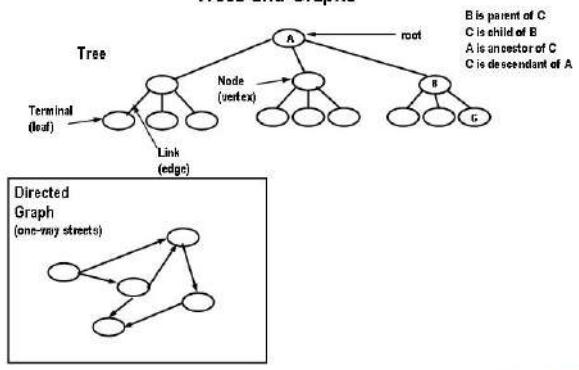
- A tree is made up of **nodes** and **links** (circles and lines) connected so that there are no loops (cycles). Nodes are sometimes referred to as vertices and links as edges (this is more common in talking about graphs).
- A tree has a **root node** (where the tree "starts"). Every node except the root has a single **parent** (aka **direct ancestor**). More generally, an **ancestor** node is a node that can be reached by repeatedly going to a parent node. Each node (except the **terminal** (aka **leaf**) nodes) has one or more **children** (aka **direct descendants**). More generally, a **descendant** node is a node that can be reached by repeatedly going to a child node.

10 / 50 12 / 1

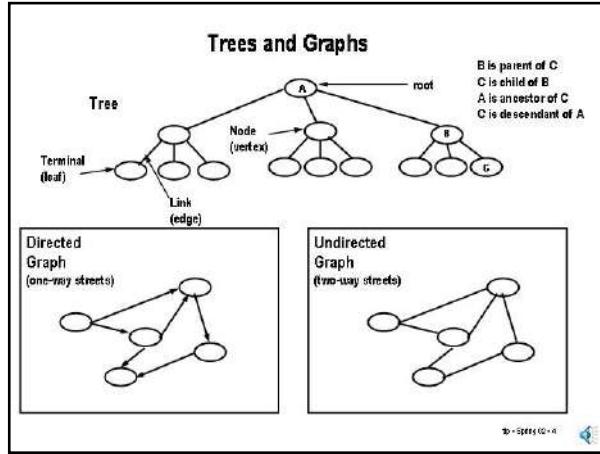
Slide 2.1.3

A graph is also a set of nodes connected by links but where loops are allowed and a node can have multiple parents. We have two kinds of graphs to deal with: **directed** graphs, where the links have direction (akin to one-way streets).

Trees and Graphs



10 / 50 12 / 1

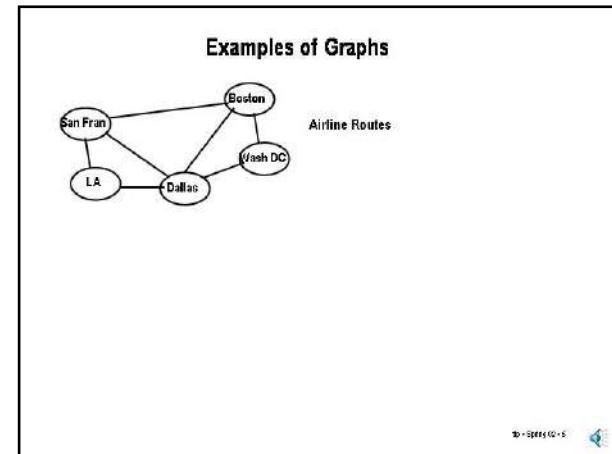


Slide 2.1.4

And, **undirected** graphs where the links go both ways. You can think of an undirected graph as shorthand for a graph with directed links going each way between connected nodes.

Slide 2.1.5

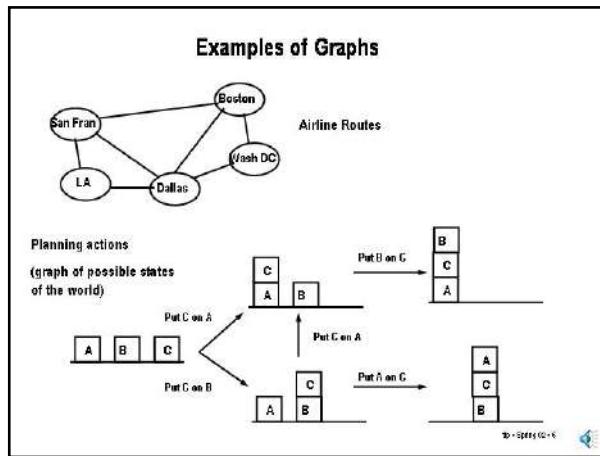
Graphs are everywhere; for example, think about road networks or airline routes or computer networks. In all of these cases we might be interested in finding a path through the graph that satisfies some property. It may be that any path will do or we may be interested in a path having the fewest "hops" or a least cost path assuming the hops are not all equivalent, etc.



Slide 2.1.6

However, graphs can also be much more abstract. Think of the graph defined as follows: the nodes denote descriptions of a state of the world, e.g., which blocks are on top of what in a blocks scene, and where the links represent actions that change from one state to the other.

A path through such a graph (from a start node to a goal node) is a "plan of action" to achieve some desired goal state from some known starting state. It is this type of graph that is of more general interest in AI.

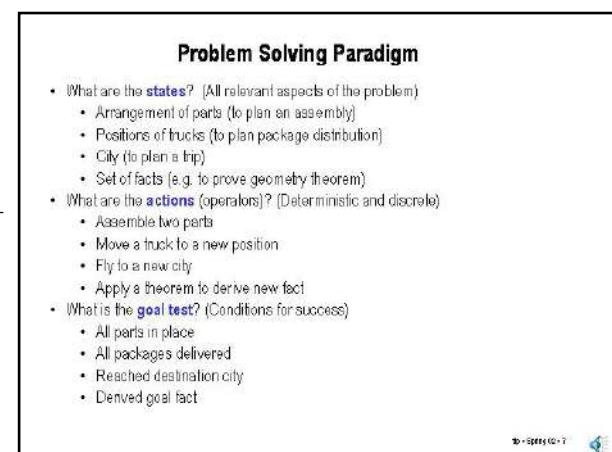


Slide 2.1.7

One general approach to problem solving in AI is to reduce the problem to be solved to one of searching a graph. To use this approach, we must specify what are the **states**, the **actions** and the **goal test**.

A state is supposed to be **complete**, that is, to represent all (and preferably only) the relevant aspects of the problem to be solved. So, for example, when we are planning the cheapest round-the-world flight plan, we don't need to know the address of the airports; knowing the identity of the airport is enough. The address will be important, however, when planning how to get from the hotel to the airport. Note that, in general, to plan an air route we need to know the airport, not just the city, since some cities have multiple airports.

We are assuming that the actions are **deterministic**, that is, we know exactly the state after the action is performed. We also assume that the actions are **discrete**, so we don't have to represent what happens while the action is happening. For example, we assume that a flight gets us to the scheduled destination and that what happens during the flight does not matter (at least when planning the route).

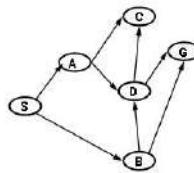


Note that we've indicated that (in general) we need a test for the goal, not just one specific goal state. So, for example, we might be interested in any city in Germany rather than specifically Frankfurt. Or, when proving a theorem, all we care is about knowing one fact in our current data base of facts. Any final set of facts that contains the desired fact is a proof.

In principle, we could also have multiple starting states, for example, if we have some uncertainty about the starting state. But, for now, we are not addressing issues of uncertainty either in the starting state or in the result of the actions.

Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having ≤ 1 parent
- We can turn graph search problems into tree search problems by:
 - replacing undirected links by 2 directed links
 - avoiding loops in path (or keeping track of visited nodes globally)



Slide 2.1.8

Note that trees are a subclass of directed graphs (even when not shown with arrows on the links). Trees don't have cycles and every node has a single parent (or is the root). Cycles are bad for searching, since, obviously, you don't want to go round and round getting nowhere.

When asked to search a graph, we can construct an equivalent problem of searching a tree by doing two things: turning undirected links into two directed links; and, more importantly, making sure we never consider a path with a loop or, even better, by never visiting the same node twice.

Slide 2.1.9

You can see an example of this converting from a graph to a tree here. If we assume that S is the start of our search and we are trying to find a path to G, then we can walk through the graph and make connections from every node to every connected node that would not create a cycle (and stop whenever we hit G). Note that such a tree has a leaf node for every non-looping path in the graph starting at S.

Also note, however, that even though we avoided loops, some nodes (the colored ones) are duplicated in the tree, that is, they were reached along different non-looping paths. This means that a complete search of this tree might do extra work.

The issue of how much effort to place in avoiding loops and avoiding extra visits to nodes is an important one that we will revisit later when we discuss the various search algorithms.

Terminology

- **State** – Used to refer to the vertices of the underlying **graph** that is being searched, that is, states in the problem domain, for example, a city, an arrangement of blocks or the arrangement of parts in a puzzle.
- **Search Node** – Refers to the vertices of the **search tree** which is being generated by the search algorithm. Each node refers to a state of the world; **many nodes may refer to the same state**. Importantly, a node implicitly represents a path (from the start state of the search to the state associated with the node). Because search nodes are part of a search tree, they have a unique ancestor node (except for the root node).

Slide 2.1.10

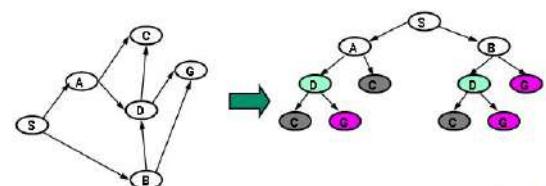
One important distinction that will help us keep things straight is that between a **state** and a **search node**.

A state is an arrangement of the real world (or at least our model of it). We assume that there is an underlying "real" state graph that we are searching (although it might not be explicitly represented in the computer; it may be implicitly defined by the actions). We assume that you can arrive at the same real world state by multiple routes, that is, by different sequences of actions.

A search node, on the other hand, is a data structure in the search algorithm, which constructs an explicit tree of nodes while searching. Each node refers to some state, but not uniquely. Note that a node also corresponds to a path from the start state to the state associated with the node. This follows from the fact that the search algorithm is generating a **tree**. So, if we return a node, we're returning a path.

Graph Search as Tree Search

- Trees are directed graphs without cycles and with nodes having ≤ 1 parent
- We can turn graph search problems (from S to G) into tree search problems by:
 - replacing undirected links by 2 directed links
 - avoiding loops in path (or keeping track of visited nodes globally)



6.034 Notes: Section 2.2

Slide 2.2.1

So, let's look at the different classes of search algorithms that we will be exploring. The simplest class is that of the **uninformed, any-path** algorithms. In particular, we will look at **depth-first** and **breadth-first** search. Both of these algorithms basically look at all the nodes in the search tree in a specific order (independent of the goal) and stop when they find the first path to a goal state.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.

10 / 30 06/12/1



Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.

10 / 30 06/12/1

Slide 2.2.2

The next class of methods are **informed, any-path** algorithms. The key idea here is to exploit a task specific measure of goodness to try to either reach the goal more quickly or find a more desirable goal state.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.

10 / 30 06/12/1



Slide 2.2.3

Next, we look at the class of **uninformed, optimal** algorithms. These methods guarantee finding the "best" path (as measured by the sum of weights on the graph edges) but do not use any information beyond what is in the graph definition.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	A*	Uses path "length" measure and heuristic. Finds "shortest" path.

10 / 30 06/12/1

Slide 2.2.4

Finally, we look at **informed, optimal** algorithms, which also guarantee finding the best path but which exploit heuristic ("rule of thumb") information to find the path faster than the uninformed methods.

Slide 2.2.5

The search strategies we will look at are all instances of a common search algorithm, which is shown here. The basic idea is to keep a list (Q) of nodes (that is, partial paths), then to pick one such node from Q , see if it reaches the goal and otherwise extend that path to its neighbors and add them back to Q . Except for details, that's all there is to it.

Note, by the way, that we are keeping track of the states we have reached (visited) and not entering them in Q more than once. This will certainly keep us from ever looping, no matter how the underlying graph is connected, since we can only ever reach a state once. We will explore the impact of this decision later.

Simple Search Algorithm

A search node is a path from some state X to the start state, e.g., (X B A S). The state of a search node is the most recent state of the path, e.g. X. Let Q be a list of search nodes, e.g. [(X B A S) (C B A S) ...]. Let S be the start state.

1. Initialize Q with search node (S) as only entry; set Visited = (S)
2. If Q is empty, fail. Else, pick some search node N from Q
3. If state(N) is a goal, return N (we've reached the goal)
4. (Otherwise) Remove N from Q
5. Find all the descendants of state(N) not in Visited and creates all the one-step extensions of N to each descendant.
6. Add the extended paths to Q ; add children of state(N) to Visited
7. Go to step 2.

10 - SP14 CS-5

Simple Search Algorithm

A search node is a path from some state X to the start state, e.g., (X B A S). The state of a search node is the most recent state of the path, e.g. X. Let Q be a list of search nodes, e.g. [(X B A S) (C B A S) ...]. Let S be the start state.

1. Initialize Q with search node (S) as only entry; set Visited = (S)
2. If Q is empty, fail. Else, pick some search node N from Q
3. If state(N) is a goal, return N (we've reached the goal)
4. (Otherwise) Remove N from Q
5. Find all the children of state(N) not in Visited and create all the one-step extensions of N to each descendant.
6. Add the extended paths to Q ; add children of state(N) to Visited
7. Go to step 2.

Critical decisions:Step 2: picking N from Q .Step 6: adding extensions of N to Q

10 - SP14 CS-5

Slide 2.2.6

The key questions, of course, are *which* entry to pick off of Q and how precisely to add the new paths back onto Q . Different choices for these operations produce the various search strategies.

Slide 2.2.7

At this point, we are ready to actually look at a specific search. For example, **depth-first search** always looks at the deepest node in the search tree first. We can get that behavior by:

- picking the first element of Q as the node to test and extend.
- adding the new (extended) paths to the FRONT of Q , so that the next path to be examined will be one of the extensions of the current path to one of the descendants of that node's state.

One good thing about depth-first search is that Q never gets very big. We will look at this in more detail later, but it's fairly easy to see that the size of the Q depends on the depth of the search tree and not on its breadth.

Implementing the Search Strategies**Depth-first:**Pick first element of Q Add path extensions to front of Q

10 - SP14 CS-5

Implementing the Search Strategies**Depth-first:**Pick first element of Q Add path extensions to front of Q **Breadth-first:**Pick first element of Q Add path extensions to end of Q **Slide 2.2.8**

Breadth-first is the other major type of uninformed (or blind) search. The basic approach is to once again pick the first element of Q to examine BUT now we place the extended paths at the back of Q . This means that the next path pulled off of Q will typically not be a descendant of the current one, but rather one at the same level in tree.

Note that in breadth-first search, Q gets very big because we postpone looking at longer paths (that go to the next level) until we have finished looking at all the paths at one level.

We'll look at how to implement other search strategies in just a bit. But, first, let's look at some of the more subtle issues in the implementation.

10 - SP14 CS-5

Slide 2.2.9

One subtle point is where in the algorithm one tests for success (that is, the goal test). There are two plausible points: one is when a path is extended and it reaches a goal, the other is when a path is pulled off of Q . We have chosen the latter (testing in step 3 of the algorithm) because it will generalize more readily to optimal searches. However, testing on extension is correct and will save some work for any-path searches.

Testing for the Goal

- This algorithm stops (in step 3) when $\text{state}(N) = G$ or, in general, when $\text{state}(N)$ satisfies the goal test.
- We could have performed this test in step 6 as each extended path is added to Q . This would catch termination earlier and be perfectly correct for the searches we have covered so far.
- However, performing the test in step 6 will be incorrect for the optimal searches. We have chosen to leave the test in step 3 to maintain uniformity with these future searches.

10 - Sprig (2) - 0

**Terminology**

- **Visited** – a state M is first visited when a path to M first gets added to Q . In general, a state is said to have been visited if it has ever shown up in a search node in Q . The intuition is that we have briefly “visited” them to place them on Q , but we have not yet examined them carefully.

10 - Sprig (2) - 10

Slide 2.2.10

At this point, we need to agree on more terminology that will play a key role in the rest of our discussion of search.

Let's start with the notion of **Visited** as opposed to **Expanded**. We say a state is visited when a path that reaches that state (that is, a node that refers to that state) gets added to Q . So, if the state is anywhere in any node in Q , it has been visited. Note that this is true even if no path to that state has been taken off of Q .

Terminology

- **Visited** – a state M is first visited when a path to M first gets added to Q . In general, a state is said to have been visited if it has ever shown up in a search node in Q . The intuition is that we have briefly “visited” them to place them on Q , but we have not yet generated its descendants.
- **Expanded** – a state M is expanded when it is the state of a search node that is pulled off of Q . At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M (instead of M itself) as being expanded. However, once a node is expanded we are done with it; we will not need to expand it again. In fact, we discard it from Q .

10 - Sprig (2) - 11

**Terminology**

- **Visited** – a state M is first visited when a path to M first gets added to Q . In general, a state is said to have been visited if it has ever shown up in a search node in Q . The intuition is that we have briefly “visited” them to place them on Q , but we have not yet examined them carefully.
- **Expanded** – a state M is expanded when it is the state of a search node that is pulled off of Q . At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M (instead of M itself) as being expanded. However, once a node is expanded we are done with it; we will not need to expand it again. In fact, we discard it from Q .
- This distinction plays a key role in our discussion of the various search algorithms; study it carefully.

10 - Sprig (2) - 12

Slide 2.2.12

Please try to get this distinction straight; it will save you no end of grief.

Slide 2.2.13

In our description of the simple search algorithm, we made use of a Visited list. This is a list of all the states corresponding to any node ever added to Q. As we mentioned earlier, avoiding nodes on the visited list will certainly keep us from looping, even if the graph has loops in it. Note that this mechanism is stronger than just avoiding loops locally in every path; this is a global mechanism across all paths. In fact, it is more general than a loop check on each path, since by definition a loop will involve visiting a state more than once.

But, in addition to avoiding loops, the Visited list will mean that our search will never expand a state more than once. The basic idea is that we do not need to search for a path from any state to the goal more than once. If we did not find a path the first time we tried it, one is not going to materialize the second time. And, it saves work, possibly an enormous amount, not to look again. More on this later.

Visited States

- Keeping track of visited states generally improves time efficiency when searching graphs, without affecting correctness. Note, however, that substantial additional space may be required to keep track of visited states.
- If all we want to do is find a path from the start to the goal, there is no advantage to adding a search node whose state is already the state of another search node.
- Any state reachable from the node the second time would have been reachable from that node the first time.
- Note that, when using Visited, each state will only ever have at most one path to it (search node) in Q.
- We'll have to revisit this issue when we look at optimal searching.

19 - Spring 02 - 16

**Implementation Issues: The Visited list**

- Although we speak of a **Visited list**, this is never the preferred implementation.
- If the graph states are known ahead of time as an explicit set, then space is allocated in the state itself to keep a mark, which makes both adding to Visited and checking if a state is Visited a constant time operation.
- Alternatively, as is more common in AI, if the states are generated on the fly, then a hash table may be used for efficient detection of previously visited states.
- Note that, in any case, the incremental space cost of a Visited list will be proportional to the number of states – which can be very high in some problems.

19 - Spring 02 - 16

Slide 2.2.14

A word on implementation: Although we speak of a "Visited list", it is never a good idea to keep track of visited states using a list, since we will continually be checking to see if some particular state is on the list, which will require scanning the list. Instead, we want to use some mechanism that takes roughly constant time. If we have a data structure for the states, we can simply include a "flag" bit indicating whether the state has been visited. In general, one can use a hash table, a data structure that allows us to check if some state has been visited in roughly constant time, independent of the size of the table. Still, no matter how fast we make the access, this table will still require additional space to store. We will see later that this can make the cost of using a Visited list prohibitive for very large problems.

Terminology

- **Heuristic** – The word generally refers to a "rule of thumb," something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal."

19 - Spring 02 - 16

**Terminology**

- **Heuristic** – The word generally refers to a "rule of thumb," something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal."
- **Heuristic function** – In search terms, a function that computes a value for a state (but does not depend on any path to that state) that may be helpful in guiding the search. There are two related forms of heuristic guidance that one sees:

19 - Spring 02 - 16

Slide 2.2.16

A heuristic function has similar connotations. It refers to a function (defined on a state - not on a path) that may be helpful in guiding search but which is not guaranteed to produce the desired outcome. Heuristic searches generally make no guarantees on shortest paths or best anything (even when they are called best-first). Nevertheless, using heuristic functions may still provide help by speeding up, at least on average, the process of finding a goal.

Slide 2.2.17

If we can get some estimate of the "distance" to a goal from the current node and we introduce a preference for nodes closer to the goal, then there is a good chance that the search will terminate more quickly. This intuition is clear when thinking about "airline" (as-the-crow-flies) distance to guide a search in Euclidean space, but it generalizes to more abstract situations (as we will see).

Terminology

- **Heuristic** – The word generally refers to a "rule of thumb," something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal."
- **Heuristic function** – In search terms, a function that computes a value for a state (but does not depend on any path to that state) that may be helpful in guiding the search.
- **Estimated distance to goal** – this type of heuristic function depends on the state and the goal. The classic example is straight-line distance used as an estimate for actual distance in a road network. This type of information can help increase the efficiency of a search.

19 - Spring 02 - 17

**Implementing the Search Strategies****Depth-first**

- Pick first element of Q
- Add path extensions to front of Q

Breadth-first

- Pick first element of Q
- Add path extensions to end of Q

Best-first

- Pick "best" (measured by heuristic value of state) element of Q
- Add path extensions anywhere in Q (it may be more efficient to keep the Q ordered in some way so as to make it easier to find the "best" element)

19 - Spring 02 - 18

Slide 2.2.18

Best-first (also known as "greedy") search is a heuristic (informed) search that uses the value of a heuristic function defined on the states to guide the search. This will not guarantee finding a "best" path, for example, the shortest path to a goal. The heuristic is used in the hope that it will steer us to a quick completion of the search or to a relatively good goal state.

Best-first search can be implemented as follows: pick the "best" path (as measured by heuristic value of the node's state) from all of Q and add the extensions somewhere on Q. So, at any step, we are always examining the pending node with the best heuristic value.

Note that, in the worst case, this search will examine all the same paths that depth or breadth first would examine, but the order of examination may be different and therefore the resulting path will generally be different. Best-first has a kind of breadth-first flavor and we expect that Q will tend to grow more than in depth-first search.

Implementation Issues: Finding the best node

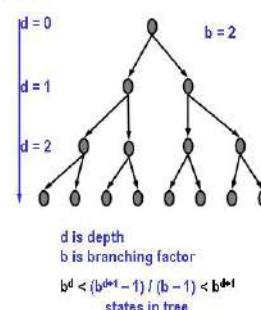
- There are many possible approaches to finding the best node in Q.
 - Scanning Q to find lowest value
 - Sorting Q and picking the first element
 - Keeping the Q sorted by doing "sorted" insertions
 - Keeping Q as a priority queue
- Which of these is best will depend among other things on how many children nodes have on average. We will look at this in more detail later.

19 - Spring 02 - 19

**Worst Case Running Time**

Max Time ~ Max #Visited

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.



19 - Spring 02 - 20

Slide 2.2.20

Let's think a bit about the worst case running time of the searches that we have been discussing. The actual running time, of course, will depend on details of the computer and of the software implementation. But, we can roughly compare the various algorithms by thinking of the number of nodes added to Q. The running time should be roughly proportional to this number.

In AI we usually think of a "typical" search space as being a tree with uniform branching factor b and depth d. The depth parameter may represent the number of steps in a plan of action or the number of moves in a game. The branching factor reflects the number of different choices that we have at each step. It is easy to see that the number of states in such a tree grows exponentially with the depth.

Slide 2.2.21

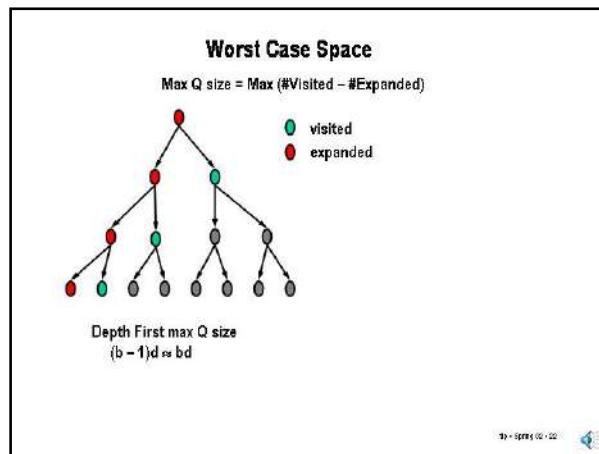
In a tree-structured search space, the nodes added to the search Q will simply correspond to the visited states. In the worst case, when the states are arranged in the worst possible way, all the search methods may end up having to visit or expand all of the states (up to some depth). In practice, we should be able to avoid this worst case but in many cases one comes pretty close to this worst case.

Worst Case Running Time

Max Time \approx Max #Visited

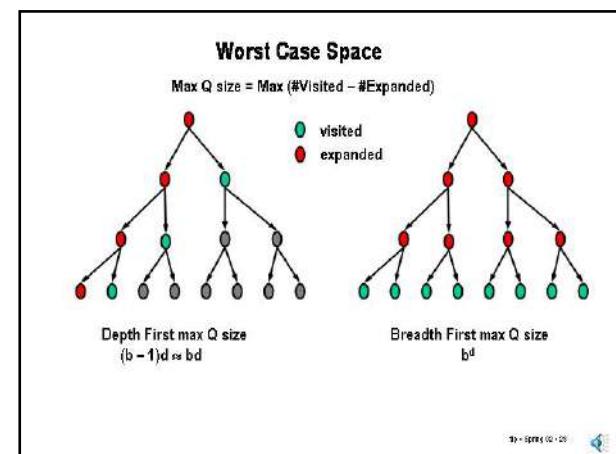
- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited list, may have to visit each state at least once, in the worst case.
- So, all searches will have worst case running times that are at least proportional to the total number of states and therefore exponential in the "depth" parameter.

19 - Spr 04 (2) 21

**Slide 2.2.22**

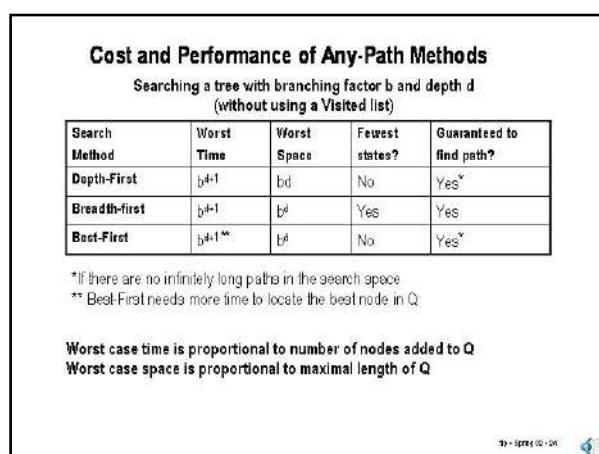
In addition to thinking about running time, we should also think about the memory space required for searches. The dominant factor in the space requirements for these searches is the maximum size of the search Q. The size of the search Q in a tree-structured search space is simply the number of visited states minus the number of expanded states.

For a depth-first search, we can see that Q holds the unexpanded "siblings" of the nodes along the path that we are currently considering. In a tree, the path length cannot be greater than d and the number of unexpanded siblings cannot be greater than b-1, so this tells us that the length of Q is always less than b*d, that is, the space requirements are linear in d.

**Slide 2.2.23**

The situation for breadth-first search is much different than that for depth-first search. Here the worst case happens after we've visited all the nodes at depth d-1. At that point, all the nodes at depth d have been visited and none expanded. So, the Q has size b^d, that is, a size exponential in d.

Note that, in the worst case, best-first behaves as breadth-first and has the same space requirements.

**Slide 2.2.24**

This table summarizes the key cost and performance properties of the different any-path search methods. We are assuming that our state space is a tree and so we cannot revisit states and a Visited list is useless.

Recall that this analysis is done for searching a tree with uniform branching factor b and depth d. Therefore, the size of this search space grows exponentially with the depth. So, it should not be surprising that methods that guarantee finding a path will require exponential time in this situation. These estimates are not intended to be tight and precise; instead they are intended to convey a feeling for the tradeoffs.

Note that we could have phrased these results in terms of V, the number of vertices (nodes) in the tree, and then everything would have worst case behavior that is linear in V. We phrase it the way we do because in many applications, the number of nodes depends in an exponential way on some depth parameter, for example, the length of an action plan, and thinking of the cost as linear in the number of nodes is misleading. However, in the algorithms literature, many of these algorithms are described as requiring time linear in the number of nodes.

There are two points of interest in this table. One is the fact that depth-first search requires much less space than the other searches. This is important, since space tends to be the limiting factor in large problems (more on this later). The other is that the time cost of best-first search is higher than that of the others. This is due to the cost of finding the best node in Q, not just the first one. We will also look at this in more detail later.

Slide 2.2.25

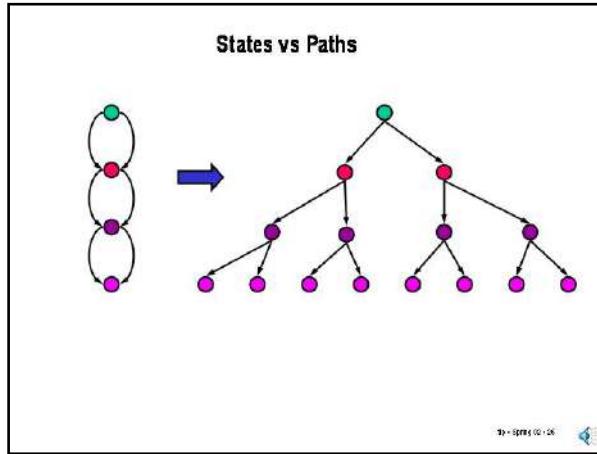
Remember that we are assuming in this slide that we are searching a tree, so states cannot be visited more than once - so the Visited list is completely superfluous when searching trees. However, if we were to use a Visited list (even implemented as a constant-time access hash table), the only thing that seems to change in this table is that the worst-case space requirements for all the searches go up (and way up for depth-first search). That does not seem to be very useful! Why would we ever use a Visited list?

Cost and Performance of Any-Path Methods				
Searching a tree with branching factor b and depth d (using a Visited list)				
Search Method	Worst Time	Worst Space	Fewest states?	Guaranteed to find path?
Depth-First	b^{d-1}	b^d b^{d-1}	No	Yes*
Breadth-first	b^{d-1}	b^d b^{d-1}	Yes	Yes
Best-First	b^{d-1}^∞	b^d b^{d-1}	No	Yes*

*If there are no infinitely long paths in the search space
** Best-First needs more time to locate the best node in Q

Worst case time is proportional to number of nodes added to Q
Worst case space is proportional to maximal length of Q (and Visited list)

19 / Spr 02 / 26



Slide 2.2.26

As we mentioned earlier, the key observation is that with a Visited list, our worst-case time performance is limited by the number of **states** in the search space (since you visit each state at most once) rather than the number of **paths** through the nodes in the space, which may be exponentially larger than the number of states, as this classic example shows. Note that none of the paths in the tree have a loop in them, that is, no path visits a state more than once. The Visited list is a way of spending space to limit this time penalty. However, it may not be appropriate for very large search spaces where the space requirements would be prohibitive.

Slide 2.2.27

So far, we have been treating time and space in parallel for our algorithms. It is tempting to focus on time as the dominant cost of searching and, for real-time applications, it is. However, for large off-line applications, space may be the limiting factor.

If you do a back of the envelope calculation on the amount of space required to store a tree with branching factor 8 and depth 10, you get a very large number. Many real applications may want to explore bigger spaces.

Space (the final frontier)	
• In large search problems, memory is often the limiting factor.	
• Imagine searching a tree with branching factor 8 and depth 10. Assume a node requires just 8 bytes of storage. Then, breadth-first search might require up to	$(2^8)^{10} \times 2^3 = 2^{33}$ bytes = 8,000 Mbytes = 8Gbytes

19 / Spr 02 / 27

Space
(the final frontier)

- In large search problems, memory is often the limiting factor.
- Imagine searching a tree with branching factor 8 and depth 10. Assume a node requires just 8 bytes of storage. Then, breadth-first search might require up to $(2^8)^{10} \times 2^3 = 2^{31}$ bytes = 8,000 Mbytes = 8Gbytes
- One strategy is to trade time for memory. For example, we can emulate breadth-first search by repeated applications of depth-first search, each up to a preset depth limit. This is called **progressive deepening search (PDS)**:
 1. C=1
 2. Do DFS to max depth C. If path found, return it.
 3. Otherwise, increment C and go to 2.

19 / SPRING 02 / 20

Slide 2.2.28

One strategy for enabling such open-ended searches, which may run for a very long time, is Progressive Deepening Search (aka Iterative Deepening Search). The basic idea is to simulate searches with a breadth-like component by a succession of depth-limited depth-first searches. Since depth-first has negligible storage requirements, this is a clean tradeoff of time for space.

Interestingly, PDS is more than just a performance tradeoff. It actually represents a merger of two algorithms that combines the best of both. Let's look at that a little more carefully.

Slide 2.2.29

Depth-first search has one strong point - its limited space requirements, which are linear in the depth of the search tree. Aside from that there's not much that can be said for it. In particular, it is susceptible to "going off the deep-end", that is, chasing very deep (possibly infinitely deep) paths. Because of this it does not guarantee, as breadth-first, does to find the shallowest goal states - those requiring the fewest actions to reach.

**Progressive Deepening Search
Best of Both Worlds**

- Depth-First Search (DFS) has small space requirements (linear in depth), but has major problems:
 - DFS can run forever in search spaces with infinite length paths
 - DFS does not guarantee finding shallowest goal

**Progressive Deepening Search
Best of Both Worlds**

- Depth-First Search (DFS) has small space requirements (linear in depth), but has major problems:
 - DFS can run forever in search spaces with infinite length paths
 - DFS does not guarantee of finding shallowest goal
- Breadth-First Search (BFS) guarantees finding shallowest goal, even in the presence of infinite paths, but is has one great problem:
 - BFS requires a great deal of space (exponential in depth)

19 / SPRING 02 / 30

Slide 2.2.30

Breadth-first search on the other hand, does guarantee finding the shallowest goal, but at the expense of space requirements that are exponential in the depth of the search tree.

Slide 2.2.31

Progressive-deepening search, on the other other hand, has both limited space requirements of DFS and the strong optimality guarantee of BFS. Great! No?

**Progressive Deepening Search
Best of Both Worlds**

- Depth-First Search (DFS) has small space requirements (linear in depth), but has major problems:
 - DFS can run forever in search spaces with infinite length paths
 - DFS does not guarantee of finding shallowest goal
- Breadth-First Search (BFS) guarantees finding shallowest goal, even in the presence of infinite paths, but is has one great problem:
 - BFS requires a great deal of space (exponential in depth)
- Progressive Deepening Search (PDS) has the advantages of DFS and BFS:
 - PDS has small space requirements (linear in depth)
 - PDS guarantees finding shallowest goal

19 / SPRING 02 / 31

Progressive Deepening Search

- Isn't Progressive Deepening (PDS) too expensive?

Slide 2.2.32

At first sight, most people find PDS horrifying. Isn't progressive deepening really wasteful? It looks at the same nodes over and over again...

19 / SPRING 02 / 98



Slide 2.2.33

In small graphs, yes it is wasteful. But, if we really are faced with an exponentially growing space (in the depth), then it turns out that the work at the deepest level dominates the total cost.

Progressive Deepening Search

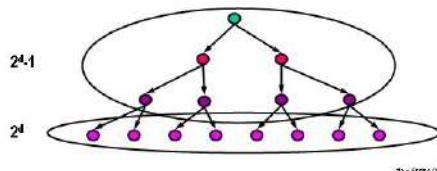
- Isn't Progressive Deepening (PDS) too expensive?
- In exponential trees, time is dominated by deepest search.

19 / SPRING 02 / 98



Progressive Deepening Search

- Isn't Progressive Deepening (PDS) too expensive?
- In exponential trees, time is dominated by deepest search.
- For example, if branching factor is 2, then the number of nodes at depth d is 2^d while the total number of nodes in all previous levels is 2^{d-1} , so the difference between looking at whole tree versus only the deepest level is at worst a factor of 2 in performance.



Slide 2.2.34

It is easy to see this for binary trees, where the number of nodes at level d is about equal to the number of nodes in the rest of the tree. The worst-case time for BFS at level d is proportional to the number of nodes at level d , while the worst case time for PDS at that level is proportional to the number of nodes in the whole tree which is almost exactly twice those at the deepest level. So, in the worst case, PDS (for binary trees) does no more than twice as much work as BFS, while using much less space.

This is a worst case analysis, it turns out that if we try to look at the expected case, the situation is even better.

Slide 2.2.35

One can derive an estimate of the ratio of the work done by progressive deepening to that done by a single depth-first search: $(b+1)/(b-1)$. This estimate is for the average work (averaging over all possible searches in the tree). As you can see from the table, this ratio approaches one as the branching factor increases (and the resulting exponential explosion gets worse).

Progressive Deepening Search

- Compare the ratio of average time spent on PDS with average time spent on a single DFS with the full depth tree:

$$(\text{Avg time for PDS})/(\text{Avg time for DFS}) \approx (b+1)/(b-1)$$

b	ratio
2	3
3	2
5	1.5
25	1.08
100	1.02

19 / SPRING 02 / 98



Progressive Deepening Search

- Compare the ratio of average time spent on PDS with average time spent on a single DFS with the full depth tree:
 $(\text{Avg time for PDS}) / (\text{Avg time for DFS}) \approx (b+1) / (b-1)$
- Progressive deepening is an effective strategy for difficult searches.

b	ratio
2	3
3	2
5	1.6
25	1.08
100	1.02

10 - Spring 02 - 96

Slide 2.2.36

For many difficult searches, progressive deepening is in fact the only way to go. There are also progressive deepening versions of the optimal searches that we will see later, but that's beyond our scope.

6.034 Notes: Section 2.3

Slide 2.3.1

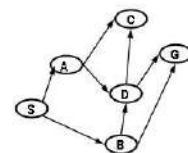
We will now step through the any-path search methods looking at their implementation in terms of the simple algorithm. We start with depth-first search using a Visited list.

The table in the center shows the contents of Q and of the Visited list at each time through the loop of the search algorithm. The nodes in Q are indicated by reversed paths, blue is used to indicate newly added nodes (paths). On the right is the graph we are searching and we will label the state of the node that is being extended at each step.

Depth-First

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1		
2		
3		
4		
5		



Added paths in blue

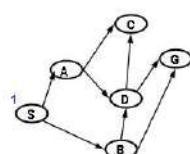
We show the paths in reversed order; the node's state is the first entry.

10 - Spring 02 - 1

Depth-First

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2		
3		
4		
5		



Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

10 - Spring 02 - 2

Slide 2.3.2

The first step is to initialize Q with a single node corresponding to the start state (S in this case) and the Visited list with the start state.

Slide 2.3.3

We pick the first element of Q, which is that initial node, remove it from Q, extend its path to its descendant states (if they have not been Visited) and add the resulting nodes to the front of Q. We also add the states corresponding to these new nodes to the Visited list. So, we get the situation on line 2.

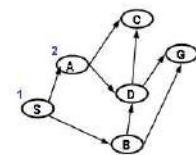
Note that the descendant nodes could have been added to Q in the other order. This would be absolutely valid. We will typically add nodes to Q in such a way that we end up visiting states in alphabetical order, when no other order is specified by the algorithm. This is purely an arbitrary decision.

We then pick the first node on Q, whose state is A, and repeat the process, extending to paths that end at C and D and placing them at the front of Q.

Depth-First

Pick first element of Q; Add path extensions to front of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A, B, S
3	
4	
5	



Added paths in blue

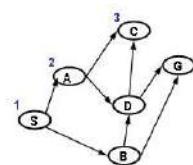
We show the paths in reversed order; the node's state is the first entry.

10 - Spines 02 - 6

Depth-First

Pick first element of Q; Add path extensions to front of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A, B, S
3 (C A S) (D A S) (B S)	C,D,B,A,S
4	
5	



Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

10 - Spines 02 - 6

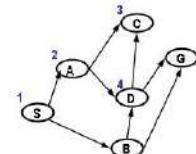
Slide 2.3.4

We pick the first node, whose state is C, and note that there are no descendants of C and so no new nodes to add.

Depth-First

Pick first element of Q; Add path extensions to front of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A, B, S
3 (C A S) (D A S) (B S)	C,D,B,A,S
4 (D A S) (B S)	C,D,B,A,S
5	



Added paths in blue

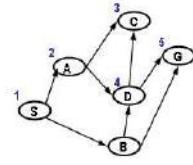
We show the paths in reversed order; the node's state is the first entry.

10 - Spines 02 - 6

Depth-First

Pick first element of Q; Add path extensions to front of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A, B, S
3 (C A S) (D A S) (B S)	C,D,B,A,S
4 (D A S) (B S)	C,D,B,A,S
5 (G D A S) (B S)	G,C,D,B,A,S



Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

10 - Spines 02 - 6

Slide 2.3.6

We pick the first node of Q, whose state is G, the intended goal state, so we stop and return the path.

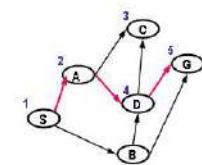
Slide 2.3.7

The final path returned goes from S to A, then to D and then to G.

Depth-First

Pick first element of Q; Add path extensions to front of Q

Q	Visited
1 (S)	S
2 (A S) (B S)	A, B, S
3 (C A S) (D A S) (B S)	C,D,B,A,S
4 (D A S) (E S)	C,D,B,A,S
5 (G D A S) (B S)	G,C,D,B,A,S



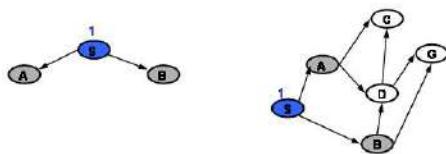
Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

10 - Sprout (21-7)

Depth-First

Another (easier?) way to see it



Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Slide 2.3.8

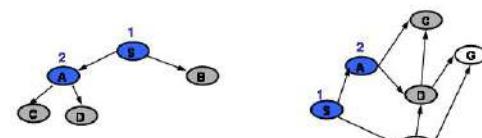
Tracing out the content of Q can get a little monotonous, although it allows one to trace the performance of the algorithms in detail. Another way to visualize simple searches is to draw out the search tree, as shown here, showing the result of the first expansion in the example we have been looking at.

Slide 2.3.9

In this view, we introduce a left to right bias in deciding which nodes to expand - this is purely arbitrary. It corresponds exactly to the arbitrary decision of which nodes to add to Q first. Giving this bias, we decide to expand the node whose state is A, which ends up visiting C and D.

Depth-First

Another (easier?) way to see it



Numbers indicate order pulled off of Q (expanded)

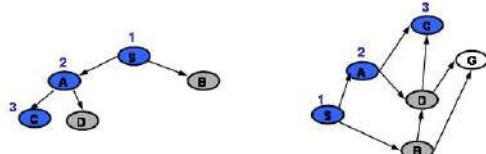
Dark blue fill = Visited & Expanded

Light gray fill = Visited

10 - Sprout (21-7)

Depth-First

Another (easier?) way to see it



Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

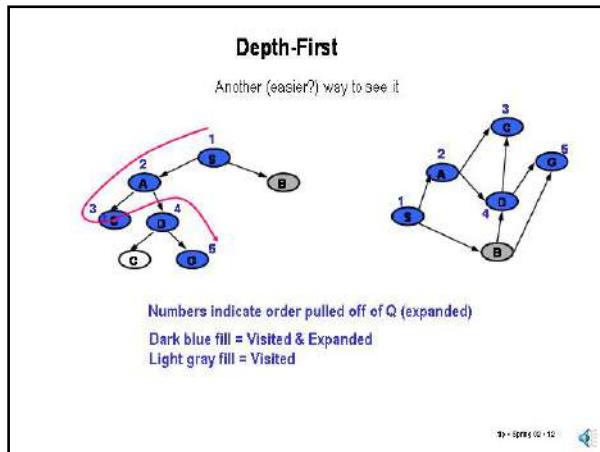
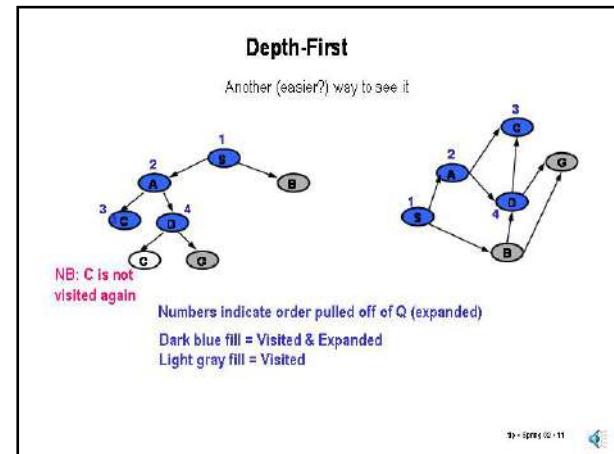
Slide 2.3.10

We now expand the node corresponding to C, which has no descendants, so we cannot continue to go deeper. At this point, one talks about having to **back up** or **backtrack** to the parent node and expanding any unexpanded descendant nodes of the parent. If there were none at that level, we would continue to keep backing up to its parent and so on until an unexpanded node is found. We declare failure if we cannot find any remaining unexpanded nodes. In this case, we find an unexpanded descendant of A, namely D.

10 - Sprout (21-10)

Slide 2.3.11

So, we expand D. Note that states C and G are both reachable from D. However, we have already visited C, so we do not add a node corresponding to that path. We add only the new node corresponding to the path to G.

**Slide 2.3.12**

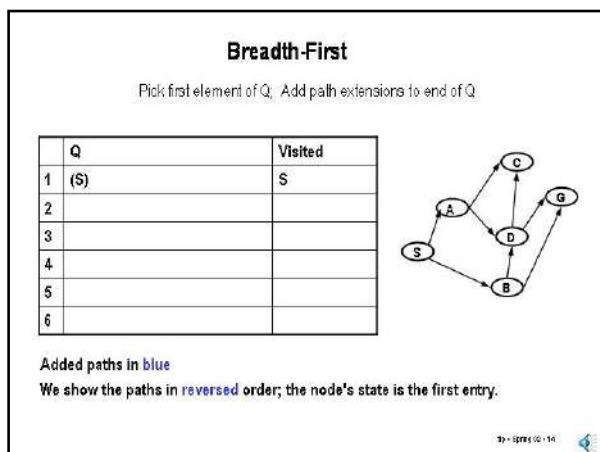
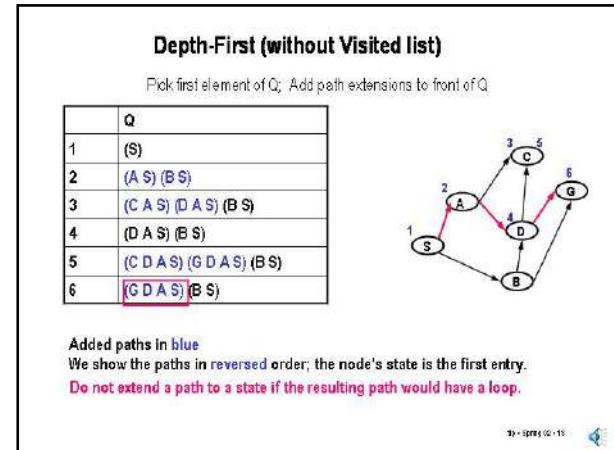
We now expand G and stop.

This view of depth-first search is the more common one (rather than tracing Q). In fact, it is in this view that one can visualize why it is called depth-first search. The red arrow shows the sequence of expansions during the search and you can see that it is always going as deep in the search tree as possible. Also, we can understand another widely used name for depth-first search, namely **backtracking** search. However, you should convince yourself that this view is just a different way to visualize the behavior of the Q-based algorithm.

Slide 2.3.13

We can repeat the depth-first process without the Visited list and, as expected, one sees the second path to C added to Q, which was blocked by the use of the Visited list. I'll leave it as an exercise to go through the steps in detail.

Note that in the absence of a Visited list, we still require that we do not form any paths with loops, so if we have visited a state along a particular path, we do not re-visit that state again in any extensions of the path.

**Slide 2.3.14**

Let's look now at breadth-first search. The difference from depth-first search is that new paths are added to the back of Q. We start as with depth-first with the initial node corresponding to S.

Slide 2.3.15

We pick it and add paths to A and B, as before.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3		
4		
5		
6		

Added paths in blue
We show the paths in reversed order; the node's state is the first entry.

19 - Sprouts (2/15)

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4		
5		
6		

Added paths in blue
We show the paths in reversed order; the node's state is the first entry.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		

Added paths in blue
We show the paths in reversed order; the node's state is the first entry.

19 - Sprouts (2/17)

Slide 2.3.17

Now, the first node in Q is the path to B so we pick that and consider its extensions to D and G. Since D is already Visited, we ignore that and add the path to G to the end of Q.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		

Added paths in blue
We show the paths in reversed order; the node's state is the first entry.
* We could have stopped here, when the first path to the goal was generated.

Slide 2.3.16

We pick the first node, whose state is A, and extend the path to C and D and add them to Q (at the back) and here we see the difference from depth-first.

Slide 2.3.18

At this point, having generated a path to G, we would be justified in stopping. But, as we mentioned earlier, we proceed until the path to the goal becomes the first path in Q.

Slide 2.3.19

We now pull out the node corresponding to C from Q but it does not generate any extensions since C has no descendants.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		

Added paths in blue
We show the paths in **reversed** order; the node's state is the first entry.
* We could have stopped here, when the first path to the goal was generated.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		

Added paths in blue
We show the paths in **reversed** order; the node's state is the first entry.
* We could have stopped here, when the first path to the goal was generated.

Slide 2.3.20

So we pull out the path to D. Its potential extensions are to previously visited states and so we get nothing added to Q.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

Added paths in blue
We show the paths in **reversed** order; the node's state is the first entry.
* We could have stopped here, when the first path to the goal was generated.

Slide 2.3.21

Finally, we get the path to G and we stop.

Breadth-First

Pick first element of Q. Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

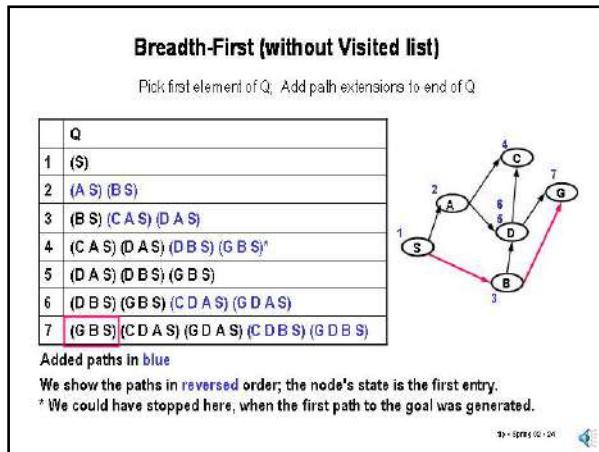
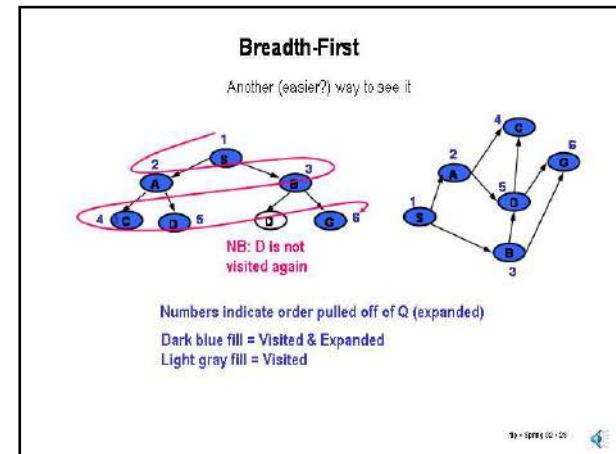
Added paths in blue
We show the paths in **reversed** order; the node's state is the first entry.
* We could have stopped here, when the first path to the goal was generated.

Slide 2.3.22

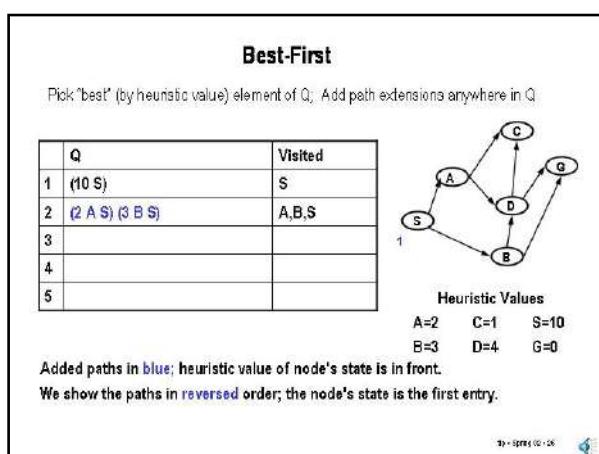
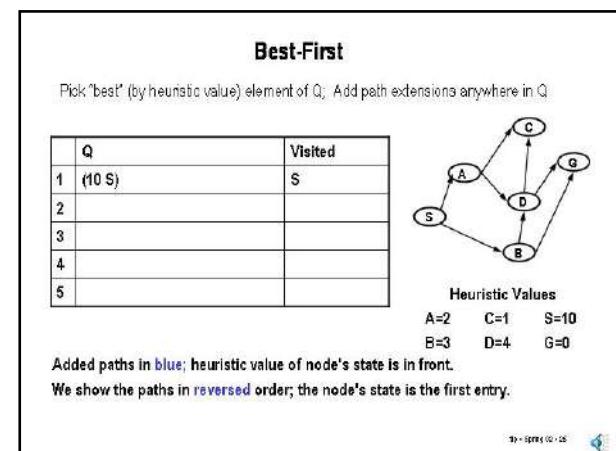
Note that we found a path with fewer states than we did with depth-first search, from S to B to G. In general, breadth-first search guarantees finding a path to the goal with the minimum number of states.

Slide 2.3.23

Here we see the behavior of breadth-first search in the search-tree view. In this view, you can see why it is called breadth-first -- it is exploring all the nodes at a single depth level of the search tree before proceeding to the next depth level.

**Slide 2.3.24**

We can repeat the breadth-first process without the Visited list and, as expected, one sees multiple paths to C, D and G are added to Q, which were blocked by the Visited test earlier. I'll leave it as an exercise to go through the steps in detail.

**Slide 2.3.26**

We pick the first node and extend to A and B.

Slide 2.3.27

We pick the node corresponding to A, since it has the best value (= 2) and extend to C and D.

Best-First

Pick "best" (by heuristic value) element of Q. Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4		
5		

Heuristic Values
 A=2 C=1 S=10
 B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.
 We show the paths in reversed order; the node's state is the first entry.

Best-First

Pick "best" (by heuristic value) element of Q. Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5		

Heuristic Values
 A=2 C=1 S=10
 B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.
 We show the paths in reversed order; the node's state is the first entry.

Slide 2.3.28

The node corresponding to C has the lowest value so we pick that one. That goes nowhere.

Best-First

Pick "best" (by heuristic value) element of Q. Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values
 A=2 C=1 S=10
 B=3 D=4 G=0

Added paths in blue; heuristic value of node's state is in front.
 We show the paths in reversed order; the node's state is the first entry.

Slide 2.3.29

Then, we pick the node corresponding to B which has lower value than the path to D and extend to G (not C because of previous Visit).

Best-First

Pick "best" (by heuristic value) element of Q. Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values
 A=2 C=1 S=10
 B=3 D=4 G=0

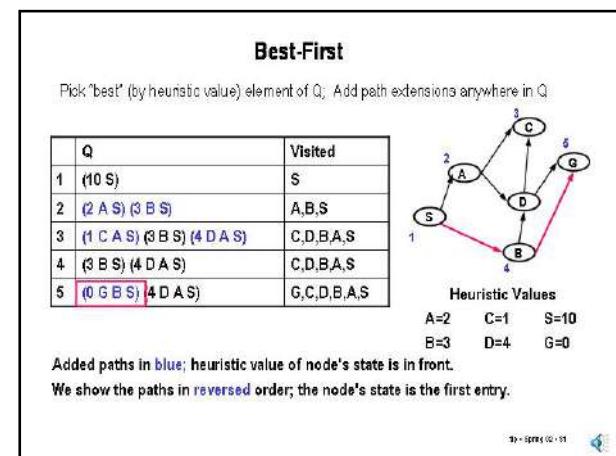
Added paths in blue; heuristic value of node's state is in front.
 We show the paths in reversed order; the node's state is the first entry.

Slide 2.3.30

We pick the node corresponding to G and rejoice.

Slide 2.3.31

We found the path to the goal from S to B to G.



6.034 Notes: Section 2.4

Slide 2.4.1

So far, we have looked at three any-path algorithms, depth-first and breadth-first, which are uninformed, and best-first, which is heuristically guided.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a node, e.g. estimated distance to goal.

10 - Spring 02 - 1



Classes of Search

Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a node, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.

Slide 2.4.2

Now, we will look at the first algorithm that searches for optimal paths, as defined by a "path length" measure. This uniform cost algorithm is uninformed about the goal, that is, it does not use any heuristic guidance.

10 - Spring 02 - 2



Slide 2.4.3

This is the simple algorithm we have been using to illustrate the various searches. As before, we will see that the key issues are picking paths from Q and adding extended paths back in.

Simple Search Algorithm

A search node is a path from some state X to the start state, e.g., (X B A S).
The state of a search node is the most recent state of the path, e.g. X.
Let Q be a list of search nodes, e.g. [(X B A S) (C B A S) ...].
Let S be the start state.

1. Initialize Q with search node (S) as only entry; set Visited = (S)
2. If Q is empty, fail. Else, pick some partial path N from Q
3. If state(N) is a goal, return N (we've reached a goal)
4. (Otherwise) Remove N from Q
5. Find all the children of state(N) not in Visited and create all the one-step extensions of N to each descendant.
6. Add all the extended paths to Q; add children of state(N) to Visited
7. Go to step 2.

Critical decisions:

Step 2: picking N from Q.

Step 6: adding extensions of N to Q.

10 - Spring 02 - 5

**Simple Search Algorithm**

A search node is a path from some state X to the start state, e.g., (X B A S).
The state of a search node is the most recent state of the path, e.g. X.
Let Q be a list of search nodes, e.g. [(X B A S) (C B A S) ...].
Let S be the start state.

1. Initialize Q with search node (S) as only entry; set Visited = (S)
2. If Q is empty, fail. Else, pick some search node N from Q
3. If state(N) is a goal, return N (we've reached a goal)
4. (Otherwise) Remove N from Q
5. Find all the children of state(N) not in Visited and create all the one-step extensions of N to each descendant.
6. Add all the extended paths to Q; add children of state(N) to Visited
7. Go to step 2.

Critical decisions:

Step 2: picking N from Q.

Step 6: adding extensions of N to Q.

10 - Spring 02 - 5

Slide 2.4.4

We will continue to use the algorithm but (as we will see) the use of the Visited list conflicts with optimal searching, so we will leave it out for now and replace it with something else later.

Slide 2.4.5

Why can't we use a Visited list in connection with optimal searching? In the earlier searches, the use of the Visited list guaranteed that we would not do extra work by re-visiting or re-expanding states. It did not cause any failures then (except possibly of intuition).

Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.

10 - Spring 02 - 5

**Why not a Visited list?**

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with optimal searches can cause us to miss the best path.

Slide 2.4.6

But, using the Visited list can cause an optimal search to overlook the best path. A simple example will illustrate this.

10 - Spring 02 - 5

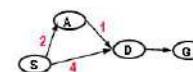


Slide 2.4.7

Clearly, the shortest path (as determined by sum of link costs) to G is (S A D G) and an optimal search had better find it.

Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with UC can cause us to miss the best path.

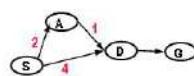


- The shortest path from S to G is (S A D G)

10 / 89 Slides (24 / 24)

Why not a Visited list?

- For the any-path algorithms, the Visited list would not cause us to fail to find a path when one existed, since the path to a state did not matter.
- However, the Visited list in connection with UC can cause us to miss the best path.



- The shortest path from S to G is (S A D G)
- But, on extending (S), A and D would be added to Visited list and so (S A) would not be extended to (S A D)

10 / 89 Slides (24 / 24)

Slide 2.4.8

However, on expanding S, A and D are Visited, which means that the extension from A to D would never be generated and we would miss the best path. So, we can't use a Visited list; nevertheless, we still have the problem of multiple paths to a state leading to wasted work. We will deal with that issue later, since it can get a bit complicated. So, first, we will focus on the basic operation of optimal searches.

Slide 2.4.9

The first, and most basic, algorithm for optimal searching is called uniform-cost search. Uniform-cost is almost identical in implementation to best-first search. That is, we always pick the best node on Q to expand. The only, but crucial, difference is that instead of assigning the node value based on the heuristic value of the node's state, we will assign the node value as the "path length" or "path cost", a measure obtained by adding the "length" or "cost" of the links making up the path.

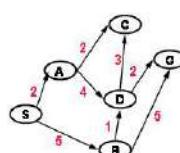
Implementing Optimal Search Strategies**Uniform Cost**

Pick best (measured by path length) element of Q
Add path extensions anywhere in Q

10 / 89 Slides (24 / 24)

Uniform Cost

- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path



10 / 89 Slides (24 / 24)

Slide 2.4.10

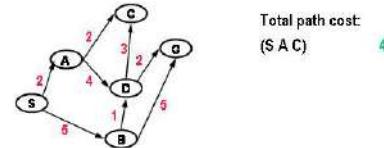
To reiterate, uniform-cost search uses the total length (or cost) of a path to decide which one to expand. Since we generally want the least-cost path, we will pick the node with the smallest path cost/length. By the way, we will often use the word "length" when talking about these types of searches, which makes intuitive sense when we talk about the pictures of graphs. However, we mean any cost measure (like length) that is positive and greater than 0 for the link between any two states.

Slide 2.4.11

The path length is the SUM of the length associated with the links in the path. For example, the path from S to A to C has total length 4, since it includes two links, each with edge 2.

**Uniform Cost**

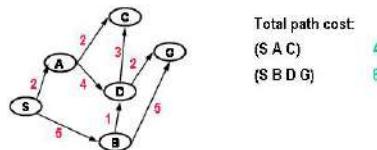
- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path



19 - Spring 02 - 11

**Uniform Cost**

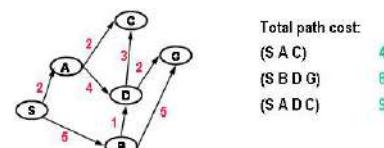
- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path



19 - Spring 02 - 12

Slide 2.4.12

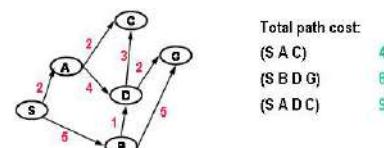
The path from S to B to D to G has length 8 since it includes links of length 5 (S-B), 1 (B-D) and 2 (D-G).



19 - Spring 02 - 13

**Uniform Cost**

- Like best-first except that it uses the "total length (cost)" of a path instead of a heuristic value for the state.
- Each link has a "length" or "cost" (which is always greater than 0)
- We want "shortest" or "least cost" path

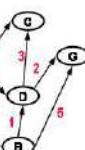


19 - Spring 02 - 14

**Uniform Cost**

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q
1 (0 S)



Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

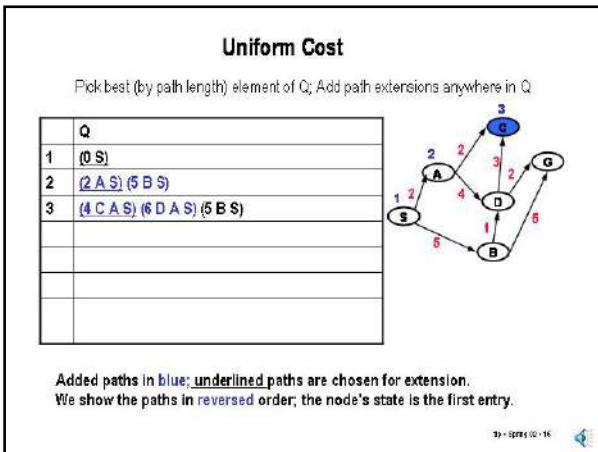
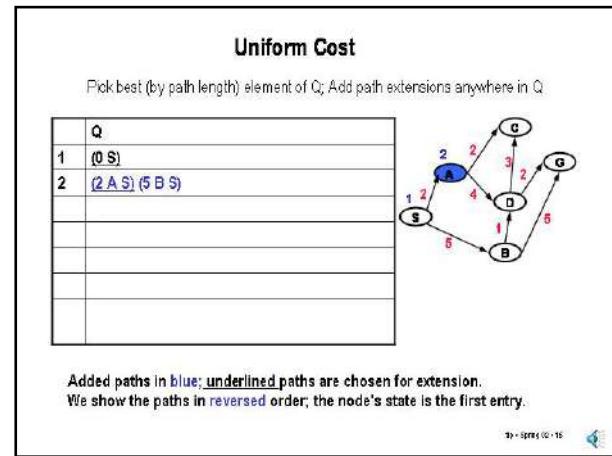
19 - Spring 02 - 15

Slide 2.4.14

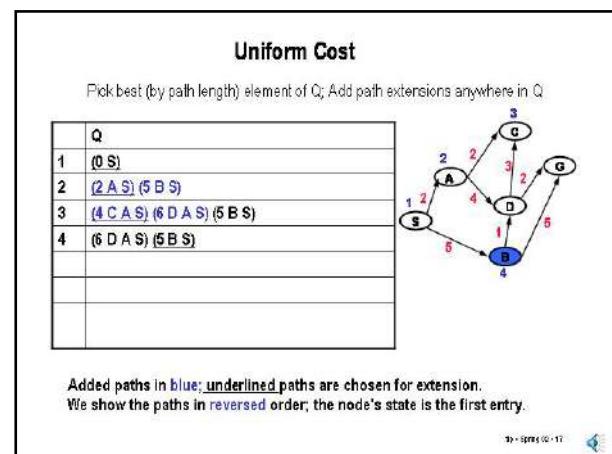
Given this, let's simulate the behavior of uniform-cost search on this simple directed graph. As usual we start with a single node containing just the start state S. This path has zero length. Of course, we choose this path for expansion.

Slide 2.4.15

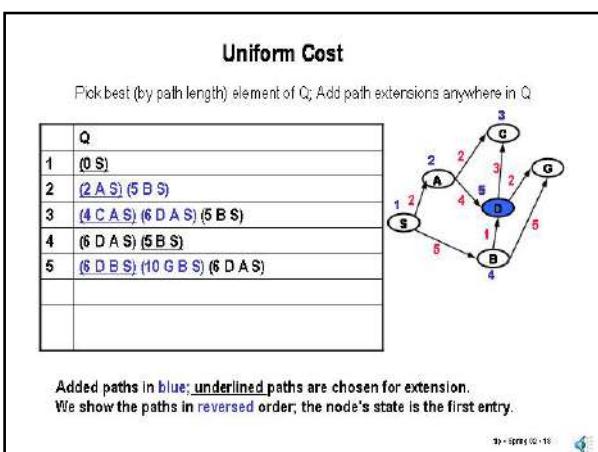
This generates two new entries on Q; the path to A has length 2 and the one to B has length 5. So, we pick the path to A to expand.

**Slide 2.4.16**

This generates two new entries on the queue. The new path to C is the shortest path on Q, so we pick it to expand.

**Slide 2.4.17**

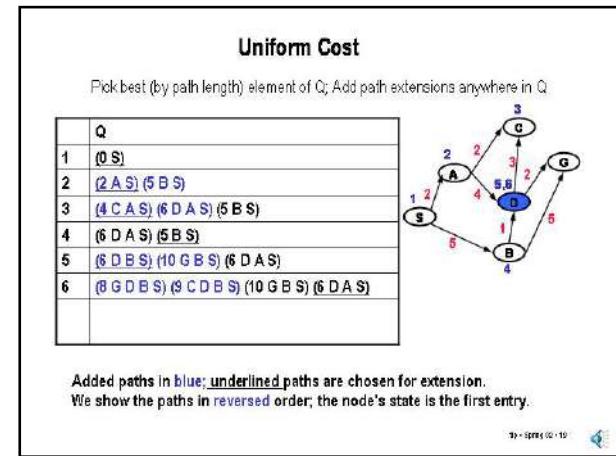
Since C has no descendants, we add no new paths to Q and we pick the best of the remaining paths, which is now the path to B.

**Slide 2.4.18**

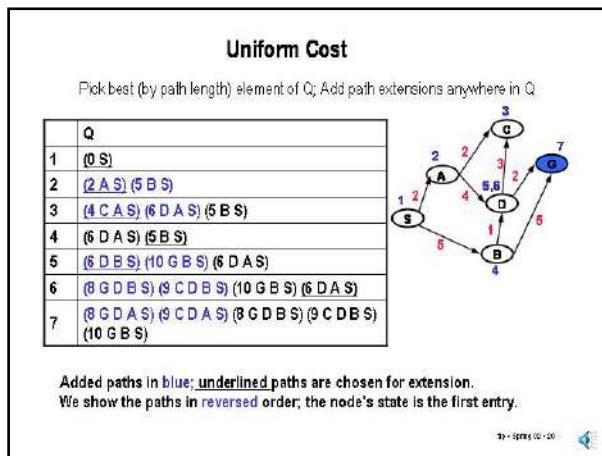
The path to B is extended to D and G and the path to D from B is tied with the path to D from A. We are using order in Q to settle ties and so we pick the path from B to expand. Note that at this point G has been visited but not expanded.

Slide 2.4.19

Expanding D adds paths to C and G. Now the earlier path to D from A is the best pending path and we choose it to expand.



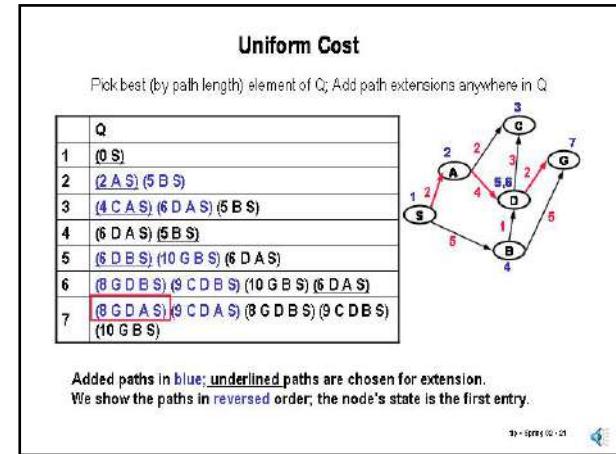
19 - Sprouts (21 / 21)



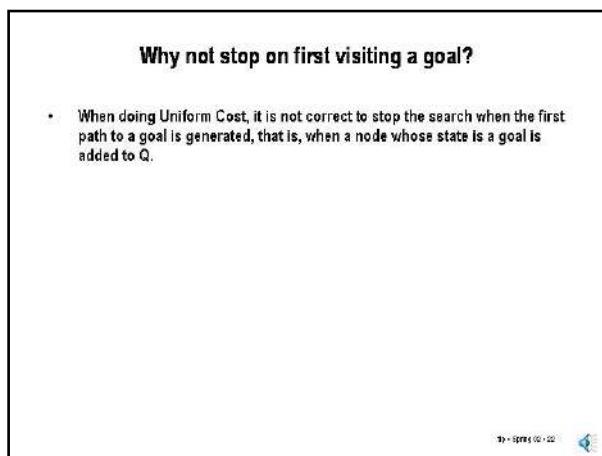
19 - Sprouts (20 / 21)

Slide 2.4.21

And we have found our shortest path (S A D G) whose length is 8.



19 - Sprouts (21 / 21)

**Slide 2.4.22**

Note that once again we are not stopping on first visiting (placing on Q) the goal. We stop when the goal gets expanded (pulled off Q).

Slide 2.4.23

In uniform-cost search, it is imperative that we only stop when G is expanded and not just when it is visited. Until a path is first expanded, we do not know for a fact that we have found the shortest path to the state.

Why not stop on first visiting a goal?

- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.

19 - Spry (2/25)

Why not stop on first visiting a goal?

- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.
- This contrasts with the non-optimal searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.

19 - Spry (2/25)

Slide 2.4.24

In the any-path searches we chose to do the same thing, but that choice was motivated at the time simply by consistency with what we HAVE to do now. In the earlier searches, we could have chosen to stop when visiting a goal state and everything would still work fine (actually better).

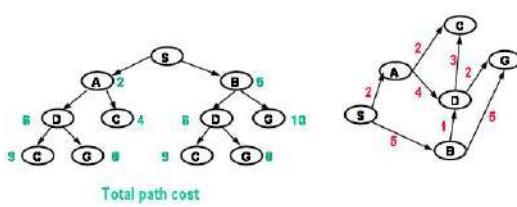
Why not stop on first visiting a goal?

- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.
- This contrasts with the Any Path searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.
- In the previous example, a path to G was generated at step 5, but it was a different, shorter, path at step 7 that we accepted.

19 - Spry (2/25)

Uniform Cost

Another (easier?) way to see it



UC enumerates paths in order of total path cost!

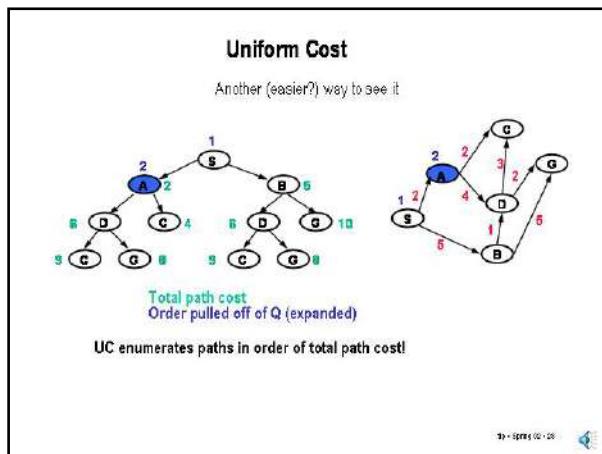
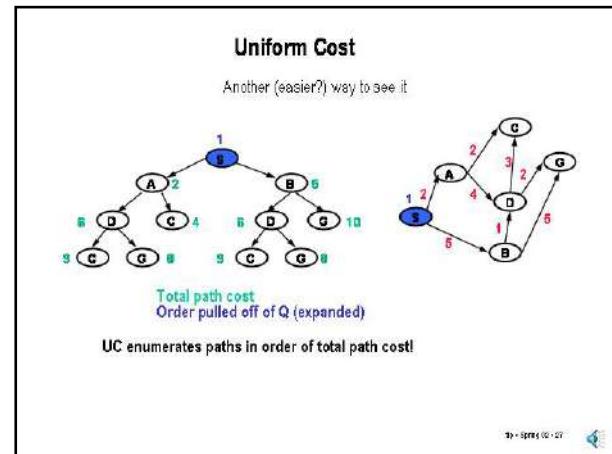
19 - Spry (2/25)

Slide 2.4.26

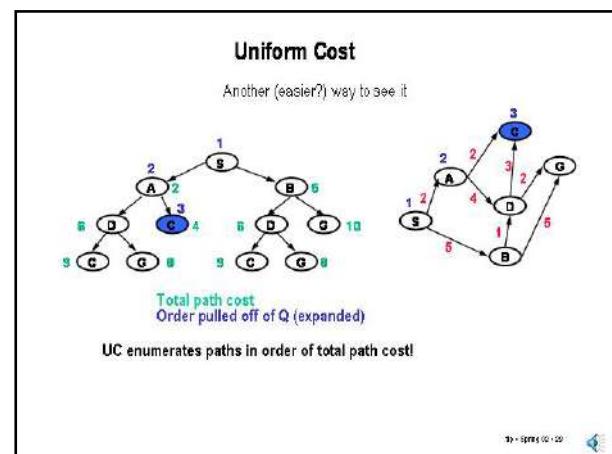
It is very important to drive home the fact that what uniform-cost search is doing (if we focus on the sequence of expanded paths) is enumerating the paths in the search tree in order of their path cost. The green numbers next to the tree on the left are the total path cost of the path to that state. Since, in a tree, there is a unique path from the root to any node, we can simply label each node by the length of that path.

Slide 2.4.27

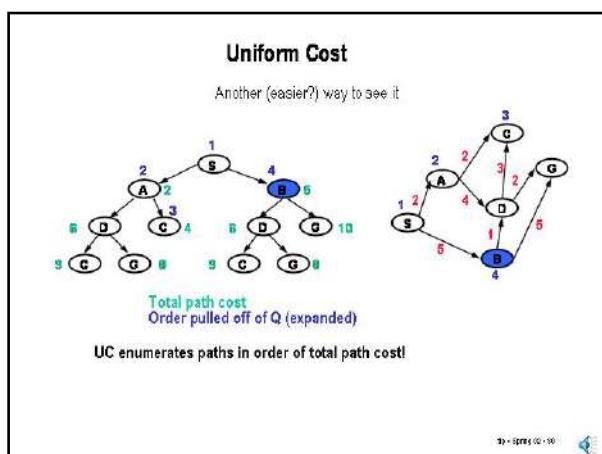
So, for example, the trivial path from S to S is the shortest path.

**Slide 2.4.28**

Then the path from S to A, with length 2, is the next shortest path.

**Slide 2.4.29**

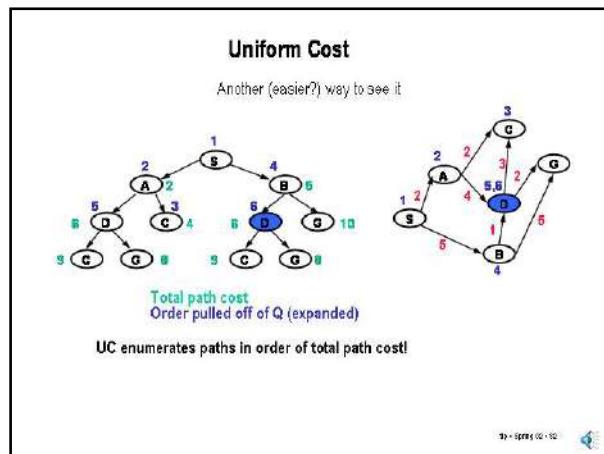
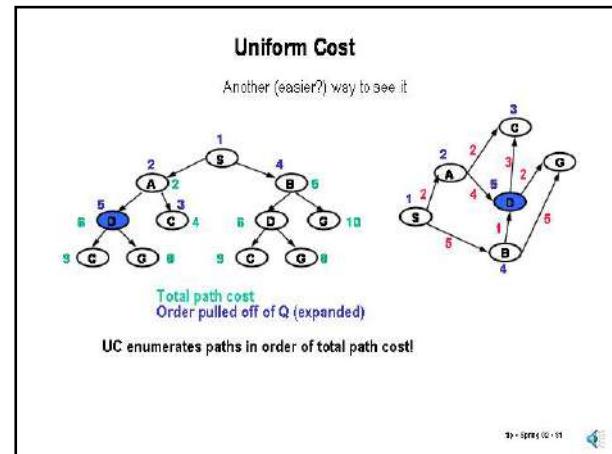
Then the path from S to A to C, with length 4, is the next shortest path.

**Slide 2.4.30**

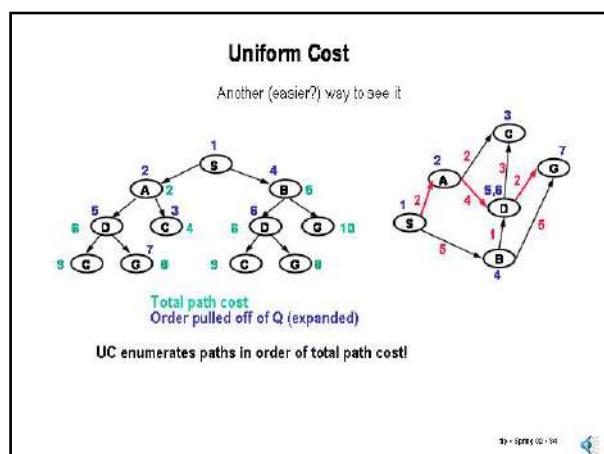
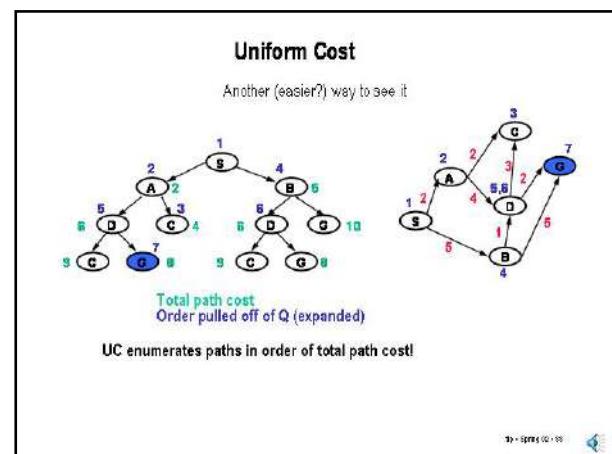
Then comes the path from S to B, with length 5.

Slide 2.4.31

Followed by the path from S to A to D, with length 6.

**Slide 2.4.32**

And the path from S to B to D, also with length 6.

**Slide 2.4.34**

This gives us the path we found. Note that the sequence of expansion corresponds precisely to path-length order, so it is not surprising we find the shortest path.

6.034 Notes: Section 2.5

Slide 2.5.1

Now, we will turn our attention to what is probably the most popular search algorithm in AI, the A* algorithm. A* is an informed, optimal search algorithm. We will spend quite a bit of time going over A*; we will start by contrasting it with uniform-cost search.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a node, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	A*	Uses path "length" measure and heuristic. Finds "shortest" path.

10 - SPRING 02 - 1



Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.

Slide 2.5.2

Uniform-cost search as described so far is concerned only with expanding short paths; it pays no particular attention to the goal (since it has no way of knowing where it is). UC is really an algorithm for finding the shortest paths to all states in a graph rather than being focused in reaching a particular goal.

Slide 2.5.3

We can bias UC to find the shortest path to the goal that we are interested in by using a heuristic estimate of remaining distance to the goal. This, of course, cannot be the exact path distance (if we knew that we would not need much of a search); instead, it is a stand-in for the actual distance that can give us some guidance.

Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function $h(N)$, which is an estimate (\hat{h}) of the distance from a state to the goal.

10 - SPRING 02 - 8



Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function $h(N)$, which is an estimate (h) of the distance from a state to the goal.
- Instead of enumerating paths in order of just length (g), enumerate paths in terms of $f = \text{estimated total path length} = g + h$.

Slide 2.5.4

What we can do is to enumerate the paths by order of the SUM of the actual path length and the estimate of the remaining distance. Think of this as our best estimate of the TOTAL distance to the goal. This makes more sense if we want to generate a path to the goal preferentially to short paths away from the goal.

Slide 2.5.5

We call an estimate that always underestimates the remaining distance from any node an admissible (heuristic) estimate.

Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function $h(N)$, which is an estimate (h) of the distance from a state to the goal.
- Instead of enumerating paths in order of just length (g), enumerate paths in terms of $f = \text{estimated total path length} = g + h$.
- An estimate that always underestimates the real path length to the goal is called admissible. For example, an estimate of 0 is admissible (but useless). Straight line distance is admissible estimate for path length in Euclidean space.

Goal Direction

- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function $h(N)$, which is an estimate (h) of the distance from a state to the goal.
- Instead of enumerating paths in order of just length (g), enumerate paths in terms of $f = \text{estimated total path length} = g + h$.
- An estimate that always underestimates the real path length to the goal is called admissible. For example, an estimate of 0 is admissible (but useless). Straight line distance is admissible estimate for path length in Euclidean space.
- Use of an admissible estimate guarantees that UC will still find the shortest path.

Slide 2.5.6

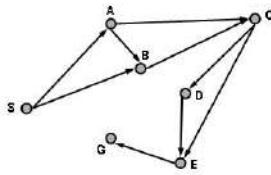
In order to preserve the guarantee that we will find the shortest path by expanding the partial paths based on the estimated total path length to the goal (like in UC without an expanded list), we must ensure that our heuristic estimate is admissible. Note that straight-line distance is always an underestimate of path-length in Euclidean space. Of course, by our constraint on distances, the constant function 0 is always admissible (but useless).

Slide 2.5.7

UC using an admissible heuristic is known as A* (A star). It is a very popular search method in AI.

Goal Direction

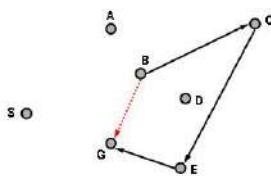
- UC is really trying to identify the shortest path to every state in the graph in order. It has no particular bias to finding a path to a goal early in the search.
- We can introduce such a bias by means of heuristic function $h(N)$, which is an estimate (h) of the distance from a state n to a goal.
- Instead of enumerating paths in order of just length (g), enumerate paths in terms of $f = \text{estimated total path length} = g + h$.
- An estimate that always underestimates the real path length to the goal is called admissible. For example, an estimate of 0 is admissible (but useless). Straight line distance is admissible estimate for path length in Euclidean space.
- Use of an admissible estimate guarantees that UC will still find the shortest path.
- UC with an admissible estimate is known as A* (pronounced "A star") search.

Straight-Line Estimate**Slide 2.5.8**

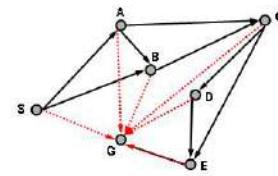
Let's look at a quick example of the straight-line distance underestimate for path length in a graph. Consider the following simple graph, which we are assuming is embedded in Euclidean space, that is, think of the states as city locations and the length of the links are proportional to the driving distance between the cities along the best roads.

Slide 2.5.9

Then, we can use the straight-line (airline) distances (shown in red) as an underestimate of the actual driving distance between any city and the goal. The best possible driving distance between two cities cannot be better than the straight-line distance. But, it can be much worse.

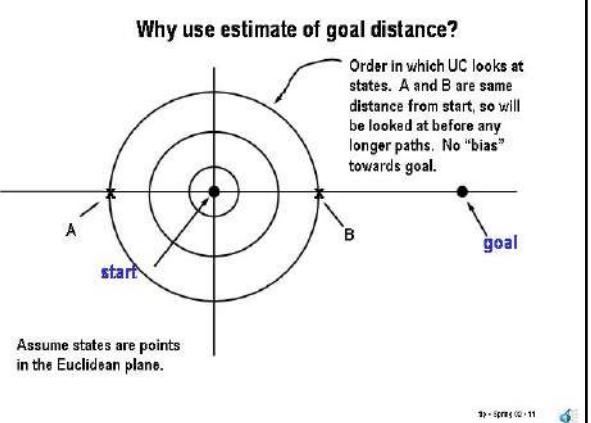
Straight-Line Estimate**Slide 2.5.10**

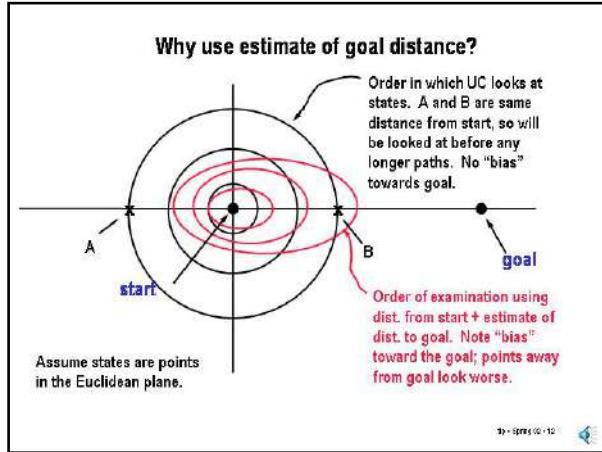
Here we see that the straight-line estimate between B and G is very bad. The actual driving distance is much longer than the straight-line underestimate. Imagine that B and G are on different sides of the Grand Canyon, for example.

Straight-Line Estimate**Slide 2.5.11**

It may help to understand why an underestimate of remaining distance may help reach the goal faster to visualize the behavior of UC in a simple example.

Imagine that the states in a graph represent points in a plane and the connectivity is to nearest neighbors. In this case, UC will expand nodes in order of distance from the start point. That is, as time goes by, the expanded points will be located within expanding circular contours centered on the start point. Note, however, that points heading away from the goal will be treated just the same as points that are heading towards the goal.



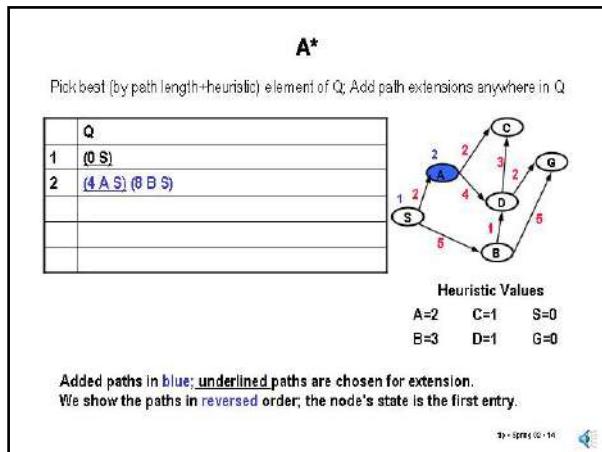
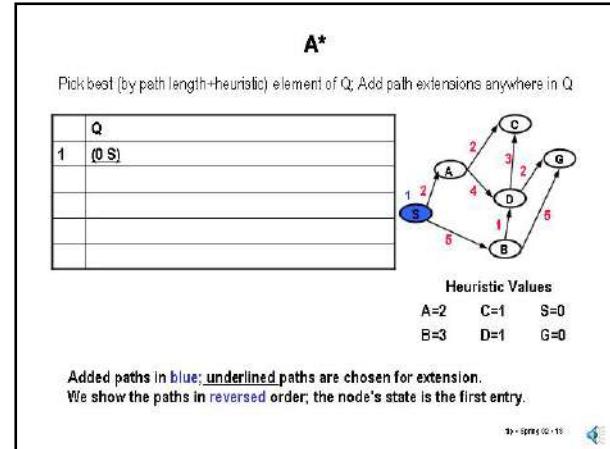
**Slide 2.5.12**

If we add in an estimate of the straight-line distance to the goal, the points expanded will be bounded contours that keep constant the sum of the distance from the start and the distance to the goal, as suggested in the figure. What the underestimate has done is to "bias" the search towards the goal.

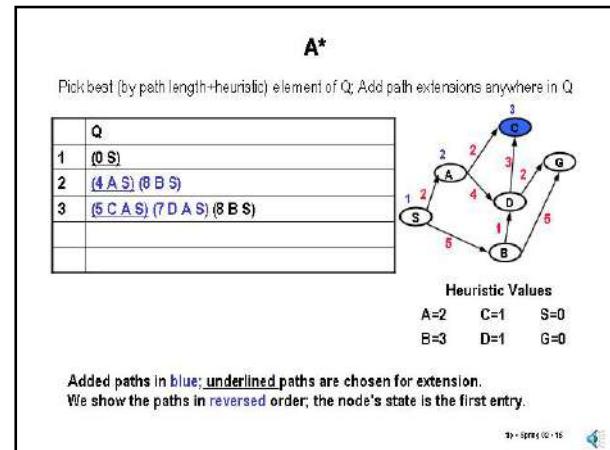
Slide 2.5.13

Let's walk through an example of A*, that is, uniform-cost search using a heuristic which is an underestimate of remaining cost to the goal. In this example we are focusing on the use of the underestimate. The heuristic we will be using is similar to the earlier one but slightly modified to be admissible.

We start at S as usual.

**Slide 2.5.14**

And expand to A and B. Note that we are using the path length + underestimate and so the S-A path has a value of 4 (length 2, estimate 2). The S-B path has a value of 8 (5 + 3). We pick the path to A.

**Slide 2.5.15**

Expand to C and D and pick the path with shorter estimate, to C.

A*

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

	Q
1	(0 S)
2	<u>(4 A S)</u> (8 B S)
3	<u>(5 C A S)</u> (7 D A S) (8 B S)
4	(7 D A S) (8 B S)

Heuristic Values
A=2 C=1 S=0
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

Slide 2.5.16

C has no descendants, so we pick the shorter path (to D).

A*

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

	Q
1	(0 S)
2	<u>(4 A S)</u> (8 B S)
3	<u>(5 C A S)</u> (7 D A S) (8 B S)
4	(7 D A S) (8 B S)
5	<u>(8 G D A S)</u> (10 C D A S) (8 B S)

Heuristic Values
A=2 C=1 S=0
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

Slide 2.5.18

So, we stop with a path to the goal of length 8.

A*

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

	Q
1	(0 S)
2	<u>(4 A S)</u> (8 B S)
3	<u>(5 C A S)</u> (7 D A S) (8 B S)
4	(7 D A S) (8 B S)
5	<u>(8 G D A S)</u> (10 C D A S) (8 B S)

Heuristic Values
A=2 C=1 S=0
B=3 D=1 G=0

Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

Slide 2.5.19

It is important to realize that not all heuristics are admissible. In fact, the rather arbitrary heuristic values we used in our best-first example are not admissible given the path lengths we later assigned. In particular, the value for D is bigger than its distance to the goal and so this set of distances is not everywhere an underestimate of distance to the goal from every node. Note that the (arbitrary) value assigned for S is also an overestimate but this value would have no ill effect since at the time S is expanded there are no alternatives.

Not all heuristics are admissible

Given the link lengths in the figure, is the table of heuristic values that we used in our earlier best-first example an admissible heuristic?

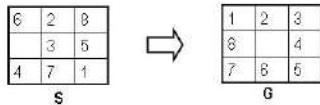
No!

- A is ok
- B is ok
- C is ok
- D is too big, needs to be ≤ 2
- S is too big, can always use 0 for start

	Heuristic Values
A=2	C=1 S=10
B=3	D=4 G=0

Admissible Heuristics

8 Puzzle: Move tiles to reach goal. Think of a move as moving "empty" tile.



Alternative underestimates of "distance" (number of moves) to goal:

- Number of misplaced tiles (7 in example above)

Slide 2.5.20

Although it is easy and intuitive to illustrate the concept of a heuristic by using the notion of straight-line distance to the goal in Euclidean space, it is important to remember that this is by no means the only example.

Take solving the so-called 8-puzzle, in which the goal is to arrange the pieces as in the goal state on the right. We can think of a move in this game as sliding the "empty" space to one of its nearest vertical or horizontal neighbors. We can help steer a search to find a short sequence of moves by using a heuristic estimate of the moves remaining to the goal.

One admissible estimate is simply the number of misplaced tiles. No move can get more than one misplaced tile into place, so this measure is a guaranteed underestimate and hence admissible.

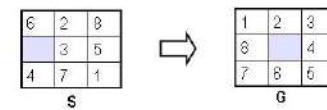
Slide 2.5.21

We can do better if we note that, in fact, each move can at best decrease by one the "Manhattan" (aka Taxicab, aka rectilinear) distance of a tile from its goal.

So, the sum of these distances for each misplaced tile is also an underestimate. Note that it is always a better (larger) underestimate than the number of misplaced tiles. In this example, there are 7 misplaced tiles (all except tile 2), but the Manhattan distance estimate is 17 (4 for tile 1, 0 for tile 2, 2 for tile 3, 3 for tile 4, 1 for tile 5, 3 for tile 6, 1 for tile 7 and 3 for tile 8).

Admissible Heuristics

8 Puzzle: Move tiles to reach goal. Think of a move as moving "empty" tile.



Alternative underestimates of "distance" (number of moves) to goal:

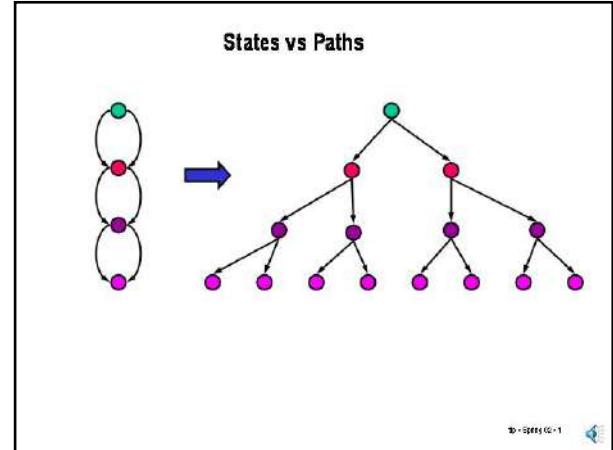
- Number of misplaced tiles (7 in example above)
- Sum of Manhattan distance of tile to its goal location (17 in example above). Manhattan distance between (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$. Each move can only decrease the distance of exactly one tile.

The second of these is much better at predicting actual number of moves.

6.034 Notes: Section 2.6

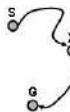
Slide 2.6.1

In our discussion of uniform-cost search and A* so far, we have ignored the issue of revisiting states. We indicated that we could not use a Visited list and still preserve optimality, but can we use something else that will keep the worst-case cost of a search proportional to the number of states in a graph rather than to the number of non-looping paths? The answer is yes. We will start looking at uniform-cost search, where the extension is straightforward and then tackle A*, where it is not.



Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".



TO SPOTLIGHT

Slide 2.6.2

What will come to our rescue is the so-called "Dynamic Programming Optimality Principle", which is fairly intuitive in this context. Namely, the shortest path from the start to the goal that goes through some state X is made up of the shortest path to X followed by the shortest path from X to G. This is easy to prove by contradiction, but we won't do it here.

Slide 2.6.3

Given this, we know that there is no reason to compute any path except the shortest path to any state, since that is the only path that can ever be part of the answer. So, if we ever find a second path to a previously visited state, we can discard the longer one. So, when adding nodes to Q, check whether another node with the same state is already in Q and keep only the one with shorter path length.

Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.

TO SPOTLIGHT

Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.
- Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.

Slide 2.6.4

We have observed that uniform-cost search pulls nodes off Q (expands them) in order of their actual path length. So, the **first** time we expand a node whose state is X, that node represents the shortest path to that state. Any subsequent path we find to that state is a waste of effort, since it cannot have a shorter path.

10 - SPOTS 12 - 6



Slide 2.6.5

So, let's remember the states that we have expanded already, in a "list" (or, better, a hash table) that we will call the Expanded list. If we try to expand a node whose state is already on the Expanded list, we can simply discard that path. We will refer to algorithms that do this, that is, no expanded state is re-visited, as using a **strict** Expanded list.

Note that when using a strict Expanded list, any visited state will either be in Q or in the Expanded list. So, when we consider a potential new node we can check whether (a) its state is in Q, in which case we accept it or discard it depending on the length of the new path versus the previous best, or (b) it is in Expanded, in which case we always discard it. If the node's state has never been visited, we add the node to Q.

Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.
- Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.
- So, once we expand one path to state X, we don't need to consider (extend) any other paths to X. We can keep a list of these states, call it Expanded. If the state of the search node we pull off of Q is in the Expanded list, we discard the node. When we use the Expanded list this way, we call it "strict".
- Note that UC without this is still correct, but inefficient for searching graphs.

10 - SPOTS 12 - 6



Slide 2.6.6

The correctness of uniform-cost search does not depend on using an expanded list or even on discarding longer paths to the same state (the Q will just be longer than necessary). We can use UC with or without these optimizations and it is still correct. Exploiting the optimality principle by discarding longer paths to states in Q and not re-visiting expanded states can, however, make UC much more efficient for densely connected graphs.

10 - SPOTS 12 - 6



Dynamic Programming Optimality Principle and the Expanded list

- Given that path length is additive, the shortest path from S to G via a state X is made up of the shortest path from S to X and the shortest path from X to G. This is the "dynamic programming optimality principle".
- This means that we only need to keep the single best path from S to any state X; if we find a new path to a state already in Q, discard the longer one.
- Note that the first time UC pulls a search node off of Q whose state is X, this path is the shortest path from S to X. This follows from the fact that UC expands nodes in order of actual path length.
- So, once we expand one path to state X, we don't need to consider (extend) any other paths to X. We can keep a list of these states, call it Expanded. If the state of the search node we pull off of Q is in the Expanded list, we discard the node. When we use the Expanded list this way, we call it "strict".

10 - SPOTS 12 - 6



Slide 2.6.7

So, now we need to modify our simple algorithm to implement uniform-cost search to take advantage of the Optimality Principle. We start with our familiar algorithm...

Simple Optimal Search Algorithm**Uniform Cost**

A search node is a path from some state X to the start state, e.g., (X B A S).
The state of a search node is the most recent state of the path, e.g. X.
Let Q be a list of search nodes, e.g. [(X B A S) (C B A S) ...].
Let S be the start state.

1. Initialize Q with search node (S) as only entry;
2. If Q is empty, fail. Else, pick least cost search node N from Q
3. If state(N) is a goal, return N (we've reached the goal)
4. (Otherwise) Remove N from Q.
5. -
6. Find all the children of state(N) and create all the one-step extensions of N to each descendant.
7. Add all the extended paths to Q;
8. Go to step 2.

10 - SPRES (2) - 7

**Simple Optimal Search Algorithm****Uniform Cost + Strict Expanded List**

A search node is a path from some state X to the start state, e.g., (X B A S).
The state of a search node is the most recent state of the path, e.g. X.
Let Q be a list of search nodes, e.g. [(X B A S) (C B A S) ...].
Let S be the start state.

1. Initialize Q with search node (S) as only entry; set Expanded = ()
2. If Q is empty, fail. Else, pick least cost search node N from Q
3. If state(N) is a goal, return N (we've reached the goal)
4. (Otherwise) Remove N from Q.
5. If state(N) in Expanded, go to step 2, otherwise add state(N) to Expanded.
6. Find all the children of state(N) (Not in Expanded) and create all the one-step extensions of N to each descendant.
7. Add all the extended paths to Q; if descendant state already in Q, keep only shorter path to the state in Q.
8. Go to step 2.

10 - SPRES (2) - 8

Slide 2.6.8

... and modify it. First we initialize the Expanded list in step 1. Since this is uniform-cost search, the algorithm picks the best element of Q, based on path length, in step 2. Then, in step 5, we check whether the state of the new node is on the Expanded list and if so, we discard it. Otherwise, we add the state of the new node to the Expanded list. In step 6, we avoid visiting nodes that are Expanded since that would be a waste of time. In step 7, we check whether there is a node in Q corresponding to each newly visited state, if so, we keep only the shorter path to that state.

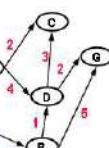
Slide 2.6.9

Let's step through the operation of this algorithm on our usual example. We start with a node for S, having a 0-length path, as usual.

Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	Expanded
1 [(0 S)]	



Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

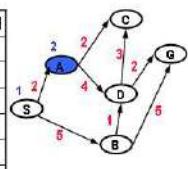
10 - SPRES (2) - 9



Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S



Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

19 - Sprouts (2/10)

Slide 2.6.10

We expand the S node, add its descendants to Q and add the state S to the Expanded list.

Slide 2.6.11

We then pick the node at A to expand since it has the shortest length among the nodes in Q. We get the two extensions of the A node, which gives us paths to C and D. Neither of the two new nodes' states is already present in Q or in Expanded so we add them both to Q. We also add A to the Expanded list.

Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S
3	<u>(4 C A S)</u> (6 D A S) (5 B S)	S,A

Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

19 - Sprouts (2/11)

Slide 2.6.12

We pick the node at C to expand, but C has no descendants. So, we add C to Expanded but there are no new nodes to add to Q.

Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S
3	<u>(4 C A S)</u> (6 D A S) (5 B S)	S,A
4	<u>(6 D A S)</u> (5 B S)	S,A,C

Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

19 - Sprouts (2/12)

Slide 2.6.13

We select the node with the shortest path in Q, which is the node at B with path length 5 and generate the new descendant nodes, one to D and one to G. Note that at this point we have generated two paths to D - (S A D) and (S B D) both with length 6. We're free to keep either one but we do not need both. We will choose to discard the new node and keep the one already in Q.

Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

	Q	Expanded
1	(0 S)	
2	<u>(2 A S)</u> (5 B S)	S
3	<u>(4 C A S)</u> (6 D A S) (5 B S)	S,A
4	<u>(6 D A S)</u> (5 B S)	S,A,C
5	<u>(6 D-B-S)</u> (10 G B S) (6 D A S)	S,A,C,B

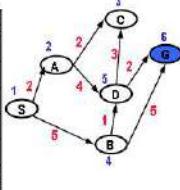
Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

19 - Sprouts (2/13)

Uniform Cost (with strict expanded list)

Pick best (by path length) element of Q; Add path extensions anywhere in Q

Q	Expanded
1 (0 S)	
2 (2 A S) (5 B S)	S
3 (4 C A S) (6 D A S) (5 B S)	S,A
4 (6 D A S) (5 B S)	S,A,C
5 (6 D B S) (10 G B S) (6 D A S)	S,A,C,B
6 (8 G D A S) (9 C D A S) (10 G B S)	S,A,C,B,D



Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

19 / 59 (12 / 16)

Slide 2.6.14

The node corresponding to the (S A D) path is now the shortest path, so we expand it and generate two descendants, one going to C and one going to G. The new C node can be discarded since C is on the Expanded list. The new G node shares its state with a node already on Q, but it corresponds to a shorter path - so we discard the older node in favor of the new one. So, at this point, Q only has one remaining node.

Slide 2.6.15

This node corresponds to the optimal path that is returned. It is easy to show that the use of an Expanded list, as well as keeping only the shortest path to any state in Q, preserve the optimality guarantee of uniform-cost search and can lead to substantial performance improvements. Will this hold true for A* as well?

A* (without expanded list)

- Let $g(N)$ be the path cost of n , where n is a search tree node, i.e. a partial path.
- Let $h(N)$ be $h(state(N))$, the heuristic estimate of the remaining path length to the goal from state(N).
- Let $f(N) = g(N) + h(state(N))$ be the total estimated path cost of a node, i.e. the estimate of a path to a goal that starts with the path given by N .
- A^* picks the node with lowest f value to expand

19 / 59 (12 / 16)

Slide 2.6.16

First, let's review A* and the notation that we have been using. The important notation to remember is that the function g represents actual path length along a partial path to a node's state. The function h represents the heuristic value at a node's state and f is the total estimated path length (to a goal) and is the sum of the actual length (g) and the heuristic estimate (h). A* picks the node with the smallest value of f to expand.

Slide 2.6.17

A^* , without using an Expanded list or discarding nodes in Q but using an admissible heuristic -- that is, one that underestimates the distance to the goal -- is guaranteed to find optimal paths.

A* (without expanded list)

- Let $g(N)$ be the path cost of n , where n is a search tree node, i.e. a partial path.
- Let $h(N)$ be $h(state(N))$, the heuristic estimate of the remaining path length to the goal from state(N).
- Let $f(N) = g(N) + h(state(N))$ be the total estimated path cost of a node, i.e. the estimate of a path to a goal that starts with the path given by N .
- A^* picks the node with lowest f value to expand
- A^* (without expanded list) and with admissible heuristic is guaranteed to find optimal paths – those with smallest path cost.

19 / 59 (12 / 16)

A* and the strict Expanded List

- The strict Expanded list (also known as a Closed list) is commonly used in implementations of A* but, to guarantee finding optimal paths, this implementation requires a stronger condition for a heuristic than simply being an underestimate.

Slide 2.6.18

If we use the search algorithm we used for uniform-cost search with a strict Expanded list for A*, adding in an admissible heuristic to the path length, then we can no longer guarantee that it will always find the optimal path. We need a stronger condition on the heuristics used than being an underestimate.

19 - SPRING 02 - 19



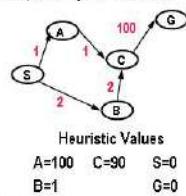
Slide 2.6.19

Here's an example that illustrates this point. The exceedingly optimistic heuristic estimate at B "lures" the A* algorithm down the wrong path.

A* and the strict Expanded List

- The strict Expanded list (also known as a Closed list) is commonly used in implementations of A* but, to guarantee finding optimal paths, this implementation requires a stronger condition for a heuristic than simply being an underestimate.
- Here's a counterexample: The heuristic values listed below are all underestimates but A* using an Expanded list will not find the optimal path. The misleading estimate at B throws the algorithm off, C is expanded before the optimal path to it is found.

Q	Expanded
1 (0 S)	
2 (3 B S) (101 A S)	S
3 (94 C B S) (101 A S)	B, S
4 (101 A S) (104 G C B S) (C, B, S)	C, B, S
5 (104 G C B S)	A, C, B, S



Added paths in blue; underlined paths are chosen for extension.

We show the paths in reversed order; the node's state is the first entry.

19 - SPRING 02 - 20



Slide 2.6.20

You can see the operation of A* in detail here, confirming that it finds the incorrect path. The correct partial path via A is blocked when the path to C via B is expanded. In step 4, when A is finally expanded, the new path to C is not put on Q, because C has already been expanded.

Consistency

- To enable implementing A* using the strict Expanded list, H needs to satisfy the following **consistency** (also known as **monotonicity**) conditions.
 - $h(s_i) = 0$, if n_i is a goal
 - $h(s_i) \leq h(s_j) + c(s_i, s_j)$, for n_j a child of n_i

Slide 2.6.21

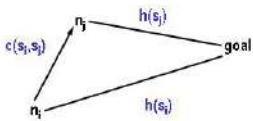
The stronger conditions on a heuristic that enables us to implement A* just the same way we implemented uniform-cost search with a strict Expanded list are known as the **consistency** conditions. They are also called monotonicity conditions by others. The first condition is simple, namely that goal states have a heuristic estimate of zero, which we have already been assuming. The next condition is the critical one. It indicates that the difference in the heuristic estimate between one state and its descendant must be less than or equal to the actual path cost on the edge connecting them.

19 - SPRING 02 - 21



Consistency

- To enable implementing A* using the strict Expanded list, H needs to satisfy the following **consistency** (also known as **monotonicity**) conditions.
 - $h(s_i) = 0$, if n_i is a goal
 - $h(s_j) \leq h(s_i) + c(s_i, s_j)$, for n_j a child of n_i
- That is, the heuristic cost in moving from one entry to the next cannot decrease by more than the arc cost between the states. This is a kind of triangle inequality. This condition is a highly desirable property of a heuristic function and often simply assumed (more on this later).



Slide 2.6.22

The best way of visualizing the consistency condition is as a "triangle inequality", that is, one side of the triangle is less than or equal the sum of the other two sides, as seen on the diagram here.

Slide 2.6.23

Here is a simple example of a (gross) violation of consistency. If you believe goal is 100 units from n_i , then moving 10 units to n_j should not bring you to a distance of 10 units from the goal. These heuristic estimates are not consistent.

A* (without expanded list)

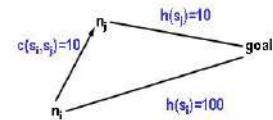
- Let $g(N)$ be the path cost of n , where n is a search tree node, i.e. a partial path.
- Let $h(N)$ be $h(state(N))$, the heuristic estimate of the remaining path length to the goal from state(N).
- Let $f(N) = g(N) + h(state(N))$ be the total estimated path cost of a node, i.e. the estimate of a path to a goal that starts with the path given by n .
- A* picks the node with lowest f value to expand
- A* (without expanded list) and with admissible heuristic is guaranteed to find optimal paths – those with the smallest path cost.**
- This is true even if heuristic is NOT consistent.**

Slide 2.6.24

I want to stress that consistency of the heuristic is only necessary for optimality when we want to discard paths from consideration, for example, because a state has already been expanded. Otherwise, plain A* without using an expanded list requires only that the heuristic be admissible to guarantee optimality.

Consistency Violation

- A simple example of a violation of consistency.
- $h(s_i) > h(s_j) + c(s_i, s_j)$
- In example, $100 - 10 > 10$
- If you believe goal is 100 units from n_i , then moving 10 units to n_j should not bring you to a distance of 10 units from the goal.



Slide 2.6.25

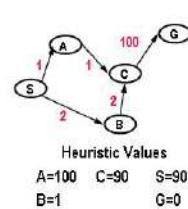
This illustrates that A* without an Expanded list has no trouble coping with the example we saw earlier that showed the pitfalls of using a strict Expanded list. This heuristic is not consistent but it is an underestimate and that is all that is needed for A* without an Expanded list to guarantee optimality.

A* (without expanded list)

Note that heuristic is admissible but not consistent

Q
1 (90 S)
2 (3 B S) (101 A S)
3 (94 C B S) (101 A S)
4 (101 A S) (104 G C B S)
5 (92 C A S) (104 G C B S)
6 (102 G C A S) (104 G C B S)

Added paths in blue; underlined paths are chosen for extension.



A* (with strict expanded list)

- Just like Uniform Cost search.
- When a node N is expanded, if state(N) is in expanded list, discard N, else add state(N) to expanded list.
- If some node in Q has the same state as some descendant of N, keep only node with smaller f, which will also correspond to smaller g.
- For A* (with strict expanded list) to be guaranteed to find the optimal path, the heuristic must be consistent.

Slide 2.6.26

The extension of A* to use a strict expanded list is just like the extension to uniform-cost search. In fact, it is the identical algorithm except that it uses f values instead of g values. But, we stress that for this algorithm to guarantee finding optimal paths, the heuristic must be consistent.

19 - SPRING 02 - 16

**Slide 2.6.27**

If we modify the heuristic in the example we have been considering so that it is consistent, as we have done here by increasing the value of h(B), then A* (even when using a strict Expanded list) will work.

Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?

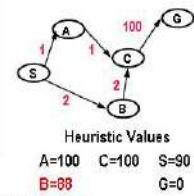
19 - SPRING 02 - 16

**Slide 2.6.29**

The key step needed to enable A* to cope with inconsistent heuristics is to detect when an overly optimistic heuristic estimate has caused us to expand a node prematurely, that is, before the shortest path to that node has been found. This is basically analogous to what we have been doing when we find a shorter path to a state already in Q, except we need to do it to states in the Expanded list. In this modified algorithm, the use of the Expanded list is not strict: we allow re-visiting states on the Expanded list.

To implement this, we will keep in the Expanded list not just the expanded states but the actual node that was expanded. In particular, this records the actual path length at the time of expansion

Q	Expanded
1 <u>(90 S)</u>	
2 <u>(90 B S)</u> (101 A S)	S
3 (101 A S) <u>(104 C B S)</u>	A, S
4 (102 C A S) (104 -C-B-S)	C,A,S
5 (102 G C A S)	G,C,A,S



Added paths in blue; underlined paths are chosen for extension.

19 - SPRING 02 - 17

Slide 2.6.28

People sometimes simply assume that the consistency condition holds and implement A* with a strict Expanded list (also called a Closed list) in the simple way we have shown before. But, this is not the only (or best) option. Later we will see that A* can be adapted to retain optimality in spite of a heuristic that is not consistent - there will be a performance price to be paid however.

Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify A* so that it detects and corrects when inconsistency has led us astray.

19 - SPRING 02 - 18



Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify A* so that it detects and corrects when inconsistency has led us astray:
- Assume we are adding node₁ to Q and node₂ is present in Expanded list with node₁.state = node₂.state.

Slide 2.6.30

Let's consider in detail the operation of the Expanded list if we want to handle inconsistent heuristics while guaranteeing optimal paths.

Assume that we are adding a node, call it node₁, to Q when using an Expanded list. So, we check to see if a node with the same state is present in the Expanded list and we find node₂ which matches.

19 / SPRING 02 / 98



Slide 2.6.31

With a strict Expanded list, we simply discard node₁; we do not add it to Q.

Dealing with inconsistent heuristic

- What can we do if we have an inconsistent heuristic but we still want optimal paths?
- Modify A* so that it detects and corrects when inconsistency has led us astray:
- Assume we are adding node₁ to Q and node₂ is present in Expanded list with node₁.state = node₂.state.
- Strict –
 - do not add node₁ to Q
- Non-Strict Expanded list –
 - If node₁.path_length < node₂.path_length, then
 - Delete node₂ from Expanded list
 - Add node₁ to Q

19 / SPRING 02 / 98



Slide 2.6.32

With a non-strict Expanded list, the situation is a bit more complicated. We want to make sure that node₁ has not found a better path to the state than node₂. If a better path has been found, we remove the old node from Expanded (since it does not represent the optimal path) and add the new node to Q.

19 / SPRING 02 / 98



Slide 2.6.33

Let's think a bit about the worst case complexity of A*, in terms of the number of nodes expanded (or visited).

As we've mentioned before, it is customary in AI to think of search complexity in terms of some "depth" parameter of the domain such as the number of steps in a plan of action or the number of moves in a game. The state space for such domains (planning or game playing) grows exponentially in the "depth", that is, because at each depth level there is some branching factor (e.g., the possible actions) and so the number of states grows exponentially with the depth.

We could equally well speak instead of the number of states as a fixed parameter, call it N, and state our complexity in terms of N. We just have to keep in mind then that in many applications, N grows exponentially with respect to the depth parameter.

Worst Case Complexity

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.

19 / SPRING 02 / 98



Worst Case Complexity

- The number of states in the search space may be exponential in some “depth” parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited or expanded lists, may have to visit (or expand) each state in the worst case.
- So, all searches will have worst case complexities that are at least proportional to the number of states and therefore exponential in the “depth” parameter.
- This is the bottom-line irreducible worst case cost of systematic searches.

Slide 2.6.34

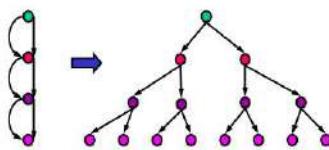
In the worst case, when the heuristics are not very useful or the nodes are arranged in the worst possible way, all the search methods may end up having to visit or expand all of the states (up to some depth). In practice, we should be able to avoid this worst case but in many cases one comes pretty close.

Slide 2.6.35

The problem is that if we have no memory of what states we've visited or expanded, then the worst case for a densely connected graph can be much, much worse than this. One may end up doing exponentially more work.

Worst Case Complexity

- A state space with N states may give rise to a search tree that has a number of nodes that is exponential in N , as in this example.



Slide 2.6.36

We've seen this example before. It shows that a state space with N states can generate a search tree with 2^N nodes.

Worst Case Complexity

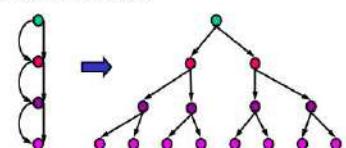
- The number of states in the search space may be exponential in some “depth” parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited or expanded lists, may have to visit (or expand) each state in the worst case.
- So, all searches will have worst case complexities that are at least proportional to the number of states and therefore exponential in the “depth” parameter.
- This is the bottom-line irreducible worst case cost of systematic searches.
- Without memory of what states have been visited (expanded), searches can do (much) worse than visit every state.

Slide 2.6.37

A search algorithm that does not keep a visited or expanded list will do exponentially more work that necessary. On the other hand, if we use a strict expanded list, we will never expand more than the (unavoidable) N states.

Worst Case Complexity

- A state space with N states may give rise to a search tree that has a number of nodes that is exponential in N , as in this example.



- Searches without a visited (expanded) list may, in the worst case, visit (expand) every node in the search tree.
- Searches with strict visited (expanded lists) will visit (expand) each state only once.

Optimality & Worst Case Complexity

Algorithm	Heuristic	Expanded List	Optimality Guaranteed?	Worst Case # Expansions
Uniform Cost	None	Strict	Yes	N
A*	Admissible	None	Yes	>N
A*	Consistent	Strict	Yes	N
A*	Admissible	Strict	No	N
A*	Admissible	Non Strict	Yes	>N

N is number of states in graph

19 / 39 (12 / 38)

Here we summarize the optimality and complexity of the various algorithms we have been examining.

6.034 Notes: Section 2.7

Slide 2.7.1

This set of slides goes into more detail on some of the topics we have covered in this chapter.

Optional Topics

- These slides go into more depth on a variety of topics we have touched upon:
 - Optimality of A*
 - Impact of a better heuristic on A*
 - Why does consistency guarantee optimal paths for A* with strict expanded list
 - Algorithmic issues for A*
 - These are not required and are provided for those interested in pursuing these topics.
- 19 / 39 (12 / 38)

Optimality of A*

- Assume A* has expanded a path to goal node G

19 / 39 (12 / 2)

Slide 2.7.2

First topic:

Let's go through a quick proof that A* actually finds the optimal path. Start by assuming that A* has selected a node G.

Slide 2.7.3

Then, we know from the operation of A* that it has expanded all nodes N whose cost $f(N)$ is strictly less than the cost of G. We also know that since the heuristic is admissible, its value at a goal node must be 0 and thus, $f(G) = g(G) + h(G) = g(G)$. Therefore, every unexpanded node N must have $f(N)$ greater or equal to the actual path length to G.

Optimality of A*

- Assume A* has expanded a path to goal node G
- Then, A* has expanded all nodes N where $f(N) < f(G)$. Since h is admissible, $f(G) = g(G)$. So, every unexpanded node has $f(N) \geq g(G)$.

10 / 39 10:12:45

**Optimality of A***

- Assume A* has expanded a path to goal node G
- Then, A* has expanded all nodes N where $f(N) < f(G)$. Since h is admissible, $f(G) = g(G)$. So, every unexpanded node has $f(N) \geq g(G)$.
- Since h is admissible, we know that any path through N that reaches a goal node G^0 has value $g(G^0) \geq f(N)$

10 / 39 10:12:45

Slide 2.7.4

Since h is admissible, we know that any path through an unexpanded node N that reaches some alternate goal node G' must have a total cost estimate $f(N)$ that is not larger than the actual cost to G' , that is, $g(G')$.

Slide 2.7.5

Combining these two statements we see that the path length to any other goal node G' must be greater or equal to the path length of the goal node A* found, that is, G.

Optimality of A*

- Assume A* has expanded a path to goal node G
- Then, A* has expanded all nodes N where $f(N) < f(G)$. Since h is admissible, $f(G) = g(G)$. So, every unexpanded node has $f(N) \geq g(G)$.
- Since h is admissible, we know that any path through N that reaches a goal node G^0 has value $g(G^0) \geq f(N)$
- So, for every unexpanded node N, we have $g(G^0) \geq f(N) \geq g(G)$. That is, any goal reachable from those nodes has a path that is at least as long as the one we found.

10 / 39 10:12:45

**Impact of better heuristic**

- Let h^* be the "perfect" heuristic – returns actual path cost to goal.

10 / 39 10:12:45

Slide 2.7.6

Next topic:

We can also show that a better heuristic in general leads to improved performance of A* (or at least no decrease). By performance, we mean number of nodes expanded. In general, there is a tradeoff in how much effort we do to compute a better heuristic and the improvement in the search time due to reduced number of expansions.

Let's postulate a "perfect" heuristic which computes the actual optimal path length to a goal. Call this heuristic h^* .

Slide 2.7.7

Then, assume we have a heuristic h_1 that is always numerically less than another heuristic h_2 , which is (admissibility) less than or equal to h^* .

Impact of better heuristic

- Let h^* be the "perfect" heuristic – returns actual path cost to goal.
- If $h_1(N) < h_2(N) \leq h^*(N)$ for all non-goal nodes, then h_2 is a better heuristic than h_1 .

10 / Spr 04 / 7

Impact of better heuristic

- Let h^* be the "perfect" heuristic – returns actual path cost to goal.
- If $h_1(N) < h_2(N) \leq h^*(N)$ for all non-goal nodes, then h_2 is a better heuristic than h_1 .
- If A_1^* uses h_1 , and A_2^* uses h_2 , then every node expanded by A_2^* is also expanded by A_1^* .
 - $f_1(G) = f_2(G) = g(G)$, so both A_1^* and A_2^* expand all nodes with $f < g(G)$
 - $f_1(N) = g(N) + h_1(N) < f_2(N) = g(N) + h_2(N) \leq g(G)$

10 / Spr 04 / 8

Slide 2.7.8

The key observation is that if we have two versions of A^* , one using h_1 and the other using h_2 , then every node expanded by the second one is also expanded by the first.

This follows from the observation we have made earlier that at a goal, the heuristic estimates all agree (they are all 0) and so we know that both versions will expand all nodes whose value of f is less than the actual path length of G .

Now, every node expanded by A_2^* , will have a path cost no greater than the actual cost to the goal G . Such a node will have a smaller cost using h_1 and so it will definitely be expanded by A_1^* as well.

10 / Spr 04 / 7

Impact of better heuristic

- Let h^* be the "perfect" heuristic – returns actual path cost to goal.
- If $h_1(N) < h_2(N) \leq h^*(N)$ for all non-goal nodes, then h_2 is a better heuristic than h_1 .
- If A_1^* uses h_1 , and A_2^* uses h_2 , then every node expanded by A_2^* is also expanded by A_1^* .
 - $f_1(G) = f_2(G) = g(G)$, so both A_1^* and A_2^* expand all nodes with $f < g(G)$
 - $f_1(N) = g(N) + h_1(N) < f_2(N) = g(N) + h_2(N) \leq g(G)$
- That is, A_1^* expands at least as many nodes as A_2^* and we say that A_1^* is better informed than A_2^* .

10 / Spr 04 / 9

Impact of better heuristic

- Let h^* be the "perfect" heuristic – returns actual path cost to goal.
- If $h_1(N) < h_2(N) \leq h^*(N)$ for all non-goal nodes, then h_2 is a better heuristic than h_1 .
- If A_1^* uses h_1 , and A_2^* uses h_2 , then every node expanded by A_2^* is also expanded by A_1^* .
 - $f_1(G) = f_2(G) = g(G)$, so both A_1^* and A_2^* expand all nodes with $f < g(G)$
 - $f_1(N) = g(N) + h_1(N) < f_2(N) = g(N) + h_2(N) \leq g(G)$
- That is, A_1^* expands at least as many nodes as A_2^* and we say that A_1^* is better informed than A_2^* .
- Note that A^* with any non-zero admissible heuristic is better informed (and therefore typically expands fewer nodes) than Uniform Cost search.

10 / Spr 04 / 10

Slide 2.7.10

Since uniform-cost search is simply A^* with a heuristic of 0, we can say that A^* is generally better informed than UC and we expect it to expand fewer nodes. But, A^* will expend additional effort computing the heuristic value -- a good heuristic can more than pay back that extra effort.

Slide 2.7.11

New topic:

Why does consistency allow us to guarantee that A* will find optimal paths? The key insight is that consistency ensures that the f values of expanded nodes will be non-decreasing over time.

Consider two nodes N_i and N_j such that the latter is a descendant of the former in the search tree. Then, we can write out the values of f as shown here, involving the actual path length $g(N_j)$, the cost of the edge between the nodes $c(N_i, N_j)$ and the heuristic values of the two corresponding states.

Consistency → Non-decreasing f

- N_j is a descendant of N_i in the search tree
- $f(N_i) = g(N_i) + h(state(N_i))$
- $f(N_j) = g(N_i) + h(state(N_j)) = g(N_i) + c(N_i, N_j) + h(state(N_j))$
- By consistency, $h(state(N_i)) \leq h(state(N_j)) + c(N_i, N_j)$
- Then, $f(N_j) \geq f(N_i)$
- Thus, when A* with a consistent heuristic, expands a node, all of its descendants have f values greater or equal to the expanded node (as do all the nodes left on Q). So, the f values of expanded nodes can never decrease.

19 / 3995 (2 / 11)

**Consistency → Non-decreasing f**

- N_j is a descendant of N_i in the search tree
- $f(N_i) = g(N_i) + h(state(N_i))$
- $f(N_j) = g(N_i) + h(state(N_j)) = g(N_i) + c(N_i, N_j) + h(state(N_j))$
- By consistency, $h(state(N_i)) \leq h(state(N_j)) + c(N_i, N_j)$
- Then, $f(N_j) \geq f(N_i)$
- Thus, when A* with a consistent heuristic, expands a node, all of its descendants have f values greater or equal to the expanded node (as do all the nodes left on Q). So, the f values of expanded nodes can never decrease.

19 / 3995 (2 / 12)

Slide 2.7.12

By consistency of the heuristic estimates, we know that the heuristic estimate cannot decrease more than the edge cost. So, the value of f in the descendant node cannot go down; it must stay the same or go up.

By this reasoning we can conclude that whenever A* expands a node, the new nodes' f values must be greater or equal to that of the expanded node. Also, since the expanded node must have had an f value that was a minimum of the f values in Q, this means that no nodes in Q after this expansion can have a lower f value than the most recently expanded node. That is, if we track the series of f values of expanded nodes over time, this series is non-decreasing.

19 / 3995 (2 / 15)

**Non-decreasing f → first path is optimal**

- A* with consistent heuristic expands nodes N in non-decreasing order of f(N)
- Then, when a node N is expanded, we have found the shortest path to the corresponding s=state(N)

19 / 3995 (2 / 15)

**Non-decreasing f → first path is optimal**

- A* with consistent heuristic expands nodes N in non-decreasing order of f(N)
- Then, when a node N is expanded, we have found the shortest path to the corresponding s=state(N)
- Imagine that we later found another node N' with the same corresponding state s then we know that
 - $f(N') \geq f(N)$
 - $f(N) = g(N) + h(s)$
 - $f(N') = g(N') + h(s)$

19 / 3995 (2 / 16)

Slide 2.7.14

To prove this, let's assume that we later found another node N' that corresponds to the same state as a previously expanded node N. We have shown that the f value of N' is greater or equal that of N. But, since the heuristic values of these nodes must be the same - since they correspond to the same underlying graph state - the difference in f values must be accounted by a difference in actual path length.

Slide 2.7.15

So, we can conclude that the second path cannot be shorter than the first path we already found, and so we can ignore the new path!

Non-decreasing f → first path is optimal

- A* with consistent heuristic expands nodes N in non-decreasing order of f(N)
- Then, when a node N is expanded, we have found the shortest path to the corresponding s=state(i)
- Imagine that we later found another node N' with the same corresponding state s then we know that
 - $f(N') \geq f(N)$
 - $f(N) = g(N) + h(s)$
 - $f(N') = g(N') + h(s)$
- So, we can conclude that
 - $g(N') \geq g(N)$
- And we can safely ignore the second path to s as we would with the strict Expanded list.

19 - SPFA (2 / 16)

**Uniform Cost + Strict Expanded List**
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

19 - SPFA (2 / 16)

Slide 2.7.16

Final topic:

Let's analyze the behavior of uniform-cost search with a strict Expanded List. This algorithm is very similar to the well known Dijkstra's algorithm for shortest paths in a graph, but we will keep the name we have been using. This analysis will apply to A* with a strict Expanded list, since in the worst case they are the same algorithm.

To simplify our approach to the analysis, we can think of the algorithm as boiled down to three steps.

1. Pulling paths off of Q,
2. Checking whether we are done and
3. Adding the relevant path extensions to Q.

In what follows, we assume that the Expanded list is not a "real" list but some constant-time way of checking that a state has been expanded (e.g., by looking at a mark on the state or via a hash-table).

We also assume that Q is implemented as a hash table, which has constant time access (and insertion) cost. This is so we can find whether a node with a given state is already on Q.

Slide 2.7.17

Later, it will become important to distinguish the case of "sparse" graphs, where the states have a nearly constant number of neighbors and "dense" graphs where the number of neighbors grows with the number of states. In the dense case, the total number of edges is $O(N^2)$, which is substantial.

Uniform Cost + Strict Expanded List

(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are dense. Graphs where the nodes have a nearly constant number of links are sparse. For dense graphs, L is $O(N^2)$.

19 - SPFA (2 / 16)



Uniform Cost + Strict Expanded List
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are **dense**. Graphs where the nodes have a nearly constant number of links are **sparse**. For dense graphs, L is $O(N^2)$.

Nodes taken from Q ?	$O(N)$
----------------------	--------

19 - Spring (2) - 18

Slide 2.7.18

So, let's ask the question, how many nodes are taken from Q (expanded) over the life of the algorithm (in the worst case)? Here we assume that when we add a node to Q, we check whether a node already exists for that state and keep only the node with the shorter path. Given this and the use of a strict Expanded list, we know that the worst-case number of expansions is N, the total number of states.

Slide 2.7.19

What's the cost of expanding a node? Assume we scan Q to pick the best paths. Then the cost is of the order of the number of paths in Q, which is $O(N)$ also, since we only keep the best path to a state.

Uniform Cost + Strict Expanded List
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are **dense**. Graphs where the nodes have a nearly constant number of links are **sparse**. For dense graphs, L is $O(N^2)$.

Nodes taken from Q ?	$O(N)$
Cost of picking a node from Q using linear scan?	$O(N)$
Attempts to add nodes to Q (many are rejected)?	$O(L)$

19 - Spring (2) - 19

Slide 2.7.20

How many times do we (attempt to) add paths to Q? Well, since we expand every state at most once and since we only add paths to direct neighbors (links) of that state, then the total number is bounded by the total number of links in the graph.

Slide 2.7.21

Adding to the Q, assuming it is a hash table, as we have been assuming here, can be done in constant time.

Uniform Cost + Strict Expanded List
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are **dense**. Graphs where the nodes have a nearly constant number of links are **sparse**. For dense graphs, L is $O(N^2)$.

Nodes taken from Q ?	$O(N)$
Cost of picking a node from Q using linear scan?	$O(N)$
Attempts to add nodes to Q (many are rejected)?	$O(L)$
Cost of adding a node to Q ?	$O(1)$

19 - Spring (2) - 20

Uniform Cost + Strict Expanded List
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are **dense**. Graphs where the nodes have a nearly constant number of links are **sparse**. For dense graphs, L is $O(N^2)$.

Nodes taken from Q ?	$O(N)$
Cost of picking a node from Q using linear scan?	$O(N)$

19 - Spring (2) - 21

Uniform Cost + Strict Expanded List
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are **dense**. Graphs where the nodes have a nearly constant number of links are **sparse**. For dense graphs, L is $O(N^2)$.

Nodes taken from Q ?	$O(N)$
Cost of picking a node from Q using linear scan?	$O(N)$
Attempts to add nodes to Q (many are rejected)?	$O(L)$
Cost of adding a node to Q ?	$O(1)$

19 - Spring (2) - 21

A*
(order of time growth in worst case)

Our simple algorithm can be summarized as follows:

1. Take the best search node from Q
2. Are we there yet?
3. Add path extensions to Q

Assume strict Expanded "list" is implemented as a hash table, which gives constant time access. Q also implemented as a hash table.

Assume we have a graph with N nodes and L links. Graphs where nodes have $O(N)$ links are **dense**. Graphs where the nodes have a nearly constant number of links are **sparse**. For dense graphs, L is $O(N^2)$.

Nodes taken from Q ?	$O(N)$
Cost of picking a node from Q using linear scan?	$O(N)$
Attempts to add nodes to Q (many are rejected)?	$O(L)$
Cost of adding a node to Q ?	$O(1)$
Total cost ?	$O(N^2 + L)$

19 / Spr 02 / 22

Slide 2.7.22

Putting it all together gives us a total cost on the order of $O(N^2+L)$ which, since L is at worst $O(N^2)$ is essentially $O(N^2)$.

Slide 2.7.23

If you know about priority queues, you might think that they are natural as implementation of Q, since one can efficiently find the best element in such a queue.

Should we use a Priority Queue?

- A priority queue is a data structure that makes it efficient to identify the "best" element of a set. A PQ is typically implemented as a balanced tree.
- The time to find best element in a PQ grows as $O(\log N)$ for a set of size N. This is very much better than N for large N. Also, note that even if we don't discard paths to Expanded nodes, the access is still $O(\log N)$, since $O(\log N^2)=O(\log N)$.

19 / Spr 02 / 22

Should we use a Priority Queue?

- A priority queue is a data structure that makes it efficient to identify the "best" element of a set. A PQ is typically implemented as a balanced tree.
- The time to find best element in a PQ grows as $O(\log N)$ for a set of size N. This is very much better than N for large N. Also, note that even if we don't discard paths to Expanded nodes, the access is still $O(\log N)$, since $O(\log N^2)=O(\log N)$.
- However, adding elements to a PQ also has time that grows as $O(\log N)$.
- Our algorithm does up to N "find best" operations and it does up to L "add" operations. If Q is a PQ, then cost is $O(N \log N + L \log N)$.

19 / Spr 02 / 24

Slide 2.7.24

Note, however, that adding elements to such a Q is more expensive than adding elements to a list or a hash table. So, whether it's worth it depends on how many additions are done. As we said, this is order of L, the number of links.

Should we use a Priority Queue?

- A priority queue is a data structure that makes it efficient to identify the "best" element of a set. A PQ is typically implemented as a balanced tree.
- The time to find best element in a PQ grows as $O(\log N)$ for a set of size N. This is very much better than N for large N. Also, note that even if we don't discard paths to Expanded nodes, the access is still $O(\log N)$, since $O(\log N^2)=O(\log N)$.
- However, adding elements to a PQ also has time that grows as $O(\log N)$.
- Our algorithm does up to N "find best" operations and it does up to L "add" operations. If Q is a PQ, then cost is $O(N \log N + L \log N)$.
- If graph is dense, and L is $O(N^2)$, then a PQ is not advisable.
- If graph is sparse (the more common case), and L is $O(N)$, then a PQ is highly desirable.

19 / Spr 02 / 25

Slide 2.7.25

For a dense graph, where L is $O(N^2)$, then the priority queue will not be worth it. But, for a sparse graph it will.

Cost and Performance				
Searching a tree with N nodes and L links				
Search Method	Worst Time (Dense)	Worst Time (Sparse)	Worst Space	Guaranteed to find shortest path
Uniform Cost A*	$O(N^2)$	$O(N \log N)$	$O(N)$	Yes
Searching a tree with branching factor b and depth d $L = N = b^{d+1}$				
Worst case time is proportional to number of nodes created Worst case space is proportional to maximal length of Q (and Expanded)				

Slide 2.7.26

Here we summarize the worst-case performance of UC (and A*, which is the same). Note, however, that we expect A* with a good heuristic to outperform UC in practice since it will expand at most as many nodes as UC. The worst case cost (with an uninformative heuristic) remains the same.

By the way, in talking about space we have focused on the number of entries in Q but have not mentioned the length of the paths. One might think that this would actually be the dominant factor. But, recall that we are unrolling the graph into the search tree and each node only needs to have a link to its unique ancestor in the tree and so a node really requires constant space.

As before, you can think of the performance of these algorithms as a low-order polynomial (N^2) or as an intractable exponential, depending on how one describes the search space.



6.034 Notes: Section 3.1

Slide 3.1.1

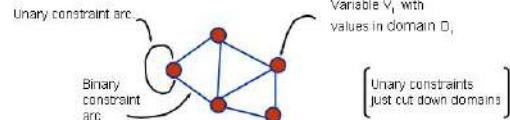
In this presentation, we'll take a look at the class of problems called Constraint Satisfaction Problems (CSPs). CSPs arise in many application areas: they can be used to formulate scheduling tasks, robot planning tasks, puzzles, molecular structures, sensory interpretation tasks, etc.

In particular, we'll look at the subclass of Binary CSPs. A binary CSP is described in term of a set of Variables (denoted V_i), a domain of Values for each of the variables (denoted D_i) and a set of constraints involving the combinations of values for two of the variables (hence the name "binary"). We'll also allow "unary" constraints (constraints on a single variable), but these can be seen simply as cutting down the domain of that variable.

We can illustrate the structure of a CSP in a diagram, such as this one, that we call a **constraint graph** for the problem.

Constraint Satisfaction Problems

General class of Problems: Binary CSP

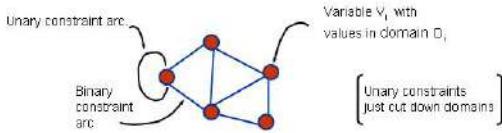


This diagram is called a **constraint graph**

16 - Sept 09 - 1

Constraint Satisfaction Problems

General class of Problems: Binary CSP



This diagram is called a **constraint graph**

Basic problem:

Find a $d_i \in D_i$ for each V_i s.t. all constraints satisfied
(finding consistent labeling for variables)

Slide 3.1.2

The solution of a CSP involves finding a value for each variable (drawn from its domain) such that all the constraints are satisfied. Before we look at how this can be done, let's look at some examples of CSP.

Slide 3.1.3

A CSP that has served as a sort of benchmark problem for the field is the so-called N-Queens problem, which is that of placing N queens on an NxN chessboard so that no two queens can attack each other.

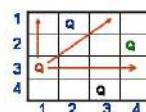
One possible formulation is that the variables are the chessboard positions and the values are either Queen or Blank. The constraints hold between any two variables representing positions that are on a line. The constraint is satisfied whenever the two values are not both Queen.

This formulation is actually very wasteful, since it has N^2 variables. A better formulation is to have variables correspond to the columns of the board and values to the index of the row where the Queen for that column is to be placed. Note that no two queens can share a column and that every column must have a Queen on it. This choice requires only N variables and also fewer constraints to be checked.

In general, we'll find that there are important choices in the formulation of a CSP.

N-Queens as CSP Classic "benchmark" problem

Place N queens on an NxN chessboard so that none can attack the other.



Variables are board positions in NxN chessboard

Domains Queen or blank

Constraints Two positions on a line (vertical, horizontal, diagonal) cannot both be Q

16 - Sept 09 - 2

Line labelings as CSP

Label lines in drawing as convex (+), concave (-), or boundary (>).

Variables are line junctions

Domains are set of legal labels for that junction type

Constraints shared lines between adjacent junctions must have same label.

All legal junction labels for four junction types

1b - Sept 00 - 4

Slide 3.1.4

The problem of labeling the lines in a line-drawing of blocks as being either convex, concave or boundary, is the problem that originally brought the whole area of CSPs into prominence. Waltz's approach to solving this problem by propagation of constraints (which we will discuss later) motivated much of the later work in this area.

In this problem, the variables are the junctions (that is, the vertices) and the values are a combination of labels (+, -, >) attached to the lines that make up the junction. Some combinations of these labels are physically realizable and others are not. The basic constraint is that junctions that share a line must agree on the label for that line.

Note that the more natural formulation that uses lines as the variables is not a BINARY CSP, since all the lines coming into a junction must be simultaneously constrained.

Slide 3.1.5

Scheduling actions that share resources is also a classic case of a CSP. The variables are the activities, the values are chunks of time and the constraints enforce exclusion on shared resources as well as proper ordering of the tasks.

Scheduling as CSP

Choose time for activities e.g. observations on Hubble telescope, or terms to take required classes.

Graph Coloring as CSP

Pick colors for map regions, avoiding coloring adjacent regions with the same color

Variables regions

Domains colors allowed

Constraints adjacent regions must have different colors

1b - Sept 00 - 6

Slide 3.1.6

Another classic CSP is that of coloring a graph given a small set of colors. Given a set of regions with defined neighbors, the problem is to assign a color to each region so that no two neighbors have the same color (so that you can tell where the boundary is). You might have heard of the famous [Four Color Theorem](#) that shows that four colors are sufficient for any planar map. This theorem was a conjecture for more than a century and was not proven until 1976. The CSP is not proving the general theorem, just constructing a solution to a particular instance of the problem.

Slide 3.1.7

A very important class of CSPs is the class of boolean satisfiability problems. One is given a formula over boolean variables in conjunctive normal form (a set of ORs connected with ANDs). The objective is to find an assignment that makes the formula true, that is, a satisfying assignment.

SAT problems are easily transformed into the CSP framework. And, it turns out that many important problems (such as constructing a plan for a robot and many circuit design problems) can be turned into (huge) SAT problems. So, a way of solving SAT problems efficiently in practice would have great practical impact.

However, SAT is the problem that was originally used to show that some problems are [NP-complete](#), that is, as hard as any problem whose solution can be checked in polynomial time. It is generally believed that there is no polynomial time algorithm for NP-complete problems. That is, that any guaranteed algorithm has a worst-case running time that grows exponentially with the size of the problem. So, at best, we can only hope to find a heuristic approach to SAT problems. More on this later.

3-SAT as CSP

The original NP-complete problem

Find values for boolean variables A,B,C,... that satisfy the formula.

(A or B or !C) and (!A or C or B) ...

Variables clauses

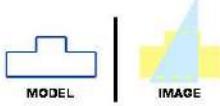
Domains boolean variable assignments that make clause true

Constraints clauses with shared boolean variables must agree on value of variable

1b - Sept 00 - 7

Model-based recognition as CSP

Find given model in edge image, with rotation and translation allowed.



Variables	edges in model
Domains	set of edges in image
Constraints	angle between model & image edges must match

1b - Sept 00 - 8

Slide 3.1.8

Model-based recognition is the problem of finding an instance of a known geometric model, described, for example, as a line-boundary in an image which has been pre-processed to identify and fit lines to the boundaries. The position and orientation of the instance, if any, is not known.

There are a number of constraints that need to be satisfied by edges in the image that correspond to edges in the model. Notably, the angles between pairs of edges must be preserved.

Slide 3.1.9

So, looking through these examples of CSPs we have some good news and bad news. The good news is that CSP is a very general class of problems containing many interesting practical problems. The bad news is that CSPs include many problems that are intractable in the worst case. So, we should not be surprised to find that we do not have efficient guaranteed solutions for CSP. At best, we can hope that our methods perform acceptably in the class of problems we are interested in. This will depend on the structure of the domain of applicability and will not follow directly from the algorithms.

Good News / Bad News

Good News - very general & interesting class problems

Bad News - includes NP-Hard (intractable) problems

So, good behavior is a function of domain not the formulation as CSP.

1b - Sept 00 - 9

CSP Example

Given 40 courses (8.01, 8.02, ..., 6.840) & 10 terms (Fall 1, Spring 1, ..., Spring 5). Find a legal schedule.

1b - Sept 00 - 10

Slide 3.1.10

Let us take a particular problem and look at the CSP formulation in detail. In particular, let's look at an example which should be very familiar to MIT EECS students.

The problem is to schedule approximately 40 courses into the 10 terms for an MEng. For simplicity, let's assume that the list of courses is given to us.

Slide 3.1.11

The constraints we need to represent and enforce are as follows:

- The pre-requisites of a course were taken in an earlier term (we assume the list contains all the pre-requisites).
- Some courses are only offered in the Fall or the Spring term.
- We want to limit the schedule to a feasible load such as 4 courses a term.
- And, we want to avoid time conflicts where we cannot sign up for two courses offered at the same time.

CSP Example

Given 40 courses (8.01, 8.02, ..., 6.840) & 10 terms (Fall 1, Spring 1, ..., Spring 5). Find a legal schedule.

Constraints	Pre-requisites
	Courses offered on limited terms
	Limited number of courses per term
	Avoid time conflicts

1b - Sept 00 - 11

CSP Example

Given 40 courses (8.01, 8.02, ..., 6.840) & 10 terms (Fall 1, Spring 1, ..., Spring 5). Find a legal schedule.

Constraints

- Pre-requisites
- Courses offered on limited terms
- Limited number of courses per term
- Avoid time conflicts

Note, CSPs are not for expressing (soft) preferences e.g., minimize difficulty, balance subject areas, etc.

11b - Sept 00 - 12

Slide 3.1.12

Note that all of these constraints are either satisfied or not. CSPs are not typically used to express preferences but rather to enforce hard and fast constraints.

Slide 3.1.13

One key question that we must answer for any CSP formulation is "What are the variables and what are the values?" For our class scheduling problem, a number of options come to mind. For example, we might pick the terms as the variables. In that case, the values are combinations of four courses that are **consistent**, meaning that they are offered in the same term and whose times don't conflict. The pre-requisite constraint would relate every pair of terms and would require that no course appear in a term before that of any of its pre-requisite course.

This perfectly valid formulation has the practical weakness that the domains for the variables are huge, which has a dramatic effect on the running time of the algorithms.

Choice of variables & values

VARIABLES

A. Terms?

DOMAINS

Legal combinations of for example 4 courses (but this is huge set of values).

11b - Sept 00 - 13

Choice of variables & values

VARIABLES

DOMAINS

A. Terms?

Legal combinations of for example 4 courses (but this is huge set of values).

B. Term Slots?

subdivide terms into slots e.g. 4 of them
(Fall 1,1) (Fall 1,2)
(Fall 1,3) (Fall 1,4)

Courses offered during that term

11b - Sept 00 - 14

Slide 3.1.14

One way of avoiding the combinatorics of using 4-course schedules as the values of the variables is to break up each term into "term slots" and assign to each term-slot a single course. This formulation, like the previous one, has the limit on the number of courses per term represented directly in the graph, instead of stating an explicit constraint. With this representation, we will still need constraints to ensure that the courses in a given term do not conflict and the pre-requisite ordering is enforced. The availability of a course in a given term could be enforced by filtering the domains of the variables.

Choice of variables & values

VARIABLES

A. Terms?

DOMAINS

Legal combinations of for example 4 courses (but this is huge set of values).

B. Term Slots?

subdivide terms into slots e.g. 4 of them
(Fall 1,1) (Fall 1,2)
(Fall 1,3) (Fall 1,4)

Courses offered during that term

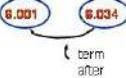
C. Courses?

Terms or term slots (Term slots allow expressing constraint on limited number of courses /term.)

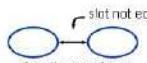
11b - Sept 00 - 15

Constraints

Use courses as variables and term slots as values.

Prerequisite \Rightarrow  For pairs of courses that must be ordered.

Courses offered only in some terms \Rightarrow Filter domain

Limit # courses \Rightarrow  for all pairs of vars. Use term-slots only once.

http://Sept 00 - 15

Slide 3.1.16

One constraint that must be represented is that the pre-requisites of a class must be taken before the actual class. This is easy to represent in this formulation. We introduce types of constraints called "term before" and "term after" which check that the values assigned to the variables, for example, 6.034 and 6.001, satisfy the correct ordering.

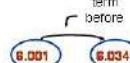
Note that the undirected links shown in prior constraint graphs are now split into two directed links, each with complementary constraints.

Slide 3.1.17

The constraint that some courses are only offered in some terms simply filters illegal term values from the domains of the variables.

Constraints

Use courses as variables and term slots as values.

Prerequisite \Rightarrow  For pairs of courses that must be ordered.

Courses offered only in some terms \Rightarrow Filter domain

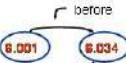
http://Sept 00 - 17

Slide 3.1.18

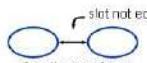
The limit on courses to be taken in a term argues for the use of term-slots as values rather than just terms. If we use term-slots, then the constraint is implicitly satisfied.

Constraints

Use courses as variables and term slots as values.

Prerequisite \Rightarrow  For pairs of courses that must be ordered.

Courses offered only in some terms \Rightarrow Filter domain

Limit # courses \Rightarrow  for all pairs of vars. Use term-slots only once.

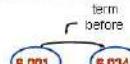
http://Sept 00 - 16

Slide 3.1.19

Avoiding time conflicts is also easily represented. If two courses occur at overlapping times then we place a constraint between those two courses. If they overlap in time every term that they are given, we can make sure that they are taken in different terms. If they overlap only on some terms, that can also be enforced by an appropriate constraint.

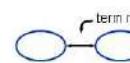
Constraints

Use courses as variables and term slots as values.

Prerequisite \Rightarrow  For pairs of courses that must be ordered.

Courses offered only in some terms \Rightarrow Filter domain

Limit # courses \Rightarrow  for all pairs of vars. Use term-slots only once.

Avoid time conflicts \Rightarrow  For pairs offered at same or overlapping times

http://Sept 00 - 19

6.034 Notes: Section 3.2

Slide 3.2.1

We now turn our attention to solving CSPs. We will see that the approaches to solving CSPs are some combination of constraint propagation and search. We will look at these in turn and then look at how they can be profitably combined.

Solving CSPs

Solving CSPs involves some combination of:

1. Constraint propagation, to eliminate values that could not be part of any solution
2. Search, to explore valid assignments

10 - Sept 00 - 1



Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

$$V_i \rightarrow V_j$$

Directed arc (V_i, V_j) is arc consistent if
 $\forall x \in D_i \exists y \in D_j$ such that (x,y) is allowed by the constraint on the arc

10 - Sept 00 - 2

Slide 3.2.2

The great success of Waltz's constraint propagation algorithm focused people's attention on CSPs. The basic idea in constraint propagation is to enforce what is known as "ARC CONSISTENCY", that is, if one looks at a directed arc in the constraint graph, say an arc from V_i to V_j , we say that this arc is consistent if for every value in the domain of V_i , there exists some value in the domain of V_j that will satisfy the constraint on the arc.

10 - Sept 00 - 1



Slide 3.2.3

Suppose there are some values in the domain at the tail of the constraint arc (for V_i) that do not have any consistent partner in the domain at the head of the arc (for V_j). We achieve arc consistency by dropping those values from D_i . Note, however, that if we change D_i , we now have to check to make sure that any other constraint arcs that have D_i at their head are still consistent. It is this phenomenon that accounts for the name "constraint propagation".

Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

$$V_i \rightarrow V_j$$

Directed arc (V_i, V_j) is arc consistent if
 $\forall x \in D_i \exists y \in D_j$ such that (x,y) is allowed by the constraint on the arc

We can achieve consistency on arc by deleting values from D_i (domain of variable at tail of constraint arc) that fail this condition.

10 - Sept 00 - 2



Constraint Propagation (aka Arc Consistency)

Arc consistency eliminates values from domain of variable that can never be part of a consistent solution.

$$V_i \rightarrow V_j$$

Directed arc (V_i, V_j) is arc consistent if $\forall x \in D_i \exists y \in D_j$ such that (x,y) is allowed by the constraint on the arc

We can achieve consistency on arc by deleting values from D_i (domain of variable at tail of constraint arc) that fail this condition.

Assume domains are size at most d and there are e binary constraints.

A simple algorithm for arc consistency is $O(ed^3)$ – note that just verifying arc consistency takes $O(d^2)$ for each arc.

16 Sept 00 4

Slide 3.2.4

What is the cost of this operation? In what follows we will reckon cost in terms of "arc tests": the number of times we have to check (evaluate) the constraint on an arc for a pair of values in the variable domains of that arc. Assuming that domains have at most d elements and that there are at most e binary constraints (arcs), then a simple constraint propagation algorithm takes $O(ed^3)$ arc tests in the worst case.

It is easy to see that checking for consistency of each arc for all the values in the corresponding domains takes $O(d^2)$ arc tests, since we have to look at all pairs of values in two domains. Going through and checking each arc once requires $O(ed^2)$ arc tests. But, we may have to go through and look at the arcs more than once as the deletions to a node's domain propagate. However, if we look at an arc only when one of its variable domains has changed (by deleting some entry), then no arc can require checking more than d times and we have the final cost of $O(ed^3)$ arc tests in the worst case.

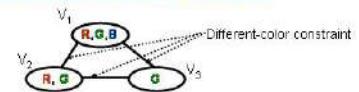
Slide 3.2.5

Let's look at a trivial example of graph coloring. We have three variables with the domains indicated. Each variable is constrained to have values different from its neighbors.

Constraint Propagation Example

Graph Coloring

Initial Domains are indicated



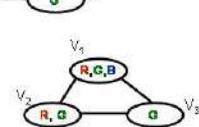
16 Sept 00 5

Constraint Propagation Example

Graph Coloring

Initial Domains are indicated

Arc examined	Value deleted



Each undirected constraint arc is really two directed constraint arcs, the effects shown above are from examining BOTH arcs.

16 Sept 00 6

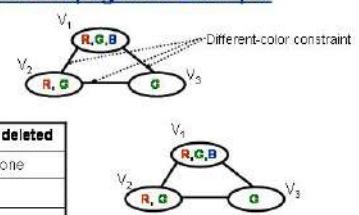
Slide 3.2.6

We will now simulate the process of constraint propagation. In the interest of space, we will deal in this example with undirected arcs, which are just a shorthand for the two directed arcs between the variables. Each step in the simulation involves examining one of these undirected arcs, seeing if the arc is consistent and, if not, deleting values from the domain of the appropriate variable.

Constraint Propagation Example

Graph Coloring

Initial Domains are indicated



Slide 3.2.7

We start with the $V_1 - V_2$ arc. Note that for every value in the domain of V_1 (R, G and B) there is some value in the domain of V_2 that it is consistent with (that is, it is different from). So, for R in V_1 there is a G in V_2 , for G in V_1 there is an R in V_2 and for B in V_1 there is either R and G in V_2 . Similarly, for each entry in V_2 there is a valid counterpart in V_1 . So, the arc is consistent and no changes are made.

Graph Coloring

Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none

16 Sept 00 7

Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(G)$
$V_2 - V_3$	
$V_2 - V_1$	
$V_3 - V_1$	
$V_3 - V_2$	

16 Sept 00 8

Slide 3.2.8

We move to $V_1 - V_3$. The situation here is different. While R and B in V_1 can co-exist with the G in V_3 , not so the G in V_1 . And, so, we remove the G from V_1 . Note that the arc in the other direction is consistent.

Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(G)$
$V_2 - V_3$	$V_2(G)$
$V_2 - V_1$	
$V_3 - V_1$	
$V_3 - V_2$	

16 Sept 00 9

Slide 3.2.9

Moving to $V_2 - V_3$, we note similarly that the G in V_2 has no valid counterpart in V_3 and so we drop it from V_2 's domain. Although we have now looked at all the arcs once, we need to keep going since we have changed the domains for V_1 and V_2 .

Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(G)$
$V_2 - V_3$	$V_2(G)$
$V_1 - V_2$	$V_1(R)$
$V_2 - V_1$	
$V_3 - V_1$	
$V_3 - V_2$	

16 Sept 00 10

Slide 3.2.10

Looking at $V_1 - V_2$ again we note that R in V_1 no longer has a valid counterpart in V_2 (since we have deleted G from V_2) and so we need to drop R from V_1 .

Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(G)$
$V_2 - V_3$	$V_2(G)$
$V_1 - V_2$	$V_1(R)$
$V_1 - V_3$	none
$V_2 - V_1$	
$V_3 - V_1$	
$V_3 - V_2$	

16 Sept 00 11

Slide 3.2.11

We test $V_1 - V_3$ and it is consistent.

Constraint Propagation Example

Graph Coloring
Initial Domains are indicated

Arc examined	Value deleted
V1 - V2	none
V1 - V3	V1(G)
V2 - V3	V3(G)
V1 - V2	V1(R)
V1 - V3	none
V2 - V3	none

http://Sept 00 ~ 12

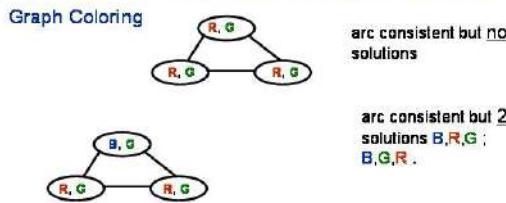
Slide 3.2.12

We test V₂-V₃ and it is consistent.

We are done; the graph is arc consistent. In general, we will need to make one pass through any arc whose head variable has changed until no further changes are observed before we can stop. If at any point some variable has an empty domain, the graph has no consistent solution.

Slide 3.2.13

Note that whereas arc consistency is required for there to be a solution for a CSP, having an arc-consistent solution is not sufficient to guarantee a unique solution or even any solution at all. For example, this first graph is arc-consistent but there are NO solutions for it (we need at least three colors and have only two).

But, arc consistency is not enough in general**Graph Coloring****But, arc consistency is not enough in general****Slide 3.2.14**

This next graph is also arc consistent but there are 2 distinct solutions: BRG and BGR.

But, arc consistency is not enough in general**Graph Coloring**

But, arc consistency is not enough in general

Graph Coloring

The first graph shows three nodes with domains {R, G} and edges between them. It is labeled "arc consistent but no solutions".

The second graph shows three nodes with domains {B, G} and edges between them. It is labeled "arc consistent but 2 solutions B,R,G; B,G,R".

The third graph shows three nodes with domains {B, G} and edges between them. It is labeled "arc consistent but 1 solution" and has a note "B, R not allowed".

Need to do search to find solutions (if any)

11b - Sept 20 - 15

Slide 3.2.16

In general, if there is more than one value in the domain of any of the variables, we do not know whether there is zero, one, or more than one answer that is globally consistent. We have to search for an answer to actually know for sure.

Slide 3.2.17

How does one search for solutions to a CSP problem? Any of the search methods we have studied is applicable. All we need to realize is that the space of assignments of values to variables can be viewed as a tree in which all the assignments of values to the first variable are descendants of the first node and all the assignments of values to the second variable form the descendants of those nodes and so forth.

The classic approach to searching such a tree is called "backtracking", which is just another name for depth-first search in this tree. Note, however, that we could use breadth-first search or any of the heuristic searches on this problem. The heuristic value could be used to either guide the search to termination or bias it to a desired solution based on preferences for certain assignments. Uniform-Cost and A* would make sense also if there were a non-uniform cost associated with a particular assignment of a value to a variable (note that this is another (better but more expensive) way of incorporating preferences).

However, you should observe that these CSP problems are different from the graph search problems we looked at before, in that we don't really care about the path to some state but just the final state itself.

Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

The tree starts with V_1 assignments. The first node has children R and G. The R node leads to V_2 assignments, which further lead to V_3 assignments. The G node leads directly to V_3 assignments. A note says "Inconsistent with $V_1=R$ ".

From the R node of V_2 , a branch to R leads to V_3 assignments, and a branch to G leads to a node where V_3 assignments are shown. A note says "Backup at inconsistent assignment".

From the G node of V_2 , branches lead to R and G, both leading to V_3 assignments.

11b - Sept 20 - 16

Slide 3.2.18

If we undertake a DFS in this tree, going left to right, we first explore assigning R to V_1 and then move to V_2 and consider assigning R to it. However, for any assignment, we need to check any constraints involving previous assignments in the tree. We note that $V_2=R$ is inconsistent with $V_1=R$ and so that assignment fails and we have to backup to find an alternative assignment for the most recently assigned variable.

Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

The tree starts with V_1 assignments. The first node has children R and G. The R node leads to V_2 assignments, which further lead to V_3 assignments. The G node leads directly to V_3 assignments. A note says "Inconsistent with $V_1=R$ ".

From the R node of V_2 , a branch to R is crossed out with a red X, and a branch to G leads to V_3 assignments. A note says "Backup at inconsistent assignment".

From the G node of V_2 , branches lead to R and G, both leading to V_3 assignments.

11b - Sept 20 - 17

Slide 3.2.19

So, we consider assigning $V_2=G$, which is consistent with the value for V_1 . We then move to $V_3=R$. Since we have a constraint between V_1 and V_3 , we have to check for consistency and find it is not consistent, and so we backup to consider another value for V_3 .

Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).

The tree starts with V_1 assignments. The first node has children R and G. The R node leads to V_2 assignments, which further lead to V_3 assignments. The G node leads directly to V_3 assignments. A note says "Inconsistent with $V_1=R$ ".

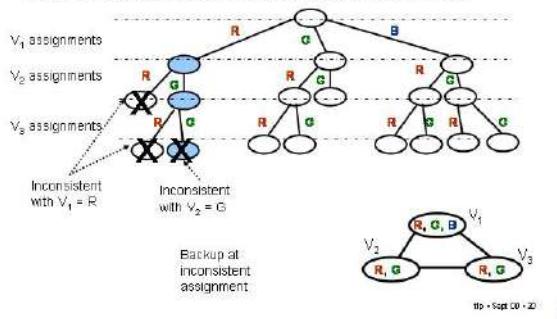
From the R node of V_2 , a branch to R is crossed out with a red X, and a branch to G leads to V_3 assignments. A note says "Backup at inconsistent assignment".

From the G node of V_2 , branches lead to R and G, both leading to V_3 assignments.

11b - Sept 20 - 18

Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search.
Simplest approach is pure backtracking (depth-first search).



Slide 3.2.20

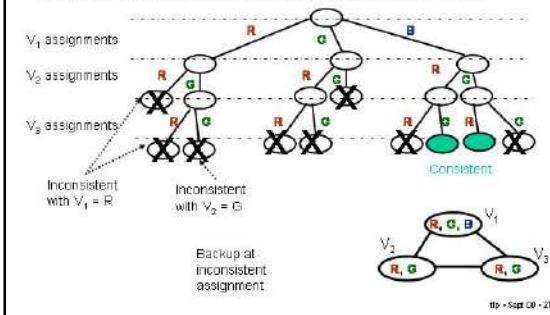
But $V_3=G$ is inconsistent with $V_2=G$, and so we have to backup. But there are no more pending values for V_3 or for V_2 and so we fail back to the V_1 level.

Slide 3.2.21

The process continues in that fashion until we find a solution. If we continue past the first success, we can find all the solutions for the problem (two in this case).

Searching for solutions – backtracking (BT)

When we have too many values in domain (and/or constraints are weak) arc consistency doesn't do much, so we need to search. Simplest approach is pure backtracking (depth-first search).



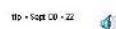
Slide 3.2.22

We can use some form of backtracking search to solve CSP independent of any form of constraint propagation. However, it is natural to consider combining them. So, for example, during a backtracking search where we have a partial assignment, where a subset of all the variables each has unique values assigned, we could then propagate these assignments throughout the constraint graph to obtain reduced domains for the remaining variables. This is, in general, advantageous since it decreases the effective branching factor of the search tree.

Combine Backtracking & Constraint Propagation

A node in BT tree is partial assignment in which the domain of each variable has been set (tentatively) to singleton set.

Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.



Slide 3.2.23

But, how much propagation should we do? Is it worth doing the full arc-consistency propagation we described earlier?

Combine Backtracking & Constraint Propagation

A node in BT tree is partial assignment in which the domain of each variable has been set (tentatively) to singleton set.

Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.

Question: How much propagation to do?



Combine Backtracking & Constraint Propagation

A node in BT tree is **partial assignment** in which the domain of each variable has been set (tentatively) to singleton set.

Use constraint propagation (arc-consistency) to propagate the effect of this tentative assignment, i.e., eliminate values inconsistent with current values.

Question: How much propagation to do?

Answer: Not much, just local propagation from domains with unique assignments, which is called **forward checking** (FC). This conclusion is not necessarily obvious, but it generally holds in practice.

11b - Sept 00 - 21

Slide 3.2.24

The answer is USUALLY no. It is generally sufficient to only propagate to the immediate neighbors of variables that have unique values (the ones assigned earlier in the search). That is, we eliminate from consideration any values for future variables that are inconsistent with the values assigned to past variables. This process is known as **forward checking** (FC) because one checks values for future variables (forward in time), as opposed to standard backtracking which checks value of past variables (backwards in time, hence back-checking).

When the domains at either end of a constraint arc each have multiple legal values, odds are that the constraint is satisfied, and so checking the constraint is usually a waste of time. This conclusion suggests that forward checking is usually as much propagation as we want to do. This is, of course, only a rule of thumb.

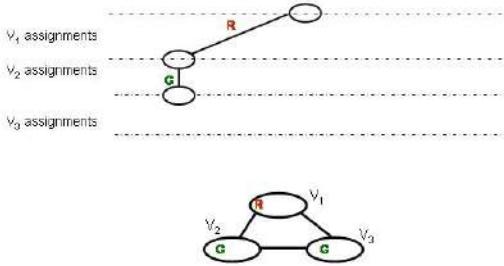
Slide 3.2.25

Let's step through a search that uses a combination of backtracking with forward checking. We start by considering an assignment of $V_1=R$.

11b - Sept 00 - 21

Backtracking with Forward Checking (BT-FC)

When examining assignment $V_1=d_{i_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



11b - Sept 00 - 25

Slide 3.2.26

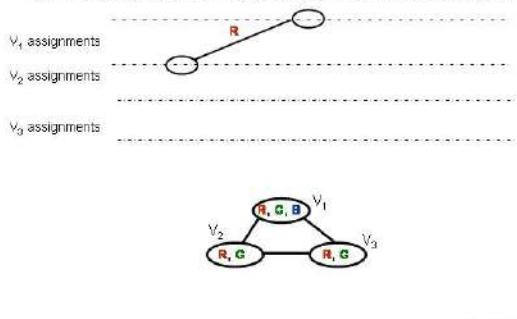
We then propagate to the neighbors of V_1 in the constraint graph and eliminate any values that are inconsistent with that assignment, namely the value R. That leaves us with the value G in the domains of V_2 and V_3 . So, we make the assignment $V_2=G$ and propagate.

Slide 3.2.27

But, when we propagate to V_3 we see that there are no remaining valid values and so we have found an inconsistency. We fail and backup. Note that we have failed much earlier than with simple backtracking, thus saving a substantial amount of work.

Backtracking with Forward Checking (BT-FC)

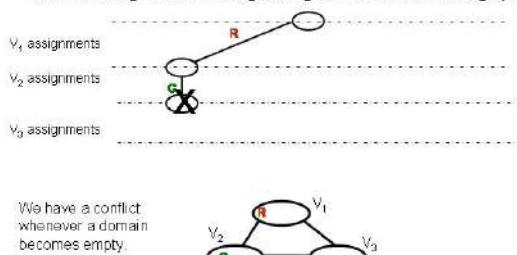
When examining assignment $V_1=d_{i_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



11b - Sept 00 - 26

Backtracking with Forward Checking (BT-FC)

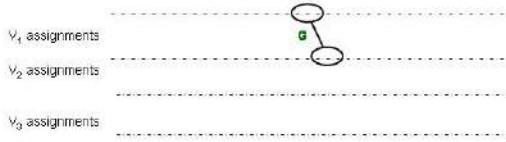
When examining assignment $V_1=d_{i_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



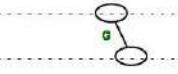
11b - Sept 00 - 27

Backtracking with Forward Checking (BT-FC)

When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



When backing up, need to restore domain values, since deletions were done to reach consistency with tentative assignments considered during search.



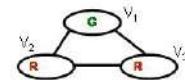
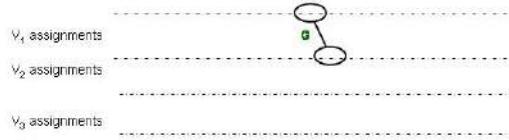
11p - Sept 00 - 26

Slide 3.2.28

We now consider $V_1=G$ and propagate.

Backtracking with Forward Checking (BT-FC)

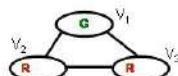
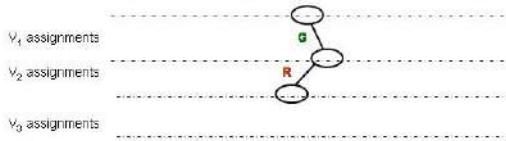
When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



11p - Sept 00 - 27

Backtracking with Forward Checking (BT-FC)

When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



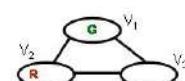
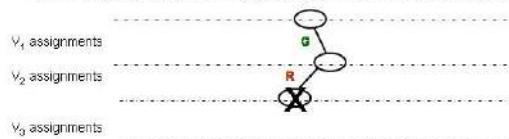
11p - Sept 00 - 30

Slide 3.2.30

We now consider $V_2=R$ and propagate.

Backtracking with Forward Checking (BT-FC)

When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



11p - Sept 00 - 31

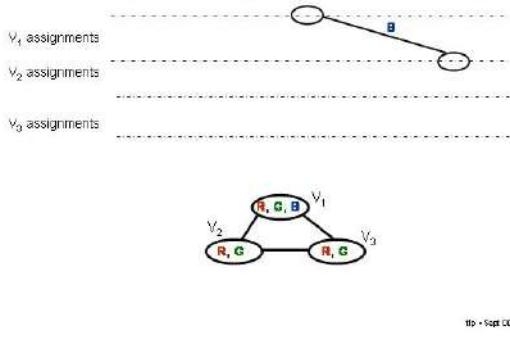
Slide 3.2.31

The domain of V_3 is empty, so we fail and backup.



Backtracking with Forward Checking (BT-FC)

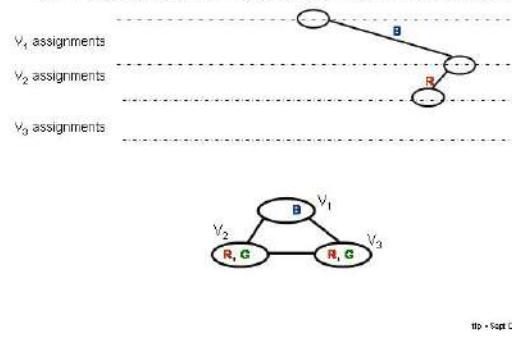
When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

**Slide 3.2.32**

So, we move to consider $V_1=B$ and propagate.

Backtracking with Forward Checking (BT-FC)

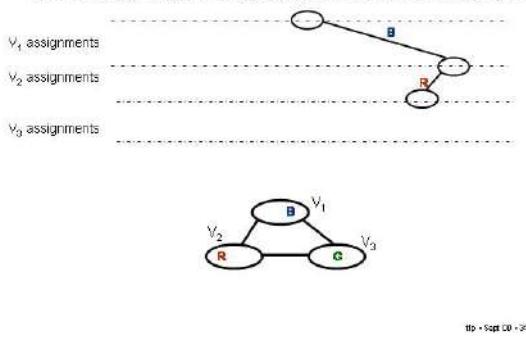
When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

**Slide 3.2.33**

This propagation does not delete any values. We pick $V_2=R$ and propagate.

Backtracking with Forward Checking (BT-FC)

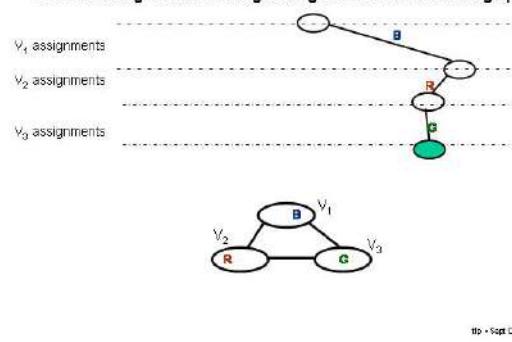
When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

**Slide 3.2.34**

This removes the R values in the domains of V_1 and V_3 .

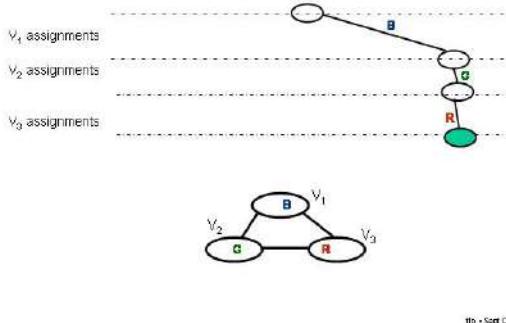
Backtracking with Forward Checking (BT-FC)

When examining assignment $V_1=d_{k_1}$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.



Backtracking with Forward Checking (BT-FC)

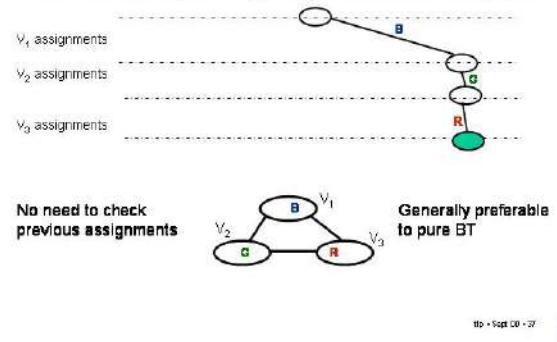
When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

**Slide 3.2.36**

We can continue the process to find the other consistent solution.

Backtracking with Forward Checking (BT-FC)

When examining assignment $V_i = d_k$, remove any values inconsistent with that assignment from neighboring domains in constraint graph.

**Slide 3.2.37**

Note that when doing forward checking there is no need to check new assignments against previous assignments. Any potential inconsistencies have been removed by the propagation. BT-FC is usually preferable to plain BT because it eliminates from consideration inconsistent assignments once and for all rather than discovering the inconsistency over and over again in different parts of the tree. For example, in pure BT, an assignment for V_3 that is inconsistent with a value of V_1 would be "discovered" independently for every value of V_2 . Whereas FC would delete it from the domain of V_3 right away.

6.034 Notes: Section 3.3**Slide 3.3.1**

We have been assuming that the order of the variables is given by some arbitrary ordering. However, the order of the variables (and values) can have a substantial effect on the cost of finding the answer. Consider, for example, the course scheduling problem using courses given in the order that they should ultimately be taken and assume that the term values are ordered as well. Then a depth first search will tend to find the answer very quickly.

Of course, we generally don't know the answer to start off with, but there are more rational ways of ordering the variables than alphabetical or numerical order. For example, we could order the variables before starting by how many constraints they have. But, we can do even better by dynamically re-ordering variables based on information available during a search.

BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g. random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

- **Most constrained variable**
when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)

Slide 3.3.2

For example, assume we are doing backtracking with forward checking. At any point, we know the size of the domain of each variable. We can order the variables below that point in the search tree so that the most constrained variable (smallest valid domain) is next. This will have the effect of reducing the average branching factor in the tree and also cause failures to happen sooner.

Slide 3.3.3

Furthermore, we can count for each value of the variable the impact on the domains of its neighbors, for example the minimum of the resulting domains after propagation. The value with the largest minimum resulting domain size (or average value or sum) would be one that least constrains the remaining choices and is least likely to lead to failure.

Of course, value ordering is only worth doing if we are looking for a single answer to the problem. If we want all answers, then all values will have to be tried eventually.

BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

- **Most constrained variable**
when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)
- **Least constraining value**
choose value that rules out the fewest values from neighboring domains

BT-FC with dynamic ordering

Traditional backtracking uses fixed ordering of variables & values, e.g., random order or place variables with many constraints first.

You can usually do better by choosing an order dynamically as the search proceeds.

- **Most constrained variable**
when doing forward-checking, pick variable with fewest legal values to assign next (minimizes branching factor)
- **Least constraining value**
choose value that rules out the fewest values from neighboring domains

E.g. this combination improves feasible n-queens performance from about n = 30 with just FC to about n = 1000 with FC & ordering.

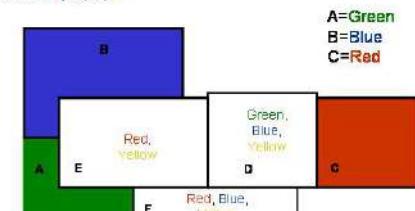
Slide 3.3.4

This combination of variable and value ordering can have dramatic impact on some problems.

Slide 3.3.5

This example of the 4-color map-coloring problem illustrates a simple situation for variable and value ordering. Here, A is colored Green, B is colored Blue and C is colored Red. What country should we color next, D or E or F?

Colors: R, G, B, Y



Which country should we color next →

What color should we pick for it? →

Colors: R, G, B, Y

A=Green
B=Blue
C=Red

Which country should we color next → E most-constrained variable (smallest domain)
What color should we pick for it? →

1b • Somp 02 - 8

Slide 3.3.6

Well, E is more constrained (has fewer) legal values so we should try it next. Which of E's values should we try next?

Colors: R, G, B, Y

A=Green
B=Blue
C=Red

Which country should we color next → E most-constrained variable (smallest domain)
What color should we pick for it? → RED least-constraining value (eliminates fewest values from neighboring domains)

1b • Somp 02 - 7

Incremental Repair (min-conflict heuristic)

1. Initialize a candidate solution using "greedy" heuristic – get solution "near" correct one.
2. Select a variable in conflict and assign it a value that minimizes the number of conflicts (break ties randomly).

Can use this heuristic as part of systematic backtracker that uses heuristics to do value ordering or in a local hill-climber (without backup).

Performance on n-queens (with good initial guesses)

1b • Somp 02 - 9

Slide 3.3.8

All of the methods for solving CSPs that we have discussed so far are systematic (guaranteed searches). More recently, researchers have had surprising success with methods that are not systematic (they are randomized) and do not involve backup.

The basic idea is to do incremental repair of a nearly correct assignment. Imagine we had some heuristic that could give us a "good" answer to any of the problems. By "good" we mean one with relatively few constraint violations. In fact, this could even be a randomly chosen solution.

Then, we could take the following approach. Identify a random variable involved in some conflict. Pick a new value for that variable that minimizes the number of resulting conflicts. Repeat.

This is a type of local "greedy" search algorithm.

There are variants of this strategy that use this heuristic to do value ordering within a backtracking search. Remarkably, this type of ordering (in connection with a good initial guess) leads to remarkable behavior for benchmark problems. Notably, the systematic versions of this strategy can solve the million-queen problem in minutes. After this, people decided N-queens was not interesting...

Slide 3.3.9

The pure "greedy" hill-climber can readily fail on any problem (by finding a local minimum where any change to a single variable causes the number of conflicts to increase). We'll look at this a bit in the problem set.

There are several ways of trying to deal with local minima. One is to introduce weights on the violated constraints. A simpler one is to re-start the search with another random initial state. This is the approach taken by GSAT, a randomized search process that solves SAT problems using a similar approach to the one described here.

GSAT's performance is nothing short of remarkable. It can solve SAT problems of mind-boggling complexity. It has forced a complete reconsideration of what it means when we say that a problem is "hard". It turns out that for SAT, almost any randomly chosen problem is "easy". There are really hard SAT problems but they are difficult to find. This is an area of active study.

Min-conflict heuristic

The pure hill climber (without backtracking) can get stuck in local minima. Can add random moves to attempt getting out of minima – generally quite effective. Can also use weights on violated constraints & increase weight every cycle it remains violated.

GSAT

Randomized hill climber used to solve SAT problems. One of the most effective methods ever found for this problem

tp - Spring 02 - 10

**GSAT as Heuristic Search**

- State space: Space of all full assignments to variables
- Initial state: A random full assignment
- Goal state: A satisfying assignment
- Actions: Flip value of one variable in current assignment
- Heuristic: The number of satisfied clauses (constraints); we want to maximize this. Alternatively, minimize the number of unsatisfied clauses (constraints).

tp - Spring 02 - 10

Slide 3.3.10

GSAT can be framed as a heuristic search strategy. Its state space is the space of all full assignments to the variables. The initial state is a random assignment, while the goal state is any assignment that satisfies the formula. The actions available to GSAT are simply to flip one variable in the assignment from true to false or vice-versa. The heuristic value used for the search, which GSAT tries to maximize, is the number of satisfied clauses (constraints). Note that this is equivalent to minimizing the number of conflicts, that is, violated constraints.

Slide 3.3.11

Here we see the GSAT algorithm, which is very simple in sketch. The critical implementation challenge is that of finding quickly the variable whose flip maximizes the score. Note that there are two user-specified variables: the number of times the outer loop is executed (MAXTRIES) and the number of times the inner loop is executed (MAXFLIPS). These parameters guard against local minima in the search, simply by starting with a new, randomly chosen assignment and trying a different sequence of flips. As we have mentioned, this works surprisingly well.

GSAT(F)

- For i=1 to Maxtries
 - Select a complete random assignment A
 - Score = number of satisfied clauses
 - For j=1 to Maxflips
 - If (A satisfies all clauses in F) return A
 - Else flip a variable that maximizes score
 - Flip a randomly chosen variable if no variable flip increases the score.

tp - Spring 02 - 11

**WALKSAT(F)**

- For i=1 to Maxtries
 - Select a complete random assignment A
 - Score = number of satisfied clauses
 - For j=1 to Maxflips
 - If (A satisfies all clauses in F) return A
 - Else
 - With probability p / GSAT */
 - » flip a variable that maximizes score
 - » Flip a randomly chosen variable if no variable flip increases the score.
 - With probability 1-p / Random Walk */
 - » Pick a random unsatisfied clause C
 - » Flip a randomly chosen variable in C

tp - Spring 02 - 12

Slide 3.3.12

An even more effective strategy turns out to add even more randomness. WALKSAT basically performs the GSAT algorithm some percentage of the time and the rest of the time it does a random walk in the space of assignments by randomly flipping variables in unsatisfied clauses (constraints).

It's a bit depressing to think that such simple randomized strategies can be so much more effective than clever deterministic strategies. There are signs at present that some of the clever deterministic strategies are becoming competitive or superior to the randomized ones. The story is not over.

6.034 Notes: Section 3.4

Slide 3.4.1

In this section, we will look at some of the basic approaches for building programs that play two-person games such as tic-tac-toe, checkers and chess.

Much of the work in this area has been motivated by playing chess, which has always been known as a "thinking person's game". The history of computer chess goes way back. Claude Shannon, the father of information theory, originated many of the ideas in a 1949 paper. Shortly after, Alan Turing did a hand simulation of a program to play checkers, based on some of these ideas. The first programs to play real chess didn't arrive until almost ten years later, and it wasn't until Greenblatt's MacHack 6 that a computer chess program defeated a good player. Slow and steady progress eventually led to the defeat of reigning world champion Garry Kasparov against IBM's Deep Blue in May 1997.

Board Games & Search	
Move generation	1949 Shannon paper
Static Evaluation	1951 Turing paper
Min Max	1958 Bernstein program
Alpha Beta	55-60 Simon-Newell program
Practical matters	($\alpha\beta$ McCarthy?)
	61 Soviet program
	66 – 67 MacHack 6 (MIT AI)
	70's NW Chess 4.5
	80's Cray Blitz
	90's Belle, Hitech, Deep Thought, Deep Blue

dp - Spring03 - 1



Game Tree Search

- Initial state: initial board position and player
- Operators: one for each legal move
- Goal states: winning board positions
- Scoring function: assigns numeric value to states
- Game tree: encodes all possible games
- We are not looking for a path, only the next move to make (that hopefully leads to a winning position)
- Our best move depends on what the other player does

dp - Spring03 - 2



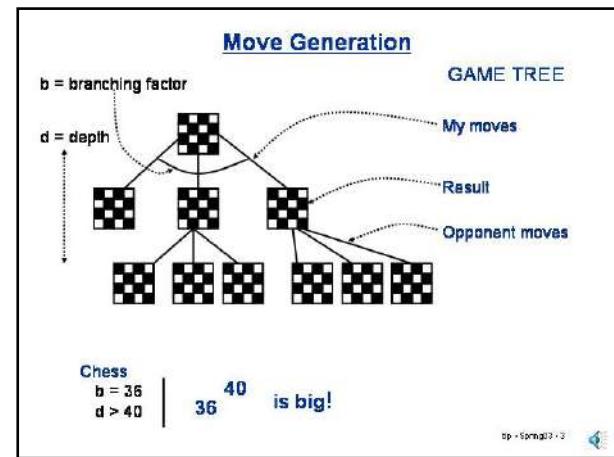
Slide 3.4.2

Game playing programs are another application of search. The states are the board positions (and the player whose turn it is to move). The operators are the legal moves. The goal states are the winning positions. A scoring function assigns values to states and also serves as a kind of heuristic function. The game tree (defined by the states and operators) is like the search tree in a typical search and it encodes all possible games.

There are a few key differences, however. For one thing, we are not looking for a path through the game tree, since that is going to depend on what moves the opponent makes. All we can do is choose the best move to make next.

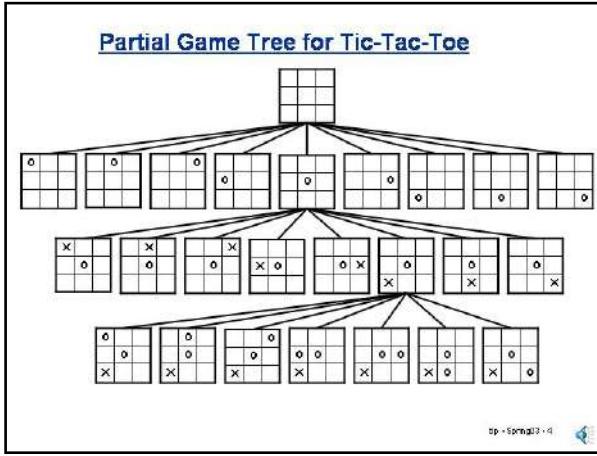
Slide 3.4.3

Let's look at the game tree in more detail. Some board position represents the initial state and it's now our turn. We generate the children of this position by making all of the legal moves available to us. Then, we consider the moves that our opponent can make to generate the descendants of each of these positions, etc. Note that these trees are enormous and cannot be explicitly represented in their entirety for any complex game.



dp - Spring03 - 3

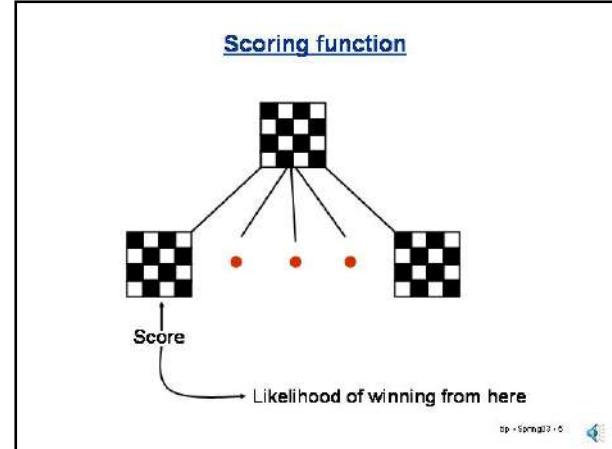


**Slide 3.4.4**

Here's a little piece of the game tree for Tic-Tac-Toe, starting from an empty board. Note that even for this trivial game, the search tree is quite big.

Slide 3.4.5

A crucial component of any game playing program is the scoring function. This function assigns a numerical value to a board position. We can think of this value as capturing the likelihood of winning from that position. Since in these games one person's win is another's person loss, we will use the same scoring function for both players, simply negating the values to represent the opponent's scores.

**Static Evaluation**

$S = c_1 \times$	material
$+ c_2 \times$	pawn structure
$+ c_3 \times$	mobility
$+ c_4 \times$	king safety
$+ c_5 \times$	center control
$+ \dots$	

P	1
K	3
B	3.5
R	5
Q	9

Too weak to predict ultimate success

Slide 3.4.6

A typical scoring function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position. Each feature is also a number that measures some characteristic of the position. One that is easy to see is "material", that is, some measure of which pieces one has on the board. A typical weighting for each type of chess piece is shown here. Other types of features try to encode something about the distribution of the pieces on the board.

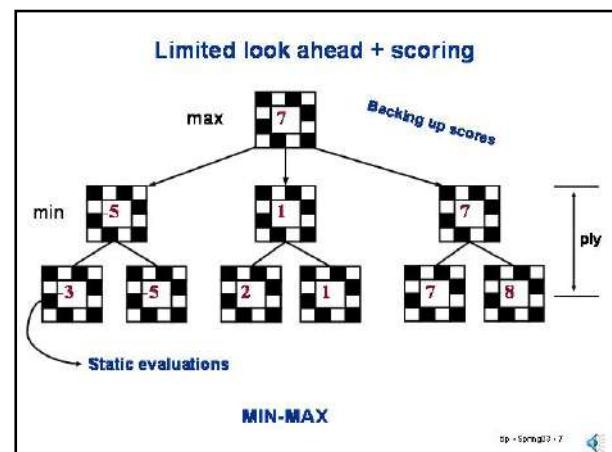
In some sense, if we had a perfect evaluation function, we could simply play chess by evaluating the positions produced by each of our legal moves and picking the one with the highest score. In principle, such a function exists, but no one knows how to write it or compute it directly.

Slide 3.4.7

The key idea that underlies game playing programs (presented in Shannon's 1949 paper) is that of limited look-ahead combined with the Min-Max algorithm.

Let's imagine that we are going to look ahead in the game-tree to a depth of 2 (or 2 ply as it is called in the literature on game playing). We can use our scoring function to see what the values are at the leaves of this tree. These are called the "static evaluations". What we want is to compute a value for each of the nodes above this one in the tree by "backing up" these static evaluations in the tree.

The player who is building the tree is trying to maximize their score. However, we assume that the opponent (who values board positions using the same static evaluation function) is trying to minimize the score (or think of this as maximizing the negative of the score). So, each layer of the tree can be classified into either a maximizing layer or a minimizing layer. In our example, the layer right above the leaves is a minimizing layer, so we assign to each node in that layer the minimum score of any of its children. At the next layer up, we're maximizing so we pick the maximum of the scores available to us, that is, 7. So, this analysis tells us that we should pick the move that gives us the best guaranteed score, independent of what our opponent does. This is the MIN-MAX algorithm.



Min-Max

```

// initial call is MAX-VALUE(state,MAX-DEPTH)

function MAX-VALUE (state, depth)
  if (depth == 0) then return EVAL. (state)
  v = -∞
  for each s in SUCCESSORS (state) do
    v = MAX (v, MIN-VALUE (s, depth-1))
  end
  return v

function MIN-VALUE (state, depth)
  if (depth == 0) then return EVAL. (state)
  v = ∞
  for each s in SUCCESSORS (state) do
    v = MIN (v, MAX-VALUE (s, depth-1))
  end
  return v

```

tip → Spring03 → 8

Slide 3.4.8

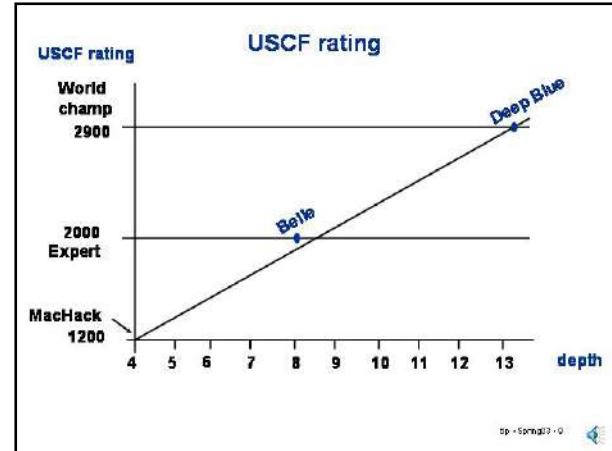
Here is pseudo-code that implements Min-Max. As you can see, it is a simple recursive alternation of maximization and minimization at each layer. We assume that we count the depth value down from the max depth so that when we reach a depth of 0, we apply our static evaluation to the board.

Slide 3.4.9

The key idea is that the more lookahead we can do, that is, the deeper in the tree we can look, the better our evaluation of a position will be, even with a simple evaluation function. In some sense, if we could look all the way to the end of the game, all we would need is an evaluation function that was 1 when we won and -1 when the opponent won.

The truly remarkable thing is how well this idea works. If you plot how well computer programs can search chess game trees versus their ranking, we see a graph that looks something like this. The earliest serious chess program (MacHack6), which had a ranking of 1200, searched on average to a depth of 4. Belle, which was one of the first hardware-assisted chess programs doubled the depth and gained about 800 points in ranking. Deep Blue, which searched to an average depth of about 13 beat the world champion with a ranking of about 2900.

At some level, this is a depressing picture, since it seems to suggest that brute-force search is all that matters.

**Deep Blue**

32 SP2 processors
each with **8 dedicated chess processors**
= **256 CP**

50 – 100 billion moves in 3 min
13-30 ply search.

tip → Spring03 → 10

Slide 3.4.10

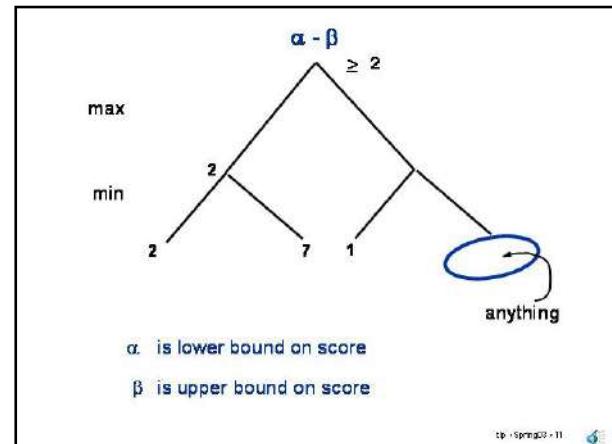
And Deep Blue is brute indeed... It had 256 specialized chess processors coupled into a 32 node supercomputer. It examined around 30 billion moves per minute. The typical search depth was 13-ply, but in some dynamic situations it could go as deep as 30.

Slide 3.4.11

There's one other idea that has played a crucial role in the development of computer game-playing programs. It is really only an optimization of Min-Max search, but it is such a powerful and important optimization that it deserves to be understood in detail. The technique is called alpha-beta pruning, from the Greek letters traditionally used to represent the lower and upper bound on the score.

Here's an example that illustrates the key idea. Suppose that we have evaluated the sub-tree on the left (whose leaves have values 2 and 7). Since this is a minimizing level, we choose the value 2. So, the maximizing player at the top of the tree knows at this point that he can guarantee a score of at least 2 by choosing the move on the left.

Now, we proceed to look at the subtree on the right. Once we look at the leftmost leaf of that subtree and see a 1, we know that if the maximizing player makes the move to the right then the minimizing player can force him into a position that is worth no more than 1. In fact, it might be much worse. The next leaf we look at might bring an even nastier surprise, but it doesn't matter what it is: we already know that this move is worse than the one to the left, so why bother looking any further? In fact, it may be that this unknown position is a great one for the maximizer, but then the minimizer would never choose it. So, no matter what happens at that leaf, the maximizer's choice will not be



tip → Spring03 → 11

affected.

 $\alpha - \beta$

```

//  $\alpha$  = best score for MAX,  $\beta$  = best score for MIN
// initial call is MAX-VALUE(state, - $\infty$ ,  $\infty$ , MAX-DEPTH)

function MAX-VALUE (state,  $\alpha$ ,  $\beta$ , depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
         $\alpha$  = MAX ( $\alpha$ , MIN-VALUE (s,  $\alpha$ ,  $\beta$ , depth-1))
        if  $\alpha \geq \beta$  then return  $\alpha$  // cutoff
    end
    return  $\alpha$ 

function MIN-VALUE (state,  $\alpha$ ,  $\beta$ , depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
         $\beta$  = MIN ( $\beta$ , MAX-VALUE (s,  $\alpha$ ,  $\beta$ , depth-1))
        if  $\beta \leq \alpha$  then return  $\beta$  // cutoff
    end
    return  $\beta$ 

```

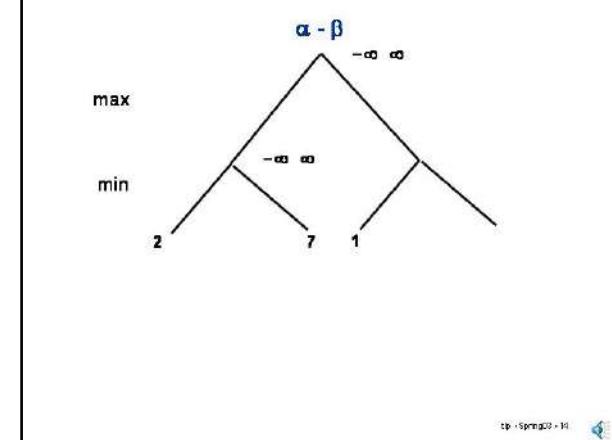
tp : Spring02 : 12

Slide 3.4.12

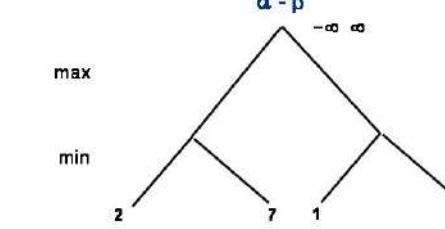
Here's some pseudo-code that captures this idea. We start out with the range of possible scores (as defined by alpha and beta) going from minus infinity to plus infinity. Alpha represents the lower bound and beta the upper bound. We call Max-Value with the current board state. If we are at a leaf, we return the static value. Otherwise, we look at each of the successors of this state (by applying the legal move function) and for each successor, we call the minimizer (Min-Value) and we keep track of the maximum value returned in alpha. If the value of alpha (the lower bound on the score) ever gets to be greater or equal to beta (the upper bound) then we know that we don't need to keep looking - this is called a cutoff - and we return alpha immediately. Otherwise we return alpha at the end of the loop. The Minimizer is completely symmetric.

Slide 3.4.13

Lets look at this program in operation on our previous example. We start with an initial call to Max-Value with the initial infinite values of alpha and beta, meaning that we know nothing about what the score is going to be.



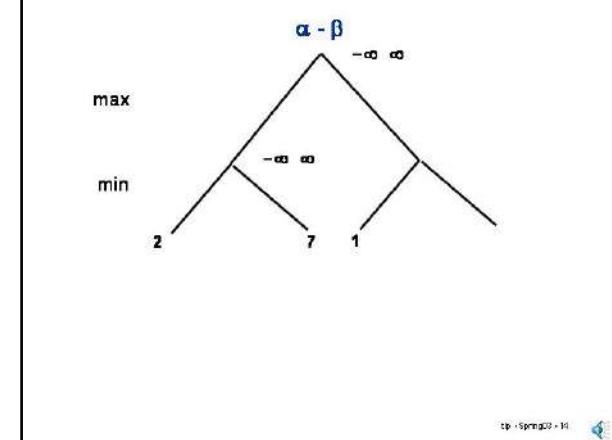
tp : Spring02 : 13

 $\alpha - \beta$ 

tp : Spring02 : 13

Slide 3.4.14

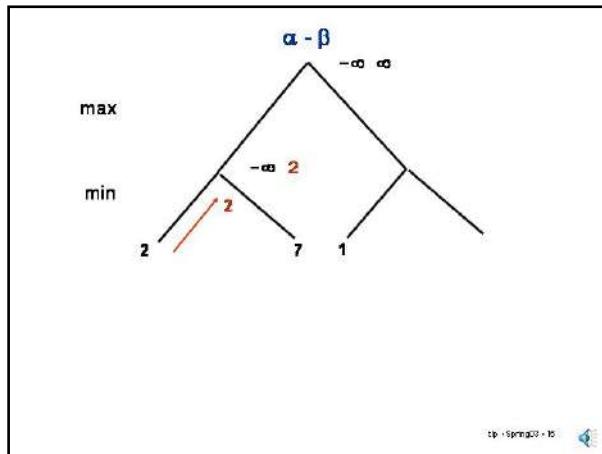
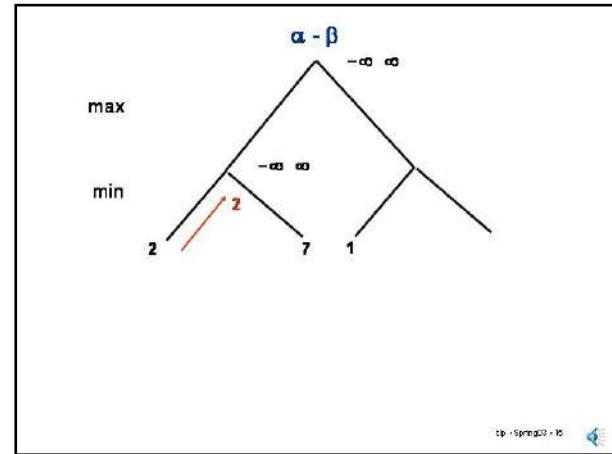
Max-Value now calls Min-Value on the left successor with the same values of alpha and beta. Min-Value now calls Max-Value on its leftmost successor.



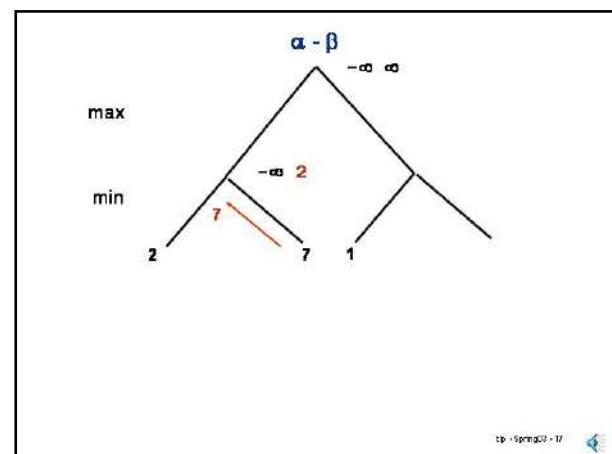
tp : Spring02 : 14

Slide 3.4.15

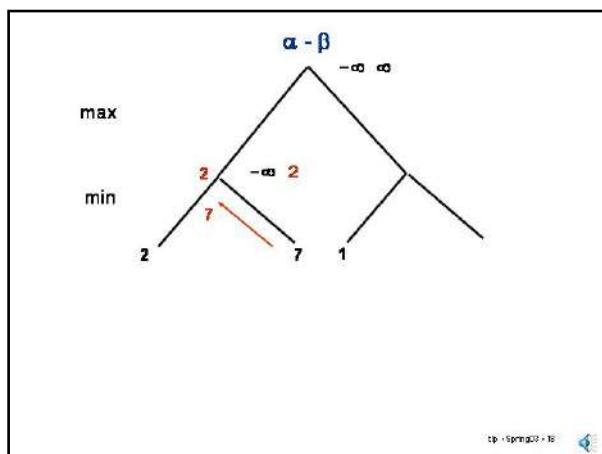
Max-Value is at the leftmost leaf, whose static value is 2 and so it returns that.

**Slide 3.4.16**

This first value, since it is less than infinity, becomes the new value of beta in Min-Value.

**Slide 3.4.17**

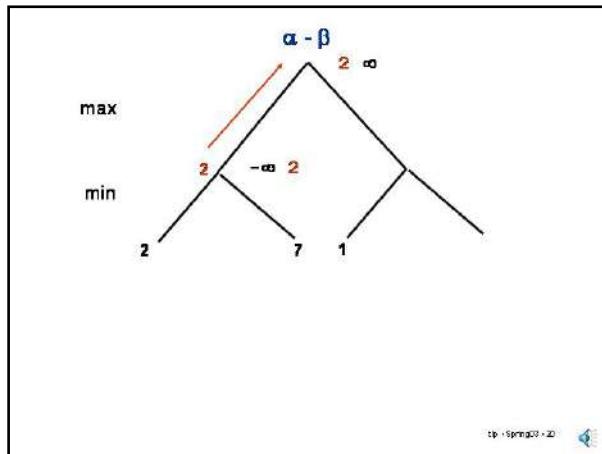
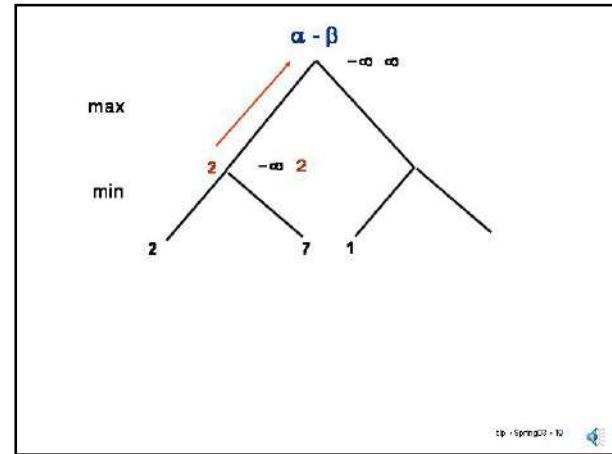
So, now we call Max-Value with the next successor, which is also a leaf whose value is 7.

**Slide 3.4.18**

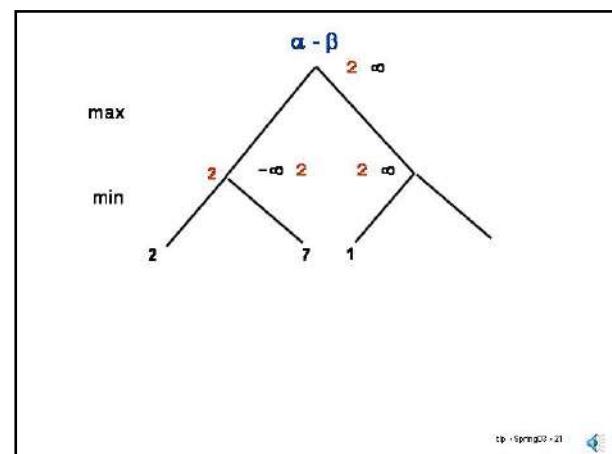
7 is not less than 2 and so the final value of beta is 2 for this node.

Slide 3.4.19

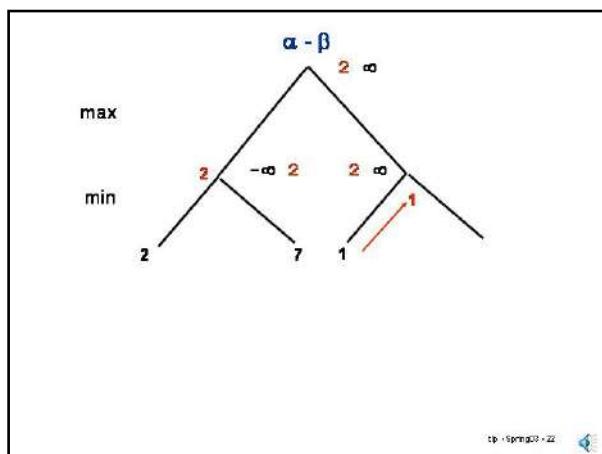
Min-Value now returns this value to its caller.

**Slide 3.4.20**

The calling Max-Value now sets alpha to this value, since it is bigger than minus infinity. Note that the range of [alpha beta] says that the score will be greater or equal to 2 (and less than infinity).

**Slide 3.4.21**

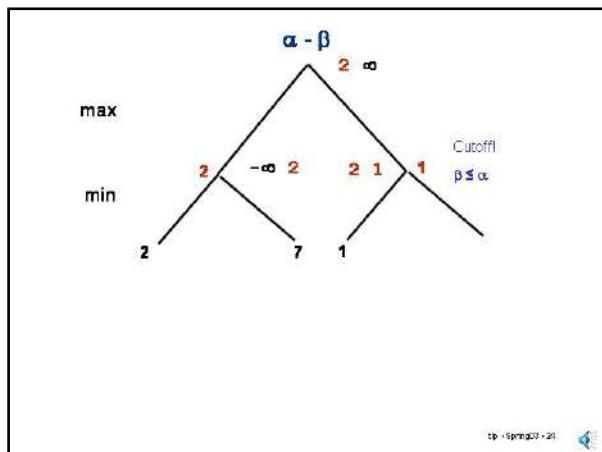
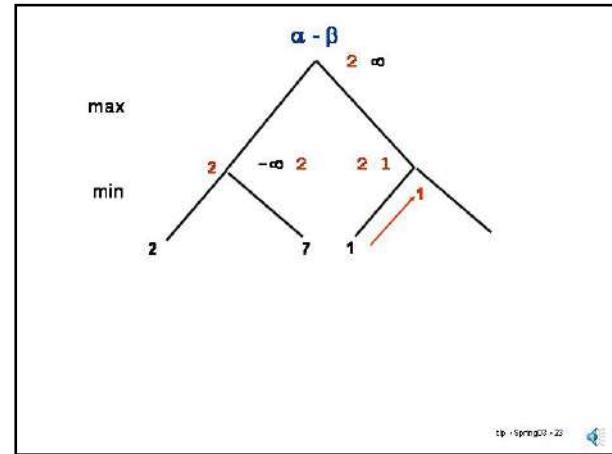
Max-Value now calls Min-Value with the updated range of [alpha beta].

**Slide 3.4.22**

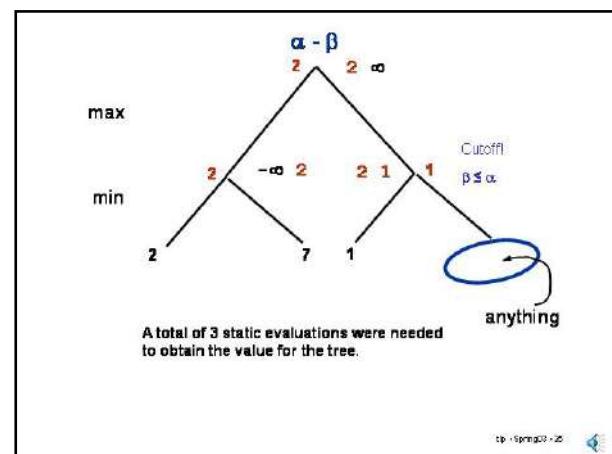
Min-Value calls Max-Value on the left leaf and it returns a value of 1.

Slide 3.4.23

This is used to update beta in Min-Value, since it is less than infinity. Note that at this point we have a range where alpha (2) is greater than beta (1).

**Slide 3.4.24**

This situation signals a cutoff in Min-Value and it returns beta (1), without looking at the right leaf.

 **$\alpha - \beta$ (NegaMax form)**

```
// α = best score for MAX, β = best score for MIN
// initial call is Alpha-Beta(state, -∞, ∞, MAX-DEPTH)

function Alpha-Beta (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX(α, -Alpha-Beta (s, -β, -α, depth-1))
        if α ≥ β then return α // cutoff
    end
    return α.
```

Slide 3.4.26

We can write alpha-beta in a more compact form that captures the symmetry between the Max-Value and Min-Value procedures. This is sometimes called the NegaMax form (instead of the Min-Max form). Basically, this exploits the idea that minimizing is the same as maximizing the negatives of the scores.

Slide 3.4.27

There are a couple of key points to remember about alpha-beta pruning. It is guaranteed to return exactly the same value as the Min-Max algorithm. It is a pure optimization without any approximations or tradeoffs.

In a perfectly ordered tree, with the best moves on the left, alpha beta reduces the cost of the search from order b^d to order $b^{(d/2)}$, that is, we can search twice as deep! We already saw the enormous impact of deeper search on performance. So, this one simple algorithm can almost double the search depth.

Now, this analysis is optimistic, since if we could order moves perfectly, we would not need alpha-beta. But, in practice, performance is close to the optimistic limit.

 $\alpha - \beta$

1. Guaranteed same value as Max-Min
2. In a perfectly ordered tree, expected work is $O(b^{d/2})$, vs $O(b^d)$ for Max-Min, so can search twice as deep with the same effort!
3. With good move ordering, the actual running time is close to the optimistic estimate.

tip : Spring03 - 27

Game Program

	Time
1. Move generator (ordered moves)	50%
2. Static evaluation	40%
3. Search control	10%

openings > databases

[all in place by late 60's.]

tip : Spring03 - 28

Slide 3.4.28

If one looks at the time spent by a typical game program, about half the time goes into generating the legal moves ordered (heuristically) in such a way to take maximal advantage of alpha-beta. Most of the remaining time is spent evaluating leaves. Only about 10% is spent on the actual search.

We should note that, in practice, chess programs typically play the first few moves and also complex end games by looking up moves in a database.

The other thing to note is that all these ideas were in place in MacHack6 in the late 60's. Much of the increased performance has come from increased computer power. The rest of the improvements come from a few other ideas that we'll look at later. First, let's look a bit more of two components that account for the bulk of the time.

Move Generator

1. Legal moves
2. Ordered by
 1. Most valuable victim
 2. Least valuable aggressor
3. Killer heuristic

tip : Spring03 - 29

Static Evaluation

Initially	-	Very Complex
70's	-	Very simple (material)
now	-	Deep searchers: moderately complex (hardware) PC programs: elaborate, hand tuned

tip : Spring03 - 30

Slide 3.4.30

The static evaluation function is the other place where substantial game knowledge is encoded. In the early chess players, the evaluation functions were very complex (and buggy). Over time it was discovered that using a simple, reliable evaluator (for example, just a weighted count of pieces on the board) and deeper search provided better results. Today, systems such as Deep Blue use static evaluators of medium complexity implemented in hardware. Not surprisingly, the "cheap" PC programs, which can't search as deeply as Deep Blue rely on quite complex evaluation functions. In general, there is a tradeoff between the complexity of the evaluator and the depth of the search.

Slide 3.4.31

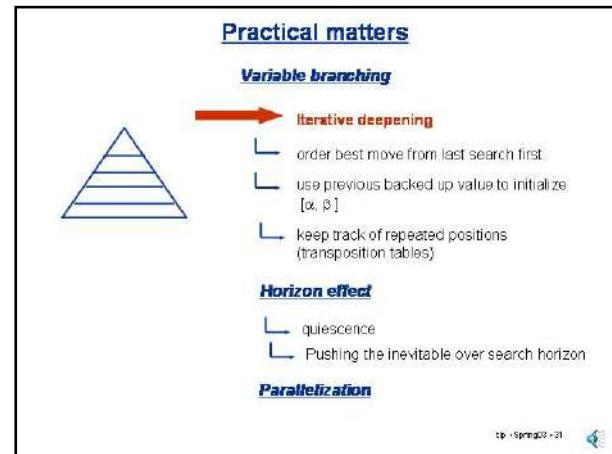
As one can imagine in an area that has received as much attention as game playing programs, there are a million and one techniques that have been tried and which make a difference in practice. Here we touch on a couple of the high points.

Chess and other such games have incredibly large trees with highly variable branching factor (especially since alpha-beta cutoffs affect the actual branching of the search). If we picked a fixed depth to search, as we've suggested earlier, then much of the time we would finish too quickly and at other times take too long. A better approach is to use iterative deepening and thus always have a move ready and then simply stop after some allotted time.

One of the nice side effects of iterative deepening is that the results of the last iteration of the search can be used to help in the next iteration. For example, we can use the last search to order the moves. A somewhat less obvious advantage is that we can use the previous results to pick an initial value of alpha and beta. Instead of starting with alpha and beta at minus and plus infinity, we can start them in a small window around the values found in the previous search. This will help us cutoff more irrelevant moves early. In fact, it is often useful to start with the tightest possible window, something like $[\alpha, \alpha + \epsilon]$ which is simply asking "is the last move we found still the best move"? In many cases, it is.

Another issue in fixed depth searches is known as the "horizon effect". That is, if we pick a fixed depth search, we could miss something very important right over the horizon. So, it would not do to stop searching right as your queen is about to be captured. Most game programs attempt to assess whether a "leaf" node is in fact "static" or "quiescent" before terminating the search. If the situation looks dynamic, the search is continued. In Deep Blue, as I mentioned earlier, some moves are searched to a depth of 30 ply because of this.

Obviously, Deep Blue makes extensive use of parallelization in its search. This turns out to be surprisingly hard to do effectively and was probably the most significant innovation in Deep Blue.



tip · Spring03 · 21

Other Games

- Backgammon**
 - Involves randomness – dice rolls
 - Machine-learning based player was able to draw the world champion human player.
- Bridge**
 - Involves hidden information – other players' cards – and communication during bidding.
 - Computer players play well but do not bid well
- Go**
 - No new elements but huge branching factor
 - No good computer players exist

tip · Spring03 · 32

Slide 3.4.32

In this section, we have focused on chess. There are a variety of other types of games that remain hard today, in spite of the relentless increase in computing power, and other games that require a different treatment.

Backgammon is interesting because of the randomness introduced by the dice. Humans are not so good at building computer players for this directly, but a machine learning system (that essentially did a lot of search and used the results to build a very good evaluation function) was able to draw the human world-champion.

Bridge is interesting because it has hidden information (the other players' cards) and communication with a partner in a restricted language. Computer players, using search, excel now in the card-play phase of the game, but are still not too good at the bidding phase (which involves all the quirks of communication with another human).

Go is actually in the same class of games as chess: there is no randomness, hidden information, or communication. But the branching factor is enormous and it seems not to be susceptible to search-based methods that work well in chess. Go players seem to rely more on a complex understanding of spatial patterns, which might argue for a method that is based more strongly on a good evaluation function than on brute-force search.

Slide 3.4.33

There are a few observations about game playing programs that actually are observations about the whole symbolic approach to AI. The great successes of machine intelligence come in areas where the rules are clear and people have a hard time doing well, for example, mathematics and chess. What has proven to be really hard for AI are the more nebulous activities of language, vision and common sense, all of which evolution has been selecting for. Most of the research in AI today focuses on these less well defined activities.

The other observation is that it takes many years of gradual refinement to achieve human-level competence even in well-defined activities such as chess. We should not expect immediate success in attacking any of the grand challenges of AI.

OBSERVATIONS

- Computers excel in well-defined activities where rules are clear
 - chess
 - mathematics
- Success comes after a long period of gradual refinement

For more detail on building game programs visit:
<http://www1.ics.uci.edu/~eppstein/180a/w99.html>

tip · Spring03 · 33

6.034 Notes: Section 4.1

Slide 4.1.1

So far, we have talked a lot about building systems that have knowledge represented in them explicitly. One way to acquire that knowledge is to build it in by hand. But that can be time-consuming and error prone. And many times, humans just don't have the relevant information available to them.

For instance, you are all experts in visual object recognition. But it's unlikely that you could sit down and write a program that would do the job even remotely as well as you can.

But, in lots of cases, when we don't have direct access to a formal description of a problem, we can learn something about it from examples.

Learning

- It is often hard to articulate the knowledge we need to build AI systems
- Often, we don't even know it!
- Frequently, we can arrange to build systems that learn it themselves

6.034 - Spring 03 • 1
What is Learning?

- memorizing something
- learning facts through observation and exploration
- improving motor and/or cognitive skills through practice
- organizing new knowledge into general, effective representations

6.034 - Spring 03 • 2
Slide 4.1.2

Herb Simon (an important historical figure in AI and in Economics) gave us this definition of learning:

Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the task or tasks drawn from the same population more efficiently and more effectively the next time.

This is not entirely precise, but it gives us the idea that learning systems have to acquire some information from examples of a problem, and perform better because of it (so we wouldn't call a system that simply logs all the images it has ever seen a learning system, because, although, in some sense, it knows a lot, it can't take advantage of its knowledge).

What is Learning?

- memorizing something
- learning facts through observation and exploration
- improving motor and/or cognitive skills through practice
- organizing new knowledge into general, effective representations

"Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the task or tasks drawn from the same population more efficiently and more effectively the next time." -- Herb Simon

6.034 - Spring 03 • 3

Induction

Piece of bread 1 was nourishing when I ate it
 Piece of bread 2 was nourishing when I ate it.
 ...
 Piece of bread 100 was nourishing when I ate it.
Therefore, all pieces of bread will be nourishing if I eat them.

Image of David Hume removed due to copyright restrictions.

David Hume

6.034 - Spring 03 • 4

Slide 4.1.4

One of the most common kinds of learning is the acquisition of information with the goal of making predictions about the future. But what exactly gives us license to imagine we can predict the future? Lots of philosophers have thought about this problem. David Hume first framed the problem of induction as follows:

- Piece of bread number 1 was nourishing when I ate it.
- Piece of bread number 2 was nourishing when I ate it.
- Piece of bread number 3 was nourishing when I ate it.
- Piece of bread number 100 was nourishing when I ate it.
- *Therefore*, all pieces of bread will be nourishing if I eat them.

Slide 4.1.5

And here, for no good reason, is a photo of David Hume's Tomb, just because I saw it once in Edinburgh.

Induction

Piece of bread 1 was nourishing when I ate it
 Piece of bread 2 was nourishing when I ate it.
 ...
 Piece of bread 100 was nourishing when I ate it.
Therefore, all pieces of bread will be nourishing if I eat them.

Image of David Hume removed due to
copyright restrictions.

Image removed due to
copyright restrictions.

David Hume

6.034 - Spring 03 • 5

Why is Induction Okay?

It has been argued that we have reason to know the future will resemble the past, because what was the future has constantly become the past, and has always been found to resemble the past, so that we really have experience of the future, namely of times which were formerly future, which we may call past futures. But such an argument really begs the very question at issue. We have experience of past futures, but not of future futures, and the question is: Will future futures resemble past futures?

Image of Bertrand Russell removed due to copyright restrictions.

Bertrand Russell

<http://www.ditext.com/russell/rus6.html>

6.034 - Spring 03 • 6

Slide 4.1.6

When and why is it okay to apply inductive reasoning??

My favorite quote on the subject is the following, excerpted from Bertrand Russell's "On Induction": (see <http://www.ditext.com/russell/rus6.html> for the whole thing)

If asked why we believe the sun will rise tomorrow, we shall naturally answer, 'Because it has always risen every day.' We have a firm belief that it will rise in the future, because it has risen in the past.

The real question is: Do any number of cases of a law being fulfilled in the past afford evidence that it will be fulfilled in the future?

It has been argued that we have reason to know the future will resemble the past, because what was the future has constantly become the past, and has always been found to resemble the past, so that we really have experience of the future, namely of times which were formerly future, which we may call past futures. But such an argument really begs the very question at issue. We have experience of past futures, but not of future futures, and the question is: Will future futures resemble past futures?

We won't worry too much about this problem. If induction is not, somehow, justified, then we have

no reason to get out of bed in the morning, let alone study machine learning!

Slide 4.1.7

When viewed technically, there are lots of different kinds of machine learning problems. We'll just sketch them out here, so you get an idea of their range.

Kinds of Learning

6.034 - Spring 03 • 7

Kinds of Learning

- **Supervised learning:** given a set of example input/output pairs, find a rule that does a good job of predicting the output associated with a new input

Slide 4.1.8

Supervised learning is the most common learning problem. Let's say you are given the weights and lengths of a bunch of individual salmon fish, and the weights and lengths of a bunch of individual tuna fish. The job of a supervised learning system would be to find a predictive rule that, given the weight and length of a fish, would predict whether it was a salmon or a tuna.

6.034 - Spring 03 • 8

Slide 4.1.9

Another, somewhat less well-specified, learning problem is **clustering**. Now you're given the descriptions of a bunch of different individual animals (or stars, or documents) in terms of a set of features (weight, number of legs, presence of hair, etc), and your job is to divide them into groups, or possibly into a hierarchy of groups that "makes sense". What makes this different from supervised learning is that you're not told in advance what groups the animals should be put into; just that you should find a natural grouping.

Kinds of Learning

- **Supervised learning:** given a set of example input/output pairs, find a rule that does a good job of predicting the output associated with a new input
- **Clustering:** given a set of examples, but no labeling of them, group the examples into "natural" clusters

6.034 - Spring 03 • 9

Kinds of Learning

- **Supervised learning:** given a set of example input/output pairs, find a rule that does a good job of predicting the output associated with a new input
- **Clustering:** given a set of examples, but no labeling of them, group the examples into "natural" clusters
- **Reinforcement learning:** an agent interacting with the world makes observations, takes actions, and is rewarded or punished; it should learn to choose actions in such a way as to obtain a lot of reward

Slide 4.1.10

Another learning problem, familiar to most of us, is learning motor skills, like riding a bike. We call this **reinforcement learning**. It's different from supervised learning because no-one explicitly tells you the right thing to do; you just have to try things and see what makes you fall over and what keeps you upright.

Supervised learning is the most straightforward, prevalent, and well-studied version of the learning problem, so we are going to start with it, and spend most of our time on it. Most of the fundamental insights into machine learning can be seen in the supervised case.

6.034 - Spring 03 • 10

Slide 4.1.11

One way to think about learning is that you're trying to find the definition of a function, given a bunch of examples of its input and output. This might seem like a pretty narrow definition, but it actually covers a lot of cases.

Learning a Function

Given a set of examples of input/output pairs, find a function that does a good job of expressing the relationship

6.034 - Spring 03 • 11

Learning a Function

Given a set of examples of input/output pairs, find a function that does a good job of expressing the relationship

- Pronunciation: function from letters to sounds
- Throw a ball: function from target locations to joint torques
- Read handwritten characters: function from collections of image pixels to letters
- Diagnose diseases: function from lab test results to disease categories

6.034 - Spring 03 • 12

Slide 4.1.12

- Learning how to pronounce words can be thought of as finding a function from letters to sounds.
- Learning how to throw a ball can be thought of as finding a function from target locations to joint torques.
- Learning to recognize handwritten characters can be thought of as finding a function from collections of image pixels to letters.
- Learning to diagnose diseases can be thought of as finding a function from lab test results to disease categories.

Slide 4.1.13

The problem of learning a function from examples, is complicated. You can think of at least three different problems being involved: memory, averaging, and generalization. We'll look at each of these problems, by example.

Aspects of Function Learning

- memory
- averaging
- generalization

6.034 - Spring 03 • 13

Example problem

When to drive the car? Depends on:

- temperature
- expected precipitation
- day of the week
- whether she needs to shop on the way home
- what she's wearing

6.034 - Spring 03 • 14

Slide 4.1.14

We'll do this in the context of a silly example problem. Imagine that I'm trying predict whether my neighbor is going to drive into work tomorrow, so I can ask for a ride. Whether she drives into work seems to depend on the following attributes of the day: the temperature, what kind of precipitation is expected, the day of the week, whether she needs to shop on the way home, and what she's wearing.

Slide 4.1.15

Okay. Let's say we observe our neighbor on three days, which are described in the table, which specifies the properties of the days and whether or not the neighbor walked or drove.

Memory

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
19	snow	mon	yes	casual	drive
65	none	tues	no	casual	walk

6.034 - Spring 03 • 15

Memory

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
19	snow	mon	yes	casual	drive
65	none	tues	no	casual	walk
19	snow	mon	yes	casual	

6.034 - Spring 03 • 16

Slide 4.1.16

The standard answer in this case is "yes". This day is just like one of the ones we've seen before, and so it seems like a good bet to predict "yes." This is about the most rudimentary form of learning, which is just to memorize the things you've seen before. Still, it can help you do a better job in the future, if those same cases arise again.

Memory

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
19	snow	mon	yes	casual	drive
65	none	tues	no	casual	walk
19	snow	mon	yes	casual	drive

6.034 - Spring 03 • 17

Averaging

Dealing with noise in the data

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	drive
80	none	sat	no	casual	drive
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk

6.034 - Spring 03 • 18

Slide 4.1.18

Things are not always as easy as they were in the previous case. What if you get this set of data?

Slide 4.1.19

Now, we ask you to predict what's going to happen. You've certainly seen this case before. But the problem is that it has had different answers. Our neighbor is not entirely reliable.

Averaging

Dealing with noise in the data

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	drive
80	none	sat	no	casual	drive
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	

6.034 - Spring 03 • 19

Averaging

Dealing with noise in the data

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	drive
80	none	sat	no	casual	drive
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk
80	none	sat	no	casual	walk

6.034 - Spring 03 • 20

Slide 4.1.21

Here's another situation in which noise is an issue. This time, the noise has corrupted our descriptions of the situation (our thermometer is somewhat unreliable, as is our ability to assess the formality of the neighbor's clothing, given her peculiar taste).

Sensor noise

Dealing with noise in the data

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
82	none	sat	no	casual	walk
78	none	sat	no	casual	walk
21	none	sat	no	casual	drive
18	none	sat	no	casual	drive
19	none	sat	no	formal	drive
17	none	sat	no	casual	drive

6.034 - Spring 03 • 21

Sensor noise

Dealing with noise in the data

temp	precip	day	shop	clothes	
80	none	sat	no	casual	walk
82	none	sat	no	casual	walk
78	none	sat	no	casual	walk
21	none	sat	no	casual	drive
18	none	sat	no	casual	drive
19	none	sat	no	formal	drive
17	none	sat	no	casual	drive
20	none	sat	no	casual	drive

6.034 - Spring 03 • 22

Slide 4.1.22

In this case, we'd like to recognize that the day that we're asked to do the prediction for is apparently similar to cases we've seen before. It is ultimately impossible to tell whether this day is *really* different from those other days, or whether our sensors are just acting up. We'll just have to accept that uncertainty.

So, we'll have to treat this as an instance of the more general problem of generalization.

Slide 4.1.23

Take a good look at this data set.

Generalization

Dealing with previously unseen cases

temp	precip	day	shop	clothes	
71	none	fri	yes	formal	drive
36	none	sun	yes	casual	walk
62	rain	weds	no	casual	walk
93	none	mon	no	casual	drive
55	none	sat	no	formal	drive
80	none	sat	no	casual	walk
19	snow	mon	yes	casual	drive
65	none	tues	no	casual	walk

6.034 - Spring 03 • 23

Generalization

Dealing with previously unseen cases

temp	precip	day	shop	clothes	
71	none	fri	yes	formal	drive
36	none	sun	yes	casual	walk
62	rain	weds	no	casual	walk
93	none	mon	no	casual	drive
55	none	sat	no	formal	drive
80	none	sat	no	casual	walk
19	snow	mon	yes	casual	drive
65	none	tues	no	casual	walk
58	rain	mon	no	casual	

6.034 - Spring 03 • 24

more formally in the next section.

Slide 4.1.25

Imagine that you were given all these points, and you needed to guess a function of their x, y coordinates that would have one output for the red ones and a different output for the black ones.

Slide 4.1.24

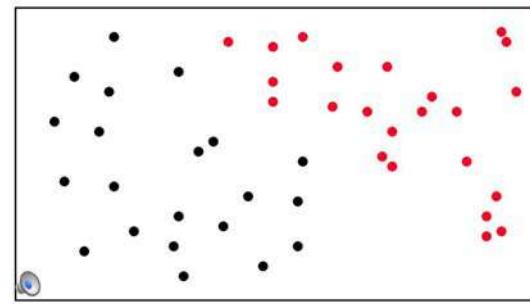
Now, consider this day. It's 58 degrees and raining on a Monday. The neighbor is wearing casual clothing and doesn't need to shop. Will she walk or drive?

The first thing to observe is that there's no obviously right answer. We have never seen this case before. We could just throw up our hands and decline to make a prediction. But ultimately, we have to decide whether or not to call the neighbor. So, we might fall back on some assumptions about the domain. We might assume, for instance, that there's a kind of *smoothness* property: similar situations will tend to have similar categories.

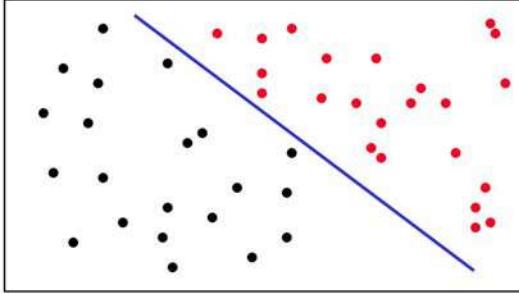
You might plausibly make any of the following arguments:

- She's going to walk because it's raining today and the only other time it rained, she walked.
- She's going to drive because she has always driven on Mondays.
- She's going to walk because she only drives if she is wearing formal clothes, or if the temperature is above 90 or below 20.

The question of which one to choose is hard. It's one of the deep underlying problems of machine learning. We'll look at some examples to try to get more intuition, then come back and revisit it a bit

The Red and the Black

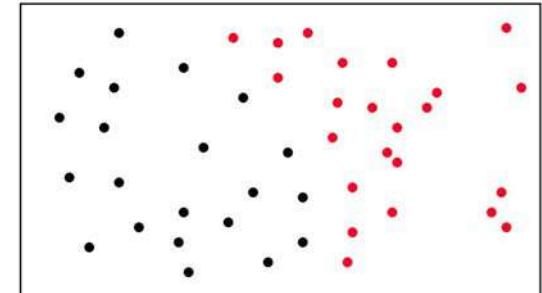
6.034 - Spring 03 • 25

What's the right hypothesis?**Slide 4.1.26**

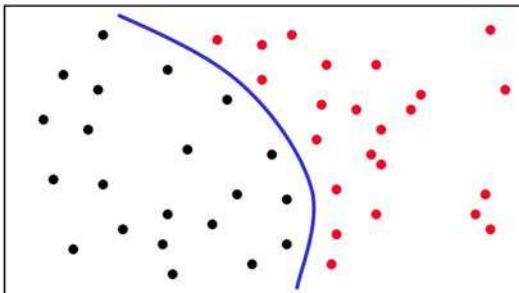
In this case, it seems like you could do pretty well by defining a line that separates the two classes.

Slide 4.1.27

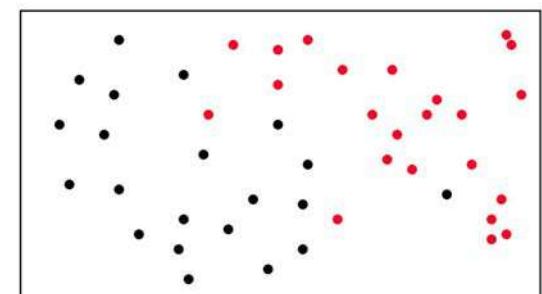
Now, what if we have a slightly different configuration of points? We can't divide them conveniently with a line.

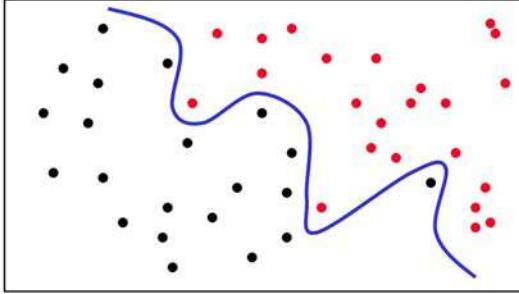
Now, what's the right hypothesis?**Slide 4.1.28**

But this parabola-like curve seems like it might be a reasonable separator.

Now, what's the right hypothesis?**Slide 4.1.29**

Now, what? This seems much trickier. There are at least a couple of reasonable kinds of answers.

How about now?

How about now? Answer 1**Slide 4.1.30**

Here's an answer. It successfully separates the reds from the blacks. But there's something sort of unsatisfactory about it. Most people have an intuition that it's too complicated.

Slide 4.1.31

Here's another answer. It's not a perfect separator; it gets some of the points wrong. But it's prettier.

In general, we will always be faced with making trade-offs between hypotheses that account for the data perfectly and hypotheses that are, in some sense, simple. There are some mathematical ways to understand these trade-offs; we'll avoid the math, but try to show you, intuitively, how they come up and what to do about them.

Variety of Learning Methods

Learning methods differ in terms of:

- the form of the hypothesis
- the way the computer finds a hypothesis given the data

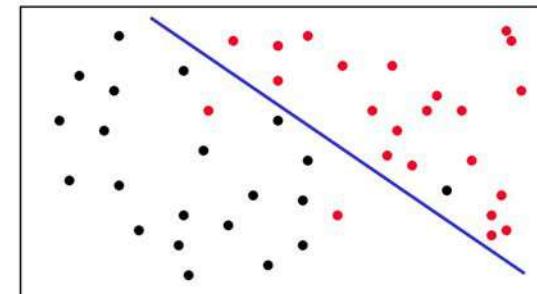
Slide 4.1.32

There are lots and lots of different kinds of learning algorithms. I want to show you cartoon versions of a couple of them first, and then we'll get down to the business of understanding them at a detailed algorithmic level.

The learning methods differ both in terms of the kinds of hypotheses they work with and the algorithms they use to find a good hypothesis given the data.

Slide 4.1.33

Here's the most simple-minded of all learning algorithms (and my personal favorite (which might imply something about me!)). It's called nearest neighbor (or lazy learning). You simply remember all the examples you've ever seen.

How about now? Answer 2

6.034 - Spring 03 • 31

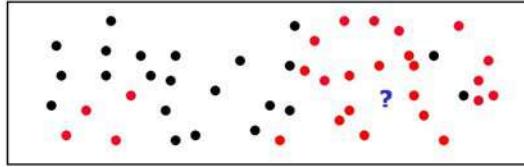
Nearest Neighbor

- Remember all your data

6.034 - Spring 03 • 32

Nearest Neighbor

- Remember all your data
- When someone asks a question,
 - find the nearest old data point
 - return the answer associated with it



Slide 4.1.34

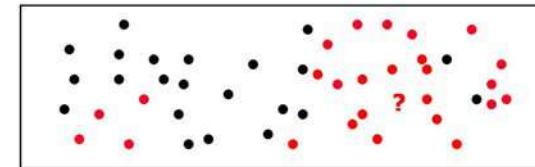
When someone asks you a question, you simply find the existing point that's nearest the one you were asked about, and return its class.

Slide 4.1.35

So, in this case, the nearest point to the query is red, so we'd return red.

Nearest Neighbor

- Remember all your data
- When someone asks a question,
 - find the nearest old data point
 - return the answer associated with it

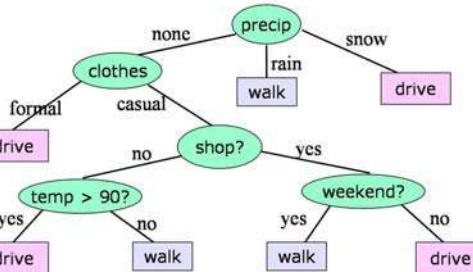


Slide 4.1.36

Another well-loved learning algorithm makes hypotheses in the form of decision trees. In a decision tree, each node represents a question, and the arcs represent possible answers. We can use this decision tree to find out what prediction we should make in the drive/walk problem.

Decision Trees

Use all the data to build a tree of questions with answers at the leaves



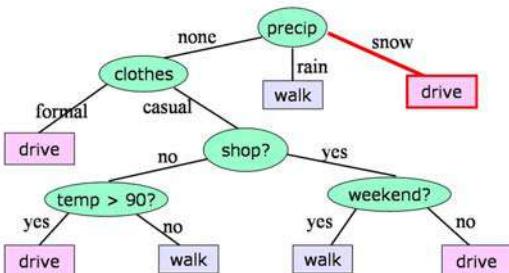
6.034 - Spring 03 • 36

Slide 4.1.37

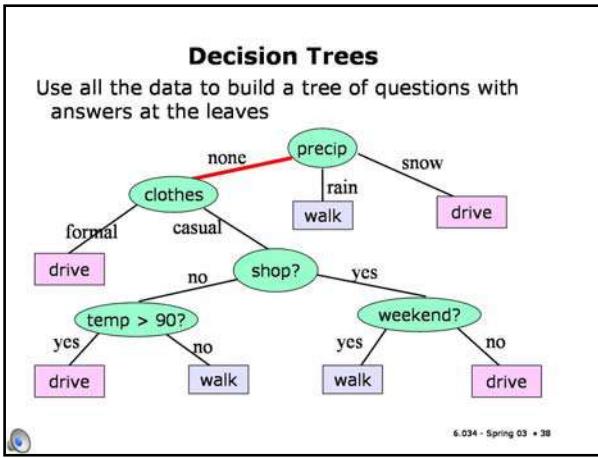
We'd start by asking what the current precipitation is. If it's snow, we just stop asking questions and drive.

Decision Trees

Use all the data to build a tree of questions with answers at the leaves



6.034 - Spring 03 • 37

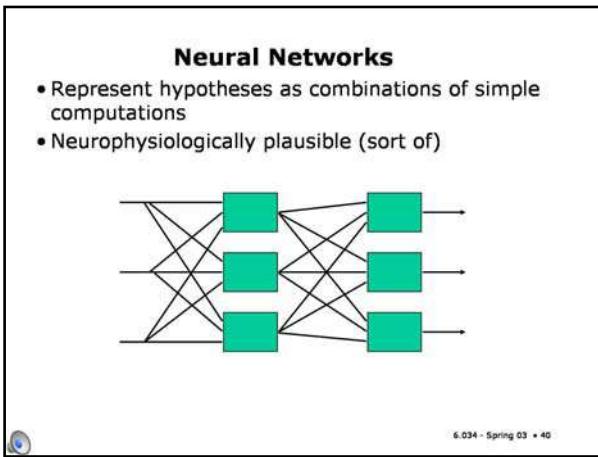
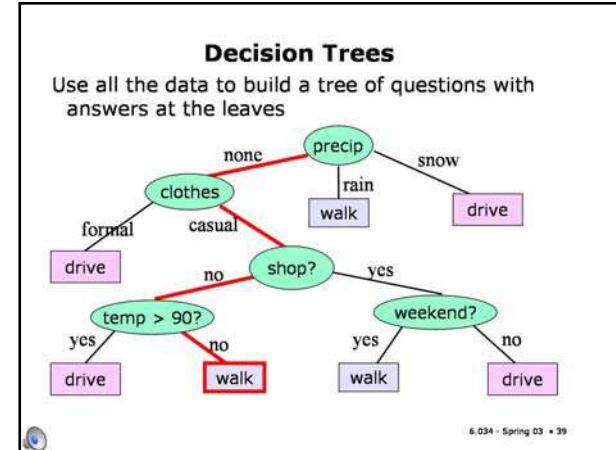
**Slide 4.1.38**

If there's no precipitation, then we have to ask what kind of clothes the neighbor is wearing. If they're formal, she'll drive. If not, we have to ask another question.

Slide 4.1.39

We can always continue asking and answering questions until we get to a leaf node of the tree; the leaf will always contain a prediction.

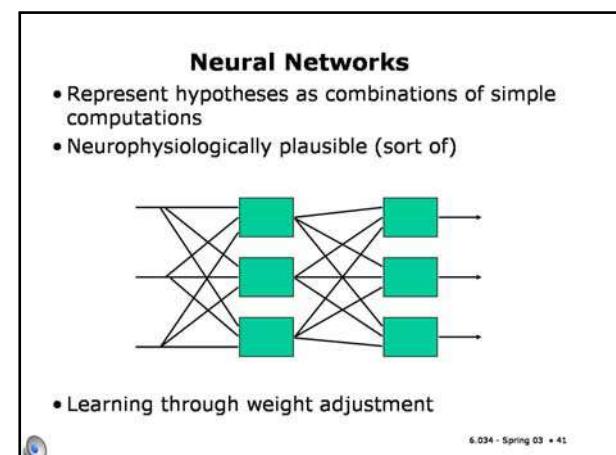
Hypotheses like this are nice for a variety of reasons. One important one is that they're relatively easily interpretable by humans. So, in some cases, we run a learning algorithm on some data and then show the results to experts in the area (astronomers, physicians), and they find that the learning algorithm has found some regularities in their data that are of real interest to them.

**Slide 4.1.40**

Most people have heard of neural networks. They can represent complicated hypotheses in high-dimensional continuous spaces. They are attractive as a computational model because they are composed of many small computing units. They were motivated by the structure of neural systems in parts of the brain. Now it is understood that they are not an exact model of neural function, but they have proved to be useful from a purely practical perspective.

Slide 4.1.41

In neural networks, the individual units do a simple, fixed computation that combines the values coming into them with "weights" on the arcs. The learning process consists of adjusting the weights on the arcs in reaction to errors made by the network.



Machine Learning Successes

- assessing loan credit risk
- detecting credit card fraud
- cataloging astronomical images
- detecting and diagnosing manufacturing faults
- helping NBA coaches analyze performance
- personalizing news and web searches
- steering an autonomous car across the US

Slide 4.1.42

Machine learning methods have been successfully fielded in a variety of applications, including

- assessing loan credit risk
- detecting credit card fraud
- cataloging astronomical images
- deciding which catalogs to send to your house
- helping NBA coaches analyze players' performance
- personalizing news and web searches
- steering an autonomous car across the US

Now, we'll roll up our sleeves, and see how all this is done.

6.034 Notes: Section 4.2

Slide 4.2.1

A supervised learning problem is made up of a number of ingredients. The first is the data (sometimes called the training set). The data, D, is a set of m input-output pairs. We will write the i^{th} element of the data set as $\langle x^i, y^i \rangle$. In the context of our old running example, an x^i would be a vector of values, one for each of the input features, and a y^i would be an output value (walk or drive).

Supervised Learning

- Given data (training set)

$$D = \{\langle x^1, y^1 \rangle, \langle x^2, y^2 \rangle, \dots, \langle x^m, y^m \rangle\}$$

6.034 - Spring 03 • 1

Supervised Learning

- Given data (training set)

$$D = \{\langle x^1, y^1 \rangle, \langle x^2, y^2 \rangle, \dots, \langle x^m, y^m \rangle\}$$

$\langle x_1^1, x_2^1, \dots, x_n^1 \rangle$
input

Slide 4.2.2

Each x^i is a vector of n values. We'll write x_j^i for the j^{th} feature of the i^{th} input-output pair. We'll consider different kinds of features. Sometimes we'll restrict ourselves to the case where the features are only 0s and 1s. Other times, we'll let them be chosen from a set of discrete elements (like "snow", "rain", "none"). And, still other times, we'll let them be real values, like temperature, or weight.

6.034 - Spring 03 • 2

Slide 4.2.3

Similarly, the output, y^i , might be a boolean, a member of a discrete set, or a real value. For a particular problem, the y^i will all be of a particular type.

When y^i is a boolean, or a member of a discrete set, we will call the problem a **classification** problem. When y^i is real-valued, we call this a **regression** problem.

Supervised Learning

- Given data (training set)

$$D = \{ \langle x^1, y^1 \rangle, \langle x^2, y^2 \rangle, \dots, \langle x^m, y^m \rangle \}$$

$\langle x_1^1, x_2^1, \dots, x_n^1 \rangle$
input

output

Classification: discrete Y

Regression: continuous Y

6.034 - Spring 03 • 3

Supervised Learning

- Given data (training set)

$$D = \{ \langle x^1, y^1 \rangle, \langle x^2, y^2 \rangle, \dots, \langle x^m, y^m \rangle \}$$

$\langle x_1^1, x_2^1, \dots, x_n^1 \rangle$
input

output

Classification: discrete Y
Regression: continuous Y

- Goal: find a hypothesis h in hypothesis class H that does a good job of mapping x to y

6.034 - Spring 03 • 4

Slide 4.2.5

So, now we have a bunch of data, and a class of possible answers (hypotheses) and we're supposed to return the best one. In order to do that, we need to know exactly what is meant by "best".

Best Hypothesis

6.034 - Spring 03 • 5

Best Hypothesis

Hypothesis should

- do a good job of describing the data

- not be too complex

Slide 4.2.6

There are typically two components to the notion of best. The hypothesis should do a good job of describing the data and it shouldn't be too complicated.

6.034 - Spring 03 • 6

Slide 4.2.7

Ideally, we would like to find a hypothesis h such that, for all data points i , $h(x^i) = y^i$. We will not always be able to (or maybe even want to) achieve this, so perhaps it will only be true for most of the data points, or the equality will be weakened into "not too far from".

We can sometimes develop a measure of the "error" of a hypothesis to the data, written $E(h, D)$. It might be the number of points that are misclassified, for example.

Best Hypothesis

Hypothesis should

- do a good job of describing the data
 - ideally: $h(x^i) = y^i$
 - number of errors: $E(h, D)$
- not be too complex

6.034 - Spring 03 • 7

Best Hypothesis

Hypothesis should

- do a good job of describing the data
 - ideally: $h(x^i) = y^i$
 - number of errors: $E(h, D)$
- not be too complex
 - measure: $C(h)$

6.034 - Spring 03 • 8

Slide 4.2.9

Why do we care about hypothesis complexity? We have an intuitive sense that, all things being equal, simpler hypotheses are better. Why is that? There are lots of statistical and philosophical and information-theoretic arguments in favor of simplicity.

So, for now, let's just make recourse to William of Ockham, a 14th century Franciscan theologian, logician, and heretic. He is famous for "Ockham's Razor", or the principle of parsimony: "Non sunt multiplicanda entia praeter necessitatem." That is, "entities are not to be multiplied beyond necessity". People interpret this to mean that we should always adopt the simplest satisfactory hypothesis.

Best Hypothesis

Hypothesis should

- do a good job of describing the data
 - ideally: $h(x^i) = y^i$
 - number of errors: $E(h, D)$
- not be too complex
 - measure: $C(h)$

Non sunt
multiplicanda
entia praeter
necessitatem

Image of William of Ockham
removed due to copyright restrictions.
William of Ockham

6.034 - Spring 03 • 9

Best Hypothesis

Hypothesis should

- do a good job of describing the data
 - ideally: $h(x^i) = y^i$
 - number of errors: $E(h, D)$
- not be too complex
 - measure: $C(h)$

Non sunt
multiplicanda
entia praeter
necessitatem

Minimize $E(h, D) + \alpha C(h)$

Image of William of Ockham
removed due to copyright restrictions.
William of Ockham

trade-off

6.034 - Spring 03 • 10

Slide 4.2.10

So, given a data set, D , a hypothesis class H , a goodness-of-fit function E , and a complexity measure C , our job will be to find the h in H that minimizes $E(h, D) + \alpha * C(h)$, where α is a number that we can vary to change how much we value fitting the data well versus having a complex hypothesis.

We won't, in general, be able to do this efficiently, so, as usual, we'll have to make a lot of approximations and short-cuts. In fact, most learning algorithms aren't described exactly this way. But it's the underlying principle behind most of what we do.

Slide 4.2.11

How can we go about finding the hypothesis with the least value of our criterion? We can certainly think of this as a search problem, although many learning algorithms don't seem to have the character of doing a search.

For some carefully-chosen combinations of hypothesis classes and error criteria, we can calculate the optimal hypothesis directly.

In other cases, the hypothesis space can be described with a vector of real-valued parameters, but for which we can't calculate the optimal values. A common practice for these problems is to do a kind of local hill-climbing search called gradient ascent (or descent).

Finally, sometimes it is possible to construct a hypothesis iteratively, starting with a simple hypothesis and adding to it in order to make it fit the data better. These methods are often called "greedy", because they start by trying to pick the best simple hypothesis, and then make the best addition to it, etc. The result isn't optimal, but it is often very good. We will start by looking at some methods of this kind.

Learning as Search

- How can we find the hypothesis with the lowest value of $E(h, D) + \alpha C(h)$? Search!

- For some hypothesis classes we can calculate the optimal h directly! (linear separators)
- For others, do local search (gradient descent in neural networks)
- For some structured hypothesis spaces, construct one greedily

6.034 - Spring 03 • 11

Propositional Logic

- Next sections use the language of propositional logic to specify hypotheses in the space of vectors of binary variables
- " A " is a variable that can take on values "true" and "false" or 1 and 0
- $A \wedge B$ means that variable A and variable B both have to be true (1)
- $A \vee B$ means that either variable A or variable B (or both) have to be true (1)
- $\neg A$ means that variable A has to be false (0)

6.034 - Spring 03 • 12

Slide 4.2.12

In the following sections, we speak about a set of problems in which the examples are described using vectors of binary variables. In such a space, it's convenient to characterize the hypotheses using propositional logic. We will talk about logic a lot in the last part of the course, but we just need to introduce some notation here.

We'll use single letters to stand for variables that can be true or false (often we'll use 1 for "true" and 0 for "false").

A "caret" B says that A and B both have to be true.

A "vee" B says that either A or B (or both) have to be true.

This thing that's sort of like a bent minus sign before a variable means that it has to be false.

Slide 4.2.13

Let's start with a very simple problem, in which all of the input features are Boolean (we'll represent them with 0's and 1's) and the desired output is also Boolean.

Learning Conjunctions

- Boolean features and output

6.034 - Spring 03 • 13

Learning Conjunctions

- Boolean features and output
- $H = \text{conjunctions of features}$

6.034 - Spring 03 • 14

Slide 4.2.14

Our hypothesis class will be conjunctions of the input features.

Slide 4.2.15

Here's an example data set. It is described using 4 features, which we'll call f_1 , f_2 , f_3 , and f_4 .

Learning Conjunctions

- Boolean features and output
- H = conjunctions of features

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

6.034 - Spring 03 • 15

**Learning Conjunctions**

- Boolean features and output
- H = conjunctions of features

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

$$h = f_1 \wedge f_3$$

6.034 - Spring 03 • 16

**Slide 4.2.16**

So, to understand the hypothesis space let's consider the hypothesis $f_1 \wedge f_3$. It would be true on examples 2 and 3, and false on all the rest.

Learning Conjunctions

- Boolean features and output
- H = conjunctions of features

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

$$h = f_1 \wedge f_3$$

$$E(h, D) = 3$$

6.034 - Spring 03 • 17

**Learning Conjunctions**

- Boolean features and output
- H = conjunctions of features

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

$$h = f_1 \wedge f_3$$

$$E(h, D) = 3$$

$$C(h) = 2$$

- Set alpha so we're looking for smallest h with 0 error

6.034 - Spring 03 • 18

**Slide 4.2.18**

Finally, we'll measure the complexity of our hypothesis by the number of features mentioned in the conjunction. So the hypothesis $f_1 \wedge f_3$ has complexity 2.

Furthermore, let's assume that alpha is set to be very small, so that our primary goal is to minimize error, but, errors being equal, we'd like to have the smallest conjunction.

Slide 4.2.19

There's now an obvious way to proceed. We could do a general-purpose search in the space of hypotheses, looking for the one that minimizes the cost function. In this case, that might work out okay, since the problem is very small. But in general, we'll want to work in domains with many more features and much more complex hypothesis spaces, making general-purpose search infeasible.

Algorithm

- Could search in hypothesis space using tools we've already studied

6.034 - Spring 03 • 19

Algorithm

- Could search in hypothesis space using tools we've already studied
- Instead, be greedy!

6.034 - Spring 03 • 20

Slide 4.2.20

We'll start out with our hypothesis set to True (that's the empty conjunction). Usually, it will make some errors. Our goal will be to add as few elements to the conjunction as necessary to make no errors.

Slide 4.2.20

Instead, we'll be greedy! (When I talk about this and other algorithms being greedy, I'll mean something slightly different than what we meant when we talked about "greedy search". That was a particular instance of a more general idea of greedy algorithm. In greedy algorithms, in general, we build up a solution to a complex problem by adding the piece to our solution that looks like it will help the most, based on a partial solution we already have. This will not, typically, result in an optimal solution, but it usually works out reasonably well, and is the only plausible option in many cases (because trying out all possible solutions would be much too expensive)).

Algorithm

- Could search in hypothesis space using tools we've already studied
- Instead, be greedy!
- Start with $h = \text{True}$

6.034 - Spring 03 • 21

Algorithm

- Could search in hypothesis space using tools we've already studied
- Instead, be greedy!
- Start with $h = \text{True}$
- All errors are on negative examples
- On each step, add conjunct that rules out most new negatives (without excluding positives)

6.034 - Spring 03 • 22

Slide 4.2.22

Notice that, because we've started with the hypothesis equal to True, all of our errors are on negative examples. So, one greedy strategy would be to add the feature into our conjunction that rules out as many negative examples as possible without ruling out any positive examples.

Slide 4.2.23

Here is pseudo-code for our algorithm. We start with N equal to all the negative examples and the hypothesis equal to true. Then we loop, adding conjuncts that rule out negative examples, until N is empty.

Pseudo-Code

```
N = negative examples in D
h = True
Loop until N is empty
```

6.034 - Spring 03 • 23

Pseudo-Code

```
N = negative examples in D
h = True
Loop until N is empty
  For every feature j that does not have value 0 on
  any positive examples
    nj := number of examples in N
    for which fj = 0
    j* := j for which nj is maximized
    h := h ^ fj*
    N := N - examples in N for which fj = 0
  If no such feature found, fail
```

6.034 - Spring 03 • 24

Slide 4.2.25

Let's simulate the algorithm on our data. We start with N equal to x^1, x^3 , and x^5 , which are the negative examples. And h starts as True. We'll cover all the examples that the hypothesis makes true pink.

Simulation

f ₁	f ₂	f ₃	f ₄	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $N = \{x^1, x^3, x^5\}$, $h = \text{True}$

6.034 - Spring 03 • 25

Simulation

f ₁	f ₂	f ₃	f ₄	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $N = \{x^1, x^3, x^5\}$, $h = \text{True}$
- $n_3 = 1$, $n_4 = 2$

6.034 - Spring 03 • 26

Slide 4.2.26

Now, we consider all the features that would not exclude any positive examples. Those are features f_3 and f_4 . f_3 would exclude 1 negative example; f_4 would exclude 2. So we pick f_4 .

Slide 4.2.27

Now we remove the examples from N that are ruled out by f_4 and add f_4 to h.

Simulation

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $N = \{x^1, x^3, x^5\}, h = \text{True}$
- $n_3 = 1, n_4 = 2$
- $N = \{x^5\}, h = f_4$

6.034 - Spring 03 • 27

Simulation

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $N = \{x^1, x^3, x^5\}, h = \text{True}$
- $n_3 = 1, n_4 = 2$
- $N = \{x^5\}, h = f_4$
- $n_3 = 1, n_4 = 0$

6.034 - Spring 03 • 28

Slide 4.2.29

Because f_3 rules out the last remaining negative example, we're done!

Simulation

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	0
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $N = \{x^1, x^3, x^5\}, h = \text{True}$
- $n_3 = 1, n_4 = 2$
- $N = \{x^5\}, h = f_4$
- $n_3 = 1, n_4 = 0$
- $N = \{\}, h = f_4 \wedge f_3$

6.034 - Spring 03 • 29

A Harder Problem

- We made one negative into a positive

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

6.034 - Spring 03 • 30

Slide 4.2.30

What if we have this data set? In which we just made one negative example into a positive?

Slide 4.2.31

If we simulate the algorithm, we will arrive at the situation in which we still have elements in N, but there are no features left that do not exclude positive examples. So we're stuck.

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- We made one negative into a positive
- Only usable feature is f_3
- Can't add any more features to h
- We're stuck

- Best we can do when H is conjunctions
- Live with error or change H

6.034 - Spring 03 • 32

Slide 4.2.32

You can kind of think of a conjunction as narrowing down on a small part of the space. And if all the positive examples can be corralled into one group this way, everything is fine. But for some concepts, it might be necessary to have multiple groups.

This corresponds to a form of Boolean hypothesis that's fairly easy to think about: disjunctive normal form. Disjunctive normal form is kind of like CNF, only this time it has the form $(A \wedge B \wedge C) \vee (D \wedge A) \vee E$

As a hypothesis class, it's like finding multiple groups of positive examples within the negative space.

A Harder Problem

- We made one negative into a positive
- Only usable feature is f_3
- Can't add any more features to h
- We're stuck

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

6.034 - Spring 03 • 31

Slide 4.2.32

What's going on? The problem is that this hypothesis class simply can't represent the data we have with no errors. So now we have a choice: we can accept the hypothesis we've got, or we can increase the size of the hypothesis class. In practice, which one you should do often depends on knowledge of the domain. But the fact is that pure conjunctions is a very limiting hypothesis class. So let's try something a little fancier.

Disjunctive Normal Form

- Like the opposite of conjunctive normal form (but, for now, without negations of the atoms)
 $(A \wedge B \wedge C) \vee (D \wedge A) \vee E$
- Think of each disjunct as narrowing in on a subset of the positive examples

6.034 - Spring 03 • 33

Disjunctive Normal Form

So, let's look at our harder data set and see if we can find a good hypothesis in DNF.

- Like the opposite of conjunctive normal form (but, for now, without negations of the atoms)
 $(A \wedge B \wedge C) \vee (D \wedge A) \vee E$
- Think of each disjunct as narrowing in on a subset of the positive examples

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

6.034 - Spring 03 • 34

Slide 4.2.35

It's pretty to see that one way to describe it is $f_3 \wedge f_4 \vee f_1 \wedge f_2$. Now let's look at an algorithm for finding it.

Disjunctive Normal Form

- Like the opposite of conjunctive normal form (but, for now, without negations of the atoms)

$$(A \wedge B \wedge C) \vee (D \wedge A) \vee E$$
- Think of each disjunct as narrowing in on a subset of the positive examples

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

$$(f_3 \wedge f_4) \vee (f_1 \wedge f_3)$$

6.034 - Spring 03 • 35

Learning DNF

- Let H be DNF expressions
- $C(h)$: number of mentions of features

$$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$$

Slide 4.2.36

We'll let our hypothesis class be expressions in disjunctive normal form. What should our measure of complexity be? One reasonable choice is the total number of mentions of features (so $f_3 \wedge f_4 \vee f_1 \wedge f_2$ would have complexity 4).

Learning DNF

- Let H be DNF expressions
- $C(h)$: number of mentions of features

$$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$$

- Really hard to search this space, so be greedy again!

6.034 - Spring 03 • 36

Learning DNF

- Let H be DNF expressions
- $C(h)$: number of mentions of features

$$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$$

- Really hard to search this space, so be greedy again!

- A conjunction **covers** an example if all of the features mentioned in the conjunction are true in the example

Slide 4.2.38

We'll say a conjunction **covers** an example if all of the features mentioned in the conjunction are true in the example.

Disjunctive Normal Form

- Like the opposite of conjunctive normal form (but, for now, without negations of the atoms)

$$(A \wedge B \wedge C) \vee (D \wedge A) \vee E$$
- Think of each disjunct as narrowing in on a subset of the positive examples

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

$$(f_3 \wedge f_4) \vee (f_1 \wedge f_3)$$

6.034 - Spring 03 • 35

6.034 - Spring 03 • 36

Learning DNF

- Let H be DNF expressions
- $C(h)$: number of mentions of features

$$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$$

- Really hard to search this space, so be greedy again!

6.034 - Spring 03 • 36

6.034 - Spring 03 • 38

Slide 4.2.39

Here's pseudocode for the algorithm. It has two main loops. The inner loop constructs a conjunction (much like our previous algorithm). The outer loop constructs multiple conjunctions and disjoins them.

The idea is that each disjunct will *cover* or account for some subset of the positive examples. So in the outer loop, we make a conjunction that includes some positive examples and no negative examples, and add it to our hypothesis. We keep doing that until no more positive examples remain to be covered.

Algorithm

```

P = set of all positive examples
h = False
Loop until P is empty
  r = True
  N = set of all negative examples
  Loop until N is empty
    If all features are in r, fail
    Else, select a feature fj to add to r
      r = r ^ fj
      N = N - examples in n for which fj = 0
  h := h v r
  Covered := examples in P covered by r
  If Covered is empty, fail
  Else P := P - Covered
end

```

6.034 - Spring 03 • 39

Choosing a Feature

$$\text{Heuristic: } v_j = \frac{n_j^+}{\max(n_j^+, 0.001)}$$

Slide 4.2.40

We also haven't said how to pick which feature to add to r. It's trickier here, because we can't restrict our attention to features that allow us to cover all possible examples. Here's one heuristic for selecting which feature to add next.

6.034 - Spring 03 • 40

Slide 4.2.41

Let v_j be n_j^+ / n_j^- , where n_j^+ is the total number of so-far uncovered positive examples that are covered by rule $r \wedge f_j$ and n_j^- is the total number of thus-far not-ruled-out negative examples that are covered by rule $r \wedge f_j$. The intuition here is that we'd like to add features that cover a lot of positive examples and exclude a lot of negative examples, because that's our overall goal.

There's one additional problem about what to do when n_j^- is 0. In that case, this is a really good feature, because it covers positives and doesn't admit any negatives. We'll prefer features with zero in the denominator over all others; if we have multiple such features, we'll prefer ones with bigger numerator. To make this fit easily into our framework, if the denominator is zero, we just return as a score 1 million times the numerator.

Choosing a Feature

$$\text{Heuristic: } v_j = \frac{n_j^+}{\max(n_j^+, 0.001)}$$

$n_j^+ = \# \text{not yet covered positive examples}$
 $\text{covered by } r \wedge f_j$

$n_j^- = \# \text{not yet ruled out negative examples}$
 $\text{covered by } r \wedge f_j$

6.034 - Spring 03 • 41

Choosing a Feature

$$\text{Heuristic: } v_j = \frac{n_j^+}{\max(n_j^+, 0.001)}$$

$n_j^+ = \# \text{not yet covered positive examples}$
 $\text{covered by } r \wedge f_j$

$n_j^- = \# \text{not yet ruled out negative examples}$
 $\text{covered by } r \wedge f_j$

Choose feature with largest value of V_j

Slide 4.2.42

Then we can replace this line in the code with one that says: Select the feature f_j with the highest value of v_j to add to r . And now we have a whole algorithm.

6.034 - Spring 03 • 42

Slide 4.2.43

Let's simulate it on our data set. Here's a trace of what happens. It's important to remember that when we compute v_j , we consider only positive and negative examples not covered so far.

We start with $h = \text{false}$, and so our current hypothesis doesn't make any examples true.

Simulation

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $h = \text{False}, P = \{x^2, x^3, x^4, x^6\}$

6.034 - Spring 03 • 43

Simulation

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $h = \text{False}, P = \{x^2, x^3, x^4, x^6\}$
- $r = \text{True}, N = \{x^1, x^5\}$
- $v_1 = 2/1, v_2 = 2/1, v_3 = 4/1, v_4 = 3/1$
- $r = f_3, N = \{x^1\}$
- $v_1 = 2/0, v_2 = 2/1, v_4 = 3/0$
- $r = f_3 \wedge f_4, N = \{\}$
- $h = f_3 \wedge f_4, P = \{x^3\}$

6.034 - Spring 03 • 44

Slide 4.2.45

After another iteration, we add a new rule, which covers the example shown in blue. And we're done.

Simulation

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	1

- $h = \text{False}, P = \{x^2, x^3, x^4, x^6\}$
- $r = \text{True}, N = \{x^1, x^5\}$
- $v_1 = 2/1, v_2 = 2/1, v_3 = 4/1, v_4 = 3/1$
- $r = f_3, N = \{x^1\}$
- $v_1 = 2/0, v_2 = 2/1, v_4 = 3/0$
- $r = f_3 \wedge f_4, N = \{\}$
- $h = f_3 \wedge f_4, P = \{x^3\}$
- $r = \text{True}, N = \{x^1, x^5\}$
- $v_1 = 1/1, v_2 = 1/1, v_3 = 1/1, v_4 = 0/1$
- $r = f_1, N = \{x^1\}$
- $v_2 = 1/0, v_3 = 1/0, v_4 = 0/1$
- $r = f_1 \wedge f_2, N = \{\}$
- $h = (f_3 \wedge f_4) \vee (f_1 \wedge f_2), P = \{\}$

6.034 - Spring 03 • 45

6.034 Notes: Section 4.3

Slide 4.3.1

Once our learning algorithm has generated a hypothesis, we'd like to be able to estimate how well it is going to work out in the "real world". What is the real world? Well, we can't expect the hypothesis to perform well in every possible world; we *can* expect it to perform well in the world that generated the data that was used to train it.

How Well Does it Work?

- We'd like to know how well our h will perform on new data (drawn from the same distribution as the training data)

6.034 - Spring 03 • 1

How Well Does it Work?

- We'd like to know how well our h will perform on new data (drawn from the same distribution as the training data)
- Performance of hypothesis on the training set is not indicative of its performance on new data

6.034 - Spring 03 • 2

Slide 4.3.2

The only way to know how well the hypothesis will do on new data is to save some of the original data in a set called the "validation set" or the "test set". The data in the validation set are not used during training. Instead, they are "held out" until the hypothesis has been derived. Then, we can test the hypothesis on the validation set and see what percentage of the cases it gets wrong. This is called "test set error".

How Well Does it Work?

- We'd like to know how well our h will perform on new data (drawn from the same distribution as the training data)
- Performance of hypothesis on the training set is not indicative of its performance on new data
- Save some data as a **test set**; performance on h is a reasonable estimate of performance on new data

6.034 - Spring 03 • 3

Cross Validation

To evaluate performance of an algorithm as a whole (rather than a particular hypothesis):

Slide 4.3.4

Sometimes you'd like to know how well a particular learning algorithm is working, rather than evaluating a single hypothesis it may have produced. Running the algorithm just once and evaluating its hypothesis only gives you one sample of its behavior, and it might be hard to tell how well it really works from just one sample.

If you have **lots** of data, you can divide it up into batches, and use half of your individual batches to train new hypotheses and the other half of them to evaluate the hypotheses. But you rarely have enough data to be able to do this.

6.034 - Spring 03 • 4

Slide 4.3.5

One solution is called cross-validation. You divide your data up into some number (let's say 10 for now) of chunks. You start by feeding chunks 1 through 9 to your learner and evaluating the resulting hypothesis using chunk 10.

Now, you feed all but chunk 9 into the learner and evaluate the resulting hypotheses on chunk 9. And so on.

Cross Validation

To evaluate performance of an algorithm as a whole (rather than a particular hypothesis):

- Divide data into k subsets
- k different times
 - train on k-1 of the subsets
 - test on the held-out subset
- return average test score over all k tests

6.034 - Spring 03 • 5

Cross Validation

To evaluate performance of an algorithm as a whole (rather than a particular hypothesis):

- Divide data into k subsets
- k different times
 - train on k-1 of the subsets
 - test on the held-out subset
- return average test score over all k tests

Useful for deciding which class of algorithms to use on a particular data set.

6.034 - Spring 03 • 6

Slide 4.3.6

One interesting thing to see is how well the learning process works as a function of how much data is available to it. I made a learning curve by secretly picking a target function to generate the data, drawing input points uniformly at random and generating their associated y using my secret function. First I generated 10 points and trained a hypothesis on them, then evaluated it. Then I did it for 20 points, and so on.

This is the learning curve for a fairly simply function: $f_{344} \wedge f_{991} \vee f_{103} \wedge f_{775}$, in a domain with 1000 features. The x axis is the amount of training data. The y axis is the number of errors made by the resulting hypothesis, on a test set of data that is drawn uniformly at random, as a percentage of the total size of the test set.

It takes surprisingly few samples before we find the correct hypothesis. With 1000 features, there are 2^{1000} possible examples. We are getting the correct hypothesis after seeing a tiny fraction of the whole space.

Learning Curves

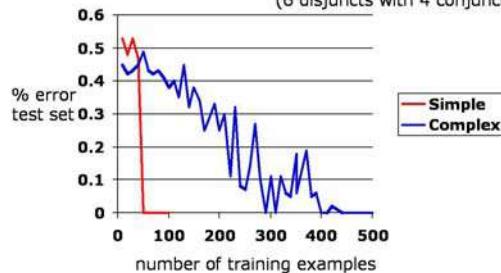
1000 input dimensions; function = $f_{344} \wedge f_{991} \vee f_{103} \wedge f_{775}$



6.034 - Spring 03 • 7

Learning Curves

1000 input dimensions; function = $(f_1 \wedge f_2 \wedge f_3 \wedge f_4) \vee (f_5 \wedge f_6 \wedge f_7 \wedge f_8) \vee \dots$
(6 disjuncts with 4 conjuncts each)

**Slide 4.3.8**

Now, let's look at a much more complex target function. It's $(f_1 \wedge f_2 \wedge f_3 \wedge f_4) \vee (f_5 \wedge f_6 \wedge f_7 \wedge f_8) \vee (f_9 \wedge f_{10} \wedge f_{11} \wedge f_{12}) \vee (f_{13} \wedge f_{14} \wedge f_{15} \wedge f_{16}) \vee (f_{17} \wedge f_{18} \wedge f_{19} \wedge f_{20}) \vee (f_{21} \wedge f_{22} \wedge f_{23} \wedge f_{24})$. It takes a lot more data to learn. Why is that?

6.034 - Spring 03 • 8

Slide 4.3.9

First, it's important to remember that, at every point in the graph, the learning algorithm has found a hypothesis that gets all of the training data correct.

Simple Gifts

- At every point in the graph, we've found an h that gets the whole training set correct (call it apparently correct)

6.034 - Spring 03 • 9

Simple Gifts

- At every point in the graph, we've found an h that gets the whole training set correct (call it apparently correct)
- When input dimensionality is high and size of training set is small, there are lots of apparently correct hypotheses
- Our algorithm tries to return (but doesn't always) the simplest apparently correct hypothesis

6.034 - Spring 03 • 10

Slide 4.3.11

So, when the target hypothesis is simple, we discover it quickly (the other simple hypotheses are ruled out with just a little bit of data, because there are so few of them).

Simple Gifts

- At every point in the graph, we've found an h that gets the whole training set correct (call it apparently correct)
- When input dimensionality is high and size of training set is small, there are lots of apparently correct hypotheses
- Our algorithm tries to return (but doesn't always) the simplest apparently correct hypothesis
- So, when the target hypothesis is simple, we discover it quickly (the other simple hypotheses are ruled out quickly because there are so few)

6.034 - Spring 03 • 11

Simple Gifts

- At every point in the graph, we've found an h that gets the whole training set correct (call it apparently correct)
- When input dimensionality is high and size of training set is small, there are lots of apparently correct hypotheses
- Our algorithm tries to return (but doesn't always) the simplest apparently correct hypothesis
- So, when the target hypothesis is simple, we discover it quickly (the other simple hypotheses are ruled out quickly because there are so few)
- When the target hypothesis is complex, it's hard to rule out all of the (many) other competitors, so it takes more data to learn

6.034 - Spring 03 • 12

Slide 4.3.12

If we thought our problems were going to have complex answers, then we might think about biasing our algorithm to prefer more complex answers. But then we start to see the wisdom of Ockham. There are lots more complex hypotheses than there are simple ones. So there will be huge numbers of complex hypotheses that agree with the data and no basis to choose among them, unless we know something about the domain that generated our data (maybe some features are thought, in advance, to be more likely to have an influence on the outcome, for example).

Slide 4.3.13

There is, in fact, a theorem in machine learning called the "No Free Lunch" theorem. It basically states that, unless you know something about the process that generated your data, any hypothesis that agrees with all the data you've seen so far is as good a guess as any other one.

The picture will get a bit more complicated when we look at data that are noisy.

No Free Lunch

- Unless you know something about the distribution of problems your learning algorithm will encounter, **any hypothesis that agrees with all your data is as good as any other.**
- You can't learn anything unless you already know something.

6.034 - Spring 03 • 13

Noisy Data

- Have to accept non-zero error on training data

Slide 4.3.14

It is actually very rare to encounter a machine learning problem in which there is a deterministic function that accounts for y's dependence on x in all cases. It is much more typical to imagine that the process that generates the x's and y's is probabilistic, possibly corrupting measurements of the features, or simply sometimes making different decisions in the same circumstances.

Noisy Data

- Have to accept non-zero error on training data
- Weaken DNF learning algorithm
 - Don't require the hypothesis to cover all positive examples
 - Don't require each rule to exclude all negative examples

6.034 - Spring 03 • 14

Pseudo Code: Noisy DNF Learning

```

P := the set of positive examples
h := False
np := epsilon * number of examples in P
nn := epsilon * number of examples in N
Loop until P has fewer than np elements
  r = True
  N = the set of negative examples
  Repeat until N has fewer than nn elements
    Select a feature fj to add to r
    r := x ^ fj
    N := N - examples in N for which fj = 0
  h := h v r
  P := P - elements in P covered by r
  allow epsilon percentage error

```

Slide 4.3.16

We can use our DNF learning algorithm almost unchanged. But, instead of requiring each rule to exclude **all** negative examples, and requiring the whole hypothesis to cover **all** positive examples, we will only require them to cover some percentage of them. Here is the pseudocode, with the changes highlighted in blue.

6.034 - Spring 03 • 16

Slide 4.3.17

We also have to add further conditions for stopping both loops. If there is no feature that can be added that reduces the size of N, then the inner loop must terminate. If the inner loop cannot generate a rule that reduces the size of P, then the outer loop must terminate and fail.

Pseudo Code: Noisy DNF Learning

```

P := the set of positive examples
h := False
np := epsilon * number of examples in P
nn := epsilon * number of examples in N
Loop until P has fewer than np elements or not progressing
    r = True
    N = the set of negative examples
    Repeat until N has fewer than nn elements or not progress
        Select a feature fj to add to r
        r := r ^ fj
        N := N - examples in N for which fj = 0
    h := h v r
    P := P - elements in P covered by r

```

allow epsilon
percentage error
Handle failure to
progress

6.034 - Spring 03 • 17

Epsilon is our Delta

- Parameter epsilon is the percentage error allowed

Slide 4.3.18

Now we have a new parameter, epsilon, which is the percentage of examples we're allowed to get wrong. (Well, we'll actually get more wrong, because each disjunct is allowed to get epsilon percent of the negatives wrong).

Epsilon is our Delta

- Parameter epsilon is the percentage error allowed
- The higher epsilon, the simpler and more error-prone the hypothesis

Slide 4.3.19

Although it is not exactly equivalent to manipulating the alpha parameter in our ideal criterion, it is a way to trade off accuracy for complexity. The higher the epsilon, the simpler and more error-prone our hypothesis.

Epsilon is our Delta

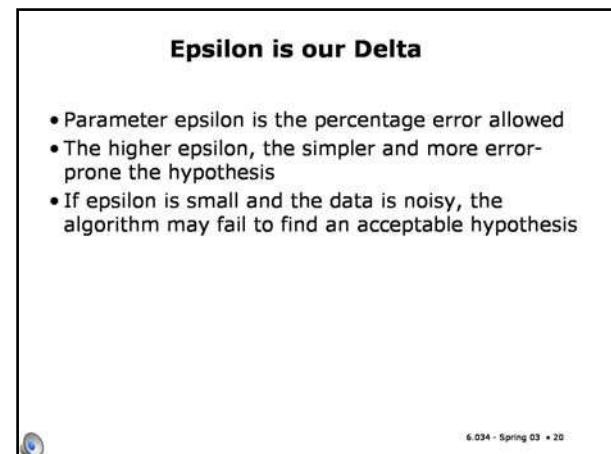
- Parameter epsilon is the percentage error allowed
- The higher epsilon, the simpler and more error-prone the hypothesis
- If epsilon is small and the data is noisy, the algorithm may fail to find an acceptable hypothesis

Slide 4.3.20

If epsilon is small and the data is noisy, the algorithm may fail to find an acceptable hypothesis.



6.034 - Spring 03 • 18



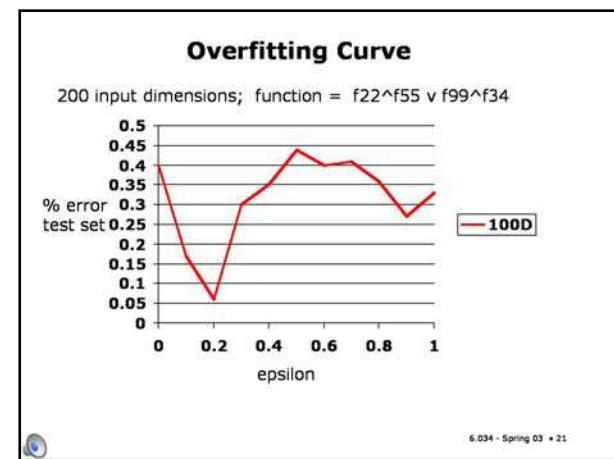
6.034 - Spring 03 • 19



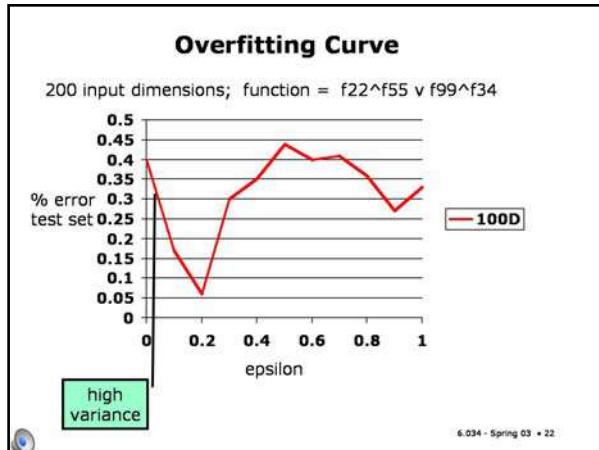
6.034 - Spring 03 • 20

Slide 4.3.21

Let's see what happens in a moderately noisy domain. This curve shows the problem of learning our easy DNF function in a domain with 100 features, on a training set of size 100. The data has 10 percent noise, which means that, in 10 percent of the cases, we expect the output set to be the opposite of the one specified by the target function. The x axis is epsilon, ranging from 0 to 1. At setting 0, it is the same as our original algorithm. At setting 1, it's willing to make an arbitrary number of errors, and will always return the hypothesis False. The y axis is percentage error on a freshly drawn set of testing data. Note that, because there is 10 percent error in the data (training as well as testing), our very best hope is to generate a hypothesis that has 10 percent error on average on testing data.



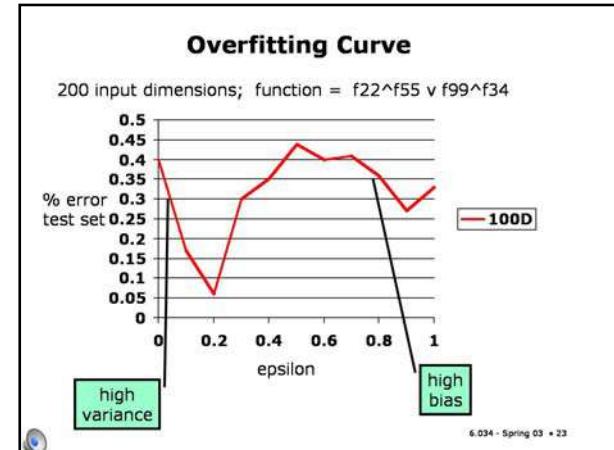
6.034 - Spring 03 • 21

**Slide 4.3.22**

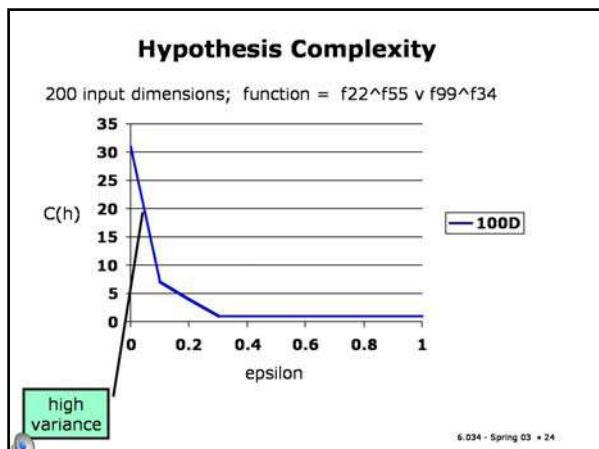
Of course, turning epsilon up too high will keep us from building a hypothesis of sufficient complexity. Then we'd do poorly because we are unable to even represent the right answer. It is said, in this case, that our algorithm has high **bias**.

Slide 4.3.22

We get the best performance for a value of epsilon that's between 0 and 1. What's the matter with setting it to some value near 0? We run into the problem of **overfitting**. In overfitting, we work very hard to reduce the error of our hypothesis on the training set. But we may have spent a lot of effort modeling the noise in the data, and perform poorly on the test set. That's what is happening here. It is said, in this case, that our algorithm has high **variance**. Another way to think about the problem is to see that if we get another noisy data set generated by the same target function, we are likely to get a wildly different answer. By turning epsilon up a bit, we generate simpler hypotheses that are not so susceptible to variations in the data set, and so can get better generalization performance.



6.034 - Spring 03 • 23

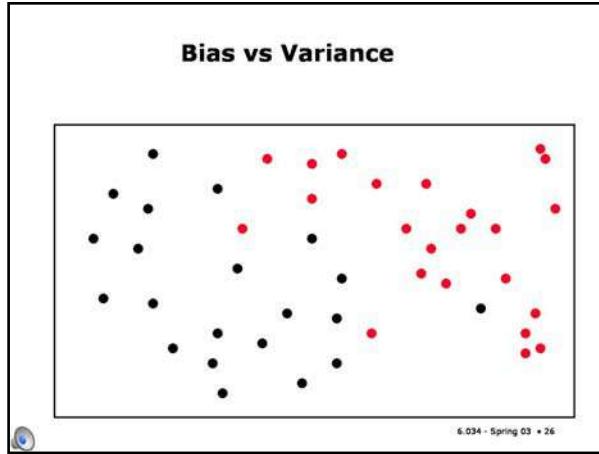
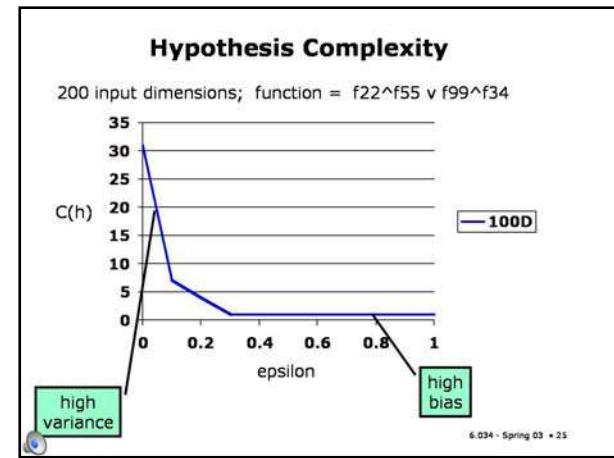
**Slide 4.3.24**

It's also instructive to see how the complexity of the resulting hypothesis varies as a function of epsilon. When epsilon is 0, we are asking for a hypothesis with zero error on the training set. We are able to find one, but it's very complex (31 literals). This is clearly a high variance situation; that hypothesis is completely influenced by the particular training set and would change radically on a newly drawn training set (and therefore, have high error on a testing set, as we saw on the previous slide).

6.034 - Spring 03 • 24

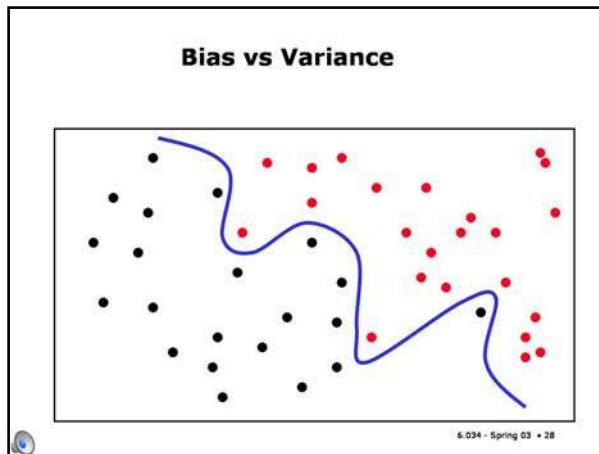
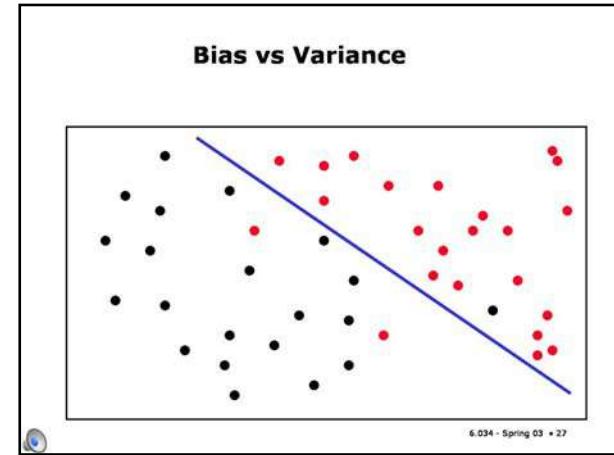
Slide 4.3.25

As epsilon increases, we rapidly go down to a complexity of 1, which is incapable of representing the target hypothesis. The low-point in the error curve was at epsilon = 0.2, which has a complexity of 4 here. And it's no coincidence that the target concept also has a complexity of 4.

**Slide 4.3.27**

Here's some noisy data.

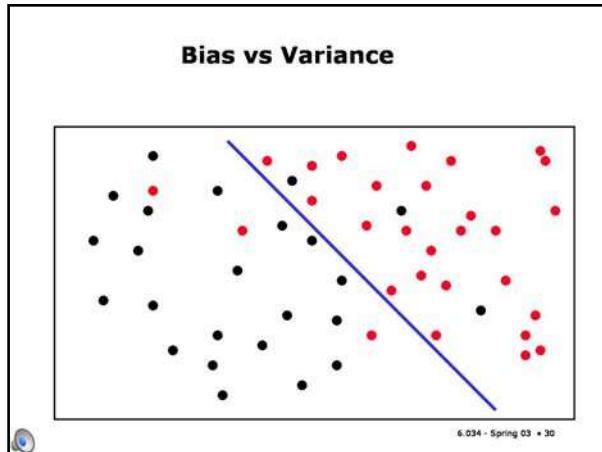
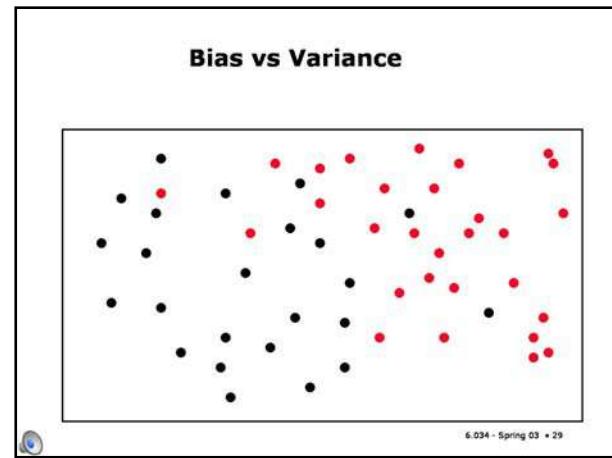
If we separate it with a line, we have low complexity, but lots of error.

**Slide 4.3.28**

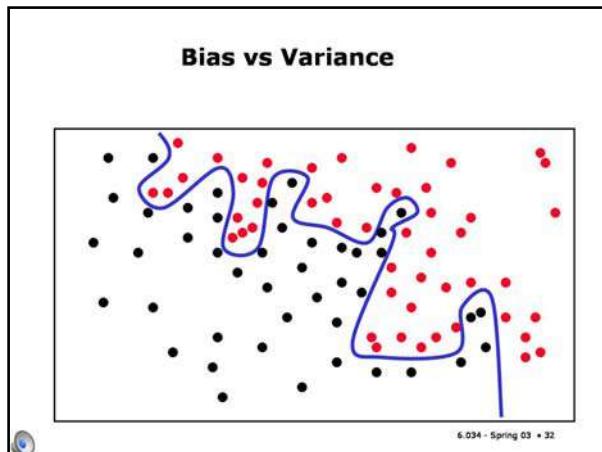
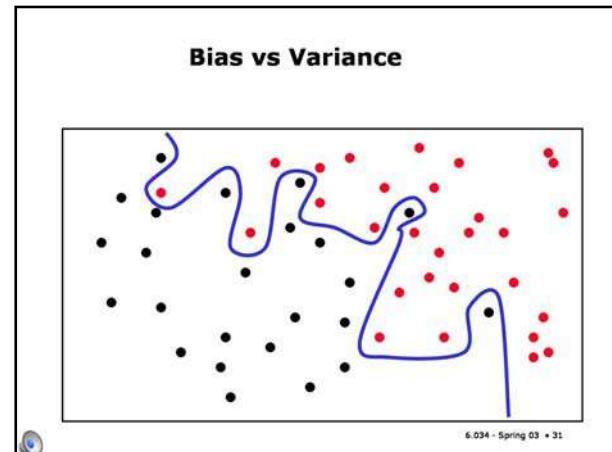
If we separate it with a squiggly snake, we can reduce the error but add a lot of complexity.

Slide 4.3.29

So, what, exactly, is so bad about complexity? Well, let's draw a bunch more points from the same underlying distribution and add them to our data set.

**Slide 4.3.30**

We wouldn't need to change our line hypothesis very much to do our best to accomodate them.

**Slide 4.3.32**

In practice, it is common to allow hypotheses of increased complexity as the amount of available training data increases. So, in this situation, we might feel justified in choosing a moderately complex hypothesis.

Slide 4.3.33

So, given a data set, how should we choose epsilon? This is where cross-validation can come in handy. You select a number of different values of epsilon, and use cross-validation to estimate the error when using that epsilon. Then, you pick the epsilon value with the least estimated error, and train your learning algorithm with all the available data using that value of epsilon. The resulting hypothesis is probably the best you can do (given what you know).

Picking Epsilon

- Pick epsilon using cross validation
- Try multiple different values of epsilon
- See which gives the lowest cross-validation error
- Use that value of epsilon to learn a hypothesis from the whole training set
- Return that hypothesis as your best answer

6.034 - Spring 03 • 33

Domains**Slide 4.3.34**

So, is this algorithm actually good for anything? Yes. There are lots of domains that can be encoded as vectors of binary features with a binary classification. Here are some examples.

6.034 - Spring 03 • 34

Slide 4.3.35

Congressional voting: given a congressperson's voting record, where the features are the individual's votes on various bills, can you predict whether they are a republican or democrat?

Domains

- Congressional voting: given a congressperson's voting record (list of 1s and 0s), predict party

6.034 - Spring 03 • 35

Domains**Slide 4.3.36**

Gene splicing: given a sequence of bases (A, C, G, or T), predict whether the mid-point is a splice junction between a coding segment of DNA and a non-coding segment. In order to apply our algorithms to this problem, we would have to change the representation somewhat. It would be easy to take each base and represent it as two binary values; A is 00, C is 01, G is 10 and T is 11. Even though it seems inefficient, for learning purposes, it is often better to represent discrete elements using a unary code; so A would be 0001, C would be 0010, G would be 0100 and T would be 1000.

6.034 - Spring 03 • 36

Slide 4.3.37

Spam filtering: is this email message spam? Represent a document using a really big feature space, with one feature for each possible word in the dictionary (leaving out some very common ones, like "a" and "the"). The feature for a word has value 1 if that word exists in the document and 0 otherwise.

Domains

- Congressional voting: given a congressperson's voting record (list of 1s and 0s), predict party
- Gene splice: predict the beginning of a coding section of the genome; input is vector of elements chosen from the set {ACGT}; encode each element with one bit (or possibly with 4)
- Spam filtering: encode every message as a vector of features, one per word; a feature is on if that word occurs in the message; predict whether or not the message is spam

6.034 - Spring 03 • 37

Domains

- Congressional voting: given a congressperson's voting record (list of 1s and 0s), predict party
- Gene splice: predict the beginning of a coding section of the genome; input is vector of elements chosen from the set {ACGT}; encode each element with one bit (or possibly with 4)
- Spam filtering: encode every message as a vector of features, one per word; a feature is on if that word occurs in the message; predict whether or not the message is spam
- Marketing: predict whether a person will buy beer based on previous purchases; encode buying habits with a feature for all products, set to 1 if previously purchased

6.034 - Spring 03 • 38

Slide 4.3.39

Just for fun, we ran the DNF learning algorithm on the congressional voting data. A lot of the records had missing data (those congress folks are always out in the hallways talking instead of in there voting); we just deleted those records (though in a machine learning class we would study better ways of handling missing data items). There were 232 training examples, each of which had 15 features. The features were the individuals' votes on these bills (unfortunately, we don't know any details about the bills, but the names are evocative).

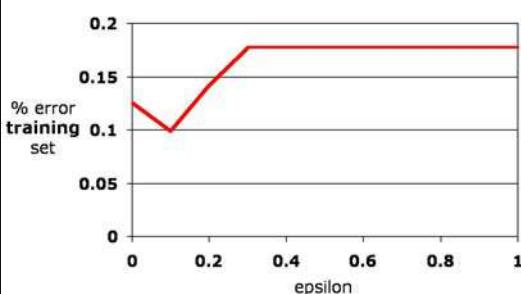
Slide 4.3.38

Marketing: is this grocery-store customer likely to buy beer? Represent a person's buying history using a big feature space, with one feature for each product in the supermarket. The feature for a product has value 1 if the person has ever bought that product and 0 otherwise.

Congressional Voting

- | | |
|---|-----------------|
| 0. handicapped-infants
1. water-project-cost-sharing
2. adoption-of-the-budget-resolution
3. physician-fee-freeze
4. el-salvador-aid
5. religious-groups-in-schools
6. anti-satellite-test-ban
7. aid-to-nicaraguan-contras
8. mx-missile
9. immigration
10. synfuels-corporation-cutback
11. education-spending
12. superfund-right-to-sue
13. crime
14. duty-free-exports
15. export-administration-act-south-africa | 232 data points |
|---|-----------------|

6.034 - Spring 03 • 39

Congressional Voting: Republican?

6.034 - Spring 03 • 40

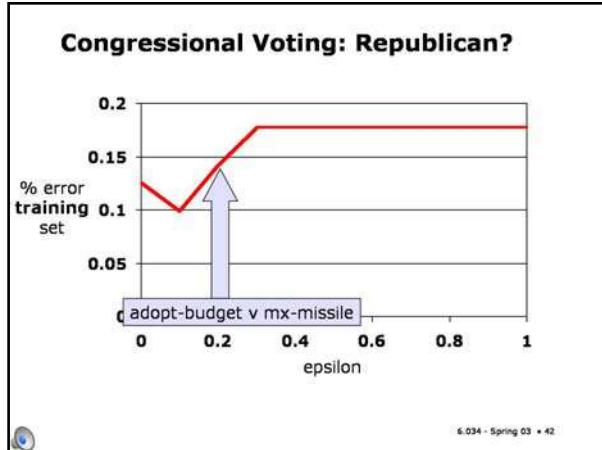
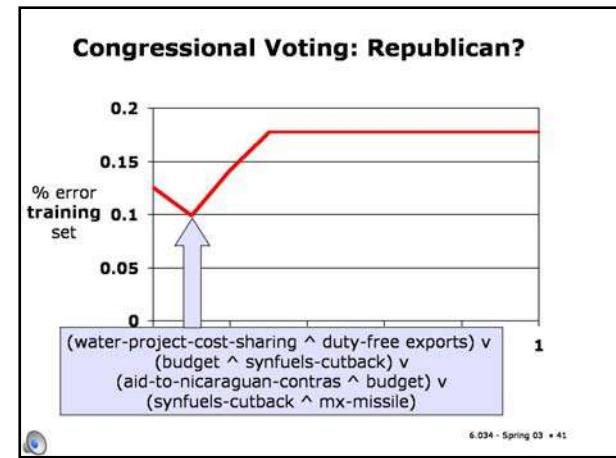
Slide 4.3.40

We are trying to predict whether a person is a Republican, based on their voting history during part of 1984. Here is a plot of the **training set error** as a function of epsilon. What I should really do is cross-validation, but instead I just ran the algorithm on the whole data set and I'm plotting the error on the training set.

Something sort of weird is going on. Training set error should get smaller as epsilon decreases. Why do we have it going up for epsilon equal zero? It's because of the greediness of our algorithm. We're not actually getting the best hypothesis in that case.

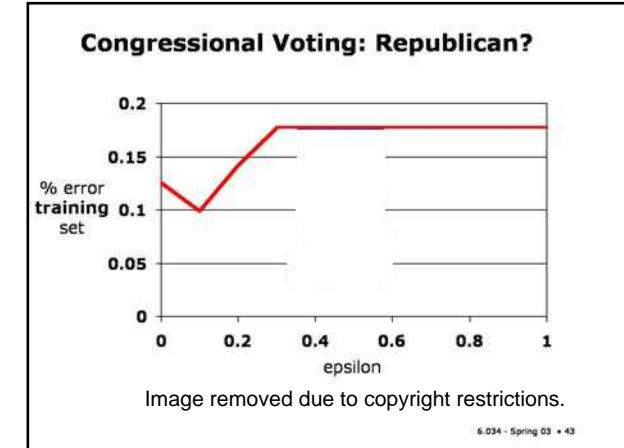
Slide 4.3.41

Here's the hypothesis we get at this point. It's kind of complicated (but mentions a lot of Republican things, like aid to the Nicaraguan contras and the mx-missile).

**Slide 4.3.42**

And here's a simpler hypothesis (with slightly worse error).

And, just because you're all too young to remember this, a little history. The MX missile, also called the "Peacekeeper" was our last big nuclear missile project. It's 71 feet long, with 10 warheads, and launched from trains. It was politically very contentious. Congress killed it at least twice before it was finally approved.



6.034 Notes: Section 5.1

Slide 5.1.1

The learning algorithm for DNF that we saw last time is a bit complicated and can be inefficient. It's also not clear that we're making good decisions about which attributes to add to a rule, especially when there's noise.

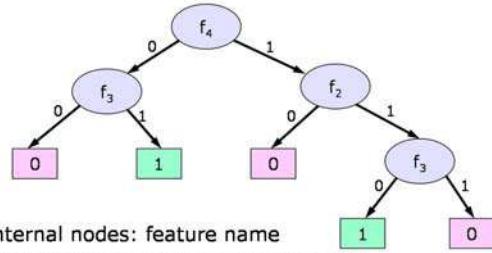
So now we're going to look at an algorithm for learning decision trees. We'll be changing our hypothesis class (sort of), our bias, and our algorithm. We'll continue to assume, for now, that the input features and the output class are boolean. We'll see later that this algorithm can be applied much more broadly.

Decision Trees

- DNF learning algorithm is a bit cumbersome and inefficient. Also, the exact effect of the heuristic is unclear.
- Still assume binary inputs and output, but much more broadly applicable.

6.034 - Spring 03 • 1

Hypothesis Class



- Internal nodes: feature name
- One child for each value of the feature
- Leaf nodes: output

6.034 - Spring 03 • 2

Slide 5.1.2

Our hypothesis class is going to be decision trees. A decision tree is a tree (big surprise!). At each internal (non-leaf) node, there is the name of a feature. There are two arcs coming out of each node, labeled 0 and 1, standing for the two possible values that the feature can take on.

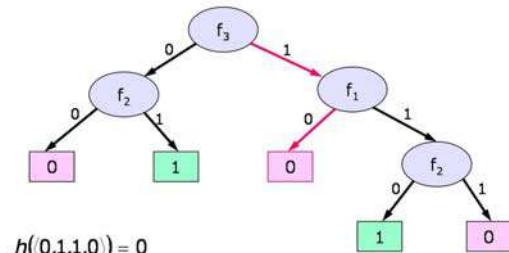
Leaf nodes are labeled with decisions, or outputs. Since our y's are Boolean, the leaves are labeled with 0 or 1.

Slide 5.1.3

Trees represent Boolean functions from x's (vectors of feature values) to Booleans. To compute the output for a given input vector x^i , we start at the root of the tree. We look at the feature there, let's say it's feature j, and then look to see what the value of x_j^i is. Then we go down the arc corresponding to that value. If we arrive at another internal node, we look up that feature value, follow the correct arc, and so on. When we arrive at a leaf node, we take the label we find there and generate that as an output.

So, in this example, input [0 1 1 0] would generate an output of 0 (because the third element of the input has value 1 and the first has value 0, which takes to a leaf labeled 0).

Hypothesis Class



6.034 - Spring 03 • 3

Hypothesis Class

$$h = (\neg f_3 \wedge f_2) \vee (f_3 \wedge f_1 \wedge \neg f_2)$$

6.034 - Spring 03 • 4

Slide 5.1.4

Decision trees are a way of writing down Boolean expressions. To convert a tree into an expression, you can make each branch of the tree with a 1 at the leaf node into a conjunction, describing the condition that had to hold of the input in order for you to have gotten to that leaf of the tree. Then, you take this collection of expressions (one for each leaf) and disjoin them.

So, our example tree describes the boolean function **not f₃ and f₂ or f₃ and f₁ and not f₂**.

Tree Bias

- Both decision trees and DNF with negation can represent any Boolean function. So why bother with trees?
- Because we have a nice algorithm for growing trees that is consistent with a bias for simple trees (few nodes)

6.034 - Spring 03 • 5

Tree Bias

- Both decision trees and DNF with negation can represent any Boolean function. So why bother with trees?
- Because we have a nice algorithm for growing trees that is consistent with a bias for simple trees (few nodes)
- Too hard to find the smallest good tree, so we'll be greedy again
- Have to watch out for overfitting

6.034 - Spring 03 • 6

Slide 5.1.6

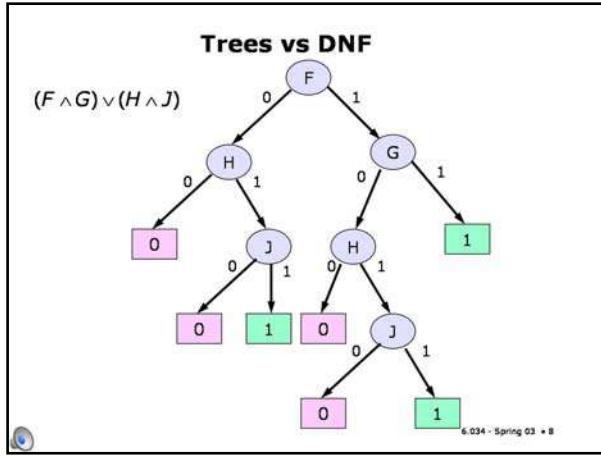
Application of Ockham's razor will lead us to prefer trees that are small, measured by the number of nodes. As before, it will be computationally intractable to find the minimum-sized tree that satisfies our error criteria. So, as before, we will be greedy, growing the tree in a way that seems like it will make it best on each step.

We'll consider a couple of methods for making sure that our resulting trees are not too large, so we can guard against overfitting.

Trees vs DNF

$$(\neg F \wedge \neg H) \vee (\neg F \wedge H \wedge J) \vee (F \wedge \neg G \wedge K) \vee (F \wedge G)$$

6.034 - Spring 03 • 7

**Slide 5.1.8**

But here's a case with a very simple DNF expression that requires a large tree to represent it. There's no particular reason to prefer trees over DNF or DNF over trees as a hypothesis class. But the tree-growing algorithm is simple and elegant, so we'll study it.

Slide 5.1.9

The idea of learning decision trees and the algorithm for doing so was, interestingly, developed independently by researchers in statistics and researchers in AI at about the same time around 1980.

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

6.034 - Spring 03 • 9

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

`BuildTree (Data)`**Slide 5.1.10**

We will build the tree from the top down. Here is pseudocode for the algorithm. It will take as input a data set, and return a tree.

6.034 - Spring 03 • 10

Slide 5.1.11

We first test to see if all the data elements have the same y value. If so, we simply make a leaf node with that y value and we're done. This is the base case of our recursive algorithm.

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

```
BuildTree (Data)
  if all elements of Data have the same y value, then
    MakeLeafNode(y)
```

6.034 - Spring 03 • 11

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

```
BuildTree (Data)
  if all elements of Data have the same y value, then
    MakeLeafNode(y)
  else
    feature := PickBestFeature(Data)
    MakeInternalNode(feature,
      BuildTree(SelectFalse(Data, feature)),
      BuildTree(SelectTrue(Data, feature)))
```

6.034 - Spring 03 • 12

Slide 5.1.12

If we have a mixture of different y values, we choose a feature to use to make a new internal node. Then we divide the data into two sets, those for which the value of the feature is 0 and those for which the value is 1. Finally, we call the algorithm recursively on each of these data sets. We use the selected feature and the two recursively created subtrees to build our new internal node.

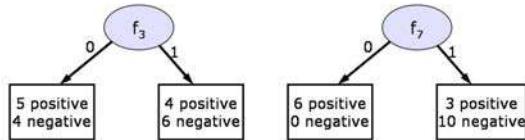
Let's Split

D: 9 positive
10 negative

6.034 - Spring 03 • 13

Let's Split

D: 9 positive
10 negative



6.034 - Spring 03 • 14

Slide 5.1.14

In this example, it looks like the split based on f_7 will be more helpful. To formalize that intuition, we need to develop a measure of the degree of uniformity of the subsets of the data we'd get by splitting on a feature.

Entropy

p : proportion of positive examples in a data set

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

6.034 - Spring 03 • 15

Slide 5.1.15

We'll start by looking at a standard measure of disorder, used in physics and information theory, called **entropy**. We'll just consider it in the binary case, for now. Let p be the proportion of positive examples in a data set (that is, the number of positive examples divided by the total number of examples). Then the entropy of that data set is

$$-p \log_2 p - (1-p) \log_2 (1-p)$$

Entropy

p : proportion of positive examples in a data set

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

$0 \log_2 0 = 0$

6.034 - Spring 03 • 16

Slide 5.1.16

Here's a plot of the entropy as a function of p . When p is 0 or 1, then the entropy is 0. We have to be a little bit careful here. When p is 1, then $1 \log 1$ is clearly 0. But what about when p is 0? Log 0 is negative infinity. But 0 wins the battle, so $0 \log 0$ is also 0.

So, when all the elements of the set have the same value, either 0 or 1, then the entropy is 0. There is no disorder (unlike in my office!).

The entropy function is maximized when $p = 0.5$. When p is one half, the set is as disordered as it can be. There's no basis for guessing what the answer might be.

Slide 5.1.17

When we split the data on feature j , we get two data sets. We'll call the set of examples for which feature j has value 1 D_j^+ , and those for which j has value 0 D_j^- .

Let's Split

D: 9 positive
10 negative

6.034 - Spring 03 • 17

Let's Split

D: 9 positive
10 negative

6.034 - Spring 03 • 18

Slide 5.1.18

We can compute the entropy for each of these subsets. For some crazy reason, people usually use the letter H to stand for entropy. We'll follow suit.

Let's Split

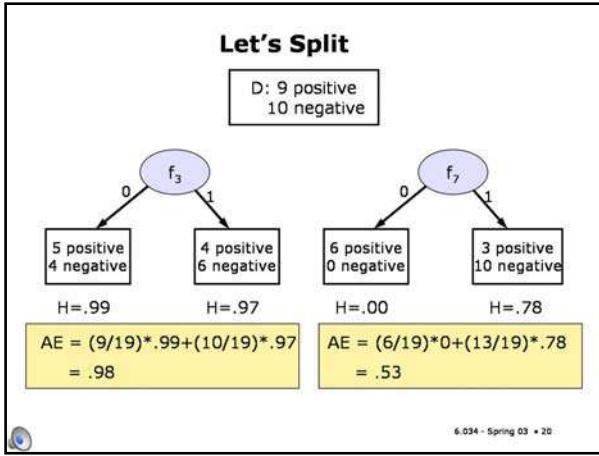
D: 9 positive
10 negative

6.034 - Spring 03 • 19

Slide 5.1.19

Now, we have to figure out how to combine these two entropy values into a measure of how good splitting on feature j is. We could just add them together, or average them. But what if have this situation, in which there's one positive example in one data set and 100 each of positive and negative examples in the other? It doesn't really seem like we've done much good with this split, but if we averaged the entropies, we'd get a value of 1/4.

So, to keep things fair, we'll use a weighted average to combine the entropies of the two sets. Let p_j be the proportion of examples in the data set D for which feature j has value 1. We'll compute a weighted **average entropy** for splitting on feature j as $AE(j) = p_j H(D_j^-) + (1 - p_j) H(D_j^+)$

**Slide 5.1.20**

So, in this example, we can see that the split on f_7 is much more useful, as reflected in the average entropy of its children.

Slide 5.1.21

Going back to our algorithm, then, we'll pick the feature at every step that minimizes the weighted average entropy of the children.

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olshen and Stone

```

BuildTree (Data)
  if all elements of Data have the same y value, then
    MakeLeafNode(y)
  else
    feature := PickBestFeature(Data)
    MakeInternalNode(feature,
      BuildTree(SelectFalse(Data, feature)),
      BuildTree(SelectTrue(Data, feature)))
  
```

- Best feature minimizes average entropy of data in the children

6.034 - Spring 03 • 21

Stopping

Image removed due to copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values

Slide 5.1.22

As usual, when there is noise in the data, it's easy to overfit. We could conceivably grow the tree down to the point where there's a single data point in each leaf node. (Or maybe not: in fact, if have two data points with the same x values but different y values, our current algorithm will never terminate, which is certainly a problem). So, at the very least, we have to include a test to be sure that there's a split that makes both of the data subsets non-empty. If there is not, we have no choice but to stop and make a leaf node.

Slide 5.1.23

What should we do if we have to stop and make a leaf node when the data points at that node have different y values? Choosing the majority y value is the best strategy. If there are equal numbers of positive and negative points here, then you can just pick a y arbitrarily.

Stopping

Image removed due to copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
 - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data

6.034 - Spring 03 • 23

Stopping

Image removed due to
copyright restrictions

- Stop recursion if data contains only multiple instances of the same x with different y values
 - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:

6.034 - Spring 03 • 24

Slide 5.1.24

But growing the tree as far down as we can will often result in overfitting.

Slide 5.1.25

The simplest solution is to change the test on the base case to be a threshold on the entropy. If the entropy is below some value epsilon, we decide that this leaf is close enough to pure.

Stopping

Image removed due to
copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
 - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:
 - entropy of a data set is below some threshold
 - number elements in a data set is below threshold

6.034 - Spring 03 • 26

Slide 5.1.26

Another simple solution is to have a threshold on the size of your leaves; if the data set at some leaf has fewer than that number of elements, then don't split it further.

Slide 5.1.27

Another possible method is to only split if the split represents a real improvement. We can compare the entropy at the current node to the average entropy for the best attribute. If the entropy is not significantly decreased, then we could just give up.

Stopping

Image removed due to
copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
 - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:
 - entropy of a data set is below some threshold

6.034 - Spring 03 • 25

Stopping

Image removed due to
copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
 - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:
 - entropy of a data set is below some threshold
 - number elements in a data set is below threshold
 - best next split doesn't decrease average entropy (but this can get us into trouble)

6.034 - Spring 03 • 27

Simulation

- $H(D) = .92$
- $AE_1=.92, AE_2=.92, AE_3=.81, AE_4=1$

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
1	0	0	1	0
0	1	1	1	0

6.034 - Spring 03 • 28

Slide 5.1.28

Let's see how our tree-learning algorithm behaves on the example we used to demonstrate the DNF-learning algorithm. Our data set has a starting entropy of .92. Then, we can compute, for each feature, what the average entropy of the children would be if we were to split on that feature.

In this case, the best feature to split on is f_3 .

Simulation

f_1	f_2	f_3	f_4	y
1	0	0	1	0
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
0	1	1	1	0

6.034 - Spring 03 • 30

Slide 5.1.30

So we make it into a leaf with output 0 and consider splitting the data set in the right child.

Simulation

f_1	f_2	f_3	f_4	y
1	0	0	1	0
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
0	1	1	1	0

6.034 - Spring 03 • 29

Slide 5.1.31

The average entropies of the possible splits are shown here. Features 1 and 2 are equally useful, and feature 4 is basically no help at all.

Simulation

- $AE_1=.55, AE_2=.55, AE_4=.95$

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	1	1
1	1	1	0	1
0	0	1	1	1
0	1	1	1	0

6.034 - Spring 03 • 31

Simulation

```

graph TD
    f3((f3)) -- 0 --> leaf0[0]
    f3 -- 1 --> f1((f1))
    f1 -- 0 --> subf1_0[f1 f2 f3 f4 y<br/>0 1 1 0 0]
    f1 -- 1 --> subf1_1[f1 f2 f3 f4 y<br/>1 0 1 1 1]
    subf1_0 --> leaf0
    subf1_1 --> leaf1[1]
  
```

6.034 - Spring 03 • 32

Slide 5.1.32

So, we decide, arbitrarily, to split on feature 1, yielding these sub-problems. All of the examples in the right-hand child have the same output.

Simulation

```

graph TD
    f3((f3)) -- 0 --> leaf0[0]
    f3 -- 1 --> f1((f1))
    f1 -- 0 --> subf1_0[f1 f2 f3 f4 y<br/>0 1 1 0 0]
    f1 -- 1 --> leaf1[1]
    subf1_0 --> leaf0
  
```

6.034 - Spring 03 • 33

Simulation

```

graph TD
    f3((f3)) -- 0 --> leaf0[0]
    f3 -- 1 --> f1((f1))
    f1 -- 0 --> f2((f2))
    f1 -- 1 --> leaf1[1]
    f2 -- 0 --> subf2_0[f1 f2 f3 f4 y<br/>0 0 1 1 1]
    f2 -- 1 --> subf2_1[f1 f2 f3 f4 y<br/>0 1 1 0 0]
    subf2_0 --> leaf0
    subf2_1 --> leaf1
  
```

6.034 - Spring 03 • 34

Slide 5.1.34

So we split on it, and now both children are homogeneous (all data points have the same output).

Simulation

```

graph TD
    f3((f3)) -- 0 --> leaf0[0]
    f3 -- 1 --> f1((f1))
    f1 -- 0 --> f2((f2))
    f1 -- 1 --> leaf1[1]
    f2 -- 0 --> leaf2[1]
    f2 -- 1 --> leaf3[0]
  
```

6.034 - Spring 03 • 35

Slide 5.1.35

So we make the leaves and we're done!

Exclusive OR

$$(A \wedge \neg B) \vee (\neg A \wedge B)$$

Slide 5.1.36

One class of functions that often haunts us in machine learning are those that are based on the "exclusive OR". Exclusive OR is the two-input boolean function that has value 1 if one input has value 1 and the other has value 0. If both inputs are 1 or both are 0, then the output is 0. This function is hard to deal with, because neither input feature is, by itself, detectably related to the output. So, local methods that try to add one feature at a time can be easily misled by xor.

6.034 - Spring 03 • 36

Slide 5.1.37

Let's look at a somewhat tricky data set. The data set has entropy .92. Furthermore, no matter what attribute we split on, the average entropy is .92. If we were using the stopping criterion that says we should stop when there is no split that improves the average entropy, we'd stop now.

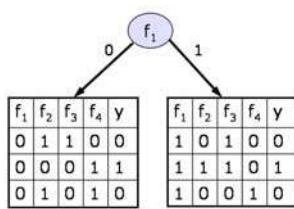
Exclusive OR

$$(A \wedge \neg B) \vee (\neg A \wedge B)$$

- $H(D) = .92$
- $AE_1 = .92, AE_2 = .92,$
 $AE_3 = .92, AE_4 = .92$

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	1	1	0	1
0	0	0	1	1
1	0	0	1	0
0	1	0	1	0

6.034 - Spring 03 • 37

Exclusive OR

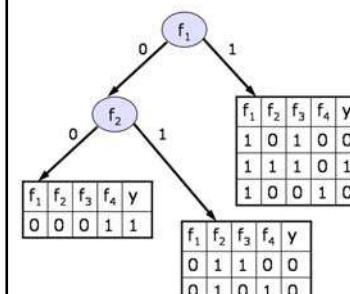
6.034 - Spring 03 • 38

Slide 5.1.38

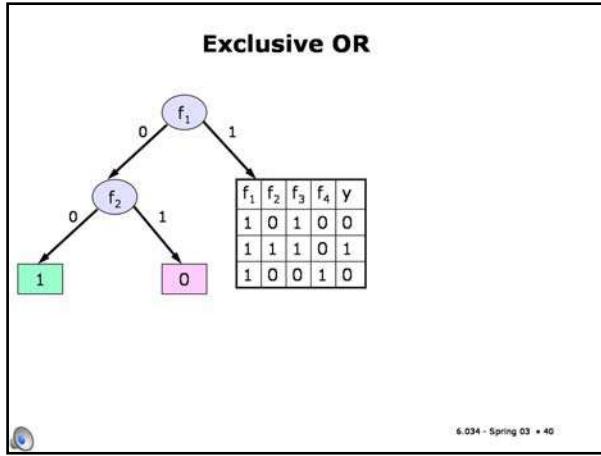
But let's go ahead and split on feature 1. Now, if we look at the left-hand data set, we'll see that feature 2 will have an average entropy of 0.

Slide 5.1.39

So we split on it, and get homogenous children,

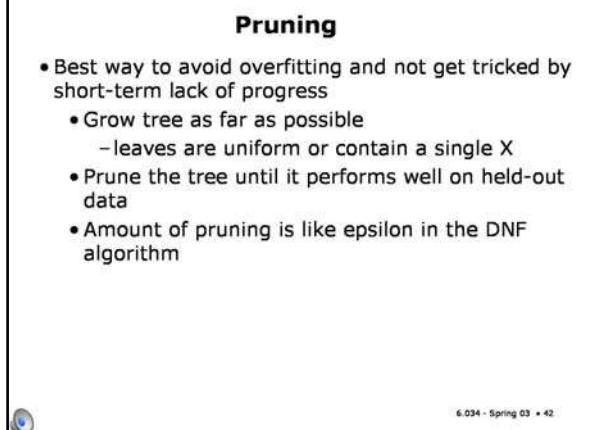
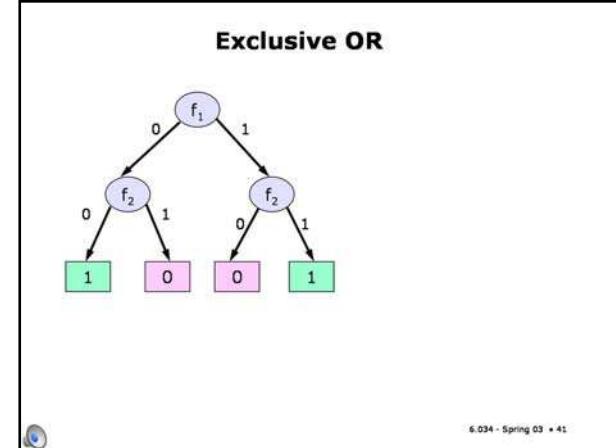
Exclusive OR

6.034 - Spring 03 • 39

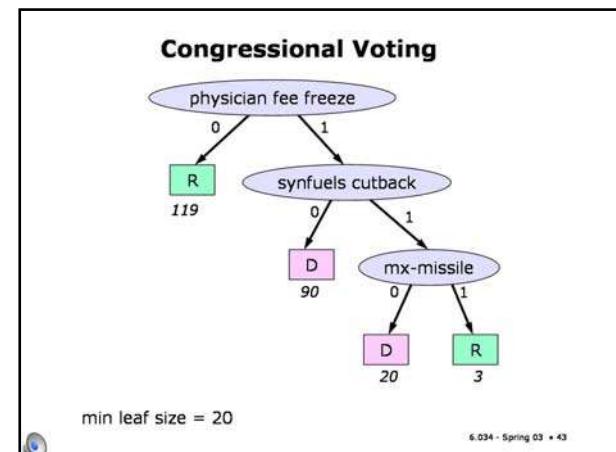
**Slide 5.1.40**

Which we can replace with leaves.

Now, it's also easy to see that feature 2 will again be useful here,

**Slide 5.1.42**

Most real decision-tree building systems avoid this problem by building the tree down significantly deeper than will probably be useful (using something like an entropy cut-off or even getting down to a single data-point per leaf). Then, they prune the tree, using cross-validation to decide what an appropriate pruning depth is.



Data Mining

- Making useful predictions in (usually corporate) applications
- Decision trees very popular because
 - easy to implement
 - efficient (even on huge data sets)
 - easy for humans to understand resulting hypotheses

Slide 5.1.44

Because decision tree learning is easy to implement and relatively computationally efficient, and especially because the hypotheses are easily understandable by humans, this technology is very widely used.

The field of data mining is the application of learning algorithms to problems of interest in many industries. Decision trees is one of the principle methods used in data mining.

In the next few sections, we'll see how to broaden the applicability of this and other algorithms to domains with much richer kinds of inputs and outputs.

6.034 - Spring 03 • 44

6.034 Notes: Section 5.2

Slide 5.2.1

Let's look at one more algorithm, which is called naive Bayes. It's named after the Reverend Thomas Bayes, who developed a very important theory of probabilistic reasoning.

Naïve Bayes

- Founded on Bayes' rule for probabilistic inference
- Update probability of hypotheses based on evidence
- Choose hypothesis with the maximum probability after the evidence has been incorporated

Image of Rev. Thomas Bayes removed due to copyright restrictions.

Rev. Thomas Bayes

6.034 - Spring 03 • 1

Naïve Bayes

- Founded on Bayes' rule for probabilistic inference
- Update probability of hypotheses based on evidence
- Choose hypothesis with the maximum probability after the evidence has been incorporated
- Algorithm is particularly useful for domains with **lots** of features

Image of Rev. Thomas Bayes removed due to copyright restrictions.

Rev. Thomas Bayes

Slide 5.2.2

It's widely used in applications with lots of features. It was derived using a somewhat different set of justifications than the ones we've given you. We'll start by going through the algorithm, and at the end I'll go through its probabilistic background. Don't worry if you don't follow it exactly. It's just motivational, but it should make sense to anyone who has studied basic probability.

6.034 - Spring 03 • 2

Slide 5.2.3

Let's start by looking at an example data set. We're going to try to characterize, for each feature individually, how it is related to the class of an example.

First, we look at the positive examples, and count up what fraction of them have feature 1 on and what fraction have feature 1 off. We'll call these fractions $R_1(1, 1)$ and $R_1(0, 1)$. We can see here that most positive examples have this feature 1 off.

f ₁	f ₂	f ₃	f ₄	y
0	1	1	0	1
0	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	0	1	0
1	0	0	1	0
1	1	0	1	0
1	0	0	0	0
1	1	0	1	0
1	0	1	1	0

Example

- $R_1(1, 1) = 1/5$: fraction of all **positive** examples that have feature 1 **on**
- $R_1(0, 1) = 4/5$: fraction of all **positive** examples that have feature 1 **off**
- $R_1(1, 0) = 5/5$: fraction of all **negative** examples that have feature 1 **on**
- $R_1(0, 0) = 0/5$: fraction of all **negative** examples that have feature 1 **off**

6.034 - Spring 03 • 4

Slide 5.2.4

We can compute these values, as shown here, for each of the other features, as well.

f ₁	f ₂	f ₃	f ₄	y
0	1	1	0	1
0	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	0	1	0
1	0	0	1	0
1	1	0	1	0
1	0	0	0	0
1	1	0	1	0
1	0	1	1	0

Example

- $R_1(1, 1) = 1/5$: fraction of all **positive** examples that have feature 1 **on**
- $R_1(0, 1) = 4/5$: fraction of all **positive** examples that have feature 1 **off**

6.034 - Spring 03 • 3

Slide 5.2.5

Now, we look at the negative examples, and figure out what fraction of negative examples have feature 1 on and what fraction have it off. We call these fractions $R_1(1, 0)$ and $R_1(0, 0)$. Here we see that **all** negative examples have this feature on.

f ₁	f ₂	f ₃	f ₄	y
0	1	1	0	1
0	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	0	1	0
1	0	0	1	0
1	1	0	1	0
1	0	0	0	0
1	1	0	1	0
1	0	1	0	0

6.034 - Spring 03 • 4

f ₁	f ₂	f ₃	f ₄	y
0	1	1	0	1
0	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	0	1	0
1	0	0	1	0
1	1	0	1	0
1	0	0	0	0
1	1	0	1	0
1	0	1	0	0

Example

- $R_1(1, 1) = 1/5$ $R_1(0, 1) = 4/5$
- $R_1(1, 0) = 5/5$ $R_1(0, 0) = 0/5$
- $R_2(1, 1) = 1/5$ $R_2(0, 1) = 4/5$
- $R_2(1, 0) = 2/5$ $R_2(0, 0) = 3/5$
- $R_3(1, 1) = 4/5$ $R_3(0, 1) = 1/5$
- $R_3(1, 0) = 1/5$ $R_3(0, 0) = 4/5$
- $R_4(1, 1) = 2/5$ $R_4(0, 1) = 3/5$
- $R_4(1, 0) = 4/5$ $R_4(0, 0) = 1/5$

6.034 - Spring 03 • 5

Prediction				
$R_1(1, 1) = 1/5$	$R_1(0, 1) = 4/5$			
$R_1(1, 0) = 5/5$	$R_1(0, 0) = 0/5$			
$R_2(1, 1) = 1/5$	$R_2(0, 1) = 4/5$			
$R_2(1, 0) = 2/5$	$R_2(0, 0) = 3/5$			
$R_3(1, 1) = 4/5$	$R_3(0, 1) = 1/5$			
$R_3(1, 0) = 1/5$	$R_3(0, 0) = 4/5$			
$R_4(1, 1) = 2/5$	$R_4(0, 1) = 3/5$			
$R_4(1, 0) = 4/5$	$R_4(0, 0) = 1/5$			

Slide 5.2.6

These R values actually represent our hypothesis in a way we'll see more clearly later. But that means that, given a new input x, we can use the R values to compute an output value Y.

6.034 - Spring 03 • 6

Slide 5.2.7

Imagine we get a new $x = \langle 0, 0, 1, 1 \rangle$. We start out by computing a "score" for this example being a positive example. We do that by multiplying the positive R values, one for each feature. So, our x has feature 1 equal to 0, so we use R_1 of 0, 1. It has feature 2 equal to zero, so we use R_2 of 0, 1. It has feature 3 equal to 1, so we use R_3 of 1, 1. And so on. I've shown the feature values in blue to make it clear which arguments to the R functions they're responsible for. Similarly, I've shown the 1's that come from the fact that we're computing the positive score in green.

Each of the factors in the score represents the degree to which this feature tends to have this value in positive examples. Multiplied all together, they give us a measure of how likely it is that this example is positive.

Prediction

$$\begin{aligned} R_1(1,1) &= 1/5 & R_1(0,1) &= 4/5 \\ R_1(1,0) &= 5/5 & R_1(0,0) &= 0/5 \\ R_2(1,1) &= 1/5 & R_2(0,1) &= 4/5 \\ R_2(1,0) &= 2/5 & R_2(0,0) &= 3/5 \\ R_3(1,1) &= 4/5 & R_3(0,1) &= 1/5 \\ R_3(1,0) &= 1/5 & R_3(0,0) &= 4/5 \\ R_4(1,1) &= 2/5 & R_4(0,1) &= 3/5 \\ R_4(1,0) &= 4/5 & R_4(0,0) &= 1/5 \end{aligned}$$

- New $x = \langle 0, 0, 1, 1 \rangle$
- $S(1) = R_1(0,1) * R_2(0,1) * R_3(1,1) * R_4(1,1) = .205$

6.034 - Spring 03 • 7

Prediction

$$\begin{aligned} R_1(1,1) &= 1/5 & R_1(0,1) &= 4/5 \\ R_1(1,0) &= 5/5 & R_1(0,0) &= 0/5 \\ R_2(1,1) &= 1/5 & R_2(0,1) &= 4/5 \\ R_2(1,0) &= 2/5 & R_2(0,0) &= 3/5 \\ R_3(1,1) &= 4/5 & R_3(0,1) &= 1/5 \\ R_3(1,0) &= 1/5 & R_3(0,0) &= 4/5 \\ R_4(1,1) &= 2/5 & R_4(0,1) &= 3/5 \\ R_4(1,0) &= 4/5 & R_4(0,0) &= 1/5 \end{aligned}$$

- New $x = \langle 0, 0, 1, 1 \rangle$
- $S(1) = R_1(0,1) * R_2(0,1) * R_3(1,1) * R_4(1,1) = .205$
- $S(0) = R_1(0,0) * R_2(0,0) * R_3(1,0) * R_4(1,0) = 0$

6.034 - Spring 03 • 8

Slide 5.2.9

Finally, we compare score 1 to score 0, and generate output 1 because score 1 is larger than score 0.

Prediction

$$\begin{aligned} R_1(1,1) &= 1/5 & R_1(0,1) &= 4/5 \\ R_1(1,0) &= 5/5 & R_1(0,0) &= 0/5 \\ R_2(1,1) &= 1/5 & R_2(0,1) &= 4/5 \\ R_2(1,0) &= 2/5 & R_2(0,0) &= 3/5 \\ R_3(1,1) &= 4/5 & R_3(0,1) &= 1/5 \\ R_3(1,0) &= 1/5 & R_3(0,0) &= 4/5 \\ R_4(1,1) &= 2/5 & R_4(0,1) &= 3/5 \\ R_4(1,0) &= 4/5 & R_4(0,0) &= 1/5 \end{aligned}$$

- New $x = \langle 0, 0, 1, 1 \rangle$
- $S(1) = R_1(0,1) * R_2(0,1) * R_3(1,1) * R_4(1,1) = .205$
- $S(0) = R_1(0,0) * R_2(0,0) * R_3(1,0) * R_4(1,0) = 0$
- $S(1) > S(0)$, so predict class 1

6.034 - Spring 03 • 9

Learning Algorithm

- Estimate from the data, for all j :

$$R_j(1,1) = \frac{\#(x_j^i = 1 \wedge y^i = 1)}{\#(y^i = 1)}$$

6.034 - Spring 03 • 10

Slide 5.2.10

Here's the learning algorithm written out just a little bit more generally. To compute R_j of 1, 1, we just count, in our data set, how many examples there have been in which feature j has had value 1 and the output was also 1, and divide that by the total number of samples with output 1.

Slide 5.2.11

Now, R_j of 0, 1 is just 1 minus R_j of 1, 1.

Learning Algorithm

- Estimate from the data, for all j:

$$R_j(1, 1) = \frac{\#(x'_j = 1 \wedge y' = 1)}{\#(y' = 1)}$$

$$R_j(0, 1) = 1 - R_j(1, 1)$$

6.034 - Spring 03 • 11

Learning Algorithm

- Estimate from the data, for all j:

$$R_j(1, 1) = \frac{\#(x'_j = 1 \wedge y' = 1)}{\#(y' = 1)}$$

$$R_j(0, 1) = 1 - R_j(1, 1)$$

$$R_j(1, 0) = \frac{\#(x'_j = 1 \wedge y' = 0)}{\#(y' = 0)}$$

$$R_j(0, 0) = 1 - R_j(1, 0)$$

6.034 - Spring 03 • 12

Slide 5.2.12

Similarly, R_j of 1, 0 is the number of examples in which feature j had value 1 and the output was 0, divided the total number of examples with output 0. And R_j of 0, 0 is just 1 minus R_j of 1, 0.

Slide 5.2.12

- Given a new x,

$$S(1) = \prod_j \begin{cases} R_j(1, 1) & \text{if } x_j = 1 \\ R_j(0, 1) & \text{otherwise} \end{cases}$$

6.034 - Spring 03 • 13

Prediction Algorithm

- Given a new x,

$$S(1) = \prod_j \begin{cases} R_j(1, 1) & \text{if } x_j = 1 \\ R_j(0, 1) & \text{otherwise} \end{cases}$$

$$S(0) = \prod_j \begin{cases} R_j(1, 0) & \text{if } x_j = 1 \\ R_j(0, 0) & \text{otherwise} \end{cases}$$

6.034 - Spring 03 • 14

Slide 5.2.14

Similarly, $S(0)$ is the product, over all j, of R_j of 1, 0 if $x_j = 1$ and R_j of 0, 0 otherwise.

Slide 5.2.15

If $S(1)$ is greater than $S(0)$, then we'll predict that $Y = 1$, else 0.

Prediction Algorithm

- Given a new x ,

$$S(1) = \prod_j \begin{cases} R_j(1, 1) & \text{if } x_j = 1 \\ R_j(0, 1) & \text{otherwise} \end{cases}$$

$$S(0) = \prod_j \begin{cases} R_j(1, 0) & \text{if } x_j = 1 \\ R_j(0, 0) & \text{otherwise} \end{cases}$$

- Output 1 if $S(1) > S(0)$

6.034 - Spring 03 • 15

Prediction Algorithm

- Given a new x ,

$$\log S(1) = \sum_j \begin{cases} \log R_j(1, 1) & \text{if } x_j = 1 \\ \log R_j(0, 1) & \text{otherwise} \end{cases}$$

$$\log S(0) = \sum_j \begin{cases} \log R_j(1, 0) & \text{if } x_j = 1 \\ \log R_j(0, 0) & \text{otherwise} \end{cases}$$

- Output 1 if $\log S(1) > \log S(0)$

Better to add logs than to multiply small probabilities

6.034 - Spring 03 • 16

Slide 5.2.17

In our example, we saw that if we had never seen a feature take value 1 in a positive example, our estimate for how likely that would be to happen in the future was 0. That seems pretty radical, especially when we only have had a few examples to learn from. There's a standard hack to fix this problem, called the "Laplace correction". When counting up events, we add a 1 to the numerator and a 2 to the denominator.

If we've never seen any positive instances, for example, our $R(1,1)$ values would be 1/2, which seems sort of reasonable in the absence of any information. And if we see lots and lots of examples, this 1 and 2 will be washed out, and we'll converge to the same estimate that we would have gotten without the correction.

There's a beautiful probabilistic justification for what looks like an obvious hack. But, sadly, it's beyond the scope of this class.

Laplace Correction

- Avoid getting 0 or 1 as an answer:

$$R_j(1, 1) = \frac{\#(x'_j = 1 \wedge y' = 1) + 1}{\#(y' = 1) + 2}$$

$$R_j(0, 1) = 1 - R_j(1, 1)$$

$$R_j(1, 0) = \frac{\#(x'_j = 1 \wedge y' = 0) + 1}{\#(y' = 0) + 2}$$

$$R_j(0, 0) = 1 - R_j(1, 0)$$

6.034 - Spring 03 • 17

Example with Correction

f_1	f_2	f_3	f_4	y
0	1	1	0	1
0	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	0	1	1
1	0	0	1	0
1	1	0	1	0
1	0	0	0	0
1	1	0	1	0
1	0	1	0	0

Slide 5.2.18

Here's what happens to our original example if we use the Laplace correction. Notably, R_1 of 0, 0 is now 1/7 instead of 0, which is less dramatic.

6.034 - Spring 03 • 18

Slide 5.2.19

And so, when it comes time to make a prediction, the score for answer 0 is no longer 0. We think it's possible, but unlikely, that this example is negative. So we still predict class 1.

Prediction with Correction

- $R_1(1,1)=2/7 \quad R_1(0,1)=5/7$
- $R_1(1,0)=6/7 \quad R_1(0,0)=1/7$
- $R_2(1,1)=2/7 \quad R_2(0,1)=5/7$
- $R_2(1,0)=3/7 \quad R_2(0,0)=4/7$
- $R_3(1,1)=5/7 \quad R_3(0,1)=2/7$
- $R_3(1,0)=2/7 \quad R_3(0,0)=5/7$
- $R_4(1,1)=3/7 \quad R_4(0,1)=4/7$
- $R_4(1,0)=5/7 \quad R_4(0,0)=2/7$

- New $x = <0,0,1,1>$
- $S(1) = R_1(0,1)*R_2(0,1)*R_3(1,1)*R_4(1,1) = .156$
- $S(0) = R_1(0,0)*R_2(0,0)*R_3(1,0)*R_4(1,0) = .017$
- $S(1) > S(0)$, so predict class 1

6.034 - Spring 03 • 19

Hypothesis Space

- Output 1 if

$$\prod_j \alpha_j x_j + (1 - \alpha_j)(1 - x_j) > \prod_j \beta_j x_j + (1 - \beta_j)(1 - x_j)$$

- Depends on parameters $\alpha_1 \dots \alpha_n, \beta_1 \dots \beta_n$ (which we set to be the R_j values)

Slide 5.2.20

What's the story of this algorithm in terms of hypothesis space? We've fixed the spaces of hypotheses to have the form shown here. This is a very restricted form. But it is still a big (infinite, in fact) hypothesis space, because we have to pick the actual values of the coefficients α_j and β_j for all j .

Hypothesis Space

- Output 1 if

$$\prod_j \alpha_j x_j + (1 - \alpha_j)(1 - x_j) > \prod_j \beta_j x_j + (1 - \beta_j)(1 - x_j)$$

- Depends on parameters $\alpha_1 \dots \alpha_n, \beta_1 \dots \beta_n$ (which we set to be the R_j values)

- Our method of computing parameters doesn't minimize training set error, but it's fast!

6.034 - Spring 03 • 21

Hypothesis Space

- Output 1 if

$$\prod_j \alpha_j x_j + (1 - \alpha_j)(1 - x_j) > \prod_j \beta_j x_j + (1 - \beta_j)(1 - x_j)$$

- Depends on parameters $\alpha_1 \dots \alpha_n, \beta_1 \dots \beta_n$ (which we set to be the R_j values)

- Our method of computing parameters doesn't minimize training set error, but it's fast!

- Weight of feature j's "vote" in favor of output 1:

$$\log \frac{\alpha_j}{1 - \alpha_j} - \log \frac{\beta_j}{1 - \beta_j}$$

Slide 5.2.22

One possible concern about this algorithm is that it's hard to interpret the hypotheses you get back. With DNF or decision trees, it's easy for a human to understand what features are playing an important role, for example.

In naive Bayes, all the features are playing some role in the categorization. You can think of each one as casting a weighted vote in favor of an answer of 1 versus 0. The weight of each feature's vote is this expression. The absolute value of this weight is a good indication of how important a feature is, and its sign tells us whether that feature is indicative of output 1 (when it's positive) or output 0 (when it's negative).

6.034 - Spring 03 • 22

Slide 5.2.23

This algorithm makes a fundamental assumption that we can characterize the influence of each feature on the class independently, and then combine them through multiplication. This assumption isn't always justified. We'll illustrate this using our old nemesis, exclusive or. Here's the data set we used before.

Exclusive Or

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	1	0
1	1	1	0	1
0	0	0	1	1

6.034 - Spring 03 • 23

Exclusive Or

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	1	0
1	1	1	0	1
0	0	0	1	1

- $R_1(1,1)=2/4 \quad R_1(0,1)=2/4$
- $R_1(1,0)=3/6 \quad R_1(0,0)=3/6$
- $R_2(1,1)=2/4 \quad R_2(0,1)=2/4$
- $R_2(1,0)=3/6 \quad R_2(0,0)=3/6$
- $R_3(1,1)=2/4 \quad R_3(0,1)=2/4$
- $R_3(1,0)=3/6 \quad R_3(0,0)=3/6$
- $R_4(1,1)=2/4 \quad R_4(0,1)=2/4$
- $R_4(1,0)=3/6 \quad R_4(0,0)=3/6$

6.034 - Spring 03 • 24

Slide 5.2.25

Sure enough, when we compute the scores for any new example, we get the same result, giving us no basis at all for predicting the output.

Exclusive Or

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	1	0
1	1	1	0	1
0	0	0	1	1

- $R_1(1,1)=2/4 \quad R_1(0,1)=2/4$
- $R_1(1,0)=3/6 \quad R_1(0,0)=3/6$
- $R_2(1,1)=2/4 \quad R_2(0,1)=2/4$
- $R_2(1,0)=3/6 \quad R_2(0,0)=3/6$
- $R_3(1,1)=2/4 \quad R_3(0,1)=2/4$
- $R_3(1,0)=3/6 \quad R_3(0,0)=3/6$
- $R_4(1,1)=2/4 \quad R_4(0,1)=2/4$
- $R_4(1,0)=3/6 \quad R_4(0,0)=3/6$

- For any new x
- $S(\textcolor{teal}{1}) = .5 * .5 * .5 * .5 = .0625$
- $S(\textcolor{teal}{0}) = .5 * .5 * .5 * .5 = .0625$
- We're indifferent between classes

6.034 - Spring 03 • 25

Congressional Voting**Slide 5.2.26**

Now we show the results of applying naive Bayes to the congressional voting domain. In this case, we might expect the independence assumption to be reasonably well satisfied (a congressperson probably doesn't decide on groups of votes together, unless there are deals being made).

6.034 - Spring 03 • 26

Slide 5.2.27

Using cross-validation, we determined that the accuracy of hypotheses generated by naive Bayes was approximately 0.91. This is not as good as that of decision trees, which had an accuracy of about 0.95. This result is not too surprising: decision trees can express more complex hypotheses that consider combinations of attributes.

Congressional Voting

- Accuracy on the congressional voting domain is about 0.91
- Somewhat worse than decision trees (0.95)
- Decision trees can express more complex hypotheses over combinations of attributes

6.034 - Spring 03 • 27

Congressional Voting

- Accuracy on the congressional voting domain is about 0.91
- Somewhat worse than decision trees (0.95)
- Decision trees can express more complex hypotheses over combinations of attributes
- Domain is small enough so that speed is not an issue
- So, prefer trees or DNF in this domain

6.034 - Spring 03 • 28

Slide 5.2.28

It's interesting to look at the weights found for the various attributes. Here, I've sorted the attributes according to magnitude of the weight assigned to them by naive Bayes. The positive ones (colored black) vote in favor of the output being 1 (republican); the negative ones (colored red) vote against the the output being 1 (and therefore in favor of democrat).

The results are consistent with the answers we've gotten from the other algorithms. The most diagnostic single issue seems to be voting on whether physician fees should be frozen; that is a strong indicator of being a democrat. The strongest indicators of being republican are accepting the budget, aid to the contras, and support of the mx missile.

Congressional Voting

-6.82	physician-fee-freeze
-4.20	el-salvador-aid
-4.20	crime
-3.56	education-spending
3.36	adoption-of-the-budget-resolution
3.25	aid-to-nicaraguan-contras
3.07	mx-missile
-2.51	superfund-right-to-sue
2.40	duty-free-exports
2.14	anti-satellite-test-ban
-2.07	religious-groups-in-schools
2.01	export-administration-act-south-africa
1.66	synfuels-corporation-cutback
1.63	handicapped-infants
-0.17	immigration
-0.08	water-project-cost-sharing

republican
democrat

6.034 - Spring 03 • 29

Probabilistic Inference**Slide 5.2.30**

Now we'll look briefly at the probabilistic justification for the algorithm. If you don't follow it, don't worry. But if you've studied probability before, this ought to provide some useful intuition.

6.034 - Spring 03 • 30

Slide 5.2.31

One way to think about the problem of deciding what class a new item belongs to is to think of the features and the output as random variables. If we knew $\Pr(Y = 1 | f_1 \dots f_n)$, then when we got a new example, we could compute the probability that it had a Y value of 1, and generate the answer 1 if the probability was over 0.5. So, we're going to concentrate on coming up with a way to estimate this probability.

Probabilistic Inference

- Think of features and output as random variables
- Learn $\Pr(Y = 1 | f_1, \dots, f_n)$
- Given new example, compute probability it has value 1
- Generate answer 1 if that value is > 0.5, else 0
- Concentrate on estimating this distribution from data
 $\Pr(Y = 1 | f_1, \dots, f_n)$

6.034 - Spring 03 • 31

Bayes' Rule

- Generically:

$$\Pr(A | B) = \Pr(B | A) \frac{\Pr(A)}{\Pr(B)}$$

Slide 5.2.32

Bayes' rule gives us a way to take the conditional probability $\Pr(A|B)$ and express it in terms of $\Pr(B|A)$ and the marginals $\Pr(A)$ and $\Pr(B)$.

Slide 5.2.33

Applying it to our problem, we get $\Pr(Y = 1 | f_1 \dots f_n) = \Pr(f_1 \dots f_n | Y = 1) \Pr(Y = 1) / \Pr(f_1 \dots f_n)$

6.034 - Spring 03 • 32

Bayes' Rule

- Generically:

$$\Pr(A | B) = \Pr(B | A) \frac{\Pr(A)}{\Pr(B)}$$

- Specifically:

$$\Pr(Y = 1 | f_1 \dots f_n) = \Pr(f_1 \dots f_n | Y = 1) \frac{\Pr(Y = 1)}{\Pr(f_1 \dots f_n)}$$

6.034 - Spring 03 • 33

Bayes' Rule

- Generically:

$$\Pr(A | B) = \Pr(B | A) \frac{\Pr(A)}{\Pr(B)}$$

- Specifically:

$$\Pr(Y = 1 | f_1 \dots f_n) = \Pr(f_1 \dots f_n | Y = 1) \frac{\Pr(Y = 1)}{\Pr(f_1 \dots f_n)}$$

independent of Y

Slide 5.2.34

Since the denominator is independent of Y, we can ignore it in figuring out whether $\Pr(Y = 1 | f_1 \dots f_n)$ is greater than $\Pr(Y = 0 | f_1 \dots f_n)$.

6.034 - Spring 03 • 34

Slide 5.2.35

The term $\Pr(Y = 1)$ is often called the **prior**. It's a way to build into our decision-making a previous belief about the proportion of things that are positive. For now, we'll just assume that it's 0.5 for both positive and negative classes, and ignore it.

Bayes' Rule

- Generically:

$$\Pr(A | B) = \Pr(B | A) \frac{\Pr(A)}{\Pr(B)}$$

- Specifically:

$$\Pr(Y = 1 | f_1 \dots f_n) = \Pr(f_1 \dots f_n | Y = 1) \frac{\Pr(Y = 1)}{\Pr(f_1 \dots f_n)}$$

6.034 - Spring 03 • 35

Bayes' Rule

- Generically:

$$\Pr(A | B) = \Pr(B | A) \frac{\Pr(A)}{\Pr(B)}$$

- Specifically:

$$\Pr(Y = 1 | f_1 \dots f_n) = \Pr(f_1 \dots f_n | Y = 1) \frac{\Pr(Y = 1)}{\Pr(f_1 \dots f_n)}$$

- Concentrate on:

$$\Pr(f_1 \dots f_n | Y = 1)$$

6.034 - Spring 03 • 36

Slide 5.2.36

This will allow us to concentrate on $\Pr(f_1 \dots f_n | Y = 1)$.

Slide 5.2.37

The algorithm is called **naive** Bayes because it makes a big assumption, which is that it can be broken down into a product like this. A probabilist would say that we are assuming that the features are conditionally independent given the class.

So, we're assuming that $\Pr(f_1 \dots f_n | Y = 1)$ is the product of all the individual conditional probabilities, $\Pr(f_j | Y = 1)$.

Why is Bayes Naïve?

- Make a big independence assumption

$$\Pr(f_1 \dots f_n | Y = 1) = \prod_j \Pr(f_j | Y = 1)$$

6.034 - Spring 03 • 37

Learning Algorithm

- Estimate from the data, for all j:

$$R(f_j = 1 | Y = 1) = \frac{\#(x_j^i = 1 \wedge y^i = 1)}{\#(y^i = 1)}$$

$$R(f_j = 0 | Y = 1) = 1 - R(f_j = 1 | Y = 1)$$

$$R(f_j = 1 | Y = 0) = \frac{\#(x_j^i = 1 \wedge y^i = 0)}{\#(y^i = 0)}$$

$$R(f_j = 0 | Y = 0) = 1 - R(f_j = 1 | Y = 0)$$

Slide 5.2.38

Here is our same learning algorithm (without the Laplace correction, for simplicity), expressed in probabilistic terms.

We can think of the R values as estimates of the underlying conditional probabilities, based on the training set as a statistical sample drawn from those distributions.

6.034 - Spring 03 • 38

Slide 5.2.39

And here's the prediction algorithm, just written out using probabilistic notation. The S is also an estimated probability. So we predict output 1 just when we think it's more likely that our new x would have come from class 1 than from class 0.

Now, we'll move on to considering the situation in which the inputs and outputs of the learning process can be real-valued.

Prediction Algorithm

- Given a new x ,

$$S(x_1 \dots x_n | Y = 1) = \prod_j \begin{cases} R(f_j = 1 | Y = 1) & \text{if } x_j = 1 \\ R(f_j = 0 | Y = 1) & \text{otherwise} \end{cases}$$

$$S(x_1 \dots x_n | Y = 0) = \prod_j \begin{cases} R(f_j = 1 | Y = 0) & \text{if } x_j = 1 \\ R(f_j = 0 | Y = 0) & \text{otherwise} \end{cases}$$

- Output 1 if

$$S(x_1 \dots x_n | Y = 1) > S(x_1 \dots x_n | Y = 0)$$



6.034 Notes: Section 10.1

Slide 10.1.1

So far, we've only talked about binary features. But real problems are typically characterized by much more complex features.

Feature Spaces

- Features can be much more complex

6.034 - Spring 03 • 1

Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set

6.034 - Spring 03 • 2

Slide 10.1.3

When the set doesn't have a natural order (actually, when it doesn't have a natural distance between the elements), then the easiest way to deal with it is to convert it into a bunch of binary attributes.

Your first thought might be to convert it using binary numbers, so that if you have four elements, you can encode them as 00, 01, 10, and 11. Although that could work, it makes hard work for the learning algorithm, which, in order to select out a particular value in the set will have to do some hard work to decode the bits in these features.

Instead, we typically make it easier on our algorithms by encoding such sets in unary, with one bit per element in the set. Then, for each value, we turn on one bit and set the rest to zero. So, we could encode a four-item set as 1000, 0100, 0010, 0001.

Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set
 - If set is unordered (4 different makes of cars, for example), use binary attributes to encode the values (1000, 0100, 0010, 0001)

6.034 - Spring 03 • 3

Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set
 - If set is unordered (4 different makes of cars, for example), use binary attributes to encode the values (1000, 0100, 0010, 0001)
 - If set is ordered, treat as real-valued

Slide 10.1.4

On the other hand, when the set has a natural order, like someone's age, or the number of bedrooms in a house, it can usually be treated as if it were a real-valued attribute using methods we're about to explore.

6.034 - Spring 03 * 4

Slide 10.1.5

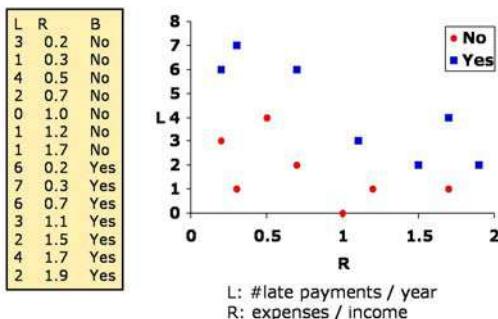
We'll spend this segment and the next looking at methods for dealing with real-valued attributes. The main goal will be to take advantage of the notion of distance between values that the reals affords us in order to build in a very deep bias that inputs whose features have "nearby" values ought, in general, to have "nearby" outputs.

Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set
 - If set is unordered (4 different makes of cars, for example), use binary attributes to encode the values (1000, 0100, 0010, 0001)
 - If set is ordered, treat as real-valued
- Real-valued: bias that inputs whose features have "nearby" values ought to have "nearby" outputs

6.034 - Spring 03 * 5

Predicting Bankruptcy



6.034 - Spring 03 * 6

Slide 10.1.6

We'll use the example of predicting whether someone is going to go bankrupt. It only has two features, to make it easy to visualize.

One feature, L, is the number of late payments they have made on their credit card this year. This is a discrete value that we're treating as a real.

The other feature, R, is the ratio of their expenses to their income. The higher it is, the more likely you'd think the person would be to go bankrupt.

We have a set of examples of people who did, in fact go bankrupt, and a set who did not. We can plot the points in a two-dimensional space, with a dimension for each attribute. We've colored the "positive" (bankrupt) points blue and the negative points red.

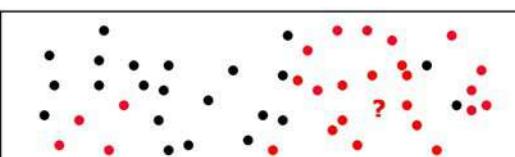
Slide 10.1.7

We took a brief look at the nearest neighbor algorithm in the first segment on learning. The idea is that you remember all the data points you've ever seen and, when you're given a query point, you find the old point that's nearest to the query point and predict its y value as your output.

Love thy Nearest Neighbor

- Remember all your data
- When someone asks a question,
 - find the nearest old data point
 - return the answer associated with it

6.034 - Spring 03 * 7



What do we mean by "Nearest"?

- Need a distance function on inputs
- Typically use Euclidean distance (length of a straight line between the points)

$$D(x', x^k) = \sqrt{\sum_j (x'_j - x^k_j)^2}$$

Slide 10.1.8

In order to say what point is nearest, we have to define what we mean by "near". Typically, we use Euclidean distance between two points, which is just the square root of the sum of the squared differences between corresponding feature values.

6.034 - Spring 03 • 8

Slide 10.1.9

In other machine learning applications, the inputs can be something other than fixed-length vectors of numbers. We can often still use nearest neighbor, with creative use of distance metrics. The distance between two DNA strings, for example, might be the number of single-character edits required to turn one into the other.

What do we mean by "Nearest"?

- Need a distance function on inputs
- Typically use Euclidean distance (length of a straight line between the points)

$$D(x', x^k) = \sqrt{\sum_j (x'_j - x^k_j)^2}$$

- Distance between character strings might be number of edits required to turn one into the other

6.034 - Spring 03 • 9

Scaling

- What if we're trying to predict a car's gas mileage?
 - f_1 = weight in pounds
 - f_2 = number of cylinders

Slide 10.1.10

The naive Euclidean distance isn't always appropriate, though.

Consider the case where we have two features describing a car. One is its weight in pounds and the other is the number of cylinders. The first will tend to have values in the thousands, whereas the second will have values between 4 and 8.

6.034 - Spring 03 • 10

Slide 10.1.11

If we just use Euclidean distance in this space, the number of cylinders will have essentially no influence on nearness. A difference of 4 pounds in a car's weight will swamp a difference between 4 and 8 cylinders.

Scaling

- What if we're trying to predict a car's gas mileage?
 - f_1 = weight in pounds
 - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales

6.034 - Spring 03 • 11

Scaling

- What if we're trying to predict a car's gas mileage?
 - f_1 = weight in pounds
 - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs

Slide 10.1.12

One standard method for addressing this problem is to re-scale the features.

In the simplest case, you might, for each feature, compute its range (the difference between its maximum and minimum values). Then scale the feature by subtracting the minimum value and dividing by the range. All features values would be between 0 and 1.



6.034 - Spring 03 • 12

Slide 10.1.13

A somewhat more robust method (in case you have a crazy measurement, perhaps due to a noise in a sensor, that would make the range huge) is to scale the inputs to have 0 mean and standard deviation 1. If you haven't seen this before, it means to compute the average value of the feature, \bar{x} , and subtract it from each feature value, which will give you features all centered at 0. Then, to deal with the range, you compute the standard deviation (which is the square root of the variance, which we'll talk about in detail in the segment on regression) and divide each value by that. This transformation, called normalization, puts all of the features on about equal footing.

Scaling

- What if we're trying to predict a car's gas mileage?
 - f_1 = weight in pounds
 - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs to have mean 0 and variance 1:

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

average
standard deviation

6.034 - Spring 03 • 13



Scaling

- What if we're trying to predict a car's gas mileage?
 - f_1 = weight in pounds
 - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs to have mean 0 and variance 1:

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

average
standard deviation
- Or, build knowledge in by scaling features differently



6.034 - Spring 03 • 14

Slide 10.1.14

Of course, you may not want to have all your features on equal footing. It may be that you happen to know, based on the nature of the domain, that some features are more important than others. In such cases, you might want to multiply them by a weight that will increase their influence in the distance calculation.

Slide 10.1.15

Another popular, but somewhat advanced, technique is to use cross validation and gradient descent to choose weightings of the features that generate the best performance on the particular data set.

Scaling

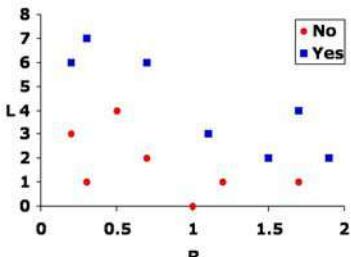
- What if we're trying to predict a car's gas mileage?
 - f_1 = weight in pounds
 - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs to have mean 0 and variance 1:

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

average
standard deviation
- Or, build knowledge in by scaling features differently
- Or use cross-validation to choose scales



6.034 - Spring 03 • 15

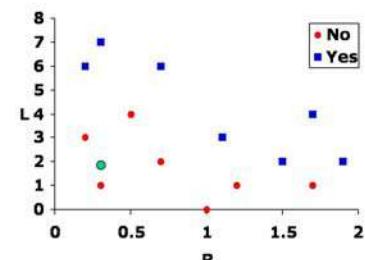
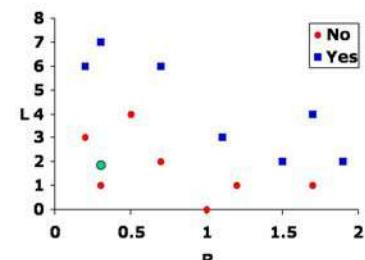
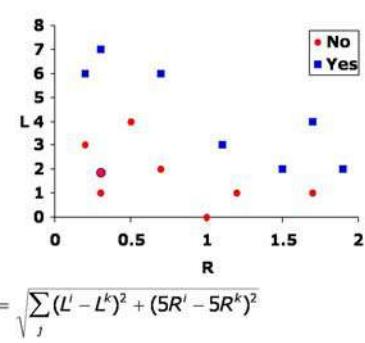
Predicting Bankruptcy

$$D(x^i, x^k) = \sqrt{\sum_j (L^j - L^k)^2 + (5R^j - 5R^k)^2}$$

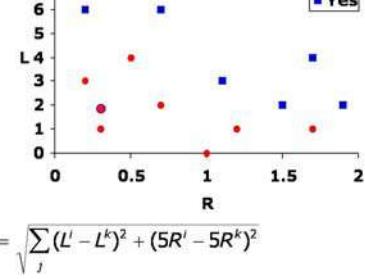
Slide 10.1.16

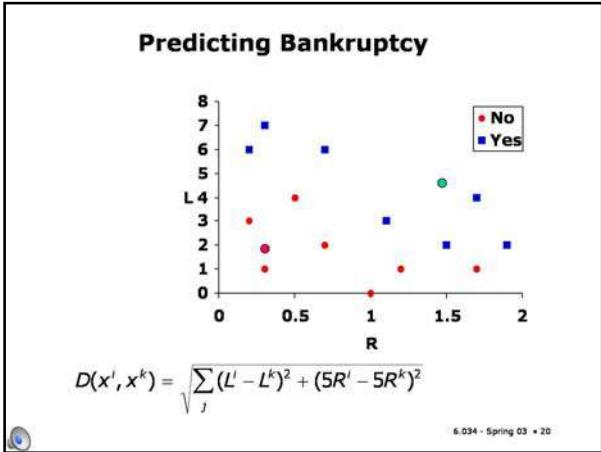
Okay. Let's see how nearest neighbor works on our bankruptcy example. Let's say we've thought about the domain and decided that the R feature (ratio between expenses and income) needs to be scaled up by 5 in order to be appropriately balanced against the L feature (number of late payments).

So we'll use Euclidian distance, but with the R values multiplied by 5 first. We've scaled the axes on the slide so that the two dimensions are graphically equal. This means that locus of points at a particular distance d from a point on our graph will appear as a circle.

Predicting Bankruptcy**Predicting Bankruptcy****Predicting Bankruptcy****Predicting Bankruptcy****Slide 10.1.19**

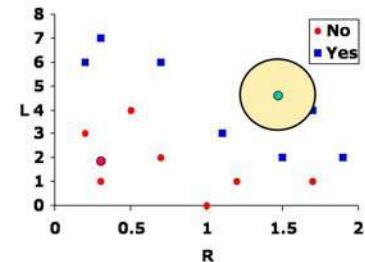
And so our answer would be "no".

Predicting Bankruptcy**Predicting Bankruptcy**

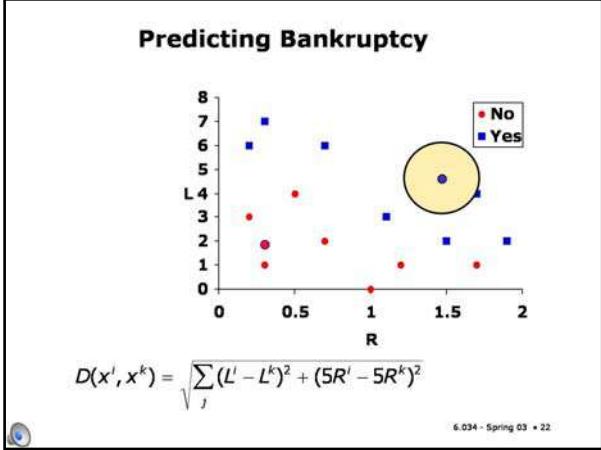
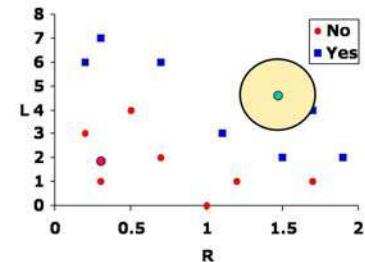


Slide 10.1.20
Similarly, for another query point,

Predicting Bankruptcy



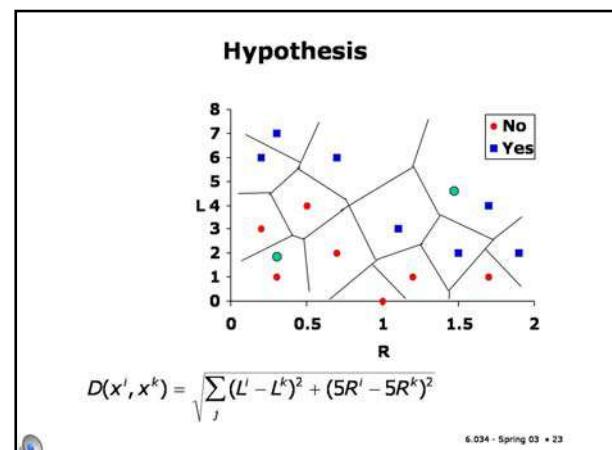
Predicting Bankruptcy

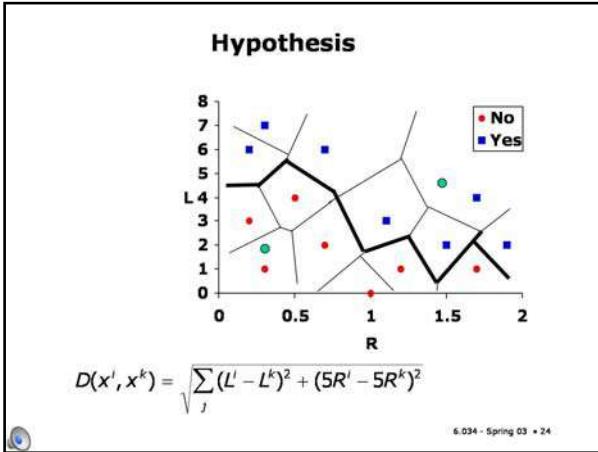


Slide 10.1.22
and generate "yes" as our prediction.

Slide 10.1.23
So, what is the hypothesis of the nearest neighbor algorithm? It's sort of different from our other algorithms, in that it isn't explicitly constructing a description of a hypothesis based on the data it sees.

Given a set of points and a distance metric, you can divide the space up into regions, one for each point, which represent the set of points in space that are nearer to this designated point than to any of the others. In this figure, I've drawn a (somewhat inaccurate) picture of the decomposition of the space into such regions. It's called a "Voronoi partition" of the space.



**Slide 10.1.24**

Now, we can think of our hypothesis as being represented by the edges in the Voronoi partition that separate a region associated with a positive point from a region associated with a negative one. In our example, that generates this bold boundary.

It's important to note that we never explicitly compute this boundary; it just arises out of the "nearest neighbor" query process.

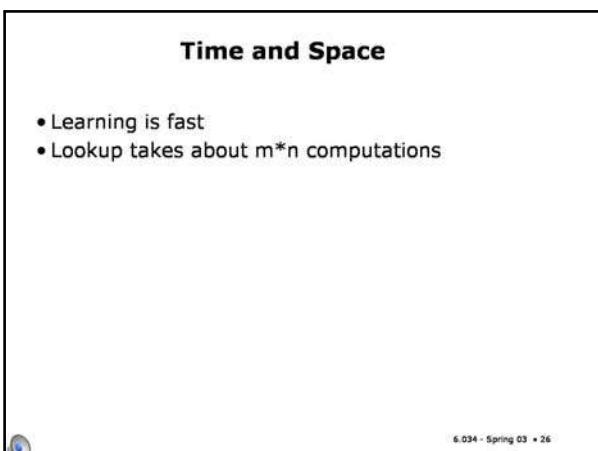
Slide 10.1.25

It's useful to spend a little bit of time thinking about how complex this algorithm is. Learning is very fast. All you have to do is remember all the data you've seen!

Time and Space

- Learning is fast

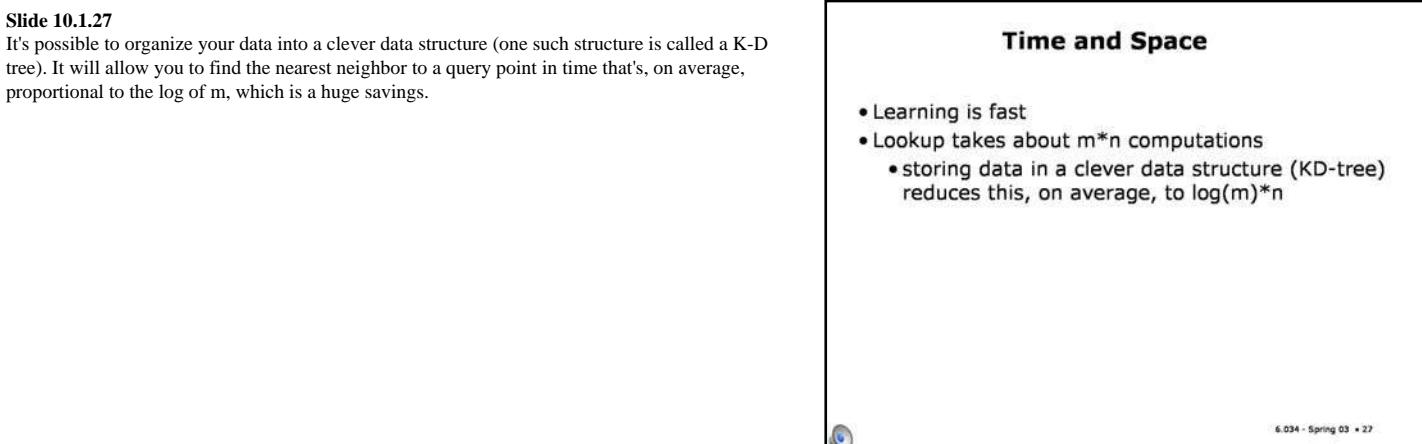
6.034 - Spring 03 • 25

**Slide 10.1.26**

What takes longer is answering a query. Naively, you have to, for each point in your training set (and there are m of them) compute the distance to the query point (which takes about n computations, since there are n features to compare). So, overall, this takes about $m * n$ time.

Slide 10.1.27

It's possible to organize your data into a clever data structure (one such structure is called a K-D tree). It will allow you to find the nearest neighbor to a query point in time that's, on average, proportional to the log of m , which is a huge savings.



Time and Space

- Learning is fast
- Lookup takes about $m \times n$ computations
 - storing data in a clever data structure (KD-tree) reduces this, on average, to $\log(m) \times n$
- Memory can fill up with all that data

Slide 10.1.28

Another issue is memory. If you gather data over time, you might worry about your memory filling up, since you have to remember it all.

6.034 - Spring 03 • 28

Slide 10.1.29

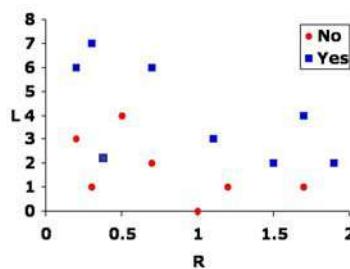
There are a number of variations on nearest neighbor that allow you to forget some of the data points; typically the ones that are most forgettable are those that are far from the current boundary between positive and negative.

Time and Space

- Learning is fast
- Lookup takes about $m \times n$ computations
 - storing data in a clever data structure (KD-tree) reduces this, on average, to $\log(m) \times n$
- Memory can fill up with all that data
 - delete points that are far away from the boundary

6.034 - Spring 03 • 29

Noise



Slide 10.1.30

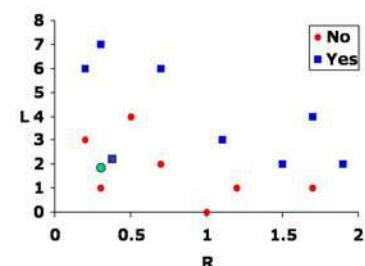
In our example so far, there has not been much (apparent) noise; the boundary between positives and negatives is clean and simple. Let's now consider the case where there's a blue point down among the reds. Someone with an apparently healthy financial record goes bankrupt.

There are, of course, two ways to deal with this data point. One is to assume that it is not noise; that is, that there is some regularity that makes people like this one go bankrupt in general. The other is to say that this example is an "outlier". It represents an unusual case that we would prefer largely to ignore, and not to incorporate it into our hypothesis.

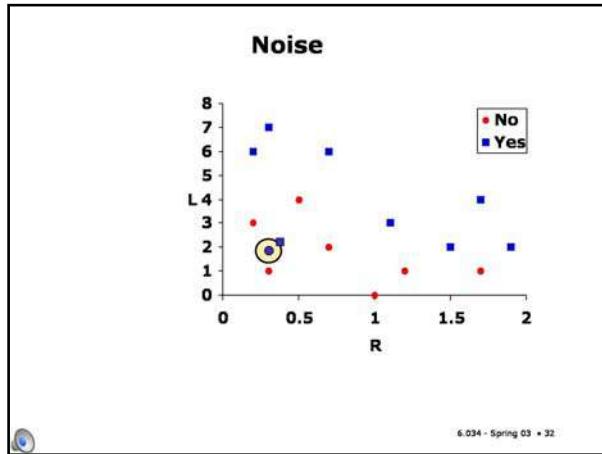
Slide 10.1.31

So, what happens in nearest neighbor if we get a query point next to this point?

Noise



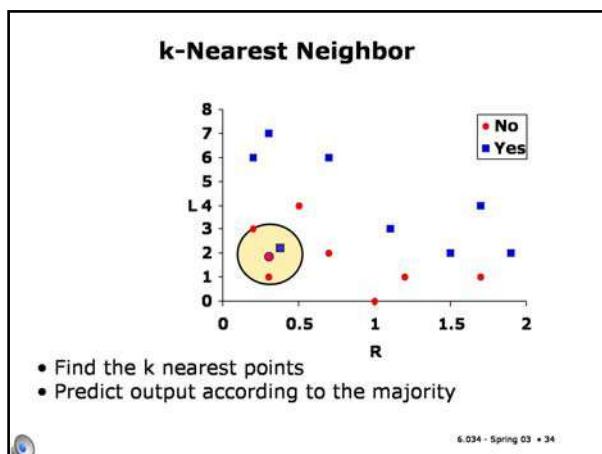
6.034 - Spring 03 • 31

**Slide 10.1.32**

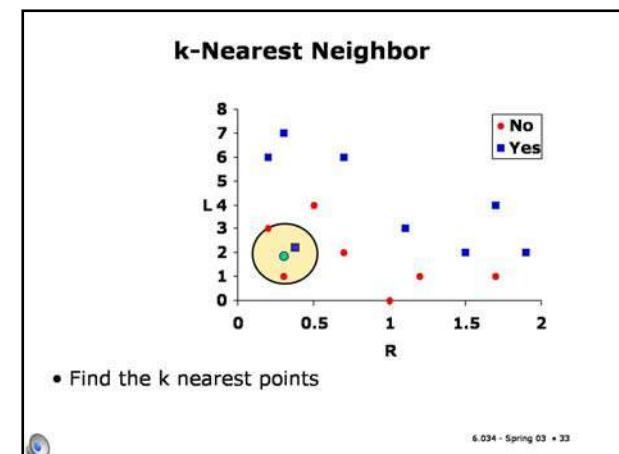
We find the nearest neighbor, which is a "yes" point, and predict the answer "yes". This outcome is consistent with the first view; that is, that this blue point represents some important property of the problem.

Slide 10.1.33

But if we think there might be noise in the data, we can change the algorithm a bit to try to ignore it. We'll move to the k-nearest neighbor algorithm. It's just like the old algorithm, except that when we get a query, we'll search for the k closest points to the query points. And we'll generate, as output, the output associated with the majority of the k closest elements.

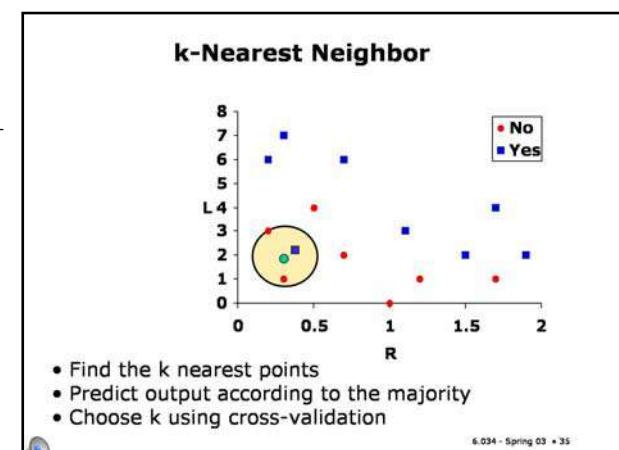
**Slide 10.1.34**

In this case, we've chosen k to be 3. The three closest points consist of two "no"s and a "yes", so our answer would be "no".

**Slide 10.1.35**

It's not entirely obvious how to choose k. The smaller the k, the more noise-sensitive your hypothesis is. The larger the k, the more "smeared out" it is. In the limit of large k, you would always just predict the output value that's associated with the majority of your training points. So, k functions kind of like a complexity-control parameter, exactly analogous to epsilon in DNF and min-leaf-size in decision trees. With smaller k, you have high variance and risk overfitting; with large k, you have high bias and risk not being able to express the hypotheses you need.

It's common to choose k using cross-validation.



Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:

Slide 10.1.36

Nearest neighbor works very well (and is often the method of choice) for problems in relatively low-dimensional real-valued spaces.

But as the dimensionality of a space increases, its geometry gets weird. Here are some surprising (to me, at least) facts about high-dimensional spaces.

Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:
 - In high dimensions, almost all points are far away from one another
 - They're almost all near the boundaries

Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:
 - In high dimensions, almost all points are far away from one another
 - They're almost all near the boundaries
- Imagine sprinkling data points uniformly within a 10-dimensional unit cube
 - To capture 10% of the points, you'd need a cube with sides of length .63!

Slide 10.1.38

Imagine sprinkling data points uniformly within a 10-dimensional unit cube (cube whose sides are of length 1).

To capture 10% of the points, you'd need a cube with sides of length .63!

Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:
 - In high dimensions, almost all points are far away from one another
 - They're almost all near the boundaries
- Imagine sprinkling data points uniformly within a 10-dimensional unit cube
 - To capture 10% of the points, you'd need a cube with sides of length .63!
- Cure: feature selection or more global models

Slide 10.1.39

All this means that the notions of nearness providing a good generalization principle, which are very effective in low-dimensional spaces, become fairly ineffective in high-dimensional spaces. There are two ways to handle this problem. One is to do "feature selection", and try to reduce the problem back down to a lower-dimensional one. The other is to fit hypotheses from a much smaller hypothesis class, such as linear separators, which we will see in the next chapter.

Test Domains

Slide 10.1.40

We'll look at how nearest neighbor performs on two different test domains.

6.034 - Spring 03 • 40

Slide 10.1.41

The first domain is predicting whether a person has heart disease, represented by a significant narrowing of the arteries, based on the results of a variety of tests. This domain has 297 different data points, each of which is characterized by 26 features. A lot of these features are actually boolean, which means that although the dimensionality is high, the curse of dimensionality, which really only bites us badly in the case of real-valued features, doesn't cause too much problem.

Test Domains

- Heart Disease: predict whether a person has significant narrowing of the arteries, based on tests
 - 26 features
 - 297 data points

6.034 - Spring 03 • 41

Test Domains

- Heart Disease: predict whether a person has significant narrowing of the arteries, based on tests
 - 26 features
 - 297 data points
- Auto MPG: predict whether a car gets more than 22 miles per gallon, based on attributes of car
 - 12 features
 - 385 data points

6.034 - Spring 03 • 42

Slide 10.1.42

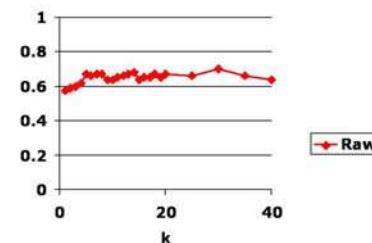
In the second domain, we're trying to predict whether a car gets more than 22 miles-per-gallon fuel efficiency. We have 385 data points, characterized by 12 features. Again, a number of the features are binary.

Slide 10.1.43

Here's a graph of the cross-validation accuracy of nearest neighbor on the heart disease data, shown as a function of k. Looking at the data, we can see that the performance is relatively insensitive to the choice of k, though it seems like maybe it's useful to have k be greater than about 5.

Heart Disease

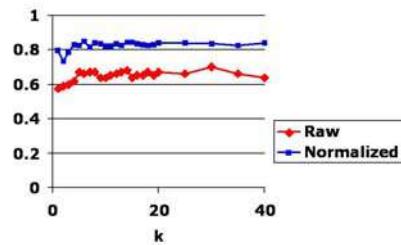
- Relatively insensitive to k



6.034 - Spring 03 • 43

Heart Disease

- Relatively insensitive to k
- Normalization matters!



Slide 10.1.44

The red curve is the performance of nearest neighbor using the features directly as they are measured, without any scaling. We then normalized all of the features to have mean 0 and standard deviation 1, and re-ran the algorithm. You can see here that it makes a noticeable increase in performance.

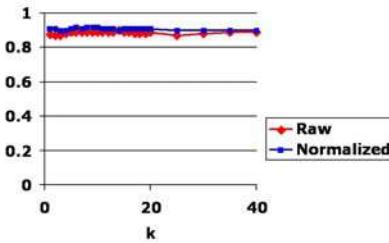
6.034 - Spring 03 • 44

Slide 10.1.45

We ran nearest neighbor with both normalized and un-normalized inputs on the auto-MPG data. It seems to perform pretty well in all cases. It is still relatively insensitive to k, and normalization only seems to help a tiny amount.

Auto MPG

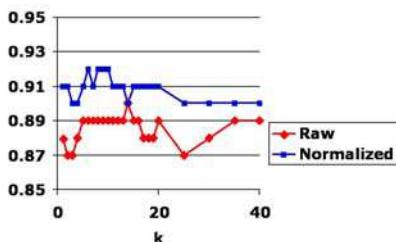
- Relatively insensitive to k
- Normalization doesn't matter much



6.034 - Spring 03 • 45

Auto MPG

- Now normalization matters a lot!
- Watch the scales on your graphs



Slide 10.1.46

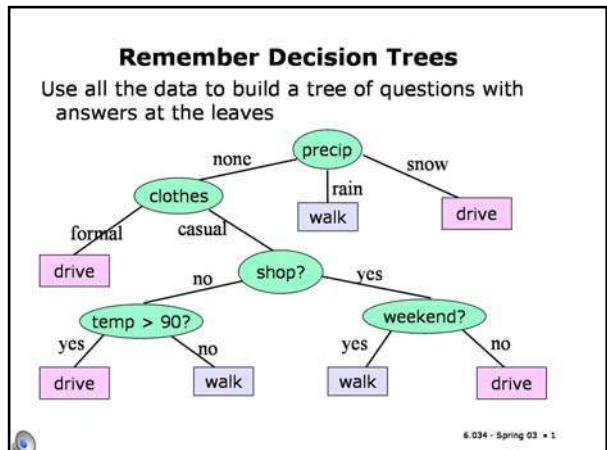
Watch out for tricky graphing! It's always possible to make your algorithm look much better than the other leading brand (as long as it's a little bit better), by changing the scale on your graphs. The previous graph had a scale of 0 to 1. This graph has a scale of 0.85 to 0.95. Now the normalized version looks much better! Be careful of such tactics when you read other peoples' papers; and certainly don't practice them in yours.

6.034 - Spring 03 • 46

6.034 Notes: Section 10.2

Slide 10.2.1

Now, let's go back to decision trees, and see if we can apply them to problems where the inputs are numeric.


Numerical Attributes

- Tests in nodes can be of the form $x_j > \text{constant}$

Slide 10.2.2

When we have features with numeric values, we have to expand our hypothesis space to include different tests on the leaves. We will allow tests on the leaves of a decision tree to be comparisons of the form $x_j > c$, where c is a constant.

Slide 10.2.3

This class of splits allows us to divide our feature-space into a set of exhaustive and mutually exclusive hyper-rectangles (that is, rectangles of potentially high dimension), with one rectangle for each leaf of the tree. So, each rectangle will have an output value (1 or 0) associated with it. The set of rectangles and their output values constitutes our hypothesis.

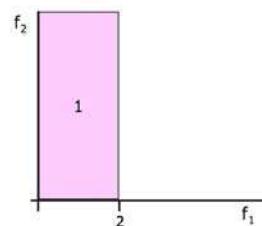
Numerical Attributes

- Tests in nodes can be of the form $x_j > \text{constant}$
- Divides the space into axis-aligned rectangles

6.034 - Spring 03 • 2

Numerical Attributes

- Tests in nodes can be of the form $x_j > \text{constant}$
- Divides the space into axis-aligned rectangles


Slide 10.2.4

So, in this example, at the top level, we split the space into two parts, according to whether feature 1 has a value greater than 2. If not, then the output is 1.

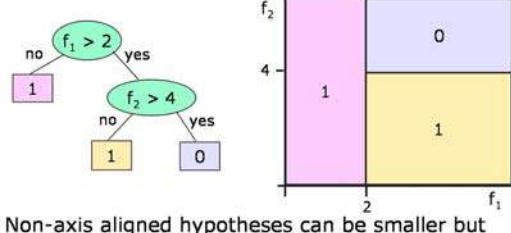
6.034 - Spring 03 • 4

Slide 10.2.5

If f_1 is greater than 2, then we have another split, this time on whether f_2 is greater than 4. If it is, the answer is 0, otherwise, it is 1. You can see the corresponding rectangles in the two-dimensional feature space.

Numerical Attributes

- Tests in nodes can be of the form $x_j > \text{constant}$
- Divides the space into axis-aligned rectangles



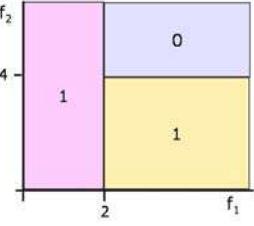
- Non-axis aligned hypotheses can be smaller but hard to find

6.034 - Spring 03 • 5

Slide 10.2.6**Slide 10.2.6**

This class of hypotheses is fairly rich, but it can be hard to express some concepts.

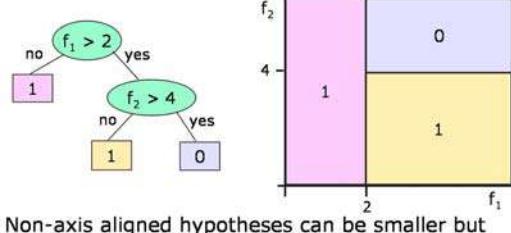
There are fancier versions of numeric decision trees that allow splits to be arbitrary hyperplanes (allowing us, for example, to make a split along a diagonal line in the 2D case), but we won't pursue them in this class.



6.034 - Spring 03 • 5

Numerical Attributes

- Tests in nodes can be of the form $x_j > \text{constant}$
- Divides the space into axis-aligned rectangles



- Non-axis aligned hypotheses can be smaller but hard to find

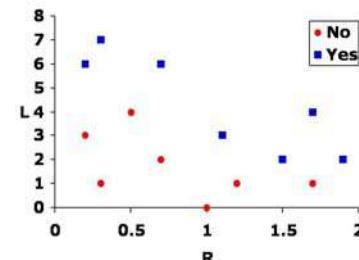
6.034 - Spring 03 • 6

Slide 10.2.7

The only thing we really need to do differently in our algorithm is to consider splitting between each data point in each dimension.

Considering Splits

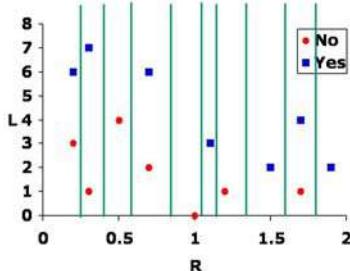
- Consider a split between each point in each dimension



6.034 - Spring 03 • 7

Considering Splits

- Consider a split between each point in each dimension



6.034 - Spring 03 • 8

Slide 10.2.8

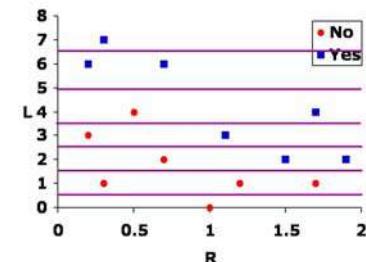
So, in our bankruptcy domain, we'd consider 9 different splits in the R dimension (in general, you'd expect to consider $m - 1$ splits, if you have m data points; but in our data set we have some examples with equal R values).

Slide 10.2.9

And there are another 6 possible splits in the L dimension (because L is an integer, really, there are lots of duplicate L values).

Considering Splits

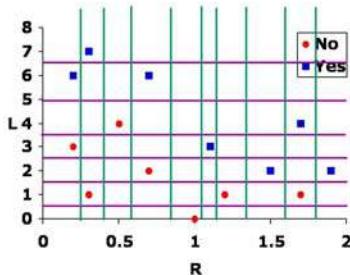
- Consider a split between each point in each dimension



6.034 - Spring 03 • 9

Considering Splits

- Choose split that minimizes average entropy of child nodes



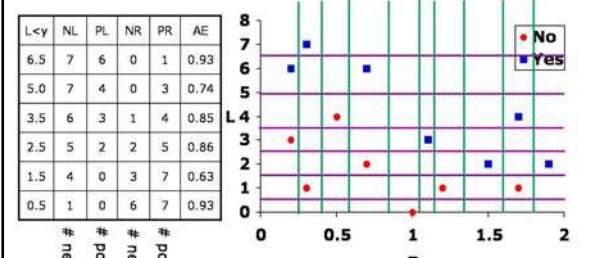
6.034 - Spring 03 • 10

Slide 10.2.10

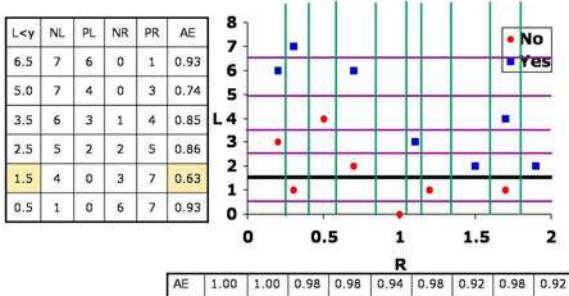
All together, this is a lot of possible splits! As before, when building a tree, we'll choose the split that minimizes the average entropy of the resulting child nodes.

Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93



6.034 - Spring 03 • 11

Bankruptcy Example

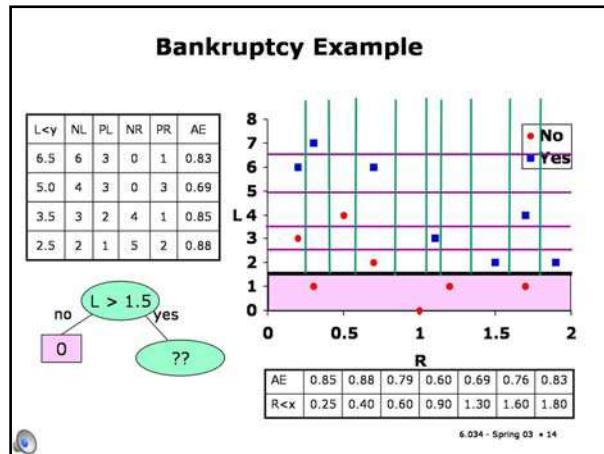
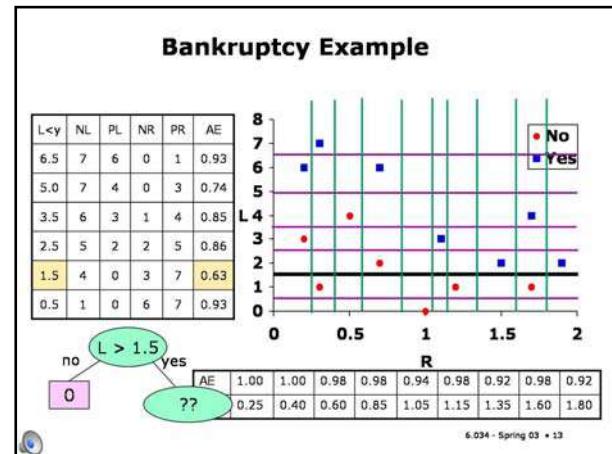
6.034 - Spring 03 • 12

Slide 10.2.12

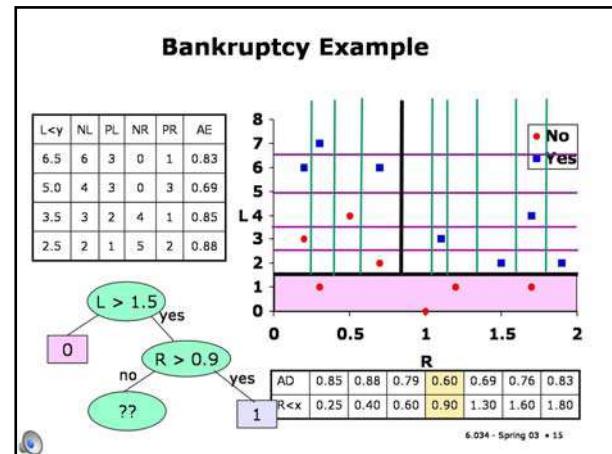
Splitting in the L dimension at 1.5 will do the best job of reducing entropy, so we pick that split.

Slide 10.2.13

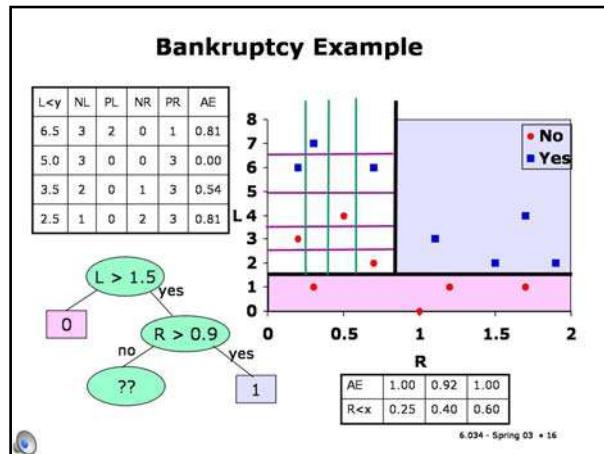
And we see that, conveniently, all the points with L not greater than 1.5 are of class 0, so we can make a leaf there.

**Slide 10.2.14**

Now, we consider all the splits of the remaining part of the space. Note that we have to recalculate all the average entropies again, because the points that fall into the leaf node are taken out of consideration.

**Slide 10.2.15**

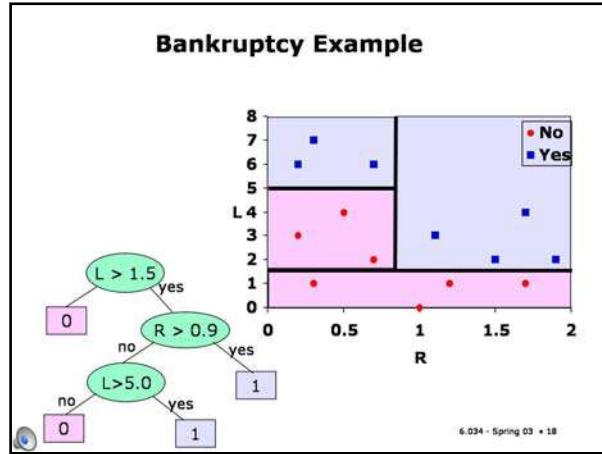
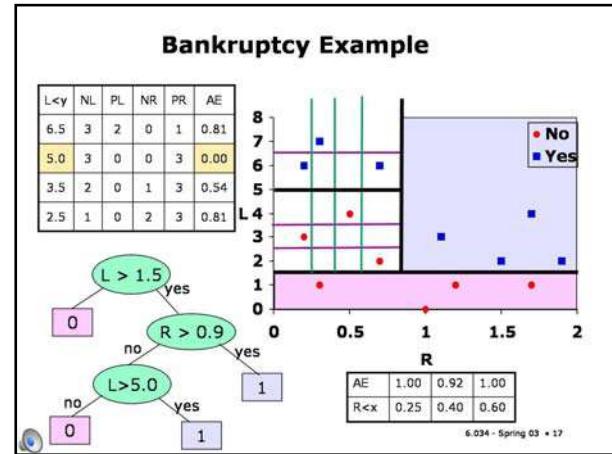
Now the best split is at $R > 0.9$. And we see that all the points for which that's true are positive, so we can make another leaf.

**Slide 10.2.16**

Again we consider all possible splits of the points that fall down the other branch of the tree.

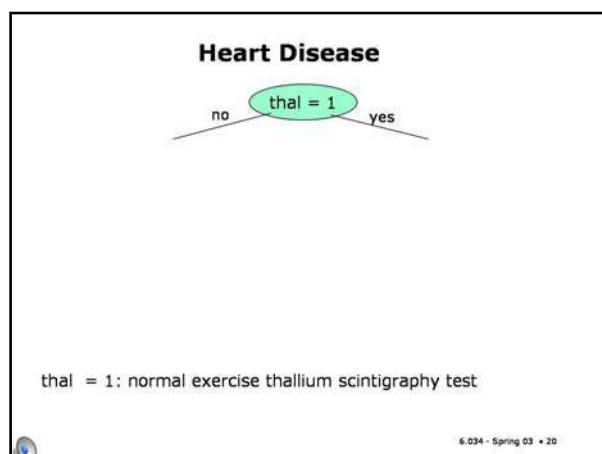
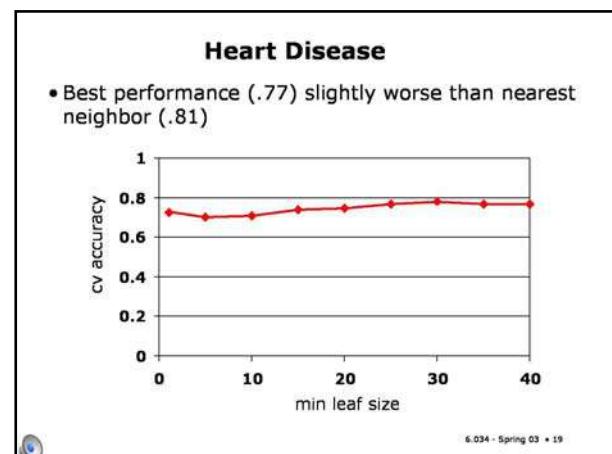
Slide 10.2.17

And we find that splitting on $L > 5.0$ gives us two homogenous leaves.

**Slide 10.2.19**

We ran this decision-tree algorithm on the heart-disease data set. This graph shows the cross-validation accuracy of the hypotheses generated by the decision-tree algorithm as a function of the min-leaf-size parameter, which stops splitting when the number of examples in a leaf gets below the specified size.

The best performance of this algorithm is about .77, which is slightly worse than the performance of nearest neighbor.

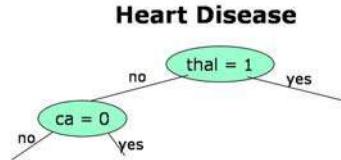
**Slide 10.2.20**

But performance isn't everything. One of the nice things about the decision tree algorithm is that we can interpret the hypothesis we get out. Here is an example decision tree resulting from the learning algorithm.

I'm not a doctor (and I don't even play one on TV), but the tree at least kind of makes sense. The top-level split is on whether a certain kind of stress test, called "thal" comes out normal.

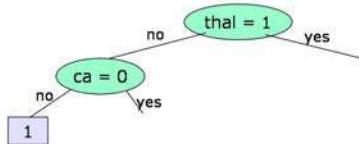
Slide 10.2.21

If thal is not normal, then we look at the results of the "ca" test. This test has as results numbers 0 through 3, indicating how many blood vessels were shown to be blocked in a different test. We chose to code this feature with 4 binary attributes.



thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy

6.034 - Spring 03 • 21

Heart Disease

thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy

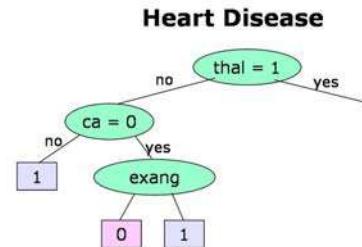
6.034 - Spring 03 • 22

Slide 10.2.22

So "ca = 0" is false if 1 or more blood vessels appeared to be blocked. If that's the case, we assert that the patient has heart disease.

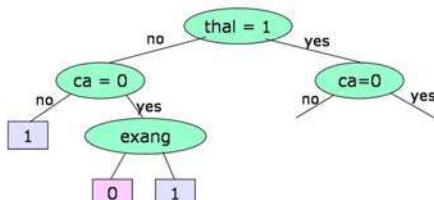
Slide 10.2.23

Now, if no blood vessels appeared to be blocked, we ask whether the patient is having exercise-induced angina (chest pain) or not. If not, we say they don't have heart disease; if so, we say they do.



thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina

6.034 - Spring 03 • 23

Heart Disease

thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina

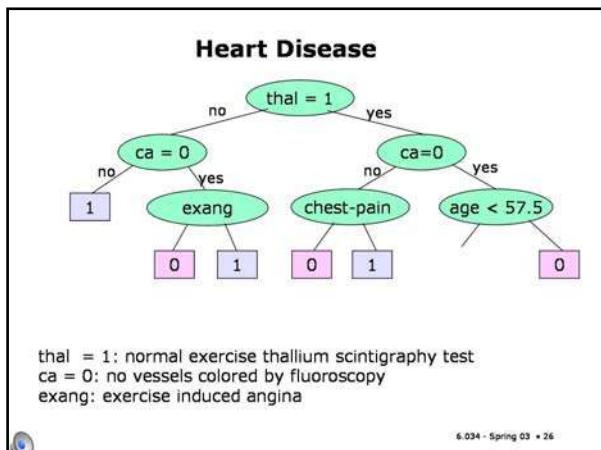
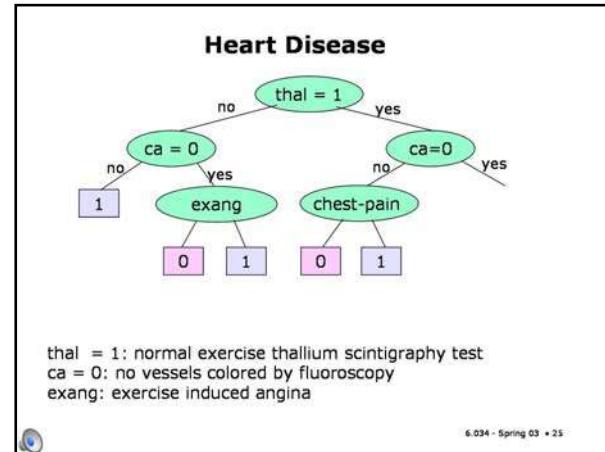
6.034 - Spring 03 • 24

Slide 10.2.24

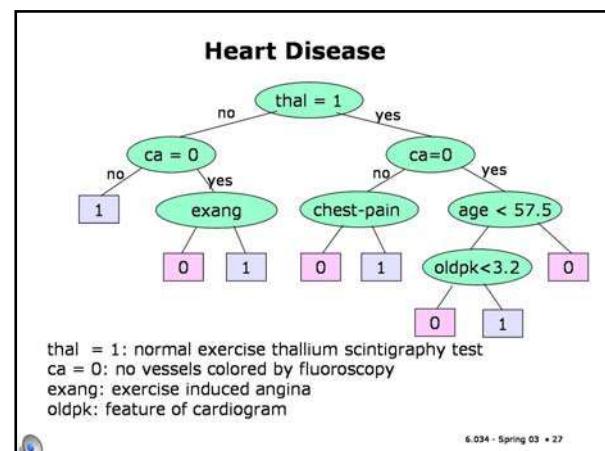
Now, over on the other side of the tree, where the first test was normal, we also look at the results of the ca test.

Slide 10.2.25

If it doesn't have value 0 (that is one or more vessels appear blocked), then we ask whether they have chest pain (presumably this is resting, not exercise-induced chest pain), and that determines the output.

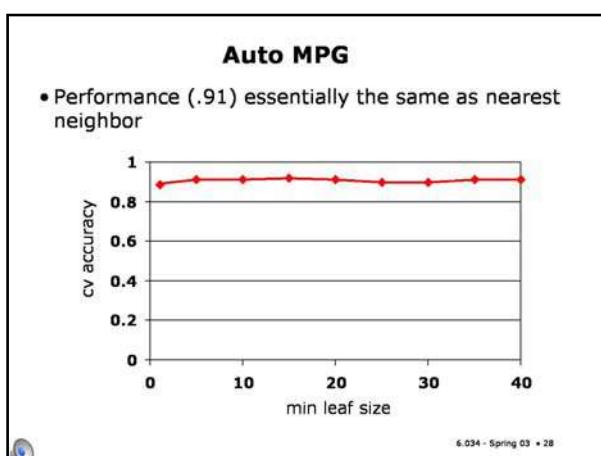
**Slide 10.2.26**

If no blood vessels appear to be blocked, we consider the person's age. If they're less than 57.5, then we declare them to be heart-disease free. Whew!

**Slide 10.2.27**

If they're older than 57.5, then we examine some technical feature of the cardiogram, and let that determine the output.

Hypotheses like this are very important in real domains. A hospital would be much more likely to base or change their policy for admitting emergency-room patients who seem to be having heart problems based on a hypothesis that they can see and interpret rather than based on the sort of numerical gobbledegook that comes out of nearest neighbor or naive Bayes.

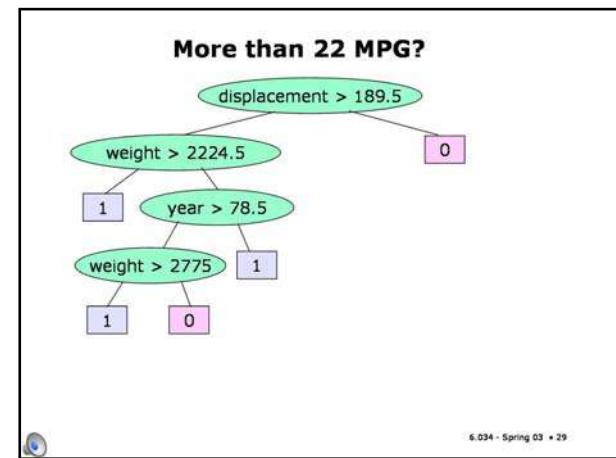
**Slide 10.2.28**

We also ran the decision-tree algorithm on the Auto MPG data. We got essentially the same performance as nearest neighbor, and a strong insensitivity to leaf size.

Slide 10.2.29

Here's a sample resulting decision tree. It seems pretty reasonable. If the engine is big, then we're unlikely to have good gas mileage. Otherwise, if the weight is low, then we probably do have good gas mileage. For a low-displacement, heavy car, we consider the model-year. If it's newer than 1978.5 (this is an old data set!) then we predict it will have good gas mileage. And if it's older, then we make a final split based on whether or not it's really heavy.

It's also possible to apply naive bayes to problems with numeric attributes, but it's hard to justify without recourse to probability, so we'll skip it. To do: add a slide showing how one non-isothetic split would do the job, but it requires a lot of rectangles.



6.034 - Spring 03 • 29

6.034 Notes: Section 10.3**Slide 10.3.1**

So far, we've spent all of our time looking at classification problems, in which the y values are either 0 or 1. Now we'll briefly consider the case where the y's are numeric values. We'll see how to extend nearest neighbor and decision trees to solve regression problems.

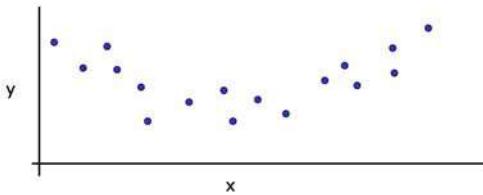
Regression

- Output is a continuous numeric value
 - Locally-weighted averaging
 - Regression trees

6.034 - Spring 03 • 1

Local Averaging

- Remember all your data

**Slide 10.3.2**

The simplest method for doing regression is based on nearest neighbor. As in nearest neighbor, you remember all your data.

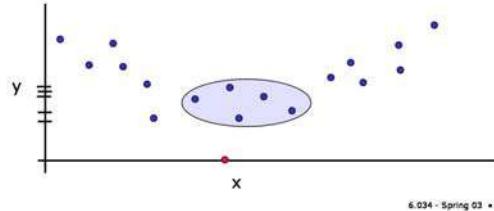
6.034 - Spring 03 • 2

Slide 10.3.3

When you get a new query point x , you find the k nearest points.

Local Averaging

- Remember all your data
- When someone asks a question,
– find the K nearest old data points

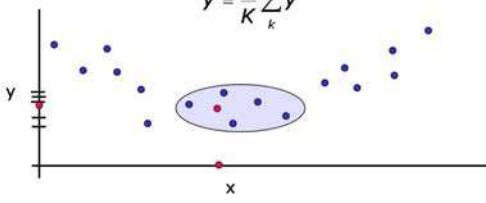


6.034 - Spring 03 • 3

Local Averaging

- Remember all your data
- When someone asks a question,
– find the K nearest old data points
– return the average of the answers associated with them

$$y = \frac{1}{K} \sum_k y^k$$



6.034 - Spring 03 • 4

Slide 10.3.4

Then average their y values and return that as your answer.

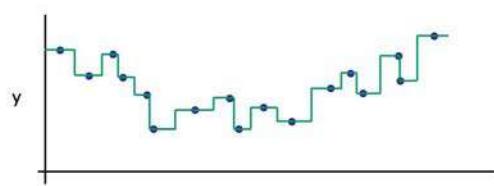
Of course, I'm showing this picture with a one-dimensional x , but the idea applies for higher-dimensional x , with the caveat that as the dimensionality of x increases, the curse of dimensionality is likely to be upon us.

Slide 10.3.5

When $k = 1$, this is like fitting a piecewise constant function to your data. It will track your data very closely, but, as in nearest neighbor, have high variance and be prone to overfitting.

 $K = 1$

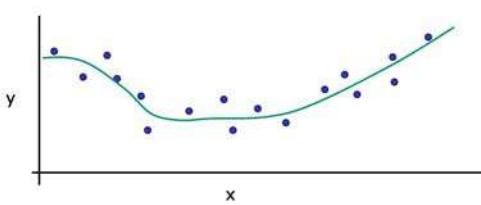
- Tracks data very closely
- Prone to overfitting



6.034 - Spring 03 • 5

Bigger K

- Smoothes out variations in data
- May introduce too much bias



6.034 - Spring 03 • 6

Slide 10.3.6

When k is larger, variations in the data will be smoothed out, but then there may be too much bias, making it hard to model the real variations in the function.

Slide 10.3.7

One problem with plain local averaging, especially as k gets large, is that we are letting all k neighbors have equal influence on the predicting the output of the query point. In locally weighted averaging, we still average the y values of multiple neighbors, but we weight them according to how close they are to the target point. That way, we let nearby points have a larger influence than farther ones.

Locally Weighted Averaging

6.034 - Spring 03 • 7

Locally Weighted Averaging

- Find all points within distance λ from target point
- Average the outputs, weighted according to how far away they are from the target point

Slide 10.3.8

The simplest way to describe locally weighted averaging involves finding all points that are within a distance λ from the target point, rather than finding the k nearest points. We'll describe it this way, but it's not too hard to go back and reformulate it to depend on the k nearest.

6.034 - Spring 03 • 8

Slide 10.3.9

Rather than committing to the details of the weighting function right now, let's just assume that we have a "kernel" function K , which takes the query point and a training point, and returns a weight, which indicates how much influence the y value of the training point should have on the predicted y value of the query point.

Then, to compute the predicted y value, we just add up all of the y values of the points used in the prediction, multiplied by their weights, and divide by the sum of the weights.

Locally Weighted Averaging

- Find all points within distance λ from target point
- Average the outputs, weighted according to how far away they are from the target point
- Given a target x , with k ranging over neighbors,

$$y = \frac{\sum_{k} K(x, x^k) y^k}{\sum_{k} K(x, x^k)}$$

weighting "kernel"

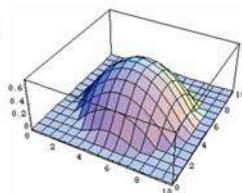
6.034 - Spring 03 • 9

Epanechnikov Kernel

- D is Euclidean distance

$$K(x, x^k) = \max\left(\frac{3}{4}\left(1 - \frac{D(x, x^k)^2}{\lambda^2}\right), 0\right)$$

- $x = <5, 5>$
- $\lambda = 4$



- Many other possible choices of kernel K

Slide 10.3.10

Here is one popular kernel, which is called the Epanechnikov kernel (I like to say that word!). You don't have to care too much about it; but see that it gives high weight to points that are near the query point (5,5 in this graph) and decreasing weights out to distance λ .

There are lots of other kernels which have various plusses and minuses, but the differences are too subtle for us to bother with at the moment.

6.034 - Spring 03 • 10

Slide 10.3.11

As usual, we have the same issue with lambda here as we have had with epsilon, min-leaf-size, and k. If it's too small, we'll have high variance; if it's too big, we'll have high bias. We can use cross-validation to choose.

In general, it's better to convert the algorithm to use k instead of lambda (it just requires making the lambda parameter in the kernel be the distance to the farthest of the k nearest neighbors). This means that we're always averaging the same number of points; so in regions where we have a lot of data, we'll look more locally, but in regions where the training data is sparse, we'll cast a wider net.

Smooth

- How should we choose λ ?
 - If small, then we aren't averaging many points
 - Worse at averaging out noise
 - Better at modeling discontinuities
 - If big, we are averaging a lot of points
 - Good at averaging out noise
 - Smears out discontinuities
- Can use cross-validation to choose λ
- May be better to let it vary according to local density of points

6.034 - Spring 03 • 11

Regression Trees

- Like decision trees, but with real-valued constant outputs at the leaves

Slide 10.3.12

Now we'll take a quick look at regression trees, which are like decision trees, but which have numeric constants at the leaves rather than booleans.

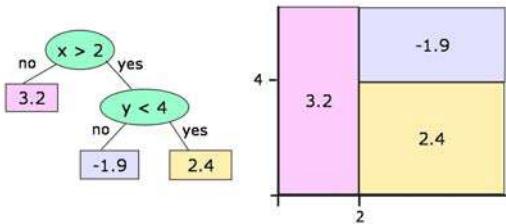
6.034 - Spring 03 • 12

Slide 10.3.13

Here's an example regression tree. It has the same kinds of splits as a regular tree (in this case, with numeric features), but what's different are the labels of the leaves.

Regression Trees

- Like decision trees, but with real-valued constant outputs at the leaves



6.034 - Spring 03 • 13

Leaf Values

- Assign a leaf node the average of the y values of the data points that fall there.

Slide 10.3.14

Let's start by thinking about how to assign a value to a leaf, assuming that multiple training points are in the leaf and we have decided, for whatever reason, to stop splitting.

In the boolean case, we used the majority output value as the value for the leaf. In the numeric case, we'll use the average output value. It makes sense, and besides there's a hairy statistical argument in favor of it, as well.

6.034 - Spring 03 • 14

Slide 10.3.15

So, if we're going to use the average value at a leaf as its output, we'd like to split up the data so that the leaf averages are not too far away from the actual items in the leaf.

Leaf Values

- Assign a leaf node the average of the y values of the data points that fall there.
- We'd like to have groups of points in a leaf that have similar y values (because then the average is a good representative)

6.034 - Spring 03 • 15

Variance

- Measure of how much a set of numbers is spread out

Slide 10.3.16

Lucky for us, the statistics folks have a good measure of how spread out a set of numbers is (and, therefore, how different the individuals are from the average); it's called the variance of a set.

6.034 - Spring 03 • 16

Slide 10.3.17

First we need to know the mean, which is traditionally called mu. It's just the average of the values. That is, the sum of the values divided by how many there are (which we call m, here).

Variance

- Measure of how much a set of numbers is spread out
- Mean of m values, z_1 through z_m :

$$\mu = \frac{1}{m} \sum_{k=1}^m z_k$$

6.034 - Spring 03 • 17

Variance

- Measure of how much a set of numbers is spread out
- Mean of m values, z_1 through z_m :

$$\mu = \frac{1}{m} \sum_{k=1}^m z_k$$

- Variance: average squared difference between z 's and the mean:

$$\sigma^2 = \frac{1}{m-1} \sum_{k=1}^m (z_k - \mu)^2$$

Slide 10.3.18

Then the variance is essentially the average of the squared distance between the individual values and the mean. If it's the average, then you might wonder why we're dividing by $m-1$ instead of m . I could tell you, but then I'd have to shoot you. Let's just say that dividing by $m-1$ makes it an unbiased estimator, which is a good thing.

6.034 - Spring 03 • 18

Slide 10.3.19

We're going to use the average variance of the children to evaluate the quality of splitting on a particular feature. Here we have a data set, for which I've just indicated the y values. It currently has a variance of 40.5.

Let's Split

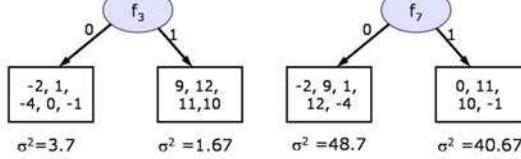
D: -2, 9, 1, 12, -4,

0, 11, 10, -1

$\sigma^2 = 40.5$

Let's SplitD: -2, 9, 1, 12, -4,
0, 11, 10, -1

$\sigma^2 = 40.5$



$\sigma^2 = 3.7$

$\sigma^2 = 1.67$

$\sigma^2 = 48.7$

$\sigma^2 = 40.67$

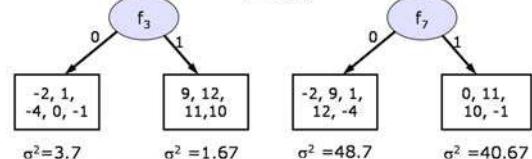
6.034 - Spring 03 • 19

Slide 10.3.20

We're considering two splits. One gives us variances of 3.7 and 1.67; the other gives us variances of 48.7 and 40.67.

Let's SplitD: -2, 9, 1, 12, -4,
0, 11, 10, -1

$\sigma^2 = 40.5$



$\sigma^2 = 3.7$

$\sigma^2 = 1.67$

$\sigma^2 = 48.7$

$\sigma^2 = 40.67$

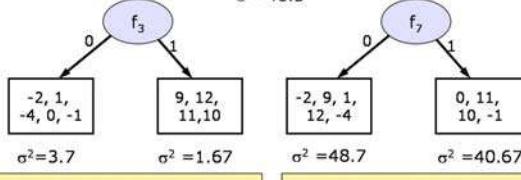
$AV(j) = p_j \sigma^2(D_j) + (1 - p_j) \sigma^2(\bar{D}_j)$

% of D with $f_j=1$ subset of D with $f_j=1$

6.034 - Spring 03 • 21

Let's SplitD: -2, 9, 1, 12, -4,
0, 11, 10, -1

$\sigma^2 = 40.5$



$$AV = (5/9)*3.7 + (4/9)*1.67 = 2.8$$

$$AV = (5/9)*48.7 + (4/9)*40.67 = 45.13$$

6.034 - Spring 03 • 22

Slide 10.3.22

Doing so, we can see that the average variance of splitting on feature 3 is **much** lower than of splitting on f7, and so we'd choose to split on f3.

Just looking at the data in the leaves, f3 seems to have done a much better job of dividing the values into similar groups.

Slide 10.3.23

We can stop growing the tree based on criteria that are similar to those we used in the binary case. One reasonable criterion is to stop when the variance at a leaf is lower than some threshold.

Stopping

- Stop when variance at a leaf is small enough

6.034 - Spring 03 • 23

Stopping

- Stop when variance at a leaf is small enough
- Or when you have fewer than min-leaf elements at a leaf

Slide 10.3.24

Or we can use our old min-leaf-size criterion.

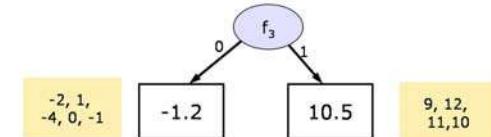
6.034 - Spring 03 • 24

Slide 10.3.25

Once we do decide to stop, we assign each leaf the average of the values of the points in it.

Stopping

- Stop when variance at a leaf is small enough
- Or when you have fewer than min-leaf elements at a leaf
- Set y at a leaf to be the mean of the y values of the elements



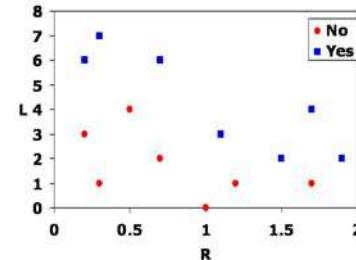
6.034 - Spring 03 • 25

6.034 Notes: Section 7.1

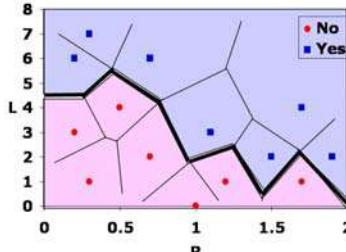
Slide 7.1.1

We have been using this simulated bankruptcy data set to illustrate the different learning algorithms that operate on continuous data. Recall that R is supposed to be the ratio of earnings to expenses while L is supposed to be the number of late payments on credit cards over the past year. We will continue using it in this section where we look at a new hypothesis class, **linear separators**.

One key observation is that each hypothesis class leads to a distinctive way of defining the **decision boundary** between the two classes. The decision boundary is where the class prediction changes from one class to another. Let's look at this in more detail.

Bankruptcy Example


6.034 - Spring 03 • 1

1-Nearest Neighbor Hypothesis


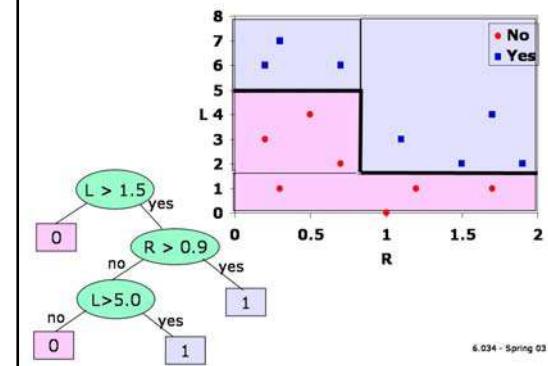
6.034 - Spring 03 • 2

Slide 7.1.2

We mentioned that a hypothesis for the 1-nearest neighbor algorithm can be understood in terms of a Voronoi partition of the feature space. The cells illustrated in this figure represent the feature space points that are closest to one of the training points. Any query in that cell will have that training point as its nearest neighbor and the prediction will be the class of that training point. The decision boundary will be the boundary between cells defined by points of different classes, as illustrated by the bold line shown here.

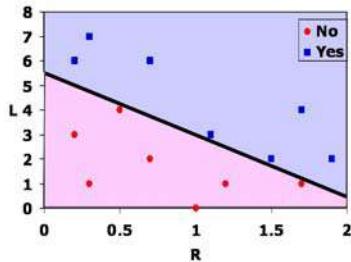
Slide 7.1.3

Similarly, a decision tree also defines a decision boundary in the feature space. Note that although both 1-NN and decision trees agree on all the training points, they disagree on the precise decision boundary and so will classify some query points differently. This is the essential difference between different learning algorithms.

Decision Tree Hypothesis


6.034 - Spring 03 • 3

Linear Hypothesis



Slide 7.1.4

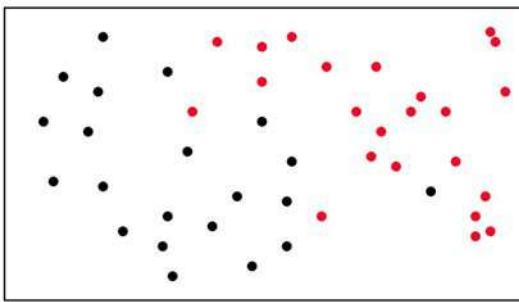
In this section we will be exploring **linear separators** which are characterized by a single linear decision boundary in the space. The bankruptcy data can be successfully separated in that manner. But, notice that in contrast to 1-NN and decision trees, there is no guarantee that a single linear separator will successfully classify any set of training data. The linear separator is a very simple hypothesis class, not nearly as powerful as either 1-NN or decision trees. However, as simple as this class is, in general, there will be many possible linear separators to choose from.

Also, note that, once again, this decision boundary disagrees with that drawn by the previous algorithms. So, there will be some data sets where a linear separator is ideally suited to the data. For example, it turns out that if the data points are generated by two Gaussian distributions with different means but the same standard deviation, then the linear separator is optimal.

Slide 7.1.5

A data set that can be successfully split by a linear separator is called, not surprisingly, **linearly separable**.

Not Linearly Separable



Slide 7.1.6

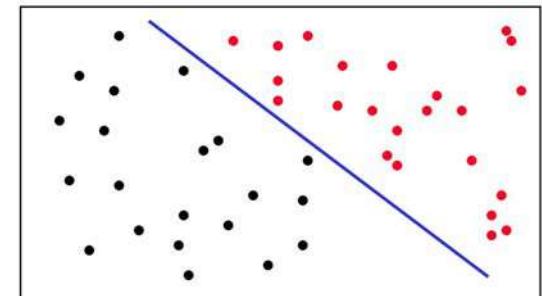
As we've mentioned, not all data sets are linearly separable. Here's one for example. Another classic non-linearly-separable data set is our old nemesis XOR.

It turns out, although it's not obvious, that the higher the dimensionality of the feature space, the more likely that a linear separator exists. This will turn out to be important later on, so let's just file that fact away.

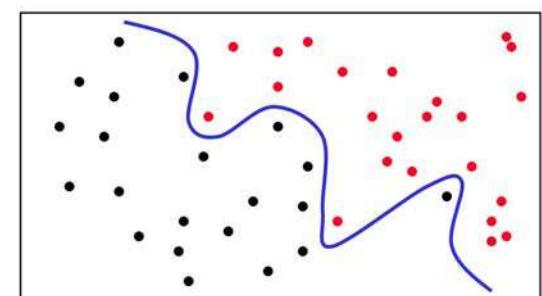
Slide 7.1.7

When faced with a non-linearly-separable data set, we have two options. One is to use a more complex hypothesis class, such as shown here.

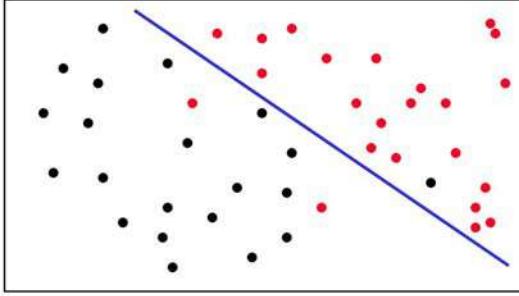
Linearly Separable



Not Linearly Separable



Not Linearly Separable



Slide 7.1.8

Or, keep the simple linear separator and accept some errors. This is the classic bias/variance tradeoff. Use a more complex hypothesis with greater variance or a simpler hypothesis with greater bias. Which is more appropriate depends on the underlying properties of the data, including the amount of noise. We can use our old friend cross-validation to make the choice if we don't have much understanding of the data.

Slide 7.1.9

So, let's look at the details of linear classifiers. First, we need to understand how to represent a particular hypothesis, that is, the equation of a linear separator. We will be illustrating everything in two dimensions but all the equations hold for an arbitrary number of dimensions.

The equation of a linear separator in an n -dimensional feature space is (surprise!) a linear equation which is determined by $n+1$ values, the components of an n -dimensional coefficient vector \mathbf{w} and a scalar value b . These $n+1$ values are what will be learned from the data. The \mathbf{x} will be some point in the feature space.

We will be using dot product notation for compactness and to highlight the geometric interpretation of this equation (more on this in a minute). Recall that the dot product is simply the sum of the componentwise products of the vector components, as shown here.

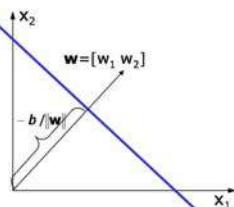
Linear Hypothesis Class

- Equation of a hyperplane in the feature space

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

$$\sum_{j=1}^n w_j x_j + b = 0$$

- \mathbf{w}, b are to be learned



Slide 7.1.10

In two dimensions, we can see the geometric interpretation of \mathbf{w} and b . The vector \mathbf{w} is perpendicular to the linear separator; such a vector is known as the **normal** vector. Often we say "the vector normal to the surface". The scalar b , which we will call the **offset**, is proportional to the perpendicular distance from the origin to the linear separator. The constant of proportionality is the negative of the magnitude of the normal vector. We'll examine this in more detail soon.

By the way, the choice of the letter "w" is traditional and meant to suggest "weights", we'll see why when we look at neural nets. The choice of "b" is meant to suggest "bias" - which is the third different connotation of this word in machine learning (the bias of a hypothesis class, bias vs. variance, bias of a separator). They are all fundamentally related; they all refer to a difference from a neutral value. To keep the confusion down to a dull roar, we won't call b a bias term but are telling you about this so you won't be surprised if you see it elsewhere.

Slide 7.1.11

Sometimes we will use the following trick to simplify the equations. We'll treat the offset as the 0th component of the weight vector \mathbf{w} and we'll augment the data vector \mathbf{x} with a 0th component that will always be equal to 1. Then we can write a linear equation as a dot product. When we do this, we will indicate it by using an overbar over the vectors.

Linear Hypothesis Class

- Equation of a hyperplane in the feature space

$$\overline{\mathbf{w}} \cdot \overline{\mathbf{x}} + b = 0$$

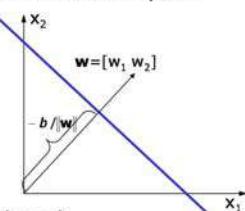
$$\sum_{j=1}^n w_j x_j + b = 0$$

- \mathbf{w}, b are to be learned

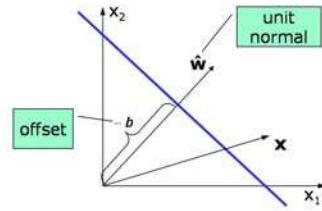
- A useful trick: let $x_0=1$ and $w_0=b$

$$\overline{\mathbf{w}} \cdot \overline{\mathbf{x}} = 0$$

$$\sum_{j=0}^n w_j x_j = 0$$



Hyperplane: Geometry



Slide 7.1.12

First a word on terminology: the equations we will be writing apply to linear separators in n dimensions. In two dimensions, such a linear separator is referred to as a "line". In three dimensions, it is called a "plane". These are familiar words. What do we call it in higher dimensions? The usual terminology is **hyperplane**. I know that sounds like some type of fast aircraft, but that's the accepted name.

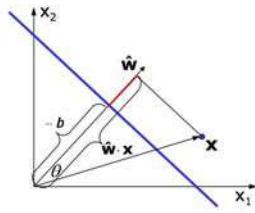
Let's look at the geometry of a hyperplane a bit more closely. We saw earlier that the offset b in the linear separator equation is proportional to the perpendicular distance from the origin to the linear separator and that the constant of proportionality is the magnitude of the \mathbf{w} vector (negated). Basically, we can multiply both sides of the equation by any number without affecting the equality. So, there are an infinite set of equations all of which represent the same separator.

If we divide the equation through by the magnitude of \mathbf{w} we end up with the situation shown in the figure. The normal vector is now unit length (denoted by the hat on the \mathbf{w}) and the offset b is now equal to the perpendicular distance from the origin (negated).

Hyperplane: Geometry

$$\hat{\mathbf{w}} \cdot \mathbf{x} + b$$

signed perpendicular
distance of point \mathbf{x} to
hyperplane.



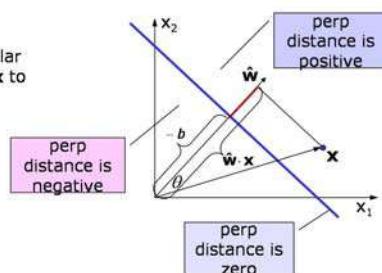
$$\text{recall: } \mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$$

6.034 - Spring 03 • 13

Hyperplane: Geometry

$$\hat{\mathbf{w}} \cdot \mathbf{x} + b$$

signed perpendicular
distance of point \mathbf{x} to
hyperplane.



$$\text{recall: } \mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$$

6.034 - Spring 03 • 14

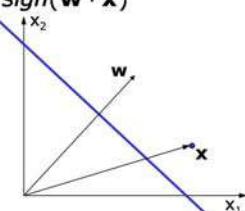
Slide 7.1.14

This distance measure from the hyperplane is **signed**. It is zero for points on the hyperplane, it is positive for points in the side of the space towards which the normal vector points, and negative for points on the other side. Notice that if you multiply the normal vector \mathbf{w} and the offset b by -1, you get an equation for the same hyperplane but you switch which side of the hyperplane has positive distances.

Linear Classifier

$$h(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \equiv \text{sign}(\mathbf{w} \cdot \mathbf{x})$$

outputs +1 or -1



6.034 - Spring 03 • 15

Linear Classifier

Margin:

$$\gamma_i = y^i(\mathbf{w} \cdot \mathbf{x}^i + b) = y^i \mathbf{w} \cdot \mathbf{x}^i$$

proportional to perpendicular distance of point \mathbf{x}^i to hyperplane.

$\gamma_i > 0$: point is correctly classified (sign of distance = y^i)

$\gamma_i < 0$: point is incorrectly classified (sign of distance ≠ y^i)

6.034 - Spring 03 • 16

Slide 7.1.16

A variant of the signed distance of a training point to a hyperplane is the **margin** of the point. The margin (gamma) is the product of the actual signed distance for the point and the desired sign of the distance, y_i . If they agree (the point is correctly classified), then the margin is positive; if they disagree (the classification is in error), then the margin is negative.

6.034 Notes: Section 7.2**Slide 7.2.1**

So far we've talked about how to represent a linear hypothesis but not how to find one. In this slide is the perceptron algorithm, developed by Rosenblatt in the mid 50's. This is not exactly the original form of the algorithm but it is equivalent and it will help us later to see it in this form.

This is a greedy, "mistake driven" algorithm not unlike the Boolean function learning algorithms we saw earlier. We will be using the extended form of the weight and data-point vectors in this algorithm. The extended weight vector is what we are trying to learn.

The first step is to start with an initial value of the weight vector, usually all zeros. Then we repeat the inner loop until all the points are correctly classified using the current weight vector. The inner loop is to consider each point. If the point's margin is positive then it is correctly classified and we do nothing. Otherwise, if it is negative or zero, we have a mistake and we want to change the weights so as to increase the margin (so that it ultimately becomes positive).

The trick is how to change the weights. It turns out that using a value proportional to $y\mathbf{x}$ is the right thing. We'll see why, formally, later. For now, let's convince ourselves that it makes sense.

Perceptron Algorithm
Rosenblatt, 1956

- Pick initial weight vector (including b), e.g. $[0 \dots 0]$
- Repeat until all points correctly classified
 - Repeat for each point
 - Calculate margin ($y^i \mathbf{w} \cdot \mathbf{x}^i$) for point i
 - If margin > 0, point is correctly classified
 - Else change weights to increase margin; change in weight proportional to $y^i \mathbf{x}^i$

6.034 - Spring 03 • 16

Perceptron Algorithm
Rosenblatt, 1956

- Pick initial weight vector (including b), e.g. $[0 \dots 0]$
- Repeat until all points correctly classified
 - Repeat for each point
 - Calculate margin ($y^i \mathbf{w} \cdot \mathbf{x}^i$) for point i
 - If margin > 0, point is correctly classified
 - Else change weights to increase margin; change in weight proportional to $y^i \mathbf{x}^i$
- Note that, if $y^i = 1$
 - if $x_j^i > 0$ then w_j increased (increases margin)
 - if $x_j^i < 0$ then w_j decreased (increases margin)
- And, similarly for $y^i = -1$

6.034 - Spring 03 • 2

Slide 7.2.2

Consider the case in which y is positive; the negative case is analogous. If the j th component of \mathbf{x} is positive then we will increase the corresponding component of \mathbf{w} . Note that the resulting effect on the margin is positive. If the j th component of \mathbf{x} is negative then we will decrease the corresponding component of \mathbf{w} , and the resulting effect on the margin is also positive.

Slide 7.2.3

So, each change of w increases the margin on a particular point. However, the changes for the different points interfere with each other, that is, different points might change the weights in opposing directions. So, it will not be the case that one pass through the points will produce a correct weight vector. In general, we will have to go around multiple times.

The remarkable fact is that the algorithm is guaranteed to terminate with the weights for a separating hyperplane as long as the data is linearly separable. The proof of this fact is beyond our scope.

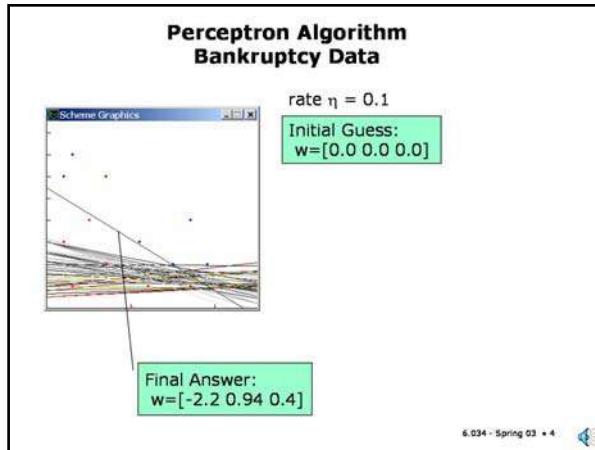
Notice that if the data is not separable, then this algorithm is an infinite loop. It turns out that it is a good idea to keep track of the best separator you've seen so far (the one that makes the fewest mistakes) and after you get tired of going around the loop, return that one. This algorithm even has a name (the **pocket** algorithm: see, it keeps the best answer in its pocket...).

Perceptron Algorithm

Rosenblatt, 1956

- Pick initial weight vector (including b), e.g. $[0 \dots 0]$
- Repeat until all points correctly classified
 - Repeat for each point
 - Calculate margin ($y'w$) for point i
 - If margin > 0, point is correctly classified
 - Else change weights to increase margin; change in weight proportional to $y'x$
- Note that, if $y' = 1$
 - if $x_j > 0$ then w_j increased (increases margin)
 - if $x_j < 0$ then w_j decreased (increases margin)
- And, similarly for $y' = -1$
- Guaranteed to find separating hyperplane if one exists
- Otherwise, data are not linearly separable, loops forever

6.034 - Spring 03 • 3

**Slide 7.2.4**

This shows a trace of the perceptron algorithm on the bankruptcy data. Here it took 49 iterations through the data (the outer loop) for the algorithm to stop. The hypothesis at the end of each loop is shown here. Recall that the first element of the weight vector is actually the offset. So, the normal vector to the separating hyperplane is $[0.94\ 0.4]$ and the offset is -2.2 (recall that is proportional to the negative perpendicular distance from origin to the line).

Note that the units in the horizontal and vertical directions in this graph are not equal (the tick marks along the axes indicate unit distances). We did this since the range of the data on each axis is so different.

One usually picks some small "rate" constant to scale the change to w . It turns out that for this algorithm the value of the rate constant does not matter. We have used 0.1 in our examples, but 1 also works well.

Slide 7.2.5

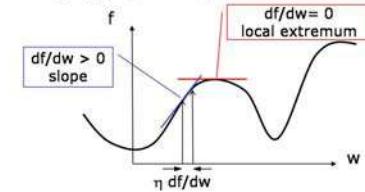
Let's revisit the issue of why we picked yx to increment w in the perceptron algorithm. It might have seemed arbitrary but it's actually an instance of a general strategy called **gradient ascent** for finding the input(s) that maximize a function's output (or gradient descent when we are minimizing).

The strategy in one input dimension is shown here. We guess an initial value of the input. We calculate the slope of the function at that input value and we take a step that is proportional to the slope. Note that the sign of the slope will tell us whether an increase of the input variable will increase or decrease the value of the output. The magnitude of the slope will tell us how fast the function is changing at that input value. The slope is basically a linear approximation of the function which is valid "near" the chosen input value. Since the approximation is only valid locally, we want to take a small step (determined by the rate constant eta) and repeat.

We want to stop when the output change is zero (or very small). This should correspond to a point where the slope is zero, which should be a local extremum of the function. This strategy will not guarantee finding the global maximal value, only a local one.

Gradient Ascent

- Why pick $y'x$ as increment to weights?
- To maximize scalar function of one variable $f(w)$
 - Pick initial w
 - Change w to $w + \eta df/dw$ ($\eta > 0$, small)
 - until f stops changing ($df/dw \approx 0$)

**Slide 7.2.6**

The generalization of this strategy to multiple input variables is based on the generalization of the notion of slope, which is the **gradient** of the function. The gradient is the vector of first (partial) derivatives of the function with respect to each of the input variables. The gradient vector points in the direction of steepest increase of the function output. So, we take a small step in that direction, recompute the gradient and repeat until the output stops changing. Once again, this will only find us a local maximum of the function, in general. However, if the function is globally convex, then it will find the global optimum.

Gradient Ascent/Descent

- To maximize $f(w)$ $\nabla_w f = \left[\frac{\partial f}{\partial w_1}, \dots, \frac{\partial f}{\partial w_n} \right]$
 - Pick initial w
 - Change w to $w + \eta \nabla_w f$ ($\eta > 0$, small)
 - until f stops changing ($\nabla_w f \approx 0$)
 - Finds local maximum; global maximum if function is globally convex.

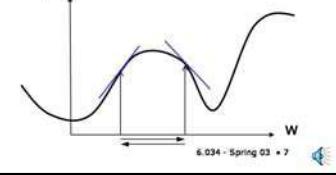
Slide 7.2.7

In general, the choice of the rate constant (η), which determines the step size, is fairly critical. Unfortunately, no single value is appropriate for all functions. If one chooses a very conservative small rate, it can take a long time to find a minimum, if one takes too big steps there is no guarantee that the algorithm will even converge to a minimum; it can oscillate as shown in the figure here where the sign of the slope changes and causes a back-and-forth search.

In more sophisticated search algorithms one does a search along the specified direction looking for a value of the step size that guarantees an increase in the function value.

Gradient Ascent/Descent

- To maximize $f(\mathbf{w})$ $\nabla_{\mathbf{w}} f = \left[\frac{\partial f}{\partial w_1}, \dots, \frac{\partial f}{\partial w_n} \right]$
 - Pick initial \mathbf{w}
 - Change \mathbf{w} to $\mathbf{w} + \eta \nabla_{\mathbf{w}} f$ ($\eta > 0$, small)
 - until f stops changing ($\nabla_{\mathbf{w}} f \approx 0$)
- Finds local maximum; global maximum if function is globally convex
- Rate (η) has to be chosen carefully.
 - Too small – slow convergence
 - Too big – oscillation

**Perceptron Training via Gradient Descent**

- Maximize sum of margins of misclassified points

$$f(\mathbf{w}) = \sum_{i \text{ misclassified}} y^i \mathbf{x}^i$$

$$\nabla_{\mathbf{w}} f = \sum_{i \text{ misclassified}} y^i \mathbf{x}^i$$

6.034 - Spring 03 • 8

Slide 7.2.8**Slide 7.2.8**

Now we can see that our choice of increment in the perceptron algorithm is related to the gradient of the sum of the margins for the misclassified points.

Slide 7.2.9

If we actually want to maximize this sum via gradient descent we should sum all the corrections for every misclassified point using a single \mathbf{w} vector and then apply that correction to get a new weight vector. We can then repeat the process until convergence. This is normally called an **off-line** algorithm in that it assumes access to all the input points.

What we actually did was a bit different, we modified \mathbf{w} based on each point as we went through the inner loop. This is called an **on-line** algorithm because, in principle, if the points were arriving over a communication link, we would make our update to the weights based on each arrival and we could discard the points after using them, counting on more arriving later.

Another way of thinking about the relationship of these algorithms is that the on-line version is using a (randomized) approximation to the gradient at each point. It is randomized in the sense that rather than taking a step based on the true gradient, we take a step based on an estimate of the gradient based on a randomly drawn example point. In fact, the on-line version is sometimes called "stochastic (randomized) gradient ascent" for this reason. In some cases, this randomness is good because it can get us out of shallow local minima.

Perceptron Training via Gradient Descent

- Maximize sum of margins of misclassified points

$$f(\mathbf{w}) = \sum_{i \text{ misclassified}} y^i \mathbf{x}^i$$

$$\nabla_{\mathbf{w}} f = \sum_{i \text{ misclassified}} y^i \mathbf{x}^i$$

- Off-line training: Compute gradient as sum over all training points.
- On-line training: Approximate gradient by one of the terms in the sum: $y^i \mathbf{x}^i$

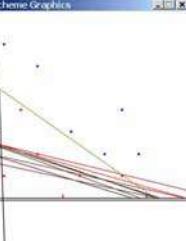
6.034 - Spring 03 • 9

Perceptron Algorithm Bankruptcy Datarate $\eta = 0.1$

w_0	w_1	w_2
-1.0	1.00	1.0
-1.3	0.71	0.6
-1.4	0.66	0.5
-1.4	0.66	0.8
-1.5	0.61	0.7
-1.6	0.56	0.6
-1.6	0.65	0.6
-1.6	0.74	0.6
-1.6	0.83	0.6
-1.7	0.81	0.3

Initial Guess:
 $w = [-1.0 \ 1.0 \ 1.0]$

Scheme Graphics

Final Answer:
 $w = [-1.7 \ 0.81 \ 0.3]$

6.034 - Spring 03 • 10

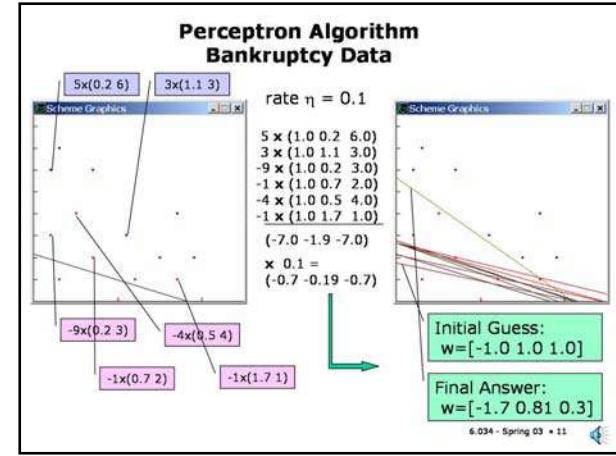
Slide 7.2.10

Here's another look at the perceptron algorithm on the bankruptcy data with a different initial starting guess of the weights. You can see the different separator hypotheses that it goes through. Note that it converges to a different set of weights from our previous example. However, recall that one can scale these weights and get the same separator. In fact these numbers are approximately 0.8 of the ones we got before, but only approximately; this is a slightly different separator.

The perceptron algorithm can be described as a gradient ascent algorithm, but its error criterion is slightly unusual in that there are many separators that all have zero error.

Slide 7.2.11

Recall that the perceptron algorithm starts with an initial guess for the weights and then adds in scaled versions of the misclassified training points to get the final weights. In this particular set of 10 iterations, the points indicated on the left are misclassified some number of times each. For example, the leftmost negative point is misclassified in each iteration except the last one. If we sum up the coordinates of each of these points, scaled by how many times each is misclassified and by the rate constant we get the total change in the weight vector.

**Dual Form**

Assume initial weights are 0; rate= $\eta > 0$

$$\begin{aligned} 5x(1.0 0.2 6.0) & \quad \alpha_1 y_1 \bar{x}_1 \\ 3x(1.0 1.1 3.0) & \quad \alpha_2 y_2 \bar{x}_2 \\ -9x(1.0 0.2 3.0) & \quad \alpha_3 y_3 \bar{x}_3 \\ -1x(1.0 0.7 2.0) & \quad \alpha_4 y_4 \bar{x}_4 \\ -4x(1.0 0.5 4.0) & \quad \alpha_5 y_5 \bar{x}_5 \\ -1x(1.0 1.7 1.0) & \quad \alpha_6 y_6 \bar{x}_6 \\ \hline (-7.0 -1.9 -7.0) \times 0.1 & \quad \alpha_7 y_7 \bar{x}_7 \\ = (-0.7 -0.19 -0.7) & \quad \alpha_8 y_8 \bar{x}_8 \\ & \quad \alpha_i \text{ is count of mistakes on point } i \text{ during training} \end{aligned}$$

6.034 - Spring 03 • 12

Slide 7.2.12

Since the rate constant does not change the separator we can simply assume that it is 1 and ignore it. Now, we can substitute this form of the weights in the classifier and we get the classifier at the bottom of the slide, which has the interesting property that the data points only appear in dot-products with other data points. This will turn out to be extremely important later; file this one away.

**Perceptron Training
Dual Form**

- $\alpha = 0$
- Repeat until all points correctly classified
 - Repeat for each point i
 - Calculate margin $\sum_{j=1}^m \alpha_j y_j \bar{x}^j \cdot \bar{x}^i$
 - If margin > 0 , point is correctly classified
 - Else increment α_i
 - Return $\mathbf{w} = \sum_{j=1}^m \alpha_j y_j \bar{x}^j$
 - If data is not linearly separable, the α_i grow without bound

6.034 - Spring 03 • 14

Slide 7.2.12

This analysis leads us to a somewhat different view of the perceptron algorithm, usually called the **dual form** of the algorithm. Call the count of how many times point i is misclassified, α_i . Then, assuming the weight vector is initialized to 0s, we can write the final weight vector in terms of these counts and the input data (as well as the rate constant).

Dual Form

Assume initial weights are 0; rate= $\eta > 0$

$$\begin{aligned} 5x(1.0 0.2 6.0) & \quad \alpha_1 y_1 \bar{x}^1 \\ 3x(1.0 1.1 3.0) & \quad \alpha_2 y_2 \bar{x}^2 \\ -9x(1.0 0.2 3.0) & \quad \alpha_3 y_3 \bar{x}^3 \\ -1x(1.0 0.7 2.0) & \quad \alpha_4 y_4 \bar{x}^4 \\ -4x(1.0 0.5 4.0) & \quad \alpha_5 y_5 \bar{x}^5 \\ -1x(1.0 1.7 1.0) & \quad \alpha_6 y_6 \bar{x}^6 \\ \hline (-7.0 -1.9 -7.0) \times 0.1 & \quad \alpha_7 y_7 \bar{x}^7 \\ = (-0.7 -0.19 -0.7) & \quad \alpha_8 y_8 \bar{x}^8 \\ & \quad \alpha_i \text{ is count of mistakes on point } i \text{ during training} \end{aligned}$$

η just scales answer, set to 1

$$h(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x}) = \text{sign}\left(\sum_{i=1}^m \alpha_i y_i \bar{x}^i \cdot \mathbf{x}\right)$$

6.034 - Spring 03 • 13

Slide 7.2.14

We can now restate the perceptron algorithm in this interesting way. The separator is described as a weighted sum of the input points, with α_i the weight for point i . Initially, set all of the alphas to zero, so the separator has all zero's as coefficients.

Then, for each point, compute its margin with respect to the current separator. If the margin is positive, the point is classified correctly, so do nothing. If the margin is negative, add that point into the weights of the separator. We can do that simply by incrementing the associated alpha.

Finally, when all of the points are classified correctly, we return the weighted sum of the inputs as the coefficients for the separator. Note that if the data is not linearly separable, then the algorithm will loop forever, the alphas growing without bound.

You should convince yourself that this dual form is equivalent to the original. Once again, you may be wondering...so what? I'll say again; file this away. It has surprising consequences.

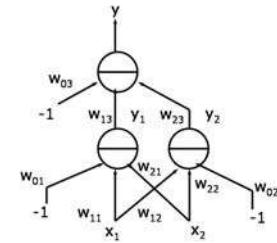
6.034 Notes: Section 7.3

Slide 7.3.1

We will now turn our attention to artificial neural nets, sometimes also called "feedforward nets".

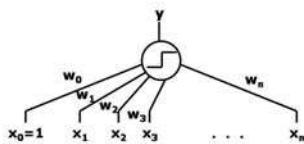
The basic idea in neural nets is to define interconnected networks of simple units (let's call them "artificial neurons") in which each connection has a weight. Weight w_{ij} is the weight of the i^{th} input into unit j . The networks have some inputs where the feature values are placed and they compute one or more output values. The learning takes place by adjusting the weights in the network so that the desired output is produced whenever a sample in the input data set is presented.

Artificial Neural Networks (Feedforward Nets)



6.034 - Spring 03 • 1

Single Perceptron Unit



6.034 - Spring 03 • 2

Slide 7.3.2

We start by looking at a simpler kind of "neural-like" unit called a **perceptron**. This is where the perceptron algorithm that we saw earlier came from. Perceptrons antedate the modern neural nets. Examining them can help us understand how the more general units work.

Slide 7.3.3

A perceptron unit basically compares a weighted combination of its inputs against a threshold value and then outputs a 1 if the weighted inputs exceed the threshold. We use our trick here of treating the (arbitrary) threshold as if it were a weight (w_0) on a constant input (x_0) whose value is -1 (note the sign is different from what we saw in our previous treatment but the idea is the same). In this way, we can write the basic rule of operation as computing the weighted sum of all the inputs and comparing to 0.

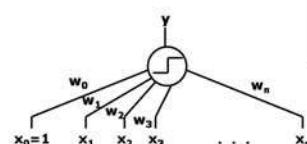
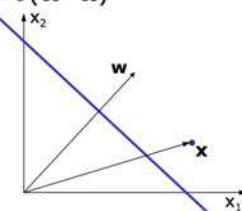
The key observation is that the decision boundary for a single perceptron unit is a hyperplane in the feature space. That is, it is a linear equation that divides the space into two half-spaces. We can easily see this in two dimensions. The equation that tells us when the perceptron's total input goes to zero is the equation of a line whose normal is the weight vector $[w_1 \ w_2]$. On one side of this line, the value of the weighted input is negative and so the perceptron's output is 0, on the other side of the line the weighted input is positive and the output is 1.

We have seen that there's a simple gradient-descent algorithm for finding such a linear separator if one exists.

Linear Classifier Single Perceptron Unit

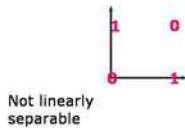
$$h(\mathbf{x}) = \theta(\mathbf{w} \cdot \mathbf{x} + b) = \theta(\mathbf{w} \cdot \mathbf{x})$$

$$\theta(z) = \begin{cases} 1 & z \geq 0 \\ 0 & \text{else} \end{cases}$$



6.034 - Spring 03 • 3

Beyond Linear Separability



Slide 7.3.4

Since a single perceptron unit can only define a single linear boundary, it is limited to solving linearly separable problems. A problem like that illustrated by the values of the XOR boolean function cannot be solved by a single perceptron unit.

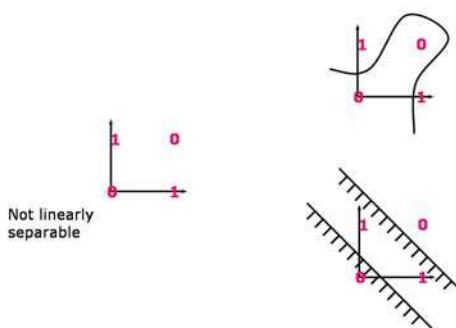
6.034 - Spring 03 • 4



Slide 7.3.5

We have already seen in our treatment of SVMs how the "kernel trick" can be used to generalize a perceptron-like classifier to produce arbitrary boundaries, basically by mapping into a high-dimensional space of non-linear mappings of the input features.

Beyond Linear Separability



Slide 7.3.6

We will now explore a different approach (although later we will also introduce non-linear mappings). What about if we consider more than one linear separator and combine their outputs; can we get a more powerful classifier?

6.034 - Spring 03 • 6

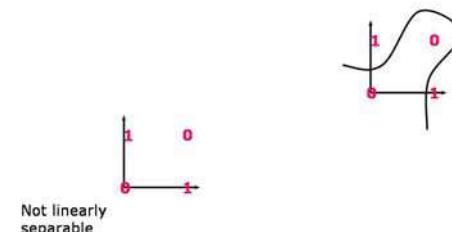


Slide 7.3.7

The answer is yes. Since a single perceptron unit is so limited, a network of these units will be less limited. In fact, the introduction of "hidden" (not connected directly to the output) units into these networks make them much more powerful: they are no longer limited to linearly separable problems.

What these networks do is basically use the earlier layers (closer to the input) to transform the problem into more tractable problems for the latter layers.

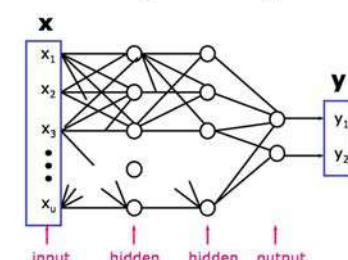
Beyond Linear Separability



6.034 - Spring 03 • 5



Multi-Layer Perceptron



- More powerful than single layer.
- Lower layers transform the input problem into more tractable (linearly separable) problems for subsequent layers.

6.034 - Spring 03 • 7



XOR Problem

The diagram illustrates the XOR problem as "Not linearly separable". It shows two parallel diagonal lines representing decision boundaries. The upper boundary is labeled o_1 and the lower boundary is labeled o_2 . The region between them contains training points (1,0), (0,1), and (1,1). The region above o_1 contains (0,0) and (1,0). The region below o_2 contains (0,0) and (0,1).

Weights and Offsets:

- For o_1 : $w_{01} = 3/2$, $w_{11} = w_{12} = 1$, $w_{21} = w_{22} = 1$, offset = 1
- For o_2 : $w_{02} = 1/2$, $w_{21} = w_{22} = 1$, offset = -1

x_1	x_2	o_1	o_2	y
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	1	1	0

6.034 - Spring 03 • 8

Slide 7.3.8

To see how having hidden units can help, let us see how a two-layer perceptron network can solve the XOR problem that a single unit failed to solve.

We see that each hidden unit defines its own "decision boundary" and the output from each of these units is fed to the output unit, which returns a solution to the whole problem. Let's look in detail at each of these boundaries and its effect.

Note that each of the weights in the first layer, except for the offsets, has been set to 1. So, we know that the decision boundaries are going to have normal vectors equal to [1 1], that is, pointing up and to the right, as shown in the diagram. The values of the offsets show that the hidden unit labeled o_1 has a larger offset (that is, distance from the origin) and the hidden unit labeled o_2 has a smaller offset. The actual distances from the line to the origin are obtained by dividing the offsets by $\sqrt{2}$, the magnitude of the normal vectors.

If we focus on the first decision boundary we see only one of the training points (the one with feature values (1,1)) is in the half space that the normal points into. This is the only point with a positive distance and thus a one output from the perceptron unit. The other points have negative distance and produce a zero output. This is shown in the shaded column in the table.

Slide 7.3.9

Looking at the second decision boundary we see that three of the training points (except for the one with feature values (0,0)) are in the half space that the normal points into. These points have a positive distance and thus a one output from the perceptron unit. The other point has negative distance and produces a zero output. This is shown in the shaded column in the table.

XOR Problem

The diagram illustrates the XOR problem as "Not linearly separable". It shows two parallel diagonal lines representing decision boundaries. The upper boundary is labeled o_1 and the lower boundary is labeled o_2 . The region between them contains training points (1,0), (0,1), and (1,1). The region above o_1 contains (0,0) and (1,0). The region below o_2 contains (0,0) and (0,1).

Weights and Offsets:

- For o_1 : $w_{01} = 3/2$, $w_{11} = w_{12} = 1$, offset = 1
- For o_2 : $w_{02} = 1/2$, $w_{21} = w_{22} = 1$, offset = -1

x_1	x_2	o_1	o_2	y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	1	1	0

6.034 - Spring 03 • 9

Slide 7.3.10

On the lower right, we see that the problem has been mapped into a linearly separable problem in the space of the outputs of the hidden units. We can now easily find a linear separator, for example, the one shown here. This mapping is where the power of the multi-layer perceptron comes from.

XOR Problem

The diagram illustrates the XOR problem as "Not linearly separable". It shows two parallel diagonal lines representing decision boundaries. The upper boundary is labeled o_1 and the lower boundary is labeled o_2 . The region between them contains training points (1,0), (0,1), and (1,1). The region above o_1 contains (0,0) and (1,0). The region below o_2 contains (0,0) and (0,1).

Weights and Offsets:

- For o_1 : $w_{01} = 3/2$, $w_{11} = w_{12} = 1$, offset = 1
- For o_2 : $w_{02} = 1/2$, $w_{21} = w_{22} = 1$, offset = -1
- For o_3 : $w_{03} = 1/2$, $w_{31} = -1$, $w_{32} = 1$, offset = 0

x_1	x_2	o_1	o_2	o_3	y
0	0	0	0	0	0
0	1	0	1	1	1
1	0	1	0	1	1
1	1	1	1	0	0

6.034 - Spring 03 • 10

Slide 7.3.11

It turns out that a three-layer perceptron (with sufficiently many units) can separate any data set. In fact, even a two-layer perceptron (with lots of units) can separate almost any data set that one would see in practice.

However, the presence of the discontinuous threshold in the operation means that there is no simple local search for a good set of weights; one is forced into trying possibilities in a combinatorial way.

The limitations of the single-layer perceptron and the lack of a good learning algorithm for multi-layer perceptrons essentially killed the field of statistical machine learning for quite a few years. The stake through the heart was a slim book entitled "Perceptrons" by Marvin Minsky and Seymour Papert of MIT.

Multi-Layer Perceptron Learning

- Any set of training points can be separated by a three-layer perceptron network.
- "Almost any" set of points separable by two-layer perceptron network.
- But, no efficient learning rule is known.

Multi-Layer Perceptron Learning

- Any set of training points can be separated by a three-layer perceptron network.
- "Almost any" set of points separable by two-layer perceptron network.
- But, no efficient learning rule is known.
- Could we use gradient ascent/descent?
- We would need smoothness: small change in weights produces small change in output.
- Threshold function is not smooth.

Slide 7.3.12

A natural question to ask is whether we could use gradient descent to train a multi-layer perceptron. The answer is that we can't as long as the output is discontinuous with respect to changes in the inputs and the weights. In a perceptron unit it doesn't matter how far a point is from the decision boundary, you will still get a 0 or a 1. We need a smooth output (as a function of changes in the network weights) if we're to do gradient descent.

6.034 - Spring 03 • 12



Slide 7.3.13

Eventually people realized that if one "softened" the thresholds, one could get information as to whether a change in the weights was helping or hurting and define a local improvement procedure that way.

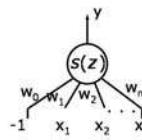
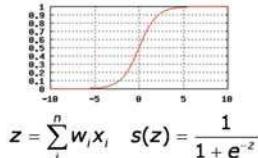
Multi-Layer Perceptron Learning

- Any set of training points can be separated by a three-layer perceptron network.
- "Almost any" set of points separable by two-layer perceptron network.
- But, no efficient learning rule is known.
- Could we use gradient ascent/descent?
- We would need smoothness: small change in weights produces small change in output.
- Threshold function is not smooth.
- Use a smooth threshold function!

6.034 - Spring 03 • 13



Sigmoid Unit



6.034 - Spring 03 • 14

Slide 7.3.14

The classic "soft threshold" that is used in neural nets is referred to as a "sigmoid" (meaning S-like) and is shown here. The variable z is the "total input" or "activation" of a neuron, that is, the weighted sum of all of its inputs.

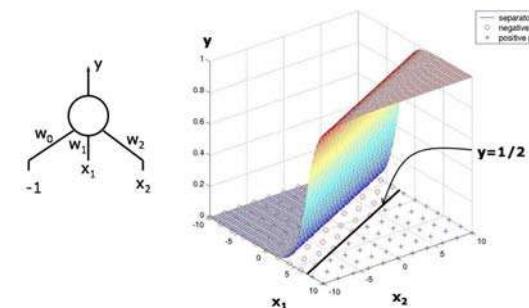
Note that when the input (z) is 0, the sigmoid's value is 1/2. The sigmoid is applied to the weighted inputs (including the threshold value as before). There are actually many different types of sigmoids that can be (and are) used in neural networks. The sigmoid shown here is actually called the logistic function.

Slide 7.3.15

We can think of a sigmoid unit as a "soft" perceptron. The line where the perceptron switches from a 0 output to a 1, is now the line along which the output of the sigmoid unit is 1/2. On one side of this line, the output tends to 0, on the other it tends to 1.

So, this "logistic perceptron" is still a linear separator in the input space. In fact, there's a well known technique in statistics, called **logistic regression** which uses this type of model to fit the probabilities of boolean-valued outputs, which are not properly handled by a linear regression. Note that since the output of the logistic function is between 0 and 1, the output can be interpreted as a probability.

Sigmoid Unit

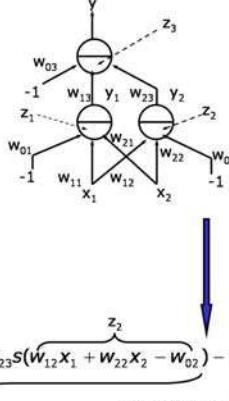


6.034 - Spring 03 • 15



Training

$y(\mathbf{x}, \mathbf{w})$
 \mathbf{w} is a vector of weights
 \mathbf{x} is a vector of inputs



$$y = s(w_{13}s(w_{11}x_1 + w_{12}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03})$$

6.034 - Spring 03 • 16

Slide 7.3.16

The key property of the sigmoid is that it is differentiable. This means that we can use gradient-based methods of minimization for training. Let's see what that means.

The output of a multi-layer net of sigmoid units is a function of two vectors, the inputs (\mathbf{x}) and the weights (\mathbf{w}). An example of what that function looks like for a simple net is shown along the bottom, where $s()$ is whatever output function we are using, for example, the logistic function we saw in the last slide.

The output of this function (y) varies smoothly with changes in the input and, importantly, with changes in the weights. In fact, the weights and inputs both play similar roles in the function.

Slide 7.3.17

Given a dataset of training points, each of which specifies the net inputs and the desired outputs, we can write an expression for the **training error**, usually defined as the sum of the squared differences between the actual output (given the weights) and the desired output. The goal of training is to find a weight vector that minimizes the training error.

We could also use the mean squared error (MSE), which simply divides the sum of the squared errors by the number of training points instead of just 2. Since the number of training points is a constant, the value of the minimum is not affected.

Gradient Descent

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

$$\nabla_{\mathbf{w}} E = \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i) \nabla_{\mathbf{w}} y(\mathbf{x}^i, \mathbf{w})$$

$$\nabla_{\mathbf{w}} y = \left[\frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} E$$

Error on training set
Gradient of Error
Gradient Descent

6.034 - Spring 03 • 18

Slide 7.3.18

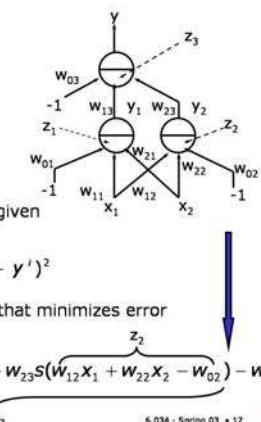
We've seen that the simplest method for minimizing a differentiable function is **gradient descent** (or ascent if you're maximizing).

Recall that we are trying to find the weights that lead to a minimum value of training error. Here we see the gradient of the training error as a function of the weights. The descent rule is basically to change the weights by taking a small step (determined by the **learning rate** η) in the direction opposite this gradient.

Note that the gradient of the error is simply the sum over all the training points of the error in the prediction for that point (given the current weights), which is the network output y minus the desired output y^i , times the gradient of the network output for that input and weight combination.

Training

$y(\mathbf{x}, \mathbf{w})$
 \mathbf{w} is a vector of weights
 \mathbf{x} is a vector of inputs
 y^i is desired output:



Error over the training set for a given weight vector:
 $E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$

Our goal is to find weight vector that minimizes error

$$y = s(w_{13}s(w_{11}x_1 + w_{12}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03})$$

6.034 - Spring 03 • 17

Slide 7.3.19

Let's look at a single sigmoid unit and see what the gradient descent rule would be in detail. We'll use the on-line version of gradient descent, that is, we will find the weight change to reduce the training error on a single training point. Thus, we will be neglecting the sum over the training points in the real gradient.

As we saw in the last slide, we will need the gradient of the unit's output with respect to the weights, that is, the vector of changes in the output due to a change in each of the weights.

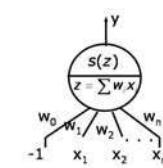
The output (y) of a single sigmoid unit is simply the output of the sigmoid function for the current activation (that is, total weighted input) of the unit. So, this output depends both on the values of the input features and the current values of the weights.

The gradient of this output function with respect to any of the weights can be found by an application of the chain rule of differentiation. The derivative of y with respect to w can be written as the product of the derivative with respect to z (the total activation) times the derivative of z with respect to the weight. The first term is the slope of the sigmoid function for the given input and weights, which we can write as $ds(z)/dz$. In this simple situation the total activation is a linear function of the weights, each with a coefficient corresponding to a feature value, x_i , for weight w_i .

Gradient Descent Single Unit

$$\nabla_{\mathbf{w}} y = \left[\frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$z = \sum_i w_i x_i, \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\begin{aligned} \frac{\partial y}{\partial w_i} &= \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i} \\ &= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_i} \\ &= \frac{\partial s(z)}{\partial z} x_i \end{aligned}$$


6.034 - Spring 03 • 19

So, the derivative of the activation with respect to the weight is just the input feature value, x_i .

Gradient Descent Single Unit

$\nabla_w Y = \left[\frac{\partial Y}{\partial w_1}, \dots, \frac{\partial Y}{\partial w_n} \right]$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} x_i$$

$$w_i \leftarrow w_i - \eta (y - y^m) \frac{\partial s(z)}{\partial z} x_i$$

$$\delta \cong \frac{\partial E}{\partial z} = (y - y^m) \frac{\partial s(z)}{\partial z}$$

$$\Delta w_i = -\eta \delta x_i \quad \text{Delta Rule}$$

6.034 - Spring 03 • 20

Slide 7.3.20

Now, we can substitute this result into the expression for the gradient descent rule we found before (for a single point).

We will define a new quantity called delta, which is defined to be the derivative of the error with respect to a change in the activation z . We can think of this value as the "sensitivity" of the network output to a change in the activation of a unit.

The important result we get is that the change in the i^{th} weight is proportional to delta times the i^{th} input. This innocent looking equation has more names than you can shake a stick at: the delta rule, the LMS rule, the Widrow-Hoff rule, etc. Or you can simply call it the chain rule applied to the squared training error.

Slide 7.3.21

The derivative of the sigmoid plays a prominent role in these gradients, not surprisingly. Here we see that this derivative has a very simple form when expressed in terms of the **output** of the sigmoid. Then, it is just the output times 1 minus the output. We will use this fact liberally later.

Derivative of the sigmoid

$$s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{ds(z)}{dz} = \frac{d}{dz} [(1 + e^{-z})^{-1}]$$

$$= [-(1 + e^{-z})^{-2}] [-e^{-z}]$$

$$= \left[\frac{1}{1 + e^{-z}} \right] \left[\frac{e^{-z}}{1 + e^{-z}} \right]$$

$$= s(z)(1 - s(z))$$

6.034 - Spring 03 • 21

Gradient of Unit Output

$\nabla_w Y = \left[\frac{\partial Y}{\partial w_1}, \dots, \frac{\partial Y}{\partial w_n} \right]$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

6.034 - Spring 03 • 22

Slide 7.3.22

Now, what happens if the input to our unit is not a direct input but the output of another unit and we're interested in the rate of change in y in response to a change to one of the weights in this second unit?

Gradient of Unit Output

$$\nabla_w Y = \left[\frac{\partial Y}{\partial w_1}, \dots, \frac{\partial Y}{\partial w_n} \right]$$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial w_j} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_j}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_j}$$

$$= \frac{\partial s(z)}{\partial z} w_j \frac{\partial y_j}{\partial w_j}$$

6.034 - Spring 03 • 23

Gradient of Unit Output

$\nabla_w Y = \left[\frac{\partial Y}{\partial w_1}, \dots, \frac{\partial Y}{\partial w_n} \right]$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$\frac{\partial Y}{\partial w_i} = \frac{\partial Y}{\partial z} \frac{\partial z}{\partial w_i}$
 $= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_i}$
 $= \frac{\partial s(z)}{\partial z} x_i$

$\frac{\partial Y}{\partial w_j} = \frac{\partial Y}{\partial z} \frac{\partial z}{\partial w_j}$
 $= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_j}$
 $= \frac{\partial s(z)}{\partial z} w_j \frac{\partial Y}{\partial w_j}$

Base Case Recursion

6.034 - Spring 03 • 24

Slide 7.3.24

We've just set up a recursive computation for the dy/dw terms. Note that these terms will be products of the slopes of the output sigmoid for the units times the weight on the input times a term of similar form for units below the input, until we get to the input with the weight we are differentiating with respect to. In the base case, we simply have the input value on that line, which could be one of the x_i or one of the y_i , since clearly the derivative of any unit with respect to w_i "below" the line with that weight will be zero.

Slide 7.3.25

Let's see how this works out for the simple case we've looked at before. There are two types of weights, the ones on the output unit, of the form, w_{*3} . And the weights on the two lower level units, w_{*1} and w_{*2} . The form of dy/dw for each of these two weights will be different as we saw in the last slide.

Gradient of Error

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

$$y = s(w_{13}s(\underbrace{w_{11}x_1 + w_{21}x_2 - w_{01}}_{z_1}) + w_{23}s(\underbrace{w_{12}x_1 + w_{22}x_2 - w_{02}}_{z_2}) - w_{03})$$

6.034 - Spring 03 • 25

Slide 7.3.26

Recall that in the derivative of the error (**for a single instance**) with respect to any of the weights, we get a term that measures the error at the output ($y - y^i$) times the change in the output which is produced by the change in the weight (dy/dw).

Gradient of Error

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

$$y = s(w_{13}s(\underbrace{w_{11}x_1 + w_{21}x_2 - w_{01}}_{z_1}) + w_{23}s(\underbrace{w_{12}x_1 + w_{22}x_2 - w_{02}}_{z_2}) - w_{03})$$

$$\frac{\partial E}{\partial w_j} = (y - y^i) \frac{\partial y}{\partial w_j}$$

6.034 - Spring 03 • 26

Slide 7.3.27

Let's pick weight w_{13} , that weights the output of unit 1 (y_1) coming into the output unit (unit 3). What is the change in the output y_3 as a result of a small change in w_{13} ? Intuitively, we should expect it to depend on the value of y_1 , the "signal" on that wire since the change in the total activation when we change w_{13} is scaled by the value of y_1 . If y_1 were 0 then changing the weight would have no effect on the unit's output.

Changing the weight changes the activation, which changes the output. Therefore, the impact of the weight change on the output depends on the slope of the output (the sigmoid output) with respect to the activation. If the slope is zero, for example, then changing the weight causes no change in the output.

When we evaluate the gradient (using the chain rule), we see exactly what we expect -- the product of the sigmoid slope (dy/dz_3) times the signal value y_1 .

Gradient of Error

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

$$y = s(w_{13}s(\underbrace{w_{11}x_1 + w_{21}x_2 - w_{01}}_{z_1}) + w_{23}s(\underbrace{w_{12}x_1 + w_{22}x_2 - w_{02}}_{z_2}) - w_{03})$$

$$\frac{\partial E}{\partial w_j} = (y - y^i) \frac{\partial y}{\partial w_j}$$

$$\frac{\partial y}{\partial w_{13}} = \frac{\partial y}{\partial z_3} \frac{\partial z_3}{\partial w_{13}} = \frac{\partial y}{\partial z_3} y_1$$

6.034 - Spring 03 • 27

Gradient of Error

$$E = \frac{1}{2} \sum_i (y(x^i, w) - y^i)^2$$

$$y = s(w_{13}s(w_{11}x_1 + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03})$$

$$\frac{\partial E}{\partial w_j} = (y - y^i) \frac{\partial y}{\partial w_j}$$

$$\frac{\partial y}{\partial w_{13}} = \frac{\partial y}{\partial z_3} \frac{\partial z_3}{\partial w_{13}} = \frac{\partial y}{\partial z_3} y_1$$

$$\frac{\partial y}{\partial w_{11}} = \frac{\partial y}{\partial z_3} \frac{\partial z_3}{\partial w_{11}} = \frac{\partial y}{\partial z_3} \left(w_{13} \frac{\partial y_1}{\partial z_1} \frac{\partial z_1}{\partial w_{11}} \right) = \frac{\partial y}{\partial z_3} \left(w_{13} \frac{\partial y_1}{\partial z_1} x_1 \right)$$

6.034 - Spring 03 • 28

Slide 7.3.28

What happens when we pick a weight that's deeper in the net, say w_{11} ? Since that weight affects y_1 , we expect that the change in the final output will be affected by the value of w_{13} and the slope of the sigmoid at unit 3 (as when we were changing w_{13}). In addition, the change in y_1 will depend on the value of the "signal" on the wire (x_1) and the slope of the sigmoid at unit 1. Which is precisely what we see.

Note that in computing the gradients deeper in the net we will use some of the gradient terms closer to the output. For example, the gradient for weights on the inputs to unit 1 change the output by changing one input to unit 3 and so the final gradient depends on the behavior of unit 3. It is the realization of this reuse of terms that leads to an efficient strategy for computing the error gradient.

Slide 7.3.29

The cases we have seen so far are not completely general in that there has been only one path through the network for the change in a weight to affect the output. It is easy to see that in more general networks there will be multiple such paths, such as shown here.

This means that a weight can affect more than one of the inputs to a unit, and so we need to add up all the effects before multiplying by the slope of the sigmoid.

Gradient of Unit Output

$$\frac{\partial y}{\partial w_{11}} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_{11}}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_{11}}$$

$$= \frac{\partial s(z)}{\partial z} \left(w_{42} \frac{\partial y_2}{\partial w_{11}} + w_{43} \frac{\partial y_3}{\partial w_{11}} \right)$$

Recursion (more general)

A change in w_{11} affects the error via a change in y_1 , which affects y_2 and y_3

6.034 - Spring 03 • 29

Slide 7.3.30

In general we will be looking at networks connected in the fashion shown on the left, where the output of every unit at one level is connected to an input of every unit at the next level. We have not shown the bias inputs for each of the units, but they are there!

A word on notation. To avoid having to spell out the names of all the weights and signals in these networks, we will give each unit an index. The output of unit k is y_k . We will specify the weights on the inputs of unit k as $w_{i \rightarrow k}$ where i is either the index of one of the inputs or another unit. Because of the "feedforward" connectivity we have adopted this terminology is unambiguous.

Generalized Delta Rule

6.034 - Spring 03 • 30

Slide 7.3.31

In this type of network we can define a generalization of the delta rule that we saw for a single unit. We still want to define the sensitivity of the training error (for an input point) to a change in the total activation of a unit. This is a quantity associated with the unit, independent of any weight. We can express the desired change in a weight that feeds into unit k as (negative of) the product of the learning rate, delta, for unit k and the value of the input associated with that weight.

The tricky part is the definition of delta. From our investigation into the form of dy/dw , the form of delta in the pink box should be plausible: the product of the slope of the output sigmoid times the sum of the products of weights and other deltas. This is exactly the form of the dy/dw expressions we saw before.

The clever part here is that by computing the deltas starting with that of the output unit and moving **backward** through the network we can compute all the deltas for every unit in the network in one pass (once we've computed all the y 's and z 's during a forward pass). It is this property that has led to the name of this algorithm, namely **backpropagation**.

It is important to remember that this is still the chain rule being applied to computing the gradient of the error. However, the computations have been arranged in a clever way to make computing the

Generalized Delta Rule

$$\delta_j = \frac{\partial E}{\partial z_j}$$

$$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$$

$$\Delta w_{i \rightarrow j} = -\eta \delta_j y_i$$

$$\delta_4 = \frac{ds(z_4)}{dz_4} (\delta_5 w_{4 \rightarrow 5} + \delta_6 w_{4 \rightarrow 6})$$

$$\Delta w_{1 \rightarrow 4} = -\eta \delta_4 y_1$$

$$\Delta w_{2 \rightarrow 4} = -\eta \delta_4 y_2$$

6.034 - Spring 03 • 31

gradient efficient.

Backpropagation

An efficient method of implementing gradient descent for neural networks

$\mathbf{w}_{i \rightarrow j} = \mathbf{w}_{i \rightarrow j} - \eta \delta_j y_i$ Descent rule
 $\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$ Backprop rule

6.034 - Spring 03 • 32

Slide 7.3.32

Thus, the algorithm for computing the gradients we need to update the weights of a net, with maximal re-using of intermediate results is known as backpropagation.

The two simple formulas we need are the ones we have just seen. One tells us how to change a weight. This is a simple gradient descent formula, except that it says that the gradient of the error is of the form δ_j times y_i where y_i is the signal on the wire with this weight, so it is either one of the inputs to the net or an output of some unit.

The delta of one unit is defined to be the slope of the sigmoid of that unit (for the current value of z , the weighted input) times the weighted sum of the deltas for the units that this unit feeds into.

Slide 7.3.33

The backprop algorithm starts off by assigning random, small values to all the weights. The reason we want to have small weights is that we want to be near the approximately linear part of the sigmoid function, which happens for activations near zero. We want to make sure that (at least initially) none of the units are saturated, that is, are stuck at 0 or 1 because the magnitude of the total input is too large (positive or negative). If we get saturation, the slope of the sigmoid is 0 and there will not be any meaningful information of which way to change the weight.

Backpropagation

An efficient method of implementing gradient descent for neural networks

$\mathbf{w}_{i \rightarrow j} = \mathbf{w}_{i \rightarrow j} - \eta \delta_j y_i$ Descent rule
 $\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$ Backprop rule

1. Initialize weights to small random values

6.034 - Spring 03 • 33

Slide 7.3.34

Now we pick a sample input feature vector. We will use this to define the gradients and therefore the weight updates. Note that by updating the weights based on one input, we are introducing some randomness into the gradient descent. Formally, gradient descent on an error function defined as the sum of the errors over all the input instances should be the sum of the gradients over all the instances. However, backprop is typically implemented as shown here, making the weight change based on each feature vector. We will have more to say on this later.

Backpropagation

An efficient method of implementing gradient descent for neural networks

$\mathbf{w}_{i \rightarrow j} = \mathbf{w}_{i \rightarrow j} - \eta \delta_j y_i$ Descent rule
 $\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$ Backprop rule

1. Initialize weights to small random values
2. Choose a random sample input feature vector

6.034 - Spring 03 • 34

Slide 7.3.35

Now that we have weights and inputs, we can do a **forward propagation**, that is, we can compute the values of all the z 's and y 's, that is, the weighted inputs and the outputs for all the units. We will need these values, so let's remember them for each unit.

Backpropagation

An efficient method of implementing gradient descent for neural networks

$\mathbf{w}_{i \rightarrow j} = \mathbf{w}_{i \rightarrow j} - \eta \delta_j y_i$ Descent rule
 $\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$ Backprop rule

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (z_j) and output (y_j) for each unit (forward prop)

6.034 - Spring 03 • 35

Backpropagation

An efficient method of implementing gradient descent for neural networks

$$\begin{aligned} W_{i \rightarrow j} &= W_{i \rightarrow j} - \eta \delta_j Y_i && \text{Descent rule} \\ \delta_j &= \frac{ds(z_j)}{dz_j} \sum_k \delta_k W_{j \rightarrow k} && \text{Backprop rule} \end{aligned}$$

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (z_j) and output (y_j) for each unit (forward prop)
4. Compute δ_n for output layer

$$\delta_n = \frac{ds(z_n)}{dz_n} (y_n - y_n^m) = y_n(1 - y_n)(y_n - y_n^m)$$

6.034 - Spring 03 • 36

Slide 7.3.36

Now, we start the process of computing the deltas. First we do it for the output units, using the formula shown here, that is, the product of the gradient of the sigmoid at the output unit times the error for that unit.

Backpropagation

An efficient method of implementing gradient descent for neural networks

$$\begin{aligned} W_{i \rightarrow j} &= W_{i \rightarrow j} - \eta \delta_j Y_i && \text{Descent rule} \\ \delta_j &= \frac{ds(z_j)}{dz_j} \sum_k \delta_k W_{j \rightarrow k} && \text{Backprop rule} \end{aligned}$$

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (z_j) and output (y_j) for each unit (forward prop)
4. Compute δ_n for output layer

$$\delta_n = \frac{ds(z_n)}{dz_n} (y_n - y_n^m) = y_n(1 - y_n)(y_n - y_n^m)$$
5. Compute δ_j for all preceding layers by backprop rule

6.034 - Spring 03 • 37

Backpropagation

An efficient method of implementing gradient descent for neural networks

$$\begin{aligned} W_{i \rightarrow j} &= W_{i \rightarrow j} - \eta \delta_j Y_i && \text{Descent rule} \\ \delta_j &= \frac{ds(z_j)}{dz_j} \sum_k \delta_k W_{j \rightarrow k} && \text{Backprop rule} \end{aligned}$$

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (z_j) and output (y_j) for each unit (forward prop)
4. Compute δ_n for output layer

$$\delta_n = \frac{ds(z_n)}{dz_n} (y_n - y_n^m) = y_n(1 - y_n)(y_n - y_n^m)$$
5. Compute δ_j for all preceding layers by backprop rule
6. Compute weight change by descent rule (repeat for all weights)

6.034 - Spring 03 • 38

Slide 7.3.38

With the deltas and the unit outputs in hand, we can update the weights using the descent rule.

Backpropagation Example

First do forward propagation:
Compute z_i and y_i given x_i , w_{ij}

$$\begin{aligned} \delta_3 &= y_3(1 - y_3)(y_3 - y_3^m) \\ \delta_2 &= y_2(1 - y_2)\delta_3 w_{23} \\ \delta_1 &= y_1(1 - y_1)\delta_3 w_{13} \end{aligned}$$

$$\begin{aligned} w_{03} &= w_{03} - r\delta_3(-1) & \rightarrow w_{13} &= w_{13} - \eta \delta_3 y_1 & w_{23} &= w_{23} - \eta \delta_3 y_2 \\ w_{02} &= w_{02} - r\delta_2(-1) & \rightarrow w_{12} &= w_{12} - \eta \delta_2 x_1 & w_{22} &= w_{22} - \eta \delta_2 x_2 \\ w_{01} &= w_{01} - r\delta_1(-1) & \rightarrow w_{11} &= w_{11} - \eta \delta_1 x_1 & w_{21} &= w_{21} - \eta \delta_1 x_2 \end{aligned}$$

Compare to the direct derivation earlier

Note that all computations are local!

6.034 - Spring 03 • 39

6.034 Notes: Section 7.4

Slide 7.4.1

Now that we have looked at the basic mathematical techniques for minimizing the training error of a neural net, we should step back and look at the whole approach to training a neural net, keeping in mind the potential problem of overfitting.

We need to worry about overfitting because of the generality of neural nets and the proliferation of parameters associated even with a relatively simple net. It is easy to construct a net that has more parameters than there are data points. Such nets, if trained so as to minimize the training error without any additional constraints, can very easily overfit the training data and generalize very poorly. Here we look at a methodology that attempts to minimize that danger.

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights.
Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

6.034 - Spring 03 • 1 

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights.
Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance

6.034 - Spring 03 • 2 

Slide 7.4.2

The first step (in the ideal case) is to separate the data into three sets. A training set for choosing the weights (using backpropagation), a validation set for deciding when to stop the training and, if possible, a separate set for evaluating the final results.

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights.
Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance
2. Pick random, small weights as initial values

6.034 - Spring 03 • 3 

Slide 7.4.3

Then we pick a set of random small weights as the initial values of the weights. As we explained earlier, this reduces the chance that we will saturate any of the units initially.

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.

6.034 - Spring 03 • 4

Slide 7.4.4

Then we perform the minimization of the training error, for example, using backpropagation. This will generally involve going through the input data and making changes to the weights many times. A common term used in this context is the **epoch**, which indicates how many times the algorithm has gone through every point in the training data. So, for example, one can plot the training error as function of the training epoch. We will see this later.

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).

6.034 - Spring 03 • 5

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Repeat training (from step 2) several times (avoid local minima)

6.034 - Spring 03 • 6

Slide 7.4.6

In neural nets we do not have the luxury that we had in SVMs of knowing that we have found the global optimum after we finished learning. In neural nets, there are many local optima and backprop (or any other minimization strategy) can only guarantee finding a local optimum (and even this guarantee depends on careful choice of learning rate). So, it is often useful to repeat the training several times to see if a better result can be found. However, even a single round of training can be very expensive so this may not be feasible.

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Repeat training (from step 2) several times (avoid local minima)
6. Use best weights to compute error on test set, which is estimate of performance on new data. Do not repeat training to improve this.

6.034 - Spring 03 • 7

Slide 7.4.7

Once we have a final set of weights, we can use them once on a held out test set to estimate the expected behavior on new data. Note the emphasis on doing this once. If we change the weights to improve this behavior, then we no longer have a held out set.

Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with m weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
 - Training set – used for picking weights
 - Validation set – used to stop training
 - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Repeat training (from step 2) several times (avoid local minima)
6. Use best weights to compute error on test set, which is estimate of performance on new data. Do not repeat training to improve this.

Can use cross-validation if data set is too small to divide into three subsets.

6.034 - Spring 03 • 8

Slide 7.4.8

In many cases, one doesn't have the luxury of having these separate sets, due to scarcity of data, in which case cross-validation may be used as a substitute.

Slide 7.4.9

Let's look at the termination/overfitting issue via some examples.

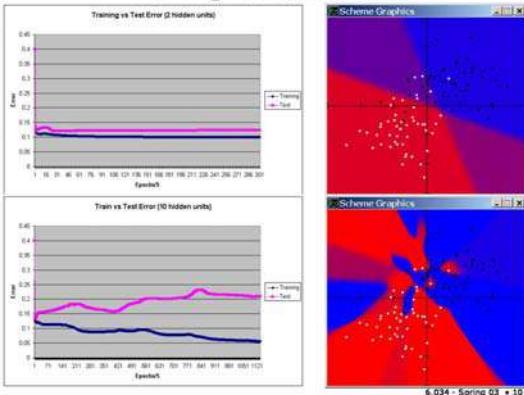
Here we see the behavior of a small neural net (two inputs, two hidden units and one output) when trained on the data shown in the picture on the right. The white and black points represent 50 instances from each of two classes (drawn from Gaussian distributions with different centers). An additional 25 instances each (drawn from the same distributions) have been reserved as a test set.

As you can see, the point distributions overlap and therefore the net cannot fully separate the data. The red region represents the area where the net's output is close to 1 and the blue region represents where the output is close to 0. Intermediate colors represent intermediate values.

The error on the training set drops quickly at the beginning and does not drop much after that. The error on the test set behaves very similarly except that it is a bit bigger than the error on the training set. This is to be expected since the detailed placement of points near the boundaries will be different in the test set.

The behavior we see here is a good one; we have not overfit the data.

Training vs. Test Error



Slide 7.4.10

Here we see the behavior of a larger neural net (with 10 hidden units) on the same data.

You can see that the training error continues to drop over a much longer set of training epochs. In fact, the error goes up slightly sometimes, then drops, then stagnates and drops again. This is typical behavior for backprop.

Note, however, that during most of the time that the training error is dropping, the test error is **increasing**. This indicates that the net is overfitting the data. If you look at the net output at the end of training, you can see what is happening. The net has constructed a baroque decision boundary to capture precisely the placement of the different instances in the training set. However, the instances in the test set are (very) unlikely to fall into that particular random arrangement.

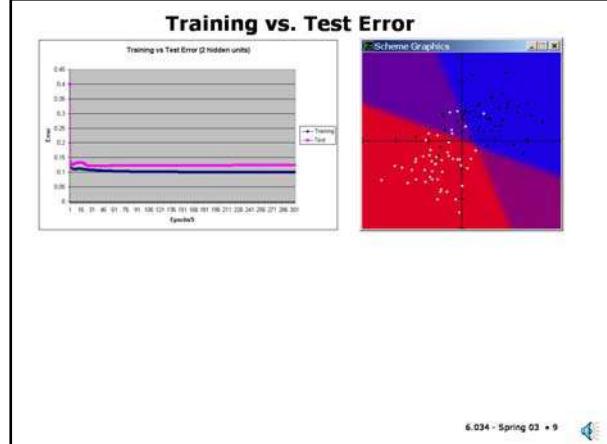
So, in fact, all that extra work in fitting the training set is wasted. Note that the test error with this net is much higher than with the simpler net. If we had used a validation set, we could have stopped training before it went too far astray.

Slide 7.4.11

Note that this type of overfitting is not unique to neural nets. In this slide you can see the behavior of 1-nearest-neighbor and decision trees on the same data. Both fit it perfectly and produce classifiers that are just as unlikely to generalize to new data.

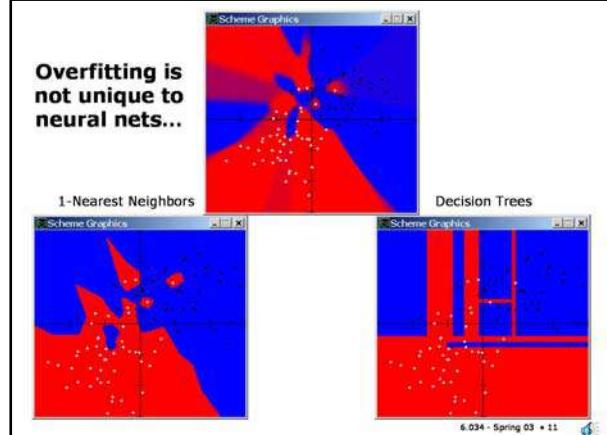
For K-nearest-neighbors, on this type of data one would want to use a value of K greater than 1. For decision trees one would want to prune back the tree somewhat. These decisions could be based on the performance on a held out validation set.

Similarly, for neural nets, one would want to choose the number of units and the stopping point based on performance on validation data.

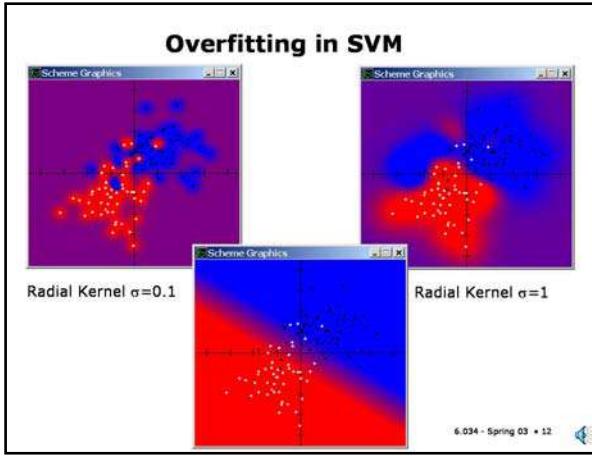


6.034 - Spring 03 • 9

Overfitting is not unique to neural nets...



6.034 - Spring 03 • 11

**Slide 7.4.12**

Even SVMs, which have relatively few parameters and a well-deserved reputation for being resistant to overfitting can overfit. In the center panel we see the output for a linear SVM. This is, in fact, the optimal type of separator for this data. On the upper left we see a fairly severe overfitting stemming from the choice of a too-small sigma for a radial kernel. On the right is the result from a larger choice of sigma (and a relatively high C). Where the points are densest, this actually approximates the linear boundary but then deviates wildly in response to an outlier near (-2, 0). This illustrates how the choice of kernel can affect the generalization ability of an SVM classifier.

Slide 7.4.13

We mentioned earlier that backpropagation is an on-line training algorithm, meaning that it changes the weights for each input instance. Although this is not a "correct" implementation of gradient descent for the total training error, it is often argued that the randomization effect is beneficial as it tends to push the system out of some shallow local minima. In any case, the alternative "off-line" approach of actually adding up the gradients for all the instances and changing the weights based on that complete gradient is also used and has some advantages for smaller systems. For larger nets and larger datasets, the on-line approach has substantial performance advantages.

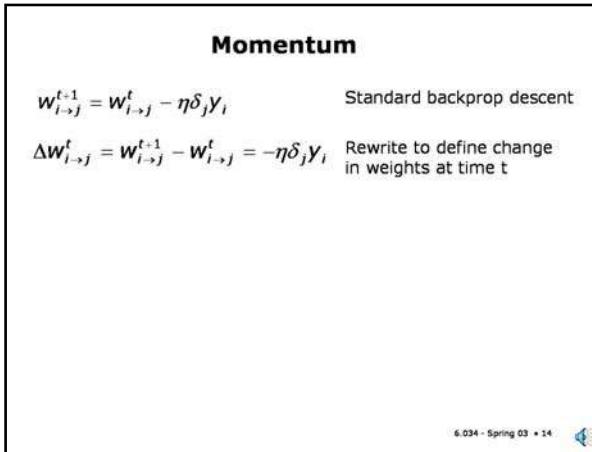
On-line vs off-line

There are two approaches to performing the error minimization:

- **On-line training** – present x^i and y^i (chosen randomly from the training set). Change the weights to reduce the error on this instance. Repeat.
- **Off-line training** – change weights to reduce the total error on training set (sum over all instances).

On-line training is an approximation to gradient descent since the gradient based on one instance is "noisy" relative to the full gradient (based on all instances). This can be beneficial in pushing the system out of shallow local minima.

6.034 - Spring 03 • 12

**Slide 7.4.14**

We have mentioned the difficulty of picking a learning rate for backprop that balances, on the one hand, the desire to move speedily towards a minimum by using a large learning rate and, on the other hand, the need to avoid overstepping the minimum and possibly getting into oscillations because of a too-large learning rate. One approach to balancing these is to effectively adjust the learning rate based on history. One of the original approaches for this is to use a **momentum** term in backprop.

Here is the standard backprop gradient descent rule, where the change to the weights is proportional to delta and y.

Momentum

$$w_{i \rightarrow j}^{t+1} = w_{i \rightarrow j}^t - \eta \delta_j y_i \quad \text{Standard backprop descent}$$

$$\Delta w_{i \rightarrow j}^t = w_{i \rightarrow j}^{t+1} - w_{i \rightarrow j}^t = -\eta \delta_j y_i \quad \text{Rewrite to define change in weights at time t}$$

$$\Delta w_{i \rightarrow j}^t = -\eta \delta_j y_i + \alpha \Delta w_{i \rightarrow j}^{t-1} \quad \text{Adding a momentum term, which adds a fraction of the weight change at the previous iteration.}$$

- Momentum can gradually increase step size when gradient is unchanged.
- Can help step through shallow local minima
- One more parameter to twiddle... not used much anymore

6.034 - Spring 03 • 15

Slide 7.4.15

We can keep around the most recent change to the weights (at time t-1) and add some fraction of that weight change to the current delta. The fraction, alpha, is the momentum weight.

The basic idea is that if the changes to the weights have had a consistent sign, then the effect is to have a larger step size in the weight. If the sign has been changing, then the net change may be smaller.

Note that even if the delta times y term is zero (denoting a local minimum in the error), in the presence of a momentum term, the change in the weights will not necessarily be zero. So, this may cause the system to move through a shallow minimum, which may be good. However, it may also lead to undesirable oscillations in some circumstances.

In practice, choosing a good value of momentum for a problem can be nearly as hard as choosing a learning rate and it's one more parameter to twiddle with.

Momentum is not that popular a technique anymore; people will tend to use more complex search strategies to ensure convergence to a local minimum.

Input Representation

- All the signals in a neural net are [0, 1]. Input values should also be scaled to this range (or approximately so) so as to speed training.

Slide 7.4.16

One question that arises in neural nets (and many other machine learning approaches) is how to represent the input data. We have discussed some of these issues before.

One issue that is prominent in neural nets is the fact that the behavior of these nets are dependent on the scale of the input. In particular, one does not want to saturate the units, since at that point it becomes impossible to train them. Note that all "signals" in a sigmoid-unit neural net are in the range [0,1] because that is the output range of the sigmoids. It is best to keep the range of the inputs in that range as well. Simple normalization (subtract the mean and divide by the standard deviation) will almost do that and is adequate for most purposes.

6.034 - Spring 03 • 16

Slide 7.4.17

Another issue has to do with the representation of discrete data (also known as "categorical" data). You could think of representing these as either unary or binary numbers. Binary numbers are generally a bad choice; unary is much preferable if the number of values is small since it decouples the values completely.

Output Representation

- A neural net with a single sigmoid output unit is aimed at binary classification. Class is 0 if $y < 0.5$ and 1 otherwise.
- For multi-class problems
 - Can use one output per class (unary encoding)
 - There may be confusing outputs (two outputs > 0.5 in unary encoding).
 - More sophisticated method is to use special softmax units, which force outputs to sum to 1.

Slide 7.4.18

Similar questions arise at the output end. So far, we have focused on binary classification. There, the representation is clear - a single output and we treat an output of 0.5 as the dividing value between one class and the other.

For multi-class problems, one can have multiple output units, for example, each aimed at recognizing one class, sharing the hidden units with the other classes.

One difficulty with this approach is that there may be ambiguous outputs, e.g. two values above the 0.5 threshold when using a unary encoding. How do we treat such a case? One reasonable approach is to choose the class with the largest output.

A more sophisticated method is to introduce a new type of unit (called "softmax") that forces the sum of the unary outputs to add to 1. One can then interpret the network outputs as the probabilities that the input belongs to each of the classes.

6.034 - Spring 03 • 18

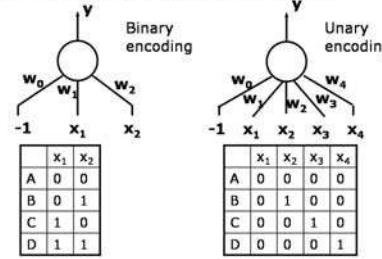
Slide 7.4.19

Another detail to consider in training is what to use as the desired (target) value for the network outputs. We have been talking as if we would use 0 or 1 as the targets. The problem with this is that those are the asymptotic values of the sigmoid only achieved for infinite values of the weights. So, if we were to attempt to train a network until the weights stop changing, then we'd be in trouble. It is common to use values such as 0.1 and 0.9 instead of 0 and 1 during neural network training.

In practice, however, the usual termination for training a network is when the training or, preferably, validation error either achieves an acceptable level, reaches a minimum or stops changing significantly. These outcomes generally happen long before we run the risk of the weights becoming infinite.

Input Representation

- All the signals in a neural net are [0, 1]. Input values should also be scaled to this range (or approximately so) so as to speed training.
- If the input values are discrete, e.g. {A, B, C, D} or {1, 2, 3, 4}, they need to be coded in unary form.



6.034 - Spring 03 • 19

Target Value

- During training it is impossible for the outputs to reach 0 or 1 (with finite weights).
- Customary to use 0.1 and 0.9 as targets
- But, most termination criteria, e.g. small change in training or validation error will stop training before targets are reached.

6.034 - Spring 03 • 19

Regression

- A sigmoid output unit is not suitable for regression, since sigmoids are designed to change quickly from 0 to 1.
- For regression, we want a linear output unit, that is, remove the output non-linearity.
- The rest of the net still retains the sigmoid units.

Slide 7.4.20

Neural nets can also do regression, that is, produce an output which is a real number outside the range of 0 to 1, for example, predicting the age of death as a function of packs of cigarettes smoked. However, to do regression, it is important that one does not try to predict a continuous value using an output sigmoid unit. The sigmoid is basically a soft threshold with a limited dynamic range. When doing regression, we want the output unit to be linear, that is, simply remove the sigmoid non-linearity and have the unit returned a weighted sum of its inputs.

6.034 - Spring 03 • 20

Slide 7.4.21

One very interesting application of neural networks is the ALVINN project from CMU. The project was the brainchild of Dean Pomerleau. ALVINN is an automatic steering system for a car based on input from a camera mounted on the vehicle. This system was successfully demonstrated on a variety of real roads in actual traffic. A successor to ALVINN, which unfortunately was not based on

ALVINN steers on highways

http://www.ri.cmu.edu/projects/project_160.html

Images removed due to copyright restrictions.

Dean Pomerleau
CMU

6.034 - Spring 03 • 21

ALVINN steers on highways

http://www.ri.cmu.edu/projects/project_160.html

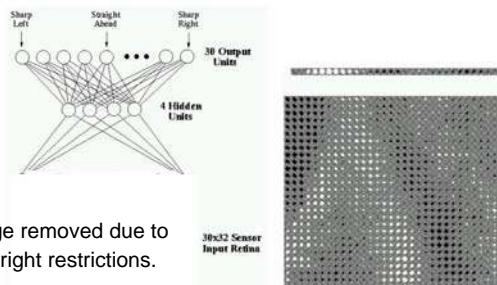


Image removed due to
copyright restrictions.

6.034 - Spring 03 • 22

Slide 7.4.22

The ALVINN neural network is shown here. It has 960 inputs (a 30x32 array derived from the pixels of an image), four hidden units and 30 output units (each representing a steering command). On the right you can see a pattern of the brightnesses of the input pixels and right above that you can see the pattern of the outputs, representing a "steer left" command.

Slide 7.4.23

One of the most interesting issues that came up in the ALVINN project was the problem of obtaining training data. It's not difficult to get images of somebody driving correctly down the middle of the road, but if that were **all** that ALVINN could do, then it would not be safe to let it on the road. What if an obstacle arose or there was a momentary glitch in the control system or a bump in the road got you off center? It is important that ALVINN be able to recover and steer the vehicle back to the center.

The researchers considered having the vehicle drive in a wobbly path during training, but that posed the danger of having the system learn to drive that way. They came up with a clever solution. Simulate what the sensory data would have looked like had ALVINN been off-center or headed in a slightly wrong direction and also, for each such input, simulate what the steering command should have been.

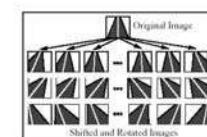
Now, you don't want to generate simulated images from scratch, as in a video game, since they are insufficiently realistic. What they did, instead, is transform the real images and fill in the few missing pixels by a form of "interpolation" on the actual pixels. The results were amazingly good.

However, it turned out that once one understood that this whole project was possible and one

ALVINN steers on highways

http://www.ri.cmu.edu/projects/project_160.html

Image removed due to
copyright restrictions.



- Problem: Getting enough diversity in training set
- Answer: Transform sensor image and steering direction

6.034 - Spring 03 • 23

understood what ALVINN was learning, it became possible to build a special purpose system that was faster and more reliable and involved no explicit on-line learning. This is not an uncommon side-effect of a machine-learning project.

Some observations...

- Although Neural Nets kicked off the current phase of interest in machine learning, they are extremely problematic...
 - Too many parameters (weights, learning rate, momentum, etc)
 - Hard to choose the architecture
 - Very slow to train
 - Easy to get stuck in local minima
- Interest has shifted to other methods, such as support vector machines, which can be viewed as variants of perceptrons (with a twist or two).

6.034 - Spring 03 • 24 

Slide 7.4.24

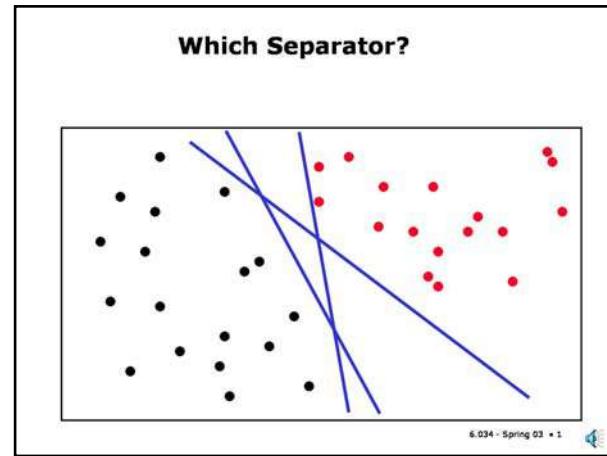
Neural nets are largely responsible for the current interest in statistical methods for machine learning. For a while, they were wildly popular in the research community and saw many applications. Over time, the enthusiasm has waned somewhat.

The big problem with neural nets is that they are very complex and have many many parameters that have to be chosen. Furthermore, training them is a bit of a nightmare. So, recent interest has shifted towards methods that are simpler to use and can be characterized better. For example, support vector machines, which at one level can be viewed as a variation of the same perceptrons that neural nets superseded, is the current darling of the machine learning research and application community.

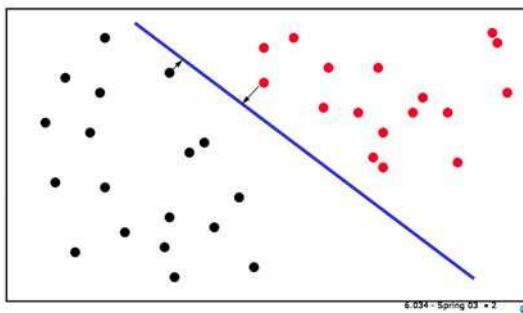
6.034 Notes: Section 8.1

Slide 8.1.1

There is no easy way to characterize which particular separator the perceptron algorithm will end up with. In general, there can be many separators for a data set. Even in the tightly constrained bankruptcy data set, we saw two runs of the algorithm with different starting points ended up with slightly different hypotheses. Is there any reason to prefer one separator over the others?

**Which Separator?**

Maximize the margin to closest points

**Slide 8.1.2**

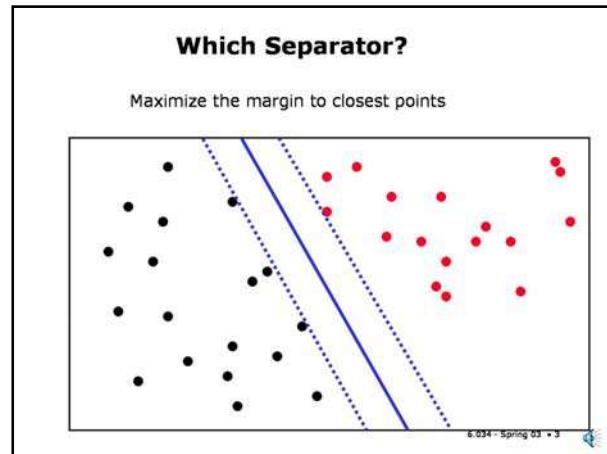
Yes. One natural choice is to pick the separator that has the maximal margin to its closest points on either side. This is the separator that seems most conservative. Any other separator will be "closer" to one class than to the other. The one shown in this figure, for example, seems like it's closer to the black points on the lower left than to the red ones.

Slide 8.1.3

This one seems safer, no?

Another way to motivate the choice of the maximal margin separator is to see that it reduces the "variance" of the hypothesis class. Recall that a hypothesis has large variance if small changes in the data result in a very different hypothesis. With a maximal margin separator, we can wiggle the data quite a bit without affecting the separator. Placing the separator very close to positive or negative points is a kind of overfitting; it makes your hypothesis very dependent on details of the input data.

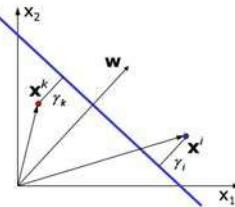
Let's see if we can figure out how to find the separator with maximal margin as suggested by this picture.



Margin of a point

$$\gamma^i \equiv y^i(\mathbf{w} \cdot \mathbf{x}^i + b)$$

- proportional to perpendicular distance of point \mathbf{x}^i to hyperplane



6.034 - Spring 03 • 4

Slide 8.1.4

First we have to define what we are trying to optimize. Clearly we want to use our old definition of margin, but we'll have to deal with a couple of issues first. Note that we're using the \mathbf{w}, b notation instead of $\bar{\mathbf{w}}$, because we will end up giving b special treatment in the future.

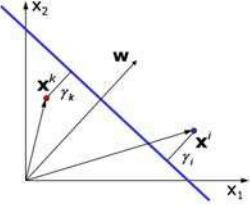
Slide 8.1.5

Remember that any scaling of \mathbf{w} and b defines the same line; but it will result in different values of gamma. To get the actual geometric distance from the point to the separator (called the *geometric margin*), we need to divide gamma through by the magnitude of \mathbf{w} .

Margin of a point

$$\gamma^i \equiv y^i(\mathbf{w} \cdot \mathbf{x}^i + b)$$

- proportional to perpendicular distance of point \mathbf{x}^i to hyperplane
- geometric margin is $\gamma^i / \|\mathbf{w}\|$



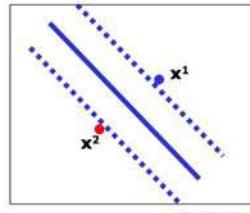
6.034 - Spring 03 • 5

Margin

$$\gamma' \equiv y^i(\mathbf{w} \cdot \mathbf{x}^i + b)$$

- Scaling \mathbf{w} changes value of margin but not actual distances to separator (geometric margin)
- Pick the margin to closest positive and negative points to be 1

$$\begin{aligned} +1(\mathbf{w} \cdot \mathbf{x}^1 + b) &= 1 \\ -1(\mathbf{w} \cdot \mathbf{x}^2 + b) &= 1 \end{aligned}$$



6.034 - Spring 03 • 6

Slide 8.1.6

The next issue is that we have defined the margin for a point relative to a separator but we don't want to just maximize the margin of some particular single point. We want to focus on one point on each side of the separator, each of which is closest to the separator. And we want to place the separator so that it is as far from these two points as possible. Then we will have the maximal margin between the two classes.

Since we have a degree of freedom in the magnitude of \mathbf{w} we're going to just define the margin for each of these points to be 1. (You can think of this 1 as having arbitrary units given by the magnitude of \mathbf{w} .)

You might be worried that we can't possibly know which will be the two closest points until we know what the separator is. It's a reasonable worry, and we'll sort it out in a couple of slides.

Slide 8.1.7

Having chosen these margins, we can add the two equations to get that the projection of the weight vector on the difference between the two chosen data points has magnitude 2. This is obvious from the setup, but it's nice to see it follows.

Then, we divide through by the magnitude of the weight vector and we have a simple expression for the margin, simply 2 over the magnitude of \mathbf{w} .

Margin

- Pick the margin to closest positive and negative points to be 1

$$\begin{aligned} +1(\mathbf{w} \cdot \mathbf{x}^1 + b) &= 1 \\ -1(\mathbf{w} \cdot \mathbf{x}^2 + b) &= 1 \end{aligned}$$

- Combining these

$$\mathbf{w} \cdot (\mathbf{x}^1 - \mathbf{x}^2) = 2$$

- Dividing by length of \mathbf{w} gives perpendicular distance between dashed lines ($2 \times$ geometric margin)

$$\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot (\mathbf{x}^1 - \mathbf{x}^2) = \frac{2}{\|\mathbf{w}\|}$$

6.034 - Spring 03 • 7

Picking w to Maximize Margin

- Pick w to maximize geometric margin

$$\frac{2}{\|w\|}$$

- or, equivalently, minimize

$$\|w\| = \sqrt{w \cdot w}$$

- or, equivalently, minimize

$$\frac{1}{2} \|w\|^2 = \frac{1}{2} w \cdot w = \frac{1}{2} \sum_j w_j^2$$

Slide 8.1.8

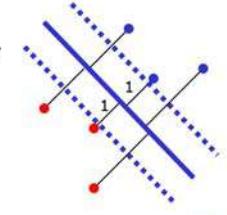
So, we want to pick w to maximize the geometric margin, that is, to maximize $\frac{2}{\|w\|}$. To maximize this expression, we want to minimize the magnitude of w . If we minimize $\frac{1}{2} \|w\|^2$ that is completely equivalent in effect but simpler analytically.

Of course, this is not enough, since we could simply pick $w = 0$ which would be completely useless.

6.034 - Spring 03 • 8

Slide 8.1.9

We'd like to find the w that specifies the maximum margin separator. To be a separator, w needs to classify the points correctly. So, we'll maximize the margin, subject to a set of constraints that require the points to be classified correctly. We will require each point in the training set to have a margin greater than or equal to 1. Requiring the margins to be positive will ensure that they are classified correctly. Requiring them to be greater than or equal to 1 will ensure that the margin of the closest points will be greater than or equal to 1. The fact that we are minimizing the magnitude of w will force the margins to be as small as possible, so that in fact the margins of the closest points will equal 1.



Picking w to Maximize Margin

- Pick w to maximize geometric margin

$$\frac{2}{\|w\|}$$

- or, equivalently, minimize

$$\frac{1}{2} \|w\|^2 = \frac{1}{2} w \cdot w = \frac{1}{2} \sum_j w_j^2$$

- while classifying points correctly

$$y'(w \cdot x' + b) \geq 1$$

- or, equivalently,

$$y'(w \cdot x' + b) - 1 \geq 0$$

6.034 - Spring 03 • 9

Constrained Optimization

$$\min_w \frac{1}{2} \|w\|^2 \text{ subject to } y'(w \cdot x' + b) - 1 \geq 0, \forall i$$

Slide 8.1.10

So, to summarize, we have defined a constrained optimization problem as shown here. It involves minimizing a quadratic function subject to a set of linear constraints. These kinds of optimization problems are very well studied. When the function to be minimized is linear, it is a particularly easy case that can be solved by a "linear programming" algorithm. In our case, it's a bit more complicated.

6.034 - Spring 03 • 10

Slide 8.1.8

The standard approach to solving this type of problem is to convert it to an unconstrained optimization problem by incorporating the constraints as additional terms in the function to be minimized. Each of the constraints is multiplied by a weighting term α_i . Think of these terms as penalty terms that will penalize values of w that do not satisfy the constraints.

Picking w to Maximize Margin

- Pick w to maximize geometric margin

$$\frac{2}{\|w\|}$$

- or, equivalently, minimize

$$\|w\| = \sqrt{w \cdot w}$$

- or, equivalently, minimize

$$\frac{1}{2} \|w\|^2 = \frac{1}{2} w \cdot w = \frac{1}{2} \sum_j w_j^2$$

6.034 - Spring 03 • 8

Constrained Optimization

$$\min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2 \text{ subject to } y^i(\mathbf{w} \cdot \mathbf{x}^i + b) - 1 \geq 0, \forall i$$

Convert to unconstrained optimization by incorporating the constraints as an additional term

$$\min_{\mathbf{w}} \left(\frac{1}{2} \|\mathbf{w}\|^2 - \sum_i \alpha_i [y^i(\mathbf{w} \cdot \mathbf{x}^i + b) - 1] \right) \quad \alpha_i \geq 0, \forall i$$

To minimize expression:

minimize first (original) term, and
maximize second (constraint) term
since $\alpha_i > 0$, encourages constraints to be satisfied
but we want least "distortion" of original term...

6.034 - Spring 03 • 12

Slide 8.1.12

To minimize the combined expression we want to minimize the first term (the magnitude of the weight vector squared) but we want to maximize the constraint term since it is negated. Since $\alpha_i > 0$, making the constraint terms bigger encourages them to be satisfied (we want the margins to be bigger than 1).

However, the bigger the constraint term, the farther we move from the original minimal value of \mathbf{w} . In general we want to minimize this "distortion" of the original problem. We want to introduce just enough distortion to satisfy the constraints. We'll look at this in more detail momentarily.

Slide 8.1.13

This method we have begun to outline here is called the **method of Lagrange multipliers** and the α_i are the individual Lagrange multipliers.

Constrained Optimization

$$\min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2 \text{ subject to } y^i(\mathbf{w} \cdot \mathbf{x}^i + b) - 1 \geq 0, \forall i$$

Convert to unconstrained optimization by incorporating the constraints as an additional term

$$\min_{\mathbf{w}} \left(\frac{1}{2} \|\mathbf{w}\|^2 - \sum_i \alpha_i [y^i(\mathbf{w} \cdot \mathbf{x}^i + b) - 1] \right) \quad \alpha_i \geq 0, \forall i$$

Lagrange multipliers

To minimize expression:

minimize first (original) term, and
maximize second (constraint) term
since $\alpha_i > 0$, encourages constraints to be satisfied
but we want least "distortion" of original term...

Method of Lagrange multipliers

6.034 - Spring 03 • 13

6.034 Notes: Section 8.2

Slide 8.2.1

The details of solving a Lagrange multiplier problem are a little bit complicated. But we are going to go through the derivation at a somewhat abstract level here, because it gives us some insights and intuitions about the resulting solution.

We have an expression, $L(\mathbf{w}, b)$, that also involves parameters alpha. If we knew what the values of alpha should be, we could just fix them, minimize L with respect to \mathbf{w} and b , and be done. The big problem is that we don't know what the alphas are supposed to be.

Maximizing the Margin

$$L(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_i \alpha_i [y^i(\mathbf{w} \cdot \mathbf{x}^i + b) - 1]$$

6.034 - Spring 03 • 14

Maximizing the Margin

$$L(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_i \alpha_i [y^i (\mathbf{w} \cdot \mathbf{x}^i + b) - 1]$$

Minimized when: $\mathbf{w}^* = \sum_i \alpha_i y^i \mathbf{x}^i$ $\sum_i \alpha_i y^i = 0$

6.034 - Spring 03 • 2

Slide 8.2.2

So, we're going to start by imagining that we know what we want the alphas to be. We'll hold them constant for now, and figure out what values of \mathbf{w} and b would optimize L for those fixed alphas. We can do this by taking the partial derivatives of L with respect to \mathbf{w} and b and setting them to zero, getting two constraints. We find that the best value of \mathbf{w} , \mathbf{w}^* is a weighted sum of the input points (in the same form as the dual perceptron); and we get an extra constraint that the sum of the alphas for the positive points has to equal the sum of the alphas for the negative points.

Slide 8.2.3

We can substitute this expression for the optimal \mathbf{w} 's back into our original expression for L , getting L as a function of alpha. Now we have an expression involving only alphas, which we don't know, and \mathbf{x} 's and y 's, which we do know. This function is known as the *dual Lagrangian*. One of the most important things about it, from our perspective, is that the feature vectors only appear in dot products with other feature vectors. We'll come back to this point later on.

Dual Lagrangian

$$\max_{\alpha} L(\alpha) \text{ subject to } \sum_i \alpha_i y^i = 0 \text{ and } \alpha_i \geq 0, \forall i$$

6.034 - Spring 03 • 4

Slide 8.2.4

Now, it's time to pick the best values for the alphas. We do so (for reasons that you'll have to learn in a math class) by choosing the alpha values that maximize this expression. We will retain the constraints that the sum of the alpha values for positive points is equal to the sum of the alpha values for negative points, and that the alphas must be positive.

Note that we will be solving for m alphas. We started with $n+1$ (the number of features, plus one) variables in the original Lagrangian and now we have m (the number of data points) variables in the dual Lagrangian. For the low-dimensional examples we have been dealing with this seems like a horrible tradeoff. We will see later that this can be a very good tradeoff in some circumstances.

We have two constraints, but they are much simpler. One constraint is simply that the alphas be non-negative--this is required because our original constraints were \geq inequalities. The constraint on the alphas comes from the setting to zero of the derivative of the Lagrangian with respect to the offset b .

This problem is not trivial to solve in general; we'll talk more about this later. For now, let us assume that we can solve it and get the optimal values of alphas.

Slide 8.2.5

In the solution, most of the alphas will be zero, corresponding to data points that do not provide binding constraints on the choice of the weights. A few of the data points will have their alphas be nonzero; they will all satisfy their constraints with equality (that is, their margin is equal to 1). These are called **support vectors** and they are the ones used to define the maximum margin separator. You could remove all the other data points and still get the same separator. Because the sparsity of support vectors is so important, this learning method is called a **support vector machine**, or SVM.

Maximizing the Margin

$$L(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_i \alpha_i [y^i (\mathbf{w} \cdot \mathbf{x}^i + b) - 1]$$

Minimized when: $\mathbf{w}^* = \sum_i \alpha_i y^i \mathbf{x}^i$ $\sum_i \alpha_i y^i = 0$

Substituting \mathbf{w}^* into L yields dual Lagrangian:

$$L(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{k=1}^m \alpha_i \alpha_k y_i y_k \mathbf{x}_i \cdot \mathbf{x}_k$$

Only dot products of the feature vectors appear

6.034 - Spring 03 • 3

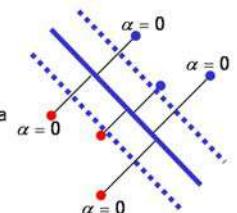
Dual Lagrangian

$$\max_{\alpha} L(\alpha) \text{ subject to } \sum_i \alpha_i y^i = 0 \text{ and } \alpha_i \geq 0, \forall i$$

In general, since $\alpha_i \geq 0$, either

$\alpha_i = 0$: constraint is satisfied with no distortion at optimum \mathbf{w}
or

$\alpha_i > 0$: constraint is satisfied with equality (in this case \mathbf{x}^i is known as a support vector)



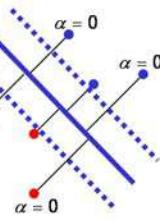
6.034 - Spring 03 • 5

Dual Lagrangian

$$\max_{\alpha} L(\alpha) \text{ subject to } \sum_i \alpha_i y^i = 0 \text{ and } \alpha_i \geq 0, \forall i$$

In general, since $\alpha_i \geq 0$, either
 $\alpha_i = 0$: constraint is satisfied with no distortion at optimum w
or
 $\alpha_i > 0$: constraint is satisfied with equality
(x^i is known as a support vector)

$$w^* = \sum_i \alpha_i y^i x^i \quad b = 1/y^i - w^* x^i$$



Slide 8.2.6

Given the optimal alphas, we can compute the weights. but this time, the coefficients in the sum are the Lagrange multipliers, the alphas, which are mostly zero. This means that the equation of the maximum margin separator depends only on the handful of data points that are closest to it. It makes sense that all the rest of the points would be irrelevant.

We can use the fact that at the support vectors the constraints hold with equality to solve for the value of the offset b . Each such constraint can be used to solve for this scalar.

6.034 - Spring 03 • 6

Slide 8.2.7

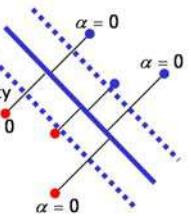
We have not discussed actual algorithms for finding the maxima of the dual Lagrangian. It turns out that the optimization problem we defined is a relatively simple form of the general class of **quadratic programming** problems, which are known to (a) have a unique maximum and (b) can be found using existing algorithms. A number of variations on these algorithms exist but they are beyond our scope.

Dual Lagrangian

$$\max_{\alpha} L(\alpha) \text{ subject to } \sum_i \alpha_i y^i = 0 \text{ and } \alpha_i \geq 0, \forall i$$

In general, since $\alpha_i \geq 0$, either
 $\alpha_i = 0$: constraint is satisfied with no distortion at optimum w
or
 $\alpha_i > 0$: constraint is satisfied with equality
(x^i is known as a support vector)

$$w^* = \sum_i \alpha_i y^i x^i \quad b = 1/y^i - w^* x^i$$



- Has a unique maximum vector
- Can be found using quadratic programming or gradient ascent

6.034 - Spring 03 • 7

SVM Classifier

- Given unknown vector u , predict class (1 or -1) as follows:

$$h(u) = \text{sign}\left(\sum_{i=1}^k \alpha_i y^i x^i \cdot u + b\right)$$

- The sum is over k support vectors

6.034 - Spring 03 • 8

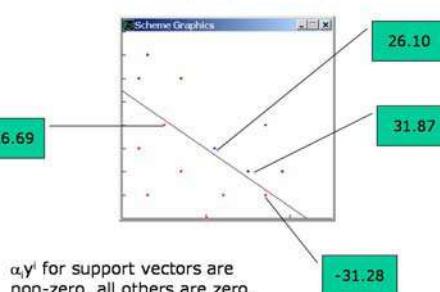
Slide 8.2.8

With the values of the optimal α_i 's and b in hand, and the knowledge of how w is defined, we now have a classifier that we can use on unknown points. Crucially, notice that once again, the only thing we care about are the dot products of the unknown vector with the data points.

Slide 8.2.9

Here's the result of running a quadratic programming algorithm to find the maximal margin separator for the bankruptcy example. Note that only four points have non-zero alpha's. They are the closest points to the line and are the ones that actually define the line.

Bankruptcy Example



6.034 - Spring 03 • 9

Key Points

- Learning depends only on dot products of sample pairs. Recognition depends only on dot products of unknown with samples.

6.034 - Spring 03 • 10

Slide 8.2.10

Let's highlight once again a few of the key points about SVM training and classification. First and foremost, and at the risk of repeating myself, recall that the training and classification of SVMs depends only on the value of the dot products of data vectors. That is, if we have a way of getting the dot products, the computation does not otherwise depend explicitly on the dimensionality of the feature space.

Slide 8.2.8

The fact that we only need dot products (as we will see next) means that we will be able to substitute more general functions for the traditional dot product operation to get more powerful classifiers without really changing anything in the actual training and classification procedures.

SVM Classifier

- Given unknown vector \mathbf{u} , predict class (1 or -1) as follows:

$$h(\mathbf{u}) = \text{sign}\left(\sum_{i=1}^k \alpha_i y^i \mathbf{x}^i \cdot \mathbf{u} + b\right)$$

- The sum is over k support vectors

6.034 - Spring 03 • 8

Key Points

- Learning depends only on dot products of sample pairs. Recognition depends only on dot products of unknown with samples.
- Exclusive reliance on dot products enables approach to non-linearly-separable problems.
- The classifier depends only on the support vectors, not on all the training points.

6.034 - Spring 03 • 12

Slide 8.2.12

Another point to remember is that the resulting classifier does not (in general) depend on all the training points but only on the ones "near the margin", that is, those that help define the boundary between the two classes.

Slide 8.2.13

The maximum margin constraint helps reduce the variance of the SVM hypotheses. Insisting on a minimum magnitude weight vector drastically cuts down on the size of the hypothesis class and helps avoid overfitting.

Key Points

- Learning depends only on dot products of sample pairs. Recognition depends only on dot products of unknown with samples.
- Exclusive reliance on dot products enables approach to non-linearly-separable problems.
- The classifier depends only on the support vectors, not on all the training points.
- Max margin lowers hypothesis variance.

6.034 - Spring 03 • 13

Key Points

- Learning depends only on dot products of sample pairs. Recognition depends only on dot products of unknown with samples.
- Exclusive reliance on dot products enables approach to non-linearly-separable problems.
- The classifier depends only on the support vectors, not on all the training points.
- Max margin lowers hypothesis variance.
- The optimal classifier is defined uniquely – there are no "local maxima" in the search space
- Polynomial in number of data points and dimensionality

6.034 - Spring 03 • 14

Slide 8.2.14

Finally, we should keep firmly in mind that the SVM training process guarantees a unique global maximum. And it runs in time polynomial in the number of data points and the dimensionality of the data.

6.034 Notes: Section 8.3

Slide 8.3.1

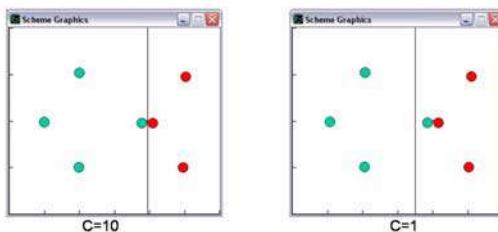
Thus far, we have only been talking about the linearly separable case. What happens for the case in which we have a "nearly separable" problem? That is, some "noise points" that are bound to be misclassified by a linear separator.

It is useful to think about the behavior of the dual perceptron on this type of problem. In that algorithm, the value of the alpha_i for a point is incremented proportionally to its distance to the separator. In fact, if the point is classified correctly, no change is made to the multiplier. We can see that if point i stubbornly resists being classified, then the value of alpha_i will continue to grow without bounds.

The alpha's in the dual perceptron are analogous to the values of the Lagrange multipliers in the SVM. In both cases, the separator is defined as a linear combination of the input points, with the alphas being the weights.

So, one strategy for dealing with these noise points in an SVM is to limit the maximal value of any of the alpha_i's (the Lagrange multipliers) to some C. And, furthermore, to ignore the points with this maximal value when computing the margin. Clearly, if we ignore enough points, we can always get back to a linearly separable problem. By choosing a large value of C, we will work very hard at correctly classifying all the points, a low value of C will allow us to give up more easily on many of the points so as to achieve a better margin.

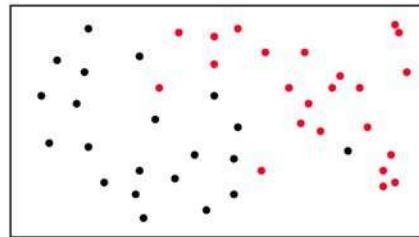
C Change



6.034 - Spring 03 • 2

Not Linearly Separable?

- Require $0 \leq \alpha_i \leq C$
- C specified by user; controls tradeoff between size of margin and classification errors
- C = ∞ for separable case



6.034 - Spring 03 • 1

Slide 8.3.2

This simple example shows how changing C causes the geometric margin to change. High values of C penalize misclassifications more. Low values may permit misclassifications to achieve better margin.

Slide 8.3.3

Here is an example of a separator on a simple data set with four points, which are linearly separable. The colors show the result returned by the classification function on each point in the space. Gray means near 1 or -1. The more intense the blue, the more positive the result; the more intense the red, the more negative. Points lying between the two gray lines return values between -1 and +1.

Note that only three of the four samples are actually used to define w , the ones circled. The other plus sample might as well not be there; its coefficient alpha is zero.

The samples that are actually used are the **support vectors**.

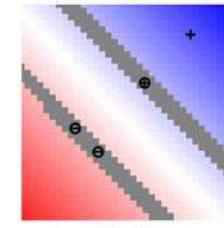
Example: Linearly Separable

Image by Patrick Winston

6.034 - Spring 03 • 3

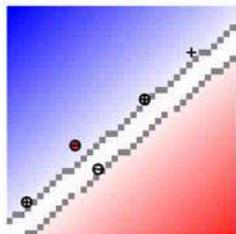
**Another example: Not linearly separable**

Image by Patrick Winston

6.034 - Spring 03 • 4

**Slide 8.3.4**

The next example is the same as the previous example, but with the addition of another plus sample in the lower left corner. There are several points of interest.

First, the optimization has failed to find a separating line, as indicated by the minus sample surrounded by a red disk. The alphas were bounded and so the contribution of this misclassified point is limited and the algorithm converges to a global optimum.

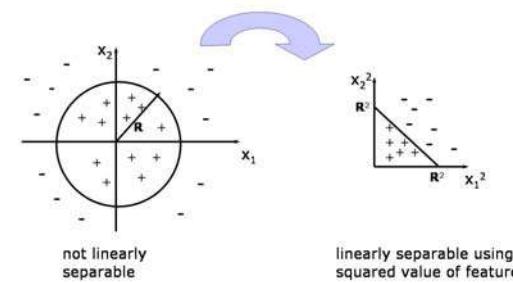
Second, the added point produced quite a different solution. The algorithm is looking for best possible dividing line; a tradeoff between margin and classification error defined by C. If we had kept a solution close to the one in the previous slide, the rogue plus point would have been misclassified by a lot, while with this solution we have reduced the misclassification margin substantially.

Slide 8.3.5

However, even if we provide a mechanism for ignoring noise points, aren't we really limited by a linear classifier? Well, yes.

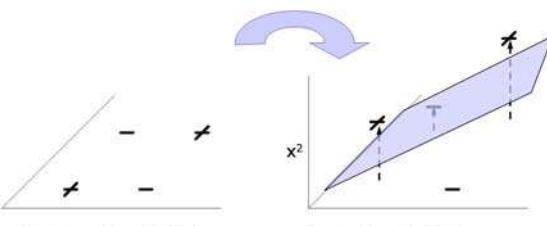
However, in many cases, if we transform the feature values in a non-linear way, we can transform a problem that was not linearly separable into one that is. This example shows that we can create a circular separator by finding a linear classifier in a feature space defined by the squares of the original feature values. That is, we can obtain a non-linear classifier in the original space by finding a linear classifier in a transformed space.

Hold that thought.

Isn't a linear classifier very limiting?

Important: Linear separator in transformed feature space maps into non-linear separator in original feature space

6.034 - Spring 03 • 5

**Not separable?
Try a higher dimensional space!**

Not separable with 2D line

Separable with 3D plane

6.034 - Spring 03 • 6

Slide 8.3.6

Furthermore, when training samples are not separable in the original space they may be separable if you perform a transformation into a higher dimensional space, especially one that is a non-linear transformation of the input space.

For the example shown here, in the original feature space, the samples all lie in a plane, and are not separable by a straight line. In the new space, the samples lie in a three dimensional space, and happen to be separable by a plane.

The heuristic of moving to a higher dimensional space is general, and does not depend on using SVMs.

However, we will see that the support vector approach lends itself to movement into higher dimensional spaces because of the exclusive dependence of the support vector approach on dot products for learning and subsequent classification.

Slide 8.3.7

First, suppose there is a function, phi, that puts the vectors into another, higher-dimensional space, which will also typically involve a non-linear mapping of the feature values. In general, the higher the dimensionality, the more likely there will be a separating hyperplane.

By moving to a higher-dimensional feature space, we are also moving to a bigger hypothesis class, and so we might be worried about overfitting. However, because we are finding the maximum margin separator, the danger of overfitting is greatly reduced.

What you need

- To get into the new feature space, you use $\Phi(\mathbf{x}')$
- The transformation can be to a higher-dimensional feature space and may be non-linear in the feature values.

6.034 - Spring 03 • 7

What you need

- To get into the new feature space, you use $\Phi(\mathbf{x}')$
- The transformation can be to a higher-dimensional feature space and may be non-linear in the feature values.
- Recall that SVM's only use dot products of the data, so
- To optimize classifier, you need $\Phi(\mathbf{x}') \cdot \Phi(\mathbf{x}^*)$
- To run classifier, you need $\Phi(\mathbf{x}') \cdot \Phi(\mathbf{u})$
- So, all you need is a way to compute dot products in transformed space as a function of vectors in original space!

6.034 - Spring 03 • 8

Slide 8.3.8

Even if we aren't in danger of overfitting, there might be computational problems if we move into higher dimensional spaces. In real applications, we might want to move to orders of magnitude more features, or even (in some sense) infinitely many features! We'll need a clever trick to manage this...

You have learned that to work in any space with the support vector approach, you will need (only) the dot products of the samples to train and you will need the dot products of the samples with unknowns to classify.

Note that you don't need anything else. So, all we need is a way of computing the dot product between the transformed feature vectors.

The "Kernel Trick"

- If dot products can be efficiently computed by $\Phi(\mathbf{x}') \cdot \Phi(\mathbf{x}^k) = K(\mathbf{x}', \mathbf{x}^k)$
- Then, all you need is a function on low-dim inputs $K(\mathbf{x}', \mathbf{x}^k)$
- You don't need ever to construct high-dimensional $\Phi(\mathbf{x}')$

6.034 - Spring 03 • 9

Standard Choices For Kernels

- No change (linear kernel)

$$\Phi(\mathbf{x}') \cdot \Phi(\mathbf{x}^k) = K(\mathbf{x}', \mathbf{x}^k) = \mathbf{x}' \cdot \mathbf{x}^k$$

6.034 - Spring 03 • 10

Slide 8.3.10

So now we need to find some phis (mappings from low to high-dimensional space) that have a convenient kernel function associated with them. The simplest case is one where phi is the identity function and K is just the dot product.

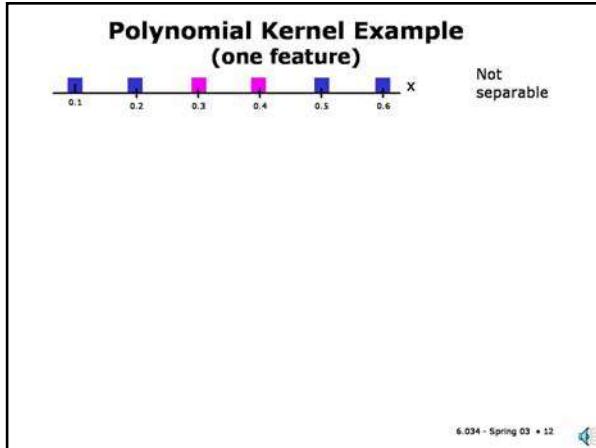
Slide 8.3.8

One such other kernel function is the dot product raised to a power; the actual power is a parameter of the learning algorithm that determines the properties of the solution.

What you need

- To get into the new feature space, you use $\Phi(x')$
- The transformation can be to a higher-dimensional feature space and may be non-linear in the feature values.
- Recall that SVM's only use dot products of the data, so
- To optimize classifier, you need $\Phi(x') \cdot \Phi(x^k)$
- To run classifier, you need $\Phi(x') \cdot \Phi(u)$
- So, all you need is a way to compute dot products in transformed space as a function of vectors in original space!

6.034 - Spring 03 • 8

**Slide 8.3.12**

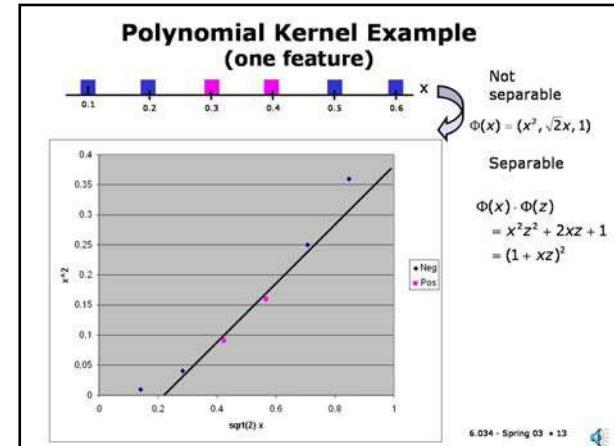
Let's look at a simple example of using a polynomial kernel. Consider the one dimensional problem shown here, which is clearly not separable. Let's map it into a higher dimensional feature space using the polynomial kernel of second degree ($n=2$).

Slide 8.3.13

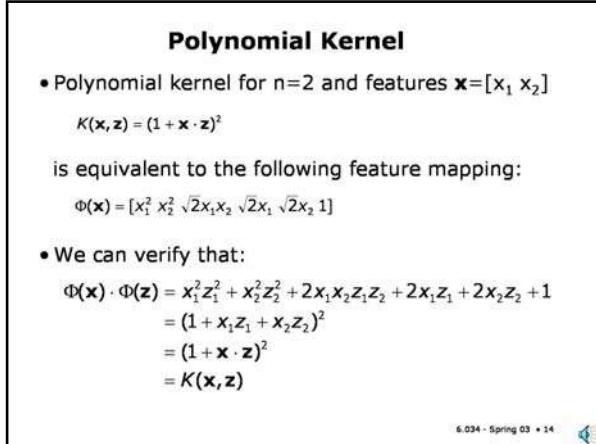
Note that a second degree polynomial kernel is equivalent to mapping the single feature value x to a three dimensional space with feature values x^2 , $\sqrt{2}x$, and 1. You can see that the dot product of two of these feature vectors is exactly the value computed by the polynomial kernel function.

If we plot the original points in the transformed feature space (using just the first two features), we see in fact that the two classes are linearly separable. Clearly, the third feature value (equal to 1) will be irrelevant in finding a separator.

The important aspect of all of this is that we can find and use such a separator without ever explicitly computing the transformed feature vectors - only the kernel function values are required.

**Slide 8.3.14**

Here is a similar transformation for a two dimensional feature vector. Note that the dimension of the transformed feature vector is now 6. In general, the dimension of the transformed feature vector will grow very rapidly with the dimension of the input vector and the degree of the polynomial.

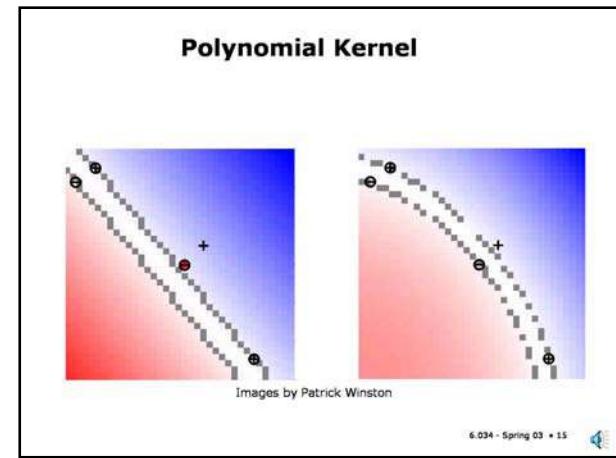


Slide 8.3.15

Let's look at the behavior of these non-linear kernels.

The decision surface produced by the non-linear kernels is curved. Here is an example for which the (unsuccessful) attempt on the left is with a simple dot product; the attempt on the right is done with a polynomial kernel of degree 3. Note the curve in the solution, and note that four of the samples have become support vectors.

Generally, the higher-dimensional the transformed space, the more complex the separator is in the original space, and the more support vectors will be required to specify it.

**Standard Choices For Kernels**

- No change (linear kernel)

$$\Phi(\mathbf{x}') \cdot \Phi(\mathbf{x}^k) = K(\mathbf{x}', \mathbf{x}^k) = \mathbf{x}' \cdot \mathbf{x}^k$$

- Polynomial kernel (n^{th} order)

$$K(\mathbf{x}', \mathbf{x}^k) = (1 + \mathbf{x}' \cdot \mathbf{x}^k)^n$$

- Radial basis kernel (σ is standard deviation)

$$K(\mathbf{x}', \mathbf{x}^k) = e^{-\frac{|\mathbf{x}' - \mathbf{x}^k|^2}{2\sigma^2}} = e^{-\frac{-(\mathbf{x}' - \mathbf{x}^k) \cdot (\mathbf{x}' - \mathbf{x}^k)}{2\sigma^2}}$$

6.034 - Spring 03 • 16

Slide 8.3.16

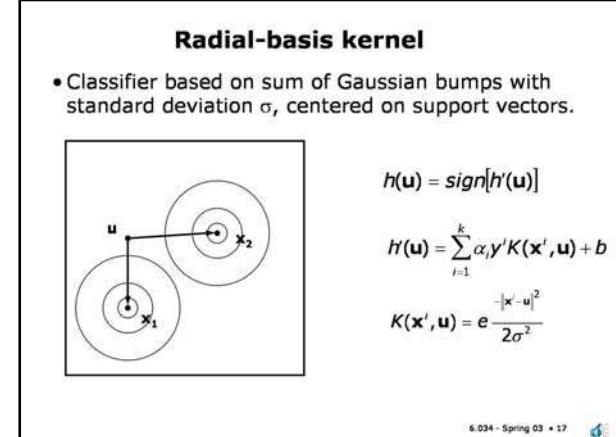
Another popular kernel function is an exponential of the square of the distance between vectors, divided by sigma squared. This is the formula for a Gaussian bump in the feature space, where sigma is the standard deviation of the Gaussian. Sigma is a parameter of the learning that determines the properties of the solution.

Slide 8.3.17

You can get a curved separator if you use radial basis functions, which give us a classifier that is a sum of the values of several Gaussian functions.

Let's pause a minute to observe something that should strike you as a bit weird. When we used the polynomial kernels, we could see that each input feature vector was being mapped into a higher-dimensional, possibly very high dimensional, feature vector. With the radial-basis kernel each input feature vector is being mapped into a **function** that is defined over the whole feature space! In fact, each input feature point is being mapped into a point in an infinite-dimensional feature space (known as a Hilbert space). We then build the classifier as sum of these functions. Whew!

The actual operation of the process is less mysterious than this "infinite-dimensional" mapping view, as we will see by a very simple example.

**Radial-basis kernel**

$$\sigma = 0.1$$



6.034 - Spring 03 • 18

Slide 8.3.18

Along the bottom you see that we're dealing with the simple one-dimensional example that we looked at earlier using a polynomial kernel. The blue points are positive and the pinkish purple ones are negative. Clearly this arrangement is not linearly separable.

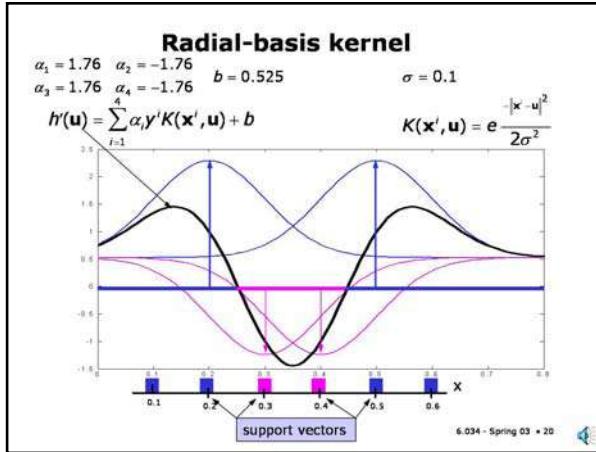
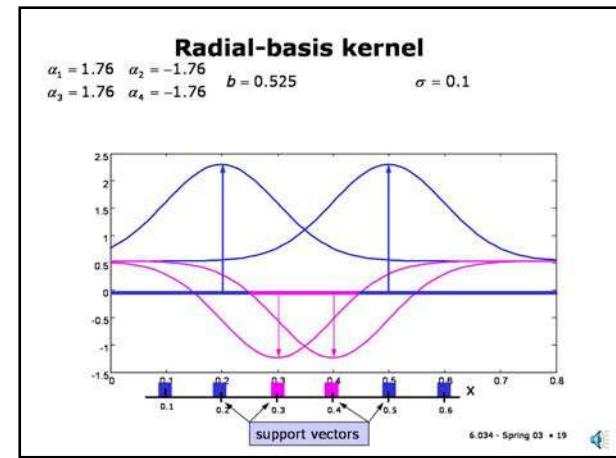
$K(x^i, u)$ can be seen as a "Gaussian bump"; that is, as a function with a maximum at $u = x^i$, that decreases monotonically with the distance between u and x^i , but is always positive and goes to 0 at infinite distance. The parameter sigma specifies how high the bump is and how fast it falls off (the area under the curve of each bump is 1, no matter what the value of sigma is). The smaller the sigma, the more sharply peaked the bump.

With a radial-basis kernel, we will be looking for a set of multipliers for Gaussian bumps with the specified sigma (here it is 0.1) so that the sum of these bumps (plus an offset) will give us a classification function that's positive where the positive points are and negative where the negative points are.

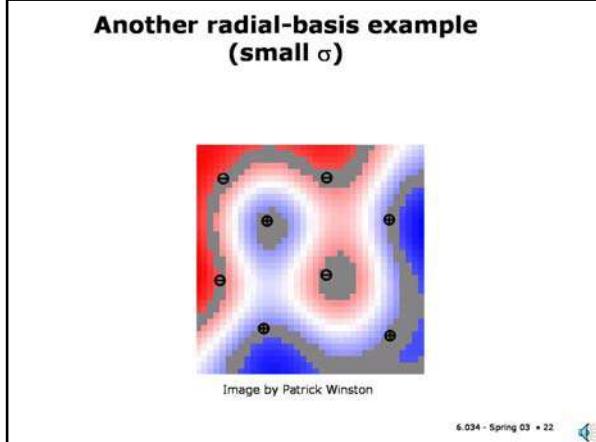
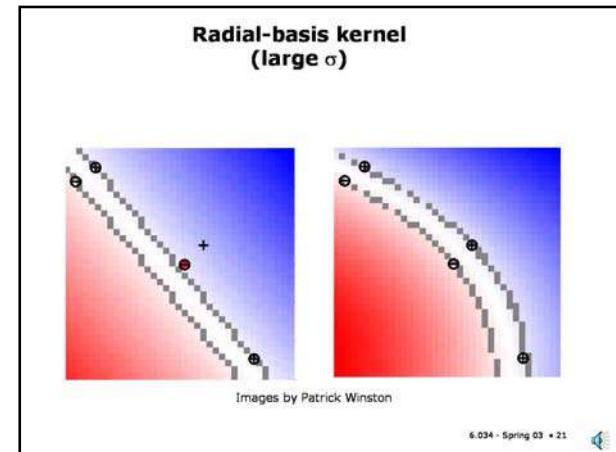
Slide 8.3.19

Here is the solution obtained from an SVM quadratic optimization algorithm. Note that four points are support vectors, as expected, the points near where the decision boundary has to be. The farther positive points receive alpha=0. The value of the offset, b is also shown.

The blue and pink Gaussian bumps correspond to copies of a Gaussian with standard deviation of 0.1 scaled by the corresponding alpha values.

**Slide 8.3.20**

The black line corresponds to the sum of the four bumps (and the offset). The important point is to notice where this line crosses zero since that's the decision surface (in one dimension). Notice that, as required, it succeeds in separating the positive from the negative points.

**Slide 8.3.22**

If a space is truly convoluted, you can always cover it with a radial basis solution with small-enough sigma. In extreme cases, like this one, each of the four plus and four minus samples has become a support vector, each specialized to the small part of the total space in its vicinity. This is basically similar to 1-nearest neighbor and is just as powerful and subject to overfitting.

Slide 8.3.23

At this point alarm bells may be ringing. By creating these very high dimensional feature vectors, are we just setting ourselves up for severe overfitting? Intuitively, the more parameters we have the better we can fit the input, but that may not lead to better performance on new data.

It turns out that the fact that the SVM decision surface depends only on the support vectors and not directly on the dimensionality of the space comes to our rescue.

Cross-Validation Error

- Does mapping to a very high-dimensional space lead to over-fitting?
- Generally, no, thanks to the fact that only the support vectors determine the decision surface.

6.034 - Spring 03 • 23

**Cross-Validation Error**

- Does mapping to a very high-dimensional space lead to over-fitting?
- Generally, no, thanks to the fact that only the support vectors determine the decision surface.
- The expected leave-one-out cross-validation error depends on number of support vectors, not dimensionality of feature space.

$$\text{Expected CV error} \leq \frac{\text{Expected \# support vectors}}{\# \text{ training samples}}$$

- If most data points are support vectors, a sign of possible overfitting, independent of the dimensionality of feature space.

6.034 - Spring 03 • 24

**Slide 8.3.24**

We can estimate the error on new data by computing the cross-validation error on the training data. If we look at the linearly separable case, it is easy to see that the expected value of leave-one-out cross-validation error is bounded by the proportion of support vectors.

If we take a data point that is not a support vector from the training set, the computation of the separator will not be affected and so it will be classified correctly. If we take a support vector out, then the classifier will in general change and there may be an error. So, the expected generalization error depends on the number of support vectors and not on the dimension.

Note that using a radial basis kernel with very small sigma gives you a high expected number of support vectors and therefore a high expected cross-validation error, as expected. Yet, a radial basis kernel with large sigma, although of similar dimensionality, has fewer expected support vectors and is likely to generalize better.

We shouldn't take this bound too seriously; it is not actually very predictive of generalization performance in practice but it does point out an important property of SVMs - that generalization performance is more related to expected number of support vectors than to dimensionality of the transformed feature space.

Slide 8.3.25

So, let's summarize the SVM story. One key point is that SVMs have a training method that guarantees a unique global optimum. This eliminates many headaches in other approaches to machine learning.

Summary

- A single global optimum
 - Quadratic programming or gradient descent

6.034 - Spring 03 • 25

**Summary**

- A single global maximum
 - Quadratic programming or gradient descent
- Fewer parameters
 - C and kernel parameters (n for polynomial, σ for radial basis kernel)

6.034 - Spring 03 • 26

**Slide 8.3.26**

The other advantage of SVMs is that there are relatively few parameters to be chosen: C, the constant used to trade off classification error and width of the margin; and the kernel parameter, such as sigma in the radial basis kernel.

These can both be continuous parameters and so there still remains a search requiring some form of validation, but these are few parameters compared to some of the other methods.

Slide 8.3.27

And, last but not least, is the kernel trick. That is, that the whole process depends only on the dot products of the feature vectors, which is the key to the generalization to non-linear classifiers.

Summary

- A single global maximum
 - Quadratic programming or gradient descent
- Fewer parameters
 - C and kernel parameters (n for polynomial, σ for radial basis kernel)
- Kernel
 - Quadratic minimization depends only on dot products of sample vectors
 - Recognition depends only on dot products of unknown vector with sample vectors
 - Reliance on only dot products enables efficient feature mapping to higher-dimensional spaces where linear separation is more effective.

6.034 - Spring 03 • 27

**Real Data**

- Wisconsin Breast Cancer Data
 - 9 features
 - $C=1$
 - 37 support vectors are used from 512 training data points
 - 12 prediction errors on training set (98% accuracy)
 - 96% accuracy on 171 held out points
 - Essentially same performance as nearest neighbors and decision trees
- Don't expect such good performance on every data set.

6.034 - Spring 03 • 28

**Slide 8.3.28**

The linear separator is very simple hypothesis class but it can perform very well on appropriate data sets. On the Wisconsin breast cancer data, the maximal margin classifier, with a linear kernel, does as well or better as any of the other classifiers we have seen on held-out data. Note that only 37 of the 512 training points are support vectors.

Slide 8.3.29

SVMs have proved useful in a wide variety of applications, particularly those with large numbers of features, such as image and text recognition problems. They are the method of choice in text classification problems, such as categorization of news articles by topic, or spam detection, because they can work in a huge feature space (typically with a linear kernel) without too much fear of overfitting.

Success Stories

- Gene microarray data
 - outperformed all other classifiers
 - specially designed kernel
- Text categorization
 - linear kernel in $>10,000$ D input space
 - best prediction performance
 - 35 times faster to train than next best classifier (decision trees)
- Many others:
 - <http://www.clopinet.com/isabelle/Projects/SVM/applist.html>

6.034 - Spring 03 • 29



6.034 Notes: Section 8.4

Slide 8.4.1

In many machine-learning applications, there are huge numbers of features. In text classification, you often have as many features as there are words in the dictionary. Gene expression arrays have five to fifty thousand elements. Images can have as many as 512 by 512 pixels.

Feature Selection

- In many machine learning applications, there are huge numbers of features
 - text classification (# words)
 - gene arrays (5,000 – 50,000)
 - images (512 x 512 pixels)

6.034 - Spring 03 • 1

Feature Selection

- In many machine learning applications, there are huge numbers of features
 - text classification (# words)
 - gene arrays (5,000 – 50,000)
 - images (512 x 512 pixels)
- Too many features
 - make algorithms run slowly
 - risk overfitting

6.034 - Spring 03 • 2

Slide 8.4.2

When there are lots of features in a domain, it can make some machine learning algorithms run much too slowly. Worse, it often causes overfitting problems: most classifiers have a complexity related to the number of features, and in many of these cases we can have many more features than training examples, which doesn't give us much confidence in our parameter estimates.

Feature Selection

- In many machine learning applications, there are huge numbers of features
 - text classification (# words)
 - gene arrays (5,000 – 50,000)
 - images (512 x 512 pixels)
- Too many features
 - make algorithms run slowly
 - risk overfitting
- Find a smaller feature space
 - subset of existing features
 - new features constructed from old ones

6.034 - Spring 03 • 3

Feature Ranking

- For each feature, compute a measure of its relevance to the output
- Choose the k features with the highest rankings
- Correlation between feature j and output

$$R(j) = \frac{\sum_i (x_j^i - \bar{x}_j)(y^i - \bar{y})}{\sqrt{\sum_i (x_j^i - \bar{x}_j)^2 \sum_i (y^i - \bar{y})^2}}$$

$$x_j = \frac{1}{n} \sum_i x_j^i \quad y = \frac{1}{n} \sum_i y^i$$

- Correlation measures how much x tends to deviate from its mean on the same examples on which y deviates from its mean

Slide 8.4.4

The simplest feature-selection strategy is to compute some score for each feature, and then select the k features with the highest rankings.

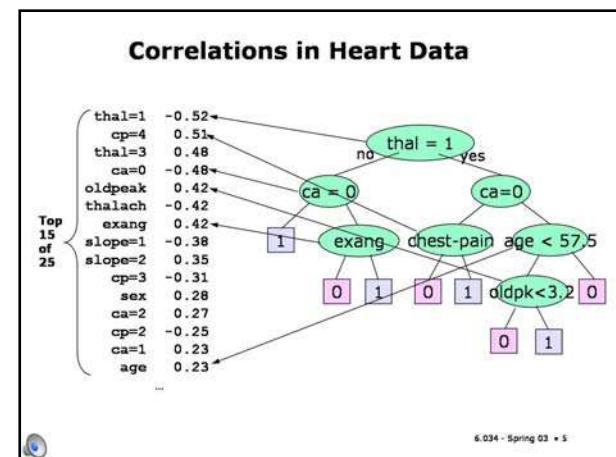
A popular feature score is the correlation between a feature and the output variable. It measures the degree to which a feature varies with the output, and is usable when the output is discrete or continuous.

6.034 - Spring 03 • 4

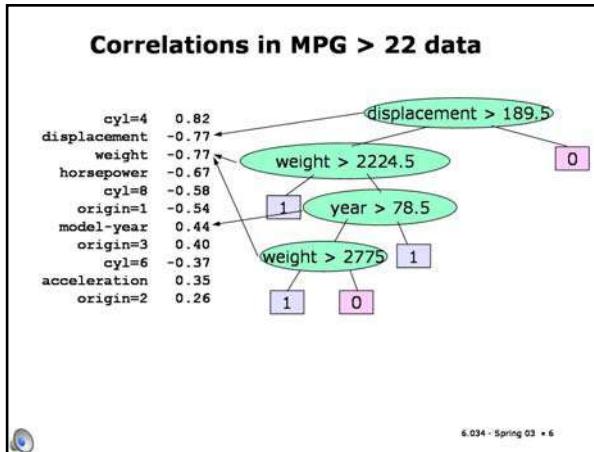
Slide 8.4.5

We computed the correlations of each of the features in the heart disease data set with the output. They are shown here in sorted order, with reference to the decision tree we learned on this data.

We can see that most of the features used in the tree show up among the top features, ranked according to correlation. You can see the features with a positive correlation score indicate that heart disease is more likely, and those with a negative score indicate that it is less likely.

**Slide 8.4.6**

Here's a similar figure for the auto fuel efficiency data. It's interesting to see that the highest-correlation feature is binary choice about whether there are 4 cylinders. It looks like binary features have a tendency to be preferred (since the output is binary, as well, and so they often match up perfectly). But displacement is also very highly ranked, and probably contains more information than the number of cylinders.

**Slide 8.4.7**

As usual, XOR will cause us trouble if we do scoring of single features. In an XOR problem, each feature will, individually, have a correlation of 0 with the output.

To solve xor problems, we need to look at groups of features together.

XOR Bites Back

- As usual, functions with XOR in them will cause us trouble
- Each feature will, individually, have a correlation of 0 (it occurs positively as much as negatively for positive outputs)
- To solve XOR, we need to look at groups of features together

Slide 8.4.8

Ideally, we'd like to try all possible subsets of the features and see which one works best. We can evaluate a subset of features by training a classifier using just that subset, and then measuring the performance using training set or cross-validation error.

Instead of trying all subsets, we'll consider greedy methods that add or subtract features one at a time.

Subset Selection

- Consider subsets of variables
 - too hard to consider all possible subsets
 - wrapper methods: use training set or cross-validation error to measure the goodness of using different feature subsets with your classifier
 - greedily construct a good subset by adding or subtracting features one by one

Slide 8.4.9

In the forward selection method, we start with no features at all in our feature set. Then, for each feature, we consider adding it to the feature set: we add it, train a classifier, and see how well it performs (on a separate validation set or by using cross-validation). We then add the feature that generated the best classifier to our existing set and continue.

We'll terminate the algorithm when we have as many features as we can handle, or when the error has quit decreasing.

Forward Selection

Given a particular classifier you want to use

```
F = {}
For each fj
    Train classifier with inputs F + {fj}
    Add fj that results in lowest-error classifier
        to F
Continue until F is the right size, or error has
quit decreasing
```

6.034 - Spring 03 • 9

Forward Selection

Given a particular classifier you want to use

```
F = {}
For each fj
    Train classifier with inputs F + {fj}
    Add fj that results in lowest-error classifier
        to F
Continue until F is the right size, or error has
quit decreasing
```

- Decision trees, by themselves, do something similar to this

6.034 - Spring 03 • 10

Slide 8.4.10

Decision trees work sort of like this: they add features one at a time, choosing the next feature in the context of the ones already chosen. However, they establish a whole tree of feature-selection contexts.

Slide 8.4.11

Even if we do forward selection, XOR can cause us trouble. Because we only consider adding features one by one, neither of the features will look particularly attractive individually, and so we would be unlikely to add them until the very end.

Forward Selection

Given a particular classifier you want to use

```
F = {}
For each fj
    Train classifier with inputs F + {fj}
    Add fj that results in lowest-error classifier
        to F
Continue until F is the right size, or error has
quit decreasing
```

- Decision trees, by themselves, do something similar to this
- Trouble with XOR

6.034 - Spring 03 • 11

Backward Elimination

Given a particular classifier you want to use

```
F = all features
For each fj
    Train classifier with inputs F - {fj}
    Remove fj that results in lowest-error
        classifier from F
Continue until F is the right size, or error
increases too much
```

6.034 - Spring 03 • 12

Slide 8.4.12

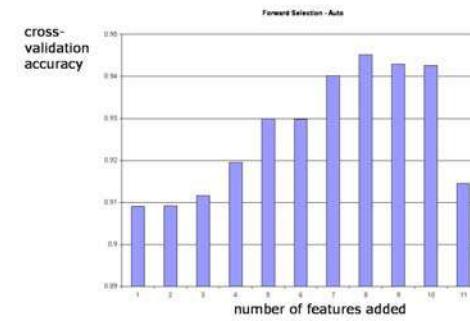
Backward elimination works in the other direction. It starts with all the features in the feature set and eliminates them one by one, removing the one that results in the best classifier at each step.

This strategy can cope effectively with XOR-like problems. But it might be impractical if the initial feature set is so large that it makes the algorithm to slow to run.

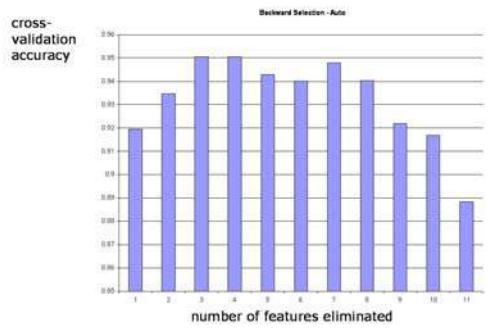
Slide 8.4.13

Here's a plot of the cross-validation accuracy against number of features chosen by forward selection on the auto data. The classifier we used was nearest neighbor.

Here we can see that adding features improves cross-validation accuracy until the last couple of features. If there were a large number of relatively noisy features, adding them would make the performance go down even further, as they would give the classifier further opportunity for overfitting.

Forward Selection on Auto Data

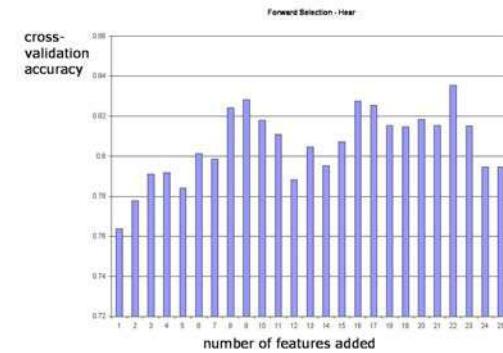
6.034 - Spring 03 • 13

Backward Elimination on Auto Data**Slide 8.4.14**

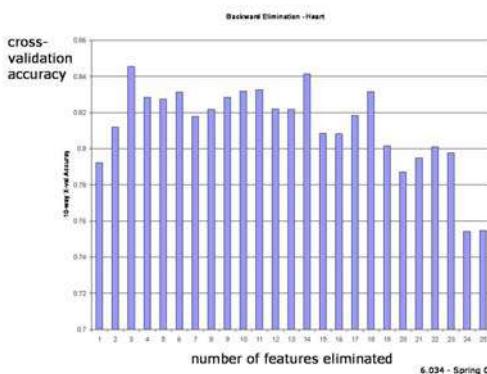
The picture for backward elimination is similar. But notice that it seems to work a bit better, even when we are eliminating a lot of features. This may be because it can decide which features to eliminate in the context of all the other features. Forward selection, especially in the early phases, picks features without much context.

Slide 8.4.15

On the heart data, we need about 8 features before we're getting reasonably good performance. The accuracies are pretty erratic after that; it's probably an indication of overall variance in the performance estimates.

Forward Selection on Heart Data

6.034 - Spring 03 • 15

Backward Elimination on Heart Data**Slide 8.4.16**

We can see similar performance with backward elimination. It's possible to get rid of a lot of features before performance suffers dramatically. And, it really seems to be worthwhile to eliminate some of the features, from a performance perspective.

Slide 8.4.17

Backward elimination and forward selection can be computationally quite expensive, because they require you, on each iteration, to train approximately as many classifiers as you have features.

In some classifiers, such as linear support-vector machines and linear neural networks, it's possible to do backward elimination more efficiently. You train the classifier once, and then remove the feature that has the smallest input weight.

These methods can be extended to non-linear SVMs and neural networks, but it gets somewhat more complicated there.

Recursive Feature Elimination

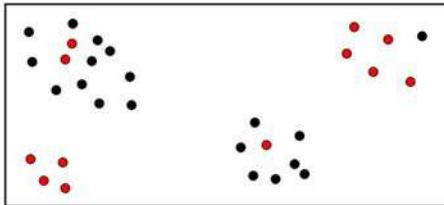
Train a linear SVM or neural network
Remove the feature with the smallest weight
Repeat

- More efficient than regular backward elimination
- Requires only one training phase per feature

6.034 - Spring 03 • 17

Clustering

- Form clusters of inputs
- Map the clusters into outputs
- Given a new example, find its cluster, and generate the associated output



6.034 - Spring 03 • 18

Slide 8.4.18

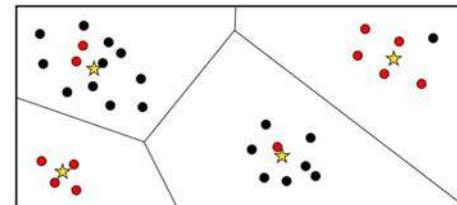
Another whole strategy for feature selection is to make new features. One very drastic method is to try to cluster all of the inputs in your data set into a relatively small number of groups, and then learn a mapping from each group into an output.

Slide 8.4.19

So, in this case, we might divide the input points into 4 clusters. The stars indicate the cluster centers.

Clustering

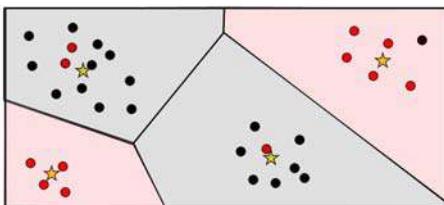
- Form clusters of inputs
- Map the clusters into outputs
- Given a new example, find its cluster, and generate the associated output



6.034 - Spring 03 • 19

Clustering

- Form clusters of inputs
- Map the clusters into outputs
- Given a new example, find its cluster, and generate the associated output



6.034 - Spring 03 • 20

Slide 8.4.20

Then, for each cluster, we would assign the majority class. Now, to predict the value of a new point, we would see which region it would land in, and predict the associated class.

This is different from nearest neighbors in that we actually discard all the data except the cluster centers. This has the advantage of increasing interpretability, since the cluster centers represent "typical" inputs.

Slide 8.4.21

So, what makes a good clustering? There are lots and lots of different technical choices. The basic idea is usually that you want to have clusters in which the distance between points in the same group is small and the distance between points in different groups is large.

Clustering, like nearest neighbor, requires a distance metric, and the results you get are as scale-sensitive as they are in nearest-neighbor.

Clustering Criteria

- small distances between points within a cluster
- large distances between clusters
- Need a distance measure, as in nearest neighbor

6.034 - Spring 03 • 21

K-Means Clustering

- Tries to minimize

$$\sum_{j=1}^k \sum_{x^i \in S_j} \|x^i - \mu_j\|^2$$

squared dist from
point to mean

of clusters elements of cluster j mean of elts in cluster j

- Only gets, greedily, to a local optimum

6.034 - Spring 03 • 22

Slide 8.4.22

One of the simplest and most popular clustering methods is K-means clustering. It tries to minimize the sum, over all the clusters, of the variance of the points within the cluster (the distances of the points to the geometric center of the cluster).

Unfortunately, it only manages to get to a local optimum of this measure, but it's usually fairly reasonable.

Slide 8.4.23

Here is the code for the k-means clustering algorithm. You start by choosing k , your desired number of clusters. Then, you can randomly choose k of your data points to serve as the initial cluster centers.

K-means Algorithm

```
Choose k
Randomly choose k points Cj to be cluster centers
```

6.034 - Spring 03 • 23

K-means Algorithm

```
Choose k
Randomly choose k points Cj to be cluster centers
Loop
  Partition the data into k classes Sj according
  to which of the Cj they're closest to
  For each Sj, compute the mean of its elements
  and let that be the new cluster center
```

Slide 8.4.24

Then, we enter a loop with two steps. The first step is to divide the data up into k classes, using the cluster centers to make a Voronoi partition of the data. That is, we assign each data point to the cluster center that it's closest to.

Now, for each new cluster, we compute a new cluster center by averaging the elements that were assigned to that cluster on the previous step.

6.034 - Spring 03 • 24

Slide 8.4.25

We stop when the centers quit moving. This process is guaranteed to terminate.

K-means Algorithm

```

Choose k
Randomly choose k points Cj to be cluster centers
Loop
    Partition the data into k classes Sj according
    to which of the Cj they're closest to
    For each Sj, compute the mean of its elements
    and let that be the new cluster center
Stop when centers quit moving

```

- Guaranteed to terminate

6.034 - Spring 03 • 25

K-means Algorithm

```

Choose k
Randomly choose k points Cj to be cluster centers
Loop
    Partition the data into k classes Sj according
    to which of the Cj they're closest to
    For each Sj, compute the mean of its elements
    and let that be the new cluster center
Stop when centers quit moving

```

- Guaranteed to terminate
- If a cluster becomes empty, re-initialize the center

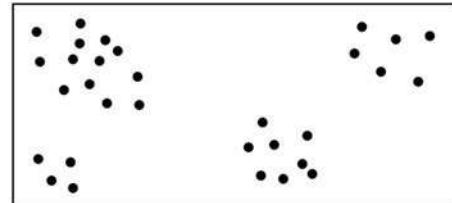
6.034 - Spring 03 • 26

Slide 8.4.26

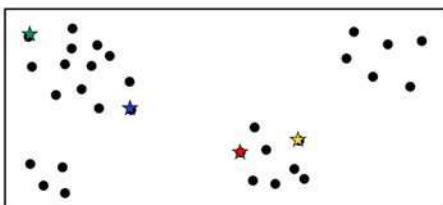
One possible problem is that cluster centers can become "orphaned". That is, they no longer have any points in them (or perhaps just a single point). A standard method for dealing with this problem is simply to randomly re-initialize that cluster center.

Slide 8.4.27

Here's a running example simulation of the k-means algorithm. We start with this set of input points.

K-Means Example

6.034 - Spring 03 • 27

K-Means Example

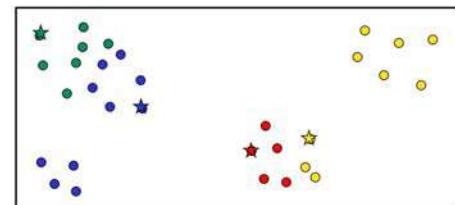
6.034 - Spring 03 • 28

Slide 8.4.28

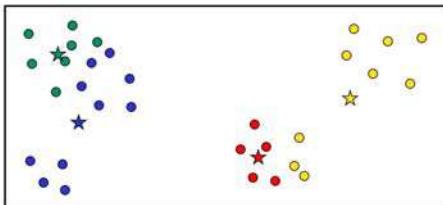
And randomly pick 4 of them to be our cluster centers.

Slide 8.4.29

Now we partition the data, assigning each point to the center to which it is closest.

K-Means Example

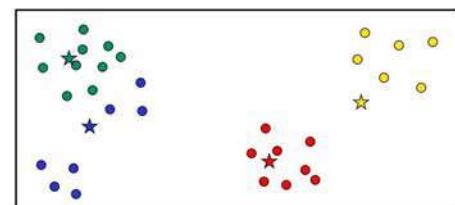
6.034 - Spring 03 • 29

K-Means Example

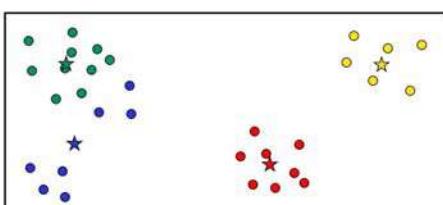
6.034 - Spring 03 • 30

Slide 8.4.30

We move each center to the mean of the points that belong to it.

K-Means Example

6.034 - Spring 03 • 31

K-Means Example

6.034 - Spring 03 • 32

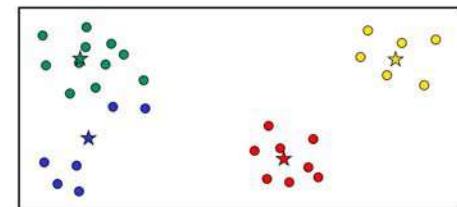
Slide 8.4.32

And recompute the centers.

Slide 8.4.33

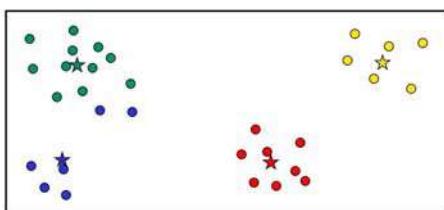
Here we reassign one more point to the green cluster,

K-Means Example



6.034 - Spring 03 • 33

K-Means Example

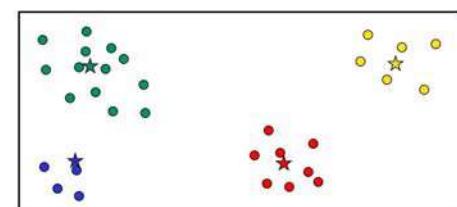


6.034 - Spring 03 • 34

Slide 8.4.34

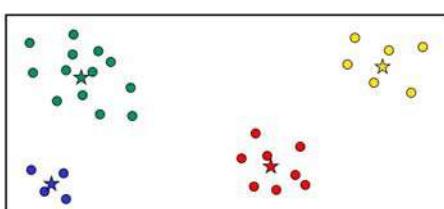
Which causes the green and blue centers to move a bit. At this point, the red and yellow clusters are stable.

K-Means Example



6.034 - Spring 03 • 35

K-Means Example



6.034 - Spring 03 • 36

Slide 8.4.36

And we recompute the centers, to get a clustering that is stable, and will not change under further iterations.

Slide 8.4.37

The k-means algorithm takes a real-valued input space and generates a one-dimensional discrete description of the inputs. In principal components analysis, we take a real-valued space, and represent the data in a new multi-dimensional real-valued space with lower dimensionality. The new coordinates are linear combinations of the originals.

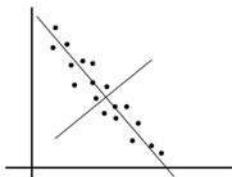
Principal Components Analysis

- Given an n-dimensional real-valued space, data are often nearly restricted to a lower-dimensional subspace
- PCA helps us find such a subspace whose coordinates are linear functions of the originals

6.034 - Spring 03 • 37

Principal Components Analysis

- Given an n-dimensional real-valued space, data are often nearly restricted to a lower-dimensional subspace
- PCA helps us find such a subspace whose coordinates are linear functions of the originals



6.034 - Spring 03 • 38

Slide 8.4.38

The idea is that even if your data are described using a large number of dimensions, they may lie in a lower-dimensional subspace of the original space. So, in this figure, the data are described with two dimensions, but a single dimension that runs diagonally through the data would describe it without losing too much information.

Slide 8.4.39

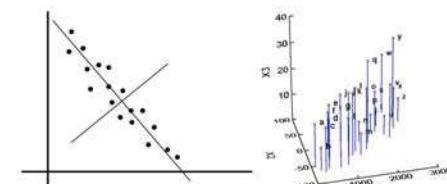
It's harder to see in three dimensions, but here's a data set that might be effectively described using only two dimensions.

Cartoon of algorithm

- Normalize the data (subtract mean, divide by stdev)

Principal Components Analysis

- Given an n-dimensional real-valued space, data are often nearly restricted to a lower-dimensional subspace
- PCA helps us find such a subspace whose coordinates are linear functions of the originals



http://www.okstate.edu/artscl/botany/ordinate/PCA.htm

Slide 8.4.40

To really understand what's going on in this algorithm, you need to have had linear algebra. We'll just give you a "cartoon" idea of how it works.

We start out by normalizing the data (subtracting the mean and dividing by the standard deviation). The new set of coordinates we construct will have its origin at the centroid of the data.

6.034 - Spring 03 • 40

Slide 8.4.41

Now, we find the single line along which the data have the most variance. It's the dimension that, were we to project the data onto it, would result in the most "spread" of the data. We'll let this be our first principal component.

Cartoon of algorithm

- Normalize the data (subtract mean, divide by stdev)
- Find the line along which the data has the most variability: that's the first principal component

6.034 - Spring 03 • 41

Cartoon of algorithm

- Normalize the data (subtract mean, divide by stdev)
- Find the line along which the data has the most variability: that's the first principal component
- Project the data into the n-1 dimensional space orthogonal to the line
- Repeat

6.034 - Spring 03 • 42

Slide 8.4.42

Now, we project the data down into the n-1 dimensional space that's orthogonal to the line we just chose, and repeat.

Cartoon of algorithm

- Normalize the data (subtract mean, divide by stdev)
- Find the line along which the data has the most variability: that's the first principal component
- Project the data into the n-1 dimensional space orthogonal to the line
- Repeat
- Result is a new orthogonal set of axes
- First k give a lower-D space that represents the variability of the data as well as possible

6.034 - Spring 03 • 43

Cartoon of algorithm

- Normalize the data (subtract mean, divide by stdev)
- Find the line along which the data has the most variability: that's the first principal component
- Project the data into the n-1 dimensional space orthogonal to the line
- Repeat
- Result is a new orthogonal set of axes
- First k give a lower-D space that represents the variability of the data as well as possible
- Really: find the eigenvectors of the covariance matrix with the k largest eigenvalues

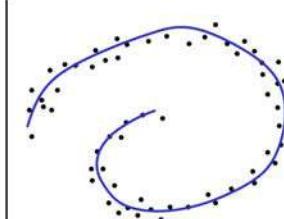
6.034 - Spring 03 • 44

Slide 8.4.44

If you have some experience with linear algebra, then I can tell you that what we really do is find the eigenvectors of the covariance matrix with the k largest eigenvalues.

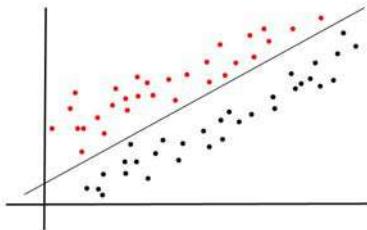
Slide 8.4.45

One problem with PCA (as it's called by its friends) is that it can only produce a set of coordinates that's a linear transformation of the originals. But here's a data set that seems to have a fundamentally one-dimensional structure. Unfortunately, we can't express its axis as a linear combination of the original ones. There are some other cool dimensionality reduction techniques that can actually find this structure!

Linear Transformations Only

There are fancier methods that can find this structure

6.034 - Spring 03 • 45

Insensitive to Classification Task

6.034 - Spring 03 • 46

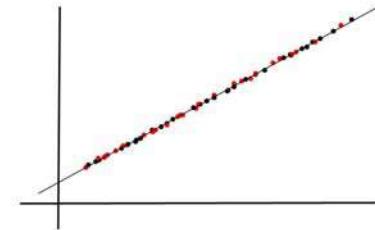
Slide 8.4.46

Another problem with PCA is that it (like k-Means clustering) ignores the classes of the points. So, in this example, the principal component is the line that goes between the two classes (it's a great separator, but that's not what we're looking for right now).

Slide 8.4.47

Now, if we project the data onto that line (which is what would happen if we wanted to reduce the dimensionality of our data set to 1), the positive and negative points are completely intermingled, and we can never get a separator.

There are dimensionality-reduction techniques, also, sadly beyond our scope, that try to optimize the discriminability of the data rather than its variability, which don't suffer from this problem.

Insensitive to Classification Task

There are fancier methods that can take class into account

6.034 - Spring 03 • 47

Validating a Classifier

		predicted y	
		0	1
true y	0	A	B
	1	C	D

6.034 - Spring 03 • 48

Slide 8.4.48

We're just going to tack one additional topic onto the end of this section. It has to do with understanding how well a classifier works. So far, we've been thinking about optimizing training error or cross-validation error, where "error" is measured as the number of examples we get wrong. Let's examine this a little more carefully.

In a binary classification problem, on a single example, there are 4 possible outcomes, depending on the true output value for the input and the predicted output value. In this table, we'll assign values A through D to be the number of times each of these outcomes happens on a data set.

Slide 8.4.49

Case B, in which the answer was supposed to be 0 but the classifier predicted 1 is called a "false positive" or a type 1 error.

Validating a Classifier

		predicted y	
		0	1
true	0	A	B
	1	C	D

false positive
type 1 error

6.034 - Spring 03 • 49

Validating a Classifier

		predicted y	
		0	1
true	0	A	B
	1	C	D

false negative
type 2 error

6.034 - Spring 03 • 50

Slide 8.4.50

Case C, in which the answer was supposed to be 1 but the classifier predicted 0 is called a "false negative" or a type 2 error.

Validating a Classifier

		predicted y	
		0	1
true	0	A	B
	1	C	D

false positive
type 1 error

- sensitivity: $P(\text{predict 1} | \text{actual 1}) = D/(C+D)$
- "true positive rate" (TP)

6.034 - Spring 03 • 51

Validating a Classifier

		predicted y	
		0	1
true	0	A	B
	1	C	D

false negative
type 2 error

- sensitivity: $P(\text{predict 1} | \text{actual 1}) = D/(C+D)$
- "true positive rate" (TP)
- specificity: $P(\text{predict 0} | \text{actual 0}) = A/(A+B)$

Slide 8.4.52

The **specificity** is the probability of predicting a 0 when the actual output is 0.

6.034 - Spring 03 • 52

Slide 8.4.53

The **false-alarm rate** is the probability of predicting a 1 when the actual output is 0. This is also called the **false positive rate**, or FP.

Classifiers are usually characterized using sensitivity and specificity, or using TP and FP.

Validating a Classifier

		predicted y	
		0	1
true y	0	A	B
	1	C	D

false negative type 2 error

• sensitivity: $P(\text{predict } 1 \mid \text{actual } 1) = D/(C+D)$

• "true positive rate" (TP)

• specificity: $P(\text{predict } 0 \mid \text{actual } 0) = A/(A+B)$

• false-alarm rate: $P(\text{predict } 1 \mid \text{actual } 0) = B/(A+B)$

• "false positive rate" (FP)

6.034 - Spring 03 • 53

Cost Sensitivity

- Predict whether a patient has pseuditis based on blood tests
 - Disease is often fatal if left untreated
 - Treatment is cheap and side-effect free

Slide 8.4.54

Imagine that you're a physician and you need to predict whether a patient has pseuditis based on the results of some blood tests. The disease is often fatal if it's left untreated, and the treatment is cheap and relatively side-effect free.

6.034 - Spring 03 • 54

Slide 8.4.55

You have two different classifiers that you could use to make the decision. The first has a true-positive rate of 0.9 and a false-positive rate of 0.4. That means that it will diagnose the disease in 90 percent of the people who actually have it; and also diagnose it in 40 percent of people who don't have it.

Cost Sensitivity

- Predict whether a patient has pseuditis based on blood tests
 - Disease is often fatal if left untreated
 - Treatment is cheap and side-effect free
- Which classifier to use?
 - Classifier 1: TP = 0.9, FP = 0.4

6.034 - Spring 03 • 55

Cost Sensitivity

- Predict whether a patient has pseuditis based on blood tests
 - Disease is often fatal if left untreated
 - Treatment is cheap and side-effect free
- Which classifier to use?
 - Classifier 1: TP = 0.9, FP = 0.4
 - Classifier 2: TP = 0.7, FP = 0.1

Slide 8.4.56

The second classifier only has a true-positive rate of 0.7, but a more reasonable false positive rate of 0.1.

Given the set-up of the problem, we might choose classifier 1, since all those false positives aren't too costly (but if it causes too much hassle, per patient, we might not want to bring 40 percent of them back for treatment).

6.034 - Spring 03 • 56

Slide 8.4.57

One way to address this problem is to start by figuring out the relative costs of the two types of errors. Then, for many classifiers, we can build these costs directly into the choice of classification.

In decision trees, we could use a different splitting criterion. For neural networks we could change the error function to be asymmetric. In SVM's, we could use two different values of C.

Build Costs into Classifier

- Assess costs of both types of error
 - use a different splitting criterion for decision trees
 - make error function for neural nets asymmetric; different costs for each kind of error
 - use different values of C for SVMs depending on kind of error

6.034 - Spring 03 • 57

Tunable Classifiers

- Classifiers that have a threshold (naive Bayes, neural nets, SVMs) can be adjusted, post learning, by changing the threshold, to make different trade-offs between type 1 and type 2 errors

Slide 8.4.58

Often it's useful to deliver a classifier that is tunable. That is, a classifier that has a parameter in it that can be used, at application time, to change the trade-offs made between type 1 and type 2 errors. Most classifiers that have a threshold (such as naive Bayes, neural nets, or SVMs), can be tuned by changing the threshold. At different values of the threshold the classifier will tend to make more errors of one type versus the other.

6.034 - Spring 03 • 58

Slide 8.4.59

In a particular application, we can choose a threshold as follows.

Let c_1 and c_2 be the costs of the two different types of errors; let p be the percentage of positive examples, let x be the threshold parameter that we are allowed to tune, and let $TP(x)$ and $FP(x)$ be the true-positive and false-positive rates, respectively, of the classifier when the threshold is set to have value x .

Then, we can characterize the average, or expected, cost based on this formula, as a function of x . We should choose the value of x that will minimize expected cost.

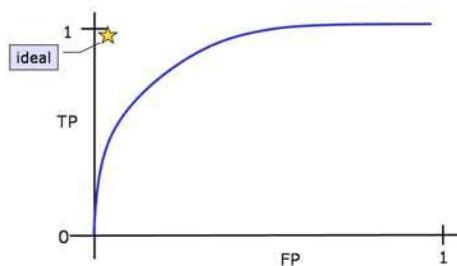
Tunable Classifiers

- Classifiers that have a threshold (naïve Bayes, neural nets, SVMs) can be adjusted, post learning, by changing the threshold, to make different trade-offs between type 1 and type 2 errors
 - C_1, C_2 : costs of errors
 - P : percentage of positive examples
 - x : tunable threshold
 - $TP(x)$: true positive rate at threshold x
 - $FP(x)$: false positive rate at threshold x
 - Expected Cost = $C_1P(1-TP(x)) + C_2(1-P)FP(x)$
 - choose x to minimize expected cost

6.034 - Spring 03 • 59

ROC Curves

- "receiver operating characteristics"



6.034 - Spring 03 • 60

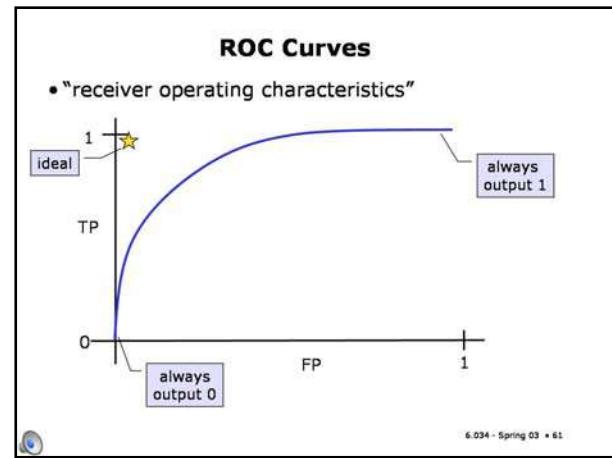
Slide 8.4.60

One way to see the overall performance of a tunable classifier is with a ROC curve. ROC stands for "receiver operating characteristics" from the days of the invention of radar.

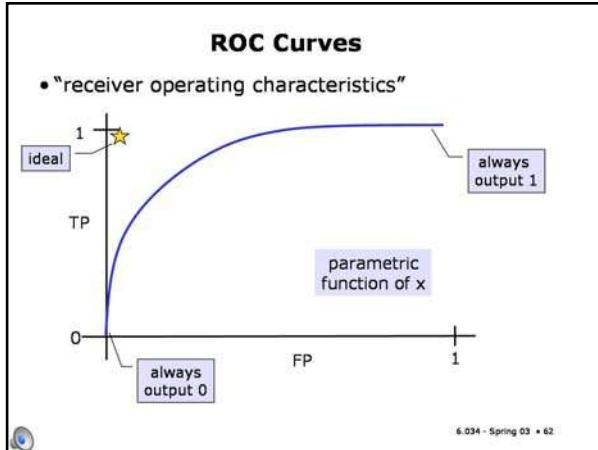
An ROC curve is plotted on two axes; the x axis is the false positive rate and the y axis is the true positive rate. In an ideal world, we would have a false positive rate of 0 and a true positive rate of 1, which would put our performance up near the star on this graph.

Slide 8.4.61

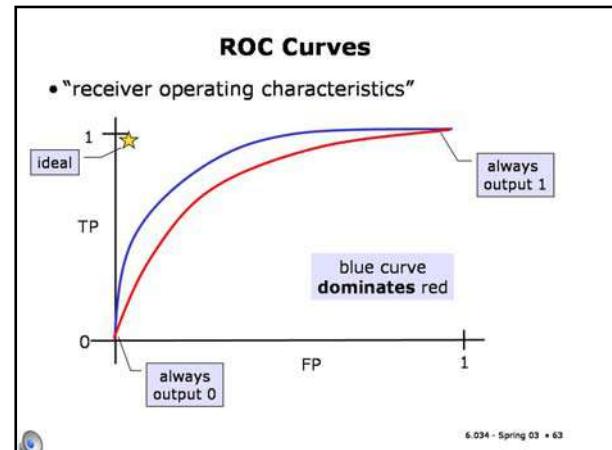
In reality, as we adjust the parameter in the classifier, we typically go from a situation in which the classifier always outputs 0, which generates no false positives and no true positives, to a situation in which the classifier always outputs 1, in which case we have both false positive and true positive rates of 1.



6.034 - Spring 03 • 61

**Slide 8.4.62**

The ROC curve itself is a parametric curve; for each value of x , we plot the pair $FP(x)$, $TP(x)$. The curve shows the range of possible behaviors of the classifier. It is typically shaped something like this blue curve; the higher the false positive rate we can stand, the higher the rate of detecting true positives we can achieve.



6.034 - Spring 03 • 63

Many more issues!

- Missing data
- Many examples in one class, few in other (fraud detection)
- Expensive data (active learning)
- ...

Slide 8.4.64

Machine learning is a huge field that we have just begun to cover. Even in the context of supervised learning, there are a variety of other issues, including how to handle missing data, what to do when you have very many negative examples and just a few positives (such as when you're trying to detect fraud), what to do when getting y values for your x 's is very expensive (you might actively choose which y 's you'd like to have labeled), and many others.

If you like this topic, take a probability course, and then take the graduate machine learning course.



6.034 - Spring 03 • 64

6.034 Notes: Section 9.1

Slide 9.1.1

What is a logic? A logic is a formal language. And what does that mean? It has a syntax and a semantics, and a way of manipulating expressions in the language. We'll talk about each of these in turn.

What is a logic?

- A formal language

6.034 - Spring 03 • 1

What is a logic?

- A formal language
 - Syntax – what expressions are legal

6.034 - Spring 03 • 2

Slide 9.1.2

The syntax is a description of what you're allowed to write down; what the expressions are that are legal in a language. We'll define the syntax of a propositional logic in complete detail later in this section.

What is a logic?

- A formal language
 - Syntax – what expressions are legal
 - Semantics – what legal expressions mean

6.034 - Spring 03 • 3

Slide 9.1.3

The semantics is a story about what the syntactic expressions mean. Syntax is form and semantics is content.

What is a logic?

- A formal language
 - Syntax – what expressions are legal
 - Semantics – what legal expressions mean
 - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)



6.034 - Spring 03 • 4

Slide 9.1.4

A logic usually comes with a proof system, which is a way of manipulating syntactic expressions to get other syntactic expressions. And, why are we interested in manipulating syntactic expressions? The idea is that if we use a proof system with the right kinds of properties, then the new syntactic expressions we create will have semantics or meanings that tell us something "new" about the world.

Slide 9.1.5

So, why would we want to do proofs? There are lots of situations.

What is a logic?

- A formal language
 - Syntax – what expressions are legal
 - Semantics – what legal expressions mean
 - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)
- Why proofs? Two kinds of inferences an agent might want to make:

6.034 - Spring 03 • 5

What is a logic?

- A formal language
 - Syntax – what expressions are legal
 - Semantics – what legal expressions mean
 - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)
- Why proofs? Two kinds of inferences an agent might want to make:
 - Multiple percepts => conclusions about the world



6.034 - Spring 03 • 6

Slide 9.1.6

In the context of an agent trying to reason about its world, think about a situation where we have a bunch of percepts. Let's say we saw somebody come in with a dripping umbrella, we saw muddy tracks in the hallway, we see that there's not much light coming in the windows, we hear pitter-patter-patter. We have all these percepts, and we'd like to draw some conclusion from them, meaning that we'd like to figure out something about what's going on in the world. We'd like to take all these percepts together and draw some conclusion about the world. We could use logic to do that.

Slide 9.1.7

Another use of logic is when you know something about the current state of the world and you know something about the effects of an action that you're considering doing. You wonder what will happen if you take that action. You have a formal description of what that action does in the world. You might want to take those things together and infer something about the next state of the world. So these are two kinds of inferences that an agent might want to do. We could come up with a lot of other ones, but those are two good examples to keep in mind.

What is a logic?

- A formal language
 - Syntax – what expressions are legal
 - Semantics – what legal expressions mean
 - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)
- Why proofs? Two kinds of inferences an agent might want to make:
 - Multiple percepts => conclusions about the world
 - Current state & operator => properties of next state

6.034 - Spring 03 • 7

Propositional Logic Syntax

Slide 9.1.8

We'll look at two kinds of logic: propositional logic, which is relatively simple, and first-order logic, which is more complicated. We're just going to dive right into propositional logic, learn something about how that works, and then try to generalize later on. We'll start by talking about the syntax of propositional logic. Syntax is what you're allowed to write on your paper.

6.034 - Spring 03 • 8

Slide 9.1.9

You're all used to rules of syntax from programming languages, right? In Java you can write a for loop. There are rules of syntax given by a formal grammar. They tell you there has to be a semicolon after fizz; that the parentheses have to match, and so on. You can't make random changes to the characters in your program and expect the compiler to be able to interpret it. So, the syntax is what symbols you're allowed to write down in what order. Not what they mean, not what computation they symbolize, but just what symbols you can write down.

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`

6.034 - Spring 03 • 9

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

6.034 - Spring 03 • 10

Slide 9.1.10

Another famous illustration of syntax is this one, due to the linguist Noam Chomsky: "Colorless green ideas sleep furiously". The idea is that it doesn't mean anything really, but it's syntactically well-formed. It's got the nouns, the verbs, and the adjectives in the right places. If you scrambled the words up, you wouldn't get a sentence, right? You'd just get a string of words that didn't obey the rules of syntax. So, "furiously ideas green sleep colorless" is not syntactically okay.

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

6.034 - Spring 03 • 11

Slide 9.1.11

Let's define the syntax of propositional logic. We'll call the legal things to write down "sentences". So if something is a sentence, it is a syntactically okay thing in our language. Sometimes sentences are called "WFFs" (which stands for "well-formed formulas") in other books.

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

- true and false are sentences

Slide 9.1.12

We're going to define the set of legal sentences recursively. So here are two base cases: The words, "true" and "false", are sentences.

6.034 - Spring 03 • 12

Slide 9.1.13

Propositional variables are sentences. I'll give you some examples. P, Q, R, Z. We're not, for right now, defining a language that a computer is going to read. And so we don't have to be absolutely rigorous about what characters are allowed in the name of a variable. But there are going to be things called variables, and we'll just use uppercase letters for them. Those are sentences. It's OK to say "P" -- that's a sentence.

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z
- If ϕ and ψ are sentences, then so are
 (ϕ) , $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$

6.034 - Spring 03 • 14

Slide 9.1.14

Now, here's the recursive part. If Phi and Psi are sentences, then so are -- Wait! What, exactly, are Phi and Psi? They're called metavariables, and they range over expressions. This rule says that if Phi and Psi are things that you already know are sentences because of one of these rules, then you can make more sentences out of them. Phi with parentheses around it is a sentence. Not Phi is a sentence (that little bent thing is our "not" symbol (but we're not really supposed to know that yet, because we're just doing syntax right now)). Phi "vee" Psi is a sentence. Phi "wedge" Psi is a sentence. Phi "arrow" Psi is a sentence. Phi "two-headed arrow" Psi is a sentence.

Slide 9.1.15

And there's one more part of the definition, which says nothing else is a sentence. OK. That's the syntax of the language.

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z

6.034 - Spring 03 • 13

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z
- If ϕ and ψ are sentences, then so are
 (ϕ) , $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$

Propositional Logic Syntax

Syntax: what you're allowed to write

- `for (thing t = fizz; t == fuzz; t++) { ... }`
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z
- If ϕ and ψ are sentences, then so are
 (ϕ) , $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$
- Nothing else is a sentence

6.034 - Spring 03 • 15

Precedence

\neg \wedge \vee \rightarrow \leftrightarrow	<p>highest</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%; padding: 5px;">$A \vee B \wedge C$</td> <td style="width: 50%; padding: 5px;">$A \vee (B \wedge C)$</td> </tr> <tr> <td style="padding: 5px;">$A \wedge B \rightarrow C \vee D$</td> <td style="padding: 5px;">$(A \wedge B) \rightarrow (C \vee D)$</td> </tr> <tr> <td style="padding: 5px;">$A \rightarrow B \vee C \leftrightarrow D$</td> <td style="padding: 5px;">$(A \rightarrow (B \vee C)) \leftrightarrow D$</td> </tr> </table> <p>lowest</p>	$A \vee B \wedge C$	$A \vee (B \wedge C)$	$A \wedge B \rightarrow C \vee D$	$(A \wedge B) \rightarrow (C \vee D)$	$A \rightarrow B \vee C \leftrightarrow D$	$(A \rightarrow (B \vee C)) \leftrightarrow D$
$A \vee B \wedge C$	$A \vee (B \wedge C)$						
$A \wedge B \rightarrow C \vee D$	$(A \wedge B) \rightarrow (C \vee D)$						
$A \rightarrow B \vee C \leftrightarrow D$	$(A \rightarrow (B \vee C)) \leftrightarrow D$						

- Precedence rules enable "shorthand" form of sentences, but formally only the fully parenthesized form is legal.
- Syntactically ambiguous forms allowed in shorthand only when semantically equivalent: $A \wedge B \wedge C$ is equivalent to $(A \wedge B) \wedge C$ and $A \wedge (B \wedge C)$

6.034 - Spring 03 • 16

Slide 9.1.16

There's actually one more issue we have to sort out. Precedence of the operations. If we were being really careful, we'd require you to put parentheses around each new sentence that you made out of component sentences using negation, vee, wedge, or arrow. But it starts getting kind of ugly if we do that. So, we allow you to leave out some of the parentheses, but then we need rules to figure out where the implicit parentheses really are. Those are precedence rules. Just as in arithmetic, where we learned that multiplication binds tighter than addition, we have similar rules in logic. So, to add the parentheses to a sentence, you start with the highest precedence operator, which is negation. For every negation, you'd add an open paren in front of the negation sign and a close parenthesis after the next whole expression. This is exactly how minus behaves in arithmetic. The next highest operator is wedge, which behaves like multiplication in arithmetic. Next is vee, which behaves like addition in arithmetic. Logic has two more operators, with weaker precedence. Next comes single arrow, and last is double arrow. Also, wedge and vee are associative.

6.034 Notes: Section 9.2**Slide 9.2.1**

Let's talk about semantics. The semantics of a sentence is its meaning. What does it say about the world? We could just write symbols on the board and play with them all day long, and it could be fun; it could be like doing puzzles. But ultimately the reason that we want to be doing something with these kinds of logical sentences is because they somehow say something about the world. And it's really important to be clear about the connections between the things that we write on the board and what we think of them as meaning in the world, what they stand for. And it's going to be something different every day. I remember once when I was a little kid, I was on the school bus. And somebody's big sister or brother had started taking algebra and this kid told me, "You know what? My big sister's taking algebra and A equals 3!" The reason that sounds so silly is that A is a variable. Our variables are going to be the same. They'll have different interpretations in different situations. So, in our study of logic, we're not going to assign particular values or meanings to the variables; rather, we're going to study the general properties of symbols and their potential meanings.

Semantics

6.034 - Spring 03 • 1

Semantics

- Meaning of a sentence is truth value {t, f}

Slide 9.2.2

Ultimately, the meaning of every sentence, in a situation, will be a truth value, t or f. Just as, in high-school algebra, the meaning of every expression is a numeric value. Note that there's already a really important difference between underlined true and false, which are syntactic entities that we can write on the board, and the truth values t and f which stand for the abstract philosophical ideals of truth and falsity.

6.034 - Spring 03 • 2

Slide 9.2.3

How can we decide whether A "wedge" B "wedge" C is true or not? Well, it has to do with what A and B and C stand for in the world. What A and B and C stand for in the world will be given by an object called an "interpretation". An interpretation is an assignment of truth values to the propositional variables. You can think of it as a possible way the world could be. So if our set of variables is P, Q, R, and V, then P true, Q false, R true, V true, that would be an interpretation. So then, given an interpretation, we can ask the question, is this sentence true in that interpretation? We will write "holds Phi comma i" to mean "sentence Phi is true in interpretation i". The "holds" symbol is not part of our language. It's part of the way logicians write things on the board when they're talking about what they're doing. This is a really important distinction. If you can think of our sentences like expressions in a programming language, then you can think of these expressions with "holds" as being about whether programs work in a certain way or not. In order to even think about whether Phi is true in interpretation I, Phi has to be a sentence. If it's not a well-formed sentence, then it doesn't even make sense to ask whether it's true or false.

Semantics

- Meaning of a sentence is truth value {**t, f**}
- Interpretation is an assignment of truth values to the propositional variables

$\text{holds}(\phi, i)$ [Sentence ϕ is **t** in interpretation i]

6.034 - Spring 03 • 3

Semantics

- Meaning of a sentence is truth value {**t, f**}
- Interpretation is an assignment of truth values to the propositional variables

$\text{holds}(\phi, i)$ [Sentence ϕ is **t** in interpretation i]
 $\text{fails}(\phi, i)$ [Sentence ϕ is **f** in interpretation i]

Semantic Rules**Slide 9.2.5**

So now we can write down the rules of the semantics. We can write down rules that specify when sentence Phi is true in interpretation i. We are going to specify the semantics of sentences recursively, based on their syntax. The definition of a semantics should look familiar to most of you, since it's very much like the specification of an evaluator for a functional programming language, such as Scheme.

6.034 - Spring 03 • 4

Semantic Rules

- $\text{holds}(\text{true}, i)$ for all i

Slide 9.2.6

First, the sentence consisting of the symbol "true" is true in all interpretations.

6.034 - Spring 03 • 5



6.034 - Spring 03 • 4



6.034 - Spring 03 • 6

Slide 9.2.7

The sentence consisting of a symbol "false" has truth value **f** in all interpretations.

Semantic Rules

- $\text{holds}(\text{true}, i)$ for all i
- $\text{fails}(\text{false}, i)$ for all i

6.034 - Spring 03 • 7

Semantic Rules

- $\text{holds}(\text{true}, i)$ for all i
- $\text{fails}(\text{false}, i)$ for all i
- $\text{holds}(\neg\phi, i)$ if and only if $\text{fails}(\phi, i)$
(negation)

Slide 9.2.8

Now we can do the connectives. We'll leave out the parentheses. The truth value of a sentence with top-level parentheses is the same as the truth value of the sentence with the parentheses removed. Now, let's think about the "not" sign. When is "not" Phi true in an interpretation i ? Whenever Phi is false in that interpretation.

Slide 9.2.9

When is Phi "wedge" Psi true in an interpretation i ? Whenever both Phi and Psi are true in i . This is called "conjunction". And we'll start calling that symbol "and" instead of "wedge", now that we know what it means.

Semantic Rules

- $\text{holds}(\text{true}, i)$ for all i
- $\text{fails}(\text{false}, i)$ for all i
- $\text{holds}(\neg\phi, i)$ if and only if $\text{fails}(\phi, i)$
(negation)
- $\text{holds}(\phi \wedge \psi, i)$ iff $\text{holds}(\phi, i)$ and $\text{holds}(\psi, i)$
(conjunction)

6.034 - Spring 03 • 8

Semantic Rules

- $\text{holds}(\text{true}, i)$ for all i
- $\text{fails}(\text{false}, i)$ for all i
- $\text{holds}(\neg\phi, i)$ if and only if $\text{fails}(\phi, i)$
(negation)
- $\text{holds}(\phi \wedge \psi, i)$ iff $\text{holds}(\phi, i)$ and $\text{holds}(\psi, i)$
(conjunction)
- $\text{holds}(\phi \vee \psi, i)$ iff $\text{holds}(\phi, i)$ or $\text{holds}(\psi, i)$
(disjunction)

Slide 9.2.10

When is Phi "vee" Psi true in an interpretation i ? Whenever either Phi or Psi is true in i . This is called "disjunction", and we'll call the "vee" symbol "or". It is not an exclusive or; so that if both Phi and Psi are true in i , then Phi "vee" Psi is also true in i .

6.034 - Spring 03 • 9

6.034 - Spring 03 • 10

Slide 9.2.11

Now we have one more clause in our definition. I'm going to do it by example. Imagine that we have a sentence P. P is one of our propositional variables. How do we know whether it is true in interpretation i? Well, since i is a mapping from variables to truth values, I can simply look P up in i and return whatever truth value was assigned to P by i.

Semantic Rules

- $\text{holds}(\text{true}, i)$ for all i
- $\text{fails}(\text{false}, i)$ for all i
- $\text{holds}(\neg\phi, i)$ if and only if $\text{fails}(\phi, i)$
(negation)
- $\text{holds}(\phi \wedge \psi, i)$ iff $\text{holds}(\phi, i)$ and $\text{holds}(\psi, i)$
(conjunction)
- $\text{holds}(\phi \vee \psi, i)$ iff $\text{holds}(\phi, i)$ or $\text{holds}(\psi, i)$
(disjunction)
- $\text{holds}(P, i)$ iff $i(P) = t$
- $\text{fails}(P, i)$ iff $i(P) = f$

6.034 - Spring 03 • 11

Some important shorthand**Slide 9.2.12**

It seems like we left out the arrows in the semantic definitions of the previous slide. But the arrows are not strictly necessary; that is, it's going to turn out that you can say anything you want to without them, but they're a convenient shorthand. (In fact, you can also do without either "or" or "and", but we'll see that later).

6.034 - Spring 03 • 12

Slide 9.2.13

So, we can define Phi "arrow" Psi as being equivalent to not Phi or Psi. That is, no matter what Phi and Psi are, and in every interpretation, (Phi "arrow" Psi) will have the same truth value as (not Phi or Psi). We will now call this arrow relationship "implication". We'll say that Phi implies Psi. We may also call this a conditional expression: Psi is true if Phi is true. In such a statement, Phi is referred to as the antecedent and Psi as the consequent.

Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (**conditional, implication**)
antecedent → consequent

6.034 - Spring 03 • 13

Some important shorthand**Slide 9.2.14**

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (**conditional, implication**)
antecedent → consequent
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (**biconditional, equivalence**)

Finally, the double arrow just means that we have single arrows going both ways. This is sometimes called a "bi-conditional" or "equivalence" statement. It means that in every interpretation, Phi and Psi have the same truth value.

6.034 - Spring 03 • 14

Slide 9.2.15

Just so you can see how all of these operators work, here are the truth tables. Consider a world with two propositional variables, P and Q. There are four possible interpretations in such a world (one for every combination of assignments to the variables; in general, in a world with n variables, there will be 2^n possible interpretations). Each row of the truth table corresponds to a possible interpretation, and we've filled in the values it assigns to P and Q in the first two columns. Once we have chosen an interpretation (a row in the table), then the semantic rules tell us exactly what the truth value of every single legal sentence must be. Here we show the truth values for six different sentences made up from P and Q.

Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (conditional, implication)
antecedent → consequent
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (biconditional, equivalence)

Truth Tables

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$Q \rightarrow P$	$P \leftrightarrow Q$
f	f	t	f	f	t	t	t
f	t	t	f	t	t	f	f
t	f	f	f	t	f	t	f
t	t	f	t	t	t	t	t

6.034 - Spring 03 • 15

Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (conditional, implication)
antecedent → consequent
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (biconditional, equivalence)

Truth Tables

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$Q \rightarrow P$	$P \leftrightarrow Q$
f	f	t	f	f	t	t	t
f	t	t	f	t	t	f	f
t	f	f	f	t	f	t	f
t	t	f	t	t	t	t	t

Note that implication is not "causality", if P is f then $P \rightarrow Q$ is t

6.034 - Spring 03 • 16

Slide 9.2.16

Most of them are fairly obvious, but it's worth studying the truth table for implication fairly closely. In particular, note that (P implies Q) is true whenever P is false. You can see that this is reasonable by thinking about an English sentence like "If pigs can fly then ...". Once you start with a false condition, you can finish with anything, and the sentence will be true. Implication doesn't mean "causes". It doesn't mean "is related" in any kind of real-world way; it is just a bare, formal definition of not P or Q.

Terminology**Slide 9.2.17**

Now we'll define some terminology on this slide and the next, then do a lot of examples.

6.034 - Spring 03 • 17

Terminology

- A sentence is **valid** iff its truth value is t in all interpretations

Valid sentences: true, false, $P \vee \neg P$

Slide 9.2.18

A sentence is **valid** if and only if it is true in all interpretations. We have already seen one example of a valid sentence. What was it? True. Another one is "not false". A more interesting one is "P or not P". No matter what truth value is assigned to P by the interpretation, "P or not P" is true.

6.034 - Spring 03 • 18

Slide 9.2.19

A sentence is satisfiable if and only if it's true in at least one interpretation. The sentence P is satisfiable. The sentence True is satisfiable. Not P is satisfiable.

Terminology

- A sentence is **valid** iff its truth value is **t** in all interpretations

Valid sentences: **true**, **\neg false**, **P $\vee \neg P$**

- A sentence is **satisfiable** iff its truth value is **t** in at least one interpretation

Satisfiable sentences: **P**, **true**, **$\neg P$**

6.034 - Spring 03 • 19

Terminology

- A sentence is **valid** iff its truth value is **t** in all interpretations
Valid sentences: **true**, **\neg false**, **P $\vee \neg P$**
- A sentence is **satisfiable** iff its truth value is **t** in at least one interpretation
Satisfiable sentences: **P**, **true**, **$\neg P$**
- A sentence is **unsatisfiable** iff its truth value is **f** in all interpretations
Unsatisfiable sentences: **P $\wedge \neg P$** , **false**, **$\neg true$**

6.034 - Spring 03 • 20

Slide 9.2.20

A sentence is unsatisfiable if and only if it's false in every interpretation. Some unsatisfiable sentences are: false, not true, P and not P.

Slide 9.2.21

We can use the method of truth tables to check these things. If I wanted to know whether a particular sentence was valid, or if I wanted to know if it was satisfiable or unsatisfiable, I could just make a truth table. I'd write down all the interpretations, figure out the value of the sentence in each interpretation, and if they're all true, it's valid. If they're all false, it's unsatisfiable. If it's somewhere in between, it's satisfiable. So there's a reliable way; there's a completely doozy, tedious, mechanical way to figure out if a sentence has one of these properties. That's not true in all logics. This is a useful, special property of propositional logic. It might take you a lot of time, but it's a finite amount of time and you can decide any of these questions.

Terminology

- A sentence is **valid** iff its truth value is **t** in all interpretations

Valid sentences: **true**, **\neg false**, **P $\vee \neg P$**

- A sentence is **satisfiable** iff its truth value is **t** in at least one interpretation

Satisfiable sentences: **P**, **true**, **$\neg P$**

- A sentence is **unsatisfiable** iff its truth value is **f** in all interpretations

Unsatisfiable sentences: **P $\wedge \neg P$** , **false**, **$\neg true$**

All are finitely decidable.

6.034 - Spring 03 • 21

Examples**Slide 9.2.22**

Let's work through some examples. We can think about whether they're valid or unsatisfiable or satisfiable.

6.034 - Spring 03 • 22

Slide 9.2.23

What about "smoke implies smoke"? Rather than doing a whole truth table it might be easier if we can convert it into smoke or not smoke, right? The definition of A implies B is not A or B. And we said that smoke or not smoke was valid already.

Examples		
Sentence	Valid?	Interpretation that make sentence's truth value = f
smoke \rightarrow smoke		
smoke v ¬smoke	{ valid	

6.034 - Spring 03 • 23



Examples		
Sentence	Valid?	Interpretation that make sentence's truth value = f
smoke \rightarrow smoke		
smoke v ¬smoke	{ valid	
smoke \rightarrow fire	{ satisfiable, not valid	smoke = t, fire = f

6.034 - Spring 03 • 24

**Slide 9.2.24**

What about "smoke implies fire"? It's satisfiable, because there's an interpretation of these two symbols that makes it true. There are other interpretations that make it false. I should say, everything that's valid is also satisfiable.

Examples		
Sentence	Valid?	Interpretation that make sentence's truth value = f
smoke \rightarrow smoke		
smoke v ¬smoke	{ valid	
smoke \rightarrow fire	{ satisfiable, not valid	smoke = t, fire = f
$(s \rightarrow f) \rightarrow (\neg s \rightarrow \neg f)$	{ satisfiable, not valid	s = f, f = t s → f = t, ¬s → ¬f = f

6.034 - Spring 03 • 25



Examples		
Sentence	Valid?	Interpretation that make sentence's truth value = f
smoke \rightarrow smoke		
smoke v ¬smoke	{ valid	
smoke \rightarrow fire	{ satisfiable, not valid	smoke = t, fire = f
$(s \rightarrow f) \rightarrow (\neg s \rightarrow \neg f)$	{ satisfiable, not valid	s = f, f = t s → f = t, ¬s → ¬f = f
contrapositive $(s \rightarrow f) \rightarrow (\neg f \rightarrow \neg s)$	{ valid	

6.034 - Spring 03 • 26

**Slide 9.2.26**

Reasoning in the other direction is okay, though. So the sentence "smoke implies fire implies not fire implies not smoke" is valid. And for those of you who love terminology, this thing is called the contrapositive. So, if there's no fire, then there's no smoke.

Slide 9.2.27

What about "b or d or (b implies d)"? We can rewrite that (using the definition of implication) into "b or d or not b or d", which is valid, because in every interpretation either b or not b must be true.

Sentence	Valid?	Interpretation that make sentence's truth value = f
$\text{smoke} \rightarrow \text{smoke}$	{ valid }	
$\text{smoke} \vee \neg \text{smoke}$		
$\text{smoke} \rightarrow \text{fire}$	{ satisfiable, not valid }	$\text{smoke} = \text{t}, \text{fire} = \text{f}$
$(\text{s} \rightarrow \text{f}) \rightarrow (\neg \text{s} \rightarrow \neg \text{f})$	{ satisfiable, not valid }	$\text{s} = \text{f}, \text{f} = \text{t}$ $\text{s} \rightarrow \text{f} = \text{t}, \neg \text{s} \rightarrow \neg \text{f} = \text{f}$
$(\text{s} \rightarrow \text{f}) \rightarrow (\neg \text{f} \rightarrow \neg \text{s})$	{ valid }	
$\text{b} \vee \text{d} \vee (\text{b} \rightarrow \text{d})$	{ valid }	
$\text{b} \vee \text{d} \vee \neg \text{b} \vee \text{d}$	{ valid }	

6.034 - Spring 03 • 27

Satisfiability

- Related to constraint satisfaction
- Given a sentence S , try to find an interpretation i such that $\text{holds}(S, i)$
- Analogous to finding an assignment of values to variables such that the constraints hold



6.034 - Spring 03 • 28

Slide 9.2.28

The problem of deciding whether a sentence is satisfiable is related to constraint satisfaction: you have to find an interpretation i such that the sentence holds in that interpretation. That's analogous to finding an assignment of values to variables so that the constraints are satisfied.

Satisfiability
• Related to constraint satisfaction
• Given a sentence S , try to find an interpretation i such that $\text{holds}(S, i)$
• Analogous to finding an assignment of values to variables such that the constraints hold
• Brute force method: enumerate all interpretations and check

6.034 - Spring 03 • 29

Satisfiability

- Related to constraint satisfaction
- Given a sentence S , try to find an interpretation i such that $\text{holds}(S, i)$
- Analogous to finding an assignment of values to variables such that the constraints hold
- Brute force method: enumerate all interpretations and check
- Better methods:
 - heuristic search
 - constraint propagation
 - stochastic search



6.034 - Spring 03 • 30

Slide 9.2.30

Better would be to use methods from constraint satisfaction. There are a number of search algorithms that have been specially adapted to solving satisfiability problems as quickly as possible, using combinations of backtracking, constraint propagation, and variable ordering.

Slide 9.2.31

There are lots of satisfiability problems in the real world. They end up being expressed essentially as lists of boolean logic expressions, where you're trying to find some assignment of values to variables that makes the sentence true.

Satisfiability problems

6.034 - Spring 03 • 31

Satisfiability problems

- Scheduling nurses to work in a hospital
 - propositional variables represent, for example, that Pat is working on Tuesday at 2
 - constraints on the schedule are represented using logical expressions over the variables

6.034 - Spring 03 • 32

Slide 9.2.32

One example is scheduling nurses to work shifts in a hospital. Different people have different constraints, some don't want to work at night, no individual can work more than this many hours out of that many hours, these two people don't want to be on the same shift, you have to have at least this many per shift and so on. So you can often describe a setting like that as a bunch of constraints on a set of variables.

Satisfiability problems

- Scheduling nurses to work in a hospital
 - propositional variables represent, for example, that Pat is working on Tuesday at 2
 - constraints on the schedule are represented using logical expressions over the variables
- Finding bugs in software
 - propositional variables represent state of the program
 - use logic to describe how the program works and to assert there is a bug
 - if the sentence is satisfiable, you've found a bug!

6.034 - Spring 03 • 33

Slide 9.2.33

There's an interesting application of satisfiability that's going on here at MIT in the Lab for Computer Science. Professor Daniel Jackson's interested in trying to find bugs in programs. That's a good thing to do, but (as you know!) it's hard for humans to do reliably, so he wants to get the computer to do it automatically.

One way to do it is to essentially make a small example instance of a program. So an example of a kind of program that he might want to try to find a bug in would be an air traffic controller. The air traffic controller has rules that specify how it works. So you could write down the logical specification of how the air traffic control protocol works, and then you could write down another sentence that says, "and there are two airplanes on the same runway at the same time." And then you could see if there is a satisfying assignment; whether there is a configuration of airplanes and things that actually satisfies the specifications of the air traffic control protocol and also has two airplanes on the same runway at the same time. And if you can find one -- if that whole sentence is satisfiable, then you have a problem in your air traffic control protocol.

6.034 Notes: Section 9.3**Slide 9.3.1**

One reason for writing down logical descriptions of situations is that they will allow us to draw conclusions about other aspects of the situation we've described.

A Good Lecture?

6.034 - Spring 03 • 1

A Good Lecture?

Imagine we knew that:

- If today is sunny, then Tomas will be happy ($S \rightarrow H$)
- If Tomas is happy, the lecture will be good ($H \rightarrow G$)
- Today is sunny (S)

Should we conclude that the lecture will be good?



6.034 - Spring 03 • 2

Slide 9.3.2

Imagine that we knew the following things to be true: If today is sunny, Tomas will be happy; if Tomas is happy, the lecture will be good; and today is sunny.

Does this mean that the lecture will be good?

**Slide 9.3.3**

One way to think about this is to start by figuring out what set of interpretations make our original sentences true. Then, if G is true in all those interpretations, it must be okay to conclude it from the sentences we started out with (sometimes called our knowledge base).

Checking Interpretations

6.034 - Spring 03 • 3

Checking Interpretations**Slide 9.3.4**

In a universe with only three variables, there are 8 possible interpretations in total.

S	H	G
t	t	t
t	t	f
t	f	t
t	f	f
f	t	t
f	t	f
f	f	t
f	f	f

6.034 - Spring 03 • 4



Slide 9.3.5

Only one of these interpretations makes all the sentences in our knowledge base true: S = true, H = true, G = true.

Checking Interpretations

S	H	G	$S \rightarrow H$	$H \rightarrow G$	S
t	t	t	t	t	t
t	t	f	t	f	t
t	f	t	f	t	t
t	f	f	f	t	t
f	t	t	t	t	f
f	t	f	t	f	f
f	f	t	t	t	f
f	f	f	t	t	f

6.034 - Spring 03 • 5

**Checking Interpretations**

S	H	G	$S \rightarrow H$	$H \rightarrow G$	S	G
t	t	t	t	t	t	t
t	t	f	t	f	t	f
t	f	t	f	t	t	t
t	f	f	f	t	t	f
f	t	t	t	f	t	t
f	t	f	t	f	f	f
f	f	t	t	f	t	t
f	f	f	t	f	f	f



6.034 - Spring 03 • 6

**Slide 9.3.6**

And it's easy enough to check that G is true in that interpretation, so it seems like it must be reasonable to draw the conclusion that the lecture will be good. (Good thing!).

Slide 9.3.7

If we added another variable to our domain, say whether Leslie is happy (L), then we'd have two interpretations that satisfy the KB: S = true, H = true, G = true, L = true; and S = true, H = true, G = true, L = false.

G is true in both of these interpretations, so, again, if the KB is true, then G must also be true.

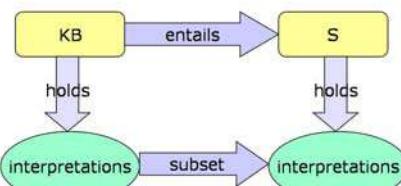
Adding a Variable

L	S	H	G	$S \rightarrow H$	$H \rightarrow G$	S	G
t	t	t	t	t	t	t	t
t	t	t	f	t	f	t	f
t	t	f	t	f	t	t	t
t	t	f	f	f	t	t	f
t	f	t	t	t	t	f	t
t	f	t	f	t	f	f	f
t	f	f	t	t	t	f	t
f	t	t	t	t	t	t	t
f	t	t	f	t	f	t	f
...					

6.034 - Spring 03 • 7

**Entailment**

A knowledge base (KB) **entails** a sentence S iff every interpretation that makes KB true also makes S true

**Slide 9.3.8**

There is a general idea called "entailment" that signifies a relationship between a knowledge base and another sentence. If whenever the KB is true, the conclusion has to be true (that is, if every interpretation that satisfies the KB also satisfies the conclusion), we'll say that the KB "entails" the conclusion. You can think of entailment as something like "follows from", or "it's okay to conclude from".



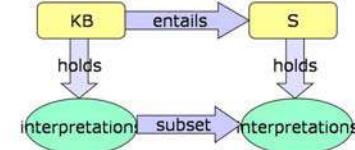
6.034 - Spring 03 • 8

Slide 9.3.9

The method of enumerating all the interpretations that satisfy the KB, and then checking to see if the conclusion is true in all of them is a correct way to test entailment.

Computing Entailment

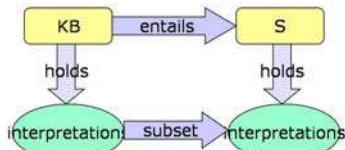
- enumerate all interpretations
- select those in which all elements of KB are true
- check to see if S is true in all of those interpretations



6.034 - Spring 03 • 9

Computing Entailment

- enumerate all interpretations
- select those in which all elements of KB are true
- check to see if S is true in all of those interpretations



Way too many interpretations, in general!!

6.034 - Spring 03 • 10

Slide 9.3.10

But now, what if we were to add 6 more propositional variables to our domain? Then we'd have $2^{10} = 1024$ interpretations to check, which is way too much work to do (and, in the first order case, we'll find that we might have infinitely many interpretations, which is definitely too much work to enumerate!!).

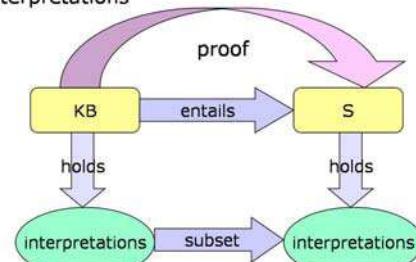
Slide 9.3.11

So what we'd really like is a way to figure out whether a KB entails a conclusion without enumerating all of the possible interpretations.

A proof is a way to test whether a KB entails a sentence, without enumerating all possible interpretations. You can think of it as a kind of shortcut arrow that works directly with the syntactic representations of the KB and the conclusion, without going into the semantic world of interpretations.

Entailment and Proof

A **proof** is a way to test whether a KB entails a sentence, without enumerating all possible interpretations



6.034 - Spring 03 • 11

Proof**Slide 9.3.12**

So what is a proof system? Well, presumably all of you have studied high-school geometry; that's often people's only exposure to formal proof. Remember that? You knew some things about the sides and angles of two triangles and then you applied the side-angle-side theorem to conclude -- at least people in American high schools were familiar with side-angle-side -- The side-angle-side theorem allowed you to conclude that the two triangles were similar, right?

That is formal proof. You've got some set of rules that you can apply. You've got some things written down on your page, and you grind through, applying the rules that you have to the things that are written down, to write some more stuff down until finally you've written down the things that you wanted to, and then you get to declare victory. That's a proof. There are (at least) two styles of proof system; we're going to talk about one briefly here and then go on to the other one at some length in the next two sections.

Natural deduction refers to a set of proof systems that are very similar to the kind of system you used in high-school geometry. We'll talk a little bit about natural deduction just to give you a flavor of how it goes in propositional logic, but it's going to turn out that it's not very good as a general strategy for computers. It's a proof system that humans like, and then we'll talk about a proof system that computers like, to the extent that computers can like anything.

6.034 - Spring 03 • 12

Slide 9.3.13

A proof is a sequence of sentences. This is going to be true in almost all proof systems.

Proof

- Proof is a sequence of sentences

6.034 - Spring 03 • 13

Proof

- Proof is a sequence of sentences
- First ones are premises (KB)

6.034 - Spring 03 • 14

Slide 9.3.14

First we'll list the premises. These are the sentences in your knowledge base. The things that you know to start out with. You're allowed to write those down on your page. Sometimes they're called the "givens." You can put the givens down.

Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines

6.034 - Spring 03 • 15

Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines
- When S is on a line, you have proved S from KB

6.034 - Spring 03 • 16

Slide 9.3.16

Then, when a sentence S is on some line, you have proved S from KB.

Slide 9.3.17

If your inference rules are **sound**, then any S you can prove from KB is, in fact, entailed by KB. That is, it's legitimate to draw the conclusion S from the assumptions in KB.

Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines
- When S is on a line, you have proved S from KB
- If inference rules are **sound**, then any S you can prove from KB is entailed by KB

6.034 - Spring 03 • 17

Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines
- When S is on a line, you have proved S from KB
- If inference rules are **sound**, then any S you can prove from KB is entailed by KB
- If inference rules are **complete**, then any S that is entailed by KB can be proved from KB

6.034 - Spring 03 • 18

Slide 9.3.18

If your rules are **complete**, then you can use KB to prove any S that is entailed by the KB. That is, you can prove any legitimate conclusion.

Wouldn't it be great if you were sound and complete derivers of answers to problems? You'd always get an answer and it would always be right!

Slide 9.3.19

So let's look at inference rules, and learn how they work by example. We'll look at natural-deduction rules first, because they're easiest to understand.

Natural Deduction

Some inference rules:

6.034 - Spring 03 • 19

Natural Deduction

Some inference rules:

$$\frac{\alpha \rightarrow \beta \\ \alpha}{\beta}$$

Modus
ponens

Slide 9.3.20

Here's a famous one (first written down by Aristotle); it has the great Latin name, "modus ponens", which means "affirming method".

It says that if you have "Alpha implies Beta" written down somewhere on your page, and you have Alpha written down somewhere on your page, then you can write beta down on a new line. (Alpha and Beta here are metavariables, like Phi and Psi, ranging over whole complicated sentences). It's important to remember that inference rules are just about ink on paper, or bits on your computer screen. They're not about anything in the world. Proof is just about writing stuff on a page, just syntax. But if you're careful in your proof rules and they're all sound, then at the end when you have some bit of syntax written down on your page, you can go back via the interpretation to some semantics. So you start out by writing down some facts about the world formally as your knowledge base. You do stuff with ink and paper for a while and now you have some other symbols written down on your page. You can go look them up in the world and say, "Oh, I see. That's what they mean."

6.034 - Spring 03 • 20

Slide 9.3.21

Here's another inference rule. "Modus tollens" (denying method) says that, from "alpha implies beta" and "not beta" you can conclude "not alpha".

Natural Deduction

Some inference rules:

$$\frac{\begin{array}{c} \alpha \rightarrow \beta \\ \alpha \\ \hline \beta \end{array}}{\text{Modus ponens}}$$

$$\frac{\begin{array}{c} \alpha \rightarrow \beta \\ \neg \beta \\ \hline \neg \alpha \end{array}}{\text{Modus tolens}}$$

6.034 - Spring 03 • 21

Natural Deduction

Some inference rules:

$$\frac{\alpha \rightarrow \beta}{\text{Modus ponens}}$$

$$\frac{\alpha \rightarrow \beta}{\text{Modus tolens}}$$

$$\frac{\begin{array}{c} \alpha \\ \beta \\ \hline \alpha \wedge \beta \end{array}}{\text{And-introduction}}$$

6.034 - Spring 03 • 22

Slide 9.3.22

And-introduction say that from "Alpha" and from "Beta" you can conclude "Alpha and Beta". That seems pretty obvious.

Natural Deduction

Some inference rules:

$$\frac{\alpha \rightarrow \beta}{\text{Modus ponens}}$$

$$\frac{\alpha \rightarrow \beta}{\text{Modus tolens}}$$

$$\frac{\begin{array}{c} \alpha \\ \beta \\ \hline \alpha \wedge \beta \end{array}}{\text{And-introduction}}$$

$$\frac{\alpha \wedge \beta}{\alpha}$$

$$\frac{\alpha \wedge \beta}{\alpha}$$

$$\frac{\alpha \wedge \beta}{\text{And-elimination}}$$

6.034 - Spring 03 • 23

Natural deduction example

Prove S

Step	Formula	Derivation

6.034 - Spring 03 • 24

Slide 9.3.24

Now let's do a sample proof just to get the idea of how it works. Pretend you're back in high school

Slide 9.3.25

We'll start with 3 sentences in our knowledge base, and we'll write them on the first three lines of our proof: (P and Q), (P implies R), and (Q and R imply S).

Natural deduction example

Prove S

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given

6.034 - Spring 03 • 25

**Natural deduction example**

Prove S

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given
4	P	1 And-Elim

6.034 - Spring 03 • 26

**Slide 9.3.26**

From line 1, using the and-elimination rule, we can conclude P, and write it down on line 4 (together with a reminder of how we derived it).

Natural deduction example

Prove S

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens

6.034 - Spring 03 • 27

**Natural deduction example**

Prove S

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim

6.034 - Spring 03 • 28

**Slide 9.3.28**

From line 1, we can use and-elimination to get Q.

Slide 9.3.29

From lines 5 and 6, we can use and-introduction to get (Q and R).

Natural deduction example

Prove S

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim
7	$Q \wedge R$	5,6 And-Intro

6.034 - Spring 03 • 29

Natural deduction example

Prove S

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim
7	$Q \wedge R$	5,6 And-Intro
8	S	7,3 Modus Ponens

6.034 - Spring 03 • 30

Slide 9.3.30

Finally, from lines 7 and 3, we can use modus ponens to get S. Whew! We did it!

Slide 9.3.31

The process of formal proof seems pretty mechanical. So why can't computers do it?

They can. For natural deduction systems, there are a lot of "proof checkers", in which you tell the system what conclusion it should try to draw from what premises. They're always sound, but nowhere near complete. You typically have to ask them to do the proof in baby steps, if you're trying to prove anything at all interesting.

Proof systems

- There are many natural deduction systems; they are typically "proof checkers", sound but not complete

6.034 - Spring 03 • 31

Proof systems

- There are many natural deduction systems; they are typically "proof checkers", sound but not complete
- Natural deduction uses lots of inference rules which introduces a large branching factor in the search for a proof.

Slide 9.3.32

Part of the problem is that they have a lot of inference rules, which introduces a very big branching factor in the search for proofs.

6.034 - Spring 03 • 32

Slide 9.3.33

Another big problem is the need to do "proof by cases". What if you wanted to prove R from (P or Q), (Q implies R), and (P implies R)? You have to do it by first assuming that P is true and proving R, then assuming Q is true and proving R. And then finally applying a rule that allows you to conclude that R follows no matter what. This kind of proof by cases introduces another large amount of branching in the space.

Proof systems

- There are many natural deduction systems; they are typically "proof checkers", sound but not complete
- Natural deduction uses lots of inference rules which introduces a large branching factor in the search for a proof.
- In general, you need to do "proof by cases" which introduces even more branching.

Prove R

1	P ∨ Q
2	Q → R
3	P → R

6.034 - Spring 03 • 33

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta}{\alpha \vee \gamma}$$
- Single inference rule is a sound and complete proof system
- Requires all sentences to be converted to conjunctive normal form

6.034 - Spring 03 • 34

Slide 9.3.34

An alternative is resolution, a single inference rule that is sound and complete, all by itself. It's not very intuitive for humans to use, but it's great for computers.

Resolution requires all sentences to be first written in a special form. So the next section will investigate that special form, and then we'll return to resolution.

Slide 9.4.1

Now we're going to start talking about first-order logic, which extends propositional logic so that we can talk about *things*.

First-Order Logic

6.034 - Spring 03 • 1

6.034 Notes: Section 9.4

First-Order Logic

- Propositional logic only deals with "facts", statements that may or may not be true of the world, e.g., "It is raining". But, one cannot have variables that stand for books or tables.



6.034 - Spring 03 • 2

Slide 9.4.2

In propositional logic, all we had were variables that stood, not for things in the world or even quantities, but just facts, Boolean statements that might or might not be true about the world, like whether it's raining, or greater than 67 degrees; but you couldn't have variables that stood for tables or books, or the temperature, or anything like that. And as it turns out, that's an enormously limiting kind of representation.

Slide 9.4.3

In first-order logic, variables refer to things in the world and you can quantify over them. That is, you can talk about all or some of them without having to name them explicitly.

First-Order Logic

- Propositional logic only deals with "facts", statements that may or may not be true of the world, e.g., "It is raining". But, one cannot have variables that stand for books or tables.
- In **first-order logic**, variables refer to things in the world and, furthermore, you can **quantify** over them: talk about all of them or some of them without having to name them explicitly.

6.034 - Spring 03 • 3

FOL motivation

- Statements that cannot be made in propositional logic but can be made in FOL



6.034 - Spring 03 • 4

Slide 9.4.4

There are lots of examples that show how propositional logic is inadequate to characterize even moderately complex domains. Here are some more examples of the kinds of things that you can say in first-order logic, but not in propositional logic.

Slide 9.4.5

"When you paint the block, it becomes green." You might have a proposition for every single aspect of the situation, like "if this block is black and I paint it, it becomes green" and "if that block is red and I paint it, it becomes green" and "if block #5 is green and I paint it, it becomes green". But you'd have to have one of those propositions for every single initial block color, or every single block, or every single object (if you have non-blocks, too) in the world. You couldn't say that, as a general fact, "after you paint something it becomes green."

FOL motivation

Statements that cannot be made in propositional logic but can be made in FOL

- When you paint a block with green paint, it becomes green.
 - In propositional logic, one would need a statement about every single block, one cannot make the general statement about all blocks.

6.034 - Spring 03 • 5

FOL motivation

Statements that cannot be made in propositional logic but can be made in FOL

- When you paint a block with green paint, it becomes green.
 - In propositional logic, one would need a statement about every single block, one cannot make the general statement about all blocks.
- When you sterilize a jar, all the bacteria are dead.
 - In FOL, we can talk about all the bacteria without naming them explicitly.



6.034 - Spring 03 * 6

Slide 9.4.6

Let's say you want to talk about what happens when you sterilize a jar. It kills all the bacteria in the jar. Now, you don't want to have to name all the bacteria; to have to say, bacterium 57 is dead, and bacterium 93 is dead. Each one of those guys is dead. All the bacteria are dead now. So you'd like to have a way not only to talk about things in the world, but to talk about all of them, or some of them, without naming any of them explicitly.

Slide 9.4.7

In the context of providing flexible computer security, you might want to prove or try to understand whether someone should be allowed access to a web site. And you could say: a person should have access to this web site if they've been personally, formally authorized to use this web site or if they are known to someone who has access to the web site. So you could write a general rule that says that and then some other system or this system could try to prove that you should have access to the web site. In this case, what that would mean would be going to look for a chain of people that are authorized or known to one another that bottoms out in somebody who's known to this web site.

FOL motivation

Statements that cannot be made in propositional logic but can be made in FOL

- When you paint a block with green paint, it becomes green.
 - In propositional logic, one would need a statement about every single block, one cannot make the general statement about all blocks.
- When you sterilize a jar, all the bacteria are dead.
 - In FOL, we can talk about all the bacteria without naming them explicitly.
- A person is allowed access to this Web site if they have been formally authorized or they are known to someone who has access.

6.034 - Spring 03 * 7

FOL syntax



6.034 - Spring 03 * 8

Slide 9.4.8

First-order logic lets us talk about things in the world. It's a logic like propositional logic, but somewhat richer and more complex. We'll go through the material in the same way that we did propositional logic: we'll start with syntax and semantics, and then do some practice with writing down statements in first-order logic.

Slide 9.4.9

The big difference between propositional logic and first-order logic is that we can talk about things, and so there's a new kind of syntactic element called a *term*. And the term, as we'll see when we do the semantics, is a name for a thing. It's an expression that somehow names a thing in the world. There are three kinds of terms:

FOL syntax

- Term

6.034 - Spring 03 * 9

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39

Slide 9.4.10

There are constant symbols. They are names like **Fred** or **Japan** or **Bacterium39**. Those are symbols that, in the context of an interpretation, name a particular thing.

6.034 - Spring 03 • 10

Slide 9.4.11

Then there are variables, which are not really syntactically differentiated from constant symbols. We'll use capital letters to start constant symbols (think of them as proper names), and lower-case letters for term variables. (It's important to note, though, that this convention is not standard, and in some logic contexts, such as the programming language Prolog, they adopt the exact opposite convention).

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39
 - Variables: x, y, a

6.034 - Spring 03 • 11

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39
 - Variables: x, y, a
 - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)

Slide 9.4.12

The last kind of term is a function symbol, applied to one or more terms. We'll use lower-case for function symbols as well. So another way to make a name for something is to say something like **f(x)**. If **f** is a function, you can give it a term and then **f(x)** names something. So, you might have **mother-of(John)** or **f(f(x))**. Note that a function with no terms would be a constant.

These three kinds of terms are our ways to name things in the world.

6.034 - Spring 03 • 12

Slide 9.4.13

In propositional logic we had sentences. Now, in first-order logic it's a little bit more complicated, but not a lot. So what's a sentence? There's another kind of symbol called a predicate symbol. A predicate symbol is applied to zero or more terms. Predicate symbols stand for relations, so we might have things like **On(A,B)** or **Sister(Jane, Joan)**. **On** and **Sister** are predicate symbols; **a**, **b**, **Jane**, **Joan**, and **mother-of(John)** are terms.

A predicate applied to zero terms is what's sometimes called a sentential variable or a propositional variable. It was our old kind of variable that we had before in propositional logic, like "it's-raining." It's a little bit of an artifice, but we'll take predicates with no arguments to be variables that have values true or false.

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39
 - Variables: x, y, a
 - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)
- Sentence
 - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)

6.034 - Spring 03 • 13

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39
 - Variables: x, y, a
 - Function symbol applied to one or more terms: $f(x)$, $f(f(x))$, mother-of(John)
- Sentence
 - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)
 - $t_1 = t_2$

6.034 - Spring 03 • 14

Slide 9.4.14

A sentence can also be of the form $t_1 = t_2$. We're going to have one special predicate called equality. You can say this thing equals that thing, written term, equal-sign, term.

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39
 - Variables: x, y, a
 - Function symbol applied to one or more terms: $f(x)$, $f(f(x))$, mother-of(John)
- Sentence
 - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)
 - $t_1 = t_2$
 - If v is a variable and Φ is a sentence, then $\forall v.\Phi$ and $\exists v.\Phi$ are sentences.

6.034 - Spring 03 • 15

FOL syntax

- Term
 - Constant symbols: Fred, Japan, Bacterium39
 - Variables: x, y, a
 - Function symbol applied to one or more terms: $f(x)$, $f(f(x))$, mother-of(John)
- Sentence
 - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)
 - $t_1 = t_2$
 - If v is a variable and Φ is a sentence, then $\forall v.\Phi$ and $\exists v.\Phi$ are sentences.
 - Closure under sentential operators: $\wedge \vee \neg \rightarrow \leftrightarrow ()$

6.034 - Spring 03 • 16

Slide 9.4.16

Finally we have closure under the sentential operators that we had before, so you can make complex sentences out of other sentences using and, or, not, implies, equivalence (also called biconditional), and parentheses, just as before in propositional logic. All that basic connective structure is still the same, but the things that we can say on either side have gotten a little bit more complicated.

All right, that's our syntax. That's what we get to write down on our page.

6.034 Notes: Section 9.5

Slide 9.5.1

We're going to do the semantics informally. This isn't really going to look informal to you, but compared to the sorts of things that logicians write down, it's pretty informal. In propositional logic, an interpretation (**I**) is an assignment of truth values to sentential variables. Now an interpretation's going to be something more complicated. An interpretation is made up of a set and three mappings.

FOL Interpretations

- Interpretation I

6.034 - Spring 03 * 1

**FOL Interpretations**

- Interpretation I
 - **U** set of objects
(called "domain of discourse" or "universe")

6.034 - Spring 03 * 2

Slide 9.5.2

The set is the universe, **U**, which is a set of objects. So what's an object? Well, really, it could be this chair and that chair and these pieces of chalk or it could be all of you guys or it could be some trees out there, or it could be rather more abstract objects like meetings or points in time or numbers. An object could be anything you can think of, and the universe can be any set (finite or infinite) of objects. The universe is also sometimes called the "domain of discourse."

Slide 9.5.2

The set is the universe, **U**, which is a set of objects. So what's an object? Well, really, it could be this chair and that chair and these pieces of chalk or it could be all of you guys or it could be some trees out there, or it could be rather more abstract objects like meetings or points in time or numbers. An object could be anything you can think of, and the universe can be any set (finite or infinite) of objects. The universe is also sometimes called the "domain of discourse."

FOL Interpretations

- Interpretation I
 - **U** set of objects
(called "domain of discourse" or "universe")
 - Maps constant symbols to elements of **U**

6.034 - Spring 03 * 3

FOL Interpretations

- Interpretation I
 - **U** set of objects
(called "domain of discourse" or "universe")
 - Maps constant symbols to elements of **U**
 - Maps predicate symbols to relations on **U**
(binary relation is a set of pairs)

6.034 - Spring 03 * 4

Slide 9.5.4

The next mapping is from predicate symbols to relations on **U**. An n-ary relation is a set of lists of n objects, saying which groups of things stand in that particular relation to one another. A binary relation is a set of pairs. So if I have a binary relation **brother-of** and **U** is a bunch of people, then the relation would be the set of all pairs of people such that the second is the brother of the first.

Slide 9.5.5

The last mapping is from function symbols to functions on U . Functions are a special kind of relation, in which, for any particular assignment of the first $n-1$ elements in each list, there is a single possible assignment of the last one. In the **brother-of** relation, there could be many pairs with the same first item and a different second item, but in a function, if you have the same first item then you have to have the same second item. So that means you just name the first item and then there's a unique thing that you get from applying the function. So it's OK for **mother-of** to be a function, discounting adoptions and other unusual situations. We will also, for now, assume that our functions are *total*, which means that there is an entry for every possible assignment of the first $n-1$ elements.

So, the last mapping is from function symbols to functions on the universe.

FOL Interpretations

• Interpretation I

- **U** set of objects
(called "domain of discourse" or "universe")
- Maps constant symbols to elements of U
- Maps predicate symbols to relations on U
(binary relation is a set of pairs)
- Maps function symbols to functions on U
(function is a binary relation with a single pair for each element in U , whose first item is that element)

6.034 - Spring 03 * 5

Denotation of Terms

Terms name objects in U

6.034 - Spring 03 * 6

Slide 9.5.6

Before we can do the part of semantics that says what sentences are true in which interpretation, we have to talk about what terms mean. Terms name things, but we like to be fancy so we say a term denotes something, so we can talk about the denotation of a term, that is, the thing that a term names.

Slide 9.5.7

The denotations of constant symbols are given directly in the interpretation.

Denotation of Terms

Terms name objects in U

- $I(Fred)$ if Fred is constant, then given

6.034 - Spring 03 * 7

Denotation of Terms

Terms name objects in U

- $I(Fred)$ if Fred is constant, then given
- $I(x)$ if x is a variable, then undefined

6.034 - Spring 03 * 8

Slide 9.5.8

The denotation of a variable is undefined. What does x mean, if x is a variable? The answer is, "mu." It doesn't mean anything. That's a Zen joke. If you don't get it, don't worry about it.

Slide 9.5.9

The denotation of a complex term is defined recursively. So, to find the interpretation of a function symbol applied to some terms, first you look up the function symbol in the interpretation and get a function. (Remember that the function symbol is a syntactic thing, ink on paper, but the function it denotes is an abstract mathematical object.) Then you find the interpretations of the component terms, which will be objects in U . Finally, you apply the function to the objects, yielding an object in U . And that object is the denotation of the complex term.

Denotation of Terms

Terms name objects in U

- $I(Fred)$ if Fred is constant, then given
- $I(x)$ if x is a variable, then undefined
- $I(f(t_1, \dots, t_n)) = I(f)(I(t_1), \dots, I(t_n))$

6.034 - Spring 03 • 9

Holds

When does a sentence hold in an interpretation?

Slide 9.5.10

In the context of propositional logic, we looked at the rules of semantics, which told us how to determine whether a sentence was true in an interpretation. Now, in first-order logic, we'll add some semantic rules, for the new kinds of sentences we've introduced. One of our new kinds of sentences is a predicate symbol applied to a bunch of terms. That's a sentence, which is going to have a truth value, true or false.

6.034 - Spring 03 • 10

Slide 9.5.11

To figure out its truth value, we first use the denotation rules to find out which objects are named by each of the terms. Then, we look up the predicate symbol in the interpretation, which gives us a mathematical relation on U . Finally, we look to see if the list of objects named by the terms is a member of the relation. If so, the sentence is true in the given interpretation.

Holds

When does a sentence hold in an interpretation?

- P is a relation symbol
- t_1, \dots, t_n are terms

$\text{holds}(P(t_1, \dots, t_n), I) \text{ iff } <I(t_1), \dots, I(t_n)> \in I(P)$

6.034 - Spring 03 • 11

Holds

When does a sentence hold in an interpretation?

- P is a relation symbol
- t_1, \dots, t_n are terms

$\text{holds}(P(t_1, \dots, t_n), I) \text{ iff } <I(t_1), \dots, I(t_n)> \in I(P)$

Brother(Jon, Joe)??

Slide 9.5.12

Let's look at an example. Imagine we want to determine whether the sentence **Brother(Jon,Joe)** is true in some interpretation.

6.034 - Spring 03 • 12

Slide 9.5.13

First, we look up the constant symbol **Jon** in the interpretation and find that it names this guy with glasses.

Holds

When does a sentence hold in an interpretation?

- P is a relation symbol
- t_1, \dots, t_n are terms

$\text{holds}(P(t_1, \dots, t_n), I)$ iff $\langle I(t_1), \dots, I(t_n) \rangle \in I(P)$

Brother(Jon, Joe)??

- $I(\text{Jon}) = \text{Jon}$ [an element of U]

Image by MIT OCW.

6.034 - Spring 03 • 13

Holds

When does a sentence hold in an interpretation?

- P is a relation symbol
- t_1, \dots, t_n are terms

$\text{holds}(P(t_1, \dots, t_n), I)$ iff $\langle I(t_1), \dots, I(t_n) \rangle \in I(P)$

Brother(Jon, Joe)??

- $I(\text{Jon}) = \text{Jon}$ [an element of U]
- $I(\text{Joe}) = \text{Joe}$ [an element of U]

Images by MIT OCW.

6.034 - Spring 03 • 14

Slide 9.5.14

Then we look up **Joe** and find that it names this angry-looking guy.

Slide 9.5.15

Now we look up the predicate symbol **Brother** and find that it denotes this complicated relation.

Holds

When does a sentence hold in an interpretation?

- P is a relation symbol
- t_1, \dots, t_n are terms

$\text{holds}(P(t_1, \dots, t_n), I)$ iff $\langle I(t_1), \dots, I(t_n) \rangle \in I(P)$

Brother(Jon, Joe)??

- $I(\text{Jon}) = \text{Jon}$ [an element of U]
- $I(\text{Joe}) = \text{Joe}$ [an element of U]
- $I(\text{Brother}) = \{\langle \text{Jon}, \text{Joe} \rangle, \langle \text{Joe}, \text{Jon} \rangle, \langle \dots, \dots \rangle, \dots\}$

Images by MIT OCW.

6.034 - Spring 03 • 15

Holds

When does a sentence hold in an interpretation?

- P is a relation symbol
- t_1, \dots, t_n are terms

$\text{holds}(P(t_1, \dots, t_n), I)$ iff $\langle I(t_1), \dots, I(t_n) \rangle \in I(P)$

Brother(Jon, Joe)??

- $I(\text{Jon}) = \text{Jon}$ [an element of U]
- $I(\text{Joe}) = \text{Joe}$ [an element of U]
- $I(\text{Brother}) = \{\langle \text{Jon}, \text{Joe} \rangle, \langle \text{Joe}, \text{Jon} \rangle, \langle \dots, \dots \rangle, \dots\}$

Images by MIT OCW.

6.034 - Spring 03 • 15

Slide 9.5.16

Finally, we look up the pair of the guy with glasses and the angry-looking guy, to see if they're in the relation. They are, so the sentence must be true in that interpretation. It's easy to think of lots of other interpretations in which it wouldn't be true (and lots of others in which it would).

Slide 9.5.17

Another new kind of sentence we introduced has the form **term₁ = term₂**. The semantics are pretty unsurprising: if the object denoted by **term₁** is the same as the object denoted by **term₂**, then the sentence holds.

Equality

$\text{holds}(t_1 = t_2, I)$ iff $I(t_1)$ is the same object as $I(t_2)$

6.034 - Spring 03 • 17

Equality

$\text{holds}(t_1 = t_2, I)$ iff $I(t_1)$ is the same object as $I(t_2)$

Jon = Jack ?

- $I(\text{Jon}) = \text{以人为例}$ [an element of U]
- $I(\text{Jack}) = \text{人为例}$ [an element of U]
- $\text{holds}(\text{Jon} = \text{Jack}, I)$

Images by MIT OCW.

6.034 - Spring 03 • 18

Slide 9.5.19

Now we have to figure out how to tell whether sentences with quantifiers in them are true.

Semantics of Quantifiers

6.034 - Spring 03 • 19

Semantics of Quantifiers

Extend an interpretation I to bind variable x to element $a \in U$: $I_{x/a}$

Slide 9.5.20

In order to talk about quantifiers we need the idea of extending an interpretation. We would like to be able to extend an interpretation to bind variable x to value a . We'll write that as I with x bound to a . Here, x is a variable and a is an object; an element of U . The idea is that, in order to understand whether a sentence that has variables in it is true or not, we have to make various temporary assignments to the variables and see what the truth value of the sentence is. Binding x to a is kind of like adding x as a constant symbol to I . It's kind of like temporarily binding a variable in a programming language.

6.034 - Spring 03 • 20

Slide 9.5.21

Now, how do we evaluate the truth under interpretation I , of the statement **for all x , Phi?** So how do we know if that's true? Well, it's true if and only if **Phi** is true if for every possible binding of variable x to thing in the world a . Okay? For every possible thing in the world that you could plug in for x , this statement's true. That's what it means to say **for all x , Phi**.

Semantics of Quantifiers

Extend an interpretation I to bind variable x to element $a \in U$: $I_{x/a}$

- $\text{holds}(\forall x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for all $a \in U$

6.034 - Spring 03 • 21

Semantics of Quantifiers

Extend an interpretation I to bind variable x to element $a \in U$: $I_{x/a}$

- $\text{holds}(\forall x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for all $a \in U$
- $\text{holds}(\exists x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for some $a \in U$

6.034 - Spring 03 • 22

Slide 9.5.22

Similarly, to say that **there exists x such that Phi**, it means that **Phi** has to be true for some a in U . That is to say, there has to be something in the world such that if we plug that in for x , then **Phi** becomes true.

Semantics of Quantifiers

Extend an interpretation I to bind variable x to element $a \in U$: $I_{x/a}$

- $\text{holds}(\forall x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for all $a \in U$
- $\text{holds}(\exists x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for some $a \in U$

Quantifier applies to formula to right until an enclosing right parenthesis:

6.034 - Spring 03 • 23

Semantics of Quantifiers

Extend an interpretation I to bind variable x to element $a \in U$: $I_{x/a}$

- $\text{holds}(\forall x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for all $a \in U$
- $\text{holds}(\exists x.\Phi, I)$ iff $\text{holds}(\Phi, I_{x/a})$ for some $a \in U$

Quantifier applies to formula to right until an enclosing right parenthesis:

$$(\forall x.P(x) \vee Q(x)) \wedge \exists x.R(x) \rightarrow Q(x)$$

6.034 - Spring 03 • 24

Slide 9.5.24

So in this example sentence, the **for all x** applies until the close paren after the first $Q(x)$; and the **exists x** applies to the end of the sentence.

Slide 9.5.25

All right, let's work on an example. Here's a picture of our world.

FOL Example Domain

The Real World

6.034 - Spring 03 • 25

FOL Example Domain

- $U = \{\square, \triangle, \circ, \text{oval}\}$

The Real World

6.034 - Spring 03 • 26

Slide 9.5.26

There are four things in our U . Here they are.

FOL Example Domain

- $U = \{\square, \triangle, \circ, \text{oval}\}$
- Constants: Fred

The Real World

6.034 - Spring 03 • 27

FOL Example Domain

- $U = \{\square, \triangle, \circ, \text{oval}\}$
- Constants: Fred
- Preds: Above², Circle¹, Oval¹, Square¹

The Real World

6.034 - Spring 03 • 28

Slide 9.5.28

We have four predicates: **Above**, **Circle**, **Oval**, **Square**. The numbers above them indicate their arity, or the number of arguments they take. Now these particular predicate names suggest a particular interpretation. The fact that I used this word, "circle", makes you guess that probably the interpretation of circle is going to be true for the red object. But of course it needn't be. The fact that those marks on the page are like an English word that we think means something about the shape of an object, that doesn't matter. The syntax is just some words that we write down on our page. But it helps us understand what's going on. It's just like using reasonable variable names in a program that you might write. When you call a variable "the number of times I've been through this loop," that doesn't mean that the computer knows what that means. It's the same thing here.

Slide 9.5.29

And we have one function symbol, called **hat**, that takes a single argument.

FOL Example Domain

- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat

The Real World

6.034 - Spring 03 • 29

FOL Example Domain

- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat
- $I(\text{Fred}) = \blacktriangle$

The Real World

6.034 - Spring 03 • 30

Slide 9.5.30

Now we can talk about a particular interpretation, **I**. We'll define **I** so that **I(Fred)** is the triangle.

FOL Example Domain

- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat
- $I(\text{Fred}) = \blacktriangle$
- $I(\text{Above}) = \{\langle \blacksquare, \blacktriangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$

The Real World

6.034 - Spring 03 • 31

FOL Example Domain

- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat
- $I(\text{Fred}) = \blacktriangle$
- $I(\text{Above}) = \{\langle \blacksquare, \blacktriangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$

The Real World

6.034 - Spring 03 • 32

Slide 9.5.32

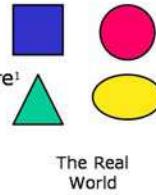
The interpretation of **Circle** is a unary relation. As you might expect in this world, it's the singleton set, whose element is a one-tuple containing the circle. (Of course, it doesn't have to be!).

Slide 9.5.33

We'll interpret the predicate **Oval** to be true of both the oval object and the round one (circles are a special case of ovals, after all).

FOL Example Domain

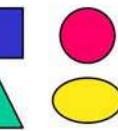
- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat
- $I(\text{Fred}) = \blacktriangle$
- $I(\text{Above}) = \{\langle \blacksquare, \blacktriangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$
- $I(\text{Oval}) = \{\langle \bullet \rangle, \langle \circlearrowleft \rangle\}$



6.034 - Spring 03 • 33

FOL Example Domain

- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat
- $I(\text{Fred}) = \blacktriangle$
- $I(\text{Above}) = \{\langle \blacksquare, \blacktriangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$
- $I(\text{Oval}) = \{\langle \bullet \rangle, \langle \circlearrowleft \rangle\}$
- $I(\text{hat}) = \{\langle \blacktriangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$



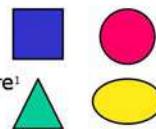
6.034 - Spring 03 • 34

Slide 9.5.34

And we'll say that the hat of the triangle is the square and the hat of the oval is the circle. If we stopped at this point, we would have a function, but it wouldn't be total (it wouldn't have an entry for every possible first argument). So, we'll make it total by saying that the square's hat is the square and the circle's hat is the circle.

FOL Example Domain

- $U = \{\blacksquare, \blacktriangle, \bullet, \circlearrowleft\}$
- Constants: Fred
- Preds: Above^2 , Circle^1 , Oval^1 , Square^1
- Function: hat
- $I(\text{Fred}) = \blacktriangle$
- $I(\text{Above}) = \{\langle \blacksquare, \blacktriangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$
- $I(\text{Oval}) = \{\langle \bullet \rangle, \langle \circlearrowleft \rangle\}$
- $I(\text{hat}) = \{\langle \blacktriangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
- $I(\text{Square}) = \{\langle \blacktriangle \rangle\}$



6.034 - Spring 03 • 35

FOL Example

- $I(\text{Fred}) = \blacktriangle$
- $I(\text{Above}) = \{\langle \blacksquare, \blacktriangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$
- $I(\text{Oval}) = \{\langle \bullet \rangle, \langle \circlearrowleft \rangle\}$
- $I(\text{hat}) = \{\langle \blacktriangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
- $I(\text{Square}) = \{\langle \blacktriangle \rangle\}$

6.034 - Spring 03 • 36

Slide 9.5.36

Now, let's find the truth values of some sentences in this interpretation. What about $\text{Square}(\text{Fred})$, is that true in this interpretation? Yes. We look to see that **Fred** denotes the triangle, and then we look for the triangle in the relation denoted by square, and we find it there. So the sentence is true.

Slide 9.5.37

What about this one? Is **Fred** above its hat?

FOL Example

- $I(Fred) = \triangle$
 - $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(Circle) = \{\langle \bullet \rangle\}$
 - $I(Oval) = \{\langle \bullet \rangle, \langle \circlearrowright \rangle\}$
 - $I(hat) = \{\langle \triangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
 - $I(Square) = \{\langle \triangle \rangle\}$
- holds(Square(Fred), I) ?
yes
 - holds(Above(Fred, hat(Fred)), I) ?

6.034 - Spring 03 • 37

FOL Example

- $I(Fred) = \triangle$
 - $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(Circle) = \{\langle \bullet \rangle\}$
 - $I(Oval) = \{\langle \bullet \rangle, \langle \circlearrowright \rangle\}$
 - $I(hat) = \{\langle \triangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
 - $I(Square) = \{\langle \triangle \rangle\}$
- holds(Square(Fred), I) ?
yes
 - holds(Above(Fred, hat(Fred)), I) ?
• $I(hat(Fred)) = \blacksquare$

6.034 - Spring 03 • 38

Slide 9.5.38

Let's start by asking the question, what's the denotation of the term, **hat(Fred)**?

It's the square, right? We look up **Fred**, and get the triangle. Then we look in the **hat** function, and, sure enough, there's a pair with triangle first and square second. So **hat(Fred)** is a square.

FOL Example

- $I(Fred) = \triangle$
 - $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(Circle) = \{\langle \bullet \rangle\}$
 - $I(Oval) = \{\langle \bullet \rangle, \langle \circlearrowright \rangle\}$
 - $I(hat) = \{\langle \triangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
 - $I(Square) = \{\langle \triangle \rangle\}$
- holds(Square(Fred), I) ?
yes
 - holds(Above(Fred, hat(Fred)), I) ?
• $I(hat(Fred)) = \blacksquare$
• holds(Above(\triangle , \blacksquare), I) ?

6.034 - Spring 03 • 39

FOL Example

- $I(Fred) = \triangle$
 - $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(Circle) = \{\langle \bullet \rangle\}$
 - $I(Oval) = \{\langle \bullet \rangle, \langle \circlearrowright \rangle\}$
 - $I(hat) = \{\langle \triangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
 - $I(Square) = \{\langle \triangle \rangle\}$
- holds(Square(Fred), I) ?
yes
 - holds(Above(Fred, hat(Fred)), I) ? no
• $I(hat(Fred)) = \blacksquare$
• holds(Above(\triangle , \blacksquare), I) ? no

6.034 - Spring 03 • 40

Slide 9.5.40

And our original sentence is false.

Slide 9.5.41

Okay. What about this sentence: there exists an x such that $\text{Oval}(x)$. Is there a thing that is an oval? Yes. So how do we show that carefully?

FOL Example

- $I(\text{Fred}) = \triangle$
 - $I(\text{Above}) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(\text{Circle}) = \{<\bullet>\}$
 - $I(\text{Oval}) = \{<\bullet>, <\circlearrowright>\}$
 - $I(\text{hat}) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet> <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
 - $I(\text{Square}) = \{<\triangle>\}$
- $\text{holds}(\text{Square}(\text{Fred}), I) ?$ yes
 - $\text{holds}(\text{Above}(\text{Fred}, \text{hat}(\text{Fred})), I) ?$ no
 - $I(\text{hat}(\text{Fred})) = \blacksquare$
 - $\text{holds}(\text{Above}(\triangle, \blacksquare), I) ?$ no
 - $\text{holds}(\exists x. \text{Oval}(x), I) ?$

6.034 - Spring 03 • 41

FOL Example

- $I(\text{Fred}) = \triangle$
 - $I(\text{Above}) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(\text{Circle}) = \{<\bullet>\}$
 - $I(\text{Oval}) = \{<\bullet>, <\circlearrowright>\}$
 - $I(\text{hat}) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet> <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
 - $I(\text{Square}) = \{<\triangle>\}$
- $\text{holds}(\text{Square}(\text{Fred}), I) ?$ yes
 - $\text{holds}(\text{Above}(\text{Fred}, \text{hat}(\text{Fred})), I) ?$ no
 - $I(\text{hat}(\text{Fred})) = \blacksquare$
 - $\text{holds}(\text{Above}(\triangle, \blacksquare), I) ?$ no
 - $\text{holds}(\exists x. \text{Oval}(x), I) ?$ yes
 - $\text{holds}(\text{Oval}(x), I/x_\bullet) ?$ yes

6.034 - Spring 03 • 42

Slide 9.5.42

We say that there's an extension of this interpretation where we take x and substitute in for it, the circle. Temporarily, I say that $I(x)$ is a circle. And now I ask, in that new interpretation, is it true that $\text{Oval}(x)$. So I look up x and I get the circle. I look up Oval and I get the relation with the circle and the oval, and so the answer's yes.

FOL Example: Continued

- $I(\text{Fred}) = \triangle$
 - $I(\text{Above}) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(\text{Circle}) = \{<\bullet>\}$
 - $I(\text{Oval}) = \{<\bullet>, <\circlearrowright>\}$
 - $I(\text{hat}) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet> <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
 - $I(\text{Square}) = \{<\triangle>\}$
- $\text{holds}(\forall x. \exists y. \text{Above}(x, y) \vee \text{Above}(y, x), I) ?$

6.034 - Spring 03 • 43

FOL Example: Continued

- $I(\text{Fred}) = \triangle$
 - $I(\text{Above}) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
 - $I(\text{Circle}) = \{<\bullet>\}$
 - $I(\text{Oval}) = \{<\bullet>, <\circlearrowright>\}$
 - $I(\text{hat}) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet> <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
 - $I(\text{Square}) = \{<\triangle>\}$
- $\text{holds}(\forall x. \exists y. \text{Above}(x, y) \vee \text{Above}(y, x), I) ?$
 - $\text{holds}(\exists y. \text{Above}(x, y) \vee \text{Above}(y, x), I/x_\triangle) ?$

6.034 - Spring 03 • 44

Slide 9.5.44

We can tell whether this is true by going through every possible object in the universe and binding it to the variable x , and then seeing whether the rest of the sentence is true. So, for example, we might put in the triangle for x , just to start with.

Slide 9.5.45

Now, having made that binding, we have to ask whether the sentence "There exists a y such that either x is above y or y is above x " true in the new interpretation. Existentials are easier than universals; we just have to come up with one y that makes the sentence true. And we can; if we bind y to the square, then that makes **Above(y,x)** true, which makes the disjunction true. So, we've proved this existential statement is true.

FOL Example: Continued

- $I(Fred) = \triangle$
- $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
- $I(Circle) = \{<\bullet>\}$
- $I(Oval) = \{<\bullet>, <\circlearrowright>\}$
- $I(hat) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet>, <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
- $I(Square) = \{<\triangle>\}$

- $\text{holds}(\forall x. \exists y. \text{Above}(x,y) \vee \text{Above}(y,x), I) ? \text{yes}$
- $\text{holds}(\exists y. \text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle) ? \text{yes}$
 $\text{holds}(\text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle, y/\blacksquare) ? \text{yes}$

6.034 - Spring 03 • 45

FOL Example: Continued

- $I(Fred) = \triangle$
- $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
- $I(Circle) = \{<\bullet>\}$
- $I(Oval) = \{<\bullet>, <\circlearrowright>\}$
- $I(hat) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet>, <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
- $I(Square) = \{<\triangle>\}$

- $\text{holds}(\forall x. \exists y. \text{Above}(x,y) \vee \text{Above}(y,x), I) ? \text{yes}$
- $\text{holds}(\exists y. \text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle) ? \text{yes}$
 $\text{holds}(\text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle, y/\blacksquare) ? \text{yes}$
- verify for all other values of x

6.034 - Spring 03 • 46

Slide 9.5.46

If we can do that for every other binding of x , then the whole universal sentence is true. You can verify that it is, in fact, true, by finding the truth value of the sentence with the other objects substituted in for x .

Slide 9.5.47

Okay. Here's our last example in this domain. What about the sentence: "for all x , for all y , x is above y or y is above x "? Is it true in interpretation I ?

FOL Example: Continued

- $I(Fred) = \triangle$
- $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
- $I(Circle) = \{<\bullet>\}$
- $I(Oval) = \{<\bullet>, <\circlearrowright>\}$
- $I(hat) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet>, <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
- $I(Square) = \{<\triangle>\}$

- $\text{holds}(\forall x. \exists y. \text{Above}(x,y) \vee \text{Above}(y,x), I) ? \text{yes}$
- $\text{holds}(\exists y. \text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle) ? \text{yes}$
 $\text{holds}(\text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle, y/\blacksquare) ? \text{yes}$
- verify for all other values of x

6.034 - Spring 03 • 47

FOL Example: Continued

- $I(Fred) = \triangle$
- $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
- $I(Circle) = \{<\bullet>\}$
- $I(Oval) = \{<\bullet>, <\circlearrowright>\}$
- $I(hat) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet>, <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
- $I(Square) = \{<\triangle>\}$

- $\text{holds}(\forall x. \exists y. \text{Above}(x,y) \vee \text{Above}(y,x), I) ? \text{yes}$
- $\text{holds}(\exists y. \text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle) ? \text{yes}$
 $\text{holds}(\text{Above}(x,y) \vee \text{Above}(y,x), Ix/\triangle, y/\blacksquare) ? \text{yes}$
- verify for all other values of x

6.034 - Spring 03 • 48

Slide 9.5.48

If it's going to be true, then it has to be true for every possible instantiation of x and y to elements of U . So, what, in particular, about the case when x is the square and y is the circle?

Slide 9.5.49

We can't find either the pair (square, circle), or the pair (circle, square) in the above relation, so this statement isn't true.

And, therefore, neither is the universally quantified statement.

FOL Example: Continued

- $I(Fred) = \triangle$
- $I(Above) = \{<\blacksquare, \triangle>, <\bullet, \circlearrowleft>\}$
- $I(Circle) = \{<\bullet>\}$
- $I(Oval) = \{<\bullet>, <\circlearrowright>\}$
- $I(hat) = \{<\triangle, \blacksquare>, <\circlearrowleft, \bullet>, <\blacksquare, \blacksquare>, <\bullet, \bullet>\}$
- $I(Square) = \{<\triangle>\}$

- $\text{holds}(\forall x. \exists y. \text{Above}(x,y) \vee \text{Above}(y,x), I) ? \text{yes}$
 - $\text{holds}(\exists y. \text{Above}(x,y) \vee \text{Above}(y,x), I/x/\triangle) ? \text{yes}$
 $\text{holds}(\text{Above}(x,y) \vee \text{Above}(y,x), I/x/\triangle, y/\bullet) ? \text{yes}$
 - verify for all other values of x
- $\text{holds}(\forall x. \forall y. \text{Above}(x,y) \vee \text{Above}(y,x), I) ? \text{no}$
 - $\text{holds}(\text{Above}(x,y) \vee \text{Above}(y,x), I/x/\triangle, y/\bullet) ? \text{no}$

6.034 - Spring 03 • 49

6.034 Notes: Section 9.6**Slide 9.6.1**

Now we're going to see how first-order logic can be used to formalize a variety of real-world concepts and situations. In this batch of problems, you should try to think of the answer before you go on to see it.

Writing FOL

6.034 - Spring 03 • 1

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]

Slide 9.6.2

How would you use first-order logic to say "Cats are mammals"? (You can use a unary predicate **cat** and another unary predicate **mammal**).

6.034 - Spring 03 • 2

Slide 9.6.3

For all x, Cat(x) implies Mammal(x). This is saying that every individual in the cat relation is also in the mammal relation. Or that cats are a subset of mammals.

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$

6.034 - Spring 03 • 3

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]

6.034 - Spring 03 • 4

Slide 9.6.4

All right. Let's let **Jane** be a constant, **Tall** and **Surveyor** can be unary predicates. How can we say Jane is a tall surveyor?

Slide 9.6.5

Surveyor(Jane) and Tall(Jane).

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - $\text{Tall}(\text{Jane}) \wedge \text{Surveyor}(\text{Jane})$

6.034 - Spring 03 • 5

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - $\text{Tall}(\text{Jane}) \wedge \text{Surveyor}(\text{Jane})$
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow$

6.034 - Spring 03 • 6

Slide 9.6.6

A nephew is a sibling's son. **Nephew**, **Sibling**, and **Son** are all binary relations. I'll start you off and say **for all x and y, x is the nephew of y if and only if** something. In English, what relationship has to hold between x and y for x to be a nephew of y? There has to be another person z who is a sibling of y and x has to be the son of z.

Slide 9.6.7

So, the answer is, "for all x and y, x is the nephew of y if and only if there exists a z such that y is a sibling of z and x is a son of z."

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - Tall(Jane) \wedge Surveyor(Jane)
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow \exists z. [\text{Sibling}(y,z) \wedge \text{Son}(x,z)]]$

6.034 - Spring 03 * 7

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - Tall(Jane) \wedge Surveyor(Jane)
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow \exists z. [\text{Sibling}(y,z) \wedge \text{Son}(x,z)]]$
- A maternal grandmother is a mother's mother
 - [functions: mgm, mother-of]

6.034 - Spring 03 * 8

Slide 9.6.8

When you have relationships that are functional, like "mother of", and "maternal grandmother of", then you can write expressions using functions and equality. So, what's the logical way of saying that someone's maternal grandmother is their mother's mother? Use **mgm**, standing for maternal grandmother, and **mother-of**, each of which is a function of a single argument.

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - Tall(Jane) \wedge Surveyor(Jane)
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow \exists z. [\text{Sibling}(y,z) \wedge \text{Son}(x,z)]]$
- A maternal grandmother is a mother's mother
 - [functions: mgm, mother-of]
 - $\forall xy. x = \text{mgm}(y) \leftrightarrow \exists z. x = \text{mother-of}(z) \wedge z = \text{mother-of}(y)$

6.034 - Spring 03 * 9

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - Tall(Jane) \wedge Surveyor(Jane)
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow \exists z. [\text{Sibling}(y,z) \wedge \text{Son}(x,z)]]$
- A maternal grandmother is a mother's mother
 - [functions: mgm, mother-of]
 - $\forall xy. x = \text{mgm}(y) \leftrightarrow \exists z. x = \text{mother-of}(z) \wedge z = \text{mother-of}(y)$
- Everybody loves somebody [loves²]

6.034 - Spring 03 * 10

Slide 9.6.10

Using a binary predicate **Loves(x,y)**, how can you say that everybody loves somebody?

Slide 9.6.11

This one's fun, because there are really two answers. The usual answer is **for all x, there exists a y such that Loves(x,y)**. So, for each person, there is someone that they love. The loved one can be different for each lover. The other interpretation is that there is a particular person that everybody loves. How would we say that?

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - Tall(Jane) \wedge Surveyor(Jane)
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow \exists z. [\text{Sibling}(y,z) \wedge \text{Son}(x,z)]]$
- A maternal grandmother is a mother's mother [functions: mgm, mother-of]
 - $\forall xy. x=\text{mgm}(y) \leftrightarrow \exists z. x=\text{mother-of}(z) \wedge z=\text{mother-of}(y)$
- Everybody loves somebody [loves²]
 - $\forall x. \exists y. \text{Loves}(x,y)$

6.034 - Spring 03 • 11

Writing FOL

- Cats are mammals [Cat¹, Mammal¹]
 - $\forall x. \text{Cat}(x) \rightarrow \text{Mammal}(x)$
- Jane is a tall surveyor [Tall¹, Surveyor¹, Jane]
 - Tall(Jane) \wedge Surveyor(Jane)
- A nephew is a sibling's son [Nephew², Sibling², Son²]
 - $\forall xy. [\text{Nephew}(x,y) \leftrightarrow \exists z. [\text{Sibling}(y,z) \wedge \text{Son}(x,z)]]$
- A maternal grandmother is a mother's mother [functions: mgm, mother-of]
 - $\forall xy. x=\text{mgm}(y) \leftrightarrow \exists z. x=\text{mother-of}(z) \wedge z=\text{mother-of}(y)$
- Everybody loves somebody [loves²]
 - $\forall x. \exists y. \text{Loves}(x,y)$
 - $\exists y. \forall x. \text{Loves}(x,y)$

6.034 - Spring 03 • 12

Slide 9.6.12

There exists a y such that for all x, Loves(x,y). So, just by changing the order of the quantifiers, we get a very different meaning.

Writing More FOL

- Nobody loves Jane

6.034 - Spring 03 • 13

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$

6.034 - Spring 03 • 14

Slide 9.6.14

For all x, not Loves(x, Jane). So, for everybody, every single person, that person doesn't love Jane.

Slide 9.6.15

An equivalent thing to write is there does not exist an x such that $\text{Loves}(x, \text{Jane})$. This is a general transformation, if you have for all x not something, then it's the same as having not there exists an x something. It's like saying I can't find a single x such that x Loves Jane.

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$

6.034 - Spring 03 • 15

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father

6.034 - Spring 03 • 16

Slide 9.6.16

Everybody has a father.

Slide 9.6.17

For all x , exists y such that $\text{Father}(y, x)$

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$

6.034 - Spring 03 • 17

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother

6.034 - Spring 03 • 18

Slide 9.6.18

Everybody has a father and a mother.

Slide 9.6.19

For all x , exists y, z such that $\text{Father}(y,x)$ and $\text{Mother}(z,x)$

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists y, z. \text{Father}(y, x) \wedge \text{Mother}(z, x)$

6.034 - Spring 03 • 19

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists y, z. \text{Father}(y, x) \wedge \text{Mother}(z, x)$

6.034 - Spring 03 • 20

Slide 9.6.20

Now, you might ask whether y and z are necessarily different. The answer is, no, that's not enforced by the logic. For that matter, they could be the same as x . Now, if you use the typical definitions of father and mother, they won't be the same, but that's up to the interpretation.

Slide 9.6.21

Whoever has a father has a mother.

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists y, z. \text{Father}(y, x) \wedge \text{Mother}(z, x)$
- Whoever has a father, has a mother

6.034 - Spring 03 • 21

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists y, z. \text{Father}(y, x) \wedge \text{Mother}(z, x)$
- Whoever has a father, has a mother
 - $\forall x.$

6.034 - Spring 03 • 22

Slide 9.6.22

This is a general statement about objects of the kind, everything that has one property has another property. All right? So we'll talk about everything by starting with **forall x**.

Slide 9.6.23

Now, how do we describe x's that have a father? **Exists y such that Father(y,x)**.

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists yz. \text{Father}(y, x) \wedge \text{Mother}(z, x)$
- Whoever has a father, has a mother
 - $\forall x. [\exists y. \text{Father}(y, x)] \rightarrow [\exists y. \text{Mother}(y, x)]$

6.034 - Spring 03 • 23

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists yz. \text{Father}(y, x) \wedge \text{Mother}(z, x)$
- Whoever has a father, has a mother
 - $\forall x. [\exists y. \text{Father}(y, x)] \rightarrow [\exists y. \text{Mother}(y, x)]$

6.034 - Spring 03 • 24

Slide 9.6.24

And we can describe x's that have a mother by **exists y such that Mother (y,x)**.

Slide 9.6.25

Finally, we put these together using implication, just as we did with the "all cats are mammals" example. We want to say objects with a Father are a subset of the set of objects with a Mother (in this case, it will turn out that the sets are equal). So, we end up with "for all x, if there exists a y such that y is the father of x, then there exists a y such that y is the mother of x".

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists yz. \text{Father}(y, x) \wedge \text{Mother}(z, x)$
- Whoever has a father, has a mother
 - $\forall x. [\exists y. \text{Father}(y, x)] \rightarrow [\exists y. \text{Mother}(y, x)]$

6.034 - Spring 03 • 25

Writing More FOL

- Nobody loves Jane
 - $\forall x. \neg \text{Loves}(x, \text{Jane})$
 - $\neg \exists x. \text{Loves}(x, \text{Jane})$
- Everybody has a father
 - $\forall x. \exists y. \text{Father}(y, x)$
- Everybody has a father and a mother
 - $\forall x. \exists yz. \text{Father}(y, x) \wedge \text{Mother}(z, x)$
- Whoever has a father, has a mother
 - $\forall x. [\exists y. \text{Father}(y, x)] \rightarrow [\exists y. \text{Mother}(y, x)]$

6.034 - Spring 03 • 26

Slide 9.6.26

Note that the two variables named y have separate scopes, and are entirely unrelated to one another. You could rename either or both of them and the semantics of the sentence would remain the same. It's technically legal to have nested quantifiers over the same variable, and there are rules for figuring out what it means, but it's very confusing, so it's just better not to do it.

6.034 Notes: Section 9.7

Slide 9.7.1

Now that we understand something about first-order logic as a language, we'll talk about how we can use it to do things. As in propositional logic, the thing that we'll most often want to do with logical statements is to figure out what conclusions we can draw from a set of assumptions. In propositional logic, we had the notion of entailment: a **KB** entails a sentence if and only if the sentence is true in every interpretation that makes **KB** true.

Entailment in First-Order Logic

6.034 – Spring 03 • 1

Entailment in First-Order Logic

- **KB** entails **S**: for every interpretation **I**, if **KB** holds in **I**, then **S** holds in **I**

6.034 – Spring 03 • 2

Slide 9.7.2

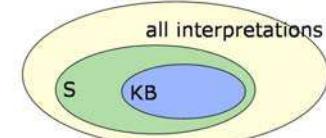
In first-order logic, the notion of entailment is the same. A knowledge base entails a sentence if and only if the sentence holds in every interpretation in which the knowledge base holds.

Slide 9.7.3

It's important that entailment is a relationship between a set of sentences, **KB**, and another sentence, **S**. It doesn't directly involve a particular intended interpretation that we might have in mind. It has to do with the subsets of all possible interpretations in which **KB** and **S** hold; entailment requires that the set of interpretations in which **KB** holds be a subset of those in which **S** holds. This is sort of a hard thing to understand at first, since the number (and potential weirdness) of all possible interpretations in first-order logic is just huge.

Entailment in First-Order Logic

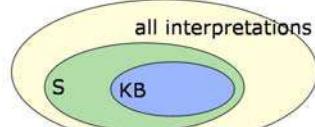
- **KB** entails **S**: for every interpretation **I**, if **KB** holds in **I**, then **S** holds in **I**



6.034 – Spring 03 • 3

Entailment in First-Order Logic

- KB entails S: for every interpretation I, if KB holds in I, then S holds in I



- Computing entailment is impossible in general, because there are infinitely many possible interpretations



6.034 - Spring 03 * 4

Slide 9.7.4

In propositional logic, we were able to think about computing entailment by doing a brute-force enumeration of all interpretations, then, in each interpretation, checking to see whether the sentence and/or the knowledge base were true in that interpretation.

This will be impossible in first-order logic for two reasons.

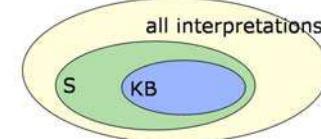
First, it's completely out of the question to enumerate all possible interpretations. How many universes are there? More than I can count...

Slide 9.7.5

Second, even for a single interpretation, it's not necessarily possible to compute whether a sentence holds in that interpretation. Why? Because if it has a universal or existential quantifier, it will require testing whether a sentence holds for every substitution of an element in U for the quantified variable. And if the universe is infinite, you just can't do that.

Entailment in First-Order Logic

- KB entails S: for every interpretation I, if KB holds in I, then S holds in I



- Computing entailment is impossible in general, because there are infinitely many possible interpretations
- Even computing holds is impossible for interpretations with infinite universes

6.034 - Spring 03 * 5

Intended Interpretations

$KB : (\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$



6.034 - Spring 03 * 6

Slide 9.7.6

Let's look at a particular situation in which we might want to do logical inference. Consider our shapes example from before. Let's say that we know, as our knowledge base, that all circles are ovals, and that no squares are ovals. We can write this as **for all x, Circle(x) implies Oval(x)**. And **for all x, Square(x) implies not Oval(x)**.

Slide 9.7.7

Now, let's say we're wondering whether it's also true that no squares are circles. We'll call that sentence **S**, and write it **for all x, Square(x) implies not Circle(x)**.

Intended Interpretations

$KB : (\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
 $S : \forall x. \text{Square}(x) \rightarrow \neg \text{Circle}(x)$



6.034 - Spring 03 * 7

Intended Interpretations

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

- We know holds(KB, I)
- We wonder whether holds(S, I)

- $I(\text{Fred}) = \triangle$
- $I(\text{Above}) = \{\langle \blacksquare, \triangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$
- $I(\text{Oval}) = \{\langle \bullet \rangle, \langle \circlearrowright \rangle\}$
- $I(\text{hat}) = \{\langle \triangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
- $I(\text{Square}) = \{\langle \triangle \rangle\}$



6.034 - Spring 03 • 8

Slide 9.7.8

We know **KB** holds in interpretation **I**, and we wonder whether **S** holds in **I**.

Slide 9.7.9

We could answer this by asking the question: Does **KB** entail **S**? Does our desired conclusion follow from our assumptions.

Intended Interpretations

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

- We know holds(KB, I)
- We wonder whether holds(S, I)
- We could ask: Does KB entail S?
- Or we could just try to check whether holds(S, I)

- $I(\text{Fred}) = \triangle$
- $I(\text{Above}) = \{\langle \blacksquare, \triangle \rangle, \langle \bullet, \circlearrowleft \rangle\}$
- $I(\text{Circle}) = \{\langle \bullet \rangle\}$
- $I(\text{Oval}) = \{\langle \bullet \rangle, \langle \circlearrowright \rangle\}$
- $I(\text{hat}) = \{\langle \triangle, \blacksquare \rangle, \langle \circlearrowleft, \bullet \rangle, \langle \blacksquare, \blacksquare \rangle, \langle \bullet, \bullet \rangle\}$
- $I(\text{Square}) = \{\langle \triangle \rangle\}$



6.034 - Spring 03 • 10

Slide 9.7.10

You might say that entailment is too big a hammer. I don't actually care whether **S** is true in all possible interpretations that satisfy **KB**. Why? because I have a particular interpretation in mind (namely, our little world of geometric shapes, embodied in interpretation **I**). And I know that **KB** holds in **I**. So what I really want to know is whether **S** holds in **I**.

Unfortunately, the computer does not know what interpretation I have in mind. We want the computer to be able to reach valid conclusions about my intended interpretation without my having to enumerate it (because it may be infinite).

For this particular example of **I**, it's not too hard to check whether **S** holds (because the universe is finite and small). But, as we said before, in general, we won't be able even to test whether a sentence holds in a particular interpretation.

An Infinite Interpretation

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$



6.034 - Spring 03 • 11

An Infinite Interpretation

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

$$U_1 = \{1, 2, 3, \dots\}$$



6.034 - Spring 03 • 12

Slide 9.7.12

The universe is the positive integers (numbers 1, 2, etc.). This universe is clearly infinite.

An Infinite Interpretation

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

$$U_1 = \{1, 2, 3, \dots\}$$

$$I_1(\text{circle}) = \{4, 8, 12, 16, \dots\}$$

6.034 - Spring 03 • 13

An Infinite Interpretation

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

$$U_1 = \{1, 2, 3, \dots\}$$

$$I_1(\text{circle}) = \{4, 8, 12, 16, \dots\}$$

$$I_1(\text{oval}) = \{2, 4, 6, 8, \dots\}$$



6.034 - Spring 03 • 14

Slide 9.7.14

We'll let **Oval** stand for the even positive integers, {2, 4, 6, 8, ...}.

An Infinite Interpretation

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
S : $\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

$$U_1 = \{1, 2, 3, \dots\}$$

$$I_1(\text{circle}) = \{4, 8, 12, 16, \dots\}$$

$$I_1(\text{oval}) = \{2, 4, 6, 8, \dots\}$$

$$I_1(\text{square}) = \{1, 3, 5, 7, \dots\}$$

6.034 - Spring 03 • 15

An Infinite Interpretation

$KB : (\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
 $S : \forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

- Does KB hold in I_1 ?

$U_1 = \{1, 2, 3, \dots\}$
 $I_1(\text{circle}) = \{4, 8, 12, 16, \dots\}$
 $I_1(\text{oval}) = \{2, 4, 6, 8, \dots\}$
 $I_1(\text{square}) = \{1, 3, 5, 7, \dots\}$

Slide 9.7.16

Now, does KB hold in I_1 ?

6.034 - Spring 03 • 16

Slide 9.7.17

We can't verify that by enumerating U and checking the sentences inside the universal quantifier. However, we all know, due to basic math knowledge, that it does.

An Infinite Interpretation

$KB : (\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
 $S : \forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

- Does KB hold in I_1 ?
- Yes, but can't answer via enumerating U

$U_1 = \{1, 2, 3, \dots\}$
 $I_1(\text{circle}) = \{4, 8, 12, 16, \dots\}$
 $I_1(\text{oval}) = \{2, 4, 6, 8, \dots\}$
 $I_1(\text{square}) = \{1, 3, 5, 7, \dots\}$

6.034 - Spring 03 • 17



An Infinite Interpretation

$KB : (\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
 $S : \forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x)$

- Does KB hold in I_1 ?
- Yes, but can't answer via enumerating U
- S also holds in I_1
- No way to verify mechanically

$U_1 = \{1, 2, 3, \dots\}$
 $I_1(\text{circle}) = \{4, 8, 12, 16, \dots\}$
 $I_1(\text{oval}) = \{2, 4, 6, 8, \dots\}$
 $I_1(\text{square}) = \{1, 3, 5, 7, \dots\}$

Slide 9.7.19

Let's think about a different S , which we'll call S_1 : For every circle and every oval that is not a circle, the circle is above the oval.

6.034 - Spring 03 • 18

An Argument for Entailment

$KB : (\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$
 $S_1 : \forall x, y. \text{Circle}(x) \wedge \text{Oval}(y) \wedge \neg \text{Circle}(y) \rightarrow \text{Above}(x, y)$

6.034 - Spring 03 • 19



An Argument for Entailment

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S₁ : $\forall x, y. \text{Circle}(x) \wedge \text{Oval}(y) \wedge \neg \text{Circle}(y) \rightarrow \text{Above}(x, y)$

- I(Fred) = ▲
- I(Above) = {<■, ▲>, <●, ○>}
- I(Circle) = {<●>}
- I(Oval) = {<●>, <○>}
- I(hat) = {<▲, ■>, <○, ●>,
 <■, ■>, <●, ●>}
- I(Square) = {<▲>}

- holds(KB, I)
- holds(S₁, I)

Slide 9.7.20

Back in I, our original geometric interpretation, this sentence holds, right?

But does it "follow from" KB? Is it entailed by KB?



6.034 - Spring 03 • 20

Slide 9.7.21

No. We can see this by going back to interpretation **I₁**, and letting the interpretation of the "above" relation be greater-than on integers.

An Argument for Entailment

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S₁ : $\forall x, y. \text{Circle}(x) \wedge \text{Oval}(y) \wedge \neg \text{Circle}(y) \rightarrow \text{Above}(x, y)$

- I(Fred) = ▲
- I(Above) = {<■, ▲>, <●, ○>}
- I(Circle) = {<●>}
- I(Oval) = {<●>, <○>}
- I(hat) = {<▲, ■>, <○, ●>,
 <■, ■>, <●, ●>}
- I(Square) = {<▲>}

- | |
|--|
| $U_1 = \{1, 2, 3, \dots\}$ |
| $I_1(\text{Circle}) = \{4, 8, 12, 16, \dots\}$ |
| $I_1(\text{Oval}) = \{2, 4, 6, 8, \dots\}$ |
| $I_1(\text{Square}) = \{1, 3, 5, 7, \dots\}$ |
| $I_1(\text{Above}) = >$ |

- holds(KB, I)
- holds(S₁, I)

- holds(KB, I₁)
- fails(S₁, I₁)

6.034 - Spring 03 • 22

Slide 9.7.22

An Argument for Entailment

Then S holds in I₁ if all integers divisible by 4 are greater than all integers divisible by 2 but not by 4, which is clearly false.

An Argument for Entailment

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S₁ : $\forall x, y. \text{Circle}(x) \wedge \text{Oval}(y) \wedge \neg \text{Circle}(y) \rightarrow \text{Above}(x, y)$

- I(Fred) = ▲
- I(Above) = {<■, ▲>, <●, ○>}
- I(Circle) = {<●>}
- I(Oval) = {<●>, <○>}
- I(hat) = {<▲, ■>, <○, ●>,
 <■, ■>, <●, ●>}
- I(Square) = {<▲>}

- | |
|--|
| $U_1 = \{1, 2, 3, \dots\}$ |
| $I_1(\text{Circle}) = \{4, 8, 12, 16, \dots\}$ |
| $I_1(\text{Oval}) = \{2, 4, 6, 8, \dots\}$ |
| $I_1(\text{Square}) = \{1, 3, 5, 7, \dots\}$ |
| $I_1(\text{Above}) = >$ |

- holds(KB, I)
- holds(S₁, I)

6.034 - Spring 03 • 21



Slide 9.7.23

So, although KB and S both hold in our original intended interpretation I, KB does not entail S, because we can find an interpretation in which KB holds but S does not.

An Argument for Entailment

KB : $(\forall x. \text{Circle}(x) \rightarrow \text{Oval}(x)) \wedge (\forall x. \text{Square}(x) \rightarrow \neg \text{Oval}(x))$

S₁ : $\forall x, y. \text{Circle}(x) \wedge \text{Oval}(y) \wedge \neg \text{Circle}(y) \rightarrow \text{Above}(x, y)$

- I(Fred) = ▲
- I(Above) = {<■, ▲>, <●, ○>}
- I(Circle) = {<●>}
- I(Oval) = {<●>, <○>}
- I(hat) = {<▲, ■>, <○, ●>,
 <■, ■>, <●, ●>}
- I(Square) = {<▲>}

- | |
|--|
| $U_1 = \{1, 2, 3, \dots\}$ |
| $I_1(\text{Circle}) = \{4, 8, 12, 16, \dots\}$ |
| $I_1(\text{Oval}) = \{2, 4, 6, 8, \dots\}$ |
| $I_1(\text{Square}) = \{1, 3, 5, 7, \dots\}$ |
| $I_1(\text{Above}) = >$ |

- holds(KB, I)
- holds(S₁, I)

- holds(KB, I₁)
- fails(S₁, I₁)

KB doesn't entail S₁!

6.034 - Spring 03 • 23



Proof and Entailment

- Entailment captures general notion of "follows from"



6.034 - Spring 03 • 24

Slide 9.7.24

We can see from this example that entailment captures this general notion of a sentence following from a set of assumptions; of being able to justify the truth of **S** based only on the truth of **KB**.

Proof and Entailment

- Entailment captures general notion of "follows from"
- Can't evaluate it directly by enumerating interpretations



6.034 - Spring 03 • 25

Proof and Entailment

- Entailment captures general notion of "follows from"
- Can't evaluate it directly by enumerating interpretations
- So, we'll do proofs



6.034 - Spring 03 • 26

Slide 9.7.26

So what do we do? As we did in propositional logic, we will stay in the domain of syntax, and do proofs to figure out whether **S** is entailed by **KB**.

Proof and Entailment

- Entailment captures general notion of "follows from"
- Can't evaluate it directly by enumerating interpretations
- So, we'll do proofs
- In FOL, if **S** is entailed by **KB**, then there is a finite proof of **S** from **KB**



6.034 - Spring 03 • 27

Axiomatization

- What if we have a particular interpretation, I , in mind, and want to test whether $\text{holds}(S, I)$?

Slide 9.7.28

We just argued that entailment is the right notion when we want to ask the question whether a sentence S follows from a **KB**. And that we're going to show entailment via proof.

But what if we have a particular interpretation in mind? We've seen that we can't in general, test whether a sentence holds in that interpretation. How can we use the ability to use proof to show entailment, in order to test whether a sentence holds in an interpretation?

6.034 - Spring 03 • 28

Slide 9.7.29

The answer is that we have to *axiomatize* our domain. That is, we have to write down a set of sentences, or axioms, which will serve as our **KB**.

Axiomatization

- What if we have a particular interpretation, I , in mind, and want to test whether $\text{holds}(S, I)$?
- Write down a set of sentences, called *axioms*, that will serve as our KB

6.034 - Spring 03 • 29

Axiomatization

- What if we have a particular interpretation, I , in mind, and want to test whether $\text{holds}(S, I)$?
- Write down a set of sentences, called *axioms*, that will serve as our KB
- We would like KB to hold in I , and as few other interpretations as possible

Slide 9.7.30

Ideally the axioms would be so specific that there was a single interpretation, our intended interpretation, in which they held. In general, though, this will be impossible. You might be able to constrain your axioms to describe domains that contain exactly 4 objects, but you'll never be able to say exactly which 4. You can often give axioms that put stringent enough requirements on the relationships between those objects that all of the interpretations in which the axioms hold are essentially the same as (isomorphic to) your intended interpretation.

6.034 - Spring 03 • 30

Slide 9.7.31

No matter how constraining your axioms are, you can rely on the fact that if your **KB** holds in your intended interpretation and **KB** entails S , then S holds in the intended interpretation.

Axiomatization

- What if we have a particular interpretation, I , in mind, and want to test whether $\text{holds}(S, I)$?
- Write down a set of sentences, called *axioms*, that will serve as our KB
- We would like KB to hold in I , and as few other interpretations as possible
- No matter what,
 - If $\text{holds}(\text{KB}, I)$ and **KB** entails S ,
 - then $\text{holds}(S, I)$

6.034 - Spring 03 • 31

Axiomatization

- What if we have a particular interpretation, I , in mind, and want to test whether $\text{holds}(S, I)$?
- Write down a set of sentences, called *axioms*, that will serve as our KB
- We would like KB to hold in I , and as few other interpretations as possible
- No matter what,
 - If $\text{holds}(\text{KB}, I)$ and KB entails S ,
 - then $\text{holds}(S, I)$
- If your axioms are weak, it might be that
 - $\text{holds}(\text{KB}, I)$ and $\text{holds}(S, I)$, but
 - KB doesn't entail S

6.034 - Spring 03 • 32

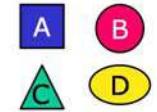
Slide 9.7.32

But that's only half of what we need. There might be some fact, S , about your intended interpretation that you would like to be able to derive from your axioms. But, if your axioms are not specific enough, then they might admit some interpretations in which S does not hold, and in that case, the axioms will not entail S , even though it might hold in the intended interpretation.



Slide 9.7.33

Let's work through an example of axiomatizing a domain. We'll think about our good old geometric domain, but, to simplify matters a bit, let's assume that we have the constant symbols **A**, **B**, **C**, and **D**. And let our interpretation specify that **A** is the square, **B** is the circle, **C** is the triangle, and **D** is the oval.



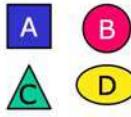
Axiomatization Example

6.034 - Spring 03 • 33



Axiomatization Example

$\text{Above}(A, C)$
 $\text{Above}(B, D)$

 KB_2 

6.034 - Spring 03 • 34

Slide 9.7.34

We propose to axiomatize this domain by specifying the above relation on these constants: **Above(A, C)** and **Above(B, D)**.



Slide 9.7.35

And we'll give some axioms that say how the hat function can be derived from **Above**: "for all x and y , if x is above y , then hat of y equals x ; and for all x , if there is no y such that y is above x , then hat of x equals x ".

Axiomatization Example

$\text{Above}(A, C)$
 $\text{Above}(B, D)$
 $\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$
 $\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$

 KB_2 

6.034 - Spring 03 • 35



Axiomatization Example

$\text{Above}(A, C)$ KB_2

$\text{Above}(B, D)$

$\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$

$\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$

S hat(A) = A

6.034 - Spring 03 • 36

Slide 9.7.36

These four axioms will constitute our **KB**. Now, we're curious to know whether it's okay to conclude that the hat of **A** is **A**. It's true in our intended interpretation, and we'd like it to be a consequence of our axioms.

Axiomatization Example

$\text{Above}(A, C)$ KB_2

$\text{Above}(B, D)$

$\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$

$\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$

S hat(A) = A

- $I_2(A) = \blacksquare$
- $I_2(B) = \bullet$
- $I_2(C) = \blacktriangle$
- $I_2(D) = \blacklozenge$
- $I_2(\text{Above}) = \{<\blacksquare, \blacktriangle>, <\bullet, \blacklozenge>, <\blacktriangle, \blacksquare>, <\bullet, \blacklozenge>\}$
- $I_2(\text{hat}) = \{<\blacktriangle, \blacksquare>, <\blacklozenge, \bullet>, <\bullet, \blacklozenge>, <\blacksquare, \blacktriangle>\}$

6.034 - Spring 03 • 37

Slide 9.7.37

So, does our **KB** entail **S**? Unfortunately not. Consider the interpretation **I**₂. It has two extra pairs in the interpretation of **Above**. Our axioms definitely hold in this interpretation, but **S** does not. In fact, in this interpretation, the sentence **hat(A) = C** will hold.

KB₂ is a Weakling!

KB_2

S

all interpretations

6.034 - Spring 03 • 38

Slide 9.7.38

Just so we can see what's going on, let's go back to our Venn diagram for entailment. In this case, the blue set of interpretations in which the **KB** holds is not a subset of the green set of interpretations in which **S** holds. So, it is possible to have an interpretation, **I**₂, in which **KB** holds but not **S**. **KB** does not entail **S** (for it to do so, the blue area would have to be a subset of the green), and so we are not licensed to conclude **S** from **KB**.

How can we fix this problem? We need to add more axioms, in order to rule out **I**₂ as a possible interpretation. (Our goal is to make the blue area smaller, until it becomes a subset of the green area).

Slide 9.7.39

Here's a reasonable axiom to add: "for all x and y, if x is above y then y is not above x". It says that above is asymmetric. With this axiom added to our **KB**, **KB** no longer holds in **I**₂, and so our immediate problem is solved.

Axiomatization Example: Another Try

$\text{Above}(A, C)$ KB_3

$\text{Above}(B, D)$

$\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$

$\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$

$\forall x, y. \text{Above}(x, y) \rightarrow \neg \text{Above}(y, x)$

S hat(A) = A

- fails(KB_3, I_2)

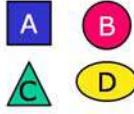
6.034 - Spring 03 • 39

Axiomatization Example: Another Try

Above(A, C)

KB₃

Above(B, D)

 $\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$ $\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$ $\forall x, y. \text{Above}(x, y) \rightarrow \neg \text{Above}(y, x)$ 

S hat(A) = A

- $I_3(A) = \blacksquare$
- $I_3(B) = \bullet$
- $I_3(C) = \blacktriangle$
- $I_3(D) = \blacksquare$
- $I_3(\text{Above}) = \{<\blacksquare, \blacktriangle>, <\bullet, \blacksquare>, <\bullet, \bullet>\}$
- $I_3(\text{hat}) = \{<\blacktriangle, \blacksquare>, <\blacksquare, \bullet>, <\bullet, \bullet>\}$

6.034 - Spring 03 * 40

Slide 9.7.40

But we're not out of the woods yet. Now consider interpretation I_3 , in which the circle is above the square. **KB** holds in I_3 , but **S** does not. So **S** is still not entailed by I_3 .

Slide 9.7.41

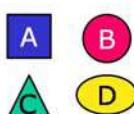
Clearly, we're missing some important information about our domain. Let's add the following important piece of information to our set of axioms: there is nothing above the square or the circle.

Axiomatization Example: One Last Time

Above(A, C)

KB₄

Above(B, D)

 $\neg \exists x. \text{Above}(x, A)$ $\neg \exists x. \text{Above}(x, B)$ $\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$ $\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$ 

S hat(A) = A

- fails(KB₄, I₃)
- KB₄ entails S

6.034 - Spring 03 * 42

Slide 9.7.42

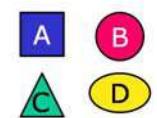
If we let our new **KB** have these axioms as well, then it fails in I_3 , and does, in fact, entail **S**. Whew.

Axiomatization Example: One Last Time

Above(A, C)

KB₄

Above(B, D)

 $\neg \exists x. \text{Above}(x, A)$ $\neg \exists x. \text{Above}(x, B)$ $\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$ $\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$ 

S hat(A) = A

6.034 - Spring 03 * 41

Slide 9.7.43

So, when you are axiomatizing a domain, it's important to be as specific as you can. You need to find a way to say everything that's crucial about your domain. You will never be able to draw false conclusions, but if you are too vague, you may not be able to draw some of the conclusions that you desire.

It turns out, in fact, that there is no way to axiomatize the natural numbers without including some weird unintended interpretations that have multiple copies of the natural numbers.

Still this shouldn't deter us from the enterprise of using logic to formalize reasoning inside computers. We don't have any substantially better alternatives, and, with care, we can make logic serve a useful purpose.

Axiomatization Example: One Last Time

Above(A, C)

KB₄

Above(B, D)

 $\neg \exists x. \text{Above}(x, A)$ $\neg \exists x. \text{Above}(x, B)$ $\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$ $\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$ 

S hat(A) = A

- fails(KB₄, I₃)
- KB₄ entails S

We'll prove S from KB₄ later.

6.034 - Spring 03 * 43

6.034 Notes: Section 10.1

Slide 10.1.1

A sentence written in conjunctive normal form looks like ((A or B or not C) and (B or D) and (not A) and (B or C)).

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

6.034 - Spring 03 • 1

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**

6.034 - Spring 03 • 2

Slide 10.1.2

Its outermost structure is a conjunction. It's a conjunction of multiple units. These units are called "clauses."

Slide 10.1.3

A clause is the disjunction of many things. The units that make up a clause are called literals.

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
- A, B, and $\neg C$ are **literals**

6.034 - Spring 03 • 3

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

- $(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$
- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
 - A, B, and $\neg C$ are **literals**, each of which is a variable or the negation of a variable.

Slide 10.1.4

And a literal is either a variable or the negation of a variable.

6.034 - Spring 03 • 4

Slide 10.1.5

So you get an expression where the negations are pushed in as tightly as possible, then you have ors, then you have ands. This is like saying that every assignment has to meet each of a set of requirements. You can think of each clause as a requirement. So somehow, the first clause has to be satisfied, and it has different ways that it can be satisfied, and the second one has to be satisfied, and the third one has to be satisfied, and so on.

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

- $(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$
- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
 - A, B, and $\neg C$ are **literals**, each of which is a variable or the negation of a variable.
 - Each clause is a requirement that must be satisfied and can be satisfied in multiple ways
 - Every sentence in propositional logic can be written in CNF

6.034 - Spring 03 • 6

Slide 10.1.6

You can take any sentence in propositional logic and write it in conjunctive normal form.

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

- $(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$
- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
 - A, B, and $\neg C$ are **literals**, each of which is a variable or the negation of a variable.
 - Each clause is a requirement that must be satisfied and can be satisfied in multiple ways

6.034 - Spring 03 • 5

Slide 10.1.7

Here's the procedure for converting sentences to conjunctive normal form.

Converting to CNF

6.034 - Spring 03 • 7

Converting to CNF

1. Eliminate arrows using definitions



6.034 - Spring 03 • 8

Slide 10.1.8

The first step is to eliminate single and double arrows using their definitions.

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\begin{aligned}\neg(\phi \vee \psi) &\equiv \neg\phi \wedge \neg\psi \\ \neg(\phi \wedge \psi) &\equiv \neg\phi \vee \neg\psi\end{aligned}$$

3. Distribute **or** over **and**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$



6.034 - Spring 03 • 10

Slide 10.1.10

The third step is to distribute or over and. That is, if we have $(A \text{ or } (B \text{ and } C))$ we can rewrite it as $(A \text{ or } B) \text{ and } (A \text{ or } C)$. You can prove to yourself, using the method of truth tables, that the distribution rule (and DeMorgan's laws) are valid.

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\begin{aligned}\neg(\phi \vee \psi) &\equiv \neg\phi \wedge \neg\psi \\ \neg(\phi \wedge \psi) &\equiv \neg\phi \vee \neg\psi\end{aligned}$$

6.034 - Spring 03 • 9

Slide 10.1.11

One problem with conjunctive normal form is that, although you can convert any sentence to conjunctive normal form, you might do it at the price of an exponential increase in the size of the expression. Because if you have A and B and C OR D and E and F, you end up making the cross-product of all of those things.

For now, we'll think about satisfiability problems, which are generally fairly efficiently converted into CNF.

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\begin{aligned}\neg(\phi \vee \psi) &\equiv \neg\phi \wedge \neg\psi \\ \neg(\phi \wedge \psi) &\equiv \neg\phi \vee \neg\psi\end{aligned}$$

3. Distribute **or** over **and**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

4. Every sentence can be converted to CNF, but it may grow exponentially in size



6.034 - Spring 03 • 11

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

Slide 10.1.12

Here's an example of converting a sentence to CNF.

6.034 - Spring 03 • 12

Slide 10.1.13

First we get rid of both arrows, using the rule that says "A implies B" is equivalent to "not A or B".

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

6.034 - Spring 03 • 13

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

2. Drive in negations

$$(\neg A \wedge \neg B) \vee (\neg C \vee D)$$

6.034 - Spring 03 • 14

Slide 10.1.14

Then we drive in the negation using deMorgan's law.

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

2. Drive in negations

$$(\neg A \wedge \neg B) \vee (\neg C \vee D)$$

3. Distribute

$$(\neg A \vee \neg C \vee D) \wedge (\neg B \vee \neg C \vee D)$$

6.034 - Spring 03 • 15

6.034 Notes: Section 10.2

Slide 10.2.1

We have talked a little bit about proof, with the idea that you write down some axioms -- statements that you're given -- and then you try to derive something from them. And we've all had practice doing that in high school geometry and we've talked a little bit about natural deduction. So what we're going to talk about now is resolution. Which is the way that pretty much every modern automated theorem-prover is implemented. It seems to be the best way for computers to think about proving things.

Propositional Resolution

6.034 - Spring 03 • 1

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta}{\neg \beta \vee \gamma} \quad \frac{\alpha \vee \beta}{\alpha \vee \gamma}$$

6.034 - Spring 03 • 2

Slide 10.2.2

So here's the **resolution** inference rule, in the propositional case. It says that if you know "alpha or beta", and you know "not beta or gamma", then you're allowed to conclude "alpha or gamma".

Remember from when we looked at inference rules before that these Greek letters are meta-variables. They can stand for big chunks of propositional logic, as long as the parts match up in the right way. So if you know something of the form "alpha or beta", and you also know that "not beta or gamma", then you can conclude "alpha or gamma".

Slide 10.2.3

It turns out that this one rule is all you need to prove anything in propositional logic. At least, to prove that a set of sentences is not satisfiable. So, let's see how this is going to work. There's a proof strategy called **resolution refutation**, with three steps. It goes like this.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta}{\neg \beta \vee \gamma} \quad \frac{\alpha \vee \beta}{\alpha \vee \gamma}$$

- Resolution refutation:

6.034 - Spring 03 • 3

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \\ \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- Resolution refutation:
 - Convert all sentences to CNF

Slide 10.2.4

First, you convert all of your sentences to conjunctive normal form. You already know how to do this! Then, you write each clause down as a premise or given in your proof.

6.034 - Spring 03 • 4

Slide 10.2.5

Then, you negate the desired conclusion -- so you have to say what you're trying to prove, but what we're going to do is essentially a proof by contradiction. You've all seen the strategy of proof by contradiction (or, if we're being fancy and Latin, reductio ad absurdum). You assert that the thing that you're trying to prove is false, and then you try to derive a contradiction. That's what we're going to do. So you negate the desired conclusion and convert that to CNF. And you add each of these clauses as a premise of your proof, as well.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \\ \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- Resolution refutation:
 - Convert all sentences to CNF
 - Negate the desired conclusion (converted to CNF)

6.034 - Spring 03 • 5

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \\ \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- Resolution refutation:
 - Convert all sentences to CNF
 - Negate the desired conclusion (converted to CNF)
 - Apply resolution rule until either
 - Derive false (a contradiction)
 - Can't apply any more

Slide 10.2.6

Now we apply the resolution rule until one of two things happens. We might derive "false", which means that the conclusion did, in fact, follow from the things that we had assumed. If you assert that the negation of the thing that you're interested in is true, and then you prove for a while and you manage to prove false, then you've succeeded in a proof by contradiction of the thing that you were trying to prove in the first place. Or, we might find ourselves in a situation where we can't apply the resolution rule any more, but we still haven't managed to derive false.

Slide 10.2.7

What if you can't apply the resolution rule anymore? Is there anything in particular that you can conclude? In fact, you can conclude that the thing that you were trying to prove can't be proved. So resolution refutation for propositional logic is a complete proof procedure. If the thing that you're trying to prove is, in fact, entailed by the things that you've assumed, then you can prove it using resolution refutation. It's guaranteed that you'll always either prove false, or run out of possible steps. It's complete, because it always generates an answer. Furthermore, the process is sound: the answer is always correct.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \\ \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- Resolution refutation:
 - Convert all sentences to CNF
 - Negate the desired conclusion (converted to CNF)
 - Apply resolution rule until either
 - Derive false (a contradiction)
 - Can't apply any more
- Resolution refutation is sound and complete
 - If we derive a contradiction, then the conclusion follows from the axioms
 - If we can't apply any more, then the conclusion cannot be proved from the axioms.

6.034 - Spring 03 • 7

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation

6.034 - Spring 03 • 8

Slide 10.2.8

So let's just do a proof. Let's say I'm given "P or Q", "P implies R" and "Q implies R". I would like to conclude R from these three axioms. I'll use the word "axiom" just to mean things that are given to me right at the moment.

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given

6.034 - Spring 03 • 9

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given

6.034 - Spring 03 • 10

Slide 10.2.10

Now, "P implies R" turns into "not P or R".

Slide 10.2.11

Similarly, "Q implies R" turns into "not Q or R"

Propositional Resolution Example

1	$P \vee Q$
2	$\neg P \vee R$
3	$\neg Q \vee R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given

6.034 - Spring 03 • 11

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion

6.034 - Spring 03 • 12

Slide 10.2.12

Now we want to add one more thing to our list of given statements. What's it going to be? Not R. Right? We're going to assert the negation of the thing we're trying to prove. We'd like to prove that R follows from these things. But what we're going to do instead is say not R, and now we're trying to prove false. And if we manage to prove false, then we will have a proof that R is entailed by the assumptions.

Slide 10.2.13

We'll draw a blue line just to divide the assumptions from the proof steps. And now, we look for opportunities to apply the resolution rule. You can do it in any order you like (though some orders of application will result in much shorter proofs than others).

Propositional Resolution Example

Prove R
1 $P \vee Q$
2 $P \rightarrow R$
3 $Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion

6.034 - Spring 03 • 13

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2

6.034 - Spring 03 • 14

Slide 10.2.14

We can apply resolution to lines 1 and 2, and get "Q or R" by resolving away P.

Propositional Resolution Example

Prove R
1 $P \vee Q$
2 $P \rightarrow R$
3 $Q \rightarrow R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4

6.034 - Spring 03 • 15

Slide 10.2.15

And we can take lines 2 and 4, resolve away R, and get "not P."

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$\neg P \vee R$
3	$\neg Q \vee R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4

6.034 - Spring 03 • 16

Slide 10.2.16

Similarly, we can take lines 3 and 4, resolve away R, and get "not Q".

Slide 10.2.17

By resolving away Q in lines 5 and 7, we get R.

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$\neg P \vee R$
3	$\neg Q \vee R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7

6.034 - Spring 03 • 17

Slide 10.2.18

And finally, resolving away R in lines 4 and 8, we get the empty clause, which is false. We'll often draw this little black box to indicate that we've reached the desired contradiction.

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$\neg P \vee R$
3	$\neg Q \vee R$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	*	4,8

6.034 - Spring 03 • 18

Slide 10.2.19

How did I do this last resolution? Let's see how the resolution rule is applied to lines 4 and 8. The way to look at it is that R is really "false or R", and that "not R" is really "not R or false". (Of course, the order of the disjuncts is irrelevant, because disjunction is commutative). So, now we resolve away R, getting "false or false", which is false.

Propositional Resolution Example

Prove R

1	$P \vee Q$
2	$\neg P \vee R$
3	$\neg Q \vee R$

$$\begin{array}{l} \text{false} \vee R \\ \neg R \vee \text{false} \\ \hline \text{false} \vee \text{false} \end{array}$$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	*	4,8

6.034 - Spring 03 • 19

Propositional Resolution Example

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	*	4,8

Prove R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

false $\vee R$
 $\neg R \vee$ false
false \vee false

6.034 - Spring 03 • 20

Slide 10.2.20

One of these steps is unnecessary. Which one? Line 6. It's a perfectly good proof step, but it doesn't contribute to the final conclusion, so we could have omitted it.

The Power of False

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion

Prove Z

1	P
2	$\neg P$

6.034 - Spring 03 • 22

Slide 10.2.22

We start by writing down the assumptions and the negation of the conclusion.

The Power of False

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Prove Z

1	P
2	$\neg P$

6.034 - Spring 03 • 23

Slide 10.2.23

Then, we can resolve away P in lines 1 and 2, getting a contradiction right away.

The Power of False

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Prove Z

1	P
2	$\neg P$

6.034 - Spring 03 • 23

The Power of False

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Prove Z

1	P
2	$\neg P$

6.034 - Spring 03 • 23

The Power of False

Prove Z	
1	P
2	$\neg P$

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Note that $(P \wedge \neg P) \rightarrow Z$ is valid

Slide 10.2.24

Because we can prove Z from "P and not P" using a sound proof procedure, then "P and not P" entails Z.



6.034 - Spring 03 • 24

Slide 10.2.25

So, we see, again, that any conclusion follows from a contradiction. This is the property that can make logical systems quite brittle; they're not robust in the face of noise. This problem has been recently addressed in AI by a move to probabilistic reasoning methods. Unfortunately, they're out of the scope of this course.

The Power of False

Prove Z	
1	P
2	$\neg P$

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Note that $(P \wedge \neg P) \rightarrow Z$ is valid

Any conclusion follows from a contradiction – and so strict logic systems are very brittle.

6.034 - Spring 03 • 25



Example Problem

Convert to CNF

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

6.034 - Spring 03 • 26

Slide 10.2.26

Here's an example problem. Stop and do the conversion into CNF before you go to the next slide.



Slide 10.2.27

So, the first formula turns into "P or Q".

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

- $\neg(\neg P \vee Q) \vee Q$
- $(P \wedge \neg Q) \vee Q$
- $(P \vee Q) \wedge (\neg Q \vee Q)$
- $(P \vee Q)$



6.034 - Spring 03 • 27

Example Problem

Convert to CNF

Example Problem

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

Convert to CNF

- $\neg(\neg P \vee Q) \vee Q$
- $(P \wedge \neg Q) \vee Q$
- $(P \vee Q) \wedge (\neg Q \vee Q)$
- $(P \vee Q)$

- $\neg(\neg P \vee P) \vee R$
- $(P \wedge \neg P) \vee R$
- $(P \vee R) \wedge (\neg P \vee R)$

6.034 - Spring 03 • 28

Slide 10.2.28

The second turns into ("P or R" and "not P or R"). We probably should have simplified it into "False or R" at the second step, which reduces just to R. But we'll leave it as is, for now.

Slide 10.2.29

Finally, the last formula requires us to do a big expansion, but one of the terms is true and can be left out. So, we get "(R or S) and (R or not Q) and (not S or not Q)".

Example Problem

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

Convert to CNF

- $\neg(\neg P \vee Q) \vee Q$
- $(P \wedge \neg Q) \vee Q$
- $(P \vee Q) \wedge (\neg Q \vee Q)$
- $(P \vee Q)$

- $\neg(\neg P \vee P) \vee R$
- $(P \wedge \neg P) \vee R$
- $(P \vee R) \wedge (\neg P \vee R)$

- $\neg(\neg R \vee S) \vee \neg(\neg S \vee Q)$
- $(R \wedge \neg S) \vee (S \wedge \neg Q)$
- $(R \vee S) \wedge (\neg S \vee S) \wedge (R \vee \neg Q) \wedge (\neg S \vee \neg Q)$
- $(R \vee S) \wedge (R \vee \neg Q) \wedge (\neg S \vee \neg Q)$

6.034 - Spring 03 • 29

Resolution Proof Example

Prove R

1	$P \vee Q$
2	$P \vee R$
3	$\neg P \vee R$
4	$R \vee S$
5	$R \vee \neg Q$
6	$\neg S \vee \neg Q$
7	$\neg R$
	Neg

6.034 - Spring 03 • 30

Slide 10.2.30

Now we can almost start the proof. We copy each of the clauses over here, and we add the negation of the query. Please stop and do this proof yourself before going on.

Resolution Proof Example

Prove R

1	$P \vee Q$
2	$P \vee R$
3	$\neg P \vee R$
4	$R \vee S$
5	$R \vee \neg Q$
6	$\neg S \vee \neg Q$
7	$\neg R$
	Neg
8	S
9	$\neg Q$
10	P
11	R
12	*

6.034 - Spring 03 • 31

Slide 10.2.31

Here's a sample proof. It's one of a whole lot of possible proofs.

Proof Strategies

Slide 10.2.32

In choosing among all the possible proof steps that you can do at any point, there are two rules of thumb that are really important.

6.034 - Spring 03 • 32

Slide 10.2.33

The unit preference rule says that if you can involve a clause that has only one literal in it, that's usually a good idea. It's good because you get back a shorter clause. And the shorter a clause is, the closer it is to false.

Proof Strategies

- Unit preference: prefer a resolution step involving a unit clause (clause with one literal).
 - Produces a shorter clause – which is good since we are trying to produce a zero-length clause, that is, a contradiction.

6.034 - Spring 03 • 33

Proof Strategies

- Unit preference: prefer a resolution step involving a unit clause (clause with one literal).
 - Produces a shorter clause – which is good since we are trying to produce a zero-length clause, that is, a contradiction.
- Set of support: Choose a resolution involving the negated goal or any clause derived from the negated goal.
 - We're trying to produce a contradiction that follows from the negated goal, so these are "relevant" clauses.
 - If a contradiction exists, one can find one using the set-of-support strategy.

6.034 - Spring 03 • 34

Slide 10.2.34

The set-of-support rule says you should involve the thing that you're trying to prove. It might be that you can derive conclusions all day long about the solutions to chess games and stuff from the axioms, but once you're trying to prove something about what way to run, it doesn't matter. So, to direct your "thought" processes toward deriving a contradiction, you should always involve a clause that came from the negated goal, or that was produced by the set of support rule. Adhering to the set-of-support rule will still make the resolution refutation process sound and complete.

6.034 Notes: Section 10.3

Slide 10.3.1

We are going to use resolution refutation to do proofs in first-order logic. It's a fair amount trickier than in propositional logic, though, because now we have variables to contend with.

First-Order Resolution

6.034 - Spring 03 * 1

First-Order Resolution

$$\begin{array}{c} \forall x. P(x) \rightarrow Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

uppercase letters:
constants
lowercase letters:
variables

6.034 - Spring 03 * 2

Slide 10.3.2

Let's try to get some intuition through an example. Imagine you knew "for all x, P of x implies Q of x." And let's say you also knew **P(A)**. What would you be able to conclude? **Q(A)**, right? You ought to be able to conclude **Q(A)**.

First-Order Resolution

$$\begin{array}{c} \forall x. P(x) \rightarrow Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

Syllogism:
All men are mortal
Socrates is a man
Socrates is mortal

uppercase letters:
constants
lowercase letters:
variables

6.034 - Spring 03 * 3

First-Order Resolution

$$\begin{array}{c} \forall x. P(x) \rightarrow Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

Syllogism:
All men are mortal
Socrates is a man
Socrates is mortal

uppercase letters:
constants
lowercase letters:
variables

$$\begin{array}{c} \forall x. \neg P(x) \vee Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

Equivalent by
definition of
implication

6.034 - Spring 03 * 4

Slide 10.3.4

So, how can we justify this conclusion formally? Well, the first step would be to get rid of the implication.

Slide 10.3.5

Next, we could substitute the constant **A** in for the variable **x** in the universally quantified sentence. By the semantics of universal quantification, that's allowed. A universally quantified statement has to be true of every object in the universe, including whatever object is denoted by the constant symbol **A**. And now, we can apply the propositional resolution rule.

The hard part is figuring out how to instantiate the variables in the universal statements. In this problem, it was clear that **A** was the relevant individual. But it not necessarily clear at all how to do that automatically.

First-Order Resolution

$\frac{\forall x. P(x) \rightarrow Q(x)}{P(A)}$	Syllogism: All men are mortal Socrates is a man Socrates is mortal	<small>uppercase letters: constants</small> <small>lowercase letters: variables</small>
$\frac{\forall x. \neg P(x) \vee Q(x)}{P(A)}$	Equivalent by definition of implication	
$\frac{\neg P(A) \vee Q(A)}{P(A)}$	Substitute A for x, still true then	Propositional resolution

6.034 - Spring 03 * 5

First-Order Resolution

$\frac{\forall x. P(x) \rightarrow Q(x)}{P(A)}$	Syllogism: All men are mortal Socrates is a man Socrates is mortal	<small>uppercase letters: constants</small> <small>lowercase letters: variables</small>
$\frac{\forall x. \neg P(x) \vee Q(x)}{P(A)}$	Equivalent by definition of implication	Two new things: <ul style="list-style-type: none"> • converting FOL to clausal form • resolution with variable substitution
$\frac{\neg P(A) \vee Q(A)}{P(A)}$	Substitute A for x, still true then	Propositional resolution

6.034 - Spring 03 * 6

Slide 10.3.6

Now, we have to do two jobs before we can see how to do first-order resolution.

The first is to figure out how to convert from sentences with the whole rich structure of quantifiers into a form that lets us use resolution. We'll need to convert to clausal form, which is a kind of generalization of CNF to first-order logic.

The second is to automatically determine which variables to substitute in for which other ones when we're performing first-order resolution. This process is called unification.

We'll do clausal form next, then unification, and finally put it all together.

Slide 10.3.7

Clausal form (which is also sometimes called "prenex normal form") is like CNF in its outer structure (a conjunction of disjunctions, or an "and" of "ors"). But it has no quantifiers. Here's an example conversion.

Clausal Form

- like CNF in outer structure
- no quantifiers

$$\frac{\forall x. \exists y. P(x) \rightarrow R(x, y)}{\neg P(x) \vee R(x, F(x))}$$

6.034 - Spring 03 * 7

Converting to Clausal Form

6.034 - Spring 03 * 8

Slide 10.3.8

We'll go through a step-by-step procedure for systematically converting any sentence in first-order logic into clausal form.

Slide 10.3.9

The first step you guys know very well is to eliminate arrows. You already know how to do that. You convert an equivalence into two implications. And anywhere you see alpha right arrow beta, you just change it into not alpha or beta.

Converting to Clausal Form**1. Eliminate arrows**

$$\begin{aligned}\alpha \leftrightarrow \beta &\Rightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha) \\ \alpha \rightarrow \beta &\Rightarrow \neg\alpha \vee \beta\end{aligned}$$

6.034 - Spring 03 • 9

Converting to Clausal Form**1. Eliminate arrows**

$$\begin{aligned}\alpha \leftrightarrow \beta &\Rightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha) \\ \alpha \rightarrow \beta &\Rightarrow \neg\alpha \vee \beta\end{aligned}$$

2. Drive in negation

$$\begin{aligned}\neg(\alpha \vee \beta) &\Rightarrow \neg\alpha \wedge \neg\beta \\ \neg(\alpha \wedge \beta) &\Rightarrow \neg\alpha \vee \neg\beta \\ \neg\neg\alpha &\Rightarrow \alpha \\ \neg\forall x. \alpha &\Rightarrow \exists x. \neg\alpha \\ \neg\exists x. \alpha &\Rightarrow \forall x. \neg\alpha\end{aligned}$$

6.034 - Spring 03 • 10

Slide 10.3.10

The next thing you do is drive in negation. You already basically know how to do that. We have deMorgan's laws to deal with conjunction and disjunction, and we can eliminate double negations.

As a kind of extension of deMorgan's laws, we also have that **not (for all x, alpha)** turns into **exists x such that not alpha**. And that **not (exists x such that alpha)** turns into **for all x, not alpha**. The reason these are extensions of deMorgan's laws, in a sense, is that a universal quantifier can be seen abstractly as a conjunction over all possible assignments of x, and an existential as a disjunction.

Slide 10.3.11

The next step is to rename variables apart. The idea here is that every quantifier in your sentence should be over a different variable. So, if you had two different quantifications over x, you should rename one of them to use a different variable (which doesn't change the semantics at all). In this example, we have two quantifications involving the variable x. It's especially confusing in this case, because they're nested. The rules are like those for a programming language: a variable is captured by the nearest enclosing quantifier. So the x in Q(x,y) is really a different variable from the x in P(x). To make this distinction clear, and to automate the downstream processing into clausal form, we'll just rename each of the variables.

Converting to Clausal Form**1. Eliminate arrows**

$$\begin{aligned}\alpha \leftrightarrow \beta &\Rightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha) \\ \alpha \rightarrow \beta &\Rightarrow \neg\alpha \vee \beta\end{aligned}$$

2. Drive in negation

$$\begin{aligned}\neg(\alpha \vee \beta) &\Rightarrow \neg\alpha \wedge \neg\beta \\ \neg(\alpha \wedge \beta) &\Rightarrow \neg\alpha \vee \neg\beta \\ \neg\neg\alpha &\Rightarrow \alpha \\ \neg\forall x. \alpha &\Rightarrow \exists x. \neg\alpha \\ \neg\exists x. \alpha &\Rightarrow \forall x. \neg\alpha\end{aligned}$$

3. Rename variables apart

$$\begin{aligned}\forall x. \exists y. (\neg P(x) \vee \exists x. Q(x, y)) &\Rightarrow \\ \forall x_1. \exists y_2. (\neg P(x_1) \vee \exists x_3. Q(x_3, y_2)) &\end{aligned}$$

6.034 - Spring 03 • 11

Skolemization**4. Skolemize**

6.034 - Spring 03 • 12

Slide 10.3.12

Now, here's the step that many people find confusing. The name is already a good one. Step four is to skolemize, named after a logician called Thoralf Skolem. Imagine that you have a sentence that looks like: there exists an x such that P(x). The goal here is to somehow arrive at a representation that doesn't have any quantifiers in it. Now, if we only had one kind of quantifier in first-order logic, it would be easy because we could just mention variables and all the variables would be implicitly quantified by the kind of quantifier that we have. But because we have two quantifiers, if we dropped all the quantifiers off, there's a mess, because you don't know which kind of quantification is supposed to apply to which variable.

Slide 10.3.13

The Skolem insight is that when you have an existential quantification like this, you're saying there is such a thing as a unicorn, let's say that P means "unicorn". There exists a thing such that it's a unicorn. You can just say, all right, well, if there is one, let's call it Fred. That's it. That's what Skolemization is. So instead of writing exists an x such that P(x), you say P(Fred). The trick is that it absolutely must be a new name. It can't be any other name of any other thing that you know about. If you're in the process of inferring things about John and Mary, then it's not good to say, oh, there's a unicorn and it's John -- because that's adding some information to the picture. So to Skolemize, in the simple case, means to substitute a brand-new name for each existentially quantified variable.

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$

6.034 - Spring 03 • 13

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$

6.034 - Spring 03 • 14

Slide 10.3.14

For example, if I have **exists x, y such that R(x,y)**, then it's going to have to turn into **R(Thing1, Thing2)**. Because we have two different variables here, they have to be given different names.

Slide 10.3.15

But the names also have to persist so that if you have **exists x such that P(x) and Q(x)**, then if you skolemize that expression you should get **P(Fleep) and Q(Fleep)**. You make up a name and you put it in there, but every occurrence of this variable has to get mapped into that same unique name.

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$
 $\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$

6.034 - Spring 03 • 15

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$
 $\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$
 $\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$

6.034 - Spring 03 • 16

Slide 10.3.16

If you have different quantifiers, then you need to use different names.

Slide 10.3.17

All right. If that's all we had to do it wouldn't be too bad. But there's one more case. We can illustrate it by looking at two interpretations of "Everyone loves someone".

In the first case, there is a single y that everyone loves. So we do ordinary skolemization and decide to call that person **Englebert**.

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$
 $\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$
 $\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$
 $\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$

6.034 - Spring 03 • 17

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$
 $\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$
 $\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$
 $\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$
- substitute new function of all universal vars in outer scopes

6.034 - Spring 03 • 18

Slide 10.3.18

In the second case, there is a different y , potentially, for each x . So, if we were just to substitute in a single constant name for y , we'd lose that information. We'd get the same result as above, which would be wrong. So, when you are skolemizing an existential variable, you have to look at the other quantifiers that contain the one you're skolemizing, and instead of substituting in a new constant, you substitute in a brand new function symbol, applied to any variables that are universally quantified in an outer scope.

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$
 $\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$
 $\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$
 $\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$
- substitute new function of all universal vars in outer scopes
 $\forall x. \exists y. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Beloved}(x))$

6.034 - Spring 03 • 19

Skolemization**4. Skolemize**

- substitute new name for each existential var
 $\exists x. P(x) \Rightarrow P(\text{Fred})$
 $\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$
 $\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$
 $\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$
 $\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$
- substitute new function of all universal vars in outer scopes
 $\forall x. \exists y. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Beloved}(x))$
 $\forall x. \exists y. \forall z. \exists w. P(x, y, z) \wedge R(y, z, w) \Rightarrow P(x, F(x), z) \wedge R(F(x), z, G(x, z))$

6.034 - Spring 03 • 20

Slide 10.3.20

So, in this example, we see that the existential variable w is contained in the scope of two universally quantified variables, x , and z . So, we replace it with $G(x, z)$, which allows it to depend on the choices of x and z .

Note also, that I've been using silly names for Skolem constants and functions (like **Englebert** and **Beloved**). But you, or the computer, are only obliged to use new ones, so things like **F123221** are completely appropriate, as well.

Slide 10.3.21

Now we can drop the universal quantifiers because we just replaced all of the existential quantifiers with Skolem constants or functions. Now there's only one kind of quantifier left, so we can just drop them without losing information.

Convert to Clausal Form: Last Steps**5. Drop universal quantifiers**

$$\forall x. \text{Loves}(x, \text{Beloved}(x)) \Rightarrow \text{Loves}(x, \text{Beloved}(x))$$

6.034 - Spring 03 • 21

Convert to Clausal Form: Last Steps**5. Drop universal quantifiers**

$$\forall x. \text{Loves}(x, \text{Beloved}(x)) \Rightarrow \text{Loves}(x, \text{Beloved}(x))$$

6. Distribute or over and; return clauses

$$\begin{aligned} P(z) \vee (Q(z, w) \wedge R(w, z)) \Rightarrow \\ \{\{P(z), Q(z, w)\}, \{P(z), R(w, z)\}\} \end{aligned}$$

6.034 - Spring 03 • 22

Slide 10.3.22

And then we convert to clauses. This just means multiplying out the and's and the or's, because we already eliminated the arrows and pushed in the negations. We'll return a set of sets of literals. A literal, in this case, is a predicate applied to some terms, or the negation of a predicate applied to some terms.

I'm using set notation here for clauses, just to emphasize that they aren't lists; that the order of the literals within a clause and the order of the clauses within a set of clauses, doesn't have any effect on its meaning.

Slide 10.3.23

Finally, we can rename the variables in each clause. It's okay to do that because **for all x, P(x)** and **Q(x)** is equivalent to **for all y, P(y)** and **for all z, P(z)**. In fact, you don't really need to do this step, because we're assuming that you're always going to rename the variables before you do a resolution step.

Convert to Clausal Form: Last Steps**5. Drop universal quantifiers**

$$\forall x. \text{Loves}(x, \text{Beloved}(x)) \Rightarrow \text{Loves}(x, \text{Beloved}(x))$$

6. Distribute or over and; return clauses

$$\begin{aligned} P(z) \vee (Q(z, w) \wedge R(w, z)) \Rightarrow \\ \{\{P(z), Q(z, w)\}, \{P(z), R(w, z)\}\} \end{aligned}$$

7. Rename the variables in each clause

$$\begin{aligned} \{\{P(z), Q(z, w)\}, \{P(z), R(w, z)\}\} \Rightarrow \\ \{\{P(z_1), Q(z_1, w_1)\}, \{P(z_2), R(w_2, z_2)\}\} \end{aligned}$$

6.034 - Spring 03 • 23

Example: Converting to clausal form**Slide 10.3.24**

So, let's do an example, starting with English sentences, writing them down in first-order logic, and converting them to clausal form. Later, we'll do a resolution proof using these clauses.

6.034 - Spring 03 • 24

Slide 10.3.25

John owns a dog. We can write that in first-order logic as **there exists an x such that $D(x)$ and $O(J, x)$** . So, we're letting **D** stand for "is a dog" and **O** stand for "owns" and **J** stand for John.

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J, x)$
$D(\text{Fido}) \wedge O(J, \text{Fido})$

6.034 - Spring 03 • 25

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J, x)$
$D(\text{Fido}) \wedge O(J, \text{Fido})$

6.034 - Spring 03 • 26

Slide 10.3.26

To convert this to clausal form, we can start at step 4, Skolemization, because the previous three steps are unnecessary for this sentence. Since we just have an existential quantifier over **x**, without any enclosing universal quantifiers, we can simply pick a new name and substitute it in for **x**. Let's call **x Fido**. This will give us two clauses with no variables, and we're done.

Slide 10.3.27

Anyone who owns a dog is a lover of animals. We can write that in FOL as **For all x , if there exists a y such that $D(y)$ and $O(x,y)$, then $L(x)$** . We've added a new predicate symbol **L** to stand for "is a lover of animals".

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J, x)$
$D(\text{Fido}) \wedge O(J, \text{Fido})$

6.034 - Spring 03 • 27

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J, x)$
$D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
$\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
$\forall x. (\neg\exists y. (D(y) \wedge O(x, y))) \vee L(x)$

6.034 - Spring 03 • 28

Slide 10.3.28

First, we get rid of the arrow. Note that the parentheses are such that the existential quantifier is part of the antecedent, but the universal quantifier is not. The answer would come out very differently if those parens weren't there; this is a place where it's easy to make mistakes.

Slide 10.3.29

Next, we drive in the negations. We'll do it in two steps. I find that whenever I try to be clever and skip steps, I do something wrong.

Example: Converting to clausal form

a. John owns a dog
 $\exists x. D(x) \wedge O(J, x)$
 $D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
 $\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
 $\forall x. (\neg\exists y. (D(y) \wedge O(x, y)) \vee L(x))$
 $\forall x. \forall y. \neg(D(y) \wedge O(x, y)) \vee L(x)$
 $\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$

6.034 - Spring 03 • 29

**Example: Converting to clausal form**

a. John owns a dog
 $\exists x. D(x) \wedge O(J, x)$
 $D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
 $\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
 $\forall x. (\neg\exists y. (D(y) \wedge O(x, y)) \vee L(x))$
 $\forall x. \forall y. \neg(D(y) \wedge O(x, y)) \vee L(x)$
 $\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$
 $\neg D(y) \vee \neg O(x, y) \vee L(x)$

6.034 - Spring 03 • 30

**Slide 10.3.30**

There's no skolemization to do, since there aren't any existential quantifiers. So, we can just drop the universal quantifiers, and we're left with a single clause.

Example: Converting to clausal form

a. John owns a dog
 $\exists x. D(x) \wedge O(J, x)$
 $D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
 $\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
 $\forall x. (\neg\exists y. (D(y) \wedge O(x, y)) \vee L(x))$
 $\forall x. \forall y. \neg(D(y) \wedge O(x, y)) \vee L(x)$
 $\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$
 $\neg D(y) \vee \neg O(x, y) \vee L(x)$

c. Lovers-of-animals do not kill animals
 $\forall x. L(x) \rightarrow (\forall y. A(y) \rightarrow \neg K(x, y))$

6.034 - Spring 03 • 31

**Example: Converting to clausal form**

a. John owns a dog
 $\exists x. D(x) \wedge O(J, x)$
 $D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
 $\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
 $\forall x. (\neg\exists y. (D(y) \wedge O(x, y)) \vee L(x))$
 $\forall x. \forall y. \neg(D(y) \wedge O(x, y)) \vee L(x)$
 $\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$
 $\neg D(y) \vee \neg O(x, y) \vee L(x)$

c. Lovers-of-animals do not kill animals
 $\forall x. L(x) \rightarrow (\forall y. A(y) \rightarrow \neg K(x, y))$
 $\forall x. \neg L(x) \vee (\forall y. A(y) \rightarrow \neg K(x, y))$
 $\forall x. \neg L(x) \vee (\forall y. \neg A(y) \vee \neg K(x, y))$

6.034 - Spring 03 • 32

**Slide 10.3.32**

First, we get rid of the arrows, in two steps.

Slide 10.3.33

Then we're left with only universal quantifiers, which we drop, yielding one clause.

Example: Converting to clausal form

a. John owns a dog
 $\exists x. D(x) \wedge O(J,x)$
 $D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
 $\forall x. (\exists y. D(y) \wedge O(x,y)) \rightarrow L(x)$
 $\forall x. (\neg\exists y. (D(y) \wedge O(x,y))) \vee L(x)$
 $\forall x. \forall y. \neg(D(y) \wedge O(x,y)) \vee L(x)$
 $\forall x. \forall y. \neg D(y) \vee \neg O(x,y) \vee L(x)$
 $\neg D(y) \vee \neg O(x,y) \vee L(x)$

c. Lovers-of-animals do not kill animals
 $\forall x. L(x) \rightarrow (\forall y. A(y) \rightarrow \neg K(x,y))$
 $\forall x. \neg L(x) \vee (\forall y. A(y) \rightarrow \neg K(x,y))$
 $\forall x. \neg L(x) \vee (\forall y. \neg A(y) \vee \neg K(x,y))$
 $\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$

6.034 - Spring 03 • 33

**More converting to clausal form**

d. Either Jack killed Tuna or curiosity killed Tuna
 $K(J,T) \vee K(C,T)$

6.034 - Spring 03 • 34

**Slide 10.3.34**

We just have three more easy ones. "Either Jack killed Tuna or curiosity killed Tuna." Everything here is a constant, so we get $K(J,T)$ or $K(C,T)$.

More converting to clausal form

d. Either Jack killed Tuna or curiosity killed Tuna
 $K(J,T) \vee K(C,T)$

e. Tuna is a cat
 $C(T)$

6.034 - Spring 03 • 35

**More converting to clausal form**

d. Either Jack killed Tuna or curiosity killed Tuna
 $K(J,T) \vee K(C,T)$

e. Tuna is a cat
 $C(T)$

f. All cats are animals
 $\neg C(x) \vee A(x)$

6.034 - Spring 03 • 36

**Slide 10.3.36**

And "All cats are animals" is **not** $C(x)$ or $A(x)$. I left out the steps here, but I'm sure you can fill them in.

Okay. Next, we'll see how to match up literals that have variables in them, and move on to resolution.

6.034 Notes: Section 10.4

Slide 10.4.1

We introduced first-order resolution and said there were two issues to resolve before we could do it. First was conversion to clausal form, which we've done. Now we have to figure out how to instantiate the variables in the universal statements. In this problem, it was clear that A was the relevant individual. But it is not necessarily clear at all how to do that automatically.

First-Order Resolution

$\begin{array}{c} \forall x. P(x) \rightarrow Q(x) \\ \hline P(A) \\ \hline Q(A) \end{array}$	Syllogism: All men are mortal Socrates is a man Socrates is mortal	uppercase letters: constants
$\begin{array}{c} \forall x. \neg P(x) \vee Q(x) \\ \hline P(A) \\ \hline Q(A) \end{array}$	Equivalent by definition of implication	lowercase letters: variables
$\begin{array}{c} \neg P(A) \vee Q(A) \\ \hline P(A) \\ \hline Q(A) \end{array}$	Substitute A for x, still true then Propositional resolution	The key is finding the correct substitutions for the variables.

6.034 - Spring 03 • 1

Substitutions

Slide 10.4.2

In order to derive an algorithmic way of finding the right instantiations for the universal variables, we need something called substitutions.



6.034 - Spring 03 • 2

Slide 10.4.3

Here's an example of what we called an atomic sentence before: a predicate applied to some terms. There are two variables here: x and y. We can think of many different ways to substitute terms into this expression. Those are called substitution instances of the expression.

Substitutions

$P(x, f(y), B)$: an atomic sentence

6.034 - Spring 03 • 3

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment



6.034 - Spring 03 * 4

Slide 10.4.4

A substitution is a set of variable-term pairs, written this way. It says that whenever you see variable v_i , you should substitute in term t_i . There should not be more than one entry for a single variable.

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	



6.034 - Spring 03 * 6

Slide 10.4.6

Here's another substitution instance of our sentence: $P(x, f(A), B)$, We've put the constant A in for the variable y.

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	

6.034 - Spring 03 * 5

Slide 10.4.7

To get $P(g(z), f(A), B)$, we substitute the term $g(z)$ in for x and the constant A for y .

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	
$P(g(z), f(A), B)$	$\{x/g(z), y/A\}$	

6.034 - Spring 03 * 7

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	
$P(g(z), f(A), B)$	$\{x/g(z), y/A\}$	
$P(C, f(A), B)$	$\{x/C, y/A\}$	Ground instance



6.034 - Spring 03 • 8

Slide 10.4.8

Here's one more -- $P(C, f(A), B)$. It's sort of interesting, because it doesn't have any variables in it. We'll call an atomic sentence with no variables a ground instance. Ground means it doesn't have any variables.

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	
$P(g(z), f(A), B)$	$\{x/g(z), y/A\}$	
$P(C, f(A), B)$	$\{x/C, y/A\}$	Ground instance

Applying a substitution:

$$P(x, f(y), B) \{y/A\} = P(x, f(A), B)$$

$$P(x, f(y), B) \{y/A, x/y\} = P(A, f(A), B)$$

6.034 - Spring 03 • 10

Slide 10.4.11

Now we'll look at the process of unification, which is finding a substitution that makes two expressions match each other exactly.

Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	
$P(g(z), f(A), B)$	$\{x/g(z), y/A\}$	
$P(C, f(A), B)$	$\{x/C, y/A\}$	Ground instance

6.034 - Spring 03 • 9

Slide 10.4.10

We'll use the notation of an expression followed by a substitution to mean the expression that we get by applying the substitution to the expression. To apply a substitution to an expression, we look to see if any of the variables in the expression have entries in the substitution. If they do, we substitute in the appropriate new expression for the variable, and continue to look for possible substitutions until no more opportunities exist.

So, in this second example, we substitute A in for y, then y in for x, and then we keep going and substitute A in for y again.

Unification

6.034 - Spring 03 • 11

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$

Slide 10.4.12

So, expressions ω_1 and ω_2 are unifiable if and only if there exists a substitution S such that we get the same thing when we apply S to ω_1 as we do when we apply S to ω_2 . That substitution, S , is called a unifier of ω_1 and ω_2 .

6.034 - Spring 03 • 12

Slide 10.4.13

So, let's look at some unifiers of the expressions x and y . Since x and y are both variables, there are lots of things you can do to make them match.

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$

6.034 - Spring 03 • 13

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
{y/x}	x	x

6.034 - Spring 03 • 14

Slide 10.4.14

If you substitute x in for y , then both expressions come out to be x .

Slide 10.4.15

If you put in y for x , then they both come out to be y .

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
{y/x}	x	x
{x/y}	y	y

6.034 - Spring 03 • 15

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
{y/x}	x	x
{x/y}	y	y
{x/f(f(A)), y/f(f(A))}	f(f(A))	f(f(A))

6.034 - Spring 03 • 16

Slide 10.4.16

But you could also substitute something else, like $f(f(A))$ for x and for y , and you'd get matching expressions.

Most General Unifier

Slide 10.4.18

So, in fact, what we're really going to be looking for is not just any unifier of two expressions, but a most general unifier, or MGU.

6.034 - Spring 03 • 18

Slide 10.4.19

G is a most general unifier of ω_1 and ω_2 if and only if for all unifiers S, there exists an S-prime such that the result of applying G followed by S-prime to ω_1 is the same as the result of applying S to ω_1 ; and the result of applying G followed by S-prime to ω_2 is the same as the result of applying S to ω_2 .

A unifier is most general if every single one of the other unifiers can be expressed as an extra substitution added onto the most general one. An MGU is a substitution that you can make that makes the fewest commitments, and can still make these two expressions equal.

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
{y/x}	x	x
{x/y}	y	y
{x/f(f(A)), y/f(f(A))}	f(f(A))	f(f(A))
{x/A, y/A}	A	A

6.034 - Spring 03 • 17

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

6.034 - Spring 03 • 19

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$

6.034 - Spring 03 • 20

Slide 10.4.20

So, let's do a few examples together. What's a most general unifier of $P(x)$ and $P(A)$? A for x .



Slide 10.4.21

What about these two expressions? We can make them match up either by substituting x for y , or y for x . It doesn't matter which one we do. They're both "most general".

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$
$P(f(x), y, g(x))$	$P(f(x), x, g(x))$	$\{y/x\}$ or $\{x/y\}$

6.034 - Spring 03 • 21

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$
$P(f(x), y, g(x))$	$P(f(x), x, g(x))$	$\{y/x\}$ or $\{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$

6.034 - Spring 03 • 22

Slide 10.4.22

Okay. What about this one? It's a bit tricky. You can kind of see that, ultimately, all of the variables are going to have to be the same. Matching the arguments to g forces y and x to be the same. And since z and y have to be the same as well (to make the middle argument match), they all have to be the same variable. Might as well make it x (though it could be any other variable).

Slide 10.4.23

What about $P(x, B, B)$ and $P(A, y, z)$? It seems pretty clear that we're going to have to substitute A for x , B for y , and B for z .

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$
$P(f(x), y, g(x))$	$P(f(x), x, g(x))$	$\{y/x\}$ or $\{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$
$P(x, B, B)$	$P(A, y, z)$	$\{x/A, y/B, z/B\}$

6.034 - Spring 03 • 23

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$
$P(f(x), y, g(x))$	$P(f(x), x, g(x))$	$\{y/x\} \text{ or } \{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$
$P(x, B, B)$	$P(A, y, z)$	$\{x/A, y/B, z/B\}$
$P(g(f(v)), g(u))$	$P(x, x)$	$\{x/g(f(v)), u/f(v)\}$

6.034 - Spring 03 • 24

Slide 10.4.24

Here's a tricky one. It looks like x is going to have to simultaneously be $g(f(v))$ and $g(u)$. How can we make that work? By substituting $f(v)$ in for u .

Slide 10.4.25

Now, let's try unifying $P(x, f(x))$ with $P(x, x)$. The temptation is to say x has to be $f(x)$, but then that x has to be $f(x)$, etc. The answer is that these expressions are not unifiable.

The last time I explained this to a class, someone asked me what would happen if f were the identity function. Then, couldn't we unify these two expressions? That's a great question, and it illustrates a point I should have made before. In unification, we are interested in ways of making expressions equivalent, in every interpretation of the constant and function symbols. So, although it might be possible for the constants A and B to be equal because they both denote the same object in some interpretation, we can't unify them, because they aren't required to be the same in every interpretation.

Most General Unifier

g is a **most general unifier** of ω_1 and ω_2 iff for all unifiers s , there exists s' such that $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$
$P(f(x), y, g(x))$	$P(f(x), x, g(x))$	$\{y/x\} \text{ or } \{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$
$P(x, B, B)$	$P(A, y, z)$	$\{x/A, y/B, z/B\}$
$P(g(f(v)), g(u))$	$P(x, x)$	$\{x/g(f(v)), u/f(v)\}$
$P(x, f(x))$	$P(x, x)$	No MGU!

6.034 - Spring 03 • 25

Unification Algorithm

```
unify(Expr x, Expr y, Subst s) {
```

6.034 - Spring 03 • 26

Slide 10.4.26

An MGU can be computed recursively, given two expressions x and y , to be unified, and a substitution that contains substitutions that must already be made. The argument s will be empty in a top-level call to unify two expressions.

Slide 10.4.27

The algorithm returns a substitution if x and y are unifiable in the context of s , and fail otherwise. If s is already a failure, we return failure.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s) {
    if s = fail, return fail
```

6.034 - Spring 03 • 27

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
    if s = fail, return fail
    else if x = y, return s
```



6.034 - Spring 03 • 28

Slide 10.4.28

If x is equal to y, then we don't have to do any work and we return s, the substitution we were given.

Slide 10.4.29

If either x or y is a variable, then we go to a special subroutine that's shown in upcoming slides.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
    if s = fail, return fail
    else if x = y, return s
    else if x is a variable, return unify-var(x, y, s)
    else if y is a variable, return unify-var(y, x, s)
```

6.034 - Spring 03 • 29

**Unification Algorithm**

```
unify(Expr x, Expr y, Subst s){
    if s = fail, return fail
    else if x = y, return s
    else if x is a variable, return unify-var(x, y, s)
    else if y is a variable, return unify-var(y, x, s)
    else if x is a predicate or function application,
```



6.034 - Spring 03 • 30

Slide 10.4.30

If x is a predicate or a function application, then y must be one also, with the same predicate or function.

Slide 10.4.31

If so, we'll unify the lists of arguments from x and y in the context of s.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
    if s = fail, return fail
    else if x = y, return s
    else if x is a variable, return unify-var(x, y, s)
    else if y is a variable, return unify-var(y, x, s)
    else if x is a predicate or function application,
        if y has the same operator,
            return unify(args(x), args(y), s)
```



6.034 - Spring 03 • 31

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
    if s = fail, return fail
    else if x = y, return s
    else if x is a variable, return unify-var(x, y, s)
    else if y is a variable, return unify-var(y, x, s)
    else if x is a predicate or function application,
        if y has the same operator,
            return unify(args(x), args(y), s)
        else return fail
}
```



6.034 - Spring 03 • 32

Slide 10.4.32

If not, that is, if x and y have different predicate or function symbols, we simply fail.

Slide 10.4.33

Finally, (if we get to this case, then x and y are either lists of predicate or function arguments, or something malformed), we go down the lists, unifying the first elements, then the second elements, and so on. Each time we unify a pair of elements, we get a new substitution that records the commitments we had to make to get that pair of expressions to unify. Each further unification must take place in the context of the commitments generated by the previous elements of the lists.

Because, at each stage, we find the most general unifier, we make as few commitments as possible as we go along, and therefore we never have to back up and try a different substitution.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
    if s = fail, return fail
    else if x = y, return s
    else if x is a variable, return unify-var(x, y, s)
    else if y is a variable, return unify-var(y, x, s)
    else if x is a predicate or function application,
        if y has the same operator,
            return unify(args(x), args(y), s)
        else return fail
    else ; x and y have to be lists
        return unify(rest(x), rest(y),
                    unify(first(x), first(y), s))
}
```

6.034 - Spring 03 • 33



Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
```



6.034 - Spring 03 • 34

Slide 10.4.34

Given a variable var, an expression x, and a substitution s, we need to return a substitution that unifies var and x in the context of s. What makes this tricky is that we have to first keep applying the existing substitutions in s to var, and to x, if it is a variable, before we're down to a new concrete problem to solve.

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
    if var is bound to val in s,
        return unify(val, x, s)
```

6.034 - Spring 03 • 35



Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s) {
    if var is bound to val in s,
        return unify(val, x, s)
    else if x is bound to val in s,
        return unify-var(var, val, s)
```



6.034 - Spring 03 • 36

Slide 10.4.36

Similarly, if x is a variable, and it is bound to val in s, then we have to unify var with val in s. (We call unify-var directly, because we know that var is still a var).

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s) {
    if var is bound to val in s,
        return unify(val, x, s)
    else if x is bound to val in s,
        return unify-var(var, val, s)
    else if var occurs anywhere in (x s), return fail
```



6.034 - Spring 03 • 37

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s) {
    if var is bound to val in s,
        return unify(val, x, s)
    else if x is bound to val in s,
        return unify-var(var, val, s)
    else if var occurs anywhere in (x s), return fail
    else return add({var/x}, s)
}
```



6.034 - Spring 03 • 38

Slide 10.4.38

Finally, we know var is a variable that doesn't already have a substitution, so we add the substitution of x for var to s, and return it.

Some Examples

ω_1	ω_2	MGU
A(B, C)	A(x, y)	{x/B, y/C}
A(x, f(D,x))	A(E, f(D,y))	{x/E, y/E}
A(x, y)	A(f(C,y), z)	{x/f(C,y), y/z}
P(A, x, f(g(y)))	P(y, f(z), f(z))	{y/A, x/f(z), z/g(y)}
P(x, g(f(A)), f(x))	P(f(y), z, y)	none
P(x, f(y))	P(z, g(w))	none

Slide 10.4.39

Here are a few more examples of unifications, just so you can practice. If you don't see the answer immediately, try simulating the algorithm.



6.034 - Spring 03 • 39

6.034 Notes: Section 10.5

Slide 10.5.1

Now we know how to convert to clausal form and how to do unification. So now it's time to put it all together into first-order resolution.

Resolution with Variables

6.034 - Spring 03 * 1

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta \quad (\alpha \vee \beta)\theta}$$

6.034 - Spring 03 * 2

Slide 10.5.2

Here's the rule for first-order resolution. It says if you have a formula **alpha or phi** and another formula **not psi or beta**, and you can unify phi and psi with unifier theta, then you're allowed to conclude **alpha or beta** with the substitution theta applied to it.

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta \quad (\alpha \vee \beta)\theta}$$

$$\frac{P(x) \vee Q(x, y)}{\neg P(A) \vee R(B, z)}$$

$$\theta = \{x/A\}$$

6.034 - Spring 03 * 3

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta}$$

$$(\alpha \vee \beta)\theta$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta}$$

$$\theta = \{x/A\}$$



6.034 - Spring 03 * 4

Slide 10.5.4

So, we get rid of the P literals, and end up with $Q(x,y)$ or $R(B,z)$, but then we have to apply our substitution (the most general unifier that was necessary to make the literals match) to the result.

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta}$$

$$(\alpha \vee \beta)\theta$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta}$$

$$Q(A, y) \vee R(B, z)$$

$$\theta = \{x/A\}$$

6.034 - Spring 03 * 5

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta}$$

$$(\alpha \vee \beta)\theta$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, x)}{\neg P(A) \vee R(B, x)}$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta}$$

$$Q(A, y) \vee R(B, z)$$

$$\theta = \{x/A\}$$



6.034 - Spring 03 * 6

Slide 10.5.6

Now let's explore what happens if we have x's in the other formula. What if we replaced the z in the second sentence by an x?

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta}$$

$$(\alpha \vee \beta)\theta$$

$$\frac{\forall x, y. \quad P(x) \vee Q(x, y) \quad \forall z. \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta}$$

$$Q(A, y) \vee R(B, z)$$

$$\theta = \{x/A\}$$

6.034 - Spring 03 * 7

Resolution with Variables

$\alpha \vee \varphi$	MGU(φ, ψ) = θ	$\forall x, y. \quad P(x) \vee Q(x, y)$
$\neg\varphi \vee \beta$		$\forall x. \quad \neg P(A) \vee R(B, x)$
$(\alpha \vee \beta)\theta$		
$\forall x, y. \quad P(x) \vee Q(x, y)$		$P(x_1) \vee Q(x_1, y_1)$
$\forall z. \quad \neg P(A) \vee R(B, z)$		$\neg P(A) \vee R(B, x_2)$
$(Q(x, y) \vee R(B, z))\theta$		$(Q(x_1, y_1) \vee R(B, x_2))\theta$
$Q(A, y) \vee R(B, z)$		$Q(A, y_1) \vee R(B, x_2)$
$\theta = \{x_1/A\}$		

6.034 - Spring 03 * 8

Slide 10.5.8

So that means that before you try to do a resolution step, you're really supposed to rename the variables in the two sentences so that they don't share any variables in common. You won't usually need to do this that explicitly on your paper as you work through a proof, but if you were going to implement resolution in a computer program, or if you find yourself with the same variable in both sentences and it's getting confusing, then you should rename the sentences apart.

The easiest thing to do is to just go through and give every variable a new name. It's OK to do that. You just have to do it consistently for each clause. So you could rename to $P(x_1)$ or $Q(x_1, y_1)$, and you can name this one **not** $P(A)$ or $R(B, x_2)$. And then you could apply the resolution rule and you don't get into any trouble.

Slide 10.5.9

Okay. Now that we know how to do resolution, let's practice it on the example that we started in the section on clausal form. We want to prove that curiosity killed the cat.

Curiosity Killed the Cat

6.034 - Spring 03 * 9

Curiosity Killed the Cat

1	D(Fido)	a
2	O(j,Fido)	a
3	$\neg D(y) \vee \neg O(x, y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x, y)$	c
5	K(j,T) $\vee K(C, T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f

6.034 - Spring 03 * 10

Slide 10.5.10

Here are the clauses that we got from the original axioms.

Curiosity Killed the Cat

1	D(Fido)	a
2	O(j,Fido)	a
3	$\neg D(y) \vee \neg O(x, y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x, y)$	c
5	K(j,T) $\vee K(C, T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C, T)$	Neg

6.034 - Spring 03 * 11

Slide 10.5.11

Now we assert the negation of the thing we're trying to prove, so we have **not** $K(C, T)$.

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	S,8

6.034 - Spring 03 * 12

Slide 10.5.12

We can apply the resolution rule to any pair of lines that contain unifiable literals. Here's one way to do the proof. We'll use the "set-of-support" heuristic (which says we should involve the negation of the conclusion in the proof), and resolve away $K(C,T)$ from lines 5 and 8, yielding $J(T)$.

**Slide 10.5.13**

Then, we can resolve $C(T)$ and **not** $C(x)$ in lines 6 and 7 by substituting T for x, and getting $A(T)$.

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	S,8
10	A(T)	6,7 {x/T}

6.034 - Spring 03 * 13

**Curiosity Killed the Cat**

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	S,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}

6.034 - Spring 03 * 14

Slide 10.5.14

Using lines 4 and 9, and substituting J for x and T for y, we get **not** $L(J)$ or **not** $A(T)$.

**Slide 10.5.15**

From lines 10 and 11, we get **not** $L(J)$.

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	S,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11

6.034 - Spring 03 * 15



Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11
13	$\neg D(y) \vee \neg O(J,y)$	3,12 {x/J}

6.034 - Spring 03 * 16

Slide 10.5.16From 3 and 12, substituting J for x, we get **not D(y) or not O(J,y)**.**Slide 10.5.17**From 13 and 2, substituting Fido for x, we get **not D(Fido)**.**Curiosity Killed the Cat**

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11
13	$\neg D(y) \vee \neg O(J,y)$	3,12 {x/J}
14	$\neg D(Fido)$	13,2 {y/Fido}
15	*	14,1

6.034 - Spring 03 * 18

Slide 10.5.18

And finally, from lines 14 and 1, we derive a contradiction. Yay! Curiosity did kill the cat.

**Slide 10.5.19**

So, if we want to use resolution refutation to prove that something is valid, what would we do? What do we normally do when we do a proof using resolution refutation?

**Curiosity Killed the Cat**

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	K(J,T) $\vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11
13	$\neg D(y) \vee \neg O(J,y)$	3,12 {x/J}
14	$\neg D(Fido)$	13,2 {y/Fido}
15	*	14,1

6.034 - Spring 03 * 19

Proving validity

- How do we use resolution refutation to prove something is valid?



6.034 - Spring 03 * 19

Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms

Slide 10.5.20

We say, well, if I know all these things, I can prove this other thing I want to prove. We prove that the premises entail the conclusion.

6.034 - Spring 03 • 20

Slide 10.5.21

What does it mean for a sentence to be valid, in the language of entailment? That it's true in all interpretations. What that means really is that it should be derivable from nothing. A valid sentence is entailed by the empty set of sentences. The valid sentence is true no matter what. So we're going to prove something with no assumptions.

Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms
- Valid sentences are entailed by the empty set of sentences

6.034 - Spring 03 • 21

Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms
- Valid sentences are entailed by the empty set of sentences
- To prove validity by refutation, negate the sentence and try to derive contradiction.

Slide 10.5.22

We can prove it by resolution refutation by negating the sentence and trying to derive a contradiction.

6.034 - Spring 03 • 22

Slide 10.5.23

So, let's do an example. Imagine that we would like to show the validity of this sentence, which is a classical Aristotelian syllogism.

Proving validity: example

- Syllogism

$$(\forall x. P(x) \rightarrow Q(x)) \wedge P(A) \rightarrow Q(A)$$

6.034 - Spring 03 • 23

Proving validity: example

- Syllogism

$$(\forall x. P(x) \rightarrow Q(x)) \wedge P(A) \rightarrow Q(A)$$

- Negate and convert to clausal form

$$\neg((\forall x. P(x) \rightarrow Q(x)) \wedge P(A) \rightarrow Q(A))$$

$$\neg((\forall x. \neg P(x) \vee Q(x)) \vee \neg P(A) \vee Q(A))$$

$$(\forall x. \neg P(x) \vee Q(x)) \wedge P(A) \wedge \neg Q(A)$$

$$(\neg P(x) \vee Q(x)) \wedge P(A) \wedge \neg Q(A)$$

Slide 10.5.24

We start by negating it and converting to clausal form. We get rid of the arrows and drive in negations to arrive at this sentence in clausal form.

6.034 - Spring 03 • 24

Slide 10.5.25

We enter the clauses into our proof.

Proving validity: example

- Do proof

1.	$\neg P(x) \vee Q(x)$	
2.	$P(A)$	
3.	$\neg Q(A)$	
4.		
5.		

6.034 - Spring 03 • 25

Proving validity: example

- Do proof

1.	$\neg P(x) \vee Q(x)$	
2.	$P(A)$	
3.	$\neg Q(A)$	
4.	$Q(A)$	1,2
5.	■	3,4

6.034 - Spring 03 • 26

Slide 10.5.26

Now, we can resolve lines 1 and 2, substituting A for X, and get Q(A).

And we can resolve 3 and 4, to get a contradiction.

6.034 Notes: Section 10.6

Slide 10.6.1

In this section, we're going to look at three techniques for making logical proof more useful, and then conclude by talking about the limits of first-order logic.

Miscellaneous Logic Topics

- Factoring
- Green's trick
- Equality
- Completeness and decidability

6.034 - Spring 03 * 1

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete

6.034 - Spring 03 * 2

Slide 10.6.2

The version of the first-order resolution rule that we have shown you is called binary resolution because it involves two literals, one from each clause being resolved. It turns out that this form of resolution is not complete for first-order logic. There are sets of unsatisfiable clauses that will not generate a contradiction by successive applications of binary resolution.

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?
 $P(x) \vee P(y)$
 $\neg P(v) \vee \neg P(w)$

6.034 - Spring 03 * 3

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?
 $P(x) \vee P(y)$
 $\neg P(v) \vee \neg P(w)$
- We should!

6.034 - Spring 03 * 4

Slide 10.6.4

We should be able to! They are clearly unsatisfiable. There is no possible interpretation that will make both of these clauses simultaneously true, since that would require **P(x)** and **not (P x)** to be true for everything in the universe.

Slide 10.6.5

But when we apply binary resolution to these clauses, all we can get is something like $P(x)$ or not $P(w)$ (your variables may vary).

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$\begin{aligned} P(x) \vee P(y) \\ \neg P(v) \vee \neg P(w) \end{aligned}$$
- We should!
- But all we can get is $P(x) \vee \neg P(w)$

6.034 - Spring 03 * 5

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$\begin{aligned} P(x) \vee P(y) \\ \neg P(v) \vee \neg P(w) \end{aligned}$$
- We should!
- But all we can get is $P(x) \vee \neg P(w)$
- And from there all we can do is get back to one of the original clauses

6.034 - Spring 03 * 6

Slide 10.6.6

If we use binary resolution on this new clause with one of the parent clauses, we get back one of the parent clauses. We do not get a contradiction. So, we have shown by counterexample that binary resolution is not, in fact, a complete strategy.

Slide 10.6.7

It turns out that there is a simple extension of binary resolution that is complete. In that version, known as generalized resolution, we look for subsets of literals in one clause that can be unified with the negation of a subset of literals in the other clause. In our example from before, each P literal in one clause can be unified with its negation in the other clause.

Factoring

- Generalized resolution lets you resolve away multiple literals at once

6.034 - Spring 03 * 7

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

6.034 - Spring 03 * 8

Slide 10.6.8

An alternative to using generalized resolution is to introduce a new inference rule, in addition to binary resolution, called factoring. In factoring, if you can unify two literals within a single clause, alpha and beta in this case, with unifier theta, then you can drop one of them from the clause (it doesn't matter which one), and then apply the unifier to the whole clause.



Slide 10.6.9

So, for example, we can apply factoring to this sentence, by unifying P(x,y) and P(v,A).

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma \quad \theta = \text{MGU}(\alpha, \beta)}{(\alpha \vee \gamma)\theta}$$

- Example

$$\frac{}{Q(y) \vee P(x, y) \vee P(v, A)}$$

6.034 - Spring 03 • 9

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma \quad \theta = \text{MGU}(\alpha, \beta)}{(\alpha \vee \gamma)\theta}$$

- Example

$$\frac{Q(y) \vee P(x, y) \vee P(v, A)}{(Q(y) \vee P(x, y))\{x / v, y / A\} \\ Q(A) \vee P(v, A)}$$

6.034 - Spring 03 • 10

Slide 10.6.10

We get $Q(Y)$ or $P(x, Y)$, and then we have to apply the substitution $\{x/v, y/A\}$, which yields the result $Q(A)$ or $P(v, A)$.

Note that factoring in propositional logic is just removing duplicate literals from sentences, which is obvious and something we've been doing without comment.

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma \quad \theta = \text{MGU}(\alpha, \beta)}{(\alpha \vee \gamma)\theta}$$

- Example

$$\frac{Q(y) \vee P(x, y) \vee P(v, A)}{(Q(y) \vee P(x, y))\{x / v, y / A\} \\ Q(A) \vee P(v, A)}$$

- Binary resolution plus factoring is complete

6.034 - Spring 03 • 11

Green's Trick

- Use resolution to get answers to existential queries

Slide 10.6.12

One thing you can do with resolution is ask for an answer to a question. If your desired conclusion is that there exists an x such that $P(x)$, then in the course of doing the proof, we'll figure out what value of x makes $P(x)$ true. We might be interested in knowing that answer. For instance, it's possible to do a kind of planning via theorem proving, in which the desired conclusion is "There exists a sequence of actions such that, if I do them in my initial state, my goal will be true at the end." Generally, you're not just interested in whether such a sequence exists, but in what it is.

One way to deal with this is Green's trick, named after Cordell Green, who pioneered the use of logic in software engineering applications. We'll see it by example.

6.034 - Spring 03 • 12

Slide 10.6.13

Let's say we know that all men are mortal and that Socrates is a man. We want to know whether there are any mortals. So, our desired conclusion, negated and turned into clausal form would be **not Mortal(x)**. Green's trick will be to add a special extra literal onto that clause, of the form **Answer(x)**.

Green's Trick

- Use resolution to get answers to existential queries

Ex. $\exists x. \text{Mortal}(x)$

1.	$\neg \text{Man}(x) \vee \text{Mortal}(x)$	
2.	$\text{Man}(\text{Socrates})$	
3.	$\neg \text{Mortal}(x) \vee \text{Answer}(x)$	
4.		
5.		

6.034 - Spring 03 • 13

Green's Trick

- Use resolution to get answers to existential queries

Ex. $\exists x. \text{Mortal}(x)$

1.	$\neg \text{Man}(x) \vee \text{Mortal}(x)$	
2.	$\text{Man}(\text{Socrates})$	
3.	$\neg \text{Mortal}(x) \vee \text{Answer}(x)$	
4.	$\text{Mortal}(\text{Socrates})$	1,2
5.	$\text{Answer}(\text{Socrates})$	3,5

6.034 - Spring 03 • 14

Slide 10.6.14

Now, we do resolution as before, but when we come to a clause that contains only the answer literal, we stop. And whatever the variable x is bound to in that literal is our answer.

Slide 10.6.15

When we defined the language of first-order logic, we defined a special equality predicate. And we also defined special semantics for it (the sentence term1 equals term2 holds in an interpretation if and only if term1 and term2 both denote the same object in that interpretation). In order to do proofs that contain equality statements in them, we have to add a bit more mechanism.

Equality

- Special predicate in syntax and semantics; need to add something to our proof system

6.034 - Spring 03 • 15

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation

Slide 10.6.16

One strategy is to add one more special proof rule, just as we did with factoring. The new proof rule is called paramodulation. It's kind of hairy and hard to use, though, so we are not going to do it in this class (though it is implemented in most serious theorem provers).

6.034 - Spring 03 • 16

Slide 10.6.17

Another strategy, which is easier to understand, and instructive, is to treat equality almost like any other predicate, but to constrain its semantics via axioms.

Just to make it clear that Equals, which we will write as Eq, is a predicate that we're going to handle normally in resolution, we'll write it with a word rather than the equals sign.

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

6.034 - Spring 03 • 17

**Equality**

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\forall x.Eq(x, x)$$

6.034 - Spring 03 • 18

Slide 10.6.18

Equals has two important sets of properties. The first three say that it is an equivalence relation. First, it's symmetric: every x is equal to itself.

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\begin{aligned} \forall x.Eq(x, x) \\ \forall x, y.Eq(x, y) \rightarrow Eq(y, x) \end{aligned}$$

6.034 - Spring 03 • 19

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\begin{aligned} \forall x.Eq(x, x) \\ \forall x, y.Eq(x, y) \rightarrow Eq(y, x) \\ \forall x, y, z.Eq(x, y) \wedge Eq(y, z) \rightarrow Eq(x, z) \end{aligned}$$

6.034 - Spring 03 • 20

Slide 10.6.20

Third, it's transitive. That means that if x equals y and y equals z, then x equals z.



Slide 10.6.21

The other thing we need is the ability to "substitute equals for equals" into any place in any predicate. That means that, for each place in each predicate, we'll need an axiom that looks like this: for all x and y, if x equals y, then if P holds of x, it holds of y.

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\begin{aligned} \forall x. Eq(x, x) \\ \forall x, y. Eq(x, y) \rightarrow Eq(y, x) \\ \forall x, y, z. Eq(x, y) \wedge Eq(y, z) \rightarrow Eq(x, z) \end{aligned}$$

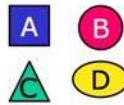
- For every predicate, allow substitutions

$$\forall x, y. Eq(x, y) \rightarrow (P(x) \rightarrow P(y))$$

6.034 - Spring 03 • 21

Proof Example

- Let's go back to our old geometry domain and try to prove what the hat of A is



6.034 - Spring 03 • 22

Slide 10.6.22

Let's go back to our old geometry domain and try to prove what the hat of A is.

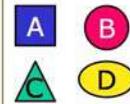
Slide 10.6.23

We know that these axioms (our old KB4) entail that $\text{hat}(A) = A$. We'll have to add in the equality axioms, as well.

Proof Example

- Let's go back to our old geometry domain and try to prove what the hat of A is
- Axioms in FOL (plus equality axioms)

$$\begin{aligned} &\text{Above}(A, C) \\ &\text{Above}(B, D) \\ &\neg \exists x. \text{Above}(x, A) \\ &\neg \exists x. \text{Above}(x, B) \\ &\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x \\ &\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x \end{aligned}$$



6.034 - Spring 03 • 23

Proof Example

- Let's go back to our old geometry domain and try to prove what the hat of A is
- Axioms in FOL (plus equality axioms)

$$\begin{aligned} &\text{Above}(A, C) \\ &\text{Above}(B, D) \\ &\neg \exists x. \text{Above}(x, A) \\ &\neg \exists x. \text{Above}(x, B) \\ &\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x \\ &\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x \end{aligned}$$



6.034 - Spring 03 • 24

Slide 10.6.24

Let's see if we can derive that, using resolution refutation and Green's trick.

Slide 10.6.25

Here's the result of my clausal-form converter run on those axioms.

The Clauses

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.		
11.		
12.		

6.034 - Spring 03 • 25

**The Query**

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(A), x) v Answer(x)	

6.034 - Spring 03 • 26

**Slide 10.6.26**

Now, our goal is to prove **exists x such that Eq(hat(A),x)**. That is negated and turned into clausal form, yielding **not Eq(hat(A),x)**. And we add in the answer literal, so we can keep track of what the answer is.

The Proof

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(A), x) v Answer(x)	conclusion
11.	Above(sk(A), A) v Answer(A)	6, 10 {x/A}
12.	Answer(A)	11, 3 {x/sk(A)}

6.034 - Spring 03 • 27

**Hat of D**

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(D), x) v Answer(x)	conclusion
11.	~Above(x,D) v Answer(x)	5, 10 {x1/x}
12.	Answer(B)	11, 2 {x/B}

6.034 - Spring 03 • 28

**Slide 10.6.28**

What if we wanted to use the same axioms to figure out what the hat of D is? We just change our query and do the proof. Here it is.

Slide 10.6.29

Here's a worked example of a problem with equality.

Who is Jane's Lover?

- Jane's lover drives a red car
- Fred is the only person who drives a red car
- Who is Jane's lover?

1.	Drives(lover(Jane))	
2.	\sim Drives(x) \vee Eq(x,Fred)	
3.	\sim Eq(lover(Jane),x) \vee Answer(x)	
4.	Eq(lover(Jane), Fred)	1,2 {x/lover(Jane)}
5.	Answer(Fred)	3,4 {x/Fred}

6.034 - Spring 03 • 29

**Completeness and Decidability**

- Complete: If KB entails S, then we can prove S from KB

**Slide 10.6.30**

Now, let's see what we can say, in general, about proof in first-order logic.

Remember that a proof system is complete, if, whenever the KB entails S, we can prove S from KB.



6.034 - Spring 03 • 30

Slide 10.6.31

In 1929, Gödel proved a completeness theorem for first-order logic: There exists a complete proof system for FOL. But, living up to his nature as a very abstract logician, he didn't come up with such a proof system; he just proved one existed.

**Completeness and Decidability**

- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*

6.034 - Spring 03 • 31

**Slide 10.6.32**

Then, in 1965, Robinson came along and showed that resolution refutation is sound and complete for FOL.

**Completeness and Decidability**

- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*
- Robinson's Completeness Theorem: *Resolution refutation is a complete proof system for FOL*



6.034 - Spring 03 • 32

Slide 10.6.33

So, we know that if a proof exists, then we can eventually find it with resolution. Unfortunately, we no longer know, as we did in propositional resolution, that eventually the process will stop. So, it's possible that there is no proof, and that the resolution process will run forever.

This makes first-order logic what is known as "semi-decidable". If the answer is "yes", that is, if there is a proof, then the theorem prover will eventually halt and say so. But if there isn't a proof, it might run forever!

Completeness and Decidability

- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*
- Robinson's Completeness Theorem: *Resolution refutation is a complete proof system for FOL*
- *FOL is semi-decidable:* if the desired conclusion follows from the premises then eventually resolution refutation will find a contradiction.
 - If there's a proof, we'll halt with it
 - If not, maybe we'll halt, maybe not

6.034 - Spring 03 • 33

Adding Arithmetic**Slide 10.6.34**

So, things are relatively good with regular first-order logic. And they're still fine if you add addition to the language, allowing statements like $P(x)$ and $(x + 2 = 3)$. But if you add addition and multiplication, it starts to get weird!

6.034 - Spring 03 • 34

Slide 10.6.35

In 1931, Gödel proved an incompleteness theorem, which says that there is no consistent, complete proof system for FOL plus arithmetic. (Consistent is the same as sound.) Either there are sentences that are true, but not provable, or there are sentences that are provable, but not true. It's not so good either way.

Adding Arithmetic

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.

6.034 - Spring 03 • 35

Adding Arithmetic**Slide 10.6.36**

Here's the roughest cartoon of how the proof goes. Arithmetic gives you the ability to construct code names for sentences within the logic, and therefore to construct sentences that are self-referential. This sentence, P, is sometimes called the Gödel-sentence. P is "P is not provable".

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.
- Arithmetic gives you the ability to construct code-names for sentences within the logic.
P = "P is not provable."

6.034 - Spring 03 • 36

Slide 10.6.37

If P is true, then P is not provable (so the system is incomplete). If P is false, then P is provable (so the system is inconsistent).

This result was a huge blow to the current work on the foundations of mathematics, where they were, essentially, trying to formalize all of mathematical reasoning in first-order logic. And it pre-figured, in some sense, Turing's work on uncomputability.

Ultimately, though, just as uncomputability doesn't worry people who use computer programs for practical applications, incompleteness shouldn't worry practical users of logic.

Adding Arithmetic

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.
- Arithmetic gives you the ability to construct code-names for sentences within the logic.
 - $P = "P \text{ is not provable.}"$
 - If P is true: it's not provable (incomplete)
 - If P is false: it's provable (inconsistent)

6.034 – Spring 03 • 37

6.034 Notes: Section 10.7**Slide 10.7.1**

Now that we've studied the syntax and semantics of logic, and know something about how to do inference in it, we're going to talk about how logic has been applied in real domains, and look at an extended example.

Logic in the Real World

6.034 – Spring 03 • 1

Logic in the Real World

- Encode information formally in web pages

Slide 10.7.2

There is currently a big resurgence of logical representations and inference in the context of the web. As it stands now, web pages were written in natural language (English or French, etc), by the people and for the people. But there is an increasing desire to have computer programs (web agents or 'bots) crawl the web and figure things out by "reading" web pages. As we'll see in the next module of this course, it can be quite hard to extract the meaning from text written in natural language. So the World-Wide Web Consortium, in conjunction with people in universities and industry, are defining a standard language, which is essentially first-order logic, for formally encoding information in web pages. Information that is written in this formal language will be much easier to extract automatically.

6.034 – Spring 03 • 2

Slide 10.7.3

It is becoming more appealing, in business, to have computers talk directly to one another, and to leave humans out of the loop. One place this can happen is in negotiating simple contracts between companies to deliver goods at some price. Benjamin Grosof, who is a professor in the Sloan School, works on using non-monotonic logic (a version of first-order logic, in which you're allowed to have conflicting rules, and have a system for deciding which ones have priority) to specify a company's business rules.

Logic in the Real World

- Encode information formally in web pages
- Business rules

6.034 - Spring 03 * 3

Logic in the Real World

- Encode information formally in web pages
- Business rules
- Airfare pricing

6.034 - Spring 03 * 4

Slide 10.7.4

Another example, which we'll pursue in detail, is the language the airlines use to specify the rules on their airfares. It turns out that every day, many times a day, airlines revise and publish (electronically) their fare structures. And, as many of you know, the rules governing the pricing of airplane tickets are pretty complicated, and certainly unintuitive. In fact, they're so complicated that the airlines had to develop a formal language that is similar to logic, in order to describe their different kinds of fares and the restrictions on them.

Amazingly, there are on the order of 20 million different fares! To generate a price for a particular proposed itinerary, it requires piecing together a set of fares to cover the parts of the itinerary. Typically, the goal is to find the cheapest such set of fares, subject to some constraints.

Slide 10.7.5

We're not going to worry about how to do the search to find the cheapest itinerary and fare structure (that's a really hard and interesting search problem!). Instead, we'll just think about pricing a particular itinerary.

Pricing an airline ticket is not as simple as adding up the prices for the individual flight legs. There are many different pricing schemes, each depending on particular attributes of the combination of flights that the passenger proposes to take. For instance, at some point in 1998, American Airlines had 29 different fares for going from Boston to San Francisco, ranging in price from \$1943 to \$231, each with a different constraint on its use.

In this discussion, we won't get into the actual ticket prices; instead we'll work on writing down the logical expressions that describe when a particular fare applies to a proposed itinerary.

Airfare Pricing

- Ignore, for now, finding the best itinerary
- Given an itinerary, what's the least amount we can pay for it?
- Can't just add up prices for the flight legs; different prices for different flights in various combinations and circumstances

Image removed due to copyright restrictions.

6.034 - Spring 03 * 5

Fare Restrictions

- Passenger under 2 or over 65
- Passenger accompanying someone paying full fare
- Doesn't go through an expensive city
- No flights during rush hour
- Stay over Saturday night
- Layovers are legal
- Round-the-world itinerary that doesn't backtrack
- Regular two phase round-trip
- No flights on another airline
- This fare would not be cheaper than the standard price

6.034 - Spring 03 * 6

Slide 10.7.6

Here are some examples of airfare restrictions that we might want to encode logically:

- The passenger is under 2 or over 65
- The passenger is accompanying another passenger who is paying full fare
- It doesn't go through an expensive city
- There are no flights during rush hour (defined in local time)
- The itinerary stays over a Saturday night
- Layovers are legal: not too short; not too long
- Round-the-world itinerary that doesn't backtrack
- The itinerary is a regular two-trip round-trip
- This price applies to one flight, as long as there is no other flight in this itinerary operated by El Cheapo Air
- If the sum of the fares for a circle trip with three legs is less than the "comparable" round-trip price between the origin and any stopover point, then you must add a surcharge.

Slide 10.7.7

The first step in making a logical formalization of a domain is coming up with an *ontology*. According to Leibniz (a philosopher from the 17th century), ontology is "the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident." Whoa! I wish our lecture notes sounded that deep.

Ontology

Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

Image of Leibniz removed due to copyright restrictions.

Leibniz

6.034 – Spring 03 • 7

Ontology

Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

The Role of Ontological Engineering in B2B Net Markets

Images removed due to copyright restrictions.

Leibniz

6.034 – Spring 03 • 8

Slide 10.7.8

Now there are web sites with paper titles like "The Role of Ontological Engineering in B2B Net Markets". That's just as scary.

Slide 10.7.9

For us, more prosaically, an ontology will be a description of the kinds of objects that you have in your world and their possible properties and relations. Making up an ontology is a lot like deciding what classes and methods you'll need when you design an object-oriented program.

Ontology

- What kinds of things are there in the world?
- What are their properties and relations?

Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

The Role of Ontological Engineering in B2B Net Markets

Images removed due to copyright restrictions.

Leibniz

6.034 – Spring 03 • 9

Airfare Domain Ontology**Slide 10.7.10**

Okay. So what are the kinds of things we have in the airfare domain? That's a hard question, because it depends on the level of abstraction at which we want to make our model. Probably we don't want to talk about particular people or airplanes; but we might need to talk about people in general, in terms of various properties (their ages, for example, but not their marital status), or airplane types. We will need a certain amount of detail though, so, for instance, it might matter which airport within a city you're using, or which terminal within an airport. Often you have to adjust the level of abstraction that you use as you go along.



6.034 – Spring 03 • 10

Slide 10.7.11

Here's a list of the relevant object types I came up with:

- passenger
- flight
- city
- airport
- terminal
- flight segment (a list of flights, to be flown all in one "day")
- itinerary (a passenger and a list of flight segments)

Airfare Domain Ontology

- passenger
- flight
- city
- airport
- terminal
- flight segment (list of flights, to be flown all in one "day")
- itinerary (a passenger and list of flight segments)

6.034 - Spring 03 • 11

Airfare Domain Ontology

- passenger
- flight
- city
- airport
- terminal
- flight segment (list of flights, to be flown all in one "day")
- itinerary (a passenger and list of flight segments)
- list
- number

6.034 - Spring 03 • 12

Slide 10.7.12

We'll also need some non-concrete object types, including

- list
- number

Slide 10.7.13

Once we know what kinds of things we have in our world, we need to come up with a vocabulary of constant names, predicate symbols, and function symbols that we'll use to talk about their properties and relations.

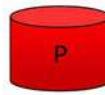
There are two parts to this problem. We have to decide what properties and relations we want to be able to represent, and then we have to decide how to represent them.

Representing Properties

6.034 - Spring 03 • 13

Representing Properties

- Object P is red
 - $\text{Red}(P)$
 - $\text{Color}(P, \text{Red})$
 - $\text{color}(P) = \text{Red}$
 - $\text{Property}(P, \text{Color}, \text{Red})$



6.034 - Spring 03 • 14

Slide 10.7.14

Let's talk, for a minute, about something simple, like saying that an object named P is red. There are a number of ways to say this, including:

- $\text{Red}(P)$
- $\text{Color}(P, \text{Red})$
- $\text{color}(P) = \text{Red}$
- $\text{Property}(P, \text{Color}, \text{Red})$

Slide 10.7.15

Let's look at the difference between the first two. $\text{Red}(P)$ seems like the most straightforward way to say that P is red. But what if we wanted to write a rule saying that all the blocks in a particular stack, S , are the same color? Using the second representation, we could say:

```
exists c. all b. In(b,S) -> Color(b, c)
```

In this case, we have *reified* redness; that is, we've made it into an object that can be named and quantified over. It will turn out that it's often useful to use this kind of representation.

Representing Properties

- Object P is red
 - $\text{Red}(P)$
 - $\text{Color}(P, \text{Red})$
 - $\text{color}(P) = \text{Red}$
 - $\text{Property}(P, \text{Color}, \text{Red})$



- All the blocks in stack S are the same color

$$\exists c. \forall b. \text{In}(b, S) \rightarrow \text{Color}(b, c)$$

6.034 - Spring 03 • 15

Representing Properties

- Object P is red
 - $\text{Red}(P)$
 - $\text{Color}(P, \text{Red})$
 - $\text{color}(P) = \text{Red}$
 - $\text{Property}(P, \text{Color}, \text{Red})$
- All the blocks in stack S are the same color

$$\exists c. \forall b. \text{In}(b, S) \rightarrow \text{Color}(b, c)$$
- All the blocks in stack S have the same properties

$$\forall p. \exists v. \forall b. \text{In}(b, S) \rightarrow \text{Property}(b, p, v)$$



6.034 - Spring 03 • 16

Slide 10.7.16

It's possible to go even farther down this road, in case, for instance, we wanted to say that all the blocks in stack S have all the same properties:

$$\text{all } p. \text{ exists } v. \text{ all } b. \text{ In}(b, S) \rightarrow \text{Property}(b, p, v)$$

That is, for every property, there's a value, such that every block in S has that value for the property.

Some people advocate writing all logical specifications using this kind of formalization, because it is very general; but it's also kind of hard to read. We'll stick to representations closer to $\text{Color}(P, \text{Red})$.

Basic Relations

6.034 - Spring 03 • 17

Basic Relations

- $\text{Age}(\text{passenger}, \text{number})$
- $\text{Nationality}(\text{passenger}, \text{country})$
- $\text{Wheelchair}(\text{passenger})$
- $\text{Origin}(\text{flight}, \text{airport})$
- $\text{Destination}(\text{flight}, \text{airport})$
- $\text{Departure_Time}(\text{flight}, \text{number})$
- $\text{Arrival_Time}(\text{flight}, \text{number})$
- $\text{Latitude}(\text{city}, \text{number})$
- $\text{Longitude}(\text{city}, \text{number})$
- $\text{In_Country}(\text{city}, \text{country})$
- $\text{In_City}(\text{airport}, \text{city})$
- $\text{Passenger}(\text{itinerary}, \text{passenger})$
- $\text{Flight_Segments}(\text{itinerary}, \text{passenger}, \text{segments})$
- Nil
- $\text{cons}(\text{object}, \text{list}) \Rightarrow \text{list}$

6.034 - Spring 03 • 18

Slide 10.7.18

Here are some of the basic relations in our domain. I've named the arguments with the types of objects that we expect to be there. This is just an informal convention to show you how we intend to use the relations. (There are logics that are strongly typed, which require you to declare the types of the arguments to each relation or function).

Slide 10.7.19

So, we might describe passenger Fred using the sentences

```
Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(Fred)
```

This only serves to encode a very simple version of this domain. We haven't started to talk about time zones, terminals, metropolitan areas (usually it's okay to fly into San Jose and then out of San Francisco, as if they were the same city), airplane types, how many reservations a flight currently has, and so on and so on.

Basic Relations

- Age(passenger, number)
- Nationality(passenger, country)
- Wheelchair(passenger)
- Origin(flight, airport)
- Destination(flight, airport)
- Departure_Time(flight, number)
- Arrival_Time(flight, number)
- Latitude(city, number)
- Longitude(city, number)
- In_Country(city, country)
- In_City(airport, city)
- Passenger(itinerary, passenger)
- Flight_Segments(itinerary, passenger, segments)
- Nil
- cons(object, list) => list

```
Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(Fred)
```

6.034 - Spring 03 • 19

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i. P(i) \wedge Q(i) \rightarrow \text{Qualifies } 37(i)$$

Slide 10.7.20

Other relations will be used to express the things we want to infer. An example in this domain might be `Qualifies_for_fare_class_37` or something else equally intuitive. As we begin trying to write down a logical expression for `Qualifies_for_fare_class_37` in terms of the basic relations, we'll find that we want to define more intermediate relations. It will be exactly analogous to defining functions when writing a Scheme program: it's not strictly necessary, but no-one would ever be able to read your program (and you probably wouldn't be able to write it correctly) if you didn't.

6.034 - Spring 03 • 20

Slide 10.7.21

We will often define relations using implications rather than equivalence. It makes it easier to add additional pieces of the definition (circumstances in which the relation would be true).

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i. P(i) \wedge Q(i) \rightarrow \text{Qualifies } 37(i)$$

- Implication rather than equivalence
 - easier to specify definitions in pieces

$$\forall i. R(i) \wedge S(i) \rightarrow \text{Qualifies } 37(i)$$

6.034 - Spring 03 • 21

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i. P(i) \wedge Q(i) \rightarrow \text{Qualifies } 37(i)$$

- Implication rather than equivalence

- easier to specify definitions in pieces

$$\forall i. R(i) \wedge S(i) \rightarrow \text{Qualifies } 37(i)$$

- can't use the other direction

$$\text{Qualifies } 37(i) \rightarrow ?$$

- if you need it, write the equivalence

$$\forall i. (P(i) \wedge Q(i)) \vee (R(i) \wedge S(i)) \leftrightarrow \text{Qualifies } 37(i)$$

Slide 10.7.22

However, written this way, we can't infer anything from knowing that the relation holds of some objects. If we need to do that, we have to write out the equivalence.

6.034 - Spring 03 • 22

Slide 10.7.23

Okay. Let's start with a very simple rule. Let's say that an itinerary has the *Infant_Fare* property if the passenger is under age 2. We can write that as

```
all i, a, p. Passenger(i,p) ^ Age(p,a) ^ a < 2 -> Infant_Fare(i)
```

Infant Fare

$$\forall i, a, p. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

6.034 - Spring 03 • 23

Infant Fare

$$\forall i, a, p. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$
Slide 10.7.24

It's not completely obvious that this is the right way to write the rule. For instance, it's useful to note that this is equivalent to saying

```
all i. (exists a, p. Passenger(i,p) ^ Age(p,a) ^ a < 2)
      -> Infant_Fare(i)
```

6.034 - Spring 03 • 24

Slide 10.7.25

This second form is clearer (though the first form is closer to what we'll use next, when we talk about rule-based systems). Note, also, that changing the implication to equivalence in these two statements makes them no longer be equivalent.

Infant Fare

$$\forall i, a, p. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence

6.034 - Spring 03 • 25

Infant Fare

$$\forall i, a, p. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence
- What about $a < 2$?

Slide 10.7.26

We just snuck something in here: $a < 2$. We'll need some basic arithmetic in almost any interesting logic domain. How can we deal with that? We saw, in the previous section, that adding arithmetic including multiplication means that our language is no longer complete. But that's the sort of thing that worries logicians more than practitioners.

6.034 - Spring 03 • 26

Slide 10.7.27

In this domain we'll need addition and subtraction and greater-than. One strategy would be to axiomatize them in logic, but that's usually wildly inefficient. Most systems for doing practical logical inference include built-in arithmetic predicates that will automatically evaluate themselves if their arguments are instantiated. So if, during the course of a resolution proof, you had a clause of the form

$$P(a) \vee (3 < 2) \vee (a > 1 + 2)$$

it would automatically simplify to

$$P(a) \vee (a > 3)$$

Infant Fare

$$\begin{aligned} \forall i, a, p. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i) \\ \forall i (\exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i) \end{aligned}$$

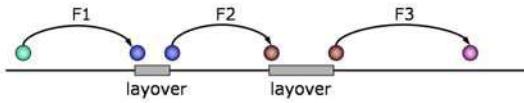
- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence

- What about $a < 2$?

- axiomatize arithmetic
- build it in to theorem prover

$$P(a) \vee (3 > 2) \vee (a > 1 + 2) \Rightarrow P(a) \vee (a > 3)$$

6.034 - Spring 03 • 27

**Well-Formed Segment**

6.034 - Spring 03 • 28

Slide 10.7.28

Okay. Now, let's go to a significantly harder one. This isn't exactly a fare restriction; it's more of a correctness criterion on the flights within the itinerary. The idea is that an itinerary can be made up of multiple flight segments; each flight segment might, itself, be made up of multiple flights. In order for the itinerary to be well-formed, the flight segments are not required to have any particular relation to each other. However, there are considerable restrictions on a flight segment. One way to think about a flight segment is as a sequence of flights that you would do in one day (though it might actually last for more than one day if you're going for a long time).

Slide 10.7.29

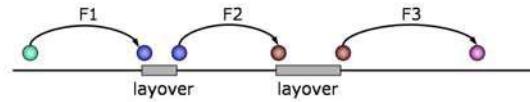
For a flight segment to be well-formed, it has to satisfy the following properties:

- The departure and arrival airports match up correctly
- The layovers (gaps between arriving in an airport and departing from it) aren't too short (so that there's a reasonable probability that the passenger will not miss the connection)
- The layovers aren't too long (so that the passenger can't spend a week enjoying him or herself in the city; we need to be sure to charge extra for that!)

So, let's work toward developing a logical specification of the well-formedness of a flight segment. A flight segment is a list of flights. So, we'll make a short detour to talk about lists in logic, then come back to well-formed flight-segments.

Well-Formed Segment

- Departure and arrival airports match up correctly
- Layovers aren't too short
- Layovers aren't too long



6.034 - Spring 03 • 29

Lists in Logic

- Nil : constant
- cons : function

Slide 10.7.30

In the list of relations for the domain, we included a constant Nil and a function cons, without explanation. Here's the explanation.

6.034 - Spring 03 • 30

Slide 10.7.31

We can make and use lists in logic, much as we might do in Scheme. We have a constant that stands for the empty list. Then, we have a function cons that, given any object and a list, denotes the list that has the object argument as its head (car) and the list argument as its tail (cdr).

So $\text{cons}(\text{A}, \text{cons}(\text{B}, \text{Nil}))$ is a list with two elements, A and B.

Lists in Logic

- Nil : constant
- cons : function
- $\text{cons}(\text{A}, \text{cons}(\text{B}, \text{Nil}))$: list with two elements

6.034 - Spring 03 • 31

Lists in Logic

- Nil : constant
 - cons : function
 - $\text{cons}(\text{A}, \text{cons}(\text{B}, \text{Nil}))$: list with two elements
 - $\forall x. \text{LengthOne}(\text{cons}(x, \text{Nil}))$
- $$\forall l, x. l = \text{cons}(x, \text{Nil}) \rightarrow \text{LengthOne}(l)$$
- $$\forall l. (\exists x. l = \text{cons}(x, \text{Nil})) \rightarrow \text{LengthOne}(l)$$

6.034 - Spring 03 • 32

Slide 10.7.32

We can also use the power of unification to specify conditions to assertions. So we can write **for all x lengthOne(cons(x,Nil))**, which is a more compact way of saying that every list that is equal to the cons of an element onto Nil has the property of being lengthOne.

Slide 10.7.33

Now that we know how to do some things with lists, we'll go back to the problem of ensuring that a flight segment is well-formed. This is basically a condition on all the layovers in the segment, so we'll have to run down the list making sure it's all okay.

Well-Formed Segment: Base Case

Define recursively, going down the list of flights

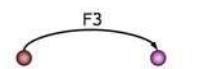
6.034 - Spring 03 • 33

Well-Formed Segment: Base Case

Define recursively, going down the list of flights

Any segment with 1 flight is well-formed

$\forall f. \text{WellFormed}(\text{cons}(f, \text{Nil}))$

**Slide 10.7.34**

We can start with a simple case. If the flight segment has one flight, then it's well-formed. We can write this as **all f. WellFormed(cons(f, Nil))**.

6.034 - Spring 03 • 34

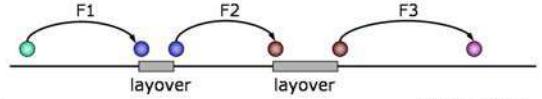
Slide 10.7.35

Now, let's do the hard case. We can say that a flight segment with more than one flight is well-formed if the first two flights are contiguous (end and start in the same airport), the layover time between the first two flights is legal, and the rest of the flight segment is well-formed.

Well-Formed Segment: Recursion

A flight segment with at least two flights is well-formed if

- first two flights are contiguous
- layover time between first two flights is legal
- rest of the flight segment is well-formed



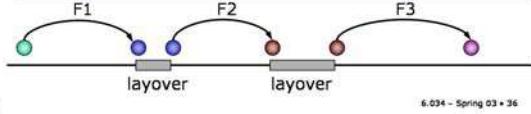
6.034 - Spring 03 • 35

Well-Formed Segment: Recursion

A flight segment with at least two flights is well-formed if

- first two flights are contiguous
- layover time between first two flights is legal
- rest of the flight segment is well-formed

$$\forall f_1, f_2, r. \text{Contiguous}(f_1, f_2) \wedge \text{LegalLayover}(f_1, f_2) \wedge \\ \text{WellFormed}(\text{cons}(f_2, r)) \rightarrow \text{WellFormed}(\text{cons}(f_1, \text{cons}(f_2, r)))$$



6.034 - Spring 03 • 36

Slide 10.7.36

In logic, that becomes

```
all f1, f2, r. Contiguous(f1, f2) ^ LegalLayover(f1, f2) ^ \\ WellFormed(cons(f2, r)) -> WellFormed(cons(f1, cons(f2, r)))
```

Slide 10.7.37

Note that we've invented some vocabulary here. Contiguous and LegalLayover are neither given to us as basic relations, nor the relation we are trying to define. We made them up, just as you make up function names, in order divide our problem into conquerable sub-parts.

Helper Relations

6.034 - Spring 03 • 37

Helper Relations

- Flights are contiguous if the arrival airport of the first is the same as the departure airport of the second

$$\forall f_1, f_2. (\exists c. \text{Destination}(f_1, c) \wedge \text{Origin}(f_2, c)) \rightarrow \text{Contiguous}(f_1, f_2)$$

Slide 10.7.38

What makes two flights contiguous? The arrival airport of the first has to be the same as the departure airport of the second. We can write this as

```
all f1, f2. (\exists c. \text{Destination}(f1, c) ^ \text{Origin}(f2, c)) \\ -> \text{Contiguous}(f1, f2)
```

6.034 - Spring 03 • 38

Slide 10.7.39

Now, what makes layovers legal? They have to be not too short and not too long.

```
all f1, f2.  
LayoverNotTooShort(f1, f2) ^ LayoverNotTooLong(f1, f2) ->  
    LayoverLegal(f1, f2)
```

Helper Relations

- Flights are contiguous if the arrival airport of the first is the same as the departure airport of the second

$$\forall f_1, f_2. (\exists c. \text{Destination}(f_1, c) \wedge \text{Origin}(f_2, c)) \rightarrow \text{Contiguous}(f_1, f_2)$$

- Layovers are legal if they're not too short and not too long

$$\forall f_1, f_2. \text{LayoverNotTooShort}(f_1, f_2) \wedge \text{LayoverNotTooLong}(f_1, f_2) \rightarrow \text{LayoverLegal}(f_1, f_2)$$

6.034 - Spring 03 • 39

Not Too Short

- A layover is not too short if it's more than 30 minutes long

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \wedge (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)$$

Slide 10.7.40

Let's say that passengers need at least 30 minutes to change planes.

That comes out fairly straightforwardly as

```
all f1, f2. (exists t1, t2.  
    \text{Arrival\_Time}(f1, t1) \wedge \text{Departure\_Time}(f2, t2) \wedge (t2 - t1 >  
    30)) ->  
        \text{Layover\_Not\_Too\_Short}(f1, f2)
```

6.034 - Spring 03 • 40

Slide 10.7.41

This is a very simple version of the problem. You could imagine making this incredibly complex and nuanced. How long does it take someone to change planes? It might depend on: whether they're in a wheelchair, whether they have to change terminals, how busy the terminals are, how effective the inter-terminal transportation is, whether they have small children, whether the airport has signs in their native language, whether there's bad weather, how long the lines are at security, whether they're from a country whose citizens take a long time to clear immigration.

You probably wouldn't want to add each of these things as a condition in the rule about layovers. Rather, you would want this system, ultimately, to be connected to a knowledge base of common sense facts and relationships, which could be used to deduce an expected time to make the connection. Common-sense reasoning is a fascinating area of AI with a long history. It seems to be (like many things!) both very important and very hard.

Not Too Short

- A layover is not too short if it's more than 30 minutes long

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \wedge (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)$$

- These are like the rules the airlines use, but it could involve all of common sense to know how long to allow someone to change planes

6.034 - Spring 03 • 41

Not Too Long

- A layover is not too long if it's less than three hours

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \wedge (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

Slide 10.7.42

We'll continue in our more circumscribed setting, to address the question of what makes a layover not be too long. This will have an easy case and a hard case. The easy case is that a layover is not too long if it's less than three hours:

```
all f1, f2. (exists t1, t2.  
    \text{ArrivalTime}(f1, t1) \wedge \text{DepartureTime}(f2, t2) \wedge  
    (t2 - t1 < 180)) -> \text{LayoverNotTooLong}(f1, f2)
```

6.034 - Spring 03 • 42

Slide 10.7.43

Now, for the hard case. Let's imagine you've just flown into Ulan Bator, trying to get to Paris. And there's only one flight per day from Ulan Bator. We might want to say that your layover is not too long if there are no flights from here to your next destination that go before the one you're scheduled to take (and that have an adequately long layover).

```
all f1, f2 (exists o, d, t2.
    Origin(f2, o) ^ Destination(f2, d) ^ DepartureTime(f2, t2) ^
    ~ exists f3, t3. (Origin(f3, o) ^ Destination(f3, d) ^
    ~ exists o, d, t2. (Origin(f2, o) ^ Destination(f2, d) ^
    DepartureTime(f2, t2) ^ (t2 < t3) ^
    ~ LayoverNotTooShort(f2, f3))) ->
LayoverNotTooLong(f1, f2)
```

Of course, you can imagine all sorts of common-sense information that might influence this definition of LayoverNotTooLong, just as in the previous case.

Not Too Long

- A layover is not too long if it's less than three hours

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \wedge (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

- A layover is also not too long if there was no other way to make the next leg of your journey sooner

$$\begin{aligned} \forall f_1, f_2. (\exists o, d, t_2. \text{Origin}(f_2, o) \wedge \text{Destination}(f_2, d) \wedge \text{DepartureTime}(f_2, t_2) \wedge \\ \neg \exists f_3, t_3. (\text{Origin}(f_3, o) \wedge \text{Destination}(f_3, d) \wedge \text{DepartureTime}(f_3, t_3) \wedge (t_3 < t_2) \wedge \\ \text{LayoverNotTooShort}(f_1, f_3))) \rightarrow \text{LayoverNotTooLong}(f_1, f_2) \end{aligned}$$

6.034 - Spring 03 • 43

6.034 Notes: Section 11.1

Slide 11.1.1

We've now spent a fair bit of time learning about the language of first-order logic and the mechanisms of automatic inference. And, we've also found that (a) it is quite difficult to write first-order logic and (b) quite expensive to do inference. Both of these conclusions are well justified. Therefore, you may be wondering why we spent the time on logic.

We can motivate our study of logic in a variety of ways. For one, it is the intellectual foundation for all other ways of representing knowledge about the world. As we have already seen, the Web Consortium has adopted a logical language for its Semantic Web project. We also saw that airlines use a language not unlike FOL to describe fare restrictions. We will see later when we talk about natural language understanding that logic also plays a key role.

There is another practical application of logic that is reasonably widespread namely **logic programming**. In this section, we will look briefly at logic programming. Later, when we study natural language understanding, we will build on these ideas.

Rules and Logic Programming

6.034 - Spring 03 • 1

Logic in Practice

- Language of logic is extremely powerful.
- Say what's true, not how to use it.
 - $\forall x, y (\exists z \text{ Parent}(x,z) \wedge \text{Parent}(z,y)) \leftrightarrow \text{GrandParent}(x,y)$
 - Given parents, find grandparents
 - Given grandparents, find parents

6.034 - Spring 03 • 2

Slide 11.1.3

However, this very power and lack of specificity about algorithms means that the general methods for performing computations on logical representations (for example, resolution refutation) are hopelessly inefficient for most practical problems.

Logic in Practice

- Language of logic is extremely powerful.
- Say what's true, not how to use it.
 - $\forall x, y (\exists z \text{ Parent}(x,z) \wedge \text{Parent}(z,y)) \leftrightarrow \text{GrandParent}(x,y)$
 - Given parents, find grandparents
 - Given grandparents, find parents
- But, resolution theorem-provers are too inefficient!

6.034 - Spring 03 • 3

Logic in Practice

- Language of logic is extremely powerful.
- Say what's true, not how to use it.
 - $\forall x, y (\exists z \text{ Parent}(x,z) \wedge \text{Parent}(z,y)) \leftrightarrow \text{GrandParent}(x,y)$
 - Given parents, find grandparents
 - Given grandparents, find parents
- But, resolution theorem-provers are too inefficient!
- To regain practicality:
 - Limit the language
 - Simplify the proof algorithm
- Rule-Based Systems
- Logic Programming

6.034 - Spring 03 * 4

Slide 11.1.4

There are, however, approaches to regaining some of the efficiency while keeping much of the power of the representation. These approaches involve both limiting the language as well as simplifying the inference algorithms to make them more predictable. Similar ideas underlie both **logic programming** and **rule-based systems**. We will bias our presentation towards logic programming.

Slide 11.1.5

In logic programming we will also use the clausal representation that we derived for resolution refutation. However, we will limit the type of clauses that we will consider to the class called **Horn clauses**. A clause is Horn if it has at most one positive literal. In the examples below, we show literals without variables, but the discussion applies both to propositional and first order logic.

There are three cases of Horn clauses:

- A **rule** is a clause with one or more negative literals and exactly one positive literal. You can see that this is the clause form of an implication of the form $P_1 \wedge \dots \wedge P_n \rightarrow Q$, that is, the conjunction of the P 's implies Q .
- A **fact** is a clause with exactly one positive literal and no negative literals. We generally will distinguish the case of a **ground fact**, that is, a literal with no variables, from the general case of a literal with variables, which is more like an unconditional rule than what one would think of as a "fact".
- In general, there is another case, known as a **consistency constraint** when the clause has no positive literals. We will not deal with these further, except for the special case of a **conjunctive goal clause** which will take this form (the negation of a conjunction of literals is a Horn clause with no positive literal). However, goal clauses are not rules.

Horn Clauses

- A clause is **Horn** if it has at most one positive literal
 - $\neg P_1 \vee \dots \vee \neg P_n \vee Q$ (**Rule**)
 - Q (**Fact**)
 - $\neg P_1 \vee \dots \vee \neg P_n$ (**Consistency Constraint**)
- We will not deal with Consistency Constraints
- Rule Notation
 - $P_1 \wedge \dots \wedge P_n \rightarrow Q$ (**Logic**)
 - If $P_1 \dots P_n$ Then Q (**Rule-Based System**)
 - $Q :- P_1, \dots, P_n$ (**Prolog**)
- P_i are called **antecedents** (or body)
- Q is called the **consequent** (or head)

6.034 - Spring 03 * 5

Horn Clauses

- A clause is **Horn** if it has at most one positive literal
 - $\neg P_1 \vee \dots \vee \neg P_n \vee Q$ (**Rule**)
 - Q (**Fact**)
 - $\neg P_1 \vee \dots \vee \neg P_n$ (**Consistency Constraint**)
- We will not deal with Consistency Constraints

6.034 - Spring 03 * 5

Slide 11.1.6

There are many notations that are in common use for Horn clauses. We could write them in standard logical notation, either as clauses, or as implications. In rule-based systems, one usually has some form of equivalent "If-Then" syntax for the rules. In Prolog, which is the most popular logic programming language, the clauses are written as a sort of reverse implication with the ":" instead of "<-".

We will call the Q (positive) literal the **consequent** of a rule and call the P_i (negative) literals the **antecedents**. This is terminology for implications borrowed from logic. In Prolog it is more common to call Q the **head** of the clause and to call the P literals the **body** of the clause.

6.034 - Spring 03 * 6

Slide 11.1.7

Note that not every logical statement can be written in Horn clause form, especially if we disallow clauses with zero positive literals (consistency constraints). Importantly, one cannot have a negation on the right hand side of an implication. That is, we cannot have rules that conclude that something is not true! This is a reasonably profound limitation in general but we can work around it in many useful situations, which we will discuss later. Note that because we are not dealing with consistency constraints (all negative literals) we will not be able to deal with negative facts either.

Limitations

- Cannot conclude negation

- $P \rightarrow \neg Q$
- $\neg P \vee \neg Q$: Consistency constraint
- $\neg P$: Consistency constraint

6.034 - Spring 03 * 7

**Limitations**

- Cannot conclude negation
 - $P \rightarrow \neg Q$
 - $\neg P \vee \neg Q$: Consistency constraint
 - $\neg P$: Consistency constraint
- Cannot conclude (or assert) disjunction
 - $P_1 \wedge P_2 \rightarrow Q_1 \vee Q_2$
 - $Q_1 \vee Q_2$
 - These are not Horn

6.034 - Spring 03 * 8

Slide 11.1.8

Similarly, if we have a disjunction on the right hand side of an implication, the resulting clause is not Horn. In fact, we cannot assert a disjunction with more than one positive literal or a disjunction of all negative literals. The former is not Horn while the latter is a consistency constraint.

Slide 11.1.9

It turns out that given our simplified language, we can use a simplified procedure for inference, called **backchaining**, which is basically a generalized form of Modus Ponens (one of the "natural deduction" rules we saw earlier).

Backchaining is relatively simple to understand given that you've seen how resolution works. We start with a literal to "prove", which we call C. We will also use Green's trick (as in Chapter 6.3) to keep track of any variable bindings in C during the proof.

We will keep a stack (first in, last out) of goals to be proved. We initialize the stack to have C (first) followed by the Answer literal (which we write as `Ans`).

Inference: Backchaining

- To "prove" a literal C
 - Push C and an Ans literal on a stack

6.034 - Spring 03 * 9

**Inference: Backchaining**

- To "prove" a literal C
 - Push C and an Ans literal on a stack
 - Repeat until stack only has Ans literal or no actions available.
 - Pop literal L off of stack

6.034 - Spring 03 * 10

Slide 11.1.10

The basic loop is to pop a literal (L) off the stack until either (a) only the Ans literal remains or (b) there are no further actions possible. The first case corresponds to a successful proof; the second case represents a failed proof.

A word of warning. This loop does not necessarily terminate. We will see examples later where simple sets of rules lead to infinite loops.

Slide 11.1.11

Given a literal L, we look for a fact that unifies with L or a rule whose consequent (head) unifies with L. If we find a match, we push the antecedent literals (if any) onto the stack, apply the unifier to the entire stack and then rename all the variables to make sure that there are no variable conflicts in the future. There are other ways of dealing with the renaming but this one will work.

In general, there will be more than one fact or rule that could match L; we will pick one now but be prepared to come back to try another one if the proof doesn't work out. More on this later.

Inference: Backchaining

- To "prove" a literal C

- Push C and an Ans literal on a stack
- Repeat until stack only has Ans literal or no actions available.
 - Pop literal L off of stack
 - Choose [with backup] a rule (or fact) whose consequent unifies with L
 - Push antecedents (in order) onto stack
 - Apply unifier to entire stack
 - Rename variables on stack

6.034 - Spring 03 • 11

**Inference: Backchaining**

- To "prove" a literal C
 - Push C and an Ans literal on a stack
 - Repeat until stack only has Ans literal or no actions available.
 - Pop literal L off of stack
 - Choose [with backup] a rule (or fact) whose consequent unifies with L
 - Push antecedents (in order) onto stack
 - Apply unifier to entire stack
 - Rename variables on stack
 - If no match, fail [backup to last choice]

6.034 - Spring 03 • 12

Slide 11.1.12

If no match can be found for L, we fail and backup to try the last choice that has other pending matches.

Slide 11.1.13

If you think about it, you'll notice that backchaining is just our familiar friend, resolution. The stack of goals can be seen as negative literals, starting with the negated goal. We don't actually show literals on the stack with explicit negation but they are implicitly negated.

At every point, we pair up a negative literal from the stack with a positive literal (the consequent) from a fact or rule and add the remaining negative literals (the antecedents) to the stack.

Backchaining and Resolution

- Backchaining is just resolution
- To prove C (propositional case)
 - Negate C $\Rightarrow \neg C$
 - Find rule $\neg P_1 \vee \dots \vee \neg P_n \vee C$
 - Resolve to get $\neg P_1 \vee \dots \vee \neg P_n$
 - Repeat for each negative literal
- First order case introduces unification but otherwise the same.

6.034 - Spring 03 • 13

**Proof Strategy**

- Depth-First search for a proof
- Order matters
 - Rule order
 - try ground facts first
 - then rules in given order
 - Antecedent order
 - left to right
- More predictable, like a program, less like logic

6.034 - Spring 03 • 14

Slide 11.1.14

When we specified backchaining we did it with a particular search algorithm (using the stack), which is basically depth-first search. Furthermore, we will assume that the facts and rules are examined in the order in which they occur in the program. Also that literals from the body of a rule are pushed onto the stack in reverse order, so that the one that occurs first in the body will be the first popped off the stack.

Given these ordering restrictions, it is much easier to understand what a logic program will do. On the other hand, one must understand that what it will do is not what a general theorem prover would do with the same rules and facts.

Slide 11.1.15

Time for an example. Let's look at the following database of facts and rules. The first two entries are ground facts, that A is Father of B and B is Mother of C. The third entry defines a grandparent rule that we would write in FOL as:

```
@x . @y. P(x,y) ^ P(y,z) -> GrandP(x,z)
```

Our rule is simply this rule written with the implication pointing "backwards". Also, our rule language does not have quantifiers; all the variables are implicitly universally quantified.

In our rule language, we will modify our notational conventions for FOL. Instead of identifying constants by prefixing them with \$, we will indicate variables by prefixing them with ?. The rationale for this is that in our logic examples we had lots more variables than constants, but that will be different in many of our logic-programming examples.

The next two rules specify that a Father is a Parent and a Mother is a parent. In usual FOL notation, these would be:

```
@x . @y. F(x,y) -> P(x,y)
@x . @y. M(x,y) -> P(x,y)
```

Example

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

```
?- GrandP(a,b).
   GrandP(a,b) , GrandP(b,c).
   3,?x/?y,?z//?y,?z , ?y>?y,?y>?y.
   GrandP(b,c) , GrandP(b,c) , GrandP(b,c).
   3,?x/?y,?z//?y,?z , ?y>?y,?y>?y.
```

6.034 - Spring 03 • 15

Example

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- * Prove:
GrandP(?g,C) , Ans(?g)

```
3,?x/?y,?z//?y,?z , ?y>?y,?y>?y.
   GrandP(b,c) , GrandP(b,c) , GrandP(b,c).
   3,?x/?y,?z//?y,?z , ?y>?y,?y>?y.
```

6.034 - Spring 03 • 16

Slide 11.1.16

Now, we set out to find the Grandparent of C. With resolution refutation, we would set out to derive a contradiction from the negation of the goal:

```
~ ]g . GrandP(g,C)
```

whose clause form is $\sim \text{GrandP}(g, C)$. The list of literals in our goal stack are implicitly negated, so we start with $\text{GrandP}(g, C)$ on the stack. We have also added the Ans literal with the variable we are interested in, ?g, hopefully the name of the grandparent.

Now, we set out to find a fact or rule consequent literal in the database that matches our goal literal.

Example

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

```
?- GrandP(?g,C) , Ans(?g)
   - [3,?x/?g,?z/C; ?y=?y1,?g=?g1]
* Parent(?g1,?y1) , Parent(?y1,C) , Ans(?g1)
```

6.034 - Spring 03 • 17

Slide 11.1.17

You can see that the grandparent goal literal unifies with the consequent of rule 3 using the unifier $\{?x/?g, ?z/C\}$. So, we push the antecedents of rule 3 onto the stack, apply the unifier and then rename all the remaining variables, as indicated. The resulting goal stack now has two Parent literals and the Ans literal. We proceed as before by popping the stack and trying to unify with the first Parent literal.

Example

```

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
GrandP(?g,C), Ans(?g)
  - [3,?x/?g,?z/C; ?y=?y1,?g=?g1]
• Parent(?g1,?y1), Parent(?y1,C), Ans(?g1)
  - [4,?x/?g1,?y/?y1; ?y1?=?y2,?g1?=?g2]
• Father(?g2,?y2), Parent(?y2,C), Ans(?g2)
  - [1,?g2/A,?y2/B]
  - [2,?y2/C]
  - [3,?y2/C]
  - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [6,?y2?=?y1,?g2?=?g1]

<fail>
      - [3,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [6,?y2?=?y1,?g2?=?g1]

Ans(?g)

```

6.034 - Spring 03 • 18

Slide 11.1.18

The first Parent goal literal unifies with the consequent of rule 4 with the unifier shown. The antecedent (the Father literal) is pushed on the stack, the unifier is applied and the variables are renamed.

Note that there are two Parent rules, we use the first one but we have the other one available should we fail with this one.

Slide 11.1.19

The Father goal literal matches the first fact, which now unifies the ?g variable to A and the ?y variable to B. Note that since we matched a fact, there are no antecedents to push on the stack (as in resolution with a unit-length clause). We apply the unifier, rename and proceed.

Example

```

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
GrandP(?g,C), Ans(?g)
  - [3,?x/?g,?z/C; ?y=?y1,?g=?g1]
• Parent(?g1,?y1), Parent(?y1,C), Ans(?g1)
  - [4,?x/?g1,?y/?y1; ?y1?=?y2,?g1?=?g2]
• Father(?g2,?y2), Parent(?y2,C), Ans(?g2)
  - [1,?g2/A,?y2/B]
• Parent(B,C), Ans(A)
  - [2,?y2/C]
  - [3,?y2/C]
  - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [6,?y2?=?y1,?g2?=?g1]

<fail>
      - [3,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [6,?y2?=?y1,?g2?=?g1]

Ans(?g)

```

6.034 - Spring 03 • 20

Slide 11.1.20

Now, we can match the Parent(B,C) goal literal to the consequent of rule 4 and get a new goal (after applying the substitution to the antecedent), Father(B,C). However we can see that this will not match anything in the database and we get a failure.

Slide 11.1.21

The last choice we made that has a pending alternative is when we matched Parent(B,C) to the consequent of rule 4. If we instead match the consequent of rule 5, we get an alternative literal to try, namely Mother(B,C).

Example

```

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
GrandP(?g,C), Ans(?g)
  - [3,?x/?g,?z/C; ?y=?y1,?g=?g1]
• Parent(?g1,?y1), Parent(?y1,C), Ans(?g1)
  - [4,?x/?g1,?y/?y1; ?y1?=?y2,?g1?=?g2]
• Father(?g2,?y2), Parent(?y2,C), Ans(?g2)
  - [1,?g2/A,?y2/B]
• Parent(B,C), Ans(A)
  - [2,?y2/C]
  - [3,?y2/C]
  - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [6,?y2?=?y1,?g2?=?g1]

• <fail>
      - [3,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [6,?y2?=?y1,?g2?=?g1]

Ans(?g)

```

6.034 - Spring 03 • 21

Example

```

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
GrandP(?g,C), Ans(?g)
  - [3,?x/?g,?z/C; ?y=?y1,?g=?g1]
• Parent(?g1,?y1), Parent(?y1,C), Ans(?g1)
  - [4,?x/?g1,?y/?y1; ?y1?=?y2,?g1?=?g2]
• Father(?g2,?y2), Parent(?y2,C), Ans(?g2)
  - [1,?g2/A,?y2/B]
• Parent(B,C), Ans(A)
  - [2,?y2/C]
  - [3,?y2/C]
  - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
  - [6,?y2?=?y1,?g2?=?g1]

• <fail>
      - [3,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [4,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [5,?x/?g2,?y/?y2; ?y2?=?y1,?g2?=?g1]
      - [6,?y2?=?y1,?g2?=?g1]

Ans(?g)

```

6.034 - Spring 03 • 21

Example

```

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
  GrandP(?g,C) , Ans(?g)
  - [3,?x/?g,?z/C; ?y=>?y1,?g=>?g1]
• Parent(?g1,?y1) , Parent(?y1,C) , Ans(?g1)
  - [4,?x/?g1,?y/?y1; ?y1=>?y2,?g1=>?g2]
• Father(?g2,?y2) , Parent(?y2,C) , Ans(?g2)
  - [1,?g2/A,?y2/B]
• Parent(B,C) , Ans(A)
  - [4,?x/B,?y/C]
• Father(B,C) , Ans(A)
• <fail>
  - [5,?x/B,?y/C]
• Mother(B,C) , Ans(A)
  - [2]
• Ans(A)

```

6.034 - Spring 03 • 22

Slide 11.1.22

This matches fact 2. At this point there are no antecedents to add to the stack and the Ans literal is on the top of the stack. Note that the binding of the variable ?g to A is in fact the correct answer to our original question.

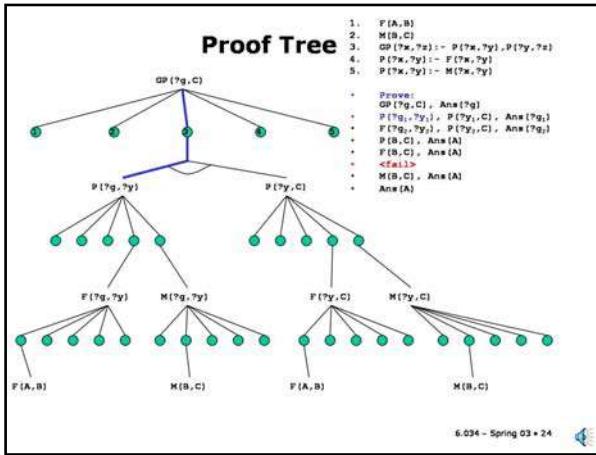
Slide 11.1.23

Another way to look at the process we have just gone through is as a form of tree search. In this search space, the states are the entries in the stack, that is, the literals that appear on our stack. The edges (shown with a green dot in the middle of each edge) are the rules or facts. However, there is one complication: a rule with multiple antecedents generates multiple children, each of which must be solved. This is indicated by the arc connecting the two descendants of rule 3 near the top of the tree.

This type of tree is called an AND-OR tree. The OR nodes come from the choice of a rule or fact to match to a goal. The AND nodes come from the multiple antecedents of a rule (all of which must be proved).

You should remember that such a tree is **implicit** in the rules and facts in our database, once we have been given a goal to prove. The tree is not constructed explicitly; it is just a way of visualizing the search process.

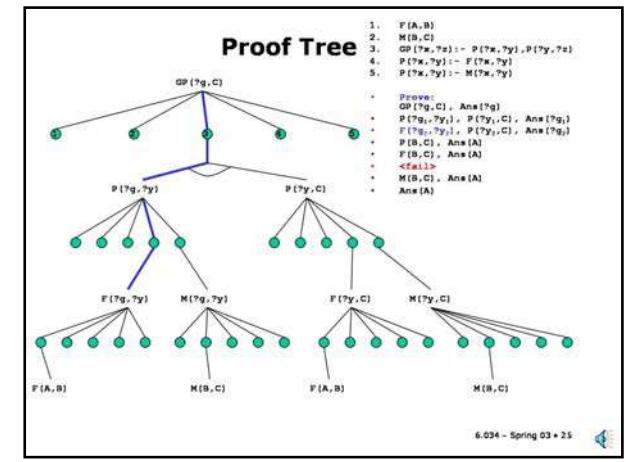
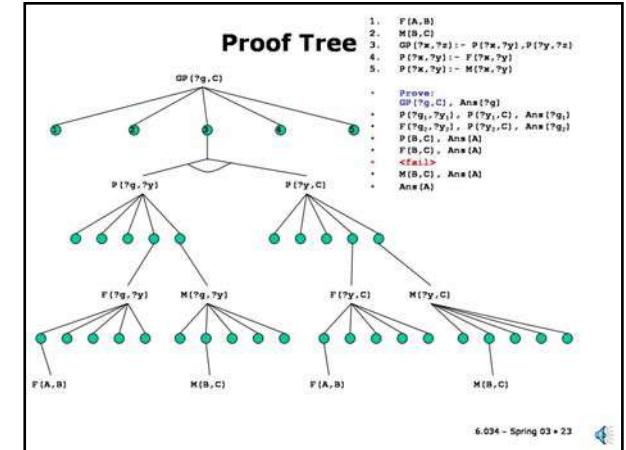
Let's go through our previous proof in this representation, which makes the choices we've made more explicit. We start with the GrandP goal at the top of the tree.

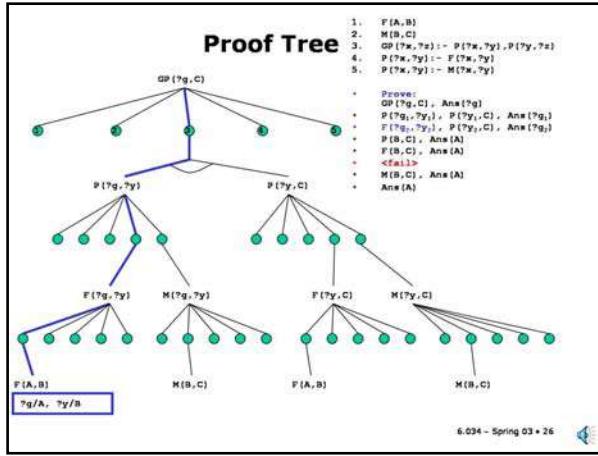
**Slide 11.1.24**

We match that goal to the consequent of rule 3 and we create two subgoals for each of the antecedents (after carrying out the substitutions from the unification). We will look at the first one (the one on the left) next.

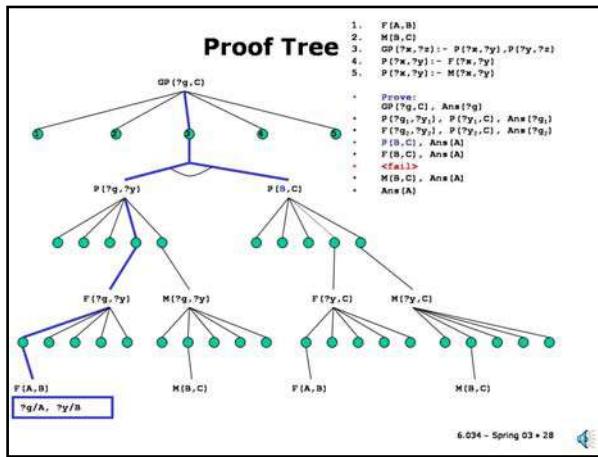
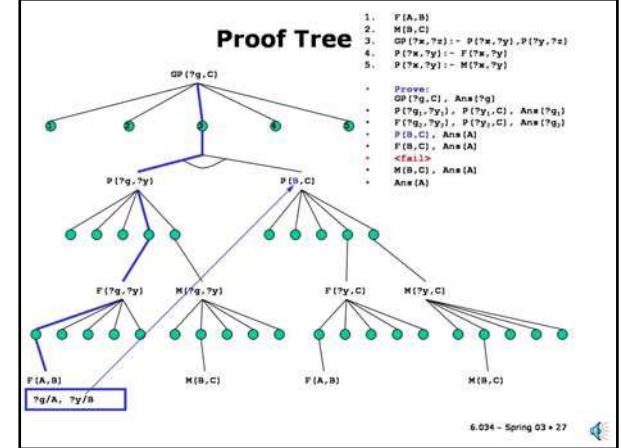
Slide 11.1.25

We match the Parent subgoal to the rule 4 and generate a Father subgoal.

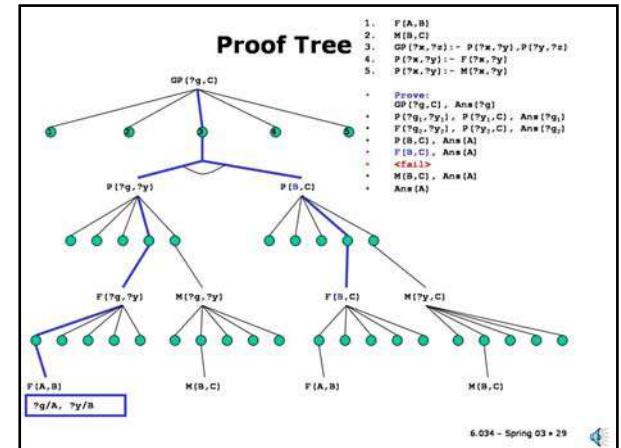


**Slide 11.1.26**

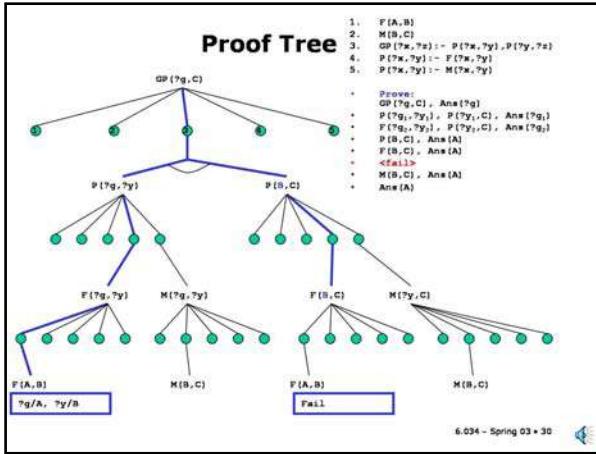
Which we match to fact 1 and create bindings for the variables in the goal. In all our previous steps we also created variable bindings but they were variable to variable bindings. Here, we finally match some variables to constants.

**Slide 11.1.28**

Now, we tackle the second Parent subgoal ...

**Slide 11.1.29**

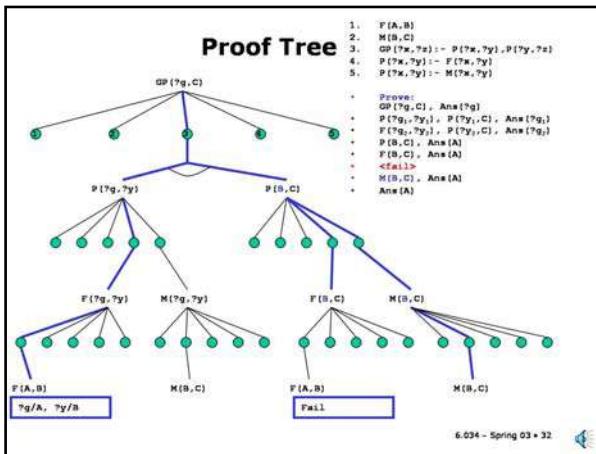
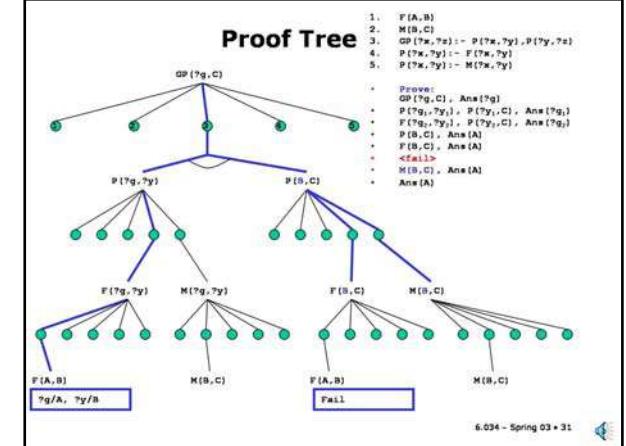
... which proceeds as before to match rule 4 and generate a Father subgoal, Father(B,C) in this case.

**Slide 11.1.30**

But, as we saw before that leads to a failure when we try to match the database.

Slide 11.1.31

So, instead, we look at the other alternative, matching the second Parent subgoal to rule 5, and generate a Mother (B, C) subgoal.

**Slide 11.1.32**

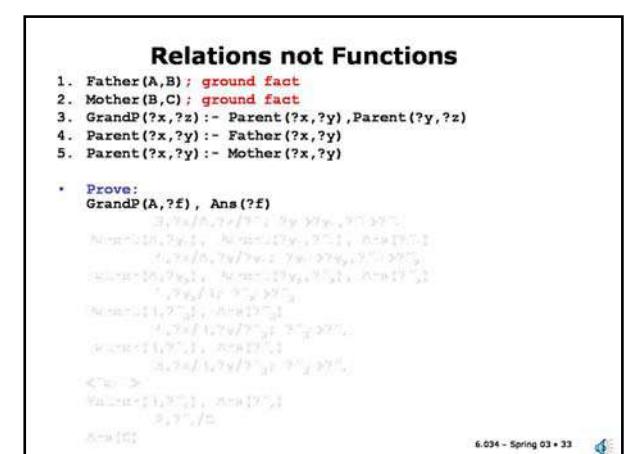
This matches the second fact in the database and we succeed with our proof since we have no pending subgoals to prove.

This view of the proof process highlights the search connection and is a useful mental model, although it is too awkward for any big problem.

Slide 11.1.33

At the beginning of this section, we indicated as one of the advantages of a logical representation that we could define the relationship between parents and grandparents without having to give an algorithm that might be specific to finding grandparents of grandchildren or vice versa. This is still (partly) true for logic programming. We have just seen how we could use the facts and rules shown here to find a grandparent of someone. Can we go the other way? The answer is yes.

The initial goal we have shown here asks for the grandchild of A, which we know is C. Let's see how we find this answer.



Relations not Functions

1. Father(A,B) : ground fact
2. Mother(B,C) : ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
GrandP(A,?f), Ans(?f)
- [3,?x/A,?z/?f; ?y=?y₁,?f=?f₁]
- Parent(A,?y₁), Parent(?y₁,?f₁), Ans(?f₁)

```
?x=?y1
?y1?y2
?y1?y2
?y1?y2
?y1?y2
```

```
<fail>
Value<(1,?y1)>, Ans(?f1)
?y1?y2/0
```

```
Ans(?f1)
```

6.034 - Spring 03 • 34

Slide 11.1.34

Once again, we match the GrandP goal to rule 3, but now the variable bindings are different. We have a constant binding in the first Parent subgoal rather than in the second.

Relations not Functions

1. Father(A,B) : ground fact
2. Mother(B,C) : ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
GrandP(A,?f), Ans(?f)
- [3,?x/A,?z/?f; ?y=?y₁,?f=?f₁]
- Parent(A,?y₁), Parent(?y₁,?f₁), Ans(?f₁)
- [4,?x/A,?y/?y₁; ?y₁?y₂,?f₁?f₂]
- Father(A,?y₂), Parent(?y₂,?f₂), Ans(?f₂)

```
?x=?y1
?y1?y2
?y1?y2
?y1?y2
```

```
<fail>
Value<(1,?y1)>, Ans(?f1)
?y1?y2/0
```

```
Ans(?f1)
```

6.034 - Spring 03 • 35

Relations not Functions

1. Father(A,B) : ground fact
2. Mother(B,C) : ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
GrandP(A,?f), Ans(?f)
- [3,?x/A,?z/?f; ?y=?y₁,?f=?f₁]
- Parent(A,?y₁), Parent(?y₁,?f₁), Ans(?f₁)
- [4,?x/A,?y/?y₁; ?y₁?y₂,?f₁?f₂]
- Father(A,?y₂), Parent(?y₂,?f₂), Ans(?f₂)
- [1,?y₂/B; ?f₂?f₁]
- Parent(B,?f₃), Ans(?f₃)

```
?x=?y1
?y1?y2
?y1?y2
?y1?y2
```

```
<fail>
Value<(1,?y1)>, Ans(?f1)
?y1?y2/0
```

```
Ans(?f1)
```

6.034 - Spring 03 • 36

Slide 11.1.36

Then, we match the first fact, namely Father(A,B), which causes us to bind the ?x variable in the second Parent subgoal to B. So, now, we look for a child of B.

Relations not Functions

1. Father(A,B) : ground fact
2. Mother(B,C) : ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
GrandP(A,?f), Ans(?f)
- [3,?x/A,?z/?f; ?y=?y₁,?f=?f₁]
- Parent(A,?y₁), Parent(?y₁,?f₁), Ans(?f₁)
- [4,?x/A,?y/?y₁; ?y₁?y₂,?f₁?f₂]
- Father(A,?y₂), Parent(?y₂,?f₂), Ans(?f₂)
- [1,?y₂/B; ?f₂?f₁]
- Parent(B,?f₃), Ans(?f₃)
- [4,?x/B,?y/?f₃; ?f₃?f₁]
- Father(B,?f₄), Ans(?f₄)

```
?x=?y1
?y1?y2
?y1?y2
?y1?y2
```

```
<fail>
```

6.034 - Spring 03 • 37

Relations not Functions

```

1. Father(A,B) : ground fact
2. Mother(B,C) : ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
  GrandP(A,?f), Ans(?f)
  - [3,?x/A,?z/?f; ?y=?y1,?f=?f1]
  • Parent(A,?y1), Parent(?y1,?f1), Ans(?f1)
    - [4,?x/A,?y/?y1; ?y1?=?y2,?f1?=?f2]
  • Father(A,?y2), Parent(?y2,?f2), Ans(?f2)
    - [1,?y2/B; ?f2?=?f3]
  • Parent(B,?f3), Ans(?f3)
    - [4,?x/B,?y/?f3; ?f3?=?f4]
  • Father(B,?f4), Ans(?f4)
  • <fail>
    - [5,?x/B,?y/?f3; ?f3?=?f4]
  • Mother(B,?f4), Ans(?f4)

```

6.034 - Spring 03 • 38

Slide 11.1.38

We now match the second Parent subgoal to rule 5 and generate a Mother(B,?f) subgoal.

Slide 11.1.39

...which succeeds and binds ?f (our query variable) to C, as expected.

Note that if we had multiple grandchildren of A in the database, we could generate them all by continuing the search at any pending subgoals that had multiple potential matches.

The bottom line is that we are representing **relations** among the elements of our domain (recall that's what a logical predicate denotes) rather than computing functions that specify a single output for a given set of inputs.

Another way of looking at it is that we do not have a pre-conceived notion of which variables represent "input variables" and which are "output variables".

Order Revisited

- Given
 - parent(A,B)
 - parent(B,C)
 - ancestor(?x,?z) :- parent(?x,?z)
 - ancestor(?x,?z) :- parent(?x,?y), ancestor(?y,?z)
 - Prove:
ancestor(?x,C), Ans(?x)
 - ...
 - Ans(A)
- 6.034 - Spring 03 • 40

Slide 11.1.40

We have seen in our examples thus far that we explore the underlying search space in order. This approach has consequences. For example, consider the following simple rules for defining an ancestor relation. It says that a parent is an ancestor (this is the base case) and that the ancestor of a parent is an ancestor (the recursive case). You could use this definition to list a person's ancestors or, as we did for grandparent, to list a person's descendants.

But what would happen if we changed the order a little bit?

Slide 11.1.41

Here we've switched the order of rules 3 and 4 and furthermore switched the order of the literals in the recursive ancestor rule. The effect of these changes, which have no logical import, is disastrous: basically it generates an infinite loop.

Relations not Functions

- ```

1. Father(A,B) : ground fact
2. Mother(B,C) : ground fact
3. GrandP(?x,?z) :- Parent(?x,?y), Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

• Prove:
 GrandP(A,?f), Ans(?f)
 - [3,?x/A,?z/?f; ?y=?y1,?f=?f1]
 • Parent(A,?y1), Parent(?y1,?f1), Ans(?f1)
 - [4,?x/A,?y/?y1; ?y1?=?y2,?f1?=?f2]
 • Father(A,?y2), Parent(?y2,?f2), Ans(?f2)
 - [1,?y2/B; ?f2?=?f3]
 • Parent(B,?f3), Ans(?f3)
 - [4,?x/B,?y/?f3; ?f3?=?f4]
 • Father(B,?f4), Ans(?f4)
 • <fail>
 - [5,?x/B,?y/?f3; ?f3?=?f4]
 • Mother(B,?f4), Ans(?f4)
 - [2,?f4/C]
 • Ans(C)

```
- 6.034 - Spring 03 • 39

**Order Revisited**

- Given
    - parent(A,B)
    - parent(B,C)
    - ancestor(?x,?z) :- parent(?x,?z)
    - ancestor(?x,?z) :- parent(?x,?y), ancestor(?y,?z)
  - Prove:  
ancestor(?x,C), Ans(?x)
  - ...
  - Ans(A)
  - How about:
    - parent(A,B)
    - parent(B,C)
    - ancestor(?x,?z) :- ancestor(?y,?z), parent(?x,?y)
    - ancestor(?x,?z) :- parent(?x,?z)
  - Prove:  
ancestor(?x,C), Ans(?x)
  - ...
  - <error: stack overflow>
- 6.034 - Spring 03 • 41

## Order Revisited

- Given
  1. parent(A,B)
  2. parent(B,C)
  3. ancestor(?x,?z) :- parent(?x,?y), ancestor(?y,?z)
  4. ancestor(?x,?z) :- parent(?x,?y), ancestor(?y,?z)
  - Prove:  
ancestor(?x,C), Ans(?x)
  - ...
  - Ans(A)
- How about:
  1. parent(A,B)
  2. parent(B,C)
  3. ancestor(?x,?z) :- ancestor(?y,?z), parent(?x,?y)
  4. ancestor(?x,?z) :- parent(?x,?z)
  - Prove:  
ancestor(?x,C), Ans(?x)
  - ...
  - <error: stack overflow>
- Clauses examined top to bottom and literals left to right.  
This is not logic!

6.034 - Spring 03 • 42

### Slide 11.1.42

This type of behavior is what you would expect from a recursive program if you put the recursive case before the base case. The key point is that logic programming is half way between traditional programming and logic and exactly like neither one.

### Slide 11.1.43

It is often the case that we want to have a condition on a rule that says that something is not true. However, that has two problems, one is that the resulting rule would not be Horn. Furthermore, as we saw earlier, we have no way of concluding a negative literal. In logic programming one typically makes a **closed world** assumption, sometimes jokingly referred to as the "closed mind" assumption, which says that we know everything to be known about our domain. And, if we don't know it (or can't prove it), then it must be false. We all know people like this...

## Negation

- We cannot have a rule such as
  - $P_1 \wedge \neg P_2 \rightarrow Q$
  - $\neg P_1 \vee P_2 \vee Q$  - not Horn (two pos literals)
  - Cannot have rule that concludes a negation
- In logic programming, we assume we have complete information about the world (**closed-world assumption**)

*Q: What's a closed world assumption?  
A: Assume we know everything  
• Prove: ; in empty KB  
not P(?x), Ans(?x)  
• Ans(?x) ; success*

6.034 - Spring 03 • 43

## Negation

- We cannot have a rule such as
  - $P_1 \wedge \neg P_2 \rightarrow Q$
  - $\neg P_1 \vee P_2 \vee Q$  - not Horn (two pos literals)
  - Cannot have rule that concludes a negation
- In logic programming, we assume we have complete information about the world (**closed-world assumption**)
- We use "failure to prove" as negation - a dangerous assumption.
  - Prove: ; in empty KB  
not P(?x), Ans(?x)
  - Ans(?x) ; success

6.034 - Spring 03 • 44

### Slide 11.1.44

Given we assume we know everything relevant, we can simulate negation by failure to prove. This is very dangerous in general situations where you may not know everything (for example, it's not a good thing to assume in exams)...

## Negation

- But often very useful in finite domains, e.g. flights database, products of a company, etc.
- For example:
 

```
Layover_not_too_long(?f1, ?f2) :-
 Arrival_time(?f1, ?t1),
 Departure_time(?f2, ?t2),
 not Alternative_connection(?f1, ?t1, ?f2, ?t2)
```
- Will succeed if the Alternative\_connection literal fails.

6.034 - Spring 03 • 45

## 6.034 Notes: Section 11.2

### Slide 11.2.1

So far, what we have seen of logic programming may not seem much like programming. Now, we will look at a number of list processing examples that will look more like the examples that you are used to writing in Scheme.

What we will see is essentially a subset of the logic programming language Prolog, which is used fairly widely. There are a number of open-source and commercial versions of Prolog available. We will use a very simple home-brew system implemented in Scheme rather than one of these systems so that there are no mysteries in the implementation. However, we will pay a substantial performance penalty for this choice.

### Logic Programming

- So far, not much like programming
- But, this framework can be used as the basis of a general purpose programming language
- Prolog is the most widely used logic programming language
- For example:
  - Gnu Prolog <http://www.gnu.org/software/prolog/prolog.html>
  - SWI Prolog <http://www.swi-prolog.org/>
  - SICStus Prolog <http://www.sics.se/sicstus/>
  - Visual Prolog <http://www.visual-prolog.com/>
  - ...

6.034 – Spring 03 • 1

### List Processing: length

```
(define (length y)
 (if (null? y)
 0
 (+ 1 (length (cdr y)))))
```

6.034 – Spring 03 • 2

### Slide 11.2.2

This would be a Prolog-like solution to the same problem. It has essentially the same structure as the Scheme program. We use a predicate "length" that has two arguments, one is the list and the other its length.

The first "fact" handles the base case; it defines the length of the null list as 0.

The second rule handles the recursive case. The consequent of the rule (the left-hand-side) is what will match a pending subgoal. Note the form of the first argument of the consequent: it is a Scheme dotted pair. It is set up to match the variable ?h to the car of a list and the variable ?x to the cdr of the list. The second argument of the consequent expresses the length of the list as a function of the length of the cdr of the list.

The right hand side of the rule is the IF part. It sets up a simpler subgoal to solve. Once we solve it, we will have bound ?l to the length of the cdr and we will know the length of the full list (including the car).

Let's look at an example.

### Slide 11.2.2

Let's start with a very simple Scheme program to compute the length of a list. It's composed of two "cases", the base case when the list is null and the recursive case, in which we reduce the problem into a simpler instance of the same problem (getting the length of the cdr of the list) and compute the final result by adding one to the result of the recursive call.

### List Processing: length

```
(define (length y)
 (if (null? y)
 0
 (+ 1 (length (cdr y)))))

• length((),0)
• length((?h . ?x), ?l+1) :- length(?x,?l)
```

Recall "dotted pair" notation (x . y) means x is car of list and y is cdr of list. (cons x y) returns (x . y). In general something like (a b . x) indicates that x is rest of list.  
 $(a . ()) \equiv (a)$   
 $(a b . (c d)) \equiv (a b c d)$

6.034 – Spring 03 • 3

## List Processing: length

```

1. length([], 0)
2. length([?h . ?x], ?l+1):- length(?x, ?l)

• Prove:
length((a b), ?a), Ans(?a)
[2, ?h/a, ?x/(b), ?a/?l+1]
a. length((a b), ?l), Ans(?l+1)
[rename, l⇒?l1]
b. length((b), ?l), Ans(?l+1)
[2, ?h/b, ?x/(), ?l1?l+1]
c. length((b), ?l), Ans(?l+1+1)
[rename, ?l=⇒?l2]
d. length(), ?l2, Ans(?l2+1+1)
[1, ?l2/0]
e. Ans(0+1+1)

```

6.034 - Spring 03 • 4

### Slide 11.2.4

You can see the operation of this little program here. The operation is very like that of the corresponding Scheme program. The sequence of subgoals corresponds to the recursive calls to the program.

We have separated the unifier substitution step from the renaming to make things a little clearer. Note that without the renaming we would be hopelessly confused with the bindings of ?l.

In practice, in a Prolog system, the arithmetic expressions would be evaluated by the system and we would get Ans(2).

### Slide 11.2.5

This is the same operation but combining the unifier substitution and renaming steps. You can see the sequence of subgoals more clearly here.

## List Processing: length?

```

1. length([], 0)
2. length(?x, ?l-1):- length((?h . ?x), ?l)

• Prove:
length((a b), ?a), Ans(?a)
[2, ?h/a, ?x/(b), ?a/?l-1]
a. length((b), ?l), Ans(?l-1)
[2, ?h/b, ?x/(), ?l-1]
b. length(), ?l, Ans(?l-1+1)
[1, ?l/0]
c. Ans(0+1-1)

```

6.034 - Spring 03 • 6

### Slide 11.2.6

You may be wondering whether this formulation of length would also work. Certainly, it seems just as valid as the one we used. Let's trace it through. We start with the same goal as before, finding the length of the list (a b).

## List Processing: length

```

1. length([], 0)
2. length((?h . ?x), ?l+1):- length(?x, ?l)

• Prove:
length((a b), ?a), Ans(?a)
[2, ?h/a, ?x/(b), ?a/?l+1]
a. length((b), ?l), Ans(?l+1)
[2, ?h/b, ?x/(), ?l+1]
b. length(), ?l, Ans(?l+1+1)
[1, ?l/0]
c. Ans(0+1+1)

```

6.034 - Spring 03 • 5

### Slide 11.2.7

We match the goal to the consequent of rule 2 and do the substitution to get a new subgoal. Note, however, that this is not a simpler subgoal. It's actually trying to find the length of a longer, not completely specified, list. If we knew the length of such a list then we could know the length of our input list. Can you smell trouble brewing?

## List Processing: length?

```

1. length([], 0)
2. length(?x, ?l-1):- length((?h . ?x), ?l)

• Prove:
length((a b), ?a), Ans(?a)
[2, ?x/(a b), ?a/?l-1]
• length((?h . (a b)), ?l), Ans(?l-1)
[rename, h⇒?h1; a⇒?a1; b⇒?b1]
length((b), ?l), Ans(?l-1)
[2, ?h1/b, ?x/(), ?l-1]
length(), ?l, Ans(?l-1+1)
[1, ?l/0]
c. Ans(0+1-1)

```

6.034 - Spring 03 • 7

### List Processing: length?

```

1. length((),0)
2. length(?x,?l-1) :- length((?h . ?x),?l)

• Prove:
 length((a b),?a),Ans(?a)
 - [2,?x/(a b),?a/?l-1]
 • length((?h . (a b)),?l),Ans(?l-1)
 - [rename,l=?l_1,?h=?h_1]
 • length((?h_1 a b),?l_1),Ans(?l_1-1)
 - [2,?x/(?h_1 a b),?l_1/?l-1]
 • length((?h . (?h_1 a b)),?l),Ans(?l-1-1)
 - [rename,?l=?l_2,?h=?h_2,?h_1=?h_3]
 • length((?h_2 ?h_3 a b),?l_2), Ans(?l_2-1-1)
 • etc

```

6.034 - Spring 03 \* 8

### Slide 11.2.8

Sure enough; we've coded an infinite loop. The moral of the story is that you want to write these recursive rules in the form of "complex consequent :- simple antecedent" and not the other way around.

### List Processing: length

- Another equivalent formulation

```

• length((),0)
• length((?h . ?x),?l) :- length(?x,?lw),?l=?lw+1

```

6.034 - Spring 03 \* 9

### List Processing: append

```

(define (append x y)
 (if (null? x)
 y
 (cons (car x) (append (cdr x) y)))
))

;Testing
;append ((1 2 3) (4 5))
;append ((1 2 3) (4 5)) ;(1 2 3 4 5)
;append ((1 2) (3 4 5))
;append ((1 2) (3 4 5)) ;(1 2 3 4 5)

```

6.034 - Spring 03 \* 10

### Slide 11.2.10

Let's look at another example, which is quite parallel to length. Here is a Scheme implementation of an append function. It too consists of two cases. The base case handles the case of the first argument being null, in which case the answer is simply the second list. The recursive case involves computing the solution to a simpler case (append of the cdr of x to y) and updating it to the final answer by consing the car of x to the result.

### List Processing: append

```

(define (append x y)
 (if (null? x)
 y
 (cons (car x) (append (cdr x) y)))
)

• append((),?y,?y)
• append((?h . ?x),?y,(?h . ?z)) :- append(?x,?y,?z)

```

Recall "dotted pair" notation ( $x . y$ ) means  $x$  is car of list and  $y$  is cdr of list. ( $\text{cons } x \ y$ ) returns  $(x . y)$ . In general something like  $(a \ b . \ x)$  indicates that  $x$  is rest of list.  
 $(a . ()) \equiv (a)$   
 $(a \ b . \ (c \ d)) \equiv (a \ b \ c \ d)$

6.034 - Spring 03 \* 11

### Slide 11.2.11

The logic program is completely analogous. The append predicate has three arguments, the lists to be appended and the result list. The first fact just says that the output of appending the null list and any list is just the second list. The second rule looks more complicated but it is just like the Scheme program. We pick out the car and the cdr in the consequent (note the use of dotted pair notation) and bind them to  $?h$  and  $?x$  respectively. Then we define a subgoal involving  $?x$  and  $?y$  and bind the result to  $?z$ . We can then construct the result for the original list by consing  $?h$  to  $?z$  (using dotted pair notation).

### List Processing: append

```

1. append([], ?y, ?y)
2. append([?h . ?x], ?y, [?h . ?z]) :- append(?x, ?y, ?z)

• Prove:
append((a b), (c d), ?1), Ans(?1)
[2, ?h/a, ?x/b, ?y/(c d), ?1/(a . ?z); ?z=?z1]
a. append((b), (c d), ?z1), Ans((a . ?z1))
[2, ?h/b, ?x/(), ?y/(c d), ?z1/(b . ?z); ?z=?z2]
b. append((()), (c d), ?z2), Ans((a . (b . ?z2)))
[1, ?y/?z2, ?z2/(c d)]
c. Ans((a . (b . (c d))))
- Note that (a . (b . (c d))) is (a b c d)

```

6.034 - Spring 03 • 12

### Slide 11.2.12

Here you can trace the operation of these rules in a very simple example. Once again note that the renaming is crucial for keeping things straight.

### Difference Lists

- A list can be represented as the difference between two lists, which we will write `diff(L1, L2)`
- For example, `(a b c)` can be written as
  - `diff( (a b c), () )`
  - `diff( (a b c d), (d) )`
  - `diff( (a b c d e), (d e) )`
  - `diff( (a b c . ?x), ?x )`
- The empty list is any list of the form
  - `diff(?x, ?x)`

6.034 - Spring 03 • 13

### Difference Lists

- A list can be represented as the difference between two lists, which we will write `diff(L1, L2)`
- For example, `(a b c)` can be written as
  - `diff( (a b c), () )`
  - `diff( (a b c d), (d) )`
  - `diff( (a b c d e), (d e) )`
  - `diff( (a b c . ?x), ?x )`
- The empty list is any list of the form
  - `diff(?x, ?x)`
- In `diff(L1, L2)` think of `L1` as a pointer to the beginning of the list and `L2` as a pointer to the end of the list.

6.034 - Spring 03 • 14

### Slide 11.2.14

The basic idea is that we can represent a list by a pair of pointers into a bigger list, one to the beginning and the other to the end of the list.

### List Processing: dappend

```

1. dappend(diff(?x, ?y), diff(?y, ?z), diff(?x, ?z))

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & | & 6 & 7 & 8 & 9 & 10 & 11 & | & 12 & 13 \\ \hline \end{array}$$

$$\begin{array}{ccc} \uparrow & \uparrow & \uparrow \\ ?x & ?y & ?z \end{array}$$

$$\begin{array}{l} ?x = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13) \\ ?y = (6\ 7\ 8\ 9\ 10\ 11\ 12\ 13) \\ ?z = (12\ 13) \end{array}$$


```

6.034 - Spring 03 • 15

### Slide 11.2.15

In this representation we can code append as a single fact! The picture shows the intuition behind the definition. On first viewing this seems like we're cheating. It's easy to see that this statement is true, but how does it actually compute anything? Partly, one has to think carefully about the representation of the input.

**List Processing: dappend**

```
1. dappend(diff(?x,?y),diff(?y,?z),diff(?x,?z))
```

- Prove:  
 $dappend(\text{diff}((a\ b\ .\ ?p),?p),\text{diff}((c\ d\ .\ ?q),?q),?w)$



6.034 - Spring 03 • 16

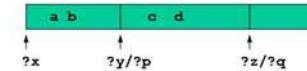
**Slide 11.2.16**

Here we see how a goal would be phrased in this representation. We have used the most general representation of the input lists.

**List Processing: dappend**

```
1. dappend(diff(?x,?y),diff(?y,?z),diff(?x,?z))
```

- Prove:  
 $dappend(\text{diff}((a\ b\ .\ ?p),?p),\text{diff}((c\ d\ .\ ?q),?q),?w)$   
 $= [\text{?x}/(a\ b\ .\ ?p),$   
 $\text{?y}/?p,$   
 $?p/(c\ d\ .\ ?q),$   
 $?z/?q,$   
 $?w/\text{diff}((a\ b\ .\ (c\ d\ .\ ?q)),?q)]$
- Note that  $\text{diff}((a\ b\ .\ (c\ d\ .\ ?q)),?q)$  is equivalent to  $\text{diff}((a\ b\ c\ d\ .\ ?q),?q)$  – which is correct.



6.034 - Spring 03 • 17

**List Processing: reverse**

```
(define (reverse l)
 (define (reversel x y)
 (if (null? x)
 y
 (reversel (cdr x) (cons (car x) y))))
 (reversel l '()))
```

6.034 - Spring 03 • 18

**Slide 11.2.18**

Let's look at another example, first without difference lists and then with.

This is Scheme for a list reverse operation. It's a bit more complicated than the cases we've seen so far. To reverse the list, we need a temporary value to serve as an accumulator for the reversed list. That's what the y argument to the inner procedure is. y starts with the null list and we cons each of the elements of the input onto this list. When the first argument is null, we return the accumulated list.

**List Processing: reverse**

```
(define (reverse l)
 (define (reversel x y)
 (if (null? x)
 y
 (reversel (cdr x) (cons (car x) y))))
 (reversel l '()))
```

- $\text{reverse}(l, \text{rev}) := \text{reversel}(l, (), \text{rev})$
- $\text{reversel}((), ?y, ?y)$
- $\text{reversel}((?h . ?r), ?y, ?z) := \text{reversel}((?r), (?h . ?y), ?z)$

6.034 - Spring 03 • 19

**Slide 11.2.19**

We follow the same pattern in the logic program. We define the predicate reverse, with two arguments, the input and output lists, in terms of a three-place auxiliary predicate reversel, which introduces the accumulator and initializes it to nil. Note that if you reverse the first argument of reversel and append it to the second argument of reversel then that gives the answer to the original query.

Reverse1 is defined by two rules: in the base case when the first argument is nil, we simply equate the output list to the accumulator. In the general case, we set up a recursive subgoal with the cdr of the list, but we cons the car of the input list to the accumulator.

**List Processing: reverse**

```

1. reverse(?l, ?rev) :- reverse1(?l, (), ?rev)
2. reverse1((), ?y, ?y)
3. reverse1(?h . ?r), ?y, ?z) :-
reverse1(?r, (?h . ?y), ?z)

• Prove:

reverse((a b), ?v), Ans(?v)

[1,?l/(a b),?rev/?v]

a. reverse1((a b), (), ?v), Ans(?v)

[3,?h/a,?r/(b),?y/(),?z/?v]

b. reverse1((b), (a . ()), ?z), Ans(?v)

[3,?h/b,?r/(),?y/(a),?z/?v]

c. reverse1((), (b . (a)), ?v), Ans(?v)

[2,?v/?y,?y/(b a)]

d. Ans((b a))

```

6.034 - Spring 03 • 20

**Slide 11.2.20**

You can see the operation on a simple example here.

**List Processing: dreverse**

```

1. dreverse(?l, ?rev) :- dreverse1(?l, diff(?rev, ()))
2. dreverse1((), diff(?y, ?y))
3. dreverse1(?h . ?r), diff(?ys, ?ye)) :-
dreverse1(?r, diff(?ys, (?h . ?ye)))

```

6.034 - Spring 03 • 21

## 6.034 Notes: Section 12.1

### Slide 12.1.1

In this chapter, we take a quick survey of some aspects of natural language understanding. Our goal will be to capture the **meaning** of sentences in some detail. This will involve finding representations for the sentences that can be connected to more general knowledge about the world. This is in contrast to approaches to dealing with language that simply try to match textual patterns, for example, web search engines.

We will briefly provide an overview of the various levels and stages of natural language processing and then begin a more in-depth exploration of language syntax.

### 6.034 Artificial Intelligence

- Natural Language Understanding
  - Getting at the meaning of text and speech
  - Not just pattern matching
- Overview
- Syntax

tip · Spring 02 · 1

### Applications of NLU

- Interfaces to databases (weather, financial,...)
- Automated customer service (banking, travel,...)
- Voice control of machines (PCs, VCRs, cars,...)
- Grammar and style checking
- Summarization (news, manuals, ...)
- Email routing
- Smarter Web Search
- Translating documents
- Etc.

tip · Spring 02 · 2

### Slide 12.1.2

The motivation for the study of natural language understanding is twofold. One is, of course, that language understanding is one of the quintessentially human abilities and an understanding of human language is one of key steps in the understanding of human intelligence.

In addition to this fundamental long-term scientific goal, there is a pragmatic shorter-term engineering goal. The potential applications of in-depth natural language understanding by computers are endless. Many of the applications listed here are already available in some limited forms and there is a great deal of research aimed at extending these capabilities.

### Slide 12.1.3

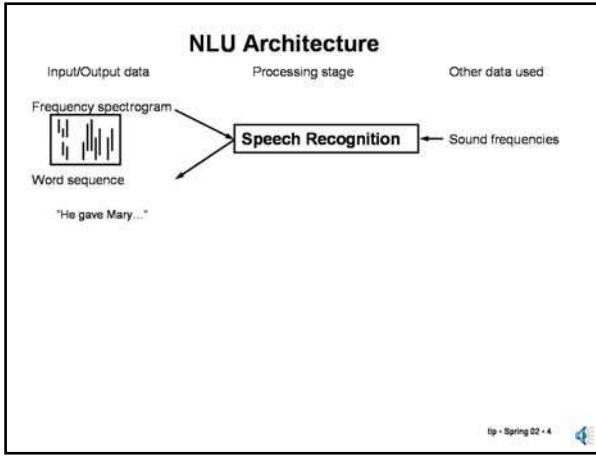
Language is an enormously complex process, which has been studied in great detail for a long time. The study of language is usually partitioned into a set of separate sub-disciplines, each with a different focus. For example, phonetics concerns the rules by which sounds (phonemes) combine to produce words. Morphology studies the structure of words: how tense, number, etc is captured in the form of the word. Syntax studies how words are combined to produce sentences. Semantics studies how the meaning of words are combined with the structure of a sentence to produce a meaning for the sentence, usually a meaning independent of context. Pragmatics concerns how context factors into the meaning (e.g. "it's cold in here") and finally there's the study of how background knowledge is used to actually understand the meaning of the utterances.

We will consider the process of understanding language as one of progressing through various "stages" or processing that break up along the lines of these various subfields. In practice, the processing may not be separated as cleanly as that, but the division into stages allows us to focus on one type of problem at a time.

### Levels of language analysis

- **Phonetics:** sounds → words
- **Morphology:** morphemes → words (jump+ed=jumped)
- **Syntax:** word sequence → sentence structure
- **Semantics:** sentence structure + word meaning → sentence meaning
- **Pragmatics:** sentence meaning + context → deeper meaning
- **Discourse and World Knowledge:** connecting sentences and background knowledge to utterances.

tip · Spring 02 · 3

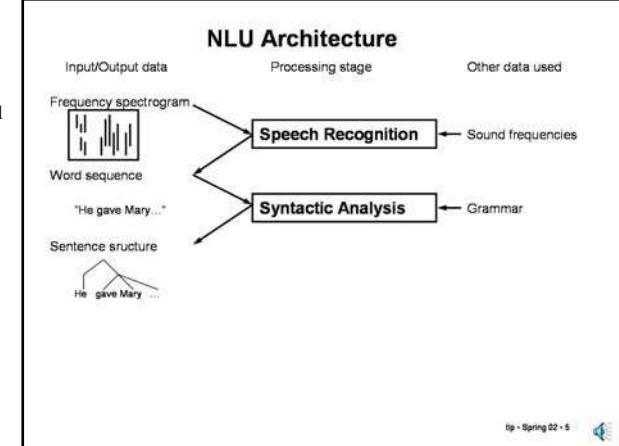
**Slide 12.1.4**

If one considers the problem of understanding speech, the first stage of processing is, conceptually, that of converting the spoken utterance into a string of words. This process is extremely complex and quite error prone and, today, cannot be solved without a great deal of knowledge about what the words are likely to be. But, in limited domains, fairly reliable transcription is possible. Even more reliability can be achieved if we think of this stage as producing a few alternative interpretations of the speech signal, one of which is very likely to be the correct interpretation.

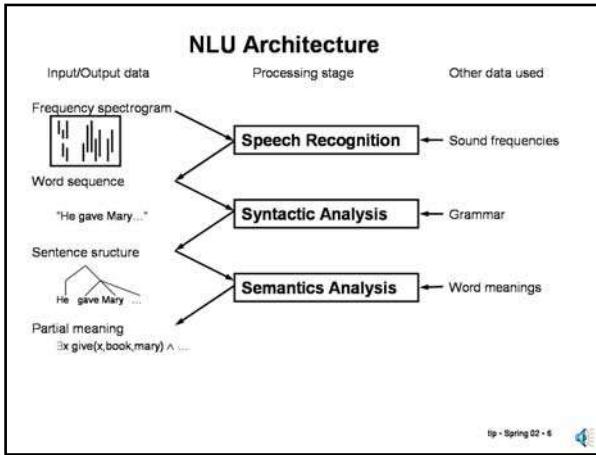
**Slide 12.1.5**

The next step is **syntax**, that is, computing the structure of the sentence, usually in terms of phrases, such as noun phrases, verb phrases and prepositional phrases. These nested phrases will be the basis of all subsequent processing. Syntactic analysis is probably the best developed area in computational linguistics but, nevertheless, there is no universally reliable "grammar of English" that one can use to parse sentences as well as trained people can. There are, however, a number of wide-coverage grammars available.

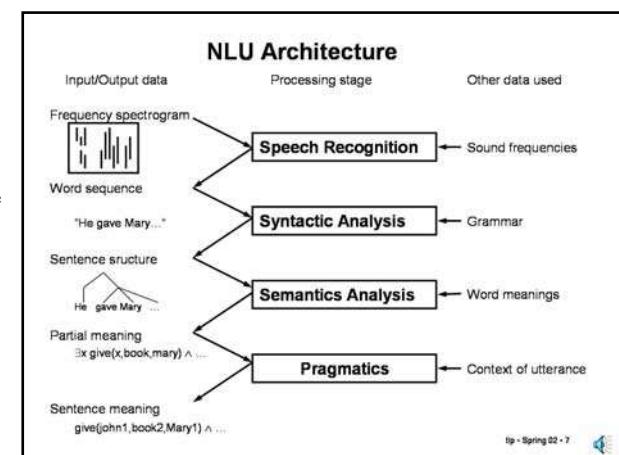
We will see later that, in general, there will not be a unique syntactic structure that can be derived from a sequence of words.

**Slide 12.1.6**

Given the sentence structure, we can begin trying to attach meaning to the sentence. The first such phase is known as **semantics**. The usual intent here is to translate the syntactic structure into some form of logical representation of the meaning - but without the benefit of context. For example, who is being referred to by a pronoun may not be determined at this point.

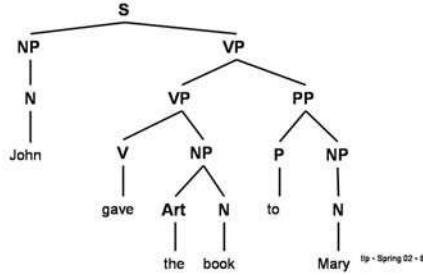
**Slide 12.1.7**

We will focus in this chapter on syntax and semantics, but clearly there is a great deal more work to be done before a sentence could be understood. One such step, sometimes known as **pragmatics**, involves among other things disambiguating the various possible senses of words, possible syntactic structures, etc. Also, trying to identify the referent of pronouns and descriptive phrases. Ultimately, we have to connect the meaning of the sentence with general knowledge in order to be able to act on it. This is by far the least developed aspect of the whole enterprise. In practice, this phase tends to be very application specific.



### Syntax

- Grammar captures legal structures in the language
- Parsing involves finding the legal structure(s) for a sentence
- The result is a **parse tree**



### Slide 12.1.8

In the rest of this section, we will focus on syntax. The description of the legal structures in a language is called a **grammar**. We'll see examples of these later. Given a sentence, we use the grammar to find the legal structures for a sentence. This process is called **parsing** the sentence. The result is one or more **parse trees**, such as the one shown here, which indicates that the sentence can be broken down into two **constituents**, a noun phrase and a verb phrase. The verb phrase, in turn, is composed of another verb phrase followed by a prepositional phrase, etc.

Our attempt to understand sentences will be based on assigning meaning to the individual constituents and then combining them to construct the meaning of the sentence. So, in this sense, the constituent phrases are the atoms of meaning.

### Slide 12.1.9

A grammar is typically written as a set of **rewrite rules** such as the ones shown here in blue. Bold-face symbols, such as S, NP and VP, are known as non-terminal symbols, in that they can be further re-written. The non-bold-face symbols, such as John, the and boy, are the words of the language - also known as the terminal symbols.

### Grammars

- We can think of grammars as a set of rules for rewriting strings of symbols:
  - **S → NP VP**
  - **NP → Name**
  - **NP → Art N**
  - **Name → John**
  - **Art → the**
  - **N → boy**

tip • Spring 02 • 9

### Grammars

- We can think of grammars as a set of rules for rewriting strings of symbols:
  - **S → NP VP**
  - **NP → Name**
  - **NP → Art N**
  - **Name → John**
  - **Art → the**
  - **N → boy**
- The string S can be rewritten as NP followed by VP

tip • Spring 02 • 10

### Slide 12.1.10

The first rule, **S → NP VP**, indicates that the symbol S (standing for sentence) can be rewritten as NP (standing for noun phrase) followed by VP (standing for verb phrase).

### Grammars

- We can think of grammars as a set of rules for rewriting strings of symbols:
  - **S → NP VP**
  - **NP → Name**
  - **NP → Art N**
  - **Name → John**
  - **Art → the**
  - **N → boy**
- The string S can be rewritten as NP followed by VP
- The string NP can be rewritten either as Name (which can be rewritten as **John**) or as an Art (such as **the**) followed by an N (such as **boy**).

tip • Spring 02 • 11

### Slide 12.1.11

The symbol NP, can be rewritten either as a Name or as an Art(icle), such as the, followed by a N (oun), such as boy.

### Grammars

- We can think of grammars as a set of rules for rewriting strings of symbols:
  - $S \rightarrow NP\ VP$
  - $NP \rightarrow Name$
  - $NP \rightarrow Art\ N$
  - $Name \rightarrow John$
  - $Art \rightarrow the$
  - $N \rightarrow boy$
- The string  $S$  can be rewritten as  $NP$  followed by  $VP$
- The string  $NP$  can be rewritten either as  $Name$  (which can be rewritten as *John*) or as an  $Art$  (such as *the*) followed by an  $N$  (such as *boy*).
- A sentence is legal if we can find a sequence of rewrite rules that, starting from the symbol  $S$ , generate the sentence. This is called **parsing** the sentence.

tip • Spring 02 • 12

**Slide 12.1.12**

If we can find a sequence of rewrite rules that will rewrite the initial  $S$  into the input sentence, the we have successfully parsed the sentence and it is legal.

Note that this is a search process like the ones we have studied before. We have an initial state,  $S$ , at any point in time, we have to decide which grammar rule to apply (there will generally be multiple choices) and the result of the application is some sequence of symbols and words. We end the search when the words in the sentence have been obtained or when we have no more rules to try.

**Slide 12.1.13**

Note that the successful sequence of rules applied to achieve the rewriting give us the parse tree.  
Note that this excludes any "wrong turns" we might have taken during the search.

### Good Grammars

- Differentiates between "correct" and "incorrect" sentences
  - The boy hit the ball      correct
  - The hit boy the ball (\*)      incorrect

tip • Spring 02 • 14

**Slide 12.1.14**

What makes a good grammar?

The primary criterion is that it differentiates correct sentences from incorrect ones. (By convention an asterisk next to a sentence indicates that it is not grammatical).

**Slide 12.1.15**

The other principal criterion is that it assigns "meaningful" structures to sentences. In our case, this literally means that it should be possible to assign meaning to the sub-structures. For example, a noun phrase will denote an object while a verb phrase will denote an event or an action, etc.

### Good Grammars

- Differentiates between "correct" and "incorrect" sentences
  - The boy hit the ball
  - The hit boy the ball (\*)
- Assigns meaningful structure to the sentences
  - (The boy) (hit the ball)
  - (The) (boy hit) (the ball)

tip • Spring 02 • 15

### Good Grammars

- Differentiates between "correct" and "incorrect" sentences
  - The boy hit the ball
  - The hit boy the ball (\*)
- Assigns meaningful structure to the sentences
  - (The boy) (hit the ball)
  - (The) (boy hit) (the ball)

tip • Spring 02 • 15

### Good Grammars

- Differentiates between "correct" and "incorrect" sentences
  - The boy hit the ball
  - The hit boy the ball (\*)
- Assigns meaningful structure to the sentences
  - (The boy) (hit the ball)
  - (The) (boy hit) (the ball)
- Compact and modular, e.g. all these NPs can be used in any context NPs are allowed:
 

|                      |                     |
|----------------------|---------------------|
| – NP → Name          | John                |
| – NP → Art N         | the boy             |
| – NP → Art Adj N     | the tall girl       |
| – NP → Art N that VP | the dog that barked |

lip - Spring 02 - 16

### Slide 12.1.16

Among the grammars that meet our principal criteria we prefer grammars that are compact, that is, have fewer rules and are modular, that is, define structures that can be re-used in different contexts - such as noun-phrase in this example. This is partly for efficiency reasons in parsing, but is partly because of Occam's Razor - the simplest interpretation is best.

### Slide 12.1.17

There are many possible types of grammars. The three types that are most common in computational linguistics are regular grammars, context-free grammars and context-sensitive grammars. These grammars can be arranged in a hierarchy (the Chomsky hierarchy) according to their generality. In this hierarchy, the grammars in higher levels fully contain those below and there are languages in the more general grammars not expressible in the less general grammars.

The least general grammar of some interest in computational linguistics are the **regular grammars**. These grammars are composed of rewrite rules of the form  $A \rightarrow x$  or  $A \rightarrow x B$ . That is, a non-terminal symbol can be rewritten as a string of terminal symbols or by a string of terminal symbols followed by a non-terminal symbol.

### Types of Grammars

- There's a hierarchy of grammar types that can be classified by their generality. Some common types in wide use (from less general to more general):
  - **Regular grammars** – Rules are of the form:
    - $A \rightarrow x$  or  $A \rightarrow x B$
  - **Context Free grammars** – Rules are of the form:
    - $A \rightarrow \gamma$

lip - Spring 02 - 18

### Slide 12.1.18

At the next level are the **context-free grammars**. In these grammars, a non-terminal symbol can be rewritten into any combination of terminal and non-terminal symbols. Note that since the non-terminal appears alone in the left-hand side (lhs) of the rule, it is re-written independent of the context in which it appears - and thus the name.

### Types of Grammars

- There's a hierarchy of grammar types that can be classified by their generality. Some common types in wide use (from less general to more general):
  - **Regular grammars** – Rules are of the form:
    - $A \rightarrow x$  or  $A \rightarrow x B$
  - **Context Free grammars** – Rules are of the form:
    - $A \rightarrow \gamma$
  - **Context Sensitive grammars** – Rules are of the form:
    - $\alpha A \beta \rightarrow \alpha \gamma \beta$

lip - Spring 02 - 19

### Slide 12.1.19

Finally, in **context-sensitive grammars**, we are allowed to specify a context for the rewriting operation.

There are even more general grammars (known as Type 0) which we will not deal with at all.

### Types of Grammars

- There's a hierarchy of grammar types that can be classified by their generality. Some common types in wide use (from less general to more general):
  - **Regular grammars** – Rules are of the form:
    - $A \rightarrow x$  or  $A \rightarrow x B$
  - **Context Free grammars** – Rules are of the form:
    - $A \rightarrow \gamma$
  - **Context Sensitive grammars** – Rules are of the form:
    - $\alpha A \beta \rightarrow \alpha \gamma \beta$

lip - Spring 02 - 19

### Types of Grammars

- Regular grammars cannot capture some of the nested structures of natural language. The language  $a^n b^n$  is not a regular language and there are legal sentences with that type of structure.

### Slide 12.1.20

The language of parenthesized expressions, that is,  $n$  left parens followed by  $n$  right parens is the classic example of a non-regular language that requires us to move to context-free grammars. There are legal sentences in natural languages whose structure is isomorphic to that of parenthesized expressions (the cat likes tuna; the cat the dog chased likes tuna; the cat the dog the rat bit chased likes tuna). Therefore, we need at least a context-free grammar to capture the structure of natural languages.

tip · Spring 02 · 20



### Slide 12.1.21

There have been several empirical proofs that there exist natural languages that have non-context-free structure.

### Types of Grammars

- Regular grammars cannot capture some of the nested structures of natural language. The language  $a^n b^n$  is not a regular language and there are legal sentences with that type of structure.
- Some constructions in some natural languages have also been shown not to be context free.

tip · Spring 02 · 21



### Types of Grammars

- Regular grammars cannot capture some of the nested structures of natural language. The language  $a^n b^n$  is not a regular language and there are legal sentences with that type of structure.
- Some constructions in some natural languages have also been shown not to be context free.
- But, much of the structure of natural languages can be captured in a context free language and we will restrict ourselves to context free grammars.

tip · Spring 02 · 22



### Slide 12.1.22

However, much of natural language can be expressed in context-free grammars extended in various ways. We will limit ourselves to this class.

### Slide 12.1.23

Here's an example of a context free grammar for a small subset of English. Note that the vertical bar is a short hand which can be read as "or"; it is a notation for combining multiple rules with identical left hand sides. Many variations on this grammar are possible but this illustrates the style of grammar that we will be considering.

### A Simple Context-Free Grammar

- $S \rightarrow NP\ VP$
- $S \rightarrow S\ Conjunction\ S$
- $NP \rightarrow Pronoun$
- $NP \rightarrow Name$
- $NP \rightarrow Article\ Noun$
- $NP \rightarrow Number$
- $NP \rightarrow NP\ PP$
- $NP \rightarrow NP\ RelClause$
- $VP \rightarrow Verb$
- $VP \rightarrow Verb\ NP$
- $VP \rightarrow Verb\ Adj$
- $VP \rightarrow VP\ PP$
- $PP \rightarrow Prep\ NP$
- $RelClause \rightarrow that\ VP$
- $Article \rightarrow the\ | a\ | an\ | this\ | that\ ...$
- $Preposition \rightarrow to\ | in\ | on\ | near\ ...$
- $Conjunction \rightarrow and\ | or\ | but\ ...$
- $Pronoun \rightarrow I\ | you\ | he\ | me\ | him\ ...$
- $Noun \rightarrow book\ | flight\ | meal\ ...$
- $Name \rightarrow John\ | Mary\ | Boston\ ...$
- $Verb \rightarrow book\ | include\ | prefer\ ...$
- $Adjective \rightarrow first\ | earliest\ | cheap\ ...$

tip · Spring 02 · 23



## Grammar Rules

- We can use our rule language to write grammar rules (we've seen this before).
- Assume words in sentence is represented as a set of facts:
  - (John 0 1)
  - (ran 1 2)
- Then a rule would be represented:
  - **S → NP VP**
  - $((S ?s1 ?s3) :- (NP ?s1 ?s2) (VP ?s2 ?s3))$
  - With, for example, ?s1=0, ?s2=1 and ?s3=2, r1 would match the sentence "John ran"

tip · Spring 02 · 24

## Slide 12.1.24

At this point, we should point out that there is a strong connection between these grammar rules that we have been discussing and the logic programming rules that we have already studied. In particular, we can write context-free grammar rules in our simple Prolog-like rule language.

We will assume that a set of facts are available that indicate where the particular words in a sentence start and end (as shown here). Then, we can write a rule such as S → NP VP as a similar Prolog-like rule, where each non-terminal is represented by a fact that indicates the type of the constituent and the start and end indices of the words.

## Slide 12.1.25

In the rest of this Chapter, we will write the rules in a simpler shorthand that leaves out the word indices. However, we will understand that we can readily convert that notation into the rules that our rule-interpreters can deal with.

## Grammar Rules

- We can use our rule language to write grammar rules (we've seen this before).
- Assume words in sentence is represented as a set of facts:
  - (John 0 1)
  - (ran 1 2)
- Then a rule would be represented:
  - **S → NP VP**
  - $((S ?s1 ?s3) :- (NP ?s1 ?s2) (VP ?s2 ?s3))$
  - With, for example, ?s1=0, ?s2=1 and ?s3=2, r1 would match the sentence "John ran"
- We will write these grammar rules in the following shorthand:
  - $((S) :- (NP) (VP))$
  - which would just generate the rule above.
- Rules indicating the category for particular words will be written:
  - $((NP) :- John)$

tip · Spring 02 · 26

## Slide 12.1.26

We can also use the same syntax to specify the word category of individual words and also turn these into rules.

## Grammar Rules

- We can use our rule language to write grammar rules (we've seen this before).
- Assume words in sentence is represented as a set of facts:
  - (John 0 1)
  - (ran 1 2)
- Then a rule would be represented:
  - **S → NP VP**
  - $((S ?s1 ?s3) :- (NP ?s1 ?s2) (VP ?s2 ?s3))$
  - With, for example, ?s1=0, ?s2=1 and ?s3=2, r1 would match the sentence "John ran"
- We will write these grammar rules in the following shorthand:
  - $((S) :- (NP) (VP))$
  - which would just generate the rule above.

tip · Spring 02 · 25

## Slide 12.1.27

We can make a small modification to the generated rule to keep track of the parse tree as the rules are being applied. The basic idea is to introduce a new argument into each of the facts which keeps track of the parse tree rooted at that component. So, the parse tree for the sentence is simply a list, starting with the symbol S, and whose other components are the trees rooted at the NP and VP constituents.

## Parsing

- We can construct a parse tree by modifying the generated rules slightly:
  - $((S (s ?np ?vp) ?s1 ?s3) :- (NP ?np ?s1 ?s2) (VP ?vp ?s2 ?s3))$
  - This rule now shows how to build the parse tree for the whole sentence (s ?np ?vp) from the parse trees for the constituent NP tree (bound to ?np) and VP tree (bound to ?vp).

tip · Spring 02 · 27

## Parsing

- We can construct a parse tree by modifying the generated rules slightly:
  - $((S (s ?np ?vp) ?s1 ?s3)) :- (NP ?np ?s1 ?s2) (VP ?vp ?s2 ?s3))$
  - This rule now shows how to build the parse tree for the whole sentence ( $s$  ?np ?vp) from the parse trees for the constituent NP tree (bound to ?np) and VP tree (bound to ?vp).
- This can all be generated automatically from the same shorthand notation:
  - $((S) :- (NP) (VP))$

tip · Spring 02 · 28

## Slide 12.1.28

This additional bit of bookkeeping can also be generated automatically from the shorthand notation for the rule.

## Slide 12.1.29

Note that given the logic rules from the grammar and the facts encoding a sentence, we can use chaining (either forward or backward) to parse the sentence. Let's look at this in more detail.

## Parsing

- We can construct a parse tree by modifying the generated rules slightly:
  - $((S (s ?np ?vp) ?s1 ?s3)) :- (NP ?np ?s1 ?s2) (VP ?vp ?s2 ?s3))$
  - This rule now shows how to build the parse tree for the whole sentence ( $s$  ?np ?vp) from the parse trees for the constituent NP tree (bound to ?np) and VP tree (bound to ?vp).
- This can all be generated automatically from the same shorthand notation:
  - $((S) :- (NP) (VP))$
- Given a set of rules (and word facts), we can use forward or backward chaining to parse a sentence.

tip · Spring 02 · 29

## Slide 12.1.30

A word on terminology. Parsers are often classified into **top-down** and **bottom-up** depending whether they work from the top of the parse tree down towards the words or vice-versa. Therefore, backward-chaining on the rules leads to a top-down parser, while forward-chaining, which we will see later, leads to a bottom-up parser. There are more sophisticated parsers that are neither purely top-down nor bottom-up, but we will not pursue them here.

## Top-Down vs Bottom-Up

- Simple parsers are often classified as **top-down** or **bottom-up** where top and bottom refer to the parse tree.
- Backwards chaining on the grammar rules we have seen is a **top-down** approach to parsing (starts with S and works towards the words).
- Forward chaining on the grammar rules is a **bottom up** approach (starts with the words and works towards S).

tip · Spring 02 · 30

## Slide 12.1.31

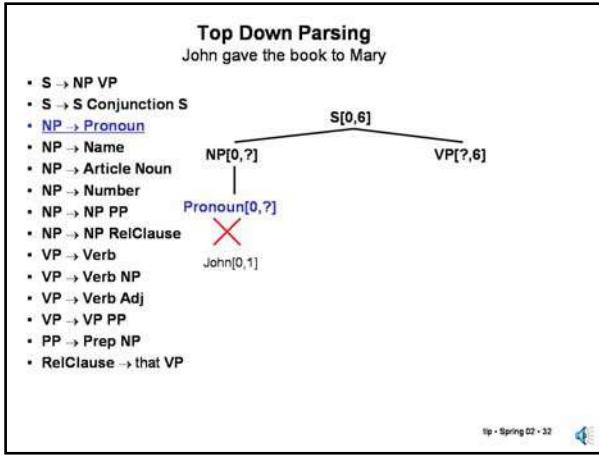
Let us look at how the sample grammar can be used in a top-down manner (backward-chaining) to parse the sentence "John gave the book to Mary". We start backchaining with the goal  $S[0,6]$ . The first relevant rule is the first one and so we generate two subgoals:  $NP[0,?]$  and  $VP[?,6]$ .

### Top Down Parsing

John gave the book to Mary

- $S \rightarrow NP VP$
- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$

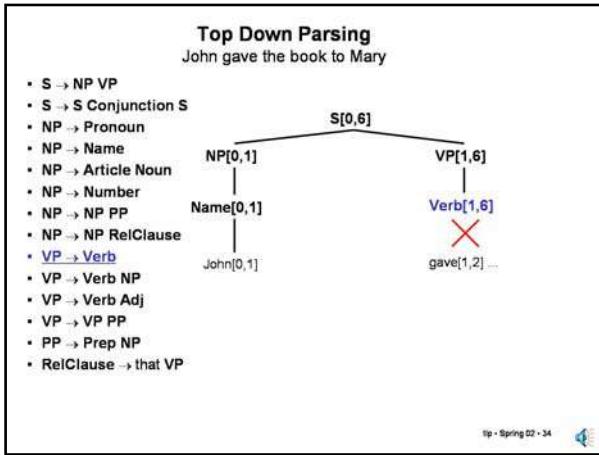
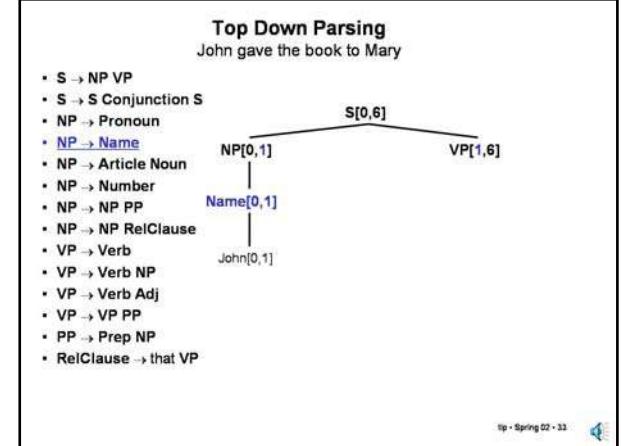
tip · Spring 02 · 31

**Slide 12.1.32**

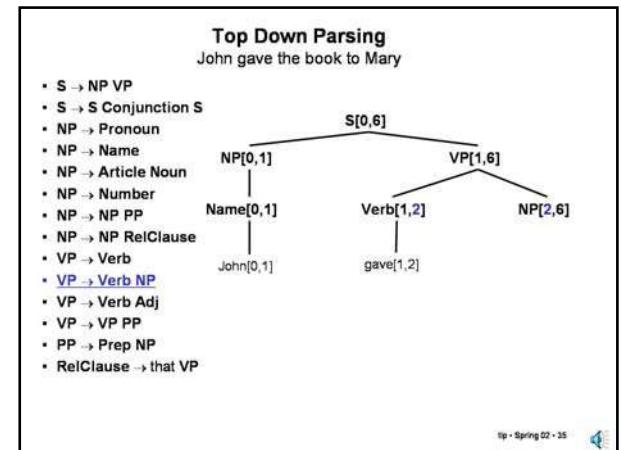
Assuming we examine the rules in order, we first attempt to apply the NP → Pronoun rule. But that will fail when we actually try to find a pronoun at location 0.

**Slide 12.1.33**

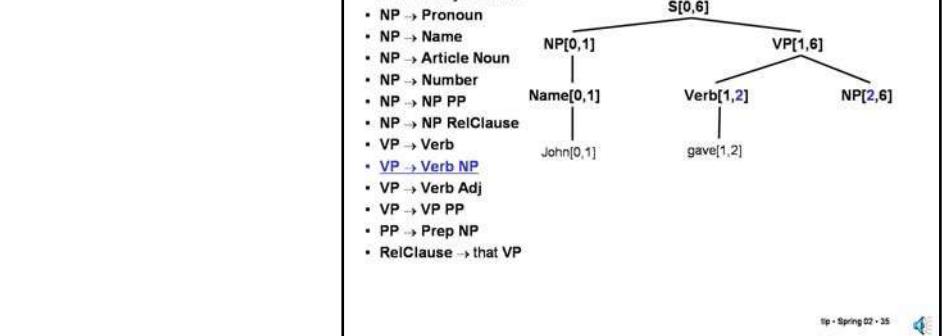
Then we try to see if NP → Name will work, which it does, since the first word is John and we have the rule that tells us that John is a Name. Note that this will also bind the end of the VP phrase and the start of the VP to be at position 1.

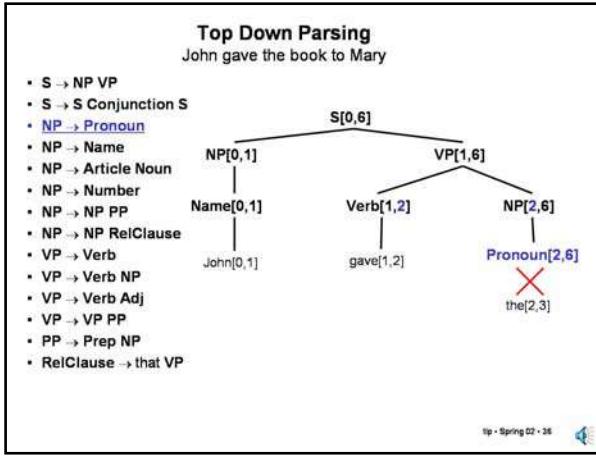
**Slide 12.1.34**

So, we move on to the pending VP. Our first relevant rule is VP → Verb, which will fail. Note, however, that there is a verb starting at location 1, but at this point we are looking for a verb phrase from positions 1 to 6, while the verb only goes from 1 to 2.

**Slide 12.1.35**

So, we try the next VP rule, which will look for a verb followed by a noun phrase, spanning from words 1 to 6. The Verb succeeds when we find "gave" in the input.

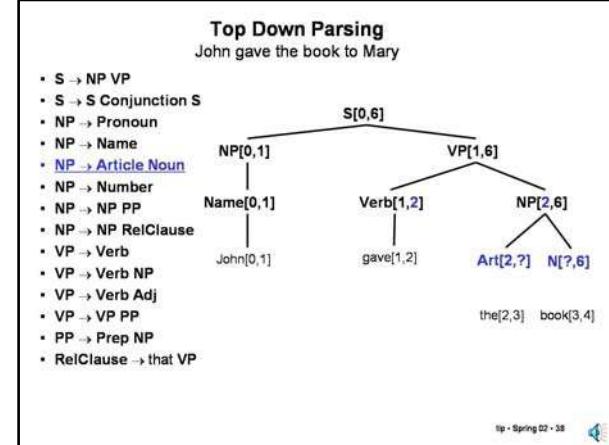


**Slide 12.1.36**

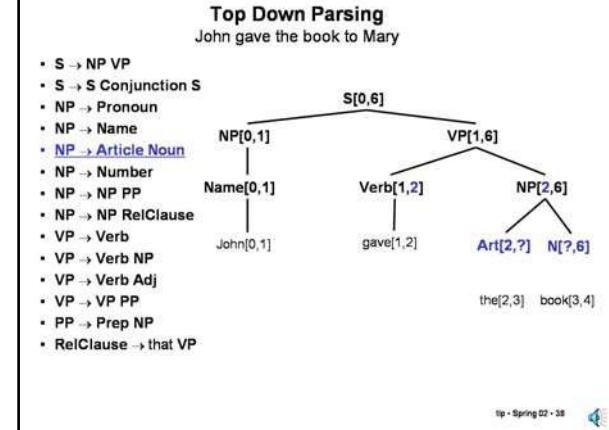
Now we try to find an NP starting at position 2. First we try the pronoun rule, which fails.

**Slide 12.1.37**

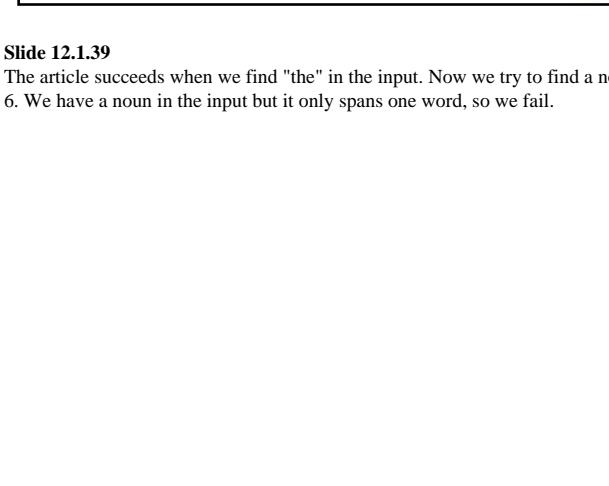
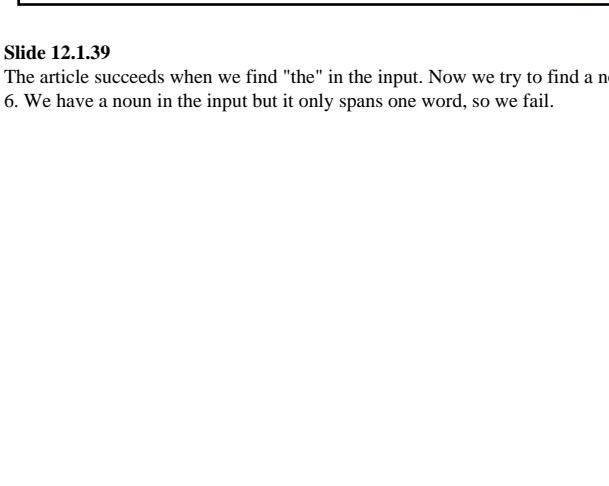
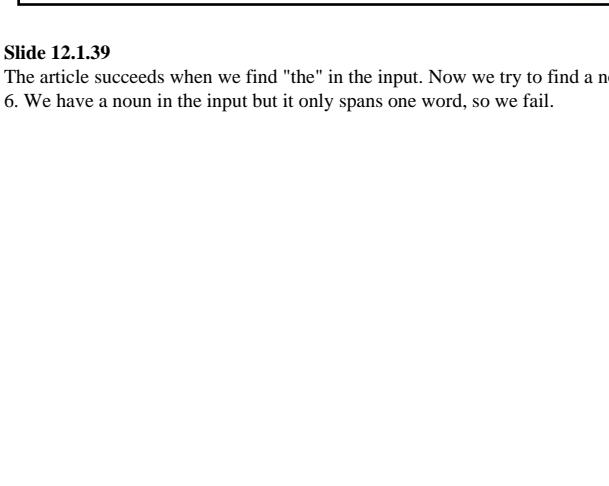
Then we try the name rule, which also fails.

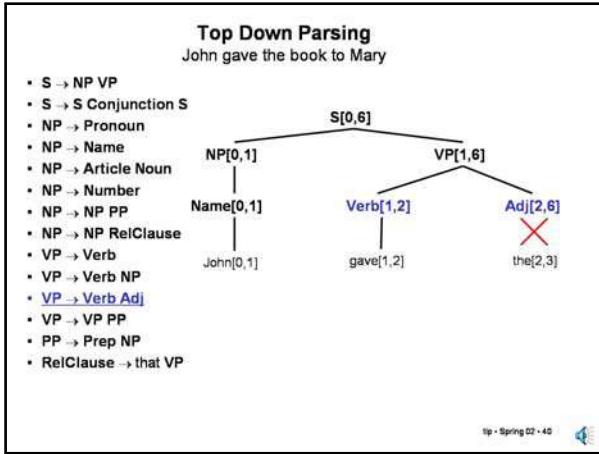
**Slide 12.1.38**

Then we try the article followed by a noun.

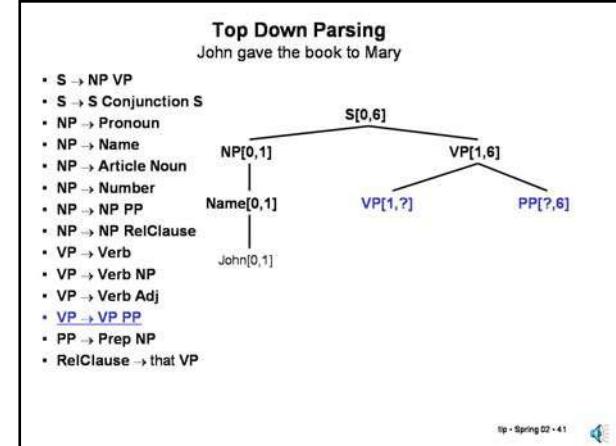
**Slide 12.1.39**

The article succeeds when we find "the" in the input. Now we try to find a noun spanning words 3 to 6. We have a noun in the input but it only spans one word, so we fail.

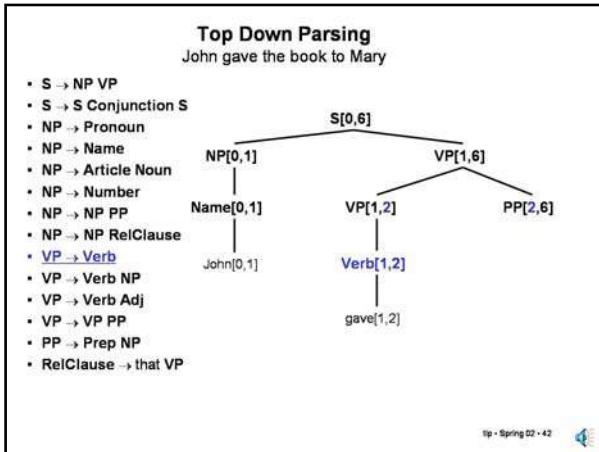


**Slide 12.1.40**

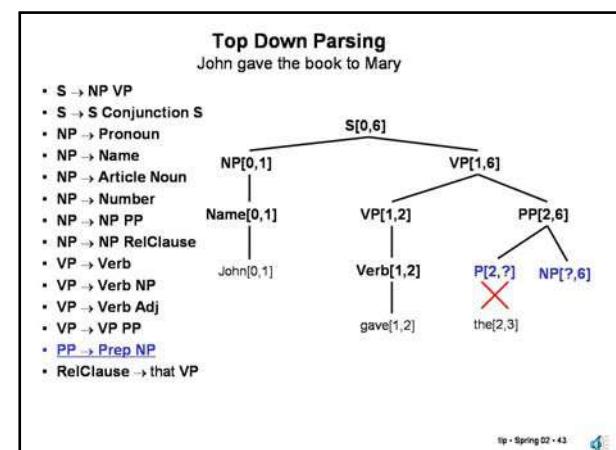
We eventually fail back to our choice of the VP rule and so we try the next VP rule candidate, involving a Verb followed by an adjective, which also fails.

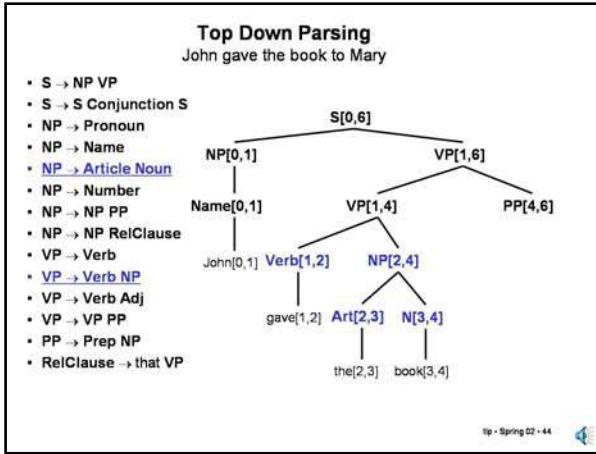
**Slide 12.1.42**

The first VP succeeds by finding the verb "gave", which now requires us to find a prepositional phrase starting at position 2.

**Slide 12.1.43**

We proceed to try to find a preposition at position 2 and fail.

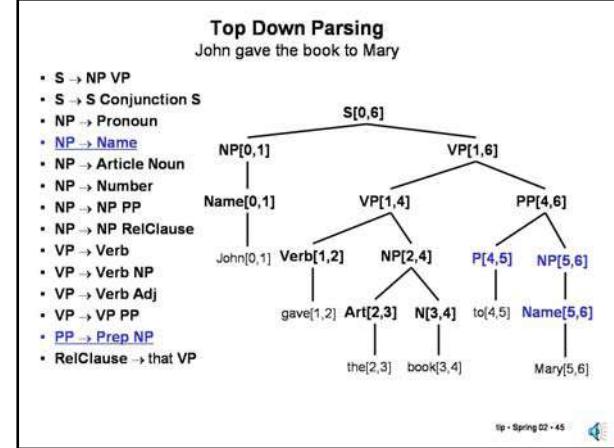


**Slide 12.1.44**

We fail back to trying an alternative rule (verb followed by NP) for the embedded VP, which now successfully parses "gave the book" and we proceed to look for a prepositional phrase in the range 4 to 6.

**Slide 12.1.45**

Which successfully parses, "to Mary", and the complete parse succeeds.

**Slide 12.1.46**

There are a number of problems with this top-down parsing strategy. One that substantially impacts efficiency is that rules are chosen without checking whether the next word in the input can possibly be compatible with that rule. There are simple extensions to the top-down strategy to overcome this difficulty (by keeping a table of constituent types and the lexical categories that can begin them).

A more substantial problem, is that rules such as NP → NP PP (left-branching rules) will cause an infinite loop for this simple top-down parsing strategy. It is possible to modify the grammar to turn such rules into right-branching rules - but that may not be the natural interpretation.

Note that the top-down strategy is carrying out a search for a correct parse and it ends up doing wasted work, repeatedly parsing parts of the sentence during its attempts. This can be avoided by building a table of parses that have been previously discovered (stored in the fact database) so they can be reused rather than re-discovered.

**Slide 12.1.47**

So far we have been using our rules together with our backtracking algorithm for logic programming to do top-down parsing. But, that's not the only way we can use the rules.

An alternative strategy starts by identifying any rules for which all the literals in their right hand side can be unified (with a single unifier) to the known facts. These rules are said to be **triggered**. For each of those triggered rules, we can add a new fact for the left hand side (with the appropriate variable substitution). Then, we repeat the process. This is known as **forward chaining** and corresponds to bottom-up parsing, as we will see next.

**Forward Chaining**

- Identify those rules whose antecedents (rhs) can be unified with the ground facts in the database.
- These rules are said to be **triggered**.
- Don't trigger rules that would not add new facts to the database. This avoids trivial infinite loops.
- For each triggered rule, apply the substitution to the consequent (lhs) of the rule and add the resulting literal to database.
- Repeat until no rule is triggered.

### Bottom Up Parsing

1.  $S \rightarrow NP VP$
- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$

John 1 gave 2 the 3 book 4 to 5 Mary 6

tip · Spring 02 · 48

### Slide 12.1.48

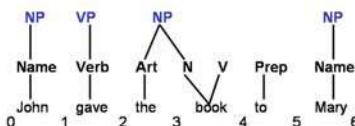
Now, let's look at bottom-up parsing. We start with the facts indicating the positions of the words in the input, shown here graphically.

### Slide 12.1.49

Note that all the rules indicating the lexical categories of the individual words, such as Name, Verb, etc, all trigger and can all be run to add the new facts shown here. Note that book is ambiguous, both a noun and a verb, and both facts are added.

### Bottom Up Parsing

1.  $S \rightarrow NP VP$
- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$



tip · Spring 02 · 50

### Slide 12.1.50

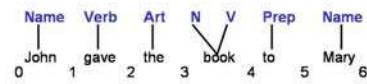
Now these three rules ( $NP \rightarrow \text{Name}$ ,  $VP \rightarrow \text{Verb}$  and  $NP \rightarrow \text{Art N}$ ) all trigger and can be run.

### Slide 12.1.51

Then, another three rules ( $S \rightarrow NP VP$ ,  $VP \rightarrow \text{Verb NP}$  and  $PP \rightarrow \text{Prep NP}$ ) trigger and can be run. Note that we now have an  $S$  fact, but it does not span the whole input.

### Bottom Up Parsing

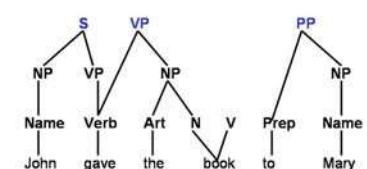
1.  $S \rightarrow NP VP$
- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$



tip · Spring 02 · 49

### Bottom Up Parsing

1.  $S \rightarrow NP VP$
- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$



tip · Spring 02 · 51

### Bottom Up Parsing

1.  $S \rightarrow NP VP$

- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$

tip · Spring 02 · 52

## Slide 12.1.52

Now, we trigger and run the S rule again as well as the VP->VP PP rule.

## Slide 12.1.53

Finally, we run the S rule covering the whole input and we can stop.

### Bottom Up Parsing

1.  $S \rightarrow NP VP$

- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$

Unused facts in red.  
tip · Spring 02 · 54

## Slide 12.1.54

Note that (not surprisingly) we generated some facts that did not make it into our final structure.

## Slide 12.1.55

Bottom-up parsing, like top-down parsing, generates wasted work in that it generates structures that cannot be extended to the final sentence structure. Note, however, that bottom-up parsing has no difficulty with left-branching rules, as top-down parsing did. Of course, rules with an empty right hand side can always be used, but this is not a fundamental problem if we require that triggering requires that a rule adds a new fact. In fact, by adding all the intermediate facts to the data base, we avoid some of the potential wasted work of a pure search-based bottom-up parser.

### Bottom Up Parsing

1.  $S \rightarrow NP VP$

- $S \rightarrow S \text{ Conjunction } S$
- $NP \rightarrow \text{Pronoun}$
- $NP \rightarrow \text{Name}$
- $NP \rightarrow \text{Article Noun}$
- $NP \rightarrow \text{Number}$
- $NP \rightarrow NP PP$
- $NP \rightarrow NP \text{ RelClause}$
- $VP \rightarrow \text{Verb}$
- $VP \rightarrow \text{Verb NP}$
- $VP \rightarrow \text{Verb Adj}$
- $VP \rightarrow VP PP$
- $PP \rightarrow \text{Prep NP}$
- $\text{RelClause} \rightarrow \text{that VP}$

tip · Spring 02 · 53

### Bottom Up Parsing

- Generates sub-trees that cannot be extended to S, for example, the interpretation of book as a verb in our example.
- No problem with left recursion, but potential problems with empty right hand side (empty antecedent).
- Saving all the facts makes this more efficient than a pure search-based bottom-up parser – does not have to redo sub-trees on failure.

tip · Spring 02 · 55

### Ambiguity

- A major problem in context-free parsing is **ambiguity**.
- Lexical class ambiguity: book is Noun and Verb
- Attachment ambiguity:
  - $S \rightarrow NP VP$
  - $NP \rightarrow NP PP$
  - $VP \rightarrow VP PP$
  - $VP \rightarrow Verb\ NP$
  - Mary (((saw (John)) (on the hill)) (with a telescope))
  - Mary ((saw (John)) (on the (hill with a telescope)))
  - Mary ((saw (John (on the hill))) (with a telescope))
  - Mary (saw ((John (on the hill)) (with a telescope)))
  - Mary (saw (John (on the (hill with a telescope))))
- Generate all parses and discard semantically inconsistent ones
- Preferences during parsing

tip • Spring 02 • 56

**Slide 12.1.56**

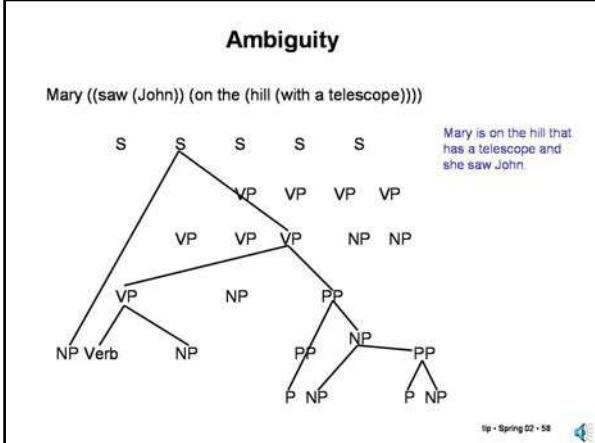
One of the key facts of natural language grammars is the presence of ambiguity of many types. We have already seen one simple example of **lexical ambiguity**, the fact that the word book is both a noun and a verb. There are many classic examples of this phenomenon, such as "Time flies like an arrow", where all of "time", "flies" and "like" are ambiguous lexical items. If you can't see the ambiguity, think about "time flies" as analogous to "fruit flies".

Perhaps a more troublesome form of ambiguity is known as **attachment ambiguity**. Consider the simple grammar shown here that allows prepositional phrases to attach both to VPs and NPs. So, the sentence "Mary saw John on the hill with a telescope" has five different structurally different parses, each with a somewhat different meaning (we'll look at them more carefully in a minute).

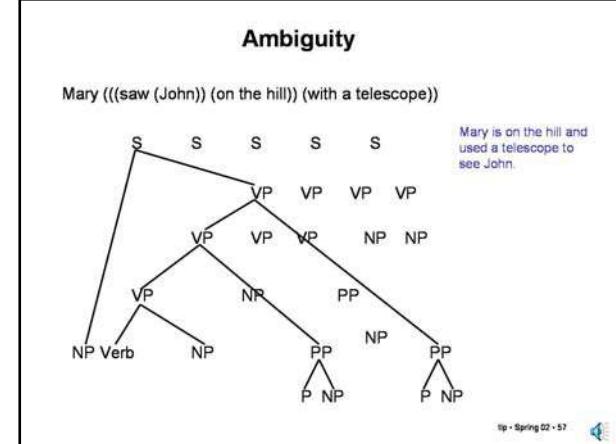
Basically we have two choices. One is to generate all the legal parses and let subsequent phases of the analysis sort them out or somehow to select one - possibly based on learned preferences based on examples. We will assume that we simply generate all legal parses.

**Slide 12.1.57**

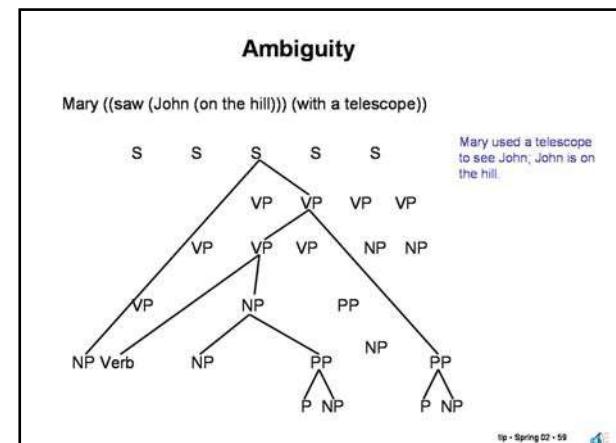
Here are the various interpretations of our ambiguous sentence. In this one, both prepositional phrases are modifying the verb phrase. Thus, Mary is on the hill she used a telescope to see John.

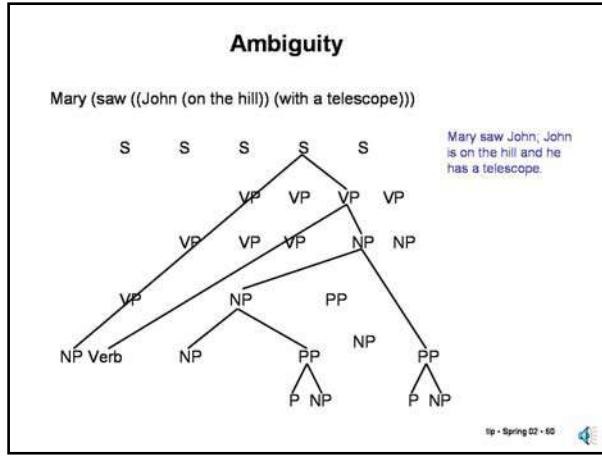
**Slide 12.1.58**

In this one, the telescope phrase has attached to the hill NP and so we are talking about a hill with a telescope. This whole phrase is modifying the verb phrase. Thus Mary is on the hill that has a telescope when she saw John.

**Slide 12.1.59**

In this one, the hill phrase is attached to John; this is clearer if you replace John with "the fool", so now Mary saw "the fool on the hill". She used a telescope for this, since that phrase is attached to the VP.



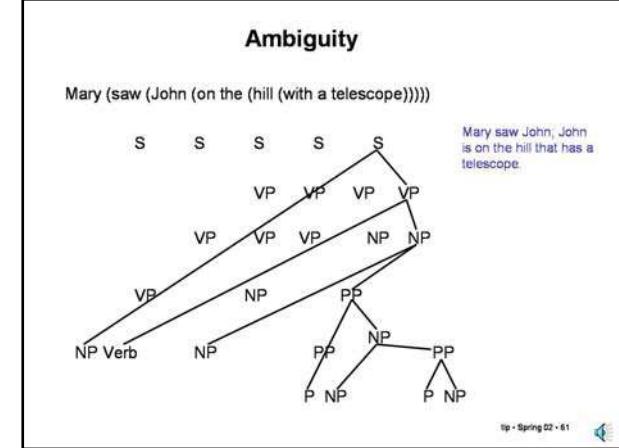
**Slide 12.1.60**

In this one, its the fool who is on the hill and who has the telescope that Mary saw.

**Slide 12.1.61**

Now its the fool who is on that hill with the telescope on it that Mary saw.

Note that the number of parses grows exponentially with the number of ambiguous prepositional phrases. This is a difficulty that only detailed knowledge of meaning and common usage can resolve.

**6.034 Notes: Section 12.2****Slide 12.2.1**

In this section we continue looking at handling the syntax of natural languages.

Thus far we have been looking at very simple grammars that do not capture nearly any of the complexity of natural language. In this section we take a quick look at some more complex issues and introduce an extension to our simple grammar rules, which will prove exceedingly useful both for syntactic and semantic analysis.

**6.034 Artificial Intelligence**

- Natural Language Understanding
  - Getting at the meaning of text and speech
  - Not just pattern matching
- Overview
- Syntax

## Additional Constraints

- **Agreement (Person/Number, Case)**

- Me read the book. [mismatched pronoun case - me]
- They reads the book. [mismatched number – they/reads]

### Slide 12.2.2

One important class of phenomena in natural language are agreement phenomena. For example, pronouns in a subject NP must be in the subjective case, such as, I, he, they, while pronouns in a direct object NP must be in the objective case, such as, me, him, them.

More interestingly, the person and number of the subject NP must match those of the verb. Some other languages, such as Spanish, require gender agreement as well. We will need some mechanism of capturing this type of agreement in our grammars.

lip • Spring 02 • 2



### Slide 12.2.3

Here is another form of agreement phenomena. Particular verbs require a particular combination of phrases as complements. For example, the verb put expects an NP, indicating the object being put, and it expects a prepositional phrase indicating the location.

In general verbs can be sub-categorized by their expected complements, called their sub-categorization frame. We need to find some way of capturing these constraints.

## Additional Constraints

- **Agreement (Person/Number, Case)**

- Me read the book. [mismatched pronoun case - me]
- They reads the book. [mismatched number – they/reads]

- **Sub-categorization**

- John put the box [in the corner]
- The verb put expects an NP and a location PP as complements

lip • Spring 02 • 3



## Additional Constraints

- **Agreement (Person/Number, Case)**

- Me read the book. [mismatched pronoun case - me]
- They reads the book. [mismatched number – they/reads]

- **Sub-categorization**

- John put the box [in the corner]
- The verb put expects an NP and a location PP as complements

- **Long-distance dependencies (movement)**

- Wh-movement:
  - The big man hurriedly put the blue ball in the red bo
  - What did the big man hurriedly put [ NP gap ] in the red box?
  - Where did the big man hurriedly put the blue ball [ PP gap ]?

lip • Spring 02 • 4



### Slide 12.2.4

Another important class of phenomena can be understood in terms of the movement of phrases in the sentence. For example, we can think of a question as moving a phrase in the corresponding declarative sentence to the front of the sentence in the form of a wh-word, leaving a sort of "hole" or "gap" in the sentence where a noun phrase or prepositional phrase would have normally appeared. We will look at this type of sentence in more detail later.

## Enforcing Constraints – New categories

- We can define NP/s, NP/p, VP/s, VP/p to handle agreement in number (singular, plural) between NP and VP.

- $S \rightarrow NP/s\ VP/s$
- $S \rightarrow NP/p\ VP/p$

lip • Spring 02 • 5



### Enforcing Constraints – New categories

- We can define NP/s, NP/p, VP/s, VP/p to handle agreement in number (singular, plural) between NP and VP.
  - $S \rightarrow NP/s VP/s$
  - $S \rightarrow NP/p VP/p$
- Similarly, we could handle pronoun case
  - $S \rightarrow NP/subj VP$
  - $NP/subj \rightarrow Pronoun/subj$
  - $NP/obj \rightarrow Pronoun/obj$
  - $VP \rightarrow Verb NP/obj$
  - $Pronoun/subj \rightarrow I | he | she | they$
  - $Pronoun/obj \rightarrow me | him | her | them$

tip • Spring 02 • 6

**Slide 12.2.6**

Note that we could extend this approach to handle the pronoun case agreement example we introduced earlier.

**Slide 12.2.7**

However, there is a substantial problem with this approach, namely the proliferation of non-terminals and the resulting proliferation of rules. Where we had a rule involving an NP before, we now need to have as many rules as there are variants of NP. Furthermore, the distinctions multiply. That is, if we want to tag each NP with two case values and two number values and 3 person values, we need 12 NP subclasses. This is not good...

### Enforcing Constraints – Feature values

- Associate **feature values** with each constituent
  - $(NP <number> <person> <case>)$
  - $(VP <number> <person>)$
  - $(S) \rightarrow (NP ?n ?p subj) (VP ?n ?p)$
  - $(VP ?n ?p) \rightarrow (Verb ?n ?p) (NP ?x ?y obj)$
  - $(NP ?n ?p ?c) \rightarrow (Pronoun ?n ?p ?c)$
  - $(Pronoun sg 1 subj) \rightarrow I$
  - $(Pronoun sg 1 obj) \rightarrow Me$
  - $(Pronoun pl 3 subj) \rightarrow They$
  - $(Verb sg 1) \rightarrow am$
  - $(Verb sg 3) \rightarrow is$
  - $(Verb pl ?p) \rightarrow are$

tip • Spring 02 • 8

**Slide 12.2.8**

An alternative approach is based on exploiting the unification mechanism that we have used in our theorem provers and rule-chaining systems. We can introduce variables to each of the non-terminals which will encode the values of a set of **features** of the constituent. These features, for example, can be number, person, and case (or anything else).

Now, we can enforce agreement of values within a rule by using the same variable name for these features - meaning that they have to match the same value. So, for example, the S rule here says that the number and person features of the NP have to match those of the VP. We also constrain the value of the case feature in the subject NP to be "subj".

In the VP rule, note that the number and person of the verb does not need to agree with that of the direct object NP, whose case is restricted to be "obj".

Most of the remaining rules encode the values for these features for the individual words.

**Slide 12.2.9**

The last rule indicates that the verb "are" is plural and agrees with any person subject. We do this by introducing a variable instead of a constant value. This is straightforward except that in the past we have restricted our forward chainer to dealing with assertions that are "ground", that is, that involve no variables. To use this rule in a forward-chaining style, we would have to relax that condition and operate more like the resolution theorem prover, in which the database contains assertions which may contain variables.

In fact, we could just use the resolution theorem prover with a particular set of preferences for the order of doing resolutions which would emulate the performance of the forward chainer.

### Enforcing Constraints – Feature values

- Associate **feature values** with each constituent
  - $(NP <number> <person> <case>)$
  - $(VP <number> <person>)$
  - $(S) \rightarrow (NP ?n ?p subj) (VP ?n ?p)$
  - $(VP ?n ?p) \rightarrow (Verb ?n ?p) (NP ?x ?y obj)$
  - $(NP ?n ?p ?c) \rightarrow (Pronoun ?n ?p ?c)$
  - $(Pronoun sg 1 subj) \rightarrow I$
  - $(Pronoun sg 1 obj) \rightarrow Me$
  - $(Pronoun pl 3 subj) \rightarrow They$
  - $(Verb sg 1) \rightarrow am$
  - $(Verb sg 3) \rightarrow is$
  - $(Verb pl ?p) \rightarrow are$
- Note that " $(Verb pl ?p) \rightarrow are$ " translates as:
  - $(Verb pl ?p ?s0 ?s1) :- (are ?s0 ?s1)$
  - Simple forward chainer would not support this rule (unbound var) – would need extension. Backward chainer has no problem with this.

tip • Spring 02 • 9

**Enforcing Constraints – Feature values****Associate feature values with each constituent**

- $(NP <number> <person> <case>)$
  - $(VP <number> <person>)$
  - $(S) \rightarrow (NP ?n ?p subj) (VP ?n ?p)$
  - $(VP ?n ?p) \rightarrow (Verb ?n ?p) (NP ?x ?y obj)$
  - $(NP ?n ?p ?c) \rightarrow (Pronoun ?n ?p ?c)$
  - $(Pronoun sg 1 subj) \rightarrow I$
  - $(Pronoun sg 1 obj) \rightarrow Me$
  - $(Pronoun pl 3 subj) \rightarrow They$
  - $(Verb sg 1) \rightarrow am$
  - $(Verb sg 3) \rightarrow is$
  - $(Verb pl ?p) \rightarrow are$
- Note that " $(Verb pl ?p) \rightarrow are$ " translates as:**
- $(Verb pl ?p ?s0 ?s1) :- (are ?s0 ?s1)$
  - Simple forward chainer would not support this rule (unbound var) – would need extension. Backward chainer has no problem with this.

tip • Spring 02 • 9

### Verb Sub-Categorization

- Features can also be used to capture the relationship between a verb and its complements. For example:
  - "give" can take an NP (direct object) and a PP starting with "to" (recipient) – "give the book to Mary" – or two NPs – "give Mary the book". Note the order of object and recipient is reversed in these two forms.
  - "place" can take an NP and a PP indicating a location – "place the ball in the box."
  - "tell" can take an NP and an infinitive verb phrase – "told the man to go" or it can also take two NPs – "told the man a secret."
  - "find" usually takes a single NP – "find the door"
- Note that the intended role of the phrase is signaled by its type and position relative to the verb.

tip · Spring 02 · 10

### Slide 12.2.10

Let's look now at verb sub-categorization. We have seen that verbs have one or more particular combinations of complement phrases that are required to be present in a legal sentence. Furthermore, the role that the phrase plays in the meaning of the sentence is determined by its type and position relative to the verb.

We will see that we can use feature variables to capture this.

### Slide 12.2.11

One simple approach we can use is simply to introduce a rule for each combination of complement phrases. The Verb would indicate this complex feature and only the rule matching the appropriate feature value would be triggered. However, we would need a large number of rules, since there are many different such combinations of phrases possible. So, a more economical approach would be desirable.

tip · Spring 02 · 11

### Verb Sub-Categorization

- We can encode the verb sub-categorization constraints through a set of rules like this:
  1.  $(VP) \rightarrow (Verb (NP)) (NP)$
  2.  $(VP) \rightarrow (Verb (NP NP)) (NP) (NP)$
  3.  $(VP) \rightarrow (Verb (NP PP)) (NP) (PP)$
  4.  $(Verb (NP)) \rightarrow find$
  5.  $(Verb (NP NP)) \rightarrow gave$
  6.  $(Verb (NP PP)) \rightarrow gave$
  7.  $(Verb (NP PP)) \rightarrow Put$
- We would need a VP rule for each sub-categorization "frame", that is, combination of verb complements.

### Slide 12.2.12

Here we see an alternative implementation that only requires one rule per complement phrase type (as opposed to combinations of such types). The basic idea is to use the rules to implement a recursive process for scanning down the list of expected phrases.

In fact, this set of rules can be read like a Scheme program. Rule 1 says that if the subcat list is empty then we do not expect to see any phrases following the VP, just the end of the sentence. So, this rule will generate the top-level structure of the sentence.

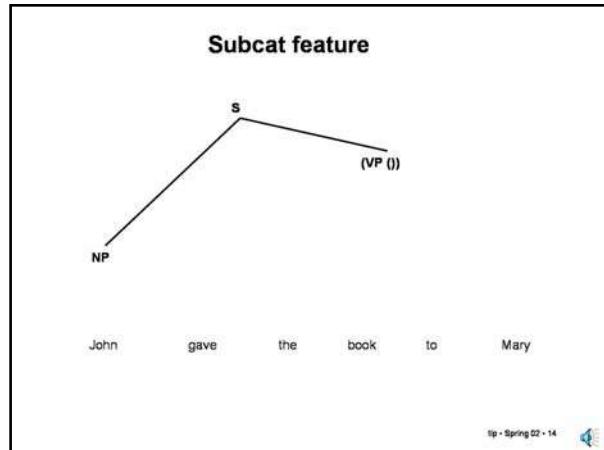
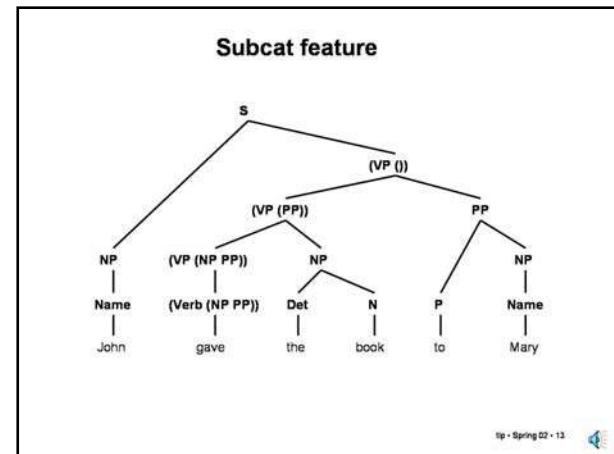
Rules 2 and 3 handle the cases of a noun phrase or propositional phrase expected after the verb phrase. If you look closely, these rules are a bit strange because they are rewriting a simpler problem, a verb phrase with a subcat list, into what appears to be a more complex phrase, namely another verb phrase with a longer subcat list which is followed by a phrase of the appropriate type. Imagine that the subcat list were null, then rule 2 expands such a VP into another VP where the subcat list contains an NP and this VP is followed by an actual NP phrase. Rule 3 is similar but for prepositional phrases.

tip · Spring 02 · 12

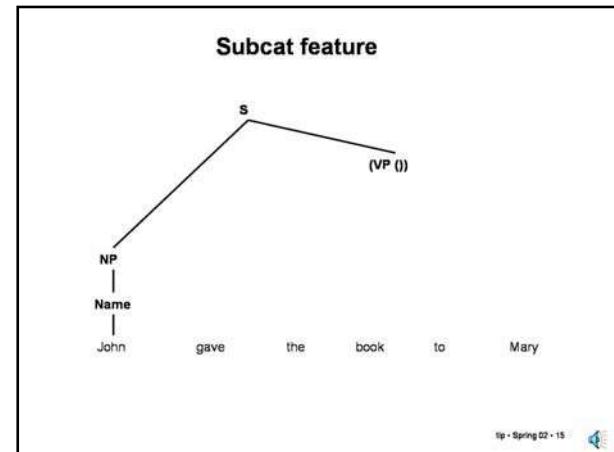
The idea of these rules is that they will expand the null subcat list into a longer list, at each point requiring that we find the corresponding type of phrase in the input. The base case that terminates the recursion is rule 5, which requires finding a verb in the input with the matching subcat list. An example should make this a bit clearer.

**Slide 12.2.13**

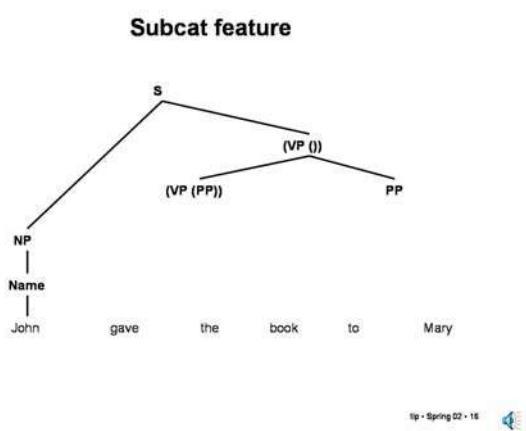
Here's an example of using this grammar fragment. You can see the recursion on the VP argument (the subcat feature) in the three nested VPs, ultimately matching a Verb with the right subcategorization frame. Let's look in detail at how this happens.

**Slide 12.2.14**

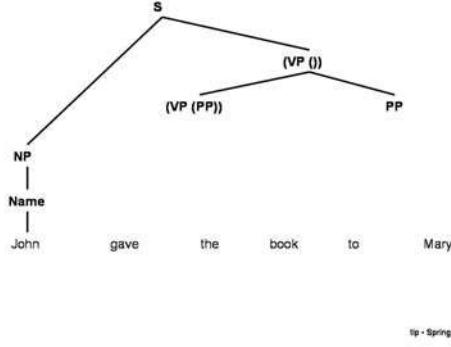
We start with the top-level S rule, which creates an NP subgoal and a VP subgoal with the ?subcat feature bound to the empty list.

**Slide 12.2.15**

The NP rule involving a name succeeds with the first input word: John.

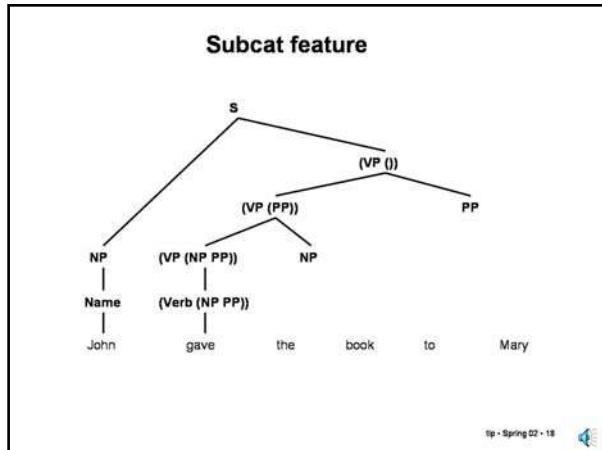
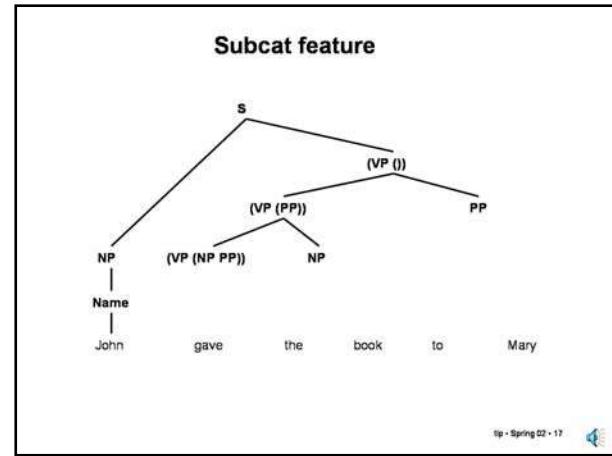
**Slide 12.2.16**

In practice, we would have to try each of the VP rules in order until we found the one that worked to parse the sentence. Here, we have just picked the correct rule, which says that the VP will end with a prepositional phrase PP. This is rule 3 in the grammar. Note that this involves binding the ?subcat variable to (). Note that this creates a new VP subgoal with ?subcat bound to (PP).

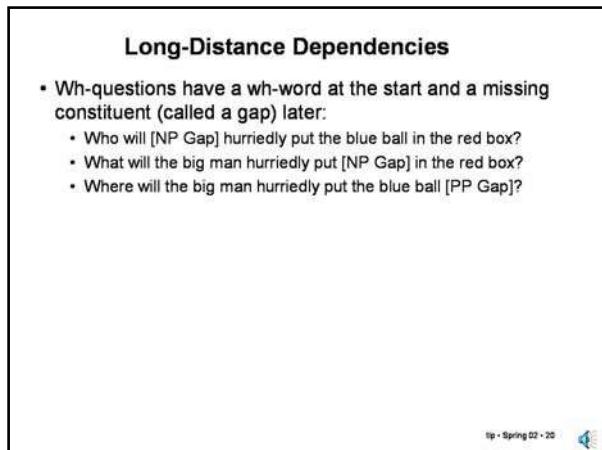
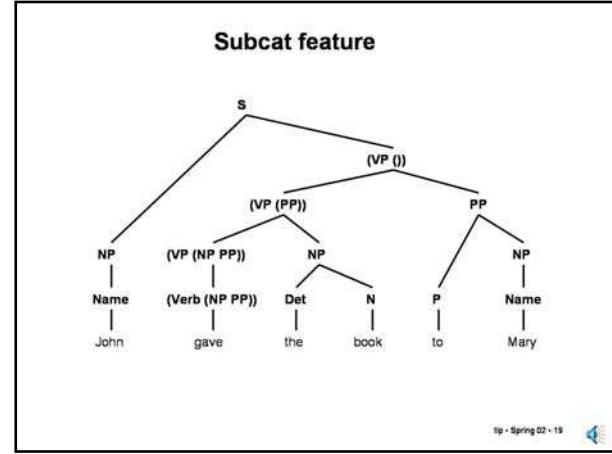


**Slide 12.2.17**

We now pick rule 2 with ?subcat bound to (PP). This rule will look for an NP in front of the PP and create a new VP subgoal with ?subcat bound to (NP PP).

**Slide 12.2.19**

The rest of the parse of the sentence can proceed as normal.

**Slide 12.2.20**

Let's consider how to parse wh-questions, which have a wh-word (what, where, who, when) at the start and a missing constituent phrase, an NP or a PP in the rest of the sentence. The missing phrase is called a **gap**. In these questions, the "will" is followed by a sentence that follows the usual rules for sentences except that in each case, the sentence is missing a phrase, indicated by the brackets.

We would like to parse these sentences without having to define a special grammar to handle missing constituents. We don't have to define a new sentence grammar that allows dropping the subject NP and another one that allows dropping an object NP or an object PP. Instead, we would like to generalize our rules for declarative sentences to handle this situation.

**Slide 12.2.21**

The same can be said about a relative clause, which is basically a sentence with a missing NP (which refers to the head noun).

**Long-Distance Dependencies**

- Wh-questions have a wh-word at the start and a missing constituent (called a gap) later:
  - Who will [NP Gap] hurriedly put the blue ball in the red box?
  - What will the big man hurriedly put [NP Gap] in the red box?
  - Where will the big man hurriedly put the blue ball [PP Gap]?
- A similar situation holds for relative clauses:
  - The person that [NP Gap] hit the ball
  - The person that John thinks [NP Gap] hit the ball

tip · Spring 02 · 21

**Long-Distance Dependencies**

- Wh-questions have a wh-word at the start and a missing constituent (called a gap) later:
  - Who will [NP Gap] hurriedly put the blue ball in the red box?
  - What will the big man hurriedly put [NP Gap] in the red box?
  - Where will the big man hurriedly put the blue ball [PP Gap]?
- A similar situation holds for relative clauses:
  - The person that [NP Gap] hit the ball
  - The person that John thinks [NP Gap] hit the ball
- We can handle these phenomena using features
  - New arguments for constituents: (VP InG OutG ...) where:
    - (VP InG OutG) specifies a difference list, analogous to diff(InG,OutG), of missing constituents (gaps) in the verb phrase.

tip · Spring 02 · 22

**Slide 12.2.22**

We saw when we studied logic programs that we could manipulate lists using a representation called **difference lists**. You can see some examples of this representation of a simple list with three elements here. The basic idea is that we can represent a list by two variables, one bound to the beginning of the list and the other to the end of the list. Note that if the first and second variables are bound to the same value, then this represents an empty list.

In the grammars we will be dealing with in this chapter, we will only need to represent lists of at most length one.

Also, note, that the crucial thing is having two variable values, the symbol diff doesn't actually do anything. In particular, we are **not** "calling" a function called diff. It's just a marker used during unification to indicate the type of the variables.

**Review: Difference Lists**

- A list can be represented as the difference between two lists, which we have written as `diff(L1, L2)`
- For example, (a b c) can be written as any of these:
  - `diff( (a b c) , () )`
  - `diff( (a b c d) , (d) )`
  - `diff( (a b c d e) , (d e) )`
  - `diff( (a b c . ?x) , ?x )`
- The empty list can be written as any of these:
  - `diff(?x, ?x)`
  - `diff((a) (a))`
  - `diff((a b c) (a b c))`

tip · Spring 02 · 23

**Slide 12.2.24**

Let's look at a piece of grammar in detail so that we can understand how gaps are treated. We will look at a piece of the grammar for relative clauses. Later, we look at a bigger piece of this grammar.

The key idea, as we've seen before, is that a relative clause is a sentence that is missing a noun phrase, maybe the subject noun phrase or an object noun phrase. The examples shown here illustrate a missing object NP, as in, "John called the man" becoming "that John called". Or, a missing subject NP, as in, "The man called John" becoming "that called John".

**Gaps for Relative Clauses**

- Parse relative clauses such as:
  - "... that John called"
  - "... that called John"
- as "that S[NP]" where S[NP] is a sentence with an NP gap.

tip · Spring 02 · 24



**Slide 12.2.25**

In our grammar, we are going to add two variables to the sentence literal, which will encode a difference list of gaps. This literal behaves exactly as a difference list, even though we don't show the "diff" symbol. The examples we show here are representing a list of one element. The one element is a list (gap NP) or (gap PP). This convention is arbitrary, we could have made the elements of this list be "foo" or "bar" as long as we used them consistently in all the rules. We have chosen to use a mnemonic element to indicate that it is a gap and the type of the missing component.

Now, if we want to have a rule that says that a relative clause is the word "that" followed by a sentence with a missing NP, we can do that by using this

```
(RelClause) :- "that" (S ((gap NP)) ())
```

**Gaps for Relative Clauses**

- Parse relative clauses such as:
  - "... that John called"
  - "... that called John"
- as "that S[NP]" where S[NP] is a sentence with an NP gap.
- In the grammar:
  - (S ((gap NP)) ()) stands for a sentence with an NP gap.
  - (S ((gap PP)) ()) stands for a sentence with a PP gap.

lip · Spring 02 · 25

**Gaps for Relative Clauses**

- Parse relative clauses such as:
  - "... that John called"
  - "... that called John"
- as "that S[NP]" where S[NP] is a sentence with an NP gap.
- In the grammar:
  - (S ((gap NP)) ()) stands for a sentence with an NP gap.
  - (S ((gap PP)) ()) stands for a sentence with a PP gap.
- Note that both
  - (S ()())
  - (S ((gap NP)) ((gap NP)))
- represent an empty list, that is, no missing phrase.

lip · Spring 02 · 26

**Slide 12.2.26**

And, just as we saw with other difference lists, if both gap variables are equal then this represents an empty list, meaning that there is no missing component. That is, the sentence needs to be complete to be parsed. In this way, we can get the behavior we had before we introduced any gap variables.

**Slide 12.2.27**

Here is a small, very simplified, grammar fragment for a sentence that allows a single NP gap. We have added two gap variables to each sentence (S), noun phrase (NP) and verb phrase (VP) literal.

The first rule has the same structure of the append logic program that we saw in the last chapter. It says that the sentence gap list is the append of the NP's gap list and the VP's gap list. This basically enforces conservation of gaps. So, if we want a sentence with one gap, we can't have a gap both in the NP and the VP. We'll see this work in an example.

The second rule shows that if there is a gap in the VP, it must be the object NP. The third rule is for a non-gapped NP, which in this very simple grammar can only be a name.

The last rule is the one that is actually used to "recognize" a missing NP. Note that this rule has no antecedent and so can be used without using any input from the sentence. However, it will only be used where a gap is allowed since the gap variables in the rule need to match those in the goal.

We have left out the rules for specific words, for example, that John is a name or that called is a transitive verb.

Note that, in general, there will be other variables associated with the grammar literals that we show here, for example, to enforce number agreement. We are not showing all the variables, only the gap variables, so as to keep the slides simpler.

**Grammar Gaps**

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - Gap could be in the NP or the VP but not both.
- (VP ?vpg0 ?vpg1) :- (Verb/tz) (NP ?vpg0 ?vpg1)
  - Only the NP can be gapped.
- (NP ?npg0 ?npg0) :- (Name) ; no gap is allowed
- (NP ((gap NP)) () ) ; accepts an NP gap.

lip · Spring 02 · 27

## Grammar Gaps

Showing only variables associated with handling gaps

- $(S \ ?sg0 \ ?sg2) :- (NP \ ?sg0 \ ?sg1) \ (VP \ ?sg1 \ ?sg2)$ 
  - Gap could be in the NP or the VP but not both.
- $(VP \ ?vpg0 \ ?vpg1) :- (\text{Verb/tr}) \ (NP \ ?vpg0 \ ?vpg1)$ 
  - Only the NP can be gapped.
- $(NP \ ?png0 \ ?png0) :- (\text{Name})$  ; no gap is allowed
- $(NP \ ((\text{gap } NP)) \ ())$  ; accepts an NP gap.

(S ((gap NP)) ())

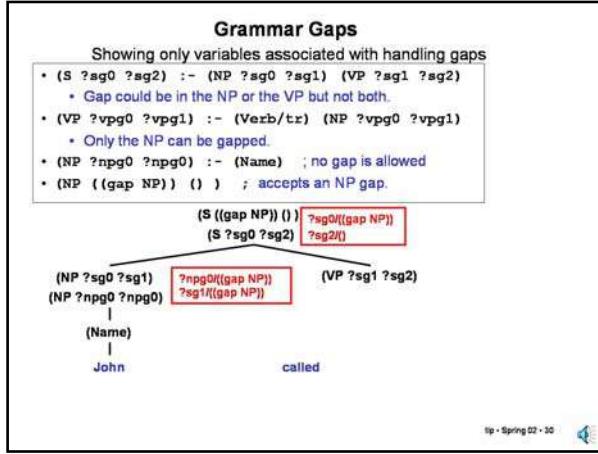
Slide 12.2.28

Let's see how this grammar could be used to parse a sentence with a missing object NP, such as "John called" that would come up in the relative clause "that John called".

The goal would be to find a sentence with an NP gap and so would be the S literal shown here.

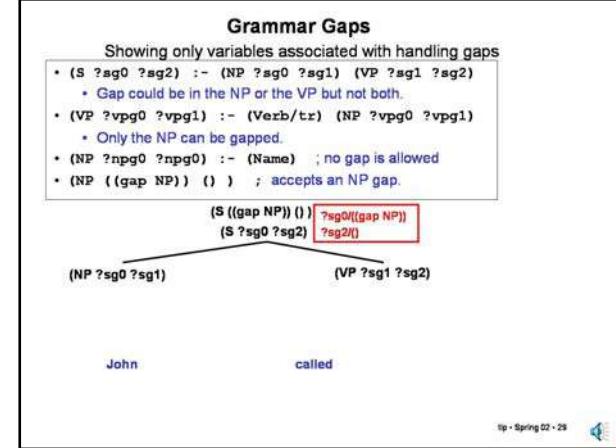
Slide 12.2.29

This goal would match the consequent of the first rule and would match the gap variables of the S literal in the process. Note that the ?sg1 variable is unbound. It is binding this variable that will determine whether the gap is in the NP or in the VP. Binding ?sg1 to ((gap NP)) would mean that the gap is in the VP, since then the gap variables in the subject NP would be equal, meaning no gap there. If ?sg1 is bound to () then the gap would be in the subject NP.



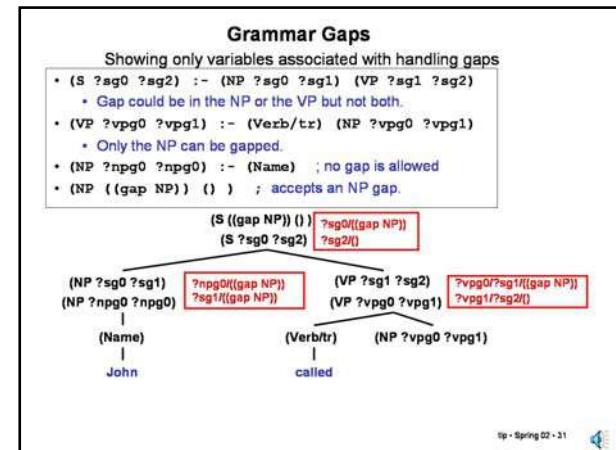
Slide 12.2.30

Using the first NP rule, we can successfully parse "John". In the process,  $?sg1$  would have to be bound to the same value as  $?sg0$  for them both to unify with  $?npg0$ . At this point, we've committed for the gap to be in the verb phrase in order for the whole parse to be successful.



### Slide 12.2.31

Now we use the VP rule. Note that at this point, the gap variables are already bound from our previous unifications.



### Grammar Gaps

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - Gap could be in the NP or the VP but not both.
- (VP ?vpg0 ?vpg1) :- (Verb/tr) (NP ?vpg0 ?vpg1)
  - Only the NP can be gapped.
- (NP ?npg0 ?npg0) :- (Name) ; no gap is allowed
- (NP ((gap NP)) ()) ; accepts an NP gap.

tip · Spring 02 · 32

**Slide 12.2.32**

Now we use the NP rule that accepts the gapped (that is, missing) NP. This rule is acceptable since the bindings of the gap variables are consistent with the rule.

So, we have successfully parsed a sentence that has the object NP missing, as required.

### Grammar Gaps

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - Gap could be in the NP or the VP but not both.
- (VP ?vpg0 ?vpg1) :- (Verb/tr) (NP ?vpg0 ?vpg1)
  - Only the NP can be gapped.
- (NP ?npg0 ?npg0) :- (Name) ; no gap is allowed
- (NP ((gap NP)) ()) ; accepts an NP gap.

(S ((gap NP))())

called      John

tip · Spring 02 · 33

### Grammar Gaps

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - Gap could be in the NP or the VP but not both.
- (VP ?vpg0 ?vpg1) :- (Verb/tr) (NP ?vpg0 ?vpg1)
  - Only the NP can be gapped.
- (NP ?npg0 ?npg0) :- (Name) ; no gap is allowed
- (NP ((gap NP)) ()) ; accepts an NP gap.

tip · Spring 02 · 34

**Slide 12.2.34**

As before, we use the top-level sentence rule, which binds the gap variables, ?sg0 and ?sg2, leaving ?sg1 unbound.

### Grammar Gaps

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - Gap could be in the NP or the VP but not both.
- (VP ?vpg0 ?vpg1) :- (Verb/tr) (NP ?vpg0 ?vpg1)
  - Only the NP can be gapped.
- (NP ?npg0 ?npg0) :- (Name) ; no gap is allowed
- (NP ((gap NP)) ()) ; accepts an NP gap.

called      John

tip · Spring 02 · 35

**Slide 12.2.35**

Now, we need to parse the subject NP. The parse would try the first NP rule, that would require finding a name in the sentence, but that would fail. Then, we would try the gapped NP rule, which succeeds and binds ?sg1 to (). At this point, we've committed to the gap being in the subject NP and not the VP.

### Grammar Gaps

Showing only variables associated with handling gaps

- $(S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)$ 
  - Gap could be in the NP or the VP but not both.
- $(VP ?vpg0 ?vpg1) :- (Verb/tz) (NP ?vpg0 ?vpg1)$ 
  - Only the NP can be gapped.
- $(NP ?npg0 ?npg0) :- (Name)$ ; no gap is allowed
- $(NP ((gap NP)) ()) ;$  accepts an NP gap.

tip - Spring 02 - 36

**Slide 12.2.36**

The VP rule would now be used and then the Verb rule would accept "called".

### Grammar Gaps

Showing only variables associated with handling gaps

- $(S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)$ 
  - Gap could be in the NP or the VP but not both.
- $(VP ?vpg0 ?vpg1) :- (Verb/tz) (NP ?vpg0 ?vpg1)$ 
  - Only the NP can be gapped.
- $(NP ?npg0 ?npg0) :- (Name)$ ; no gap is allowed
- $(NP ((gap NP)) ()) ;$  accepts an NP gap.

tip - Spring 02 - 37

**Slide 12.2.37**

The NP rule would then be used. Note that the gap variables are already both bound to (), so there is no problem there. The Name rule would then recognize John.

So, we see that using the same set of rules that we would use to parse a normal sentence, we can parse sentences missing an NP, either in the subject or object position. In general, we can arrange to handle gaps for any type of phrase.

### Relative Clauses

Showing only variables associated with handling gaps

- $(S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)$ 
  - An NP with gap [diff(?sg0,?sg1)] and a VP with gap [diff(?sg1,?sg2)] define a sentence with gap [diff(?sg0,?sg2)] – this is append!

tip - Spring 02 - 38

**Slide 12.2.38**

So, let's look at a more complete grammar for relative clauses using gaps. The Sentence rule simply indicates that the gap list is distributed in some way among the NP and VP constituents.

### Relative Clauses

Showing only variables associated with handling gaps

- $(S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)$ 
  - An NP with gap [diff(?sg0,?sg1)] and a VP with gap [diff(?sg1,?sg2)] define a sentence with gap [diff(?sg0,?sg2)] – this is append!
- $(VP ?vpg0 ?vpg1) :- (Verb/tz) (NP ?vpg0 ?vpg1)$ 
  - In this VP definition, only the NP can be gapped.

tip - Spring 02 - 39

**Slide 12.2.39**

The transitive VP rule indicates that only the NP can be gapped, we can't drop the Verb.

### Relative Clauses

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - An NP with gap [diff(?sg0,?sg1)] and a VP with gap [diff(?sg1,?sg2)] define a sentence with gap [diff(?sg0,?sg2)] – this is append!
- (VP ?vpg0 ?vpg1) :- (Verb/tr) (NP ?vpg0 ?vpg1)
  - In this VP definition, only the NP can be gapped.
- (VP ?vpg0 ?vpg0) :- (Verb/itr)
- (NP ?npg0 ?npg0) :- (Name)
- (NP ?npg0 ?npg0) :- (Det) (Noun) (RelClause)
  - The previous 3 rules do not involve any gapped constituents, since the same variable is used for both gap list variables, the difference is empty.

tip · Spring 02 · 40

### Slide 12.2.40

The next three rules say that there can be no gapped constituents since the first and second gap features are constrained to be the same, since the same variable name is used for both.

### Slide 12.2.41

Now, this is an important rule. This says that if we use a gapped NP in this constituent we don't need any additional input in order to succeed in parsing an NP. This rule "fills the gap".

### Relative Clauses

Showing only variables associated with handling gaps

- (S ?sg0 ?sg2) :- (NP ?sg0 ?sg1) (VP ?sg1 ?sg2)
  - An NP with gap [diff(?sg0,?sg1)] and a VP with gap [diff(?sg1,?sg2)] define a sentence with gap [diff(?sg0,?sg2)] – this is append!
- (VP ?vpg0 ?vpg1) :- (Verb/tr) (NP ?vpg0 ?vpg1)
  - In this VP definition, only the NP can be gapped.
- (VP ?vpg0 ?vpg0) :- (Verb/itr)
- (NP ?npg0 ?npg0) :- (Name)
- (NP ?npg0 ?npg0) :- (Det) (Noun) (RelClause)
  - The previous 3 rules do not involve any gapped constituents, since the same variable is used for both gap list variables, the difference is empty.
- (NP ((gap NP)) ())
  - If we have an NP subgoal and a gapped NP is available, use it.
- (RelClause)
- (RelClause) :- "that" (S ((gap NP)) ())
  - A relative clause is empty or it is of the form "that" followed by a Sentence with an NP gap.

tip · Spring 02 · 42

### Slide 12.2.42

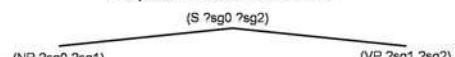
Finally, we get the the definition of the relative clause. The first rule just says that the RelClause is optional. The second rule is the key one, it says that a RelClause is composed of the word "that" followed by a sentence with a missing NP and it provides the NP to be used to fill the gap as necessary while parsing S.

### Slide 12.2.43

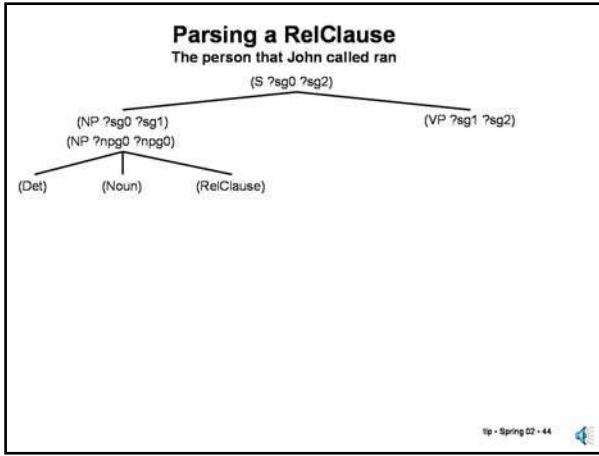
Let's see how we can parse the sentence "The person that John called ran". We start with the S rule.

### Parsing a RelClause

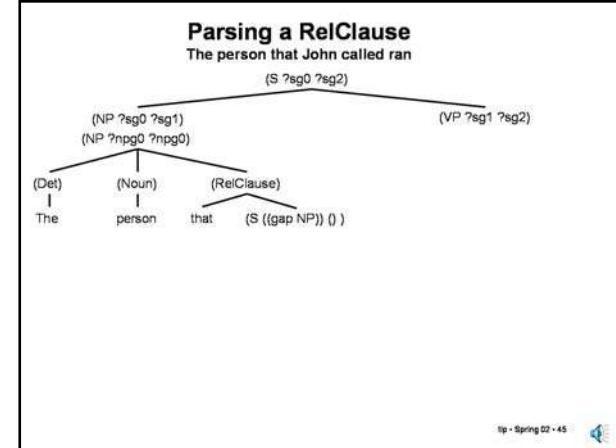
The person that John called ran



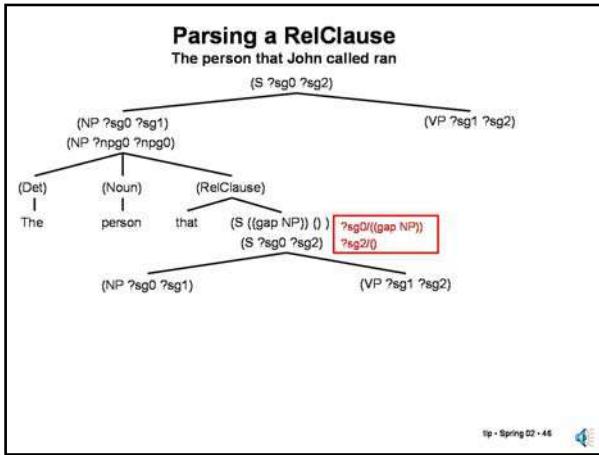
tip · Spring 02 · 43

**Slide 12.2.44**

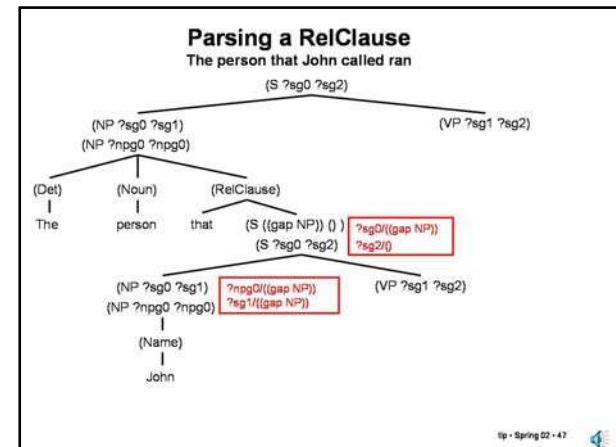
We then use the NP rule which generates three subgoals.

**Slide 12.2.45**

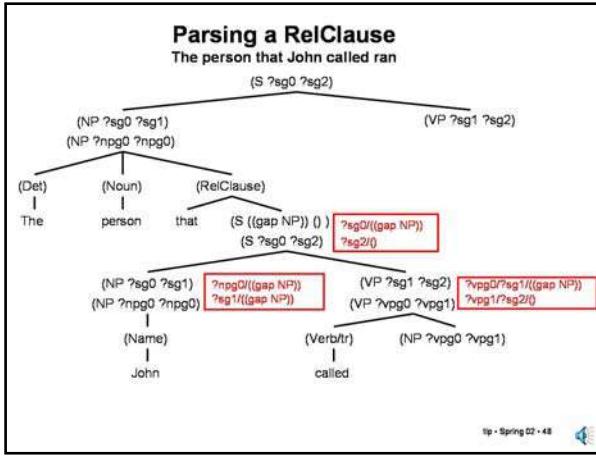
We proceed to parse the Determiner and the noun, the RelClause then sets the subgoal of parsing a sentence with a missing NP.

**Slide 12.2.46**

We use our S rule again, but now we have ?sg0 bound to ((gap NP)) and ?sg2 is bound to the empty list. We now proceed with the subject NP for this embedded sentence.

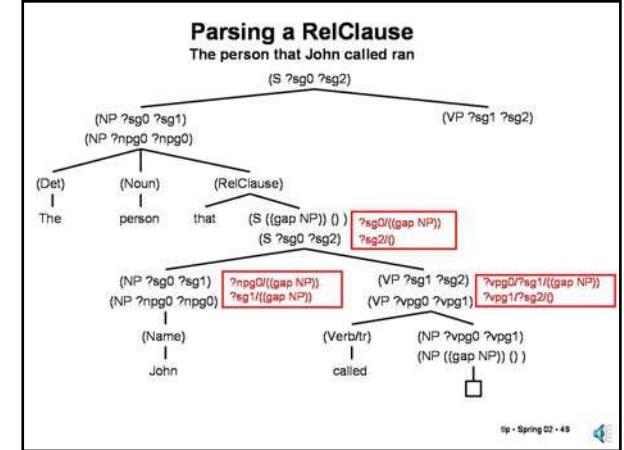
**Slide 12.2.47**

Note that we end up using the Name rule in which the gap features are constrained to be equal. So that means that the gap NP is not used here. As a result of this match, we have that ?sg1 is now equal to ((gap NP)).

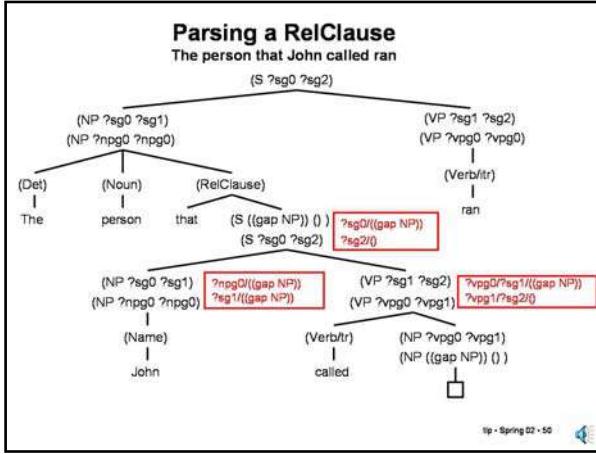
**Slide 12.2.48**

Now, we proceed to parse the VP of the embedded sentence, noting that ?vpg0 is ((gap NP)) and the ?vpg1 is the empty list. This means that we expect to use the gap in parsing the VP.

We proceed to parse the Verb - "called".

**Slide 12.2.49**

Now, we need an NP but we want to use a gap NP, so we succeed with no input.

**Slide 12.2.50**

We finish up by parsing the VP of the top-level sentence using the remaining word, the verb "ran".

Which is kind of cool...

## 6.034 Notes: Section 12.3

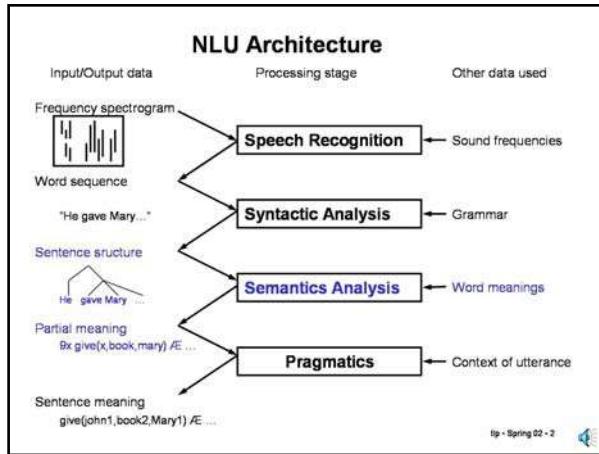
### Slide 12.3.1

Now, we move to consider the semantics phase of processing natural language.

### 6.034 Artificial Intelligence

- Natural Language Understanding
  - Getting at the meaning of text and speech
  - Not just pattern matching
- Overview
- Syntax
- Semantics

lip - Spring 02 - 1



### Slide 12.3.2

Recall that our goal is to take in the parse trees produced by syntactic analysis and produce a meaning representation.

### Slide 12.3.3

We want semantics to produce a representation that is somewhat independent of syntax. So, for example, we would like the equivalent active and passive voice versions of a sentence to produce equivalent semantics representations.

We will assume that the meaning representation is some variant of first order predicate logic. We will specify what type of variant later.

We have limited the scope of the role of semantics by ruling out context. So, for example, given the sentence "He gave her the book", we will be happy with indicating that some male gave the book to some female, without identifying who these people might be.

### Semantics

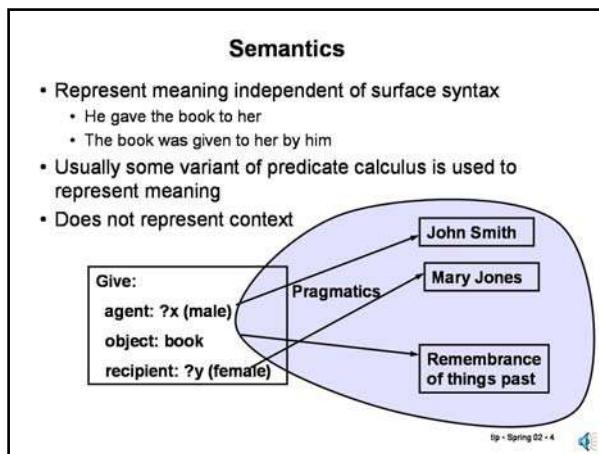
- Represent meaning independent of surface syntax
  - He gave the book to her
  - The book was given to her by him
- Usually some variant of predicate calculus is used to represent meaning
- Does not represent context

```

Give:
agent: ?x (male)
object: book
recipient: ?y (female)

```

lip - Spring 02 - 3

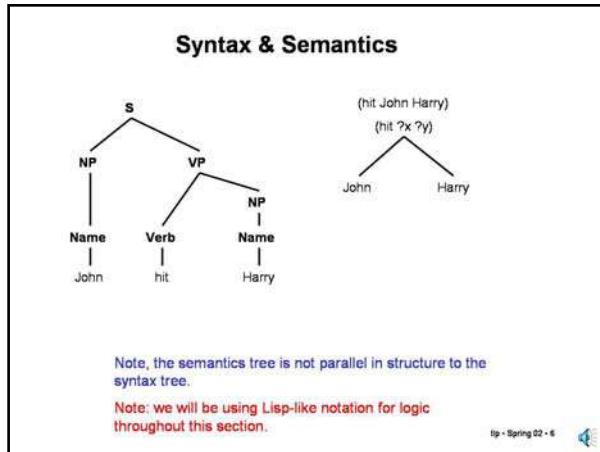
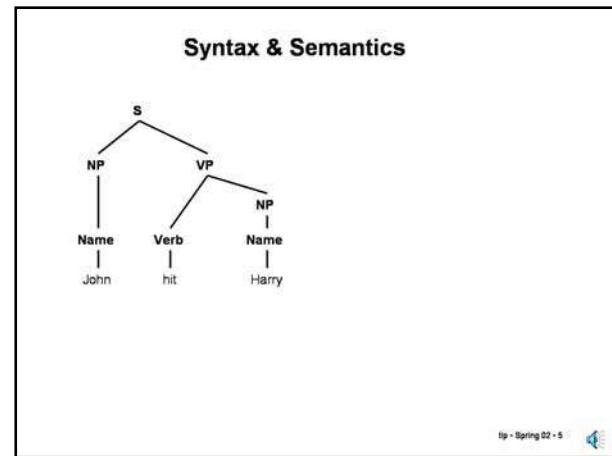


### Slide 12.3.4

Part of the role of pragmatics, the next phase of processing, is to try to make those connections.

**Slide 12.3.5**

So, let's consider a very simple sentence "John hit Harry". We have here the simple parse tree. What should we expect the semantic representation to be?

**Slide 12.3.7**

Our guiding principle will be that the semantics of a constituent can be constructed by composing the semantics of its constituents. However, the composition will be a bit subtle and we will be using feature values to carry it out.

Let's look at the sentence rule. We will be exploiting the "two way" matching properties of unification strongly here. This rule says that the meaning of the sentence is picked up from the meaning of the VP, since the second argument of the VP is the same as the semantics of the sentence as a whole. We already saw this in our simple example, so it comes as no surprise. Note also that the semantics of the subject NP is passed as the first argument of the VP (by using the same variable name).

**Slide 12.3.6**

In this simple case, we might want something like this, where hit is a predicate and John and Harry are constant terms in the logical language. The key thing to notice is that even for this simple sentence the semantic structure produced is not perfectly parallel to the syntactic structure.

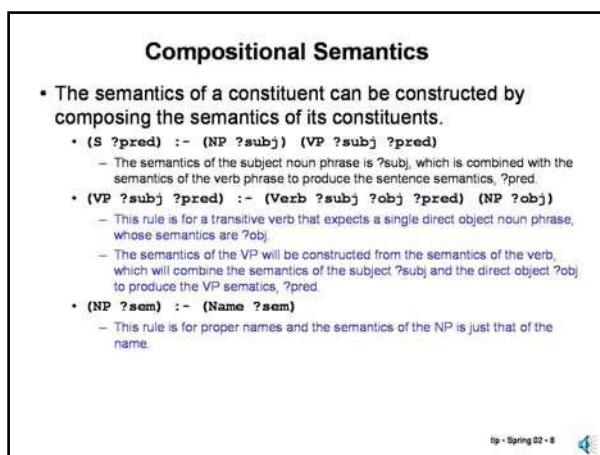
In this interpretation, the meaning of the verb is the center of the semantics. The meaning representation of the subject NP is embedded in the meaning representation of the verb phrase. This suggests that producing the semantics will not be a trivial variant of the parse tree. So, let's see how we can achieve this.

**Compositional Semantics**

- The semantics of a constituent can be constructed by composing the semantics of its constituents.

- (S ?pred) :- (NP ?subj) (VP ?subj ?pred)
  - The semantics of the subject noun phrase is ?subj, which is combined with the semantics of the verb phrase to produce the sentence semantics, ?pred.
- (VP ?subj ?pred) :- (Verb ?subj ?obj ?pred) (NP ?obj)
  - (NP ?sem) :- (Name ?sem)

lip - Spring 02 - 7

**Slide 12.3.8**

The VP has two arguments, the semantics of the subject NP (which will be an input) and the resulting semantics of the VP. In the VP rule, we see that the result semantics is coming from the Verb, which is combining the semantics of the subject and the object NPs to produce the result for the VP (and ultimately the sentence).

**Slide 12.3.9**

Let's look at the rule for a particular Verb. Note that the first two arguments are simply variables which are then included in the expression for the verb semantics, the predicate hit with two arguments (the subject and the object).

**Compositional Semantics**

- The semantics of a constituent can be constructed by composing the semantics of its constituents.
  - (S ?pred) :- (NP ?subj) (VP ?subj ?pred)
  - (VP ?subj ?pred) :- (Verb ?subj ?obj ?pred) (NP ?obj)
  - (NP ?sem) :- (Name ?sem)
- The semantics of individual words are given in the lexicon.
  - (Verb ?x ?y (hit ?x ?y)) :- hit
    - The verb semantics for hit. Note that the subject will match ?x and the direct object will match ?y and the final semantics will be (hit ?x ?y)
  - (Name John) :- John
  - (Name Harry) :- Harry
    - Trivial semantics

tip · Spring 02 · 8

**Compositional Semantics**

- The semantics of a constituent can be constructed by composing the semantics of its constituents.
  - (S ?pred) :- (NP ?subj) (VP ?subj ?pred)
  - (VP ?subj ?pred) :- (Verb ?subj ?obj ?pred) (NP ?obj)
  - (NP ?sem) :- (Name ?sem)
- The semantics of individual words are given in the lexicon.
  - (Verb ?x ?y (hit ?x ?y)) :- hit
  - (Name John) :- John
  - (Name Harry) :- Harry
- The sentence: "John hit Harry"
  - (backchain '(S ?sem 0 3))
  - ?sem = (hit John Harry)

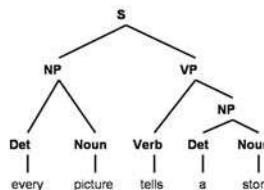
tip · Spring 02 · 10

**Slide 12.3.10**

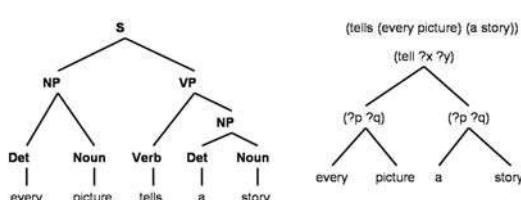
We can pull this altogether by simply calling backchain with the goal pattern for a successful parse. We will want to retrieve the value of the binding for ?sem, which is the semantics for the sentence.

**Slide 12.3.11**

Let's look at a somewhat more complex example - "Every picture tells a story". Here is the syntactic analysis.

**Syntax & Semantics**

tip · Spring 02 · 11

**Syntax & Semantics**

Note, the semantics tree is not parallel in structure to the syntax tree.

tip · Spring 02 · 12

**Slide 12.3.12**

This is one possible semantic analysis. Note that it follows the pattern of our earlier example. The top-level predicate is derived from the verb and it includes as arguments the semantics of the subject and direct object.

**Slide 12.3.13**

The only innovation in this grammar, besides the new words is a simple semantics for a noun phrase formed from a Determiner and a Noun - just placing them in a list. We can interpret the result as a quantifier operating on a predicate. But, what does this mean? It's certainly not legal logic notation.

**Another Example**

## • The grammar

- (S ?pred) :- (NP ?subj) (VP ?subj ?pred)
- (VP ?subj ?pred) :- (Verb ?subj ?obj ?pred) (NP ?obj)
- (NP ?sem) :- (Name ?sem)
- (NP (?detsem ?nsem)) :- (Det ?detsem) (Noun ?nsem)
- (Verb ?x ?y (tells ?x ?y)) :- tells
- (Noun picture) :- picture
- (Noun story) :- story
- (Det every) :- every
- (Det a) :- a

## • The sentence: "Every picture tells a story"

- (backchain '(S ?sem 0 5))
- ?sem = (tell (every picture) (a story))

tip · Spring 02 · 13

**Quantifiers**

- (tell (every picture) (a story)) is ambiguous:
  - $\forall x \text{ Picture}(x) \rightarrow \exists y \text{ Story}(y) \wedge \text{Tell}(x,y)$
  - $\exists y \text{ Story}(y) \wedge \forall x \text{ Picture}(x) \rightarrow \text{Tell}(x,y)$
- The first of these is the usual interpretation, but consider:
  - Every US citizen has a president

tip · Spring 02 · 14

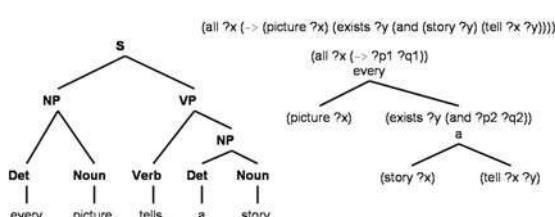
**Slide 12.3.15**

Let's pick one of the interpretations and see how we could generate it. At the heart of this attempt is a definition of the meaning of the determiners "every" and "a", which now become patterns for universally and existentially quantified statements. Note also that the nouns become patterns for predicate expressions.

**Quantifiers**

- (tell (every picture) (a story)) is ambiguous:
  - $\forall x \text{ Picture}(x) \rightarrow \exists y \text{ Story}(y) \wedge \text{Tell}(x,y)$
  - $\exists y \text{ Story}(y) \wedge \forall x \text{ Picture}(x) \rightarrow \text{Tell}(x,y)$
- The first of these is the usual interpretation, but consider:
  - Every US citizen has a president
- Let's consider how we could generate:
  - $\forall x \text{ Picture}(x) \rightarrow \exists y \text{ Story}(y) \wedge \text{Tell}(x,y)$
  - ( $\forall x (\rightarrow (\text{picture } ?x) (\exists y (\text{and} (\text{story } ?y) (\text{tell } ?x ?y))))$ )
    - every = ( $\forall x (\rightarrow ?p1 ?q1)$ )
    - picture = ( $\text{picture } ?x$ )
    - tells = ( $\text{tell } ?x ?y$ )
    - a = ( $\exists y (\text{and} ?p2 ?q2)$ )
    - story = ( $\text{story } ?x$ )

tip · Spring 02 · 15

**Syntax & Semantics**

Note, the semantics tree is not parallel in structure to the syntax tree.

tip · Spring 02 · 16

**Slide 12.3.16**

Our target semantic representation is shown here. Note that by requiring the semantics to be a legal logical sentence, we've had to switch the key role from the verb to the determiner. That is, the top node in the sentence semantics comes from the determiner, not the verb. The semantics of the verb is fairly deeply nested in the final semantics - but it still needs to combine the semantics of the subject and direct object NPs. Note, however, that it is incorporating them by using the quantified variable introduced by the determiners of the subject and object NPs.

**Slide 12.3.17**

Let's start with the definitions of the words. Here's the definition for the verb "tells". We have seen this before. It combines the semantics of the subject NP (bound to ?x) and the semantics of the object NP (bound to ?y) with the predicate representing the verb to produce the VP semantics.

**Quantifiers**

- (Verb ?x ?y (tell ?x ?y)) :- tells
  - ?x denotes the subject and ?y the direct object, the resulting semantics is (tell ?x ?y).

tip · Spring 02 · 17

**Quantifiers**

- (Verb ?x ?y (tell ?x ?y)) :- tells
  - ?x denotes the subject and ?y the direct object, the resulting semantics is (tell ?x ?y).
- (Noun ?x (picture ?x)) :- picture
- (Noun ?x (story ?x)) :- story
  - ?x will typically be a variable, which we restrict to denote a picture or a story or (and (young ?x) (male ?x)) for boys, etc.

tip · Spring 02 · 18

**Slide 12.3.18**

The nouns will be denoted by one of the quantified variables introduced by the quantifiers. The noun places a restriction on the entities that the variable can refer to. In this definition, the quantified variable will be bound to ?x and incorporated into the predicate representing the noun.

**Quantifiers**

- (Verb ?x ?y (tell ?x ?y)) :- tells
  - ?x denotes the subject and ?y the direct object, the resulting semantics is (tell ?x ?y).
- (Noun ?x (picture ?x)) :- picture
- (Noun ?x (story ?x)) :- story
  - ?x will typically be a variable, which we restrict to denote a picture or a story or (and (young ?x) (male ?x)) for boys, etc.
- (Det ?x ?p ?q (all ?x (-> ?p ?q))) :- every
- (Det ?x ?p ?q (exists ?x (and ?p ?q))) :- a
  - The ?x is the formal variable, ?p denotes the semantics of the noun and ?q the semantics of the predicate. For a subject NP, the predicate comes from the VP of the sentence. For an object NP, the predicate comes from the verb.

tip · Spring 02 · 19

**Quantifiers**

- (S ?sent) :- (NP ?x ?vp ?sent) (VP ?x ?vp)
  - The semantics of the sentence will be derived from the NP, since the determiner provides the quantifier, which is the top node in the semantics.
  - ?x will be the "formal variable" for the quantifier, e.g. (all ?x ...)
- (VP ?xs ?vp) :- (Verb ?xs ?xo ?verb) (NP ?xo ?verb ?vp)
- (NP ?x ?p ?np) :- (Det ?x ?noun ?p ?np) (Noun ?x ?noun)
- (Verb ?x ?y (tell ?x ?y)) :- tells
- (Noun ?x (picture ?x)) :- picture
- (Noun ?x (story ?x)) :- story
- (Det ?x ?p ?q (all ?x (-> ?p ?q))) :- every
- (Det ?x ?p ?q (exists ?x (and ?p ?q))) :- a

tip · Spring 02 · 20

**Slide 12.3.20**

The new sentence (S) rule reflects the difference in where the top-level semantics is being assembled. Before, we passed the semantics of the subject NP into the VP, now we go the other way. The semantics of the VP is an argument to the subject NP.

Note that the variable ?x here will not be bound to anything, it is the variable that will be used as the quantified variable by the determiner's semantics.

**Slide 12.3.21**

The VP rule is analogous. The semantics of the Verb will combine a reference to the subject and object semantics (through their corresponding quantified variables) and the resulting semantics of the Verb will be combined into the semantics of the NP (which will ultimately be derived from the semantics of the determiner).

**Quantifiers**

- (S ?sent) :- (NP ?x ?vp ?sent) (VP ?x ?vp)
  - The semantics of the sentence will be derived from the NP, since the determiner provides the quantifier, which is the top node in the semantics.
  - ?x will be the "formal variable" for the quantifier, e.g. (all ?x ...)
- (VP ?xs ?vp) :- (Verb ?xs ?xo ?verb) (NP ?xo ?verb ?vp)
  - Similarly, the semantics of the VP (?vp) will be derived from that of the direct object NP, e.g. (exists ?y (and (story ?y) (tell ?x ?y)))
  - Note that ?xs will be formal variable from subject NP and ?xo will be the variable from the object NP, they will be combined to form the Verb semantics (tell ?xs ?xo).
- (NP ?x ?p ?np) :- (Det ?x ?noun ?p ?np) (Noun ?x ?noun)
  - The semantics of the NP is produced by the determiner, which incorporates the semantics of the noun and that of the predicate.
- (Verb ?y (tell ?x ?y)) :- tells
- (Noun ?x (picture ?x)) :- picture
- (Noun ?x (story ?x)) :- story
- (Det ?x ?p ?q (all ?x (-> ?p ?q))) :- every
- (Det ?x ?p ?q (exists ?x (and ?p ?q))) :- a

tip · Spring 02 · 21

**Slide 12.3.22**

The NP rule in fact takes ?p, which will be the semantics of the Verb phrase and combine them with the semantics of the noun in the semantics of the Determiner.

**Quantifiers**

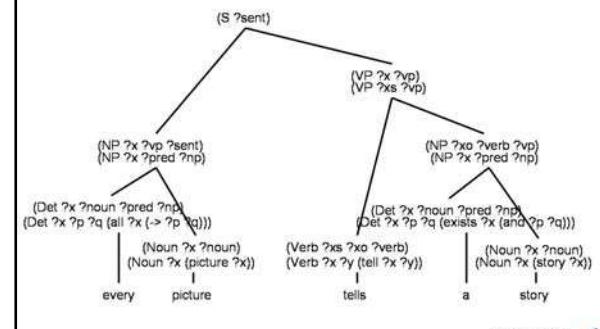
- (S ?sent) :- (NP ?x ?vp ?sent) (VP ?x ?vp)
  - The semantics of the sentence will be derived from the NP, since the determiner provides the quantifier, which is the top node in the semantics.
  - ?x will be the "formal variable" for the quantifier, e.g. (all ?x ...)
- (VP ?xs ?vp) :- (Verb ?xs ?xo ?verb) (NP ?xo ?verb ?vp)
  - Similarly, the semantics of the VP (?vp) will be derived from that of the direct object NP, e.g. (exists ?y (and (story ?y) (tell ?x ?y)))
  - Note that ?xs will be formal variable from subject NP and ?xo will be the variable from the object NP, they will be combined to form the Verb semantics (tell ?xs ?xo).
- (NP ?x ?p ?np) :- (Det ?x ?noun ?p ?np) (Noun ?x ?noun)
  - The semantics of the NP is produced by the determiner, which incorporates the semantics of the noun and that of the predicate.
- (Verb ?y (tell ?x ?y)) :- tells
- (Noun ?x (picture ?x)) :- picture
- (Noun ?x (story ?x)) :- story
- (Det ?x ?p ?q (all ?x (-> ?p ?q))) :- every
- (Det ?x ?p ?q (exists ?x (and ?p ?q))) :- a

tip · Spring 02 · 22

**Slide 12.3.23**

Here we see how the parse works out. You have to follow the bindings carefully to see how it all works out.

What is remarkable about this is that we were able to map from a set of words to a first-order logic representation (which does not appear to be very similar) with a relatively compact grammar and with quite generic mechanisms.

**Parsing with Quantifiers**

tip · Spring 02 · 23

**Quasi-Logical Form**

- Semantics tries to capture sentence meaning independent of context. Producing the correct representation in First Order Logic usually requires context to resolve the ambiguity in language:
  - Syntactic ambiguity: "Mary saw John on the hill with a telescope"
  - Lexical ambiguity: "We went to the bank" {of the river? Fleet Bank?}
  - Quantifier scope ambiguity: "Every man loves a woman"
  - Referential ambiguity: "He gave her the book", "Stop that!"

tip · Spring 02 · 24

**Slide 12.3.24**

The quantified expression we produced in the previous example is unambiguous, as required to be able to write an expression in first order logic. However, natural language is far from unambiguous. We have seen examples of syntactic ambiguity, lexical and attachment ambiguity in particular, plus there are many examples of semantic ambiguity, for example, ambiguity in quantifier scope and ambiguity on who or what pronouns refer to are examples.

**Slide 12.3.25**

One common approach to semantics is to have it produce a representation that is not quite the usual logical notation, sometimes called **quasi-logical form**, that preserves some of the ambiguity in the input, leaving it to the pragmatics phase to resolve the ambiguities employing contextual information.

**Quasi-Logical Form**

- Semantics tries to capture sentence meaning independent of context. Producing the correct representation in First Order Logic usually requires context to resolve the ambiguity in language:
  - Syntactic ambiguity: "Mary saw John on the hill with a telescope"
  - Lexical ambiguity: "We went to the bank" {of the river? Fleet Bank?}
  - Quantifier scope ambiguity: "Every man loves a woman"
  - Referential ambiguity: "He gave her the book", "Stop that!"
- Instead of producing FOL, produce **quasi-logical form** that preserves some of the ambiguity. Leave it for next phase to resolve the ambiguity.
  - (tell (every ?x (picture ?x)) (exists ?x (story ?x)))

tip · Spring 02 · 25

**Quasi-Logical Form**

- Allow the use of **quantified terms** such as
  - (every ?x (picture ?x))
  - (exists ?x (story ?x))

tip · Spring 02 · 26

**Slide 12.3.26**

One common aspect of quasi-logical notation is the use of **quantified terms**. These terms indicate the nature of the intended quantification but do not specify the scope of the quantifier in the sentence and thus preserves the ambiguity in the natural language. Note that we are treating these quantified expressions as terms, and using them as arguments to functions and predicates - which is not legal FOL.

**Slide 12.3.27**

In quasi-logical notation, one also typically extends the range of available quantifiers to correspond more closely to the range of determiners available in natural language. One important case, is the determiner "the", which indicates a unique descriptor.

**Quasi-Logical Form**

- Allow the use of **quantified terms** such as
  - (every ?x (picture ?x))
  - (exists ?x (story ?x))
- Allow a more general class of quantifiers
  - (the ?x (and (big ?x) (picture ?x) (author ?x "Sargent")))
  - (most ?x (child ?x))
  - (name ?x John)
  - (pronoun ?x he)

tip · Spring 02 · 27

**Quasi-Logical Form**

- Allow the use of **quantified terms** such as
  - (every ?x (picture ?x))
  - (exists ?x (story ?x))
- Allow a more general class of quantifiers
  - (the ?x (and (big ?x) (picture ?x) (author ?x "Sargent")))
  - (most ?x (child ?x))
  - (name ?x John)
  - (pronoun ?x he)
- These will have to be converted to FOL and given an appropriate axiomatization.

tip · Spring 02 · 28

**Slide 12.3.28**

These quantified terms and generalized quantifiers will require conversion to standard FOL together with a careful axiomatization of their intended meaning before the resulting semantics can be used for inference.

**Slide 12.3.29**

Let's illustrate how the type of language processing we have been discussing here could be used to build an extremely simple database system. We'll assume that we want to deal with a simple genealogy domain. We will have facts in our database describing the family relationships between some set of people. We will not restrict ourselves to just the minimal set of facts, such as parent, male and female, we will also keep derived relationships such as grandparent and cousin.

**A very simple language system**

The Database

## • Genealogy database

- (parent x y), (male x), (female x)
- (grandparent x y), (aunt/uncle x y),  
(sibling x y), (cousin x y)

▷ *parent* is a relation with two arguments▷ *parent* is a relation with three arguments, the third being the child▷ *parent* is a relation with four arguments, the fourth being the sibling▷ *parent* is a relation with five arguments▷ *parent* is a relation with six arguments▷ *parent* is a relation with seven arguments▷ *parent* is a relation with eight arguments

tip • Spring 02 • 28

**A very simple language system**

The Database

## • Genealogy database

- (parent x y), (male x), (female x)
- (grandparent x y), (aunt/uncle x y),  
(sibling x y), (cousin x y)

## • Assume relations explicit in database.

## • Use forward-chaining of rules to expand relations when new facts added.

▷ *parent* is a relation with two arguments▷ *parent* is a relation with three arguments▷ *parent* is a relation with four arguments▷ *parent* is a relation with five arguments▷ *parent* is a relation with six arguments

tip • Spring 02 • 30

**Slide 12.3.30**

In fact, we will assume that all the relationships between people we know about are explicitly in the database. We can accomplish them by running a set of Prolog-like rules in forward chaining fashion whenever a new fact is added. We do this, rather than do deduction at retrieval time because of issues of equality, which we will discuss momentarily.

**A very simple language system**

The Database

## • Genealogy database

- (parent x y), (male x), (female x)
- (grandparent x y), (aunt/uncle x y),  
(sibling x y), (cousin x y)

▷ *parent* is a relation with two arguments▷ *parent* is a relation with three arguments▷ *parent* is a relation with four arguments▷ *parent* is a relation with five arguments▷ *parent* is a relation with six arguments▷ *parent* is a relation with seven arguments

tip • Spring 02 • 31

**A very simple language system**

The Database

## • Genealogy database

- (parent x y), (male x), (female x)
- (grandparent x y), (aunt/uncle x y),  
(sibling x y), (cousin x y)

## • Assume relations explicit in database.

## • Use forward-chaining of rules to expand relations when new facts added.

• (*is* x y) indicates two symbols denote same person

## • Retrieval query examples:

- (and (female ?x) (parent ?x John))
- (and (male ?x) (cousin Mary ?x))
- (grandparent Harry ?x)

tip • Spring 02 • 32

**Slide 12.3.32**

We can now do very simple retrieval from this database of facts using our backchaining algorithm. We initialize the goal stack in backchaining with the query. If the query is a conjunction, we initialize the stack with all the conjuncts.

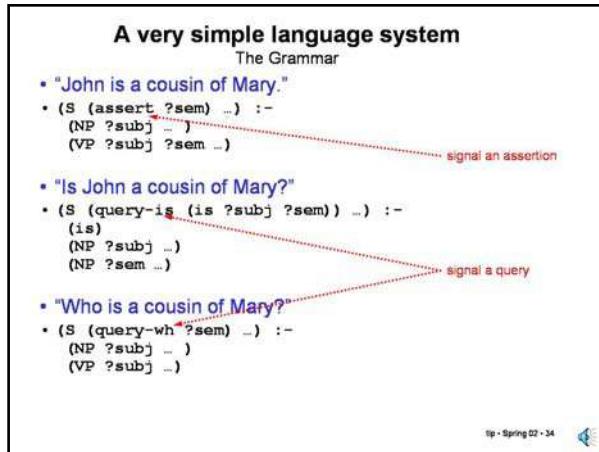
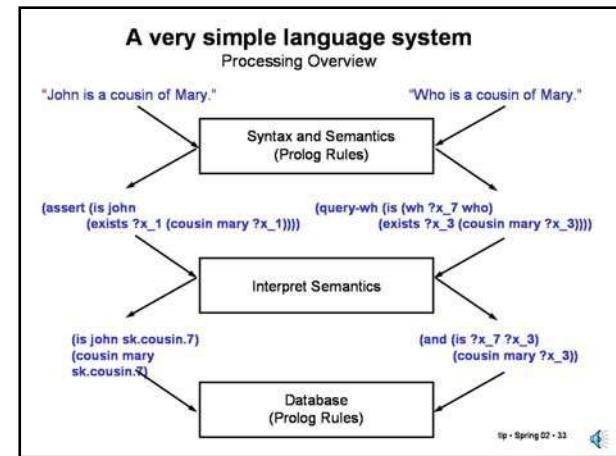
**Slide 12.3.33**

Here we see a brief overview of the processing that we will do to interact with the genealogy database.

We will be able to accept declarative sentences, such as "John is a cousin of Mary". These sentences will be processed by a grammar to obtain a semantic representation. This representation will then be interpreted as a set of facts to be added to the database.

We can also ask questions, such as "Who is a cousin of Mary". Our grammar will produce a semantic representation. The semantics of this type of sentence is converted into a database query and passed to the database.

Let's look in more detail at the steps of this process.

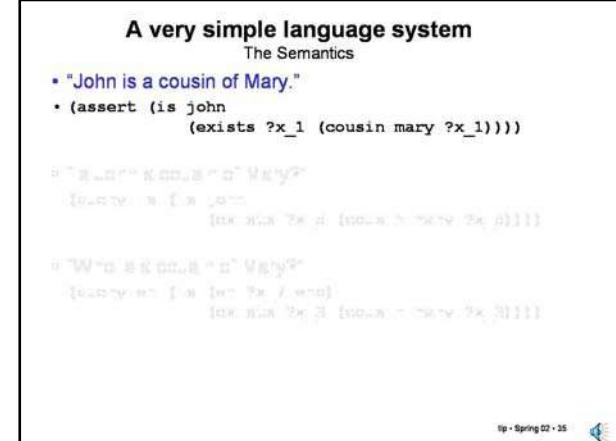
**Slide 12.3.34**

We will need a grammar built along the lines we have been discussing. One of the things the grammar does is classify the sentences into declarative sentences, such as "John is a cousin of Mary", which will cause us to assert a fact in our database, and questions, such as, "Is John a cousin of Mary" or "Who is a cousin of Mary", which will cause us to query the database.

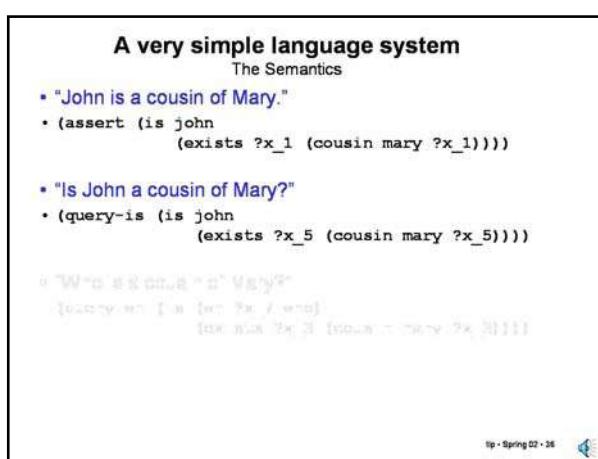
**Slide 12.3.35**

Here we see one possible semantics for the declarative sentence "John is a cousin of Mary". The operation `assert` indicates the action to be taken. The body is in quasi-logical form; the quantified term `(exists ?x_1 (cousin mary ?x_1))` is basically telling us there exists a person that is in the cousin relationship to Mary. The outermost `is` assertion is saying that John denotes that person. This is basically interpreting this quasi-logical form as:

```
| x . (is John x) ^ (cousin Mary x)
```

**Slide 12.3.36**

The semantics of the question "Is John a cousin of Mary?" is essentially identical to that of the declarative form, but it is prefixed by a query operation rather than an assertion. So, we would want to use this to query the database rather than for asserting new facts.



**Slide 12.3.37**

We can also have a question such as "Who is a cousin of Mary", which is similar except that John is replaced by a term indicating that we are interested in determining the value of this term.

**A very simple language system**

Using the Semantics

- "John is a cousin of Mary."
- (assert (is john
   
          (exists ?x\_1 (cousin mary ?x\_1))))
- "Is John a cousin of Mary?"
- (query-is (is john
   
          (exists ?x\_5 (cousin mary ?x\_5))))
- "Who is a cousin of Mary?"
- (query-wh (is (wh ?x\_7 who)
   
          (exists ?x\_3 (cousin mary ?x\_3))))

tip · Spring 02 · 37

**A very simple language system**

Using the Semantics

- "John is a cousin of Mary."
- (assert (is john
   
          (exists ?x\_1 (cousin mary ?x\_1))))
- Assign skolem constant for ?x\_1, e.g. sk.cousin.7
- Convert body of exists into one or more facts
- Replace (exists ?x ...) with skolem constant
- (is john sk.cousin.7)
   
      (cousin mary sk.cousin.7))

tip · Spring 02 · 38

**Slide 12.3.38**

Given the semantics, we have to actually decide how to add new facts and do the retrieval. Here we show an extremely simple approach that operates for these very simple types of queries (note that we are only using existentially quantified terms).

We are basically going to turn the assertion into a list of ground facts to add to the database. We will do this by skolemizing. Since we have only existentially quantified variables, this will eliminate all variables.

We replace the quantified terms with the corresponding skolem constant and we convert the body of the quantified term into a set of facts that describe the constant.

**A very simple language system**

Using the Semantics

- "John is a cousin of Mary."
- (assert (is john
   
          (exists ?x\_1 (cousin mary ?x\_1))))
- Assign skolem constant for ?x\_1, e.g. sk.cousin.7
- Convert body of exists into one or more facts
- Replace (exists ?x ...) with skolem constant
- Add to the database:
  - (is john sk.cousin.7)
  - (cousin mary sk.cousin.7)

tip · Spring 02 · 39

**A very simple language system**

Using the Semantics

- "Who is a cousin of Mary?"
- (query-wh (is (wh ?x\_7 who)
   
          (exists ?x\_3 (cousin mary ?x\_3))))
- Convert body of exists into one or more additional conditions for query
- Replace (exists ?x ...) with ?x
- Replace (wh ?y ...) with ?y
- Retrieve from database:
  - (and (is ?x\_7 ?x\_3) (cousin mary ?x\_3))
    - ?x\_7/John
    - ?x\_3/sk.cousin.7

tip · Spring 02 · 40

**Slide 12.3.40**

We process the question in a similar way except that instead of using skolem constants we keep the variables, since we want those to match the constants in the database. When we perform the query, ?x\_7 is bound to John as expected. In general, there may be multiple matches for the query, some may be skolem constants and some may be people names. We would want to return the specific names whenever possible.

**Slide 12.3.41**

Here are some examples that show that this approach can be used to do a little inference above and beyond what is explicitly stated. Note that the assertions do not mention cousin, uncle, sister or sibling relations, those are inferred. So, we are going beyond what an Internet search engine can do, that is, pattern match on the presence of particular words.

This example has been extremely simple but hopefully it illustrates the flavor of how such a system may be built using the tools we have been developing and what could be done with such a system.

**Some Examples**

## • Assertions

- John is the father of Tom.
- Mary is the female parent of Tom.
- Bill is the brother of John.
- Jim is the male child of Bill.
- Jane is the daughter of John.
- Mary is the mother of Jane.

## • Questions

- Is Jim the cousin of Tom? ) Yes
- Who is the uncle of Tom? ) Bill
- Is Bill the uncle of Tom? ) Yes
- Is Jane the sister of Tom? ) Yes
- Who is a child of Mary? ) Tom
- Who is a sibling of Tom? ) Jane

tip • Spring 02 • 41

**Discourse Context**

- Anaphora = "use of a word referring to or replacing earlier words"
  - Jack lost his book. He looked for it for hours. Eventually he found it in his backpack.

tip • Spring 02 • 42

**Slide 12.3.42**

At this point, we'll touch briefly on a set of phenomena that are beyond the scope of pure semantics because they start dealing with the issue of context.

One general class of language phenomena is called **anaphora**. this includes pronoun use, where a word is used to refer to other words appearing either elsewhere in the sentence or in another sentence.

**Discourse Context**

- Anaphora = "use of a word referring to or replacing earlier words"
  - Jack lost his book. He looked for it for hours. Eventually he found it in his backpack.
- Ellipsis = "omission from a sentence of words needed to complete construction of meaning"
  - You did not complete the job as well as he did.

tip • Spring 02 • 43

**Discourse Context**

- Anaphora = "use of a word referring to or replacing earlier words"
  - Jack lost his book. He looked for it for hours. Eventually he found it in his backpack.
- Ellipsis = "omission from a sentence of words needed to complete construction of meaning"
  - You did not complete the job as well as he did.
- Definite descriptions = "used to refer to uniquely identifiable entity (or entities)"
  - the tall man, the red book, the president

tip • Spring 02 • 44

**Slide 12.3.44**

Another important mechanism in language is the use of **definite descriptions**, signaled by the determiner "the". The intent is that the listener be able to identify an entity previously mentioned or expected to be known.

All of these are linguistic mechanisms for incorporating context and require that a language understanding system that is engaged in an interaction with a human keep a context and be able to identify entities and actions in context based on the clues in the sentence. This is an area of active research and some systems with competence in this area have been built.

**Slide 12.3.45**

Even beyond conversational context, understanding human language requires access to the whole range of human knowledge. Even when speaking with a child, one assumes a great deal of "common sense" knowledge that computers are, as yet, sorely lacking in. The problem of language understanding at this point merges with the general problem of knowledge representation and use.

**World Knowledge**

- John needed money. He went to the bank.
- "bank of the river Charles?" "Fleet Bank?"
- Need to know that Fleet Bank has money but the bank of the Charles does not.
- John went to the store. He bought some bread.
- Did John go to the hardware store?
- Etc.

tip · Spring 02 · 45

**Applications**

- Human computer interaction:
  - Restricted domains – flight reservations, classifying e-mails into a few classes, redirecting caller to one of a few destinations.
  - Limited syntax
  - Limited vocabulary
  - Limited context
  - Limited actions
  - It is very hard for humans to understand what the limits of the system are. Can be frustrating.

tip · Spring 02 · 46

**Slide 12.3.46**

Real applications of natural language technology for human computer interfaces require a very limited scope so that the computer can get by with limited language skills and can have enough knowledge about the domain to be useful. However, it is difficult to keep people completely within the language and knowledge boundaries of the system. This is why the use of natural language interfaces is still limited.

**Applications**

- Human computer interaction:
  - Restricted domains – flight reservations, classifying e-mails into a few classes, redirecting caller to one of a few destinations.
  - Limited syntax
  - Limited vocabulary
  - Limited context
  - Limited actions
  - It is very hard for humans to understand what the limits of the system are. Can be frustrating.
- Summarization, Search, Translation
  - Broader domain
  - Performance does not have to be perfect to be useful

tip · Spring 02 · 47

**Sources**

- James Allen, *Natural Language Understanding*, Benjamin/Cummings
- Peter Norvig, *Paradigms of Artificial Intelligence Programming*, Morgan Kauffman
- Slides by Alison Cawsey ([www.cse.hw.ac.uk/~alison/nl.htm](http://www.cse.hw.ac.uk/~alison/nl.htm))

tip · Spring 02 · 48

**Slide 12.3.48**

Here are some sources that were used in the preparation of these slides and which can serve as additional reading material.