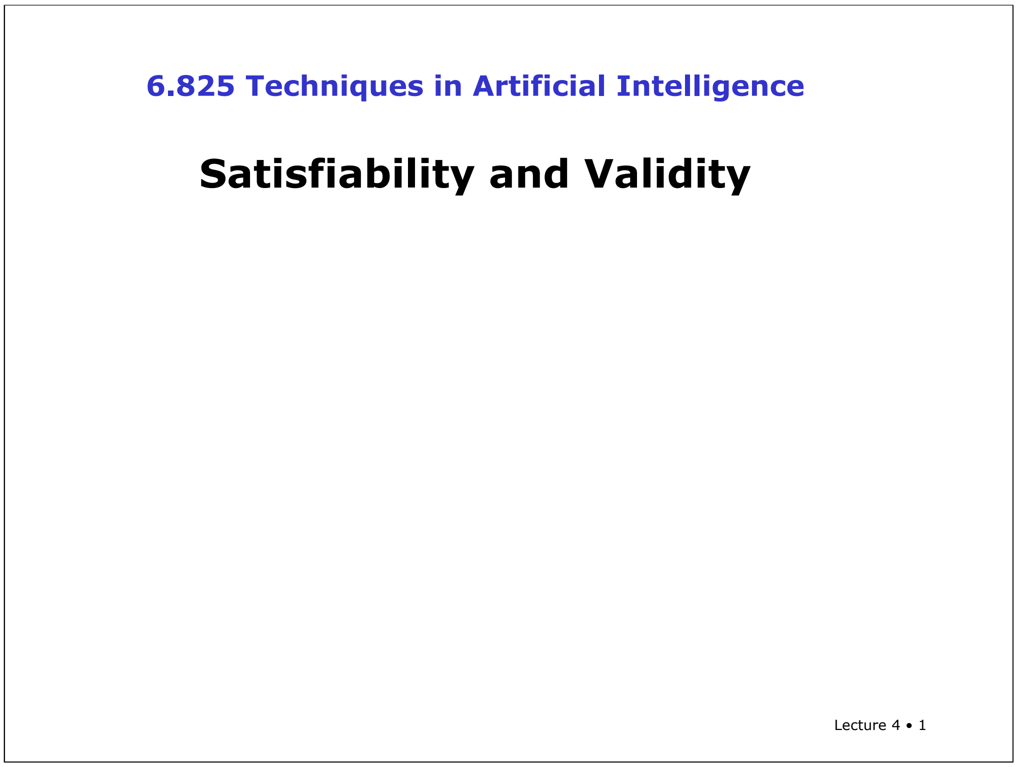
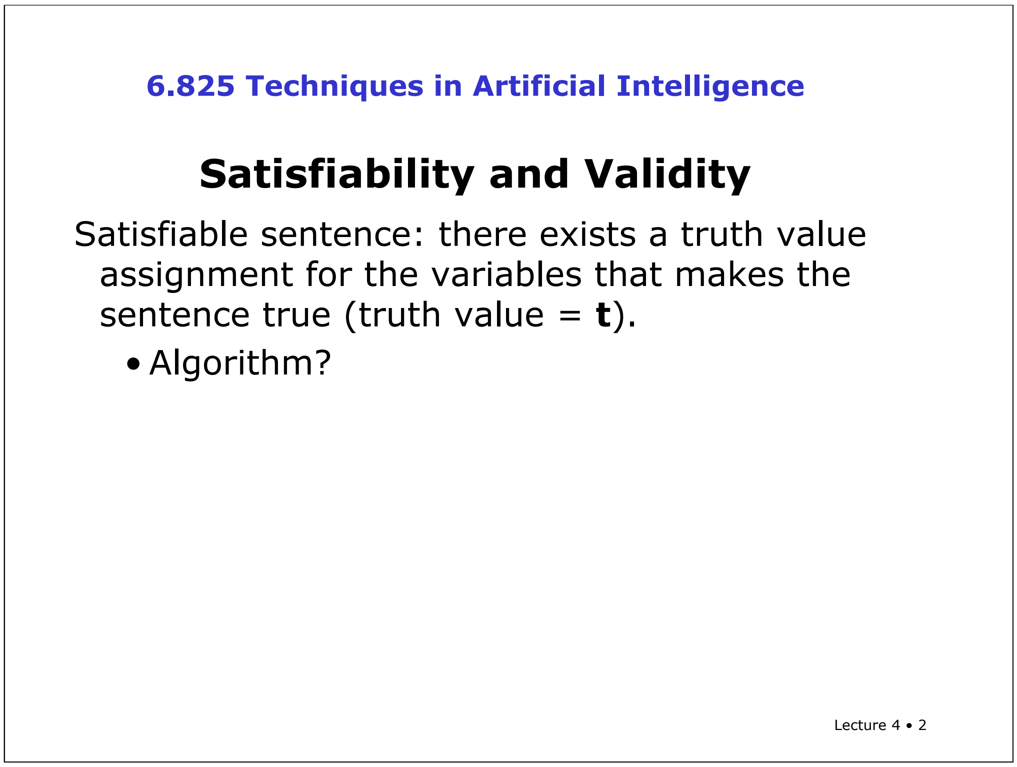


6.825 Techniques in Artificial Intelligence

Satisfiability and Validity

Lecture 4 • 1

Last time we talked about propositional logic. There's no better way to empty out a room than to talk about logic. So now, -- having gone to all that work of establishing syntax and semantics -- what might you actually want to do with some descriptions that are written down in logic? There are two things that we might want to automatically determine about a sentence of logic. One is satisfiability, and another is validity.



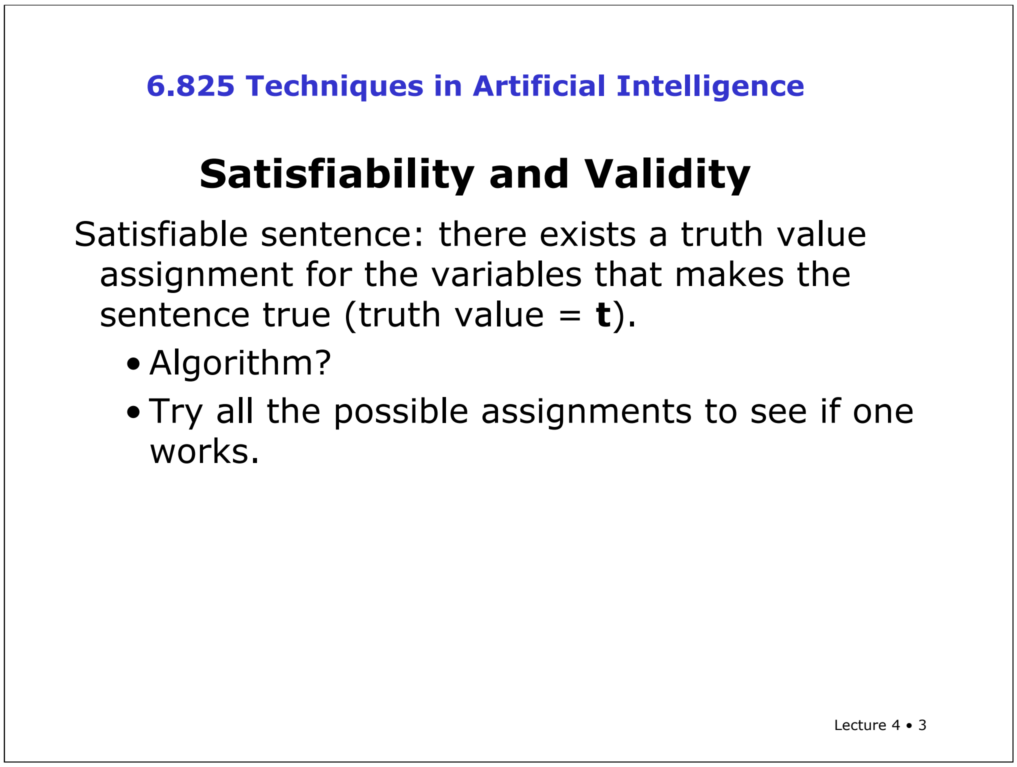
6.825 Techniques in Artificial Intelligence

Satisfiability and Validity

Satisfiable sentence: there exists a truth value assignment for the variables that makes the sentence true (truth value = **t**).

- Algorithm?

Last time we talked about a way to determine whether a sentence is satisfiable. Can you remember what it is? You know an algorithm for this.

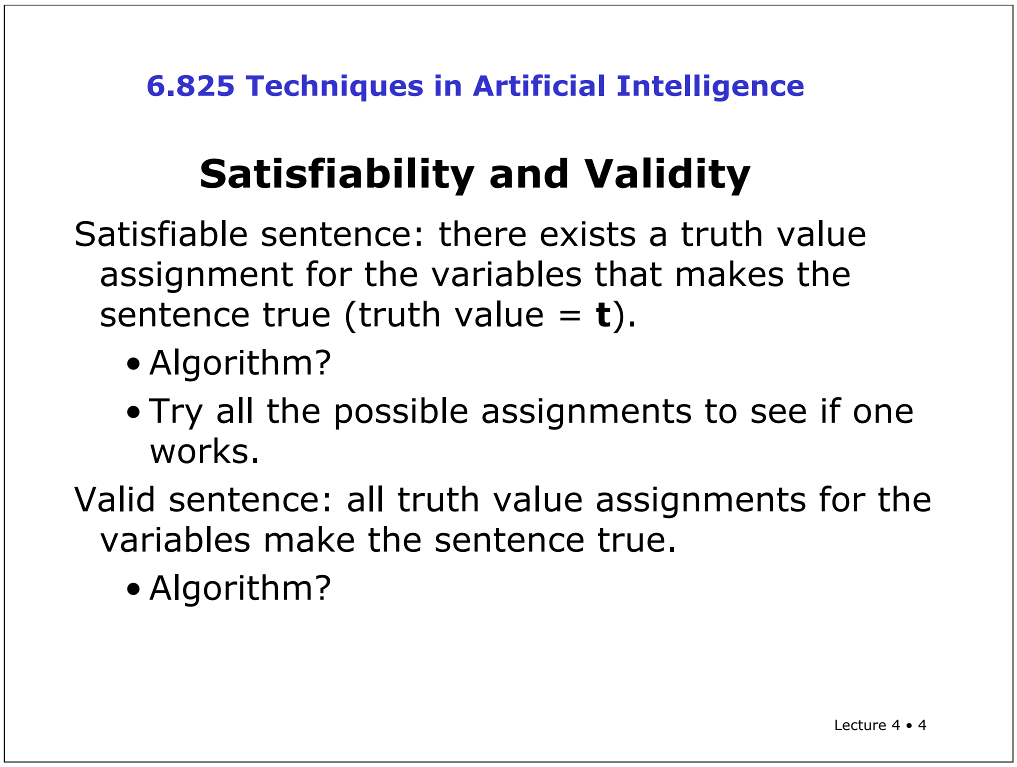


Satisfiability and Validity

Satisfiable sentence: there exists a truth value assignment for the variables that makes the sentence true (truth value = **t**).

- Algorithm?
- Try all the possible assignments to see if one works.

Try all possible assignments and see if there is one that makes the sentence true.



6.825 Techniques in Artificial Intelligence

Satisfiability and Validity

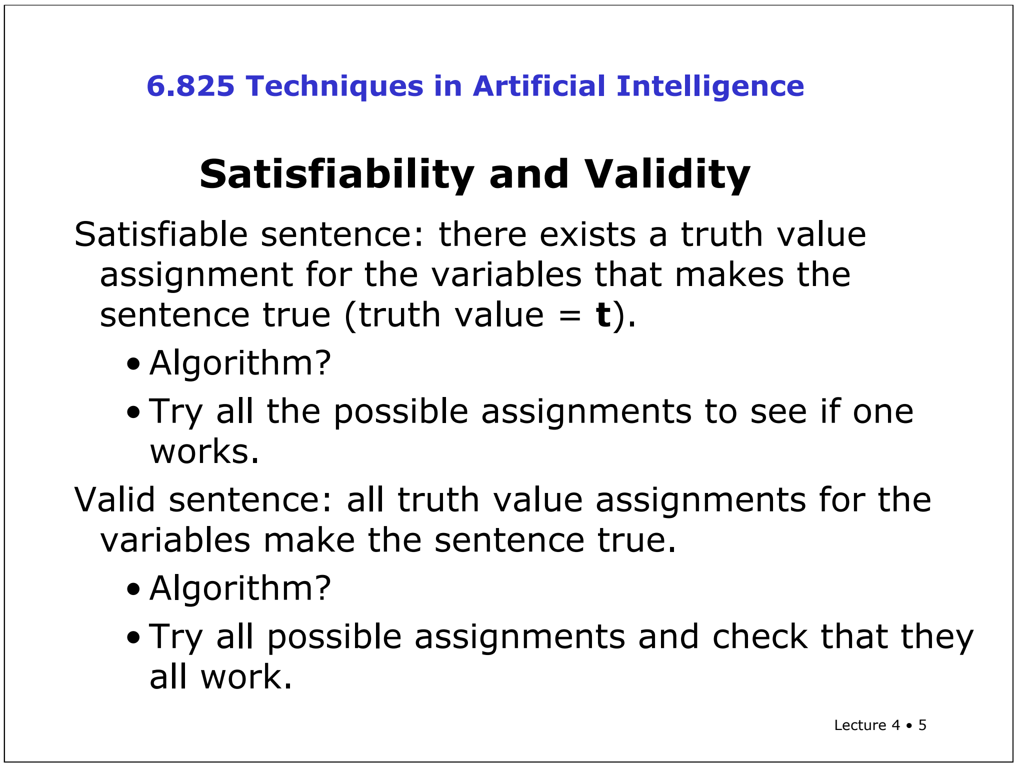
Satisfiable sentence: there exists a truth value assignment for the variables that makes the sentence true (truth value = **t**).

- Algorithm?
- Try all the possible assignments to see if one works.

Valid sentence: all truth value assignments for the variables make the sentence true.

- Algorithm?

And how do you tell if a sentence is valid? What's the algorithm?



Satisfiability and Validity

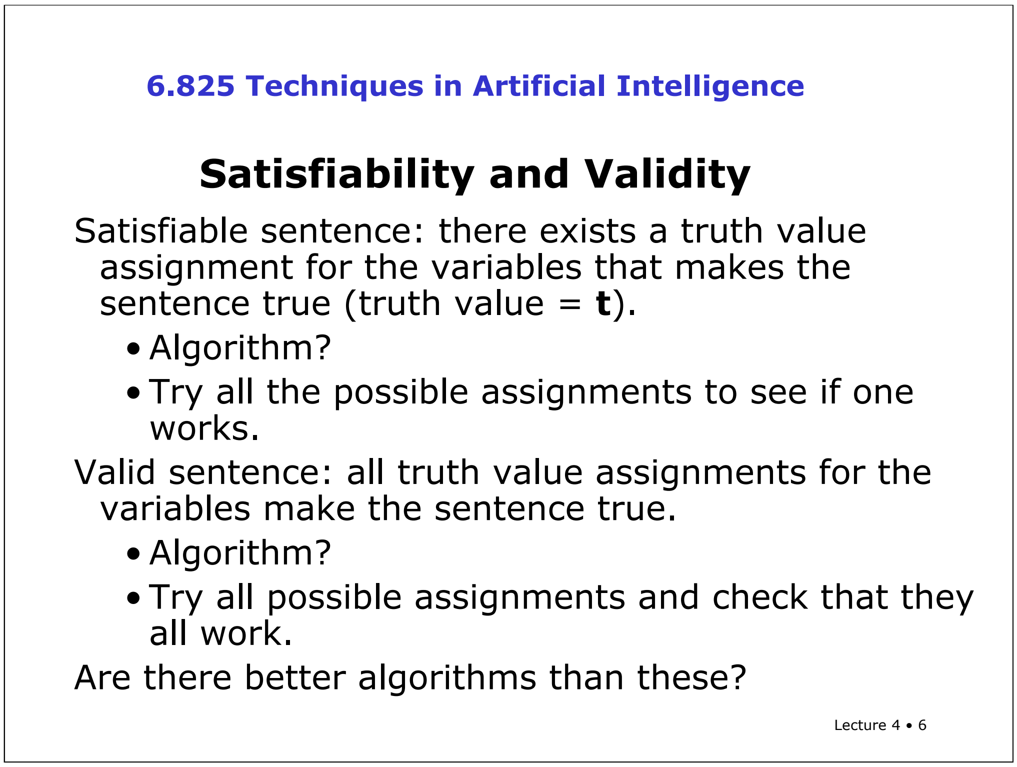
Satisfiable sentence: there exists a truth value assignment for the variables that makes the sentence true (truth value = **t**).

- Algorithm?
- Try all the possible assignments to see if one works.

Valid sentence: all truth value assignments for the variables make the sentence true.

- Algorithm?
- Try all possible assignments and check that they all work.

Try all possible assignments and be sure that all of them make the sentence true.



6.825 Techniques in Artificial Intelligence

Satisfiability and Validity

Satisfiable sentence: there exists a truth value assignment for the variables that makes the sentence true (truth value = **t**).

- Algorithm?
- Try all the possible assignments to see if one works.

Valid sentence: all truth value assignments for the variables make the sentence true.

- Algorithm?
- Try all possible assignments and check that they all work.

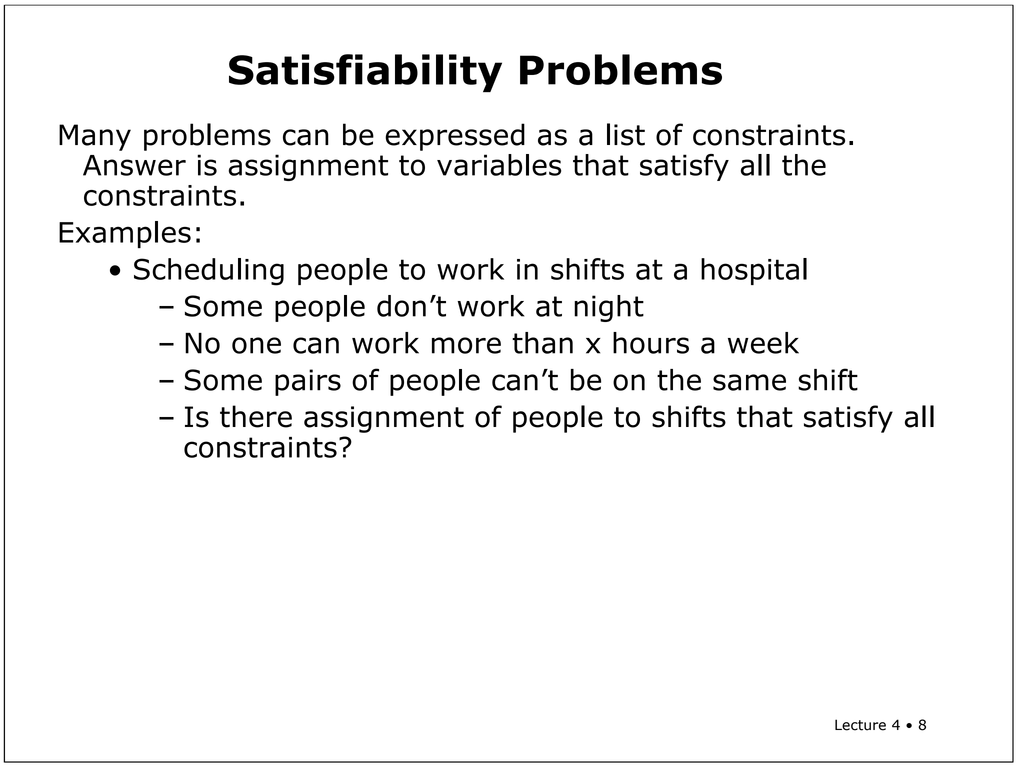
Are there better algorithms than these?

We're going to spend some time talking about better ways to compute satisfiability and better ways to compute validity.

Satisfiability Problems

Many problems can be expressed as a list of constraints. Answer is assignment to variables that satisfy all the constraints.

There are lots of satisfiability problems in the real world. They end up being expressed essentially as lists of constraints, where you're trying to find some assignment of values to variables that satisfy the constraints.



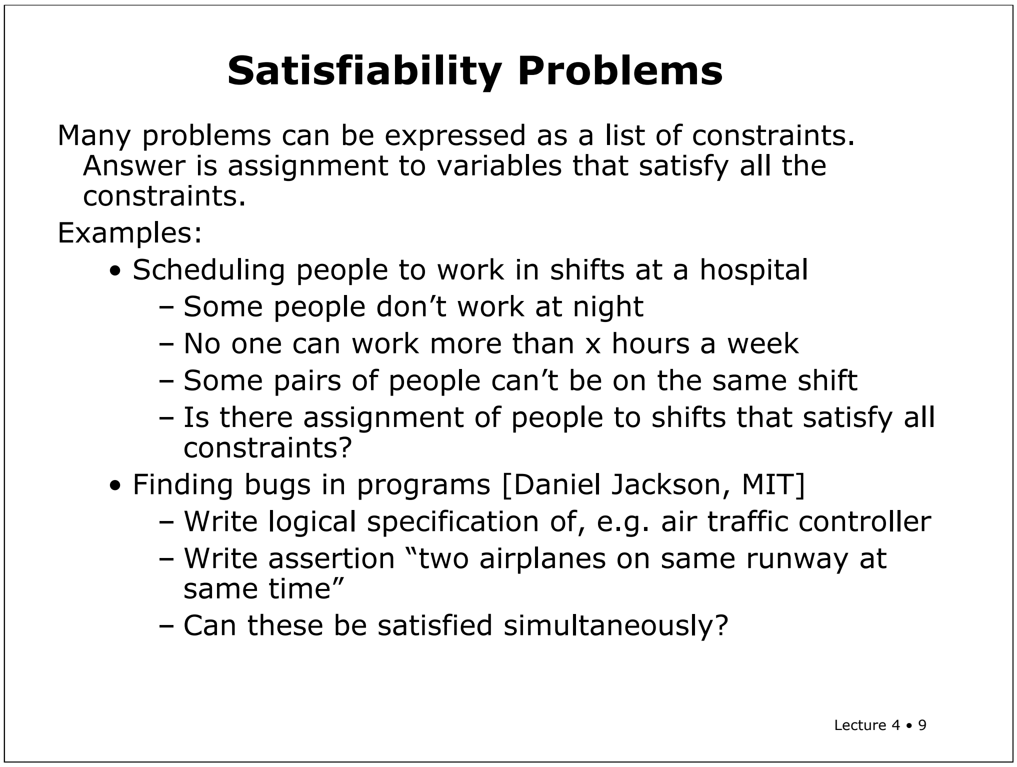
Satisfiability Problems

Many problems can be expressed as a list of constraints. Answer is assignment to variables that satisfy all the constraints.

Examples:

- Scheduling people to work in shifts at a hospital
  - Some people don't work at night
  - No one can work more than x hours a week
  - Some pairs of people can't be on the same shift
  - Is there assignment of people to shifts that satisfy all constraints?

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

Many problems can be expressed as a list of constraints. Answer is assignment to variables that satisfy all the constraints.

Examples:

- Scheduling people to work in shifts at a hospital
  - Some people don't work at night
  - No one can work more than x hours a week
  - Some pairs of people can't be on the same shift
  - Is there assignment of people to shifts that satisfy all constraints?
- Finding bugs in programs [Daniel Jackson, MIT]
  - Write logical specification of, e.g. air traffic controller
  - Write assertion "two airplanes on same runway at same time"
  - Can these be satisfied simultaneously?

Lecture 4 • 9

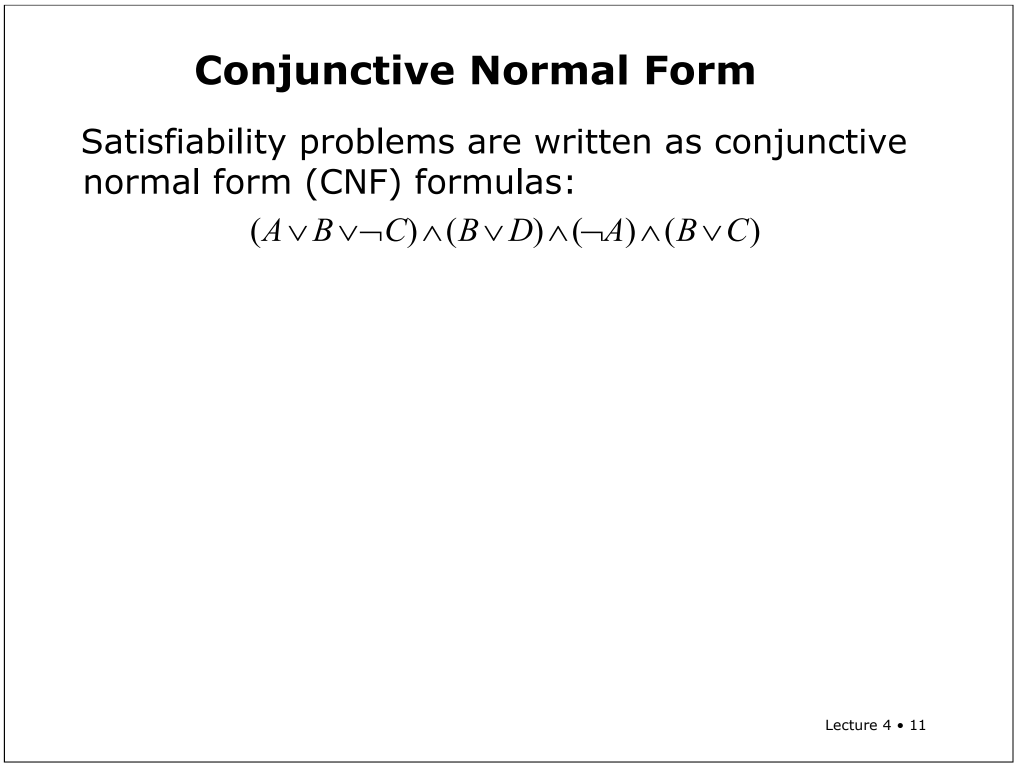
There's an interesting application of satisfiability that's going on here at MIT in the Lab for Computer Science. Professor Daniel Jackson's interested in trying to find bugs in programs. That's a good thing to do, but (as you know!) it's hard for humans to do reliably, so he wants to get the computer 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 all these rules about how it works, right?   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 -- if that whole sentence is satisfiable, then you have a problem in your air traffic control protocol.

Conjunctive Normal Form

Satisfiability problems are written as conjunctive normal form (CNF) formulas:

Satisfiability problems are typically written as sets of constraints, and that means that they're often written – just about always written -- in conjunctive normal form.



Conjunctive Normal Form

Satisfiability problems are written as conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

A sentence is written in conjunctive normal form looks like ((A or B or not C) and (B or D) and (not A) and (B or C or F)). Or something like that.

Conjunctive Normal Form

Satisfiability problems are written as 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

Its outermost structure is a conjunction. It's a conjunction of multiple units. These units are called "clauses."

Conjunctive Normal Form

Satisfiability problems are written as 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 ¬ C are literals

A clause is the disjunction of many things. The units that make up a clause are called literals.

Conjunctive Normal Form

Satisfiability problems are written as 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 ¬ C are literals, each of which is a variable or the negation of a variable.

And a literal is either a variable or the negation of a variable.

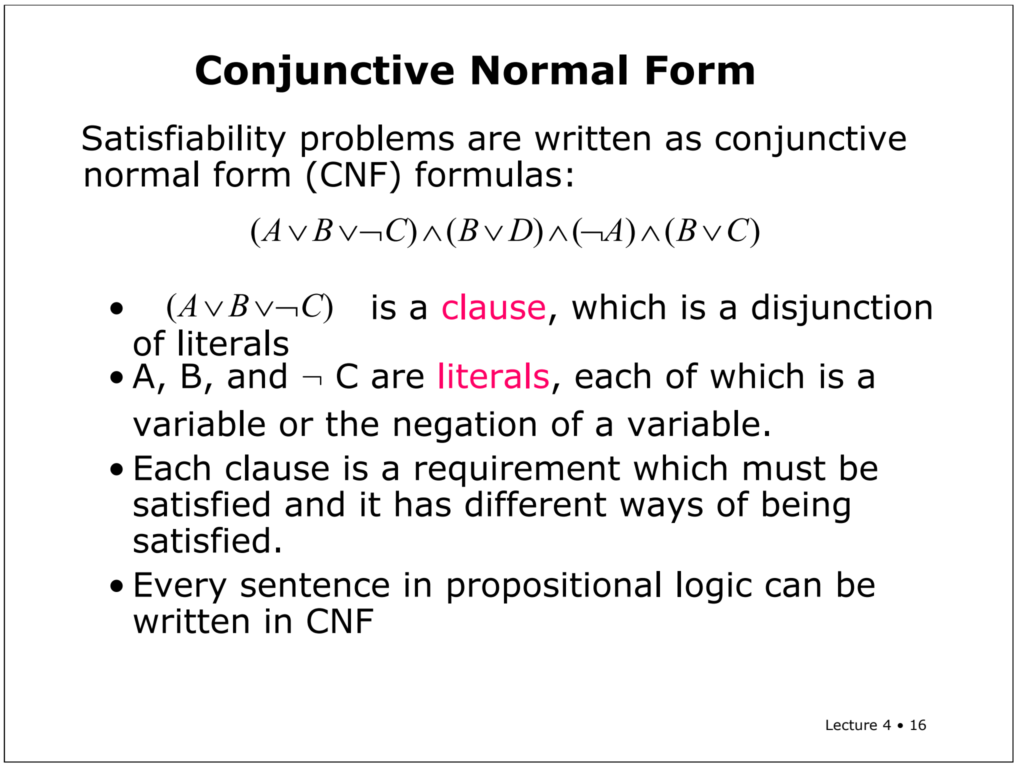
Conjunctive Normal Form

Satisfiability problems are written as 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 ¬ C are literals, each of which is a variable or the negation of a variable.
- Each clause is a requirement which must be satisfied and it has different ways of being satisfied.

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

Satisfiability problems are written as conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

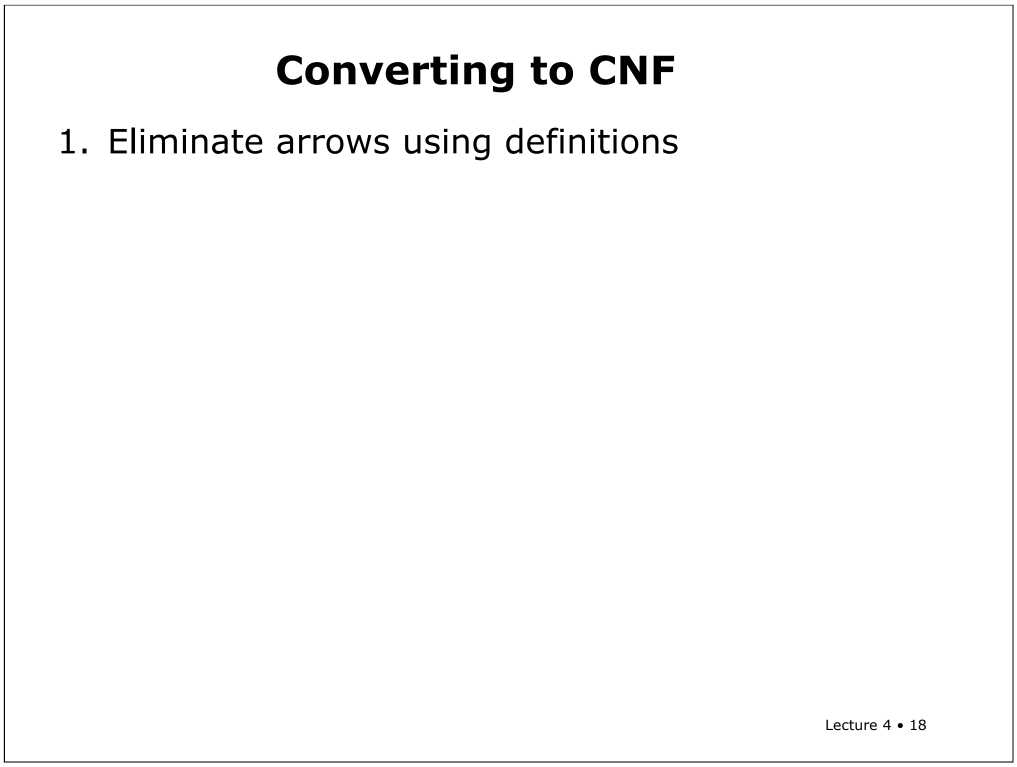
- $(A \vee B \vee \neg C)$ is a clause, which is a disjunction of literals
- A, B, and ¬ C are literals, each of which is a variable or the negation of a variable.
- Each clause is a requirement which must be satisfied and it has different ways of being satisfied.
- Every sentence in propositional logic can be written in CNF

You can take any sentence in propositional logic and write it in conjunctive normal form.

Converting to CNF

Lecture 4 • 17

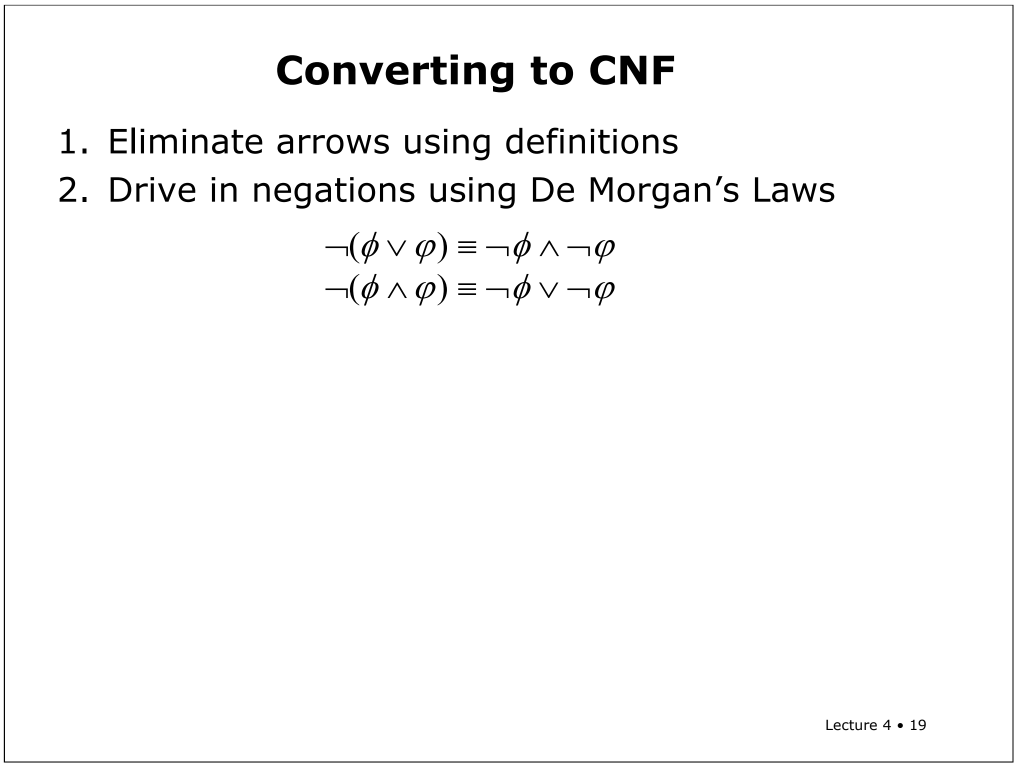
Here’s the procedure for converting sentences to conjunctive normal form.



Converting to CNF

1. Eliminate arrows using definitions

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

$$\neg(\phi \vee \varphi) \equiv \neg\phi \wedge \neg\varphi$$
$$\neg(\phi \wedge \varphi) \equiv \neg\phi \vee \neg\varphi$$

The next step is to drive in negation. We do it using DeMorgan's Laws. You might have seen them in a digital logic class. Not (phi or psi) is equivalent to (not phi and not psi). And, Not (phi and psi) is equivalent to (not phi or not psi).

So if you have a negation on the outside, you can push it in and change the connective from **and** to **or**, or from **or** to **and**.

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\neg(\phi \vee \varphi) \equiv \neg\phi \wedge \neg\varphi$$
$$\neg(\phi \wedge \varphi) \equiv \neg\phi \vee \neg\varphi$$

3. Distribute **or** over **and**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

The third step is to distribute **or** over **and**. That is, if we have (A or (B and C)) we can rewrite it as (A or B) and (A 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

$$\neg(\phi \vee \varphi) \equiv \neg\phi \wedge \neg\varphi$$
$$\neg(\phi \wedge \varphi) \equiv \neg\phi \vee \neg\varphi$$

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

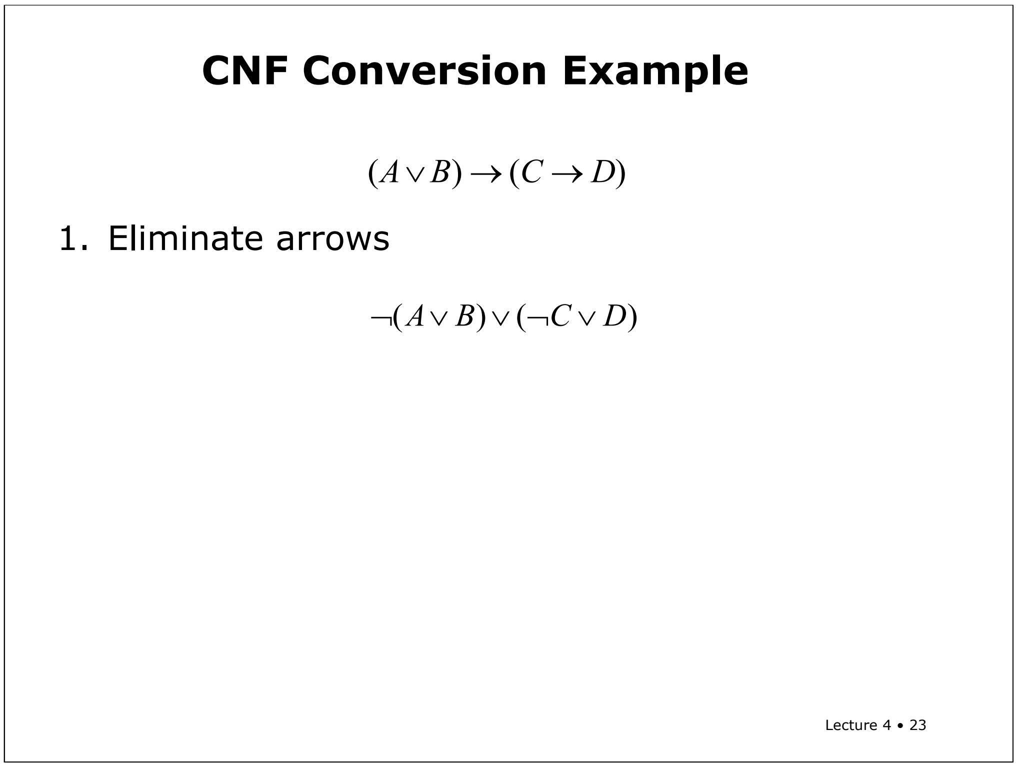
One problem with conjunctive normal form is that, although you can convert any sentence to conjunctive normal form, you might do it at the price of an exponential increase in the size of the expression. Because if you have A and B and C OR D and E and F, you end up making the cross- product of all of those things.

For now, we'll think about satisfiability problems, which are generally fairly efficiently converted into CNF. But on homework 1, we'll have to think a lot about the size of expressions in CNF.

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

Here's an example of converting a sentence to CNF.



CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

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

$$(A \vee B) \rightarrow (C \rightarrow D)$$

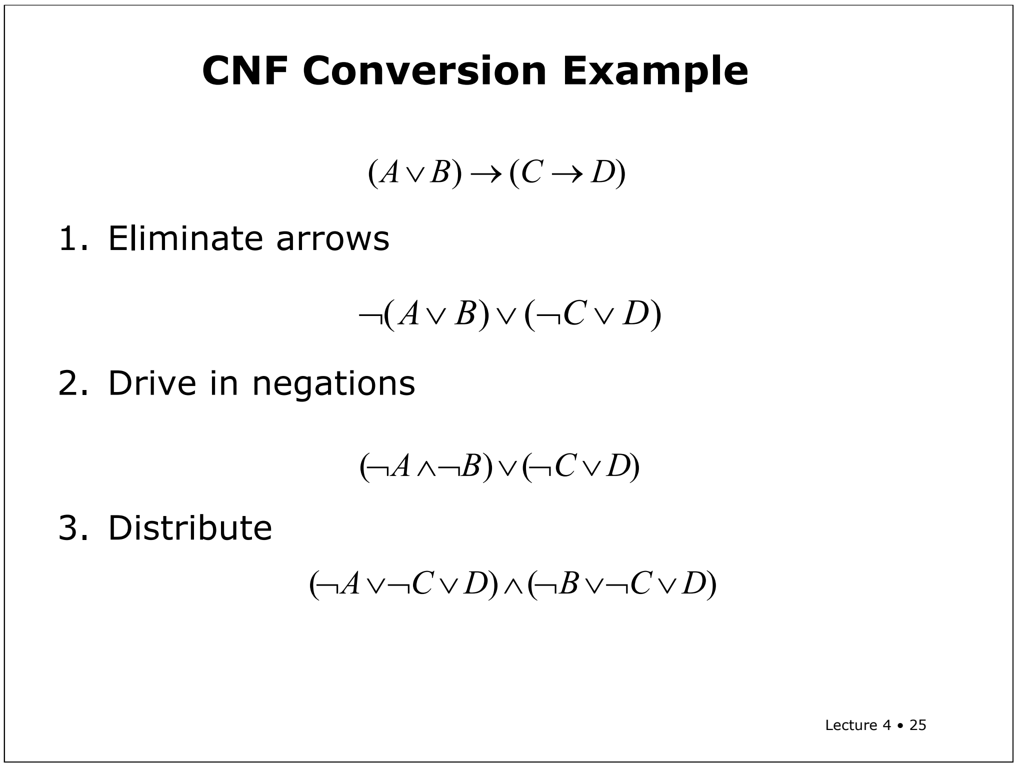
1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

2. Drive in negations

$$(\neg A \wedge \neg B) \vee (\neg C \vee D)$$

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

$$(\neg A \vee \neg C \vee D) \wedge (\neg B \vee \neg C \vee D)$$

Finally, we dstribute **or** over **and** to get the final CNF expression.

Simplifying CNF

We’re going to be doing a lot of manipulations of CNF sentences, and we’ll sometimes end up with these degenerate cases. Let’s understand what they mean.

Simplifying CNF

- An empty clause is false (no options to satisfy)

An empty clause is false. In general, a disjunction with no disjuncts is false. In order to make such an expression true, you have to satisfy one of the options, and if there aren't any, you can't make it true.

Simplifying CNF

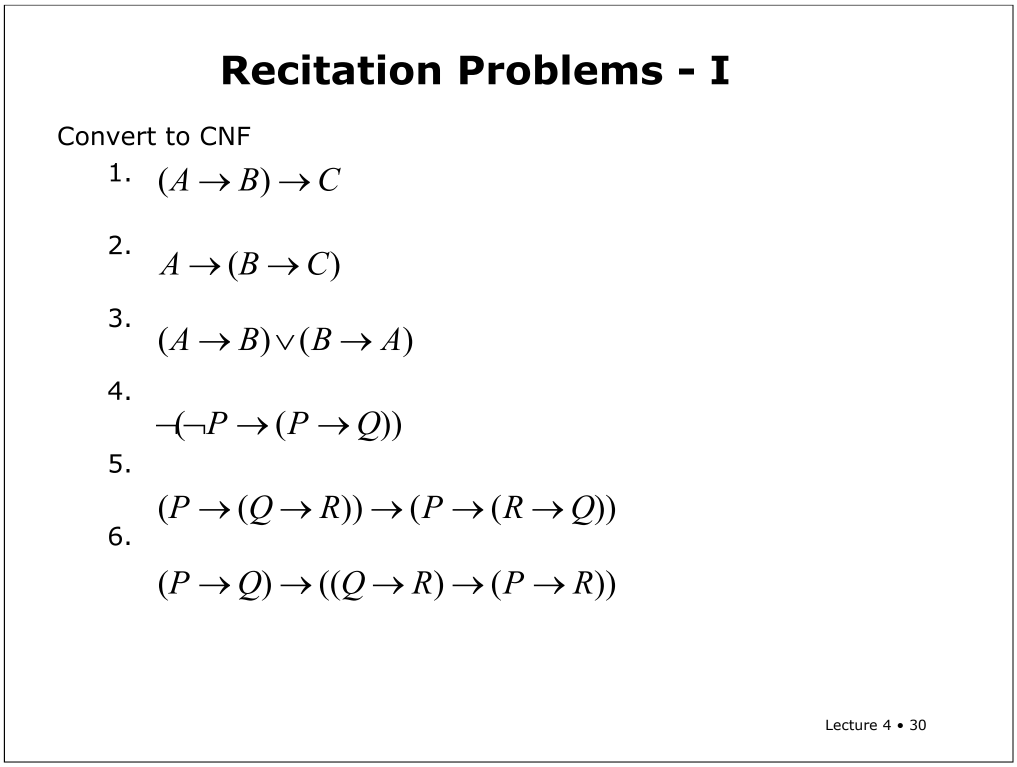
- An empty clause is false (no options to satisfy)
- A sentence with no clauses is true (no requirements)

A sentence with no clauses is true. In general a conjunction with no conjuncts is true. In order to make such an expression true, you have to make all of its conditions true, and if there aren't any, then it's just true.

Simplifying CNF

- An empty clause is false (no options to satisfy)
- A sentence with no clauses is true (no requirements)
- A sentence containing an empty clause is false (there is an impossible requirement)

A sentence containing an empty clause is false. This is because the empty clause is false, and false conjoined with anything else is always false.



Recitation Problems - I

Convert to CNF

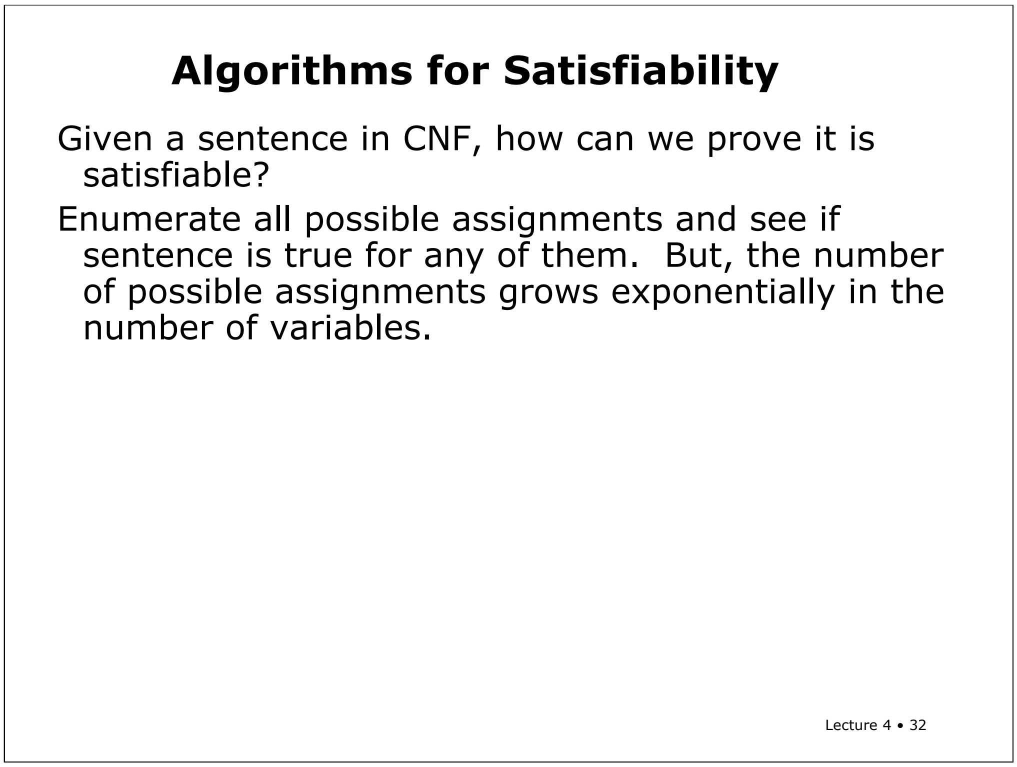
1. $(A \rightarrow B) \rightarrow C$
2. $A \rightarrow (B \rightarrow C)$
3. $(A \rightarrow B) \vee (B \rightarrow A)$
4. $\neg(\neg P \rightarrow (P \rightarrow Q))$
5. $(P \rightarrow (Q \rightarrow R)) \rightarrow (P \rightarrow (R \rightarrow Q))$
6. $(P \rightarrow Q) \rightarrow ((Q \rightarrow R) \rightarrow (P \rightarrow R))$

Please do at least two of these problems before going on with the rest of the lecture (and do the rest of them before recitation).

Algorithms for Satisfiability

Given a sentence in CNF, how can we prove it is satisfiable?

How can we prove that a CNF sentence is satisfiable? By showing that there is a satisfying assignment, that is, an assignment of truth values to variables that makes the sentence true. So, we have to try to find a satisfying assignment.

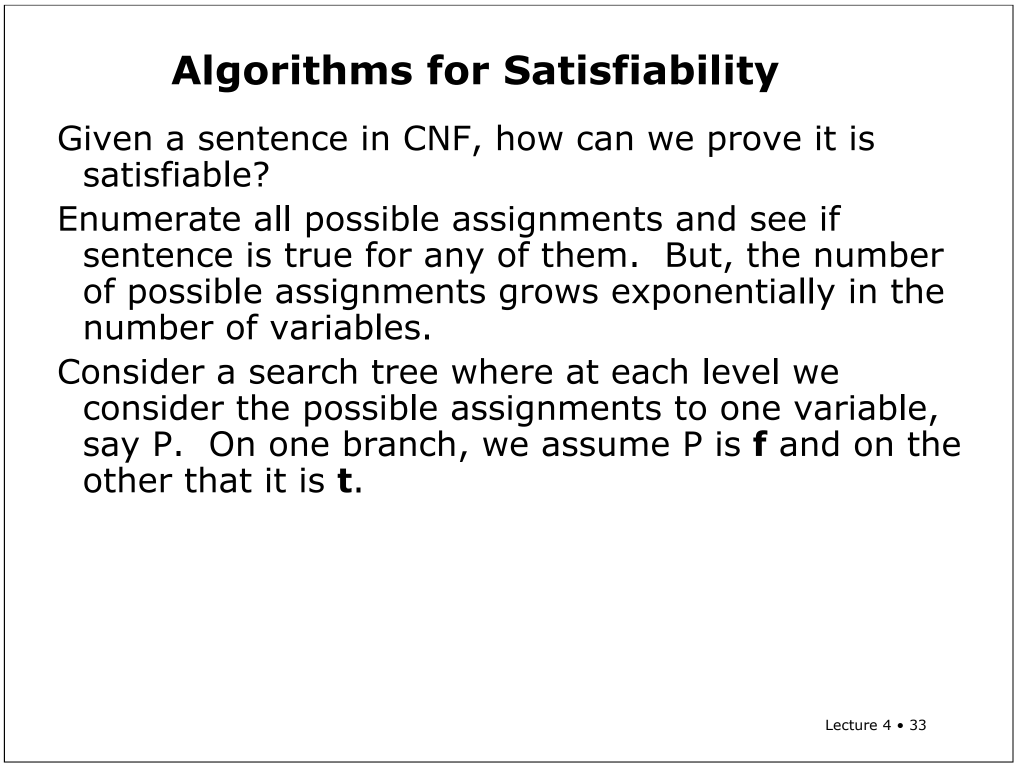


Algorithms for Satisfiability

Given a sentence in CNF, how can we prove it is satisfiable?

Enumerate all possible assignments and see if sentence is true for any of them. But, the number of possible assignments grows exponentially in the number of variables.

One strategy would be to enumerate all possible assignments, and evaluate the sentence in each one. But the number of possible assignments grows in the number of variables, and it would be way too slow.



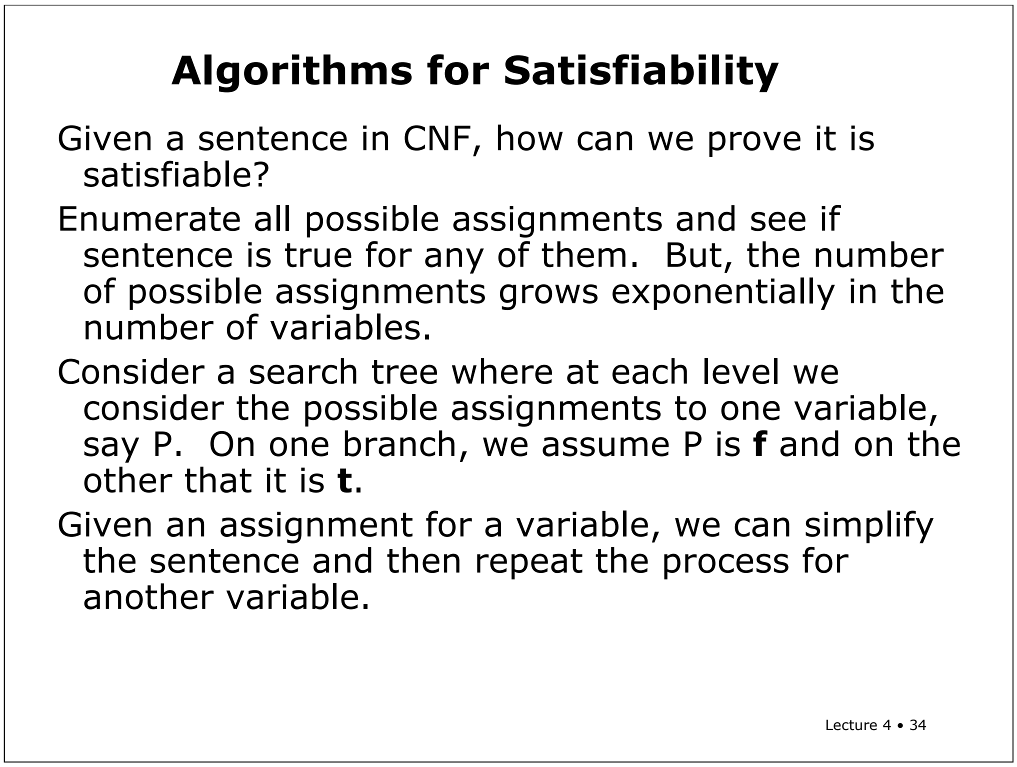
Algorithms for Satisfiability

Given a sentence in CNF, how can we prove it is satisfiable?

Enumerate all possible assignments and see if sentence is true for any of them. But, the number of possible assignments grows exponentially in the number of variables.

Consider a search tree where at each level we consider the possible assignments to one variable, say P. On one branch, we assume P is **f** and on the other that it is **t**.

Let's make a search tree. We'll start out by considering the possible assignments that we can make to the variable P. We can assign it true or false.



Algorithms for Satisfiability

Given a sentence in CNF, how can we prove it is satisfiable?

Enumerate all possible assignments and see if sentence is true for any of them. But, the number of possible assignments grows exponentially in the number of variables.

Consider a search tree where at each level we consider the possible assignments to one variable, say P. On one branch, we assume P is **f** and on the other that it is **t**.

Given an assignment for a variable, we can simplify the sentence and then repeat the process for another variable.

Now, if I assign P "false", that simplifies my problem a little bit. You could say, before I made any variable assignments, I had to find an assignment to all the variables that would satisfy this set of requirements. Having assigned P the value "false", now there is a simpler set of requirements on the rest of the assignment.

Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

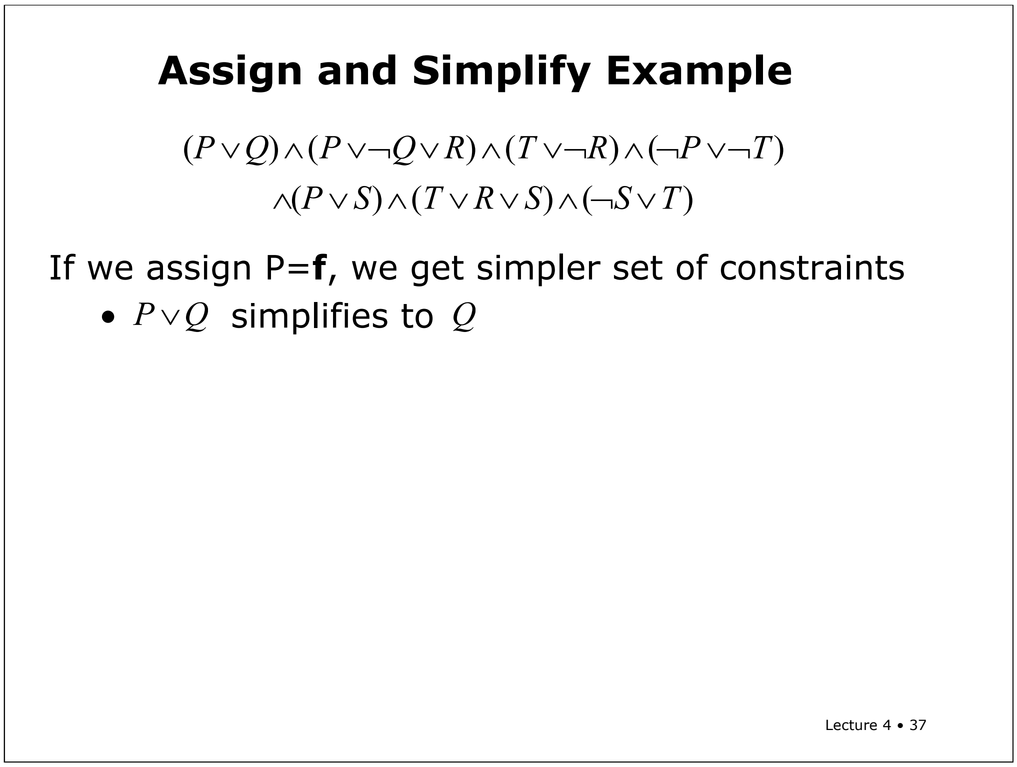
So let's think about how we can simplify a sentence based on a partial assignment. Here’s a complicated sentence. Let’s actually figure out how to simplify the sentence in this case, and then we can write down the general rule.

Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

If we assign P=**f**, we get simpler set of constraints

OK, so if I assign P the value "false", what happens?



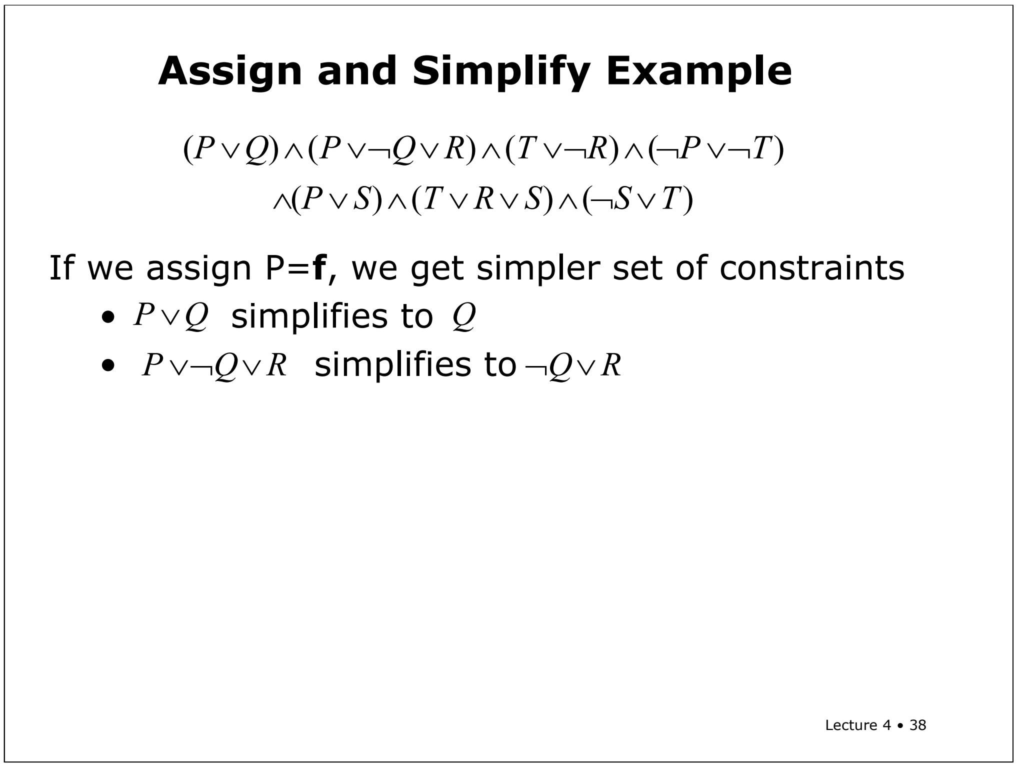
Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

If we assign P=**f**, we get simpler set of constraints

- $P \vee Q$ simplifies to Q

The first clause (P or Q) simplifies to Q. If we force P to be false, then the only possible way to satisfy is requirement is for Q to be true.



Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

If we assign P=**f**, we get simpler set of constraints

- $P \vee Q$ simplifies to Q
- $P \vee \neg Q \vee R$ simplifies to $\neg Q \vee R$

Similarly, (P or not Q or R) simplifies to (not Q or R).

Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

If we assign P=**f**, we get simpler set of constraints

- $P \vee Q$ simplifies to Q
- $P \vee \neg Q \vee R$ simplifies to $\neg Q \vee R$
- $\neg P \vee \neg T$ is satisfied and can be removed

The clause (not P or not T) can be removed entirely. Once we've decided to make P false, we've satisfied this clauses (made it true) and we don't have to worry about it any more.

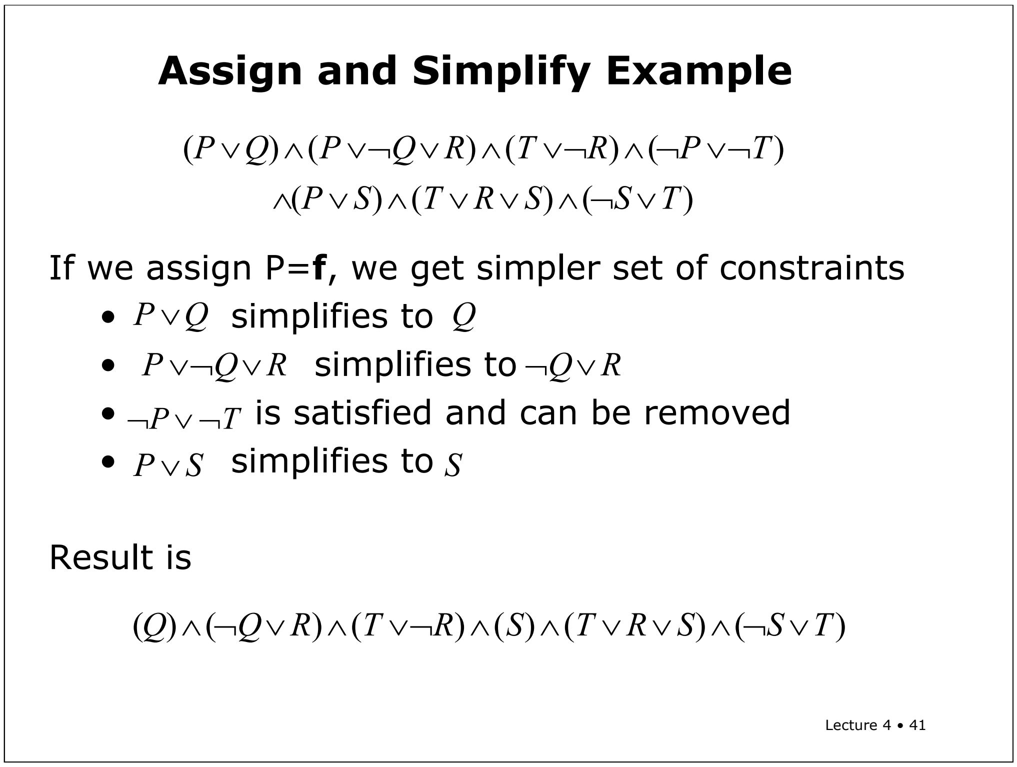
Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

If we assign P=**f**, we get simpler set of constraints

- $P \vee Q$ simplifies to Q
- $P \vee \neg Q \vee R$ simplifies to $\neg Q \vee R$
- $\neg P \vee \neg T$ is satisfied and can be removed
- $P \vee S$ simplifies to S

P or S simplifies to S.



Assign and Simplify Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

If we assign P=**f**, we get simpler set of constraints

- $P \vee Q$ simplifies to Q
- $P \vee \neg Q \vee R$ simplifies to $\neg Q \vee R$
- $\neg P \vee \neg T$ is satisfied and can be removed
- $P \vee S$ simplifies to S

Result is

$$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R) \wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

So, now we have a resulting expression that doesn’t mention P, and is simpler than the one we started with.

Assign and Simplify

Given a CNF sentence ϕ and a literal U

So, a little bit more formally, the “assign and simplify” process goes like this:

Given a CNF sentence phi and a literal U (remember a literal is either a variable or a negated variable),

Assign and Simplify

Given a CNF sentence ϕ and a literal U

- Delete all clauses containing U (they're satisfied)

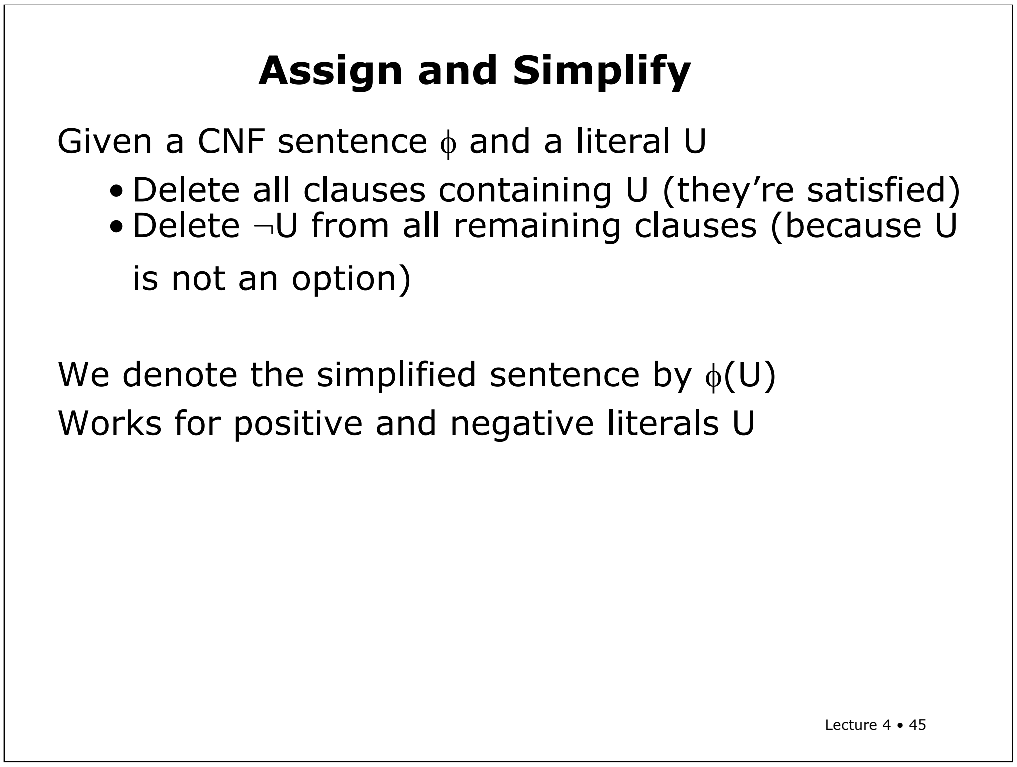
delete all clauses from Phi that contain U (because they're satisfied)

Assign and Simplify

Given a CNF sentence ϕ and a literal U

- Delete all clauses containing U (they're satisfied)
- Delete $\neg$U from all remaining clauses (because U is not an option)

delete not U from all remaining clauses (because U is not an option)



Assign and Simplify

Given a CNF sentence ϕ and a literal U

- Delete all clauses containing U (they're satisfied)
- Delete $\neg$U from all remaining clauses (because U is not an option)

We denote the simplified sentence by ϕ(U)
Works for positive and negative literals U

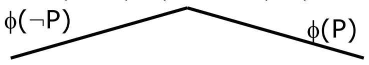
We'll call the resulting sentence phi of u.

Search Example

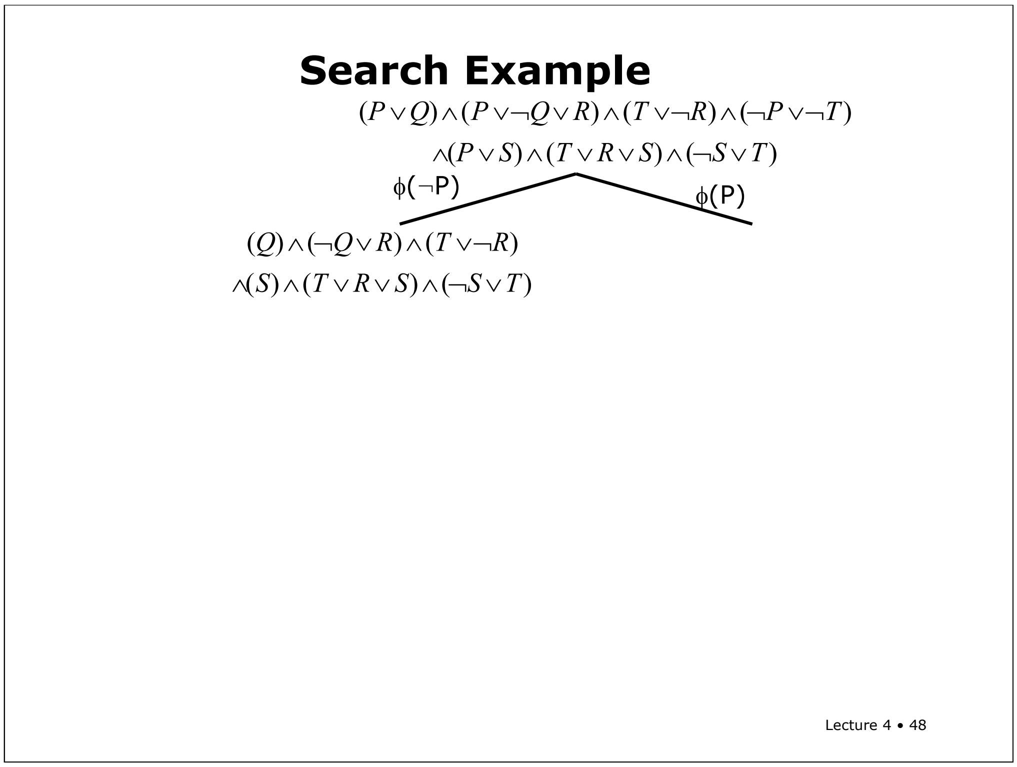
Here’s a big example, illustrating a tree-structured process of searching for a satisfying assignment by assigning values to variables and simplifying the resulting expressions.

Search Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$



We’ll start with our previous example formula. And we’ll arbitrarily pick the variable P to start with and consider what happens if we assign it to have the value **f**.



Search Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

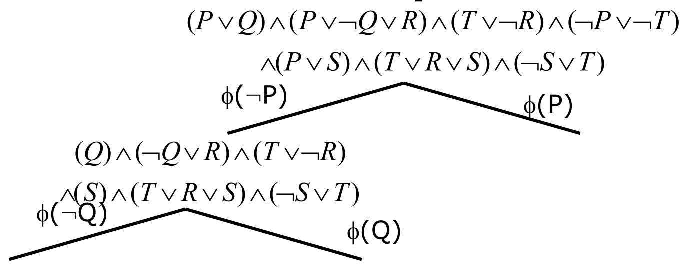
$\phi(\neg P)$ $\phi(P)$

$$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$$
$$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

Lecture 4 • 48

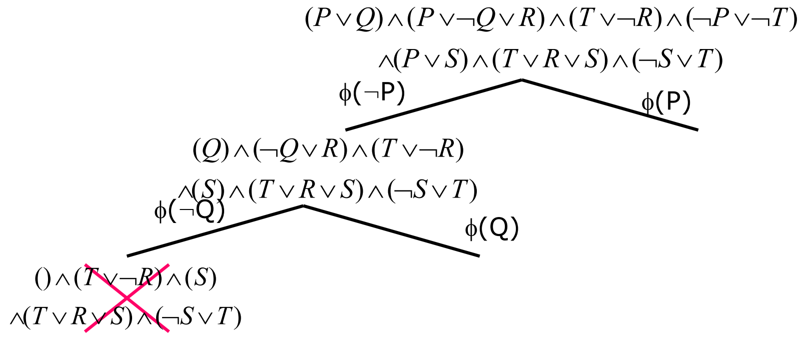
We do an “assign and simplify” operation, and end up with the smaller expression we got when we did this example before.

Search Example


$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$
$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
ϕ(¬P)
ϕ(P)
$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$
$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
ϕ(¬Q)
ϕ(Q)

Now, let’s pick Q as our variable, and try assigning it to **f**.

Search Example

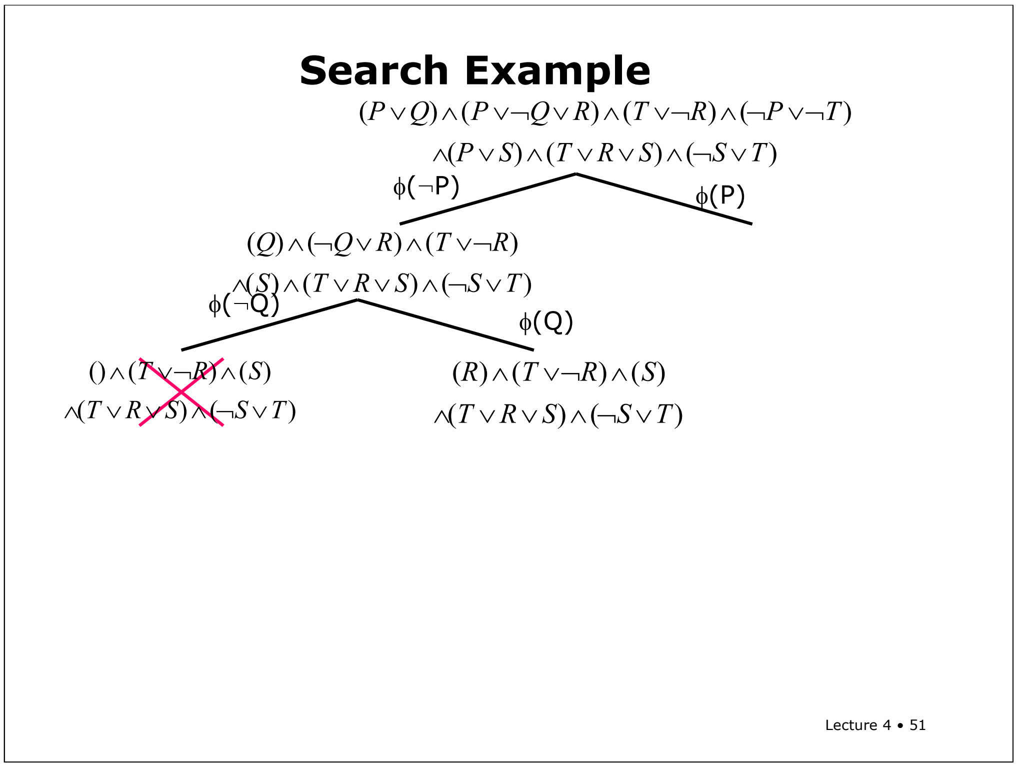


$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$$
$$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$$\cancel{() \wedge (T \vee \neg R) \wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)}$$

When we assign and simplify, we find that the resulting expression has an empty clause, which means that it’s false. That means that, given the assignments we’ve made on this path of the tree (P false and Q false), the sentence is unsatisfiable. There’s no reason to continue on with this branch, so we’ll have to back up and try a different choice somewhere.



Search Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$\phi(\neg P)$ $\phi(P)$

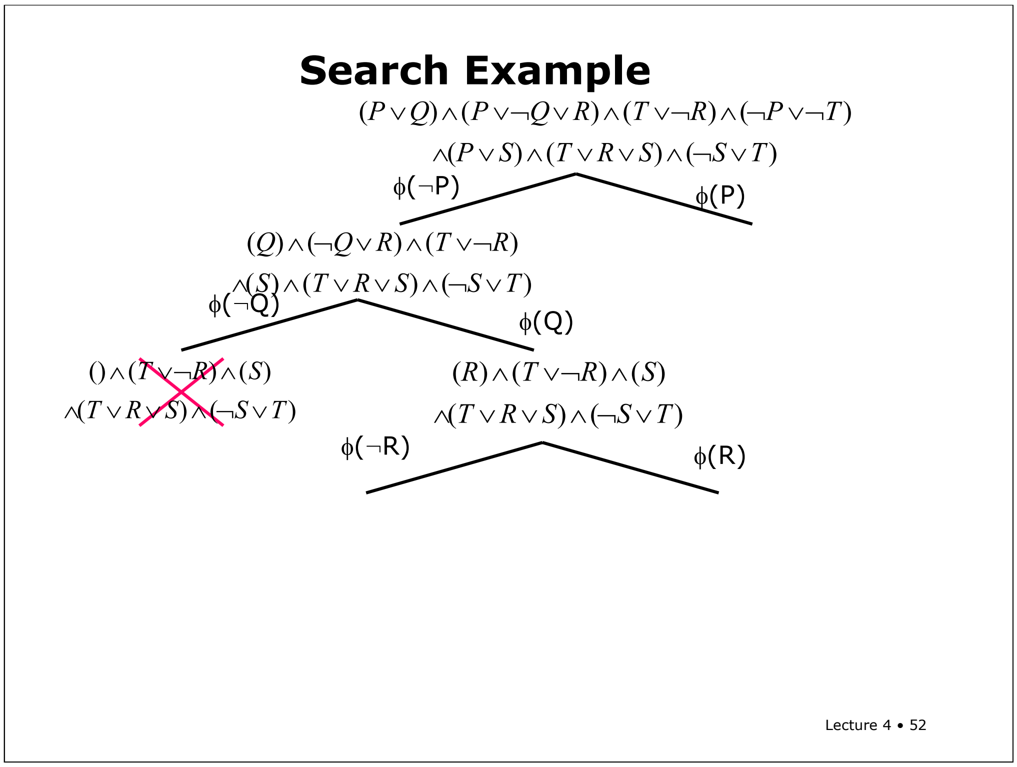
$$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$$
$$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$\phi(\neg Q)$ $\phi(Q)$

$$\cancel{() \wedge (T \vee \neg R) \wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)}$$

$$(R) \wedge (T \vee \neg R) \wedge (S)$$
$$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

Let’s go up to our most recent decision and try assigning Q to be **t**. Simplifying gives us this expression.



Search Example

$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T) \wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R) \wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

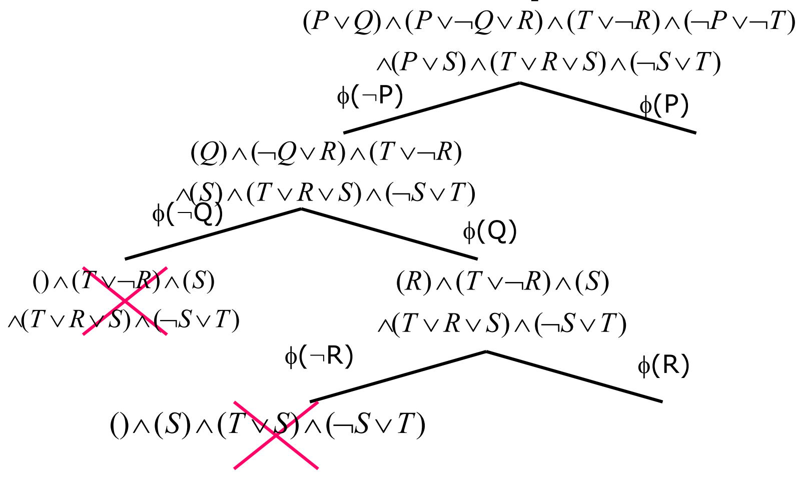
$$\cancel{() \wedge (T \vee \neg R) \wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)}$$

$$(R) \wedge (T \vee \neg R) \wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

Lecture 4 • 52

Now, let’s try assigning R to be **f**.

Search Example



$$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$$
$$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

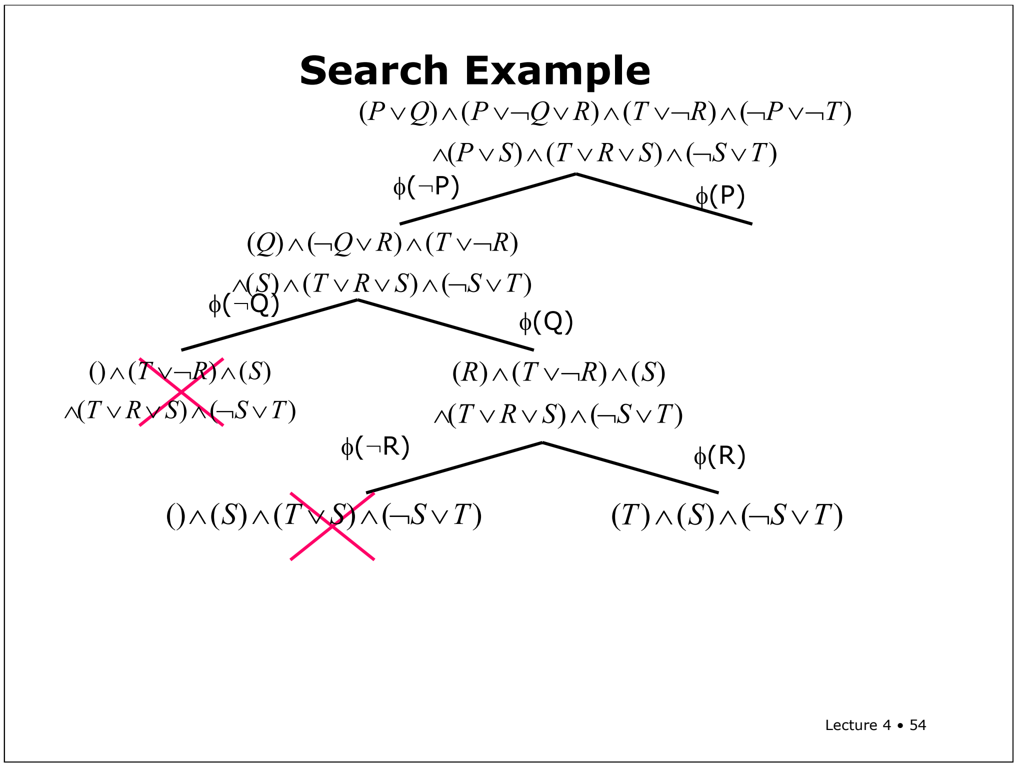
$$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$$
$$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$$() \wedge (T \vee \neg R) \wedge (S)$$
$$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$$(R) \wedge (T \vee \neg R) \wedge (S)$$
$$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$$

$$() \wedge (S) \wedge (T \vee S) \wedge (\neg S \vee T)$$

Again, when we assign and simplify, we get an empty clause, signaling failure.

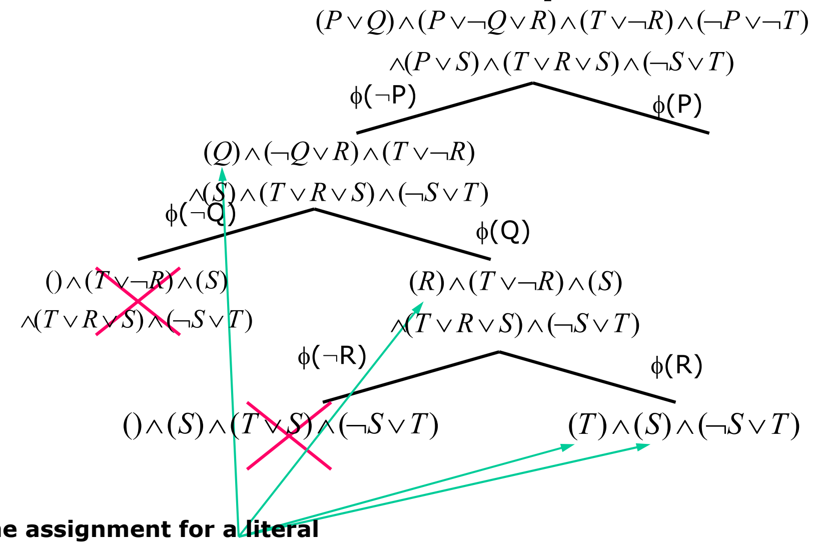


Search Example

$$\begin{tikzcd}[column sep=small]
 & & {(P\lor Q)\land(P\lor\neg Q\lor R)\land(T\lor\neg R)\land(\neg P\lor\neg T) \land(P\lor S)\land(T\lor R\lor S)\land(\neg S\lor T)} \arrow[dl, dash, "\phi(\neg P)"'] \arrow[dr, dash, "\phi(P)"] & \\
 & {(Q)\land(\neg Q\lor R)\land(T\lor\neg R) \land(S)\land(T\lor R\lor S)\land(\neg S\lor T)} \arrow[dl, dash, "\phi(\neg Q)"'] \arrow[dr, dash, "\phi(Q)"] & & {} \\
 {\cancel{()\land(T\lor\neg R)\land(S) \land(T\lor R\lor S)\land(\neg S\lor T)}} & & {(R)\land(T\lor\neg R)\land(S) \land(T\lor R\lor S)\land(\neg S\lor T)} \arrow[dl, dash, "\phi(\neg R)"'] \arrow[dr, dash, "\phi(R)"] & \\
 & {\cancel{()\land(S)\land(T\lor S)\land(\neg S\lor T)}} & & {(T)\land(S)\land(\neg S\lor T)}
\end{tikzcd}$$

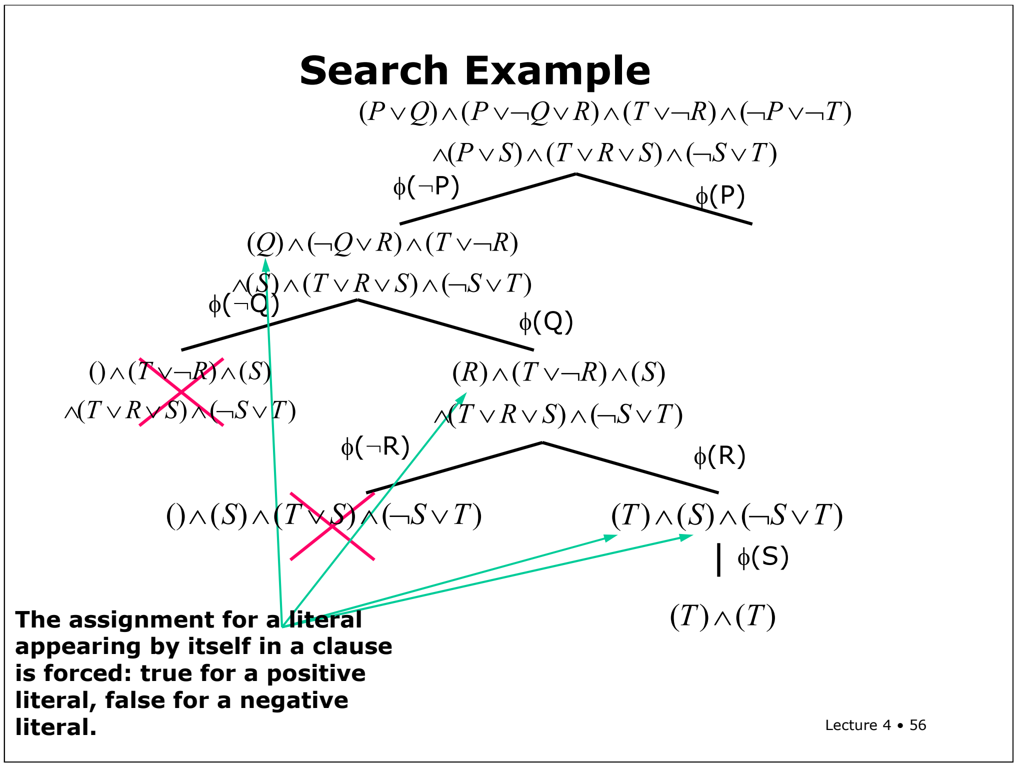
So, we go back up, assign R to be **t** and simplify.

Search Example


$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$
$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$\phi(\neg P)$
$\phi(P)$
$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$
$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$\phi(\neg Q)$
$\phi(Q)$
$() \wedge (T \vee \neg R) \wedge (S)$
$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$(R) \wedge (T \vee \neg R) \wedge (S)$
$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$\phi(\neg R)$
$\phi(R)$
$() \wedge (S) \wedge (T \vee S) \wedge (\neg S \vee T)$
$(T) \wedge (S) \wedge (\neg S \vee T)$

The assignment for a literal appearing by itself in a clause is forced: true for a positive literal, false for a negative literal.

At this point, we can see a way to be smarter about choosing an assignment to try first. As we saw with Q and with R, if a literal appears by itself in a clause, its assignment is forced: true for a positive literal, false for a negative literal. If you try the negation of that assignment, you'll reach a dead end and have to back up.



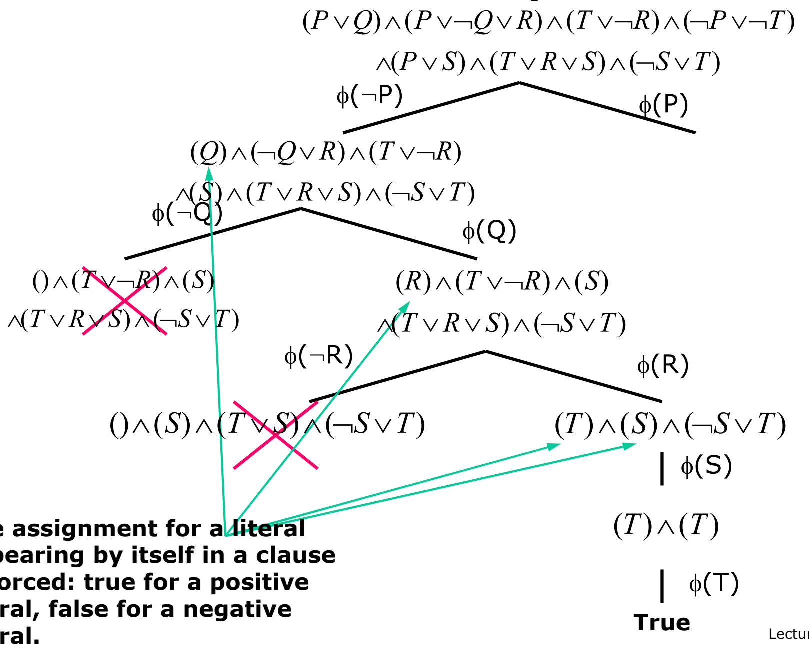
Search Example

$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$
$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
φ(¬P)
φ(P)
$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$
$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
φ(¬Q)
φ(Q)
$() \wedge (T \vee \neg R) \wedge (S)$
$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$(R) \wedge (T \vee \neg R) \wedge (S)$
$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
φ(¬R)
φ(R)
$() \wedge (S) \wedge (T \vee S) \wedge (\neg S \vee T)$
$(T) \wedge (S) \wedge (\neg S \vee T)$
φ(S)
$(T) \wedge (T)$

The assignment for a literal appearing by itself in a clause is forced: true for a positive literal, false for a negative literal.

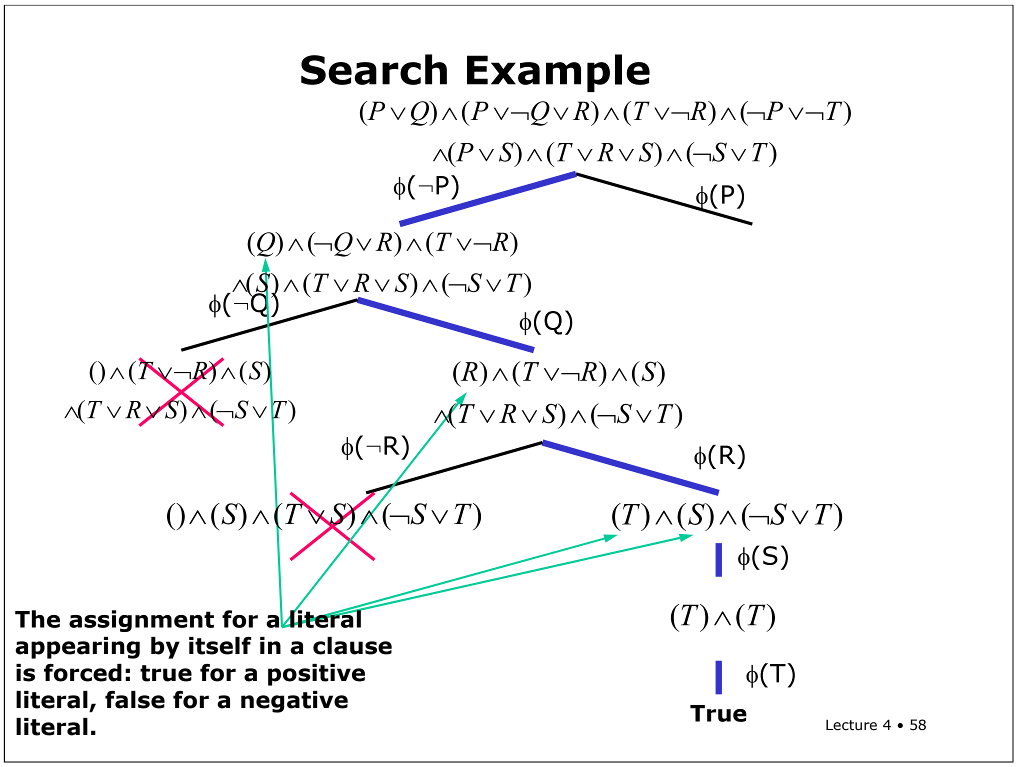
So, we'll be smarter and try assigning S to **t**, which gives us a simple sentence.

Search Example


$(P \lor Q) \land (P \lor \neg Q \lor R) \land (T \lor \neg R) \land (\neg P \lor \neg T)$
$\land (P \lor S) \land (T \lor R \lor S) \land (\neg S \lor T)$
$\phi(\neg P)$
$\phi(P)$
$(Q) \land (\neg Q \lor R) \land (T \lor \neg R)$
$\land (S) \land (T \lor R \lor S) \land (\neg S \lor T)$
$\phi(\neg Q)$
$\phi(Q)$
$() \land (T \lor \neg R) \land (S)$
$\land (T \lor R \lor S) \land (\neg S \lor T)$
$(R) \land (T \lor \neg R) \land (S)$
$\land (T \lor R \lor S) \land (\neg S \lor T)$
$\phi(\neg R)$
$\phi(R)$
$() \land (S) \land (T \lor S) \land (\neg S \lor T)$
$(T) \land (S) \land (\neg S \lor T)$
$\phi(S)$
$(T) \land (T)$
$\phi(T)$
True

The assignment for a literal appearing by itself in a clause is forced: true for a positive literal, false for a negative literal.

Again, we’re forced to assign T to **t**, yielding a final result of “True”.

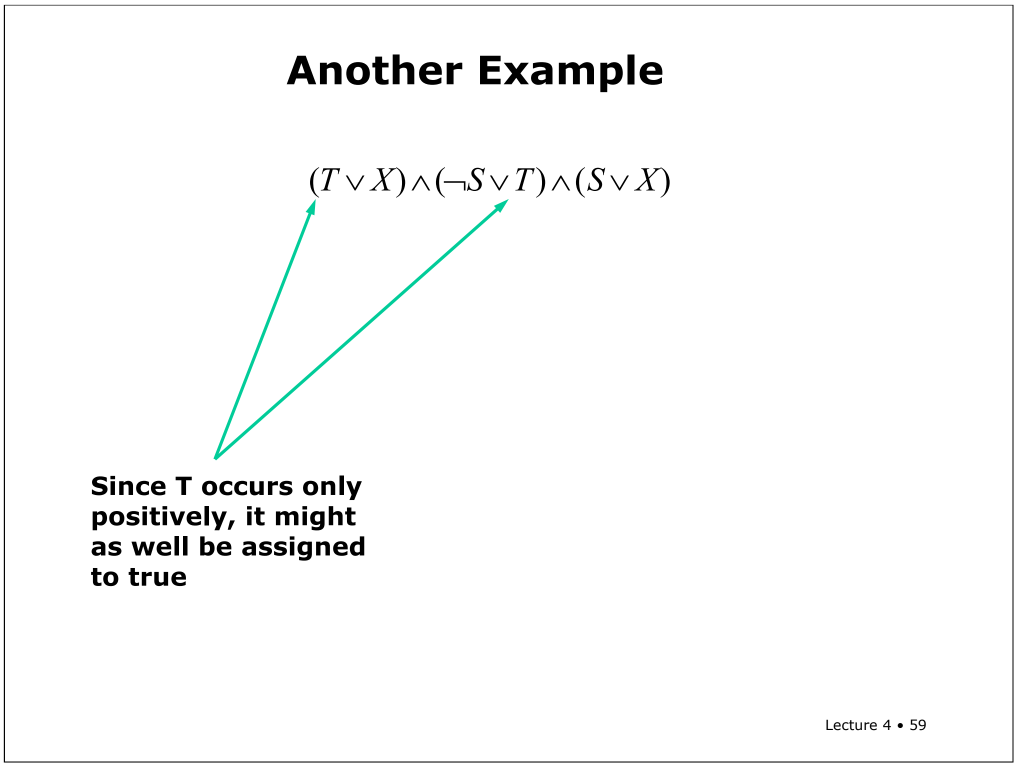


Search Example

$(P \vee Q) \wedge (P \vee \neg Q \vee R) \wedge (T \vee \neg R) \wedge (\neg P \vee \neg T)$
$\wedge (P \vee S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$\phi(\neg P)$
$\phi(P)$
$(Q) \wedge (\neg Q \vee R) \wedge (T \vee \neg R)$
$\wedge (S) \wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$\phi(\neg Q)$
$\phi(Q)$
$() \wedge (T \vee \neg R) \wedge (S)$
$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$(R) \wedge (T \vee \neg R) \wedge (S)$
$\wedge (T \vee R \vee S) \wedge (\neg S \vee T)$
$\phi(\neg R)$
$\phi(R)$
$() \wedge (S) \wedge (T \vee S) \wedge (\neg S \vee T)$
$(T) \wedge (S) \wedge (\neg S \vee T)$
$\phi(S)$
$(T) \wedge (T)$
$\phi(T)$
True

The assignment for a literal appearing by itself in a clause is forced: true for a positive literal, false for a negative literal.

Now, this path through the tree represents the assignment: P false, Q true, R true, S true, and T true. And because , given those assignments, the sentence simplified to “true”, that is a satisfying assignment.



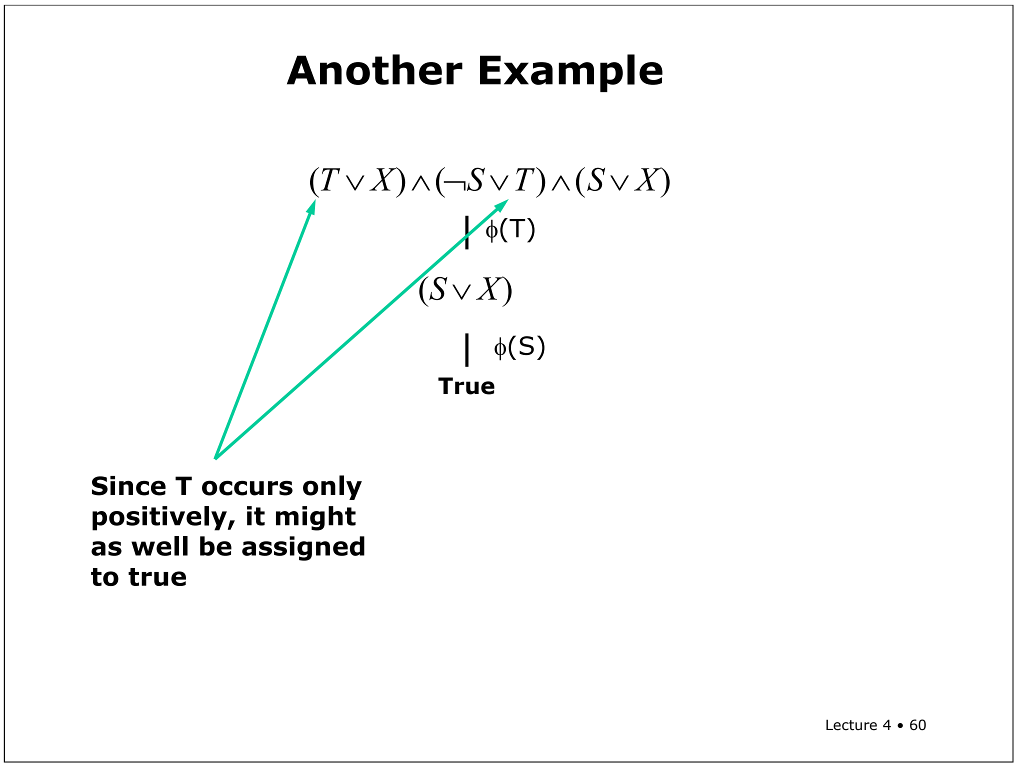
Another Example

$$(T \vee X) \wedge (\neg S \vee T) \wedge (S \vee X)$$

Since T occurs only positively, it might as well be assigned to true

Here's one more small example to illustrate another way to make searching for a satisfying assignment more directed. Consider this sentence. The variable T occurs only positively. Although we don't **have** to make it true, we don't lose anything by doing so.

So, if you have a sentence in which a variable occurs always positively, you should just set it to true. If a variable occurs always negatively, you should just set it to false.



Another Example

$$(T \vee X) \wedge (\neg S \vee T) \wedge (S \vee X)$$

$$\Big| \ \phi(T)$$

$$(S \vee X)$$

$$\Big| \ \phi(S)$$

True

Since T occurs only positively, it might as well be assigned to true

Lecture 4 • 60

Once we assign T to true, all of the clauses containing it drop out, and we're left with a very simple problem to finish.

DPLL(ϕ)

All the insight we gained from the previous example can be condensed into an algorithm. It's called DPLL, which stands for the names of the inventors of the algorithm (Davis, Putnam, Logeman and Loveland). It's very well described in a paper by Cook, which we have linked into the syllabus (the Cook paper also describes the GSAT and WalkSAT algorithms that we'll talk about later).

The DPLL algorithm takes a CNF sentence phi as input, and returns true if it is satisfiable and false otherwise. It works recursively.

DPLL(ϕ)

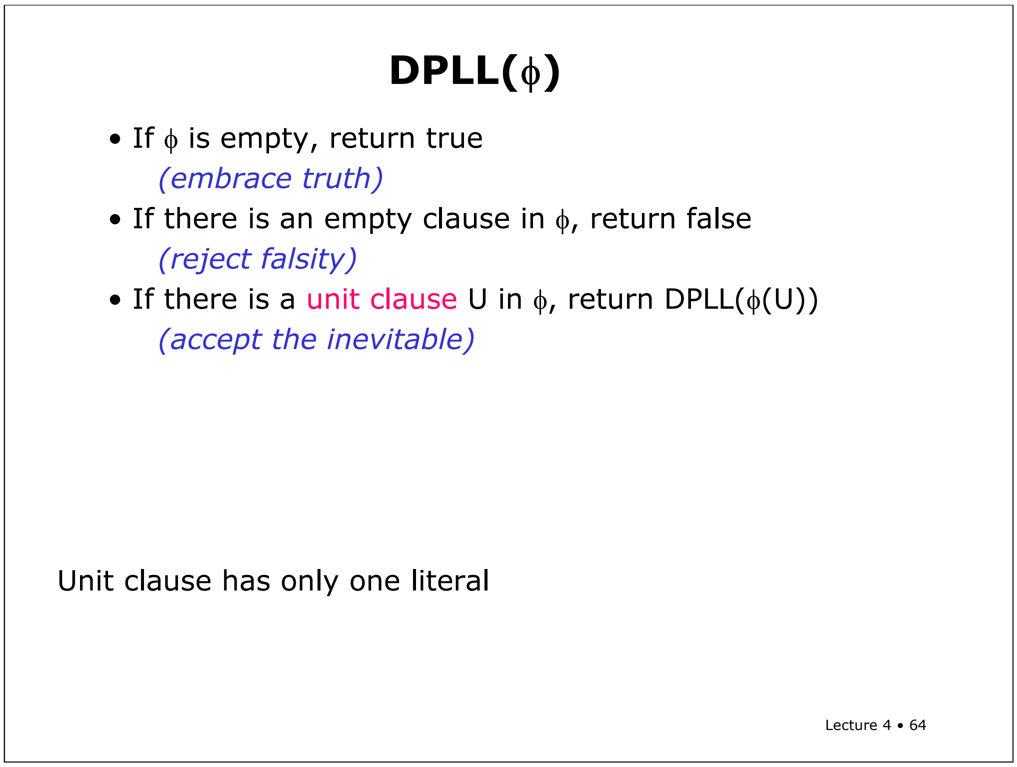
- If ϕ is empty, return true
  (embrace truth)

If phi is empty, then return true. Our work is done!

DPLL(ϕ)

- If ϕ is empty, return true
  (embrace truth)
- If there is an empty clause in ϕ, return false
  (reject falsity)

If there is an empty clause in phi, then return false. Remember than an empty clause is false, and once we have one false clause, the whole sentence is false.

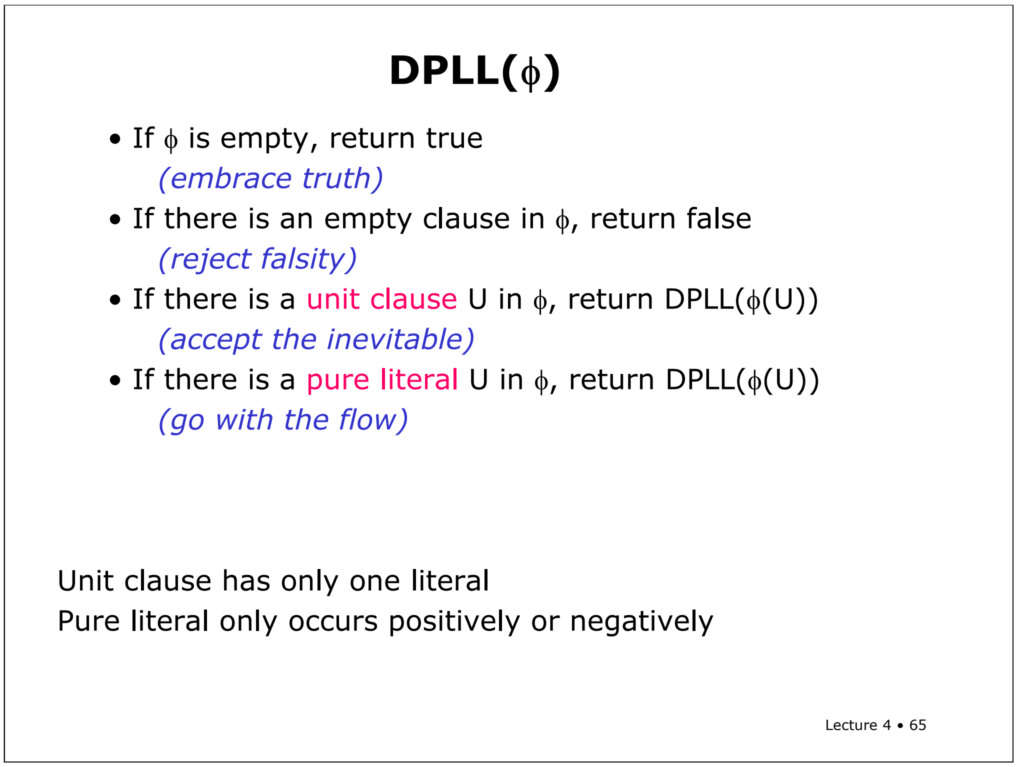


DPLL(ϕ)

- If ϕ is empty, return true
  (embrace truth)
- If there is an empty clause in ϕ, return false
  (reject falsity)
- If there is a unit clause U in ϕ, return DPLL(ϕ(U))
  (accept the inevitable)

Unit clause has only one literal

If there is a unit clause containing literal U in phi (remember, a unit clause has only one literal, and so its assignment is forced), then assign the literal, simplify, and call DPLL recursively on the simplified sentence.



DPLL(ϕ)

- If ϕ is empty, return true
  (embrace truth)
- If there is an empty clause in ϕ, return false
  (reject falsity)
- If there is a unit clause U in ϕ, return DPLL(ϕ(U))
  (accept the inevitable)
- If there is a pure literal U in ϕ, return DPLL(ϕ(U))
  (go with the flow)

Unit clause has only one literal
Pure literal only occurs positively or negatively

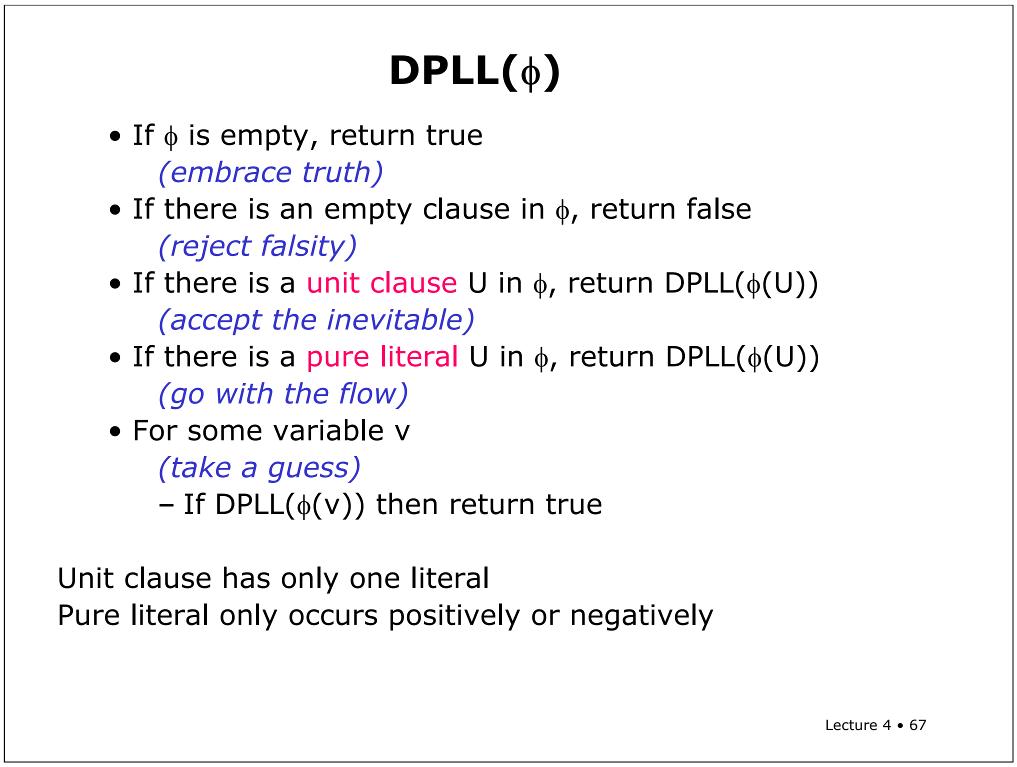
If there is a pure literal U in phi (that is, the variable in the literal U always occurs either positively or negatively in phi), then assign the literal, simplify, and call DPLL recursively on the simplified sentence.

DPLL(ϕ)

- If ϕ is empty, return true
  (embrace truth)
- If there is an empty clause in ϕ, return false
  (reject falsity)
- If there is a unit clause U in ϕ, return DPLL(ϕ(U))
  (accept the inevitable)
- If there is a pure literal U in ϕ, return DPLL(ϕ(U))
  (go with the flow)
- For some variable v
  (take a guess)

Unit clause has only one literal
Pure literal only occurs positively or negatively

If none of the previous conditions hold, then we have to take a guess. Choose any variable v occurring in phi.

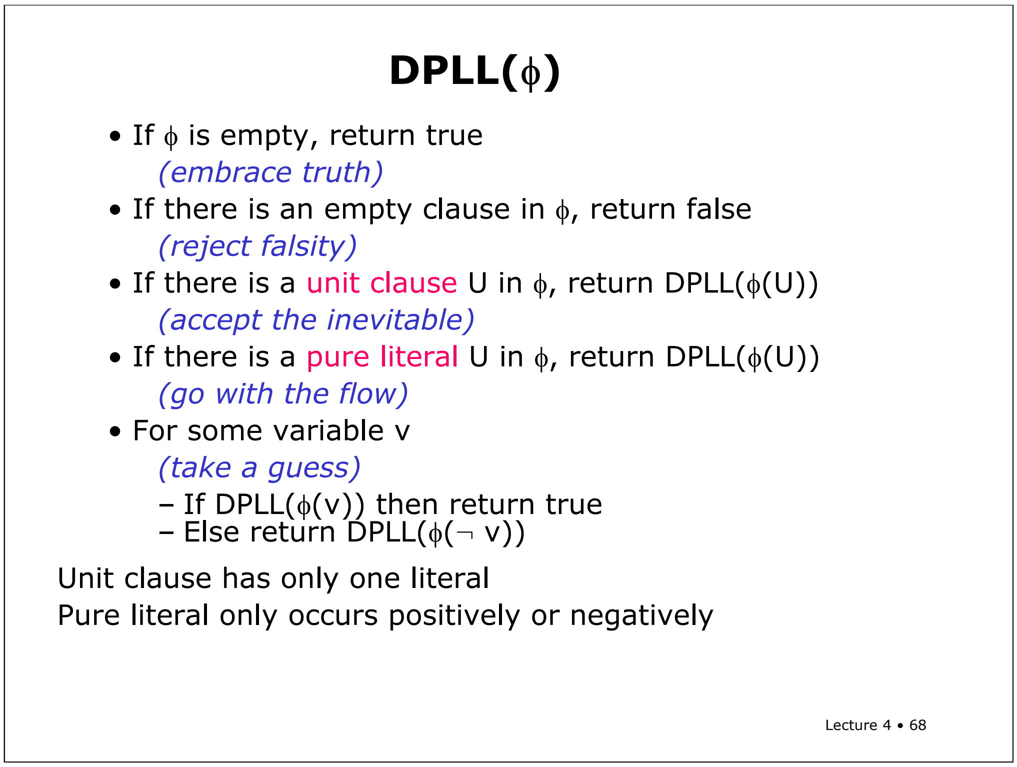


DPLL(ϕ)

- If ϕ is empty, return true
  (embrace truth)
- If there is an empty clause in ϕ, return false
  (reject falsity)
- If there is a unit clause U in ϕ, return DPLL(ϕ(U))
  (accept the inevitable)
- If there is a pure literal U in ϕ, return DPLL(ϕ(U))
  (go with the flow)
- For some variable v
  (take a guess)
  – If DPLL(ϕ(v)) then return true

Unit clause has only one literal
Pure literal only occurs positively or negatively

Try assigning it to be true: simplify and call DPLL recursively on the simplified sentence. If it returns true, then the sentence is satisfiable, and we can return true as well.

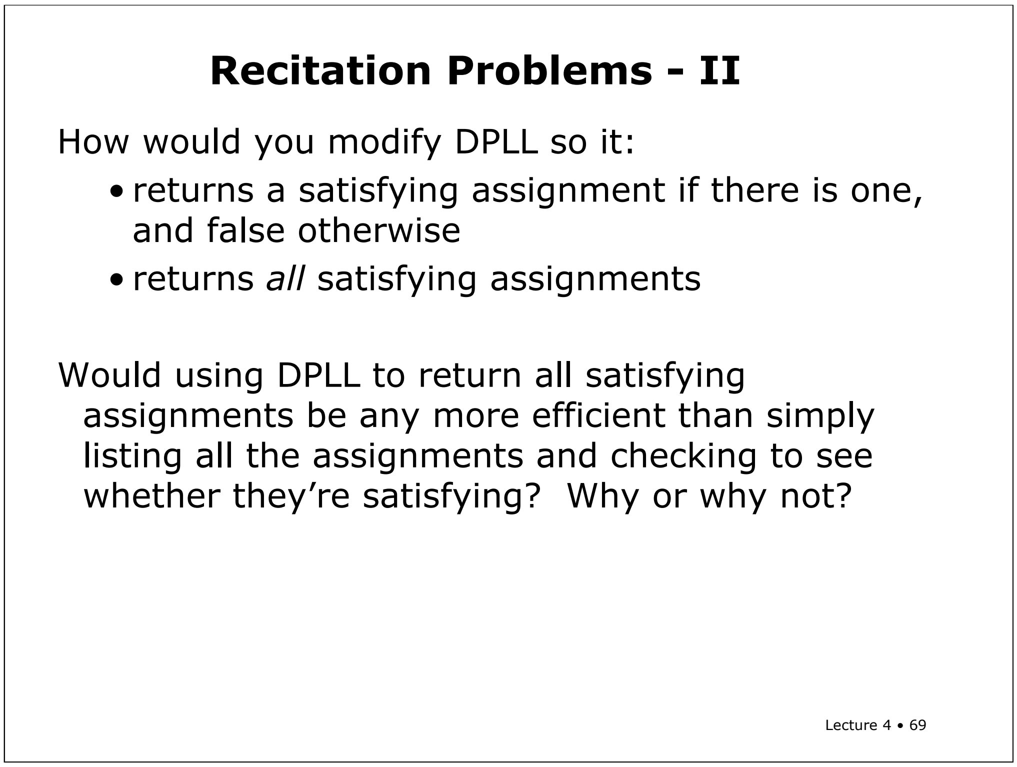


DPLL(ϕ)

- If ϕ is empty, return true
  (embrace truth)
- If there is an empty clause in ϕ, return false
  (reject falsity)
- If there is a unit clause U in ϕ, return DPLL(ϕ(U))
  (accept the inevitable)
- If there is a pure literal U in ϕ, return DPLL(ϕ(U))
  (go with the flow)
- For some variable v
  (take a guess)
  - If DPLL(ϕ(v)) then return true
  - Else return DPLL(ϕ($\neg$ v))

Unit clause has only one literal
Pure literal only occurs positively or negatively

If not, then try assigning v to be false, simplify, and call DPLL recursively.



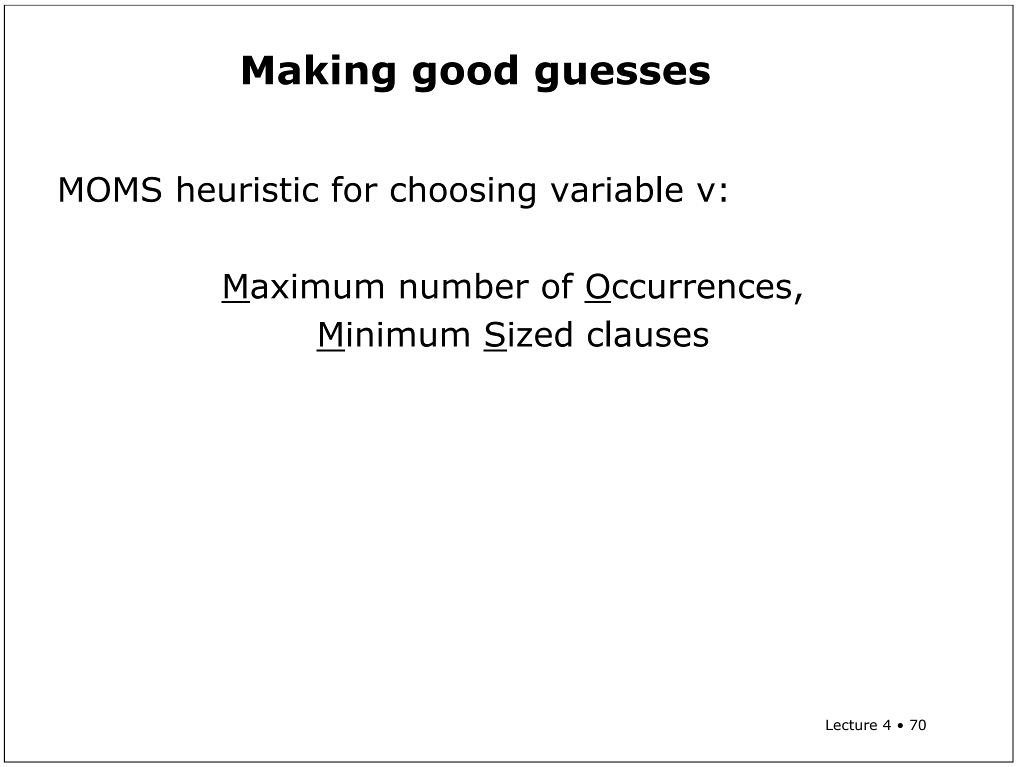
Recitation Problems - II

How would you modify DPLL so it:

- returns a satisfying assignment if there is one, and false otherwise
- returns *all* satisfying assignments

Would using DPLL to return all satisfying assignments be any more efficient than simply listing all the assignments and checking to see whether they're satisfying? Why or why not?

Please do these problems before going on with the lecture.

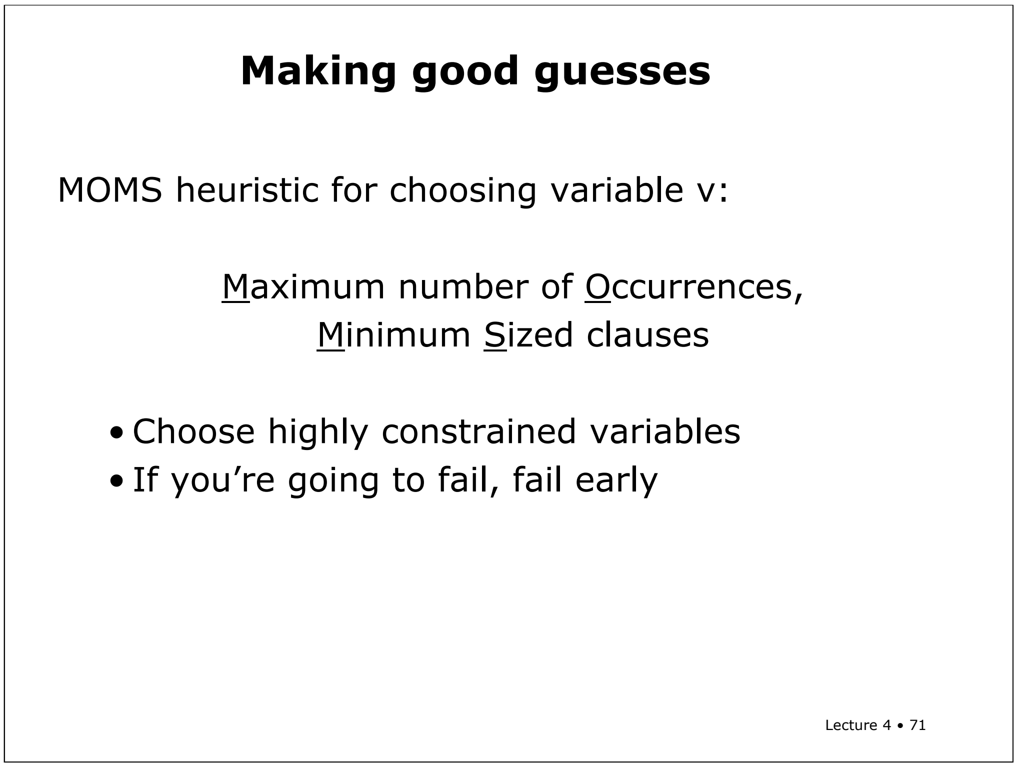


Making good guesses

MOMS heuristic for choosing variable v:

Maximum number of Occurrences,
Minimum Sized clauses

What's a good way to choose the variable to assign? There are lots of different heuristics. One that seems to work out reasonably well in practice is the "MOMS" heuristic: choose the variable that has the maximum number of occurrences in minimum sized clauses.



Making good guesses

MOMS heuristic for choosing variable v:

Maximum number of Occurrences,
Minimum Sized clauses

- Choose highly constrained variables
- If you're going to fail, fail early

The idea is that such variables are highly constrained. If you are going to fail, you'd like to fail early (that is, if you've made some bad assignments that will lead to a false Phi, you might as well know that before you make a lot of other assignments and grow out a huge tree).

So, intuitively, assigning values to the variables that are most constrained is more likely to reveal problems soon.

Soundness and Completeness

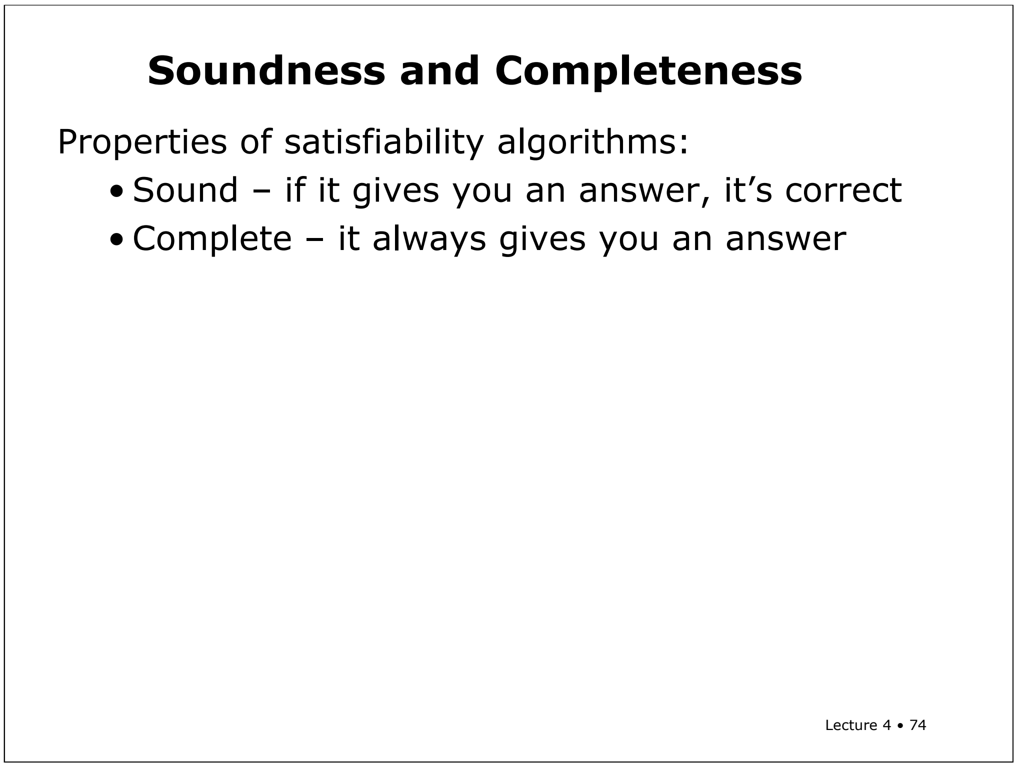
The correctness of a variety of algorithms can be described in terms of soundness and completeness

Soundness and Completeness

Properties of satisfiability algorithms:

- Sound – if it gives you an answer, it's correct

An algorithm is sound if, whenever it gives you an answer, it's correct.

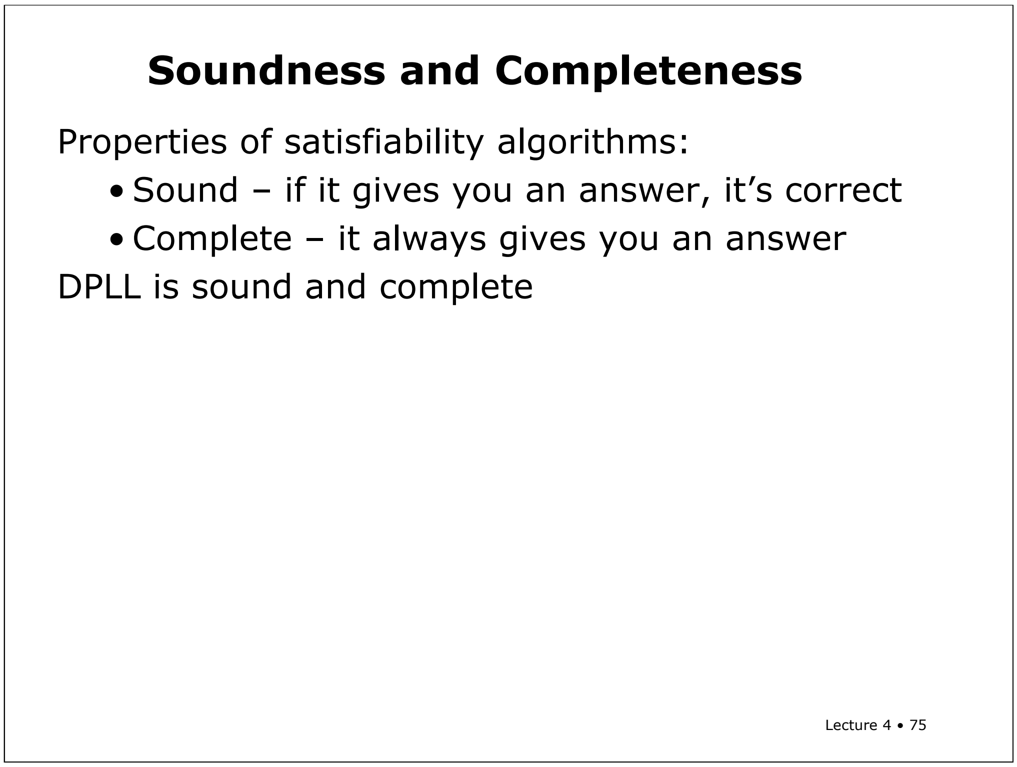


Soundness and Completeness

Properties of satisfiability algorithms:

- Sound – if it gives you an answer, it's correct
- Complete – it always gives you an answer

An algorithm is complete if it always gives you an answer.



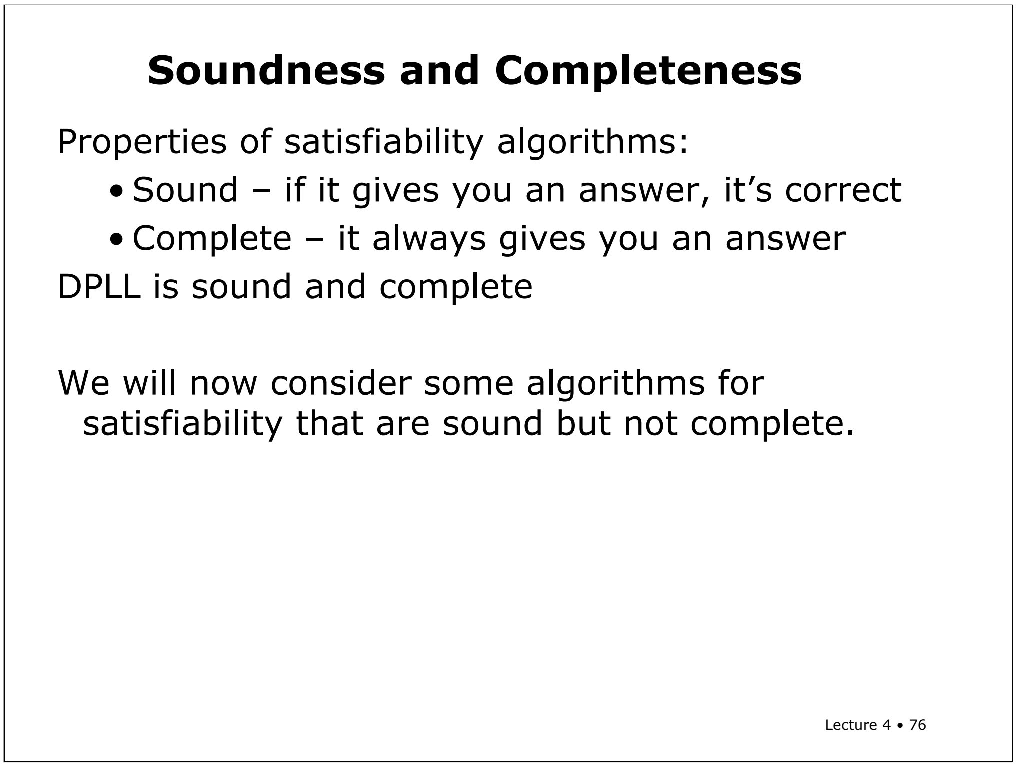
Soundness and Completeness

Properties of satisfiability algorithms:

- Sound – if it gives you an answer, it's correct
- Complete – it always gives you an answer

DPLL is sound and complete

The DPLL algorithm, being a systematic search algorithm that only skips assignments that are **sure** to be unsatisfactory, is sound and complete. But sometimes it can be slow!



Soundness and Completeness

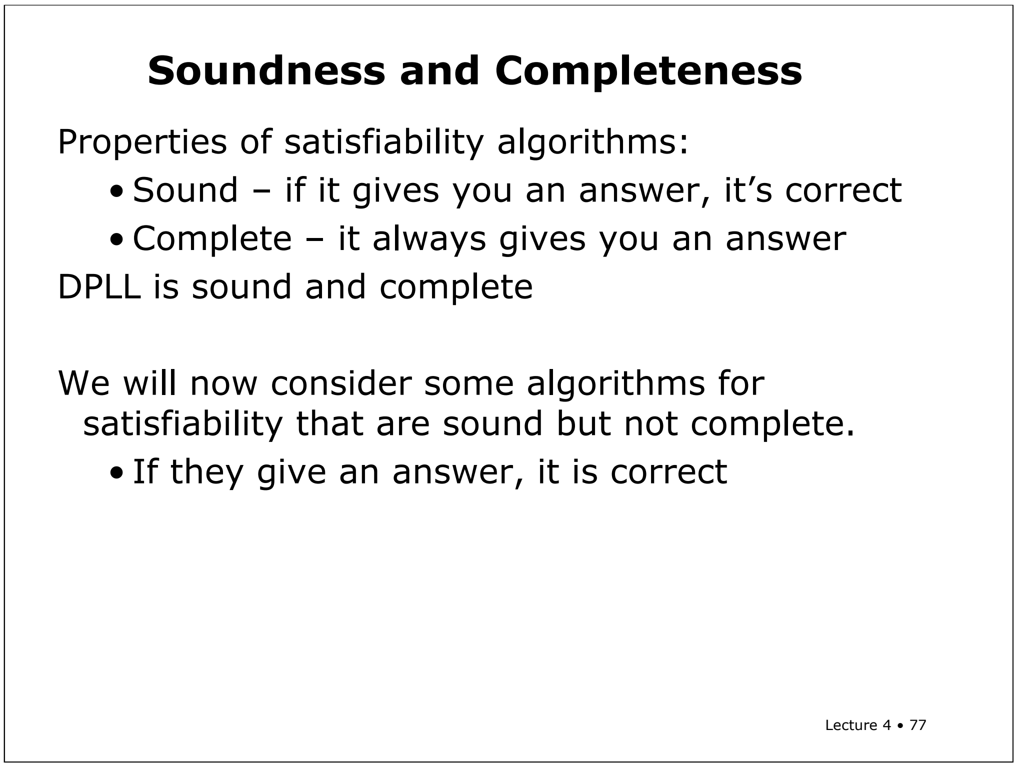
Properties of satisfiability algorithms:

- Sound – if it gives you an answer, it’s correct
- Complete – it always gives you an answer

DPLL is sound and complete

We will now consider some algorithms for satisfiability that are sound but not complete.

Now we’re going to consider a couple of algorithms for solving satisfiability problems that have been found to be very effective in practice. They are sound, but not complete.



Soundness and Completeness

Properties of satisfiability algorithms:

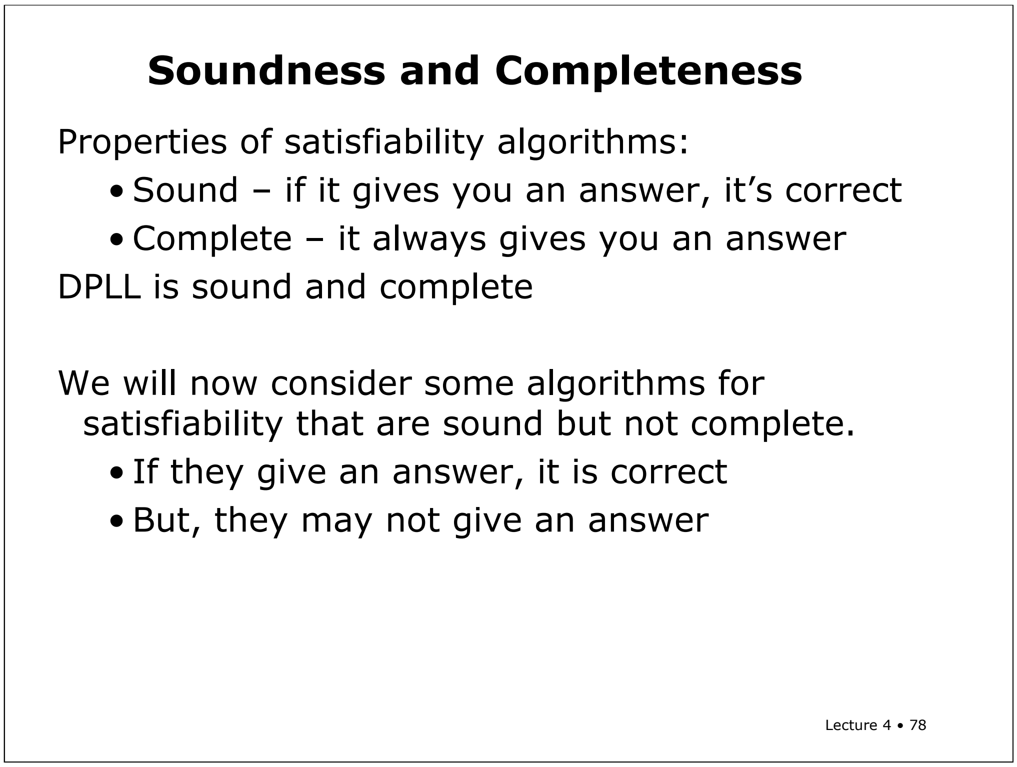
- Sound – if it gives you an answer, it’s correct
- Complete – it always gives you an answer

DPLL is sound and complete

We will now consider some algorithms for satisfiability that are sound but not complete.

- If they give an answer, it is correct

So, if they give an answer, it’s correct.



Soundness and Completeness

Properties of satisfiability algorithms:

- Sound – if it gives you an answer, it's correct
- Complete – it always gives you an answer

DPLL is sound and complete

We will now consider some algorithms for satisfiability that are sound but not complete.

- If they give an answer, it is correct
- But, they may not give an answer

But they may not always give an answer

Soundness and Completeness

Properties of satisfiability algorithms:

- Sound – if it gives you an answer, it's correct
- Complete – it always gives you an answer

DPLL is sound and complete

We will now consider some algorithms for satisfiability that are sound but not complete.

- If they give an answer, it is correct
- But, they may not give an answer
- They may be faster than any complete algorithm

And, on average, they tend to be much faster than any complete algorithm.

GSAT

Hill climbing in the space of total assignments

The GSAT algorithm is an example of an ‘iterative improvement’ algorithm, such as those discussed in section 4.4 of the book. It does hill-climbing in the space of complete assignments, with random restarts.

GSAT

Hill climbing in the space of total assignments

- Starts with random assignment for all variables
- Moves to "neighboring" assignment with least cost (flip a single bit)

We start with a random assignment to the variables, and then move to the "neighboring" assignment with the least cost. The assignments that are neighbors of the current assignment are those that can be reached by "flipping" a single bit of the current assignment. "Flipping" a bit is changing the assignment of one variable from true to false, or from false to true.

GSAT

Hill climbing in the space of total assignments

- Starts with random assignment for all variables
- Moves to “neighboring” assignment with least cost (flip a single bit)

Cost(assignment) = number of unsatisfied clauses

The cost of an assignment is the number of clauses in the sentence that are unsatisfied under the assignment.

GSAT

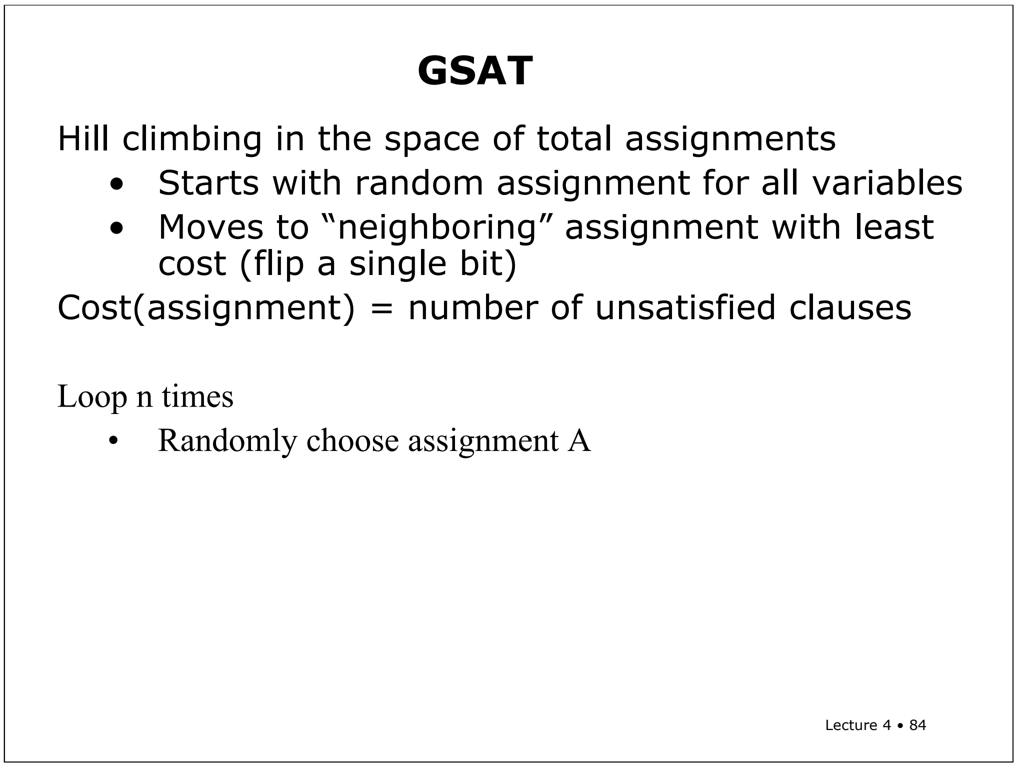
Hill climbing in the space of total assignments

- Starts with random assignment for all variables
- Moves to "neighboring" assignment with least cost (flip a single bit)

Cost(assignment) = number of unsatisfied clauses

```
Loop n times
```

Okay. Here's the algorithm in pseudocode. We're going to do **n** different hill-climbing runs,



GSAT

Hill climbing in the space of total assignments

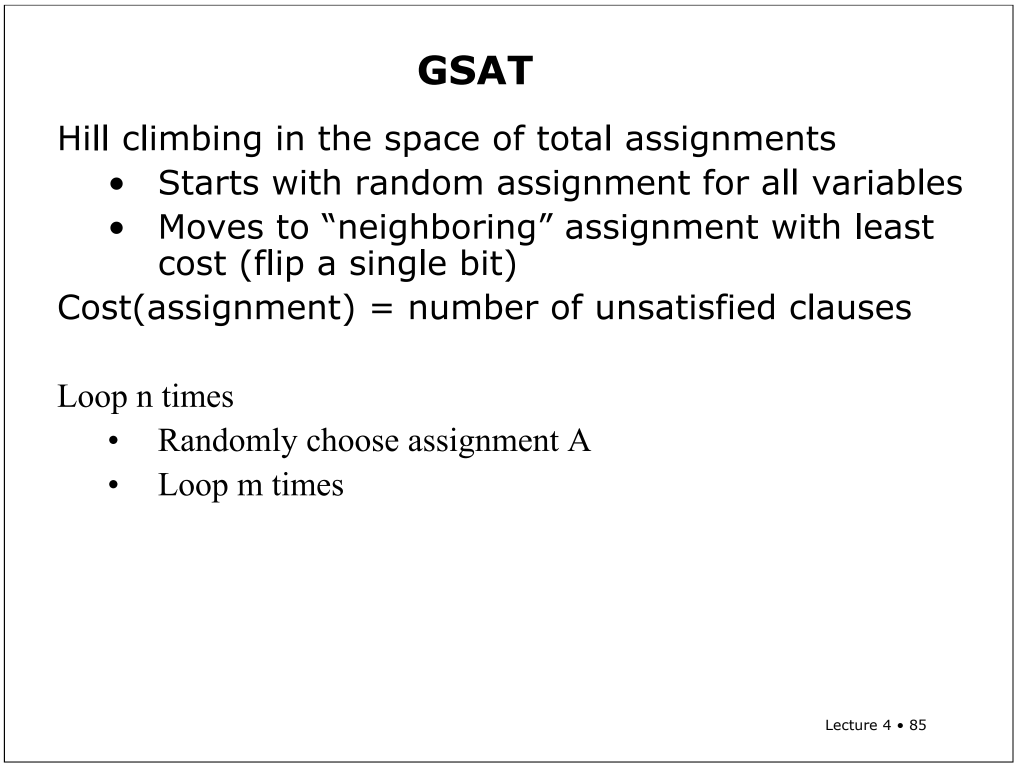
- Starts with random assignment for all variables
- Moves to “neighboring” assignment with least cost (flip a single bit)

Cost(assignment) = number of unsatisfied clauses

Loop n times

- Randomly choose assignment A

starting from different randomly chosen initial assignments.



GSAT

Hill climbing in the space of total assignments

- Starts with random assignment for all variables
- Moves to “neighboring” assignment with least cost (flip a single bit)

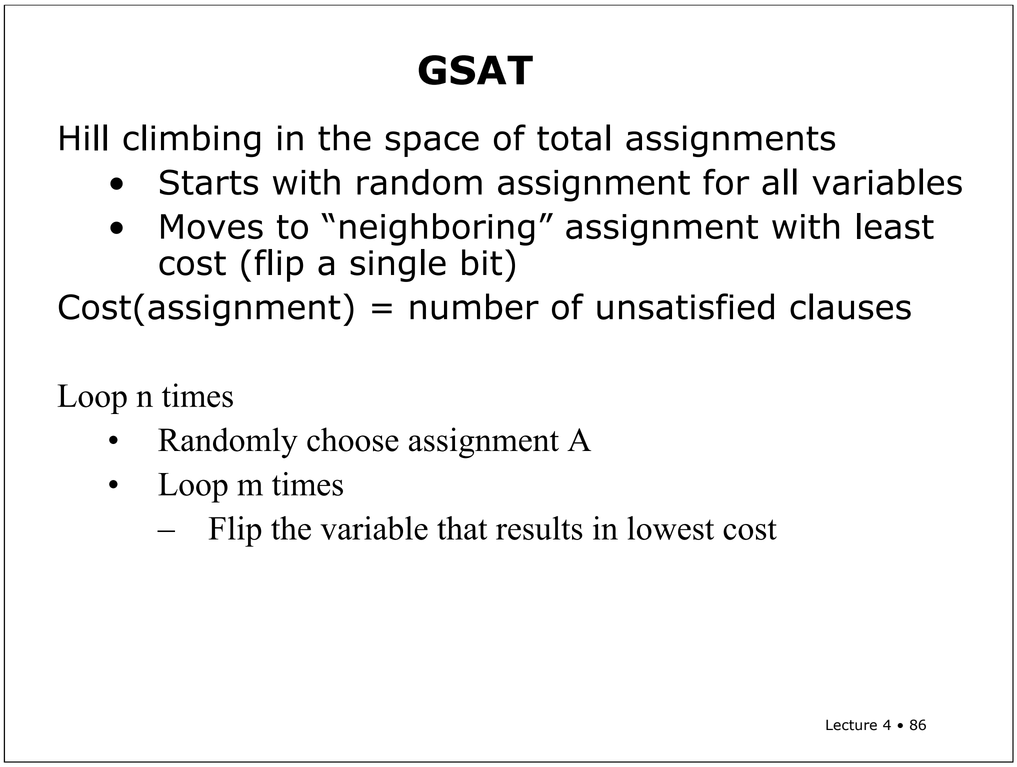
Cost(assignment) = number of unsatisfied clauses

Loop n times

- Randomly choose assignment A
- Loop m times

Lecture 4 • 85

Now, we loop for **m** steps, we consider the cost of all the neighboring assignments (those with a single variable assigned differently), and we



GSAT

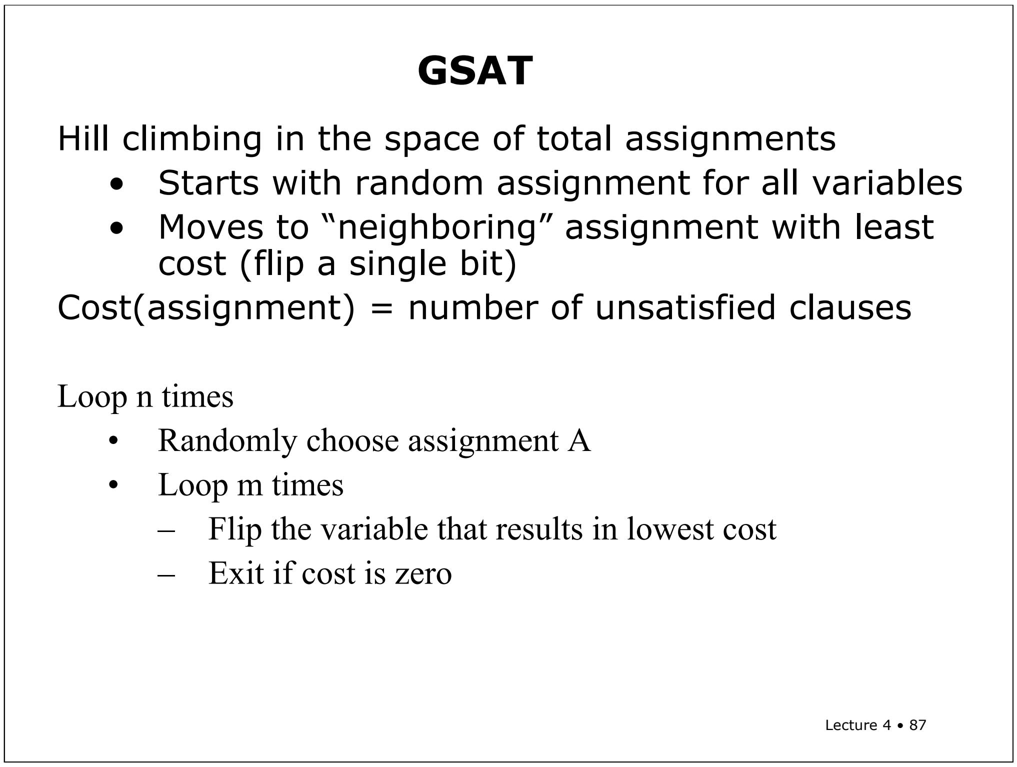
Hill climbing in the space of total assignments

- Starts with random assignment for all variables
- Moves to "neighboring" assignment with least cost (flip a single bit)

Cost(assignment) = number of unsatisfied clauses

```
Loop n times
  • Randomly choose assignment A
  • Loop m times
      – Flip the variable that results in lowest cost
```

flip the variable that results in the lowest cost (even if that cost is higher than the cost of the current assignment! This may keep us walking out of some local minima).



GSAT

Hill climbing in the space of total assignments

- Starts with random assignment for all variables
- Moves to "neighboring" assignment with least cost (flip a single bit)

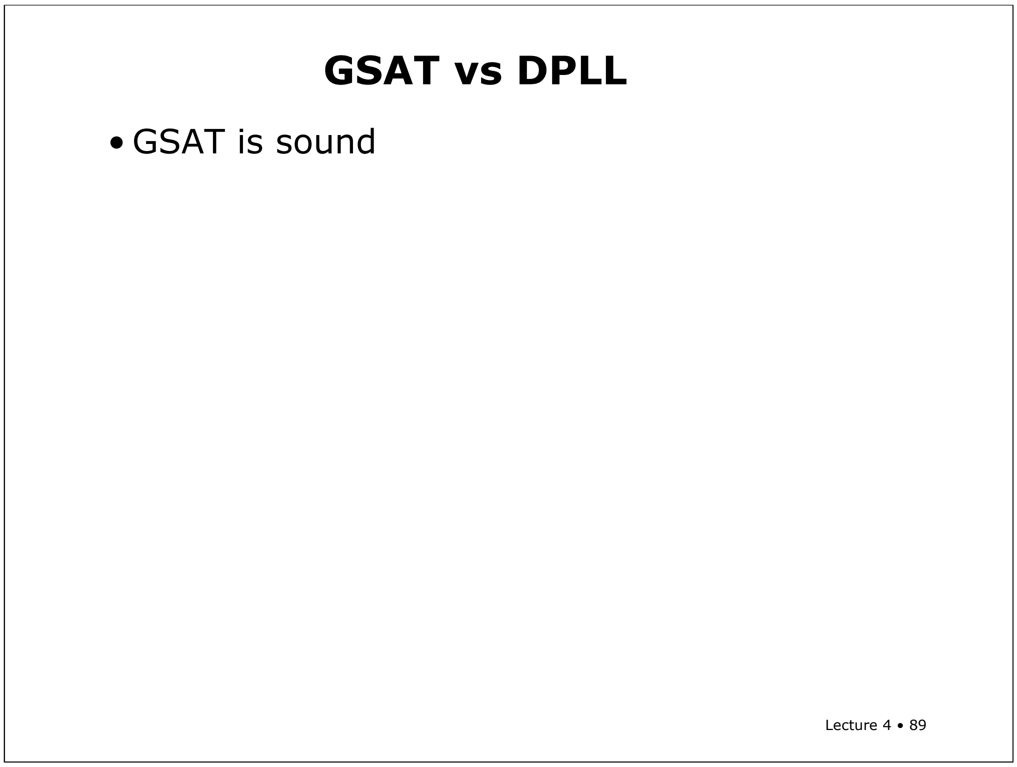
Cost(assignment) = number of unsatisfied clauses

```
Loop n times
    • Randomly choose assignment A
    • Loop m times
        – Flip the variable that results in lowest cost
        – Exit if cost is zero
```

If the cost is zero, we've found a satisfying assignment. Yay! Exit.

GSAT vs DPLL

So, how does GSAT compare to DPLL?



GSAT vs DPLL

- GSAT is sound

GSAT is sound. If it gives you an answer, it's correct.

GSAT vs DPLL

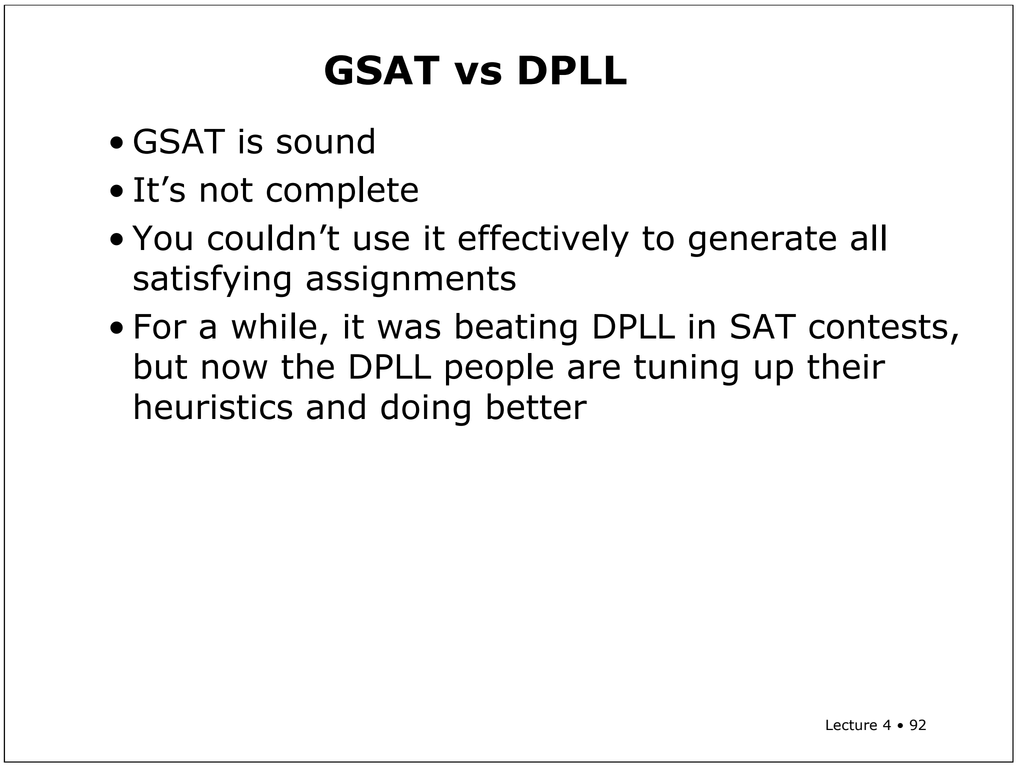
- GSAT is sound
- It’s not complete

GSAT is not complete. No matter how long you give it to wander around in the space of assignments, or how many times you restart it, there’s always a chance it will miss an existing solution.

GSAT vs DPLL

- GSAT is sound
- It’s not complete
- You couldn’t use it effectively to generate all satisfying assignments

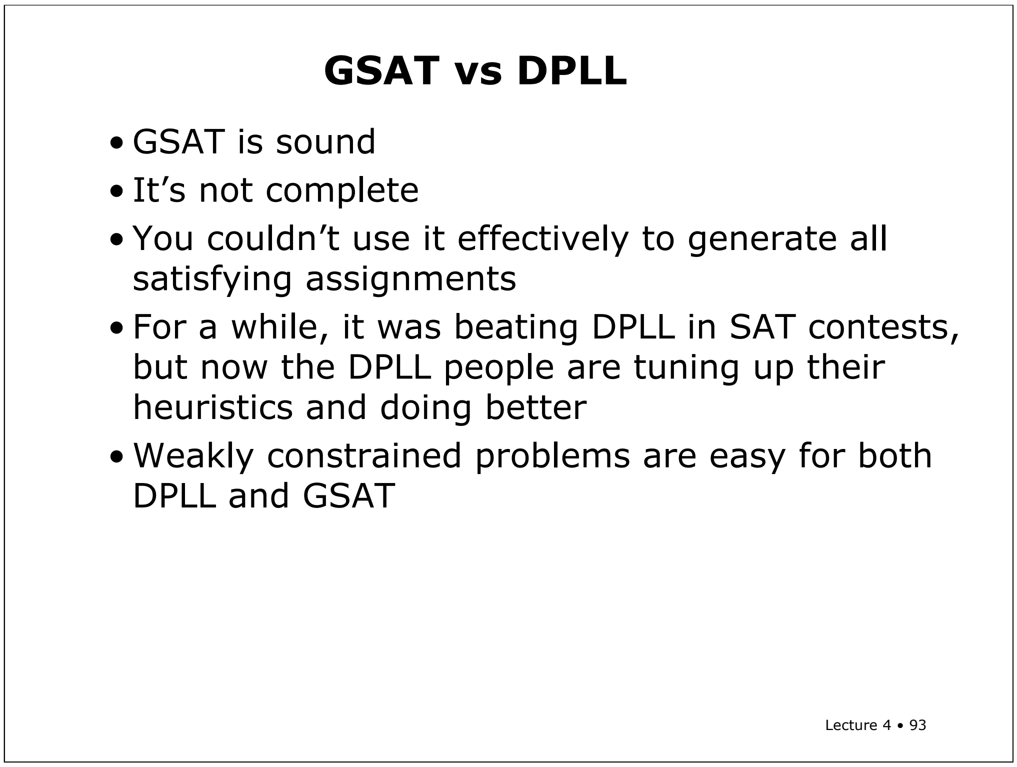
It’s particularly unhelpful if you want to enumerate all the satisfying assignments; since it’s not systematic, you could never know whether you had gotten all of them.



GSAT vs DPLL

- GSAT is sound
- It’s not complete
- You couldn’t use it effectively to generate all satisfying assignments
- For a while, it was beating DPLL in SAT contests, but now the DPLL people are tuning up their heuristics and doing better

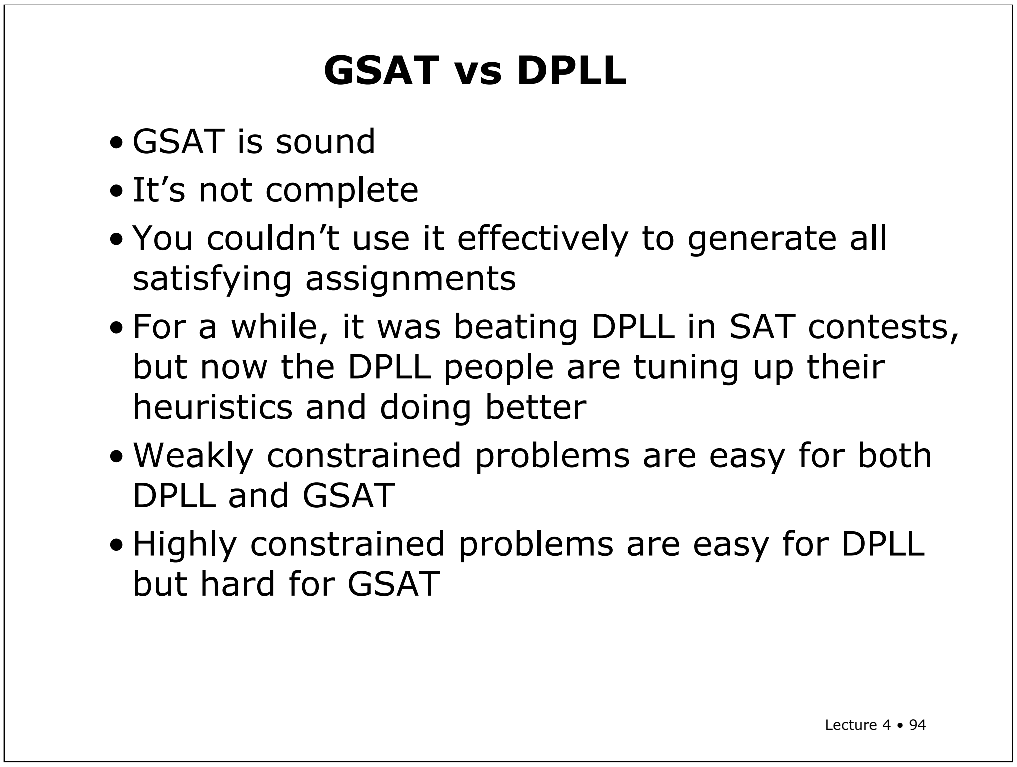
For a while, GSAT was doing hugely better than DPLL in contests. But now people are adding better heuristics to DPLL and it is starting to do better than GSAT.



GSAT vs DPLL

- GSAT is sound
- It’s not complete
- You couldn’t use it effectively to generate all satisfying assignments
- For a while, it was beating DPLL in SAT contests, but now the DPLL people are tuning up their heuristics and doing better
- Weakly constrained problems are easy for both DPLL and GSAT

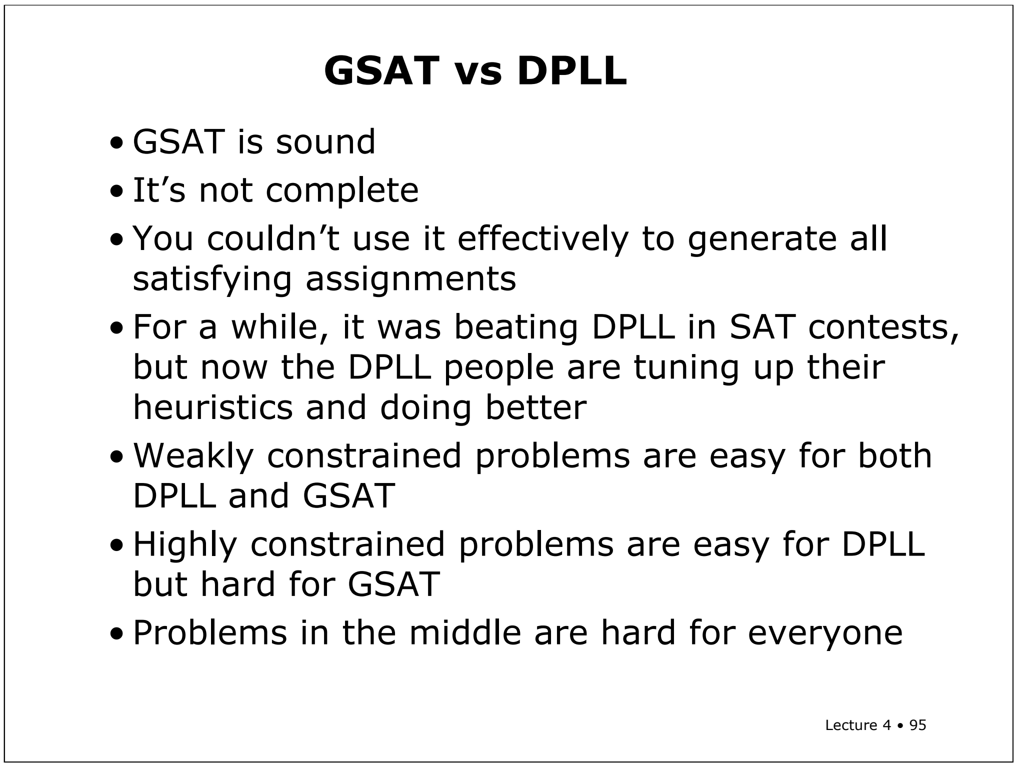
The Cook paper has an interesting discussion of which kinds of problems are easy and hard. Problems that are weakly constrained have many solutions. They’re pretty easy for both DPLL and GSAT to solve.



GSAT vs DPLL

- GSAT is sound
- It’s not complete
- You couldn’t use it effectively to generate all satisfying assignments
- For a while, it was beating DPLL in SAT contests, but now the DPLL people are tuning up their heuristics and doing better
- Weakly constrained problems are easy for both DPLL and GSAT
- Highly constrained problems are easy for DPLL but hard for GSAT

Highly constrained problems, have only one, or very few solutions. They’re easy for DPLL, because the simplification process will tend to quickly realize that a particular partial assignment has no possible satisfying extensions, and cut off huge chucks of the search space at once. For GSAT, on the other hand, it’s like looking for a needle in a haystack.



GSAT vs DPLL

- GSAT is sound
- It’s not complete
- You couldn’t use it effectively to generate all satisfying assignments
- For a while, it was beating DPLL in SAT contests, but now the DPLL people are tuning up their heuristics and doing better
- Weakly constrained problems are easy for both DPLL and GSAT
- Highly constrained problems are easy for DPLL but hard for GSAT
- Problems in the middle are hard for everyone

There is a class of problems that are neither weakly nor highly constrained. They’re very hard for all known algorithms.

WALKSAT

Like GSAT with additional “noise”

Here’s another algorithm that’s sort of like GSAT, called WalkSAT. It also moves through the space of complete assignments, but with a good deal more randomness than GSAT.

WALKSAT

Like GSAT with additional "noise"

```
Loop n times
    • Randomly choose assignment A
```

It has the same external structure as GSAT. There's an outer loop of **n** restarts at randomly chosen assignments.

WALKSAT

Like GSAT with additional "noise"

Loop n times
- Randomly choose assignment A
- Loop m times

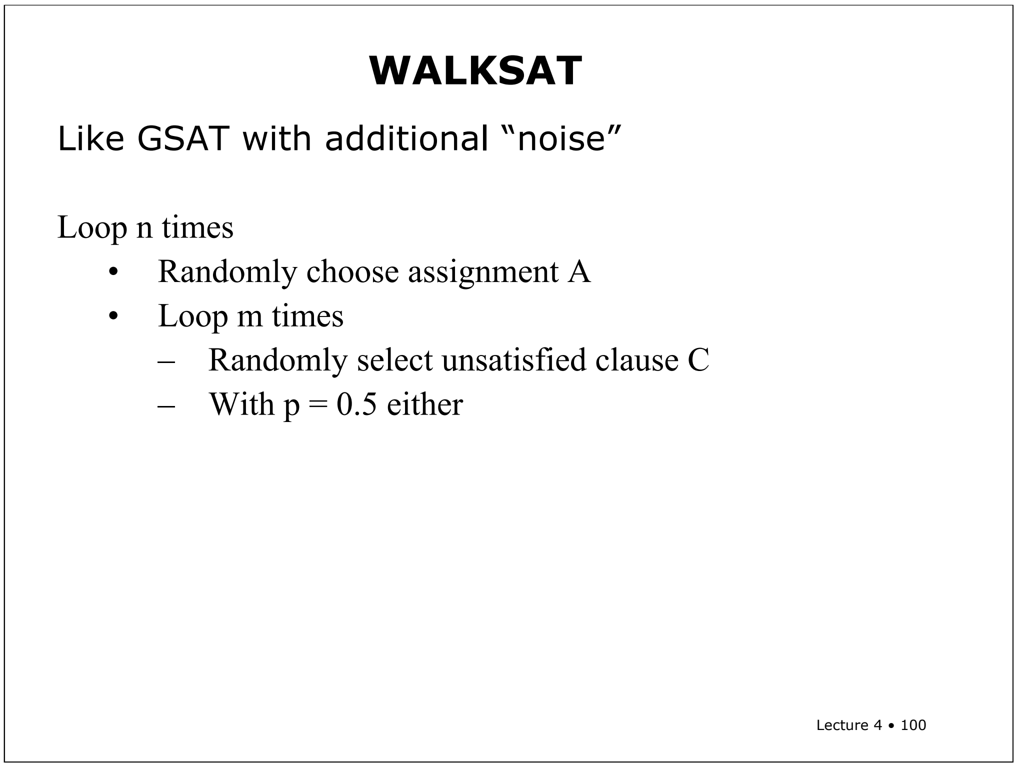
Then, we take **m** steps, but the steps are somewhat different.

WALKSAT

Like GSAT with additional “noise”

```
Loop n times
    • Randomly choose assignment A
    • Loop m times
        – Randomly select unsatisfied clause C
```

First we randomly pick an unsatisfied clause C (on the grounds that, in order to find a solution, we have to find a way to satisfy all the unsatisfied clauses).



WALKSAT

Like GSAT with additional “noise”

Loop n times

- Randomly choose assignment A
- Loop m times
  - Randomly select unsatisfied clause C
  - With $p = 0.5$ either

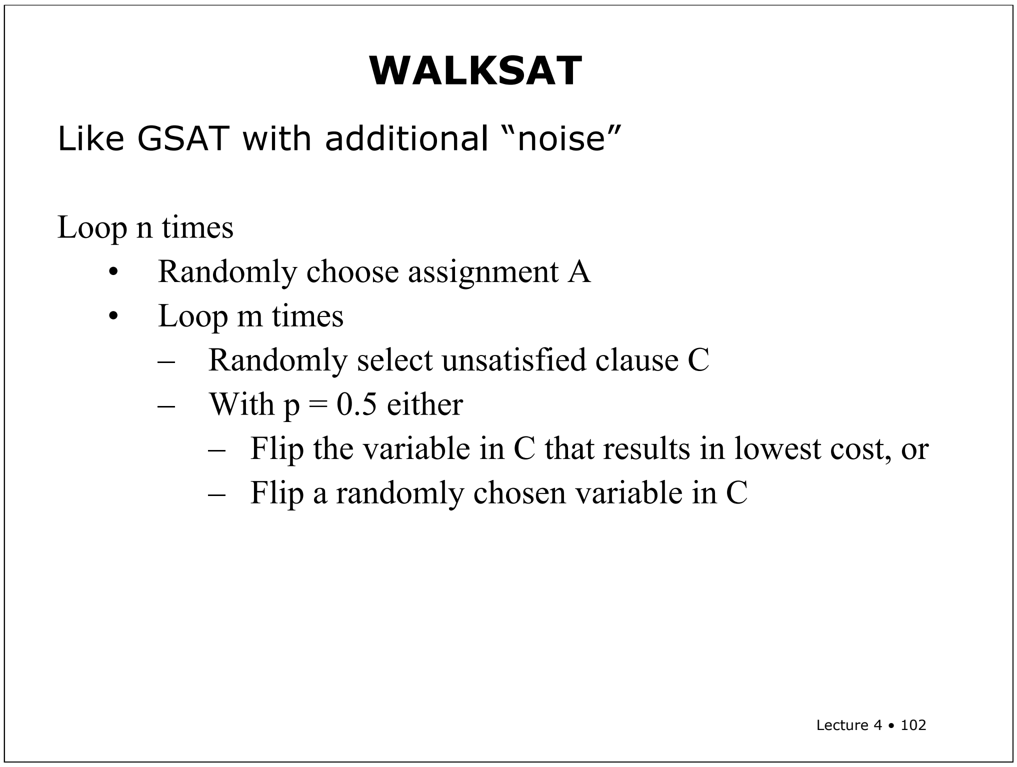
Then, we flip a coin. With probability .5, we either

WALKSAT

Like GSAT with additional "noise"

```
Loop n times
    • Randomly choose assignment A
    • Loop m times
        – Randomly select unsatisfied clause C
        – With p = 0.5 either
            – Flip the variable in C that results in lowest cost, or
```

Flip the variable in C that results in the lowest cost, or

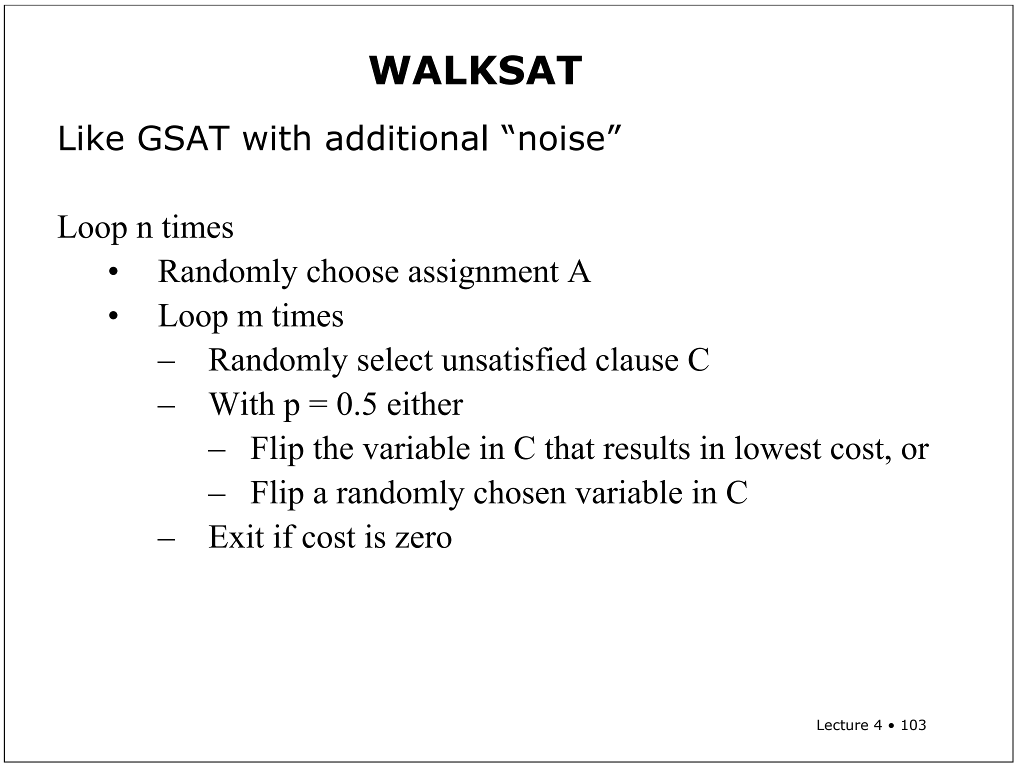


WALKSAT

Like GSAT with additional “noise”

```
Loop n times
    • Randomly choose assignment A
    • Loop m times
        – Randomly select unsatisfied clause C
        – With p = 0.5 either
            – Flip the variable in C that results in lowest cost, or
            – Flip a randomly chosen variable in C
```

Simply flip a randomly chosen variable in C. The reason for flipping randomly chosen variables is that sometimes (as in simulated annealing), its important to take steps that make things worse temporarily, but have the potential to get us into a much better part of the space.



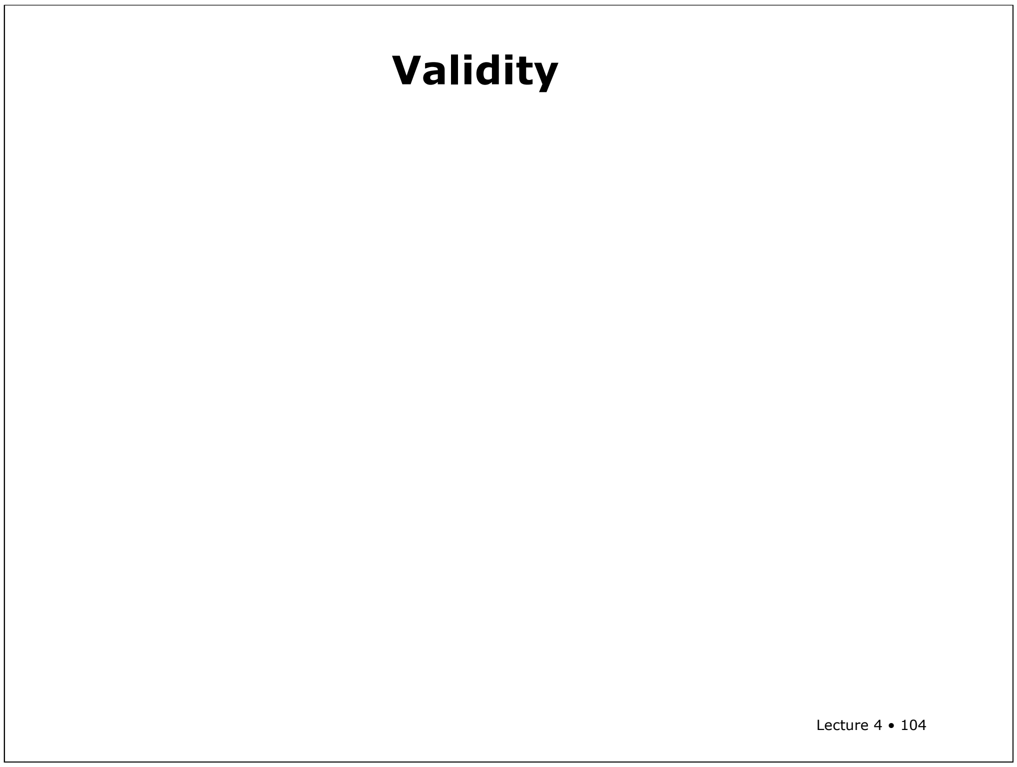
WALKSAT

Like GSAT with additional "noise"

```
Loop n times
    • Randomly choose assignment A
    • Loop m times
        – Randomly select unsatisfied clause C
        – With p = 0.5 either
            – Flip the variable in C that results in lowest cost, or
            – Flip a randomly chosen variable in C
        – Exit if cost is zero
```

Of course, if we find an assignment with cost 0, we're done.

The extra randomness in this algorithm has made it perform better, empirically, than GSAT. But, as you can probably guess from looking at this crazy algorithm, there's no real science to crafting such a local search algorithm. You just have to try some things and see how well they work out in your domain.



Validity

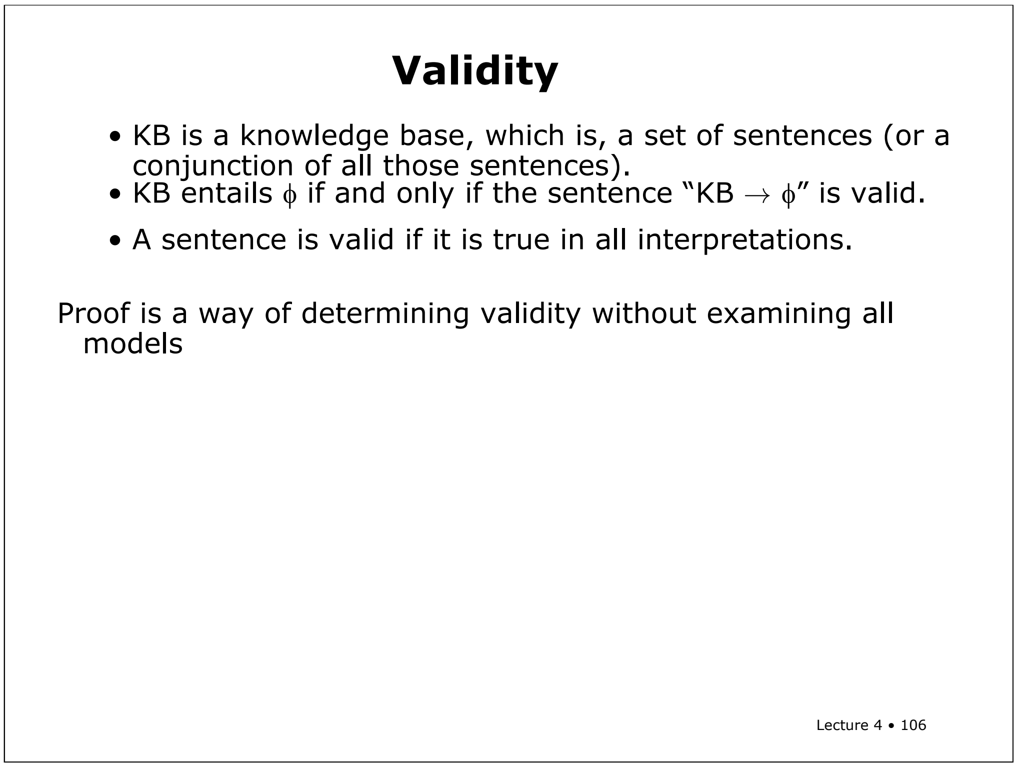
Okay. Now we’re going to switch gears a bit. We have been thinking about procedures to test whether a sentence is satisfiable. Now, we’re going to look at procedures for testing validity.

Why are we interested in validity? Remember the discussion we had near the end of the last lecture, with the complicated diagram? It ended with the following theorem:

Validity

- KB is a knowledge base, which is, a set of sentences (or a conjunction of all those sentences).
- KB entails ϕ if and only if the sentence "KB $\rightarrow \phi$" is valid.
- A sentence is valid if it is true in all interpretations.

KB entails phi if and only if the sentence "KB implies phi" is valid. So, if we can test the validity of sentences, we can tell whether a conclusion is entailed by, or "follows from" some premises.

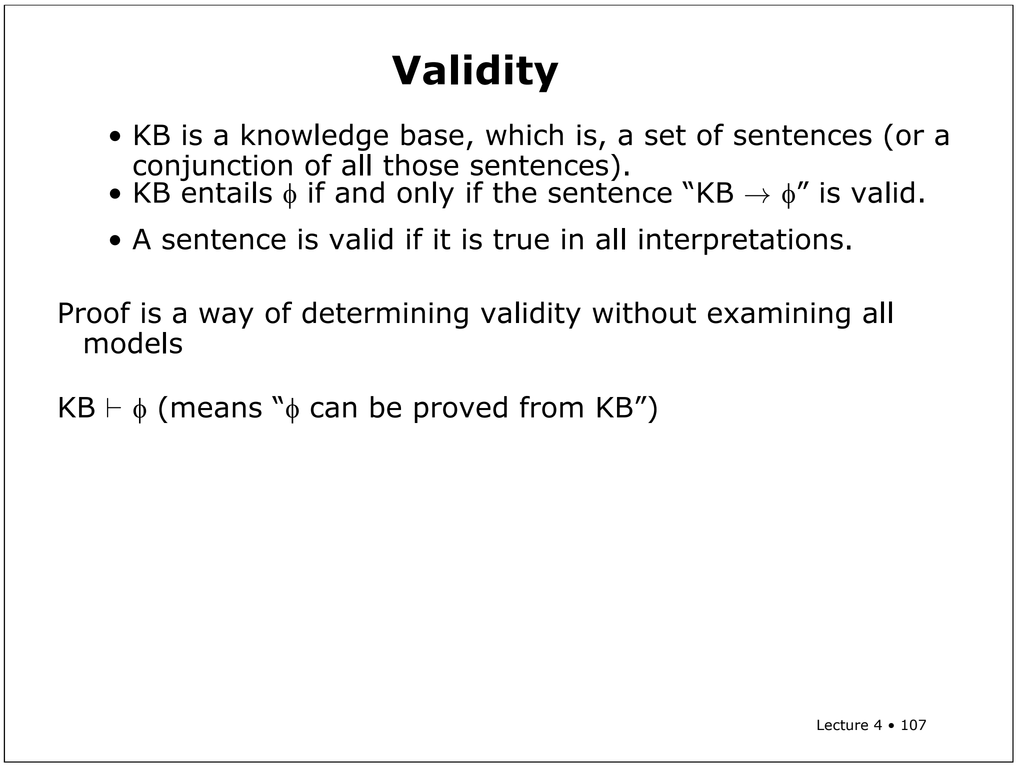


Validity

- KB is a knowledge base, which is, a set of sentences (or a conjunction of all those sentences).
- KB entails ϕ if and only if the sentence "KB $\rightarrow \phi$" is valid.
- A sentence is valid if it is true in all interpretations.

Proof is a way of determining validity without examining all models

Proof is a way of determining validity without examining all models. It works by manipulating the syntactic expressions directly.



Validity

- KB is a knowledge base, which is, a set of sentences (or a conjunction of all those sentences).
- KB entails ϕ if and only if the sentence "KB $\rightarrow \phi$" is valid.
- A sentence is valid if it is true in all interpretations.

Proof is a way of determining validity without examining all models

KB $\vdash \phi$ (means "ϕ can be proved from KB")

We'll introduce a new symbol, single-turnstile, so that KB single-turnstyle Phi means "phi can be proved from KB").

A proof system is a mechanical means of getting new sentences from a set of old ones.

Validity

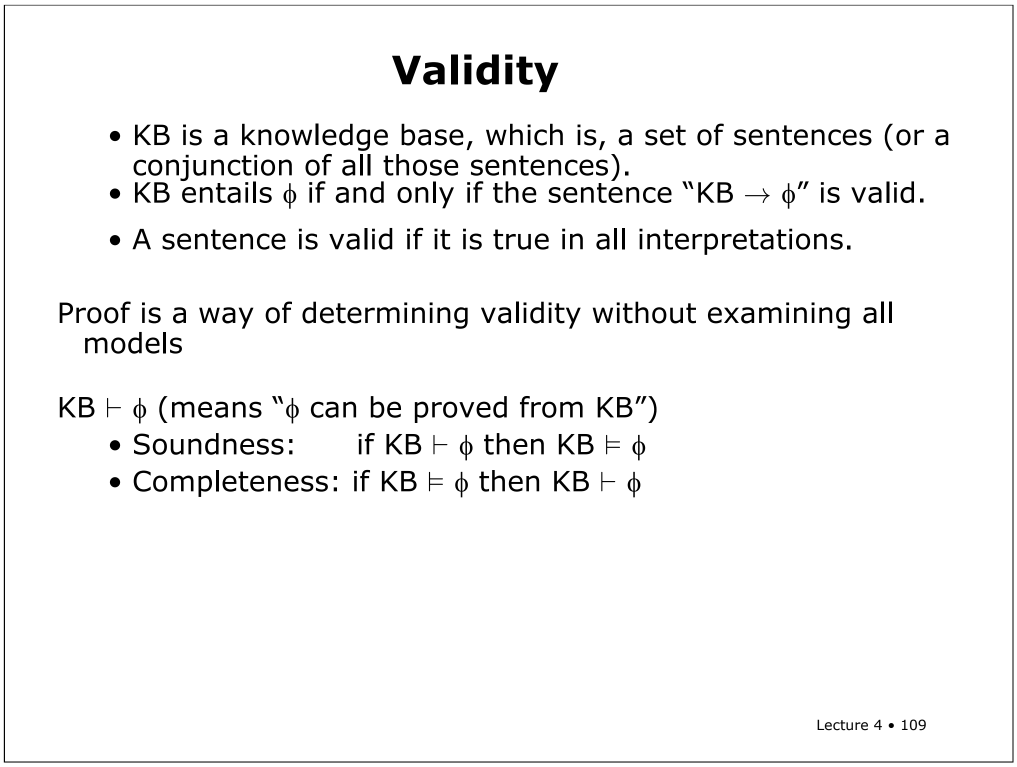
- KB is a knowledge base, which is, a set of sentences (or a conjunction of all those sentences).
- KB entails ϕ if and only if the sentence "KB $\rightarrow \phi$" is valid.
- A sentence is valid if it is true in all interpretations.

Proof is a way of determining validity without examining all models

KB $\vdash \phi$ (means "ϕ can be proved from KB")

- Soundness: if KB $\vdash \phi$ then KB $\models \phi$

A proof system is sound if whenever something is provable from KB it is entailed by KB.



Validity

- KB is a knowledge base, which is, a set of sentences (or a conjunction of all those sentences).
- KB entails ϕ if and only if the sentence "KB $\rightarrow \phi$" is valid.
- A sentence is valid if it is true in all interpretations.

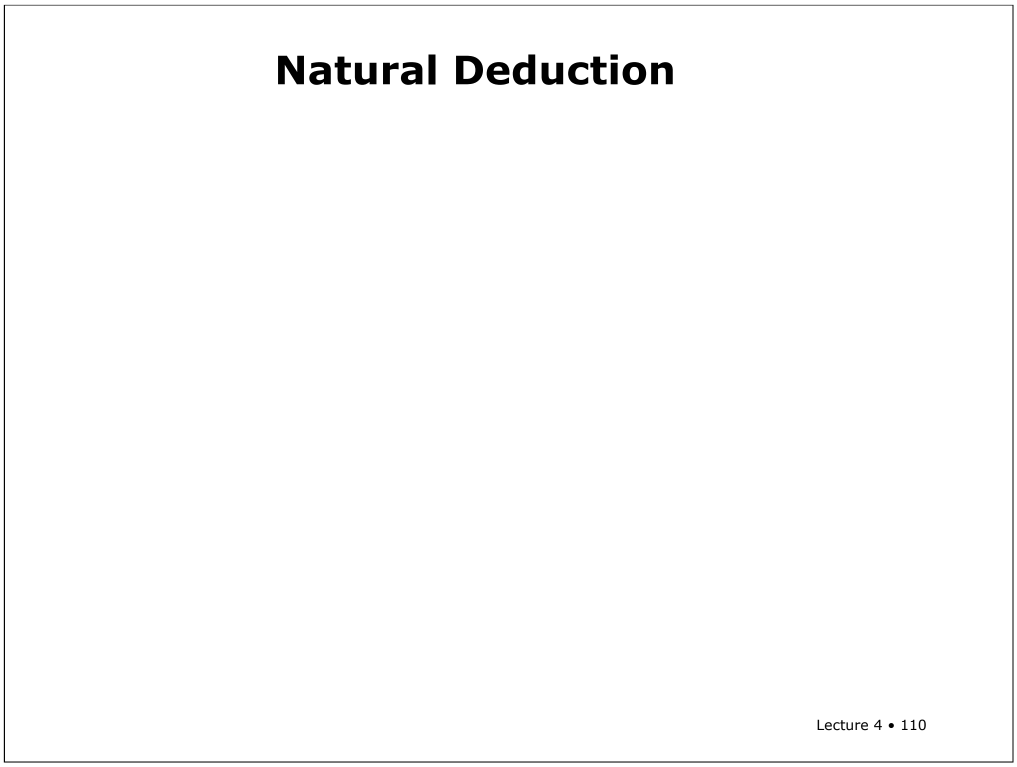
Proof is a way of determining validity without examining all models

KB $\vdash \phi$ (means "ϕ can be proved from KB")

- Soundness: if KB $\vdash \phi$ then KB $\models \phi$
- Completeness: if KB $\models \phi$ then KB $\vdash \phi$

A proof system is complete if whenever something is entailed by KB it is provable from KB.

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!



Natural Deduction

1. So what **is** a proof system? What is this single turnstile about, anyway? 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 kind of grind through, applying the rules that you have to the things that are written down, to write some more stuff down and so finally you've written down the things that you wanted to, and then you to declare victory. That's the single turnstile.

There are (at least) two styles of proof system; we're going to talk about one briefly today and then the other one at some length next time.

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. So this is 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.

Natural Deduction

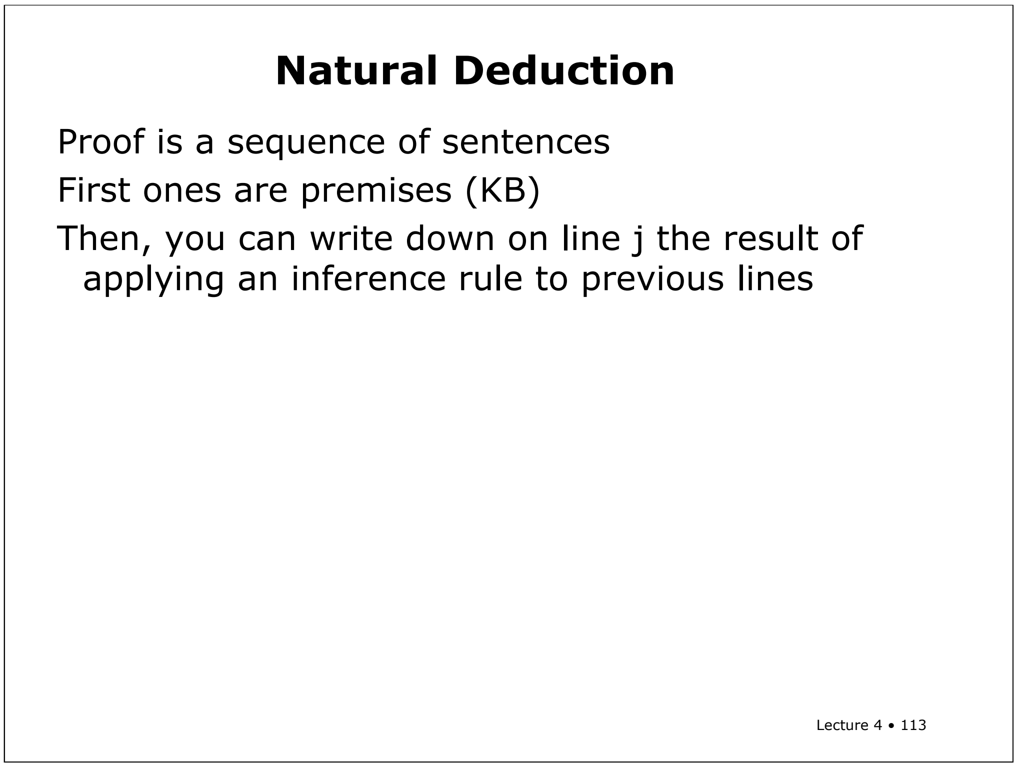
Proof is a sequence of sentences

A proof is a sequence of sentences. This is going to be true in almost all proof systems.

Natural Deduction

Proof is a sequence of sentences
First ones are premises (KB)

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.



Natural Deduction

Proof is a sequence of sentences

First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

Lecture 4 • 113

Then, you can write down on a new line of your proof the results of applying an inference rule to the previous lines.

Natural Deduction

Proof is a sequence of sentences

First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

When ϕ is on a line, you know KB $\vdash \phi$

Then, when Phi is on some line, you just proved Phi from KB.

Natural Deduction

Proof is a sequence of sentences

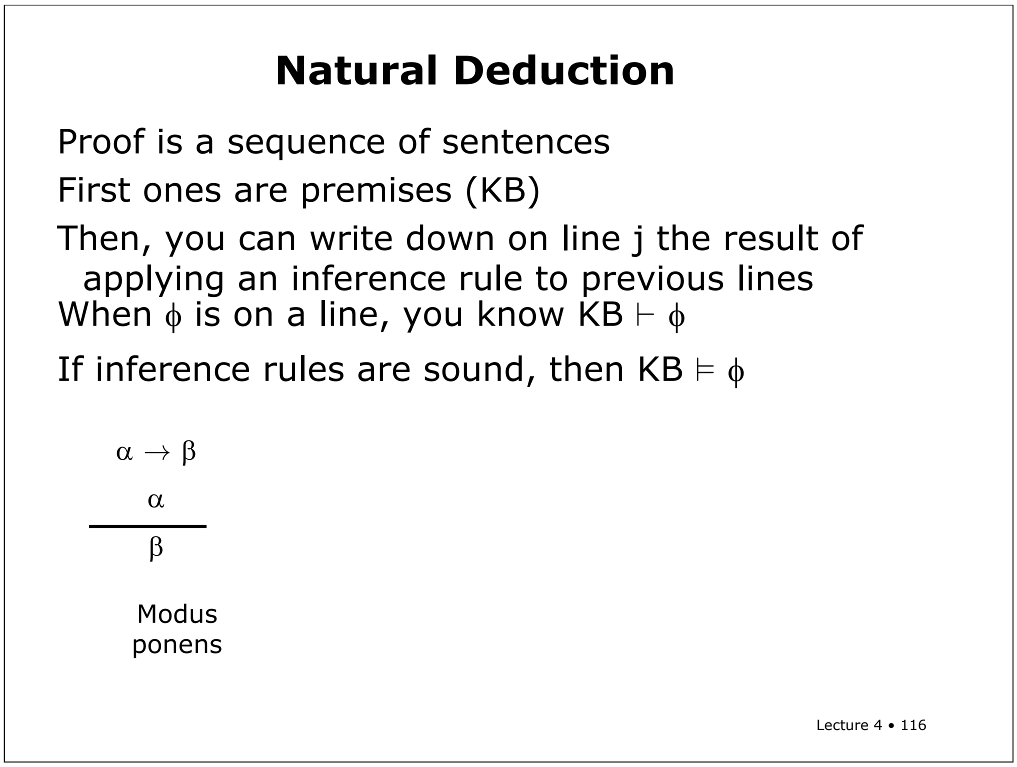
First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

When ϕ is on a line, you know KB $\vdash \phi$

If inference rules are sound, then KB $\vDash \phi$

And if your inference rules are sound, and they'd better be, then KB entails Phi.



Natural Deduction

Proof is a sequence of sentences

First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

When ϕ is on a line, you know KB $\vdash \phi$

If inference rules are sound, then KB $\models \phi$

$$\frac{\alpha \rightarrow \beta \quad \alpha}{\beta}$$

Modus ponens

So let's look at inference rules, and learn how they work by example. Here's a famous one (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."

Natural Deduction

Proof is a sequence of sentences

First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

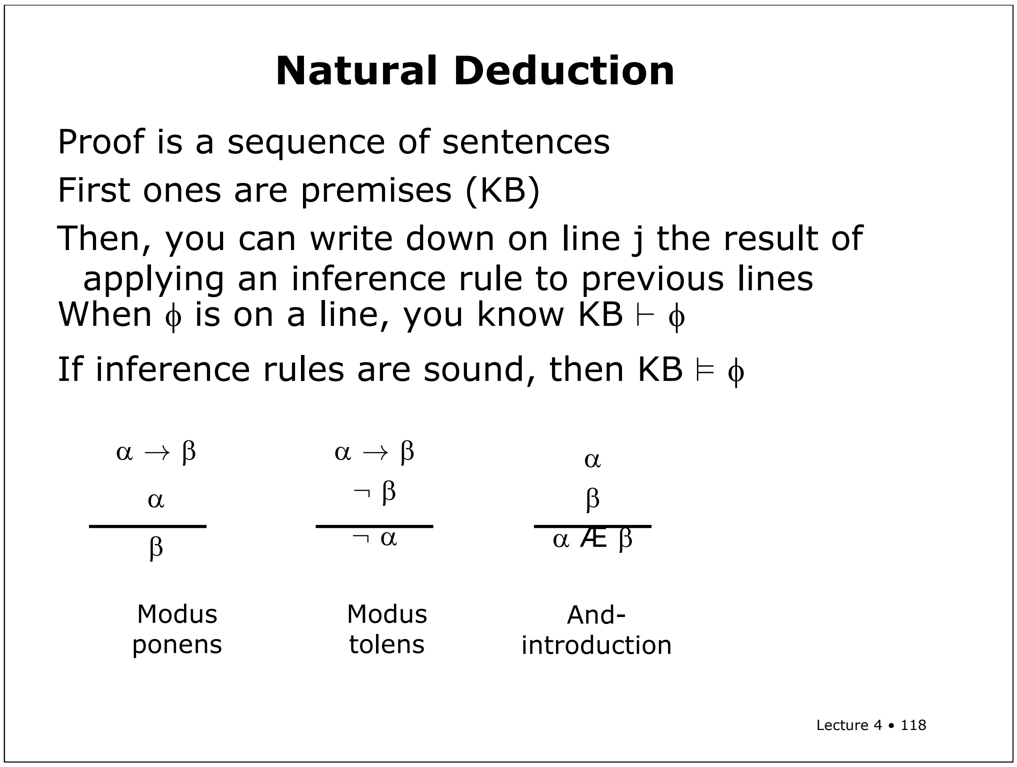
When ϕ is on a line, you know $KB \vdash \phi$

If inference rules are sound, then $KB \models \phi$

$$\frac{\begin{array}{c}\alpha \rightarrow \beta \\ \alpha\end{array}}{\beta} \qquad \frac{\begin{array}{c}\alpha \rightarrow \beta \\ \neg \beta\end{array}}{\neg \alpha}$$

Modus ponens　　Modus tolens

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

Proof is a sequence of sentences

First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

When ϕ is on a line, you know KB $\vdash \phi$

If inference rules are sound, then KB $\vDash \phi$

$$\frac{\begin{array}{c}\alpha \to \beta \\ \alpha\end{array}}{\beta} \qquad \frac{\begin{array}{c}\alpha \to \beta \\ \neg \beta\end{array}}{\neg \alpha} \qquad \frac{\begin{array}{c}\alpha \\ \beta\end{array}}{\alpha \text{ Æ } \beta}$$

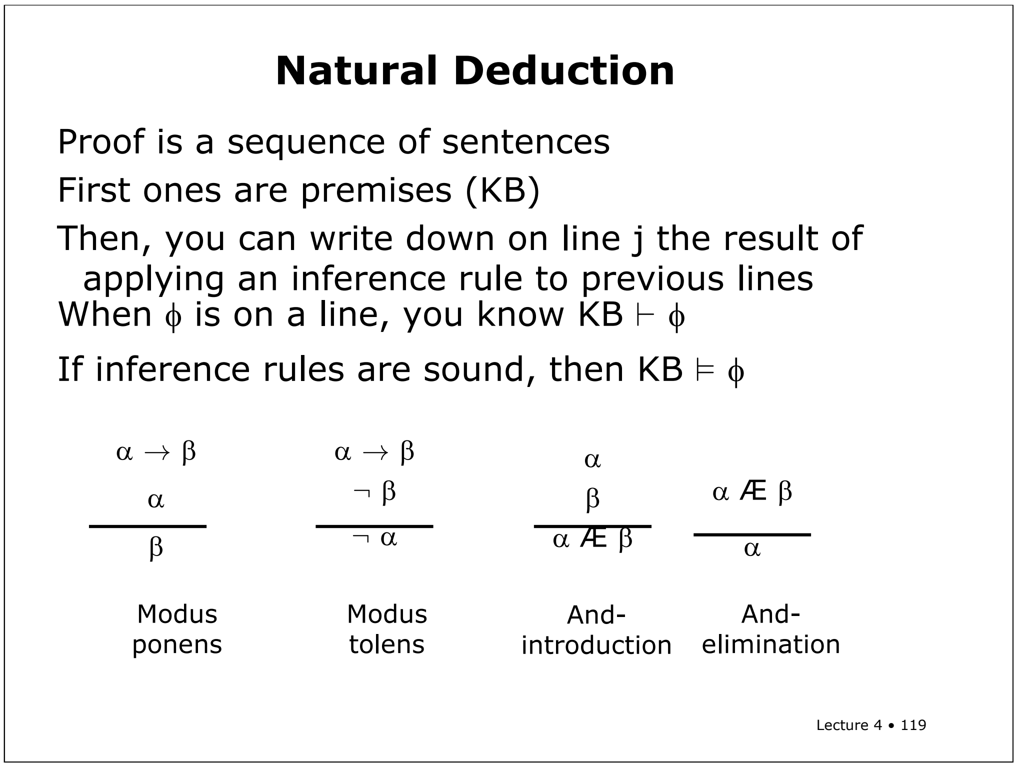
Modus ponens

Modus tolens

And-introduction

Lecture 4 • 118

And-introduction say that from “alpha” and from “beta” you can conclude “alpha and beta”. That seems pretty obvious.



Natural Deduction

Proof is a sequence of sentences

First ones are premises (KB)

Then, you can write down on line j the result of applying an inference rule to previous lines

When ϕ is on a line, you know KB $\vdash \phi$

If inference rules are sound, then KB $\models \phi$

$$\frac{\begin{matrix}\alpha \rightarrow \beta \\ \alpha\end{matrix}}{\beta}$$

Modus ponens

$$\frac{\begin{matrix}\alpha \rightarrow \beta \\ \neg \beta\end{matrix}}{\neg \alpha}$$

Modus tolens

$$\frac{\begin{matrix}\alpha \\ \beta\end{matrix}}{\alpha \text{ Æ } \beta}$$

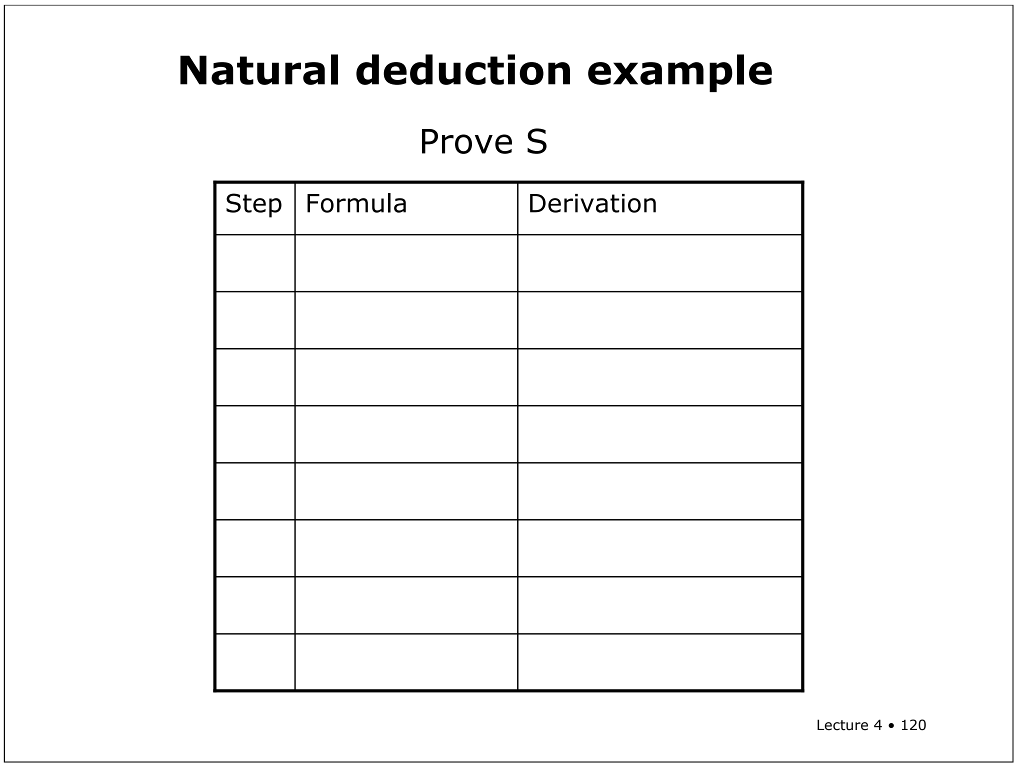
And-introduction

$$\frac{\alpha \text{ Æ } \beta}{\alpha}$$

And-elimination

Lecture 4 • 119

Conversely, and-elimination says that from “alpha and beta” you can conclude “alpha”.

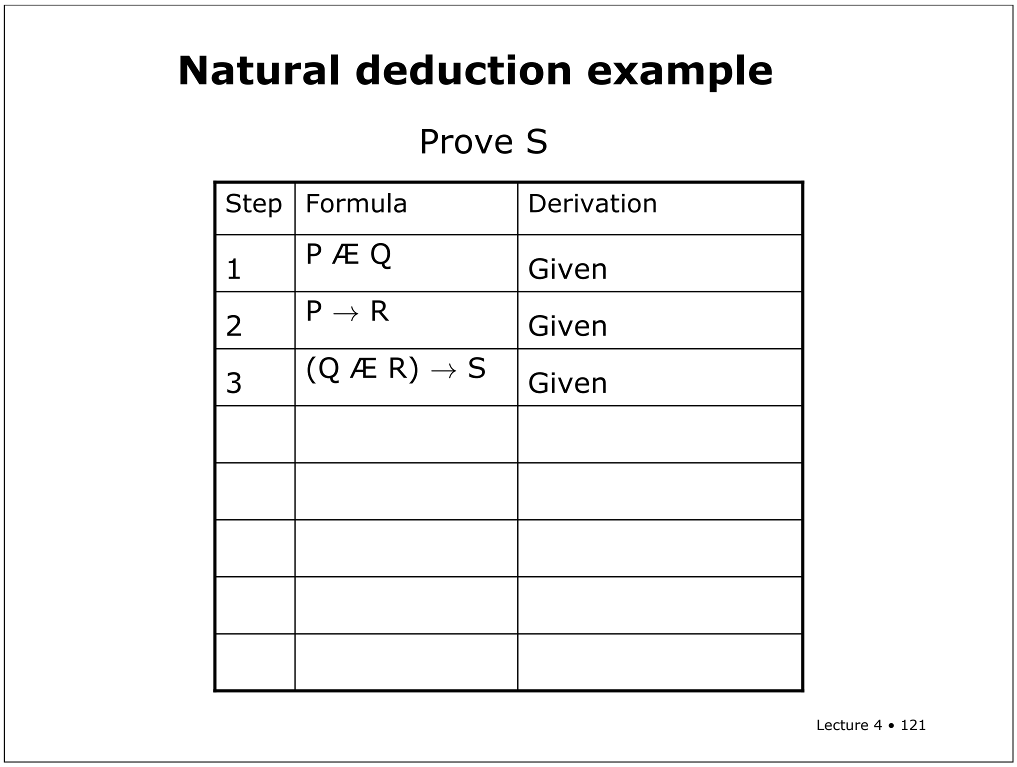


Natural deduction example

Prove S

Step	Formula	Derivation

Now let’s do a sample proof just to get the idea of how it works. Pretend you’re back in high school…

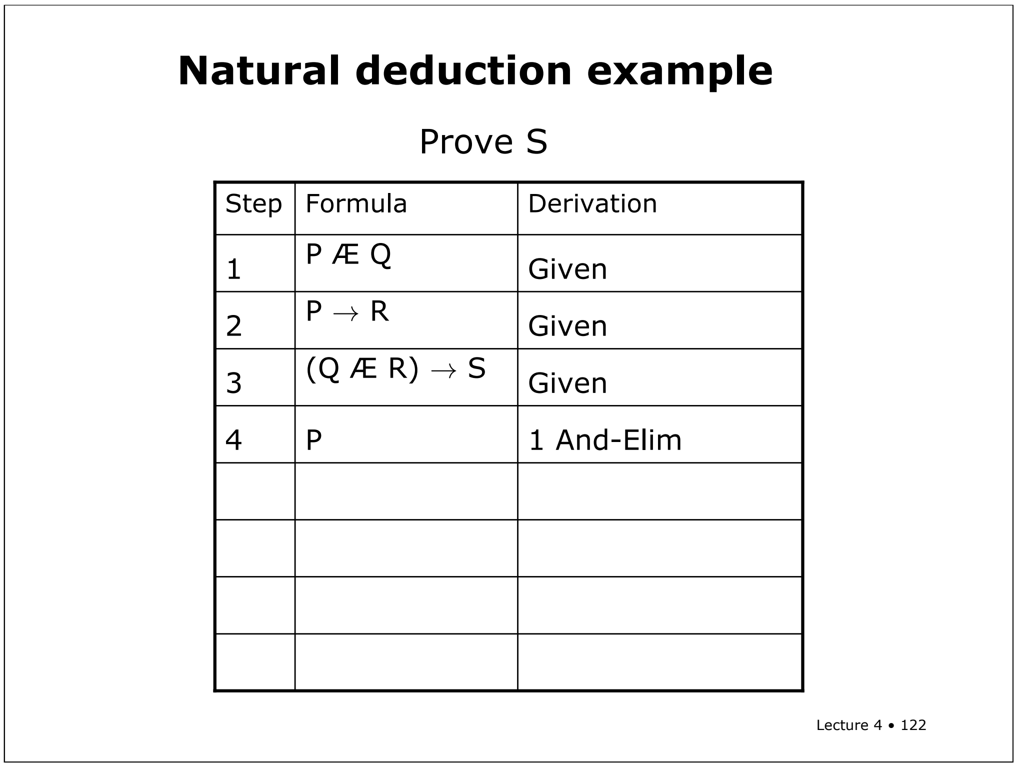


Natural deduction example

Prove S

Step	Formula	Derivation
1	P Æ Q	Given
2	P $\rightarrow$ R	Given
3	(Q Æ R) $\rightarrow$ S	Given

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).

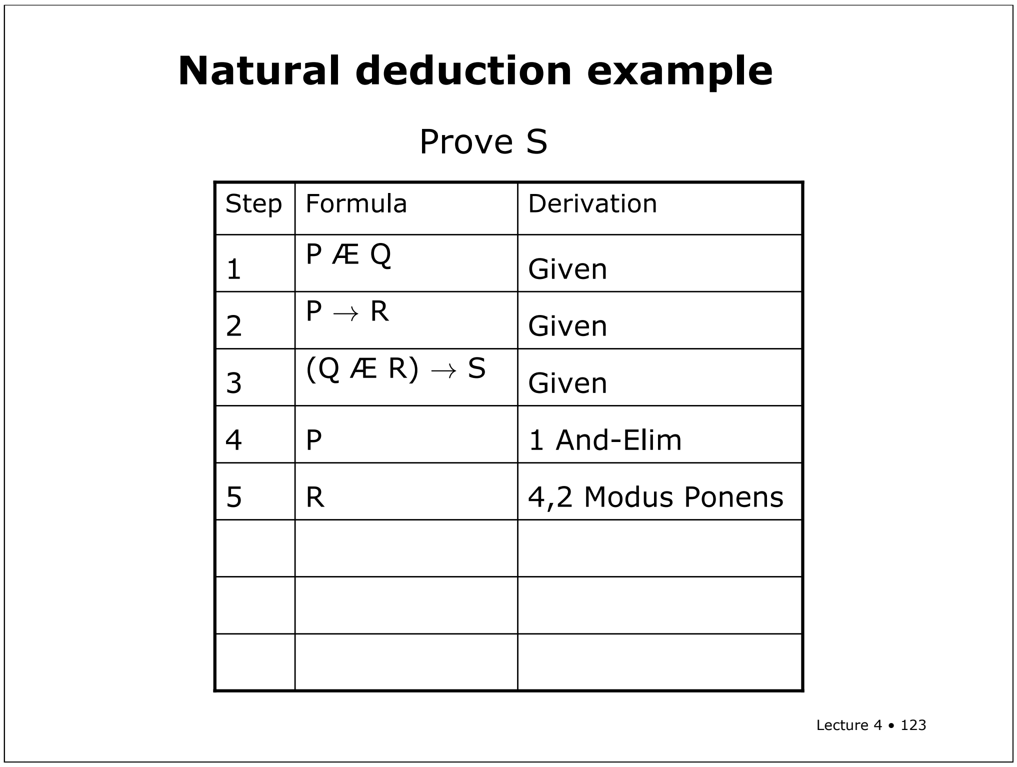


Natural deduction example

Prove S

Step	Formula	Derivation
1	P Æ Q	Given
2	P → R	Given
3	(Q Æ R) → S	Given
4	P	1 And-Elim

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).

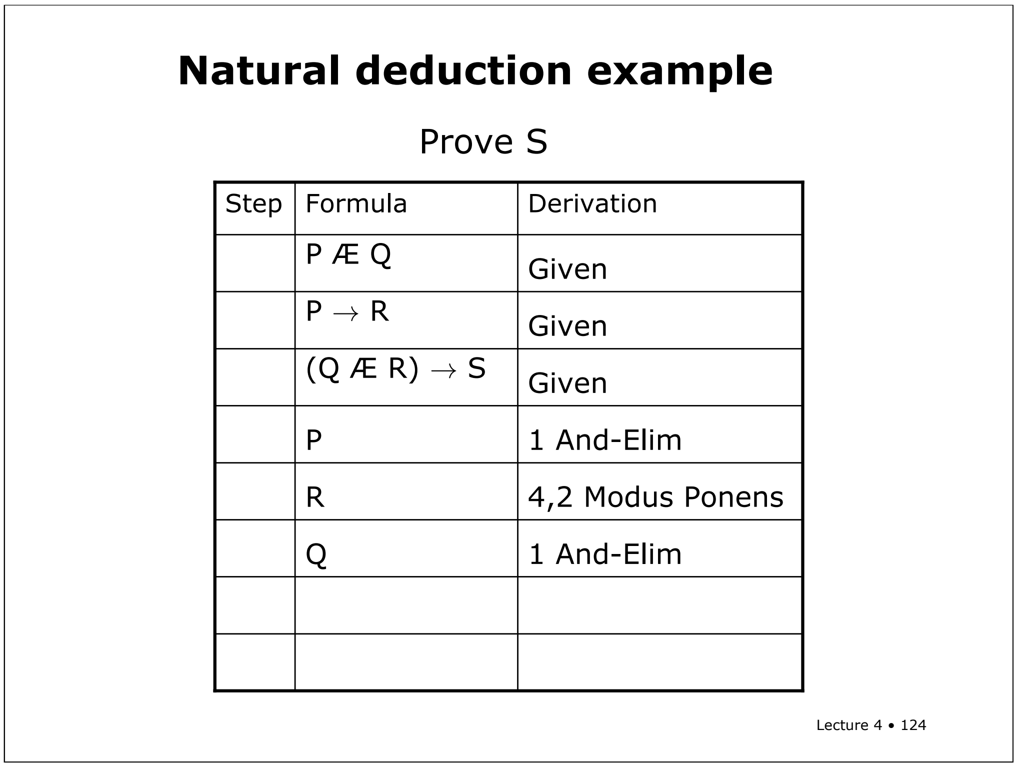


Natural deduction example

Prove S

Step	Formula	Derivation
1	P Æ Q	Given
2	P → R	Given
3	(Q Æ R) → S	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens

From lines 4 and 2, using modus ponens, we can conclude R.

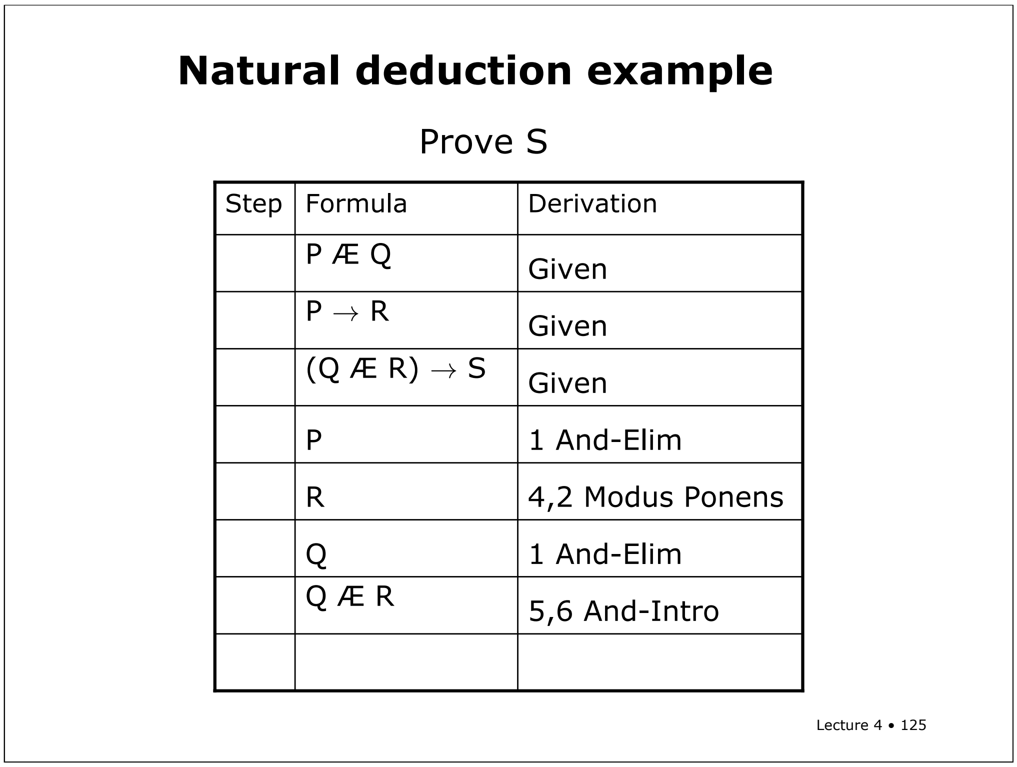


Natural deduction example

Prove S

Step	Formula	Derivation
	P Æ Q	Given
	P → R	Given
	(Q Æ R) → S	Given
	P	1 And-Elim
	R	4,2 Modus Ponens
	Q	1 And-Elim

From line 1, we can use and-elimination to get Q.

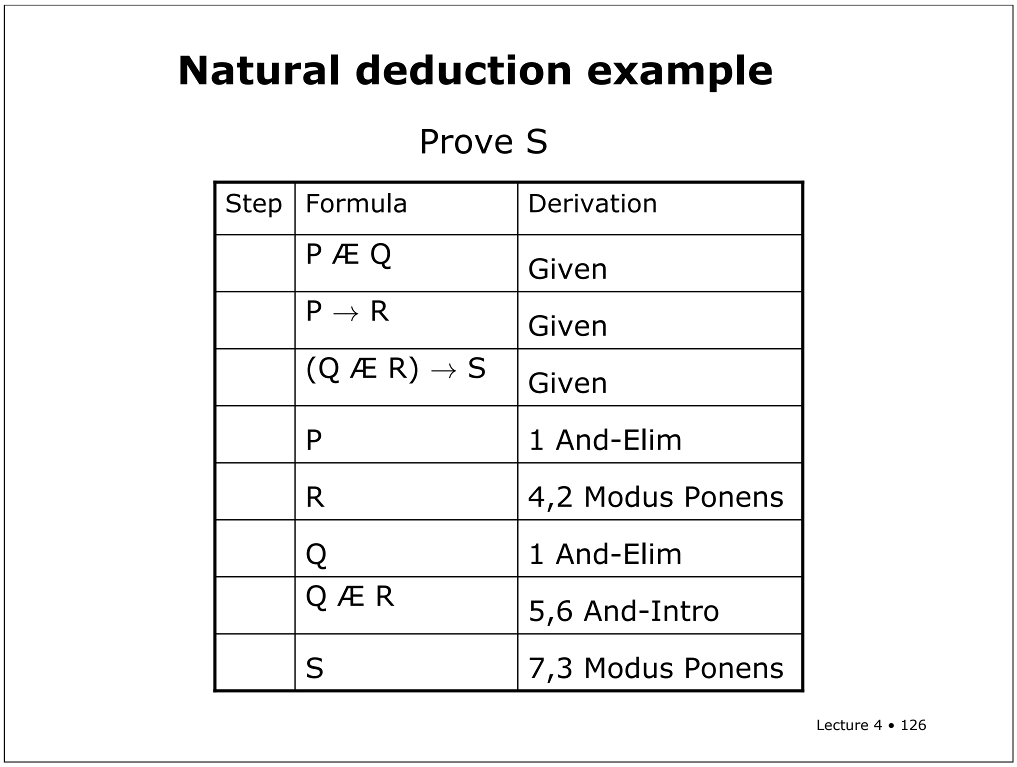


Natural deduction example

Prove S

Step	Formula	Derivation
	P Æ Q	Given
	P $\rightarrow$ R	Given
	(Q Æ R) $\rightarrow$ S	Given
	P	1 And-Elim
	R	4,2 Modus Ponens
	Q	1 And-Elim
	Q Æ R	5,6 And-Intro

From lines 5 and 6, we can use and-introduction to get (Q and R)

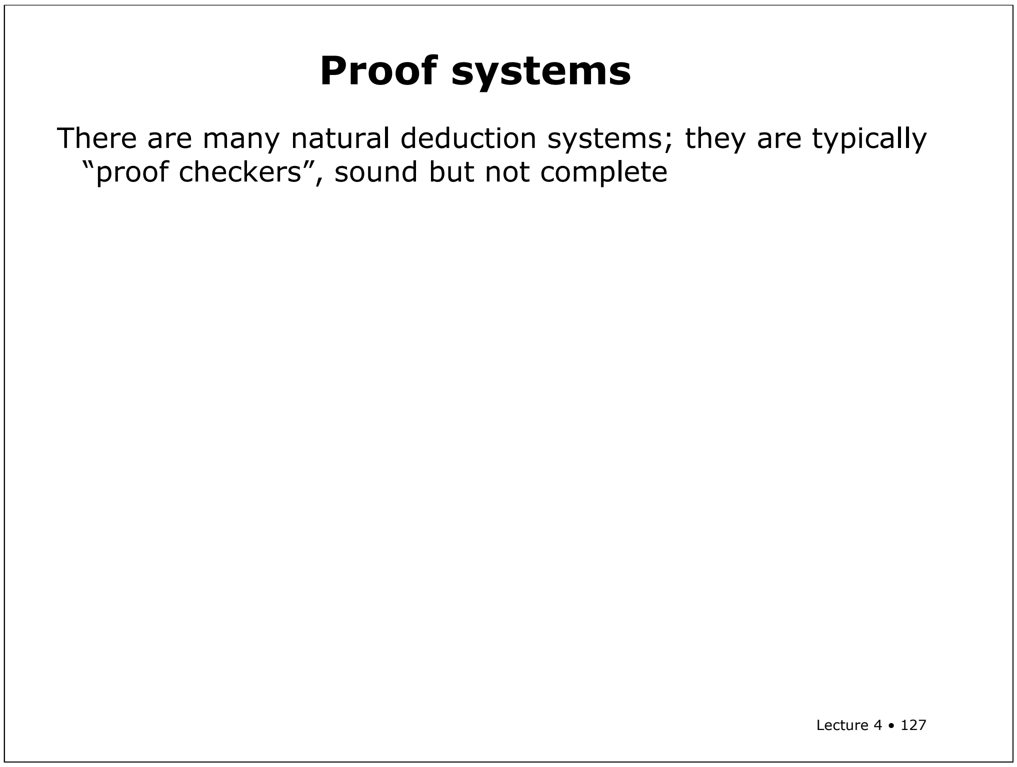


Natural deduction example

Prove S

Step	Formula	Derivation
	P Æ Q	Given
	P → R	Given
	(Q Æ R) → S	Given
	P	1 And-Elim
	R	4,2 Modus Ponens
	Q	1 And-Elim
	Q Æ R	5,6 And-Intro
	S	7,3 Modus Ponens

Finally, from lines 7 and 3, we can use modus ponens to get S. Whew! We did it!

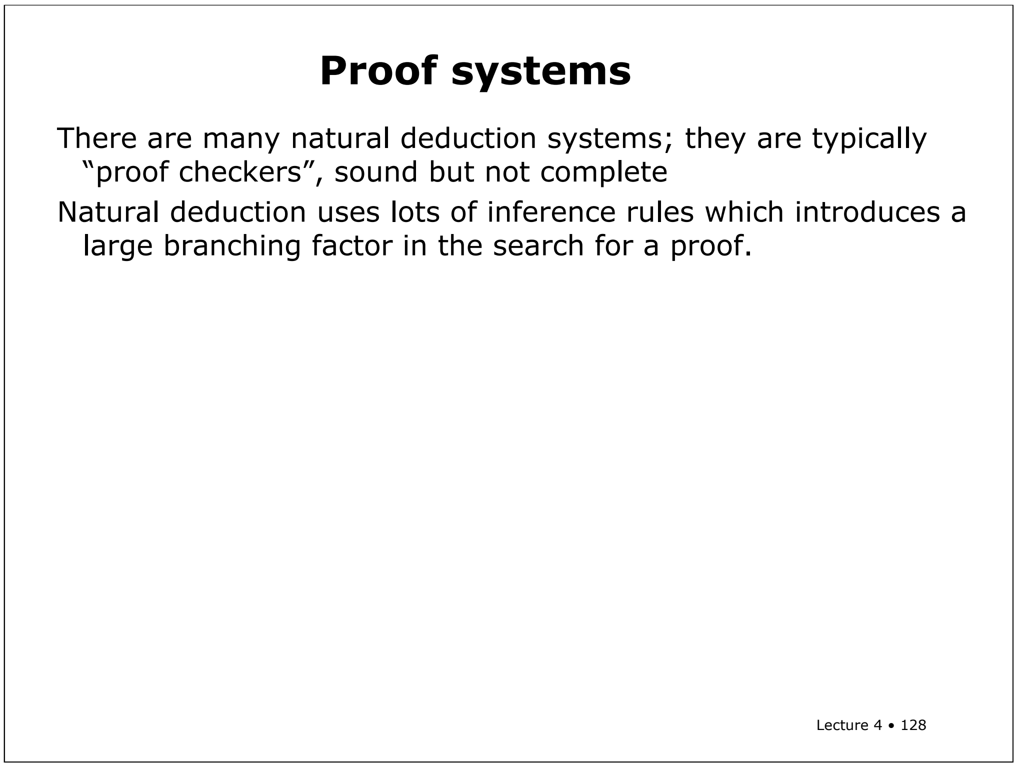


Proof systems

There are many natural deduction systems; they are typically "proof checkers", sound but not complete

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.

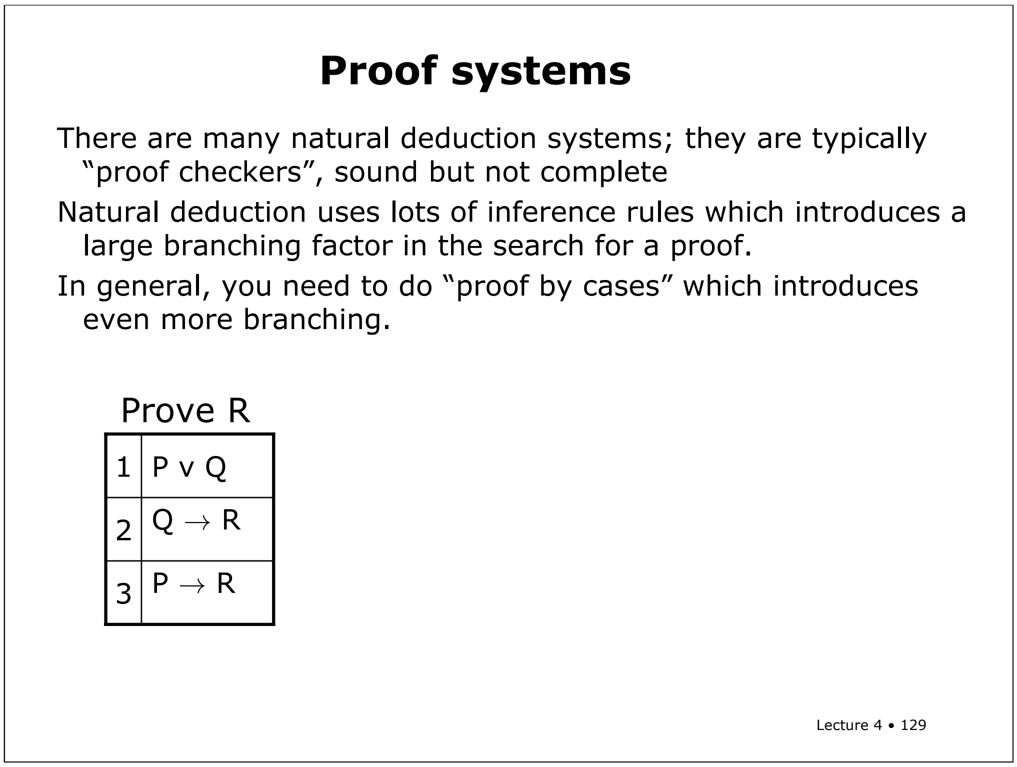


Proof systems

There are many natural deduction systems; they are typically "proof checkers", sound but not complete

Natural deduction uses lots of inference rules which introduces a large branching factor in the search for a proof.

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.



Proof systems

There are many natural deduction systems; they are typically "proof checkers", sound but not complete

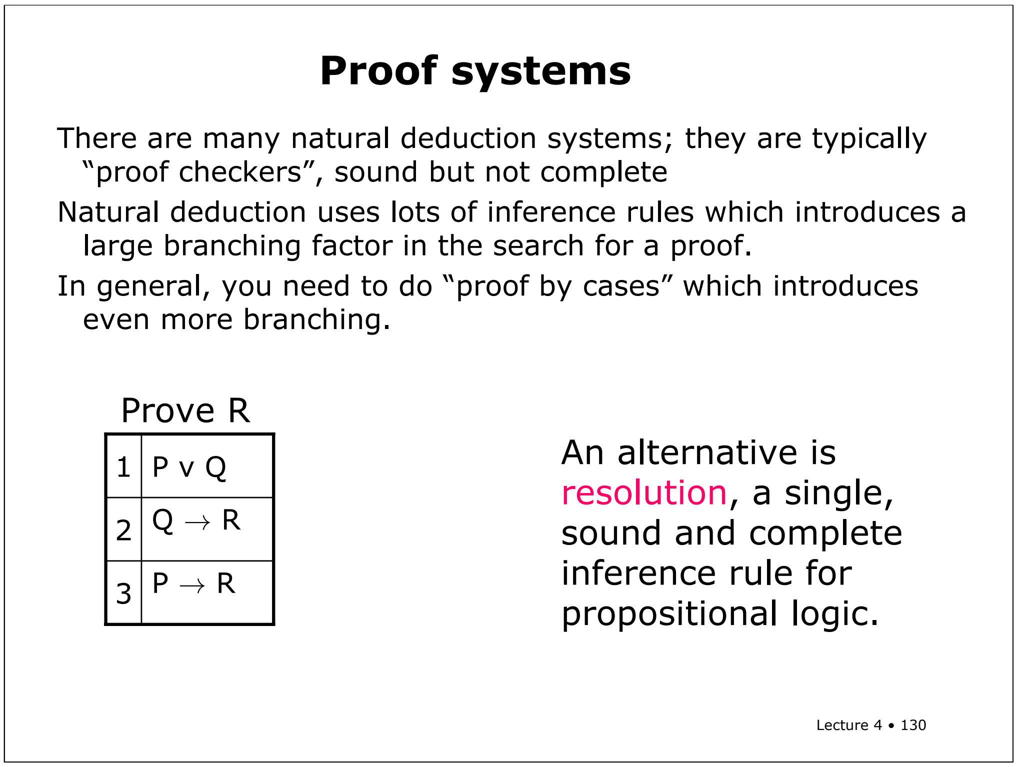
Natural deduction uses lots of inference rules which introduces a large branching factor in the search for a proof.

In general, you need to do "proof by cases" which introduces even more branching.

Prove R

1	P v Q
2	$Q \rightarrow R$
3	$P \rightarrow R$

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 try 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 v Q
2	$Q \rightarrow R$
3	$P \rightarrow R$

An alternative is resolution, a single, sound and complete inference rule for propositional logic.

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. We'll look at it in great detail next time.