Satisfiability

Generate-and-test / NP / NP-complete
SAT, weighted MAX-SAT, CNF-SAT, DNF-SAT, 3-CNF-SAT, Tseitin, TAUT, QSAT
Some applications: LSAT ☺, register allocation
From a previous lecture …

- So, Prof. Eisner, what are declarative methods??
  - A declarative program states only what is to be achieved
  - A procedural program describes explicitly how to achieve it

- Sorting in a declarative language
  - “Given array X, find an array Y such that
    - (1) Y is a permutation of X
    - (2) Y’s elements are in non-decreasing order”
  - Compiler is free to invent any sorting algorithm! (Hard?!)  
  - You should be aware of when compiler will be efficient/inefficient

- Sorting in a procedural language
  - “Given array X, run through it from start to finish, swapping adjacent elements that are out of order …”
  - Longer and probably buggier
  - Never mentions conditions (1) and (2), except in comments
Generate-and-test problems
(a common form of declarative programming)

- Problem specified by a fast “checking” function f(X,Y)
- Input: string x
- Output: Some string y such that f(x,y) = true
  x, y may encode any data you like

- Examples:
  - f(x,y) is true iff y is sorted permutation of x
  - f(x,y) is true iff y is timetable that satisfies everyone’s preferences x
    (note that x and y are encodings as strings)

- NP = {all generate-and-test problems with “easy” f}
  - f(x,y) can be computed in polynomial time O(|x|^k), for any y we should consider (e.g., legal timetables)
  - Could do this for all y in parallel (NP = “nondeterministic polynomial time”)
Generate-and-\textit{compare} problems
(a common form of declarative programming)

- Problem is specified by a fast “scoring” function $f(X,Y)$
- \textbf{Input:} string $x$
- \textbf{Output:} Some string $y$ such that $f(x,y)$ is maximized
  \hspace{1cm} $x, y$ may encode any data you like

- \textbf{Examples:}
  \begin{itemize}
    \item $f(x,y)$ evaluates how well the timetable $y$ satisfies everyone’s preferences $x$ (\textit{note that $x$ and $y$ are encodings as strings})
    \item $\text{OptP} = \{\text{all generate-\&-compare problems with easy } f\}$
      \begin{itemize}
        \item $f(x,y)$ can be computed in polynomial time $O(|x|^k)$, for \textit{any} $y$ we should consider (e.g., legal timetables)
      \end{itemize}
  \end{itemize}
An LSAT Practice Problem
(can we encode this in logic?)

- When the animated “Creature Buddies” go on tour, they are played by puppets.

- Creatures: Dragon, Gorilla, Kangaroo, Tiger
- Names: Audrey, Hamish, Melville, Rex
- Chief Puppeteers: Ben, Jill, Paul, Sue
- Asst. Puppeteers: Chris, Emily, Faye, Zeke
An LSAT Practice Problem
(can we encode this in logic?)

Creatures: Dragon, Gorilla, Kangaroo, Tiger
Names: Audrey, Hamish, Melville, Rex
Chief Puppeteers: Ben, Jill, Paul, Sue
Asst. Puppeteers: Chris, Emily, Faye, Zeke

- Melville isn’t the puppet operated by Sue and Faye.
- Hamish’s chief puppeteer (not Jill) is assisted by Zeke.
- Ben does the dragon, but Jill doesn’t do the kangaroo.
- Chris assists with the tiger.
- Rex (operated by Paul) isn’t the gorilla (not named Melville).

1. What is the Dragon’s name?
2. Who assists with puppet Melville?
3. Which chief puppeteer does Zeke assist?
4. What kind of animal does Emily assist with?

Is there a technique that is guaranteed to solve these?
• Generate & test?
• Is the “grid method” faster? Always works?

from http://www.testprepreview.com/modules/logicalreasoning.htm
Melville isn’t the puppet operated by Sue and Faye.
Melville isn’t the puppet operated by Sue and Faye.
Hamish’s chief puppeteer (not Jill) is assisted by Zeke.

This English gave a few facts: e.g., Jill is not assisted by Zeke.
Hamish's chief puppeteer (not Jill) is assisted by Zeke.
Ben does the dragon, but Jill doesn’t do the kangaroo.
Chris assists with the tiger.
Rex (operated by Paul) isn’t the gorilla (not named Melville).
Rex (operated by Paul) isn’t the gorilla (not named Melville).

Paul doesn’t operate the gorilla.
Grid method

(for puzzles that match people with roles, etc.)

- Each statement simply tells you to fill a particular cell with ✔️ or ❌
- If you fill a cell with ✔️, then fill the rest of its row and column with ❌
- If a row or column has one blank and the rest are ❌, then fill the blank with ✔️
Is that enough?

No! It didn’t even solve the puzzle before.

We need more powerful reasoning patterns ("propagators"):  
  - We were told that Ben operates the dragon.  
  - We already deduced that Ben doesn’t work with Faye.  
  - What should we conclude?
### Declarative Methods

- **J. Eisner**

#### Combine facts from different 4x4 grids:

If $XY \checkmark$ and $YZ \times$ then conclude $XZ \times$

<table>
<thead>
<tr>
<th></th>
<th>creature</th>
<th>assistant</th>
<th>chief</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Name**

- A
- H
- M
- R
- B
- J
- P
- S

**Chief**

- C

**Assistant**

- E
- F
- Z
Combine facts from different 4x4 grids:

If XY ✓ and YZ ✓
then conclude XZ ✓
### Declarative Methods

**J. Eisner**

#### Combine facts from different 4x4 grids:

**If** $XY \checkmark$ **and** $YZ \times$ **then conclude** $XZ \times$
Combine facts from different 4x4 grids:
If $XY \checkmark$ and $YZ \times$ then conclude $XZ \times$
**Declarative Methods**

<table>
<thead>
<tr>
<th>creature</th>
<th>assistant</th>
<th>chief</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>G</td>
<td>K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>B</td>
<td>J</td>
<td>P</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Name**

<table>
<thead>
<tr>
<th>A</th>
<th>H</th>
<th>M</th>
<th>R</th>
<th>B</th>
<th>J</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Chief**

<table>
<thead>
<tr>
<th>A</th>
<th>H</th>
<th>M</th>
<th>R</th>
<th>B</th>
<th>J</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assistant**

<table>
<thead>
<tr>
<th>A</th>
<th>H</th>
<th>M</th>
<th>R</th>
<th>B</th>
<th>J</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Combine facts from different 4x4 grids:
If XY [✓] and YZ [×]
then conclude XZ [×]
Combine facts from different 4x4 grids:
If XY ✓ and YZ ✗ then conclude XZ ✗
### Combine facts from different 4x4 grids:

If $XY \checkmark$ and $YZ \xmark$ then conclude $XZ \xmark$.
**Combine facts from different 4x4 grids:**

If \( XY \) \( \checkmark \) and \( YZ \) \( \times \) then conclude \( XZ \) \( \times \)

No new conclusions from this one

---

**Declarative Methods**

J. Eisner

---

<table>
<thead>
<tr>
<th>creature</th>
<th>assistant</th>
<th>chief</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>G</td>
<td>K</td>
</tr>
<tr>
<td>T</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>Z</td>
<td>B</td>
</tr>
<tr>
<td>J</td>
<td>P</td>
<td>S</td>
</tr>
</tbody>
</table>

**name**

A

H

M

R

B

J

P

S

C

E

F

Z

**chief**

No new conclusions from this one

**assistant**

Combine facts from different 4x4 grids:
Now we can make some easy progress!
Now we can make some easy progress!
Now we can make some easy progress!
The “transitivity rules” let us continue as before. (In fact, they’ll let us completely merge name and chief: we don’t need to keep track of both anymore.)
But does the grid method always apply?
(assuming the statements given are enough to solve the problem)

“Each statement simply tells you to fill a particular cell with ❌ or ✅” – always true?

People: Alice, Bob, Cindy
Roles: Hero, Villain, Jester

- Alice is not the Jester.
- The Villain and the Hero got legally married in Nigeria.

Let’s try it on the board and see what happens …
But does the grid method always apply?
(assuming the statements given are enough to solve the problem)

- Alice is not the Jester.
- The Villain and the Hero got legally married in Nigeria.

The problem is that the second statement didn’t tell us exactly which cells to fill in. It gave us choices.

If these English statements aren’t just telling us about cells to fill in, what kinds of things are they telling us? Can we encode them formally using some little language? Then the computer can solve them …
Logical formulas (inductive definition)

- If $A$ is a [boolean] variable, then $A$ and $\neg A$ are “literal” formulas.
- If $F$ is a formula, then so is $\neg F$ (“not $F$”).

<table>
<thead>
<tr>
<th>F</th>
<th>\neg F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F is a formula, then so is ~F (“not F”).
- If F and G are formulas, then so are F ∧ G, F ∨ G, etc.
- “Truth tables” – like the multiplication table, but for booleans

"not F"

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>~F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Logical formulas (inductive definition)**

- If $A$ is a [boolean] variable, then $A$ and $\sim A$ are “literal” formulas.
- If $F$ is a formula, then so is $\sim F$ (“not $F$”).
- If $F$ and $G$ are formulas, then so are $F \land G$, $F \lor G$, etc.
- “Truth tables” – like the multiplication table, but for booleans

<table>
<thead>
<tr>
<th>$F$</th>
<th>$G$</th>
<th>$\sim F$</th>
<th>$F \land G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical formulas (inductive definition)

- If $A$ is a [boolean] variable, then $A$ and $\neg A$ are “literal” formulas.
- If $F$ is a formula, then so is $\neg F$ (“not $F$”).
- If $F$ and $G$ are formulas, then so are $F \land G$, $F \lor G$, etc.
- “Truth tables” – like the multiplication table, but for booleans

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>¬F</th>
<th>F ∧ G</th>
<th>F ∨ G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F is a formula, then so is ~F (“not F”).
- If F and G are formulas, then so are F ∧ G, F ∨ G, etc.
- “Truth tables” – like the multiplication table, but for booleans

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>~F</th>
<th>F ^ G</th>
<th>F v G</th>
<th>F ⊕ G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Logical formulas (inductive definition)**

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F is a formula, then so is ~F (“not F”).
- If F and G are formulas, then so are F ^ G, F v G, etc.
- “Truth tables” – like the multiplication table, but for booleans

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>~F</th>
<th>F ^ G</th>
<th>F v G</th>
<th>F ⊕ G</th>
<th>F ↔ G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F is a formula, then so is ~F ("not F").
- If F and G are formulas, then so are F ^ G, F v G, etc.
- “Truth tables” – like the multiplication table, but for booleans

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>~F</th>
<th>F &amp; G</th>
<th>F</th>
<th>G</th>
<th>F ↔ G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and \( \neg A \) are “literal” formulas.
- If F is a formula, then so is \( \neg F \) (“not F”).
- If F and G are formulas, then so are \( F \land G \), \( F \lor G \), etc.

So this is a formula: \( (\neg F \rightarrow G) \land \neg(G \lor H) \)

Given an assignment of values to the variables F, G, H,
- We can compute the value of the whole formula, bottom-up
- Just like evaluating arithmetic expressions: \((-F/G) \times -(F \times H)\)
Logical formulas (inductive definition)

- If \( A \) is a [boolean] variable, then \( A \) and \( \neg A \) are “literal” formulas.
- If \( F \) is a formula, then so is \( \neg F \) (“not \( F \)”).
- If \( F \) and \( G \) are formulas, then so are \( F \land G \), \( F \lor G \), etc.

So this is a formula: \( (\neg F \rightarrow G) \land \neg (G \lor H) \)

- Given an assignment of values to the variables \( F, G, H \),
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions: \( (-F/G) \times -(F \times H) \)
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and \( \sim A \) are “literal” formulas.
- If F is a formula, then so is \( \sim F \) (“not F”).
- If F and G are formulas, then so are \( F \land G \), \( F \lor G \), etc.

- So this is a formula: \( (\sim F \rightarrow G) \land \sim(G \lor H) \)
- Given an assignment of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions: \((-F/G) \times -(F \times H)\)
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F is a formula, then so is ~F (“not F”).
- If F and G are formulas, then so are F ^ G, F v G, etc.

So this is a formula: (~F -> G) ^ ~(G v H)

Given an assignment of values to the variables F, G, H,
- We can compute the value of the whole formula, bottom-up
- Just like evaluating arithmetic expressions: (-F/G) × -(F × H)

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>H</th>
<th>(¬F → G) ^ ¬(G v H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

This particular formula is satisfied (has value 1) for only one assignment: F=1, G=0, H=0

So it’s equivalent to F ^ ¬G ^ ¬H
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and \( \sim A \) are “literal” formulas.
- If F is a formula, then so is \( \sim F \) (“not F”).
- If F and G are formulas, then so are \( F \land G \), \( F \lor G \), etc.

- So this is a formula: \( (\sim F \implies G) \land \sim(G \lor H) \)
- Given an assignment of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions: \((-F/G) \times -(F \times H)\)

This particular formula is satisfied (has value 1) for only one assignment: F=1, G=0, H=0

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>H</th>
<th>((\sim F \implies G) \land \sim(G \lor H))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>otherwise</td>
</tr>
</tbody>
</table>

So it’s equivalent to \( F \land \sim G \land \sim H \)
Logical formulas (inductive definition)

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F is a formula, then so is ~F (“not F”).
- If F and G are formulas, then so are F ^ G, F v G, etc.

- So this is a formula: (~F \(\rightarrow\) G) ^ ~ (G v H)
- Given an assignment of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions: (-F/G) \times -(F \times H)

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>H</th>
<th>(~F (\rightarrow) G) ^ ~ (G v H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
<td>0</td>
</tr>
</tbody>
</table>

This particular formula is satisfied (has value 1) for only one assignment: F=1, G=0, H=0

So it’s equivalent to F ^ ~G ^ ~H

Compressed version with “don’t care” values *
Satisfying assignments

- An “assignment” of values to boolean variables is a choice of 0 (false) or 1 (true) for each variable
  - Given an assignment, can compute whether formula is satisfied (true)

Satisfiability problem:
- Input: a formula
- Output: a satisfying assignment to its variables, if one exists. Otherwise return “UNSAT”.
Asking for a satisfying assignment

Let’s try encoding the hero-villain problem on the board:

People: Alice, Bob, Cindy
Roles: Hero, Villain, Jester

- Alice is not the Jester.
- The Villain and the Hero got legally married in Nigeria.

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>~F</th>
<th>F ^ G</th>
<th>F v G</th>
<th>F ⊕ G</th>
<th>F ↔ G</th>
<th>F → G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Satisfying assignments

- An “assignment” of values to boolean variables is a choice of 0 (false) or 1 (true) for each variable
  - Given an assignment, can compute whether formula is satisfied (true)

Satisfiability problem:
- Input: a formula
- Output: a satisfying assignment to its variables, if one exists. Otherwise return “UNSAT”.

Traditional version:
- Input: a formula
- Output: “true” if there exists a satisfying assignment, else “false.”
- Why isn’t this useless in practice??
A Reduction
Unlocking an A..Z assignment in 26 steps

SAT(\phi)?
  yes
  SAT(A^\phi)?
    yes
    SAT(A^B^\phi)?
      yes
      SAT(A^B^...^Z^\phi)?
        yes
        return (A,B,...,Z)=(1,1,1,...1)
      no
      SAT(A^B^...^~Z^\phi)?
        yes
        return (A,B,...,Z)=(0,1,1,...1)
    no
    SAT(\neg A^\phi)\n
return UNSAT

now know SAT(\neg A^\phi)

now know SAT(A^\neg B^\phi)

now know SAT(A^B^...^\neg Z^\phi)

Find a satisfying assignment
Another LSAT Practice Problem

- When the goalie has been chosen, the Smalltown Bluebirds hockey team has a starting lineup that is selected from two groups:

- 1st Group: John, Dexter, Bart, Erwin
- 2nd Group: Leanne, Roger, George, Marlene, Patricia

- Certain requirements are always observed (for the sake of balance, cooperation, and fairness).
Another LSAT Practice Problem

1st Group:  John, Dexter, Bart, Erwin
2nd Group:  Leanne, Roger, George, Marlene, Patricia

- Two players are always chosen from the 1st group. Three players are always chosen from the 2nd group.
- George will only start if Bart also starts.
- Dexter and Bart will not start together.
- If George starts, Marlene won’t start.
- The 4 fastest players are: John, Bart, George and Patricia. 3 of the 4 fastest players will always be chosen.

1. Who always starts?
2. If Marlene starts, what subset of first-group players starts with her?
3. If George starts, who must also start? (M or J, D or L, D or J, J or P, M or R)
4. Which of these pairs cannot start together? (ED, GJ, RJ, JB, PM)

These questions are different: $\forall$ not $\exists$
Encoding “at least 13 of 26”
(without listing all 38,754,732 subsets!)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
<th>L</th>
<th>M</th>
<th>...</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A≥1</td>
<td>A-B≥1</td>
<td>A-C≥1</td>
<td>A-L≥1</td>
<td>A-M≥1</td>
<td>A-Y≥1</td>
<td>A-Z≥1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26 original variables A ... Z, plus < 26^2 new variables such as A-L≥3

- SAT formula should require that A-Z≥13 is true … and what else?
- yadayada \( \wedge A-Z\geq13 \wedge (A-Z\geq13 \rightarrow (A-Y\geq13 \lor (A-Y\geq12 \wedge Z))) \)
  \( \wedge (A-Y\geq13 \rightarrow (A-X\geq13 \lor (A-X\geq12 \wedge Y))) \) \( \wedge \ldots \)
Encoding “at least 13 of 26”
(without listing all 38,754,732 subsets!)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
<th>L</th>
<th>M</th>
<th>...</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A≥1</td>
<td>A-B≥1</td>
<td>A-C≥1</td>
<td></td>
<td>A-L≥1</td>
<td>A-M≥1</td>
<td></td>
<td>A-Y≥1</td>
<td>A-Z≥1</td>
</tr>
</tbody>
</table>

26 original variables A ... Z, plus < 26^2 new variables such as A-L≥3

- SAT formula should require that A-Z≥13 is true ... and what else?

one “only if” definitional constraint for each new variable
Chinese Dinner After the LSAT

- Soup
  - Must be Hot&Sour
- Appetizer
- Pork Dish
- Seafood
- Chicken Dish
  - No Peanuts
- Vegetable
  - No Peanuts
- Rice
  - Not Both Spicy
- Not Chow Mein

Total Cost < $30

slide thanks to Henry Kautz
A general-purpose "generate-and-test solver" is really called a Nondeterministic Turing Machine.

Inner solver is general-purpose and can be given any checking function $f$ as part of its input.

How does this reduction work?

Recall the inner solver’s input and output:

**Input:** string $x$; checking function $f(x,y)$

**Output:** Some string $y$ such that $f(x,y) = true$
**Relation to generate-and-test**

![Diagram](image)

Interreducible problems! (under polynomial encoding/decoding)

So if we could find a fast solver for SAT, we’d solve half the world’s problems (but create new problems, e.g., cryptography wouldn’t work anymore) (neither would some theoretical computer scientists) So probably impossible. But a “pretty good” SAT solver would help the world.
Any generate-and-test problem in NP

These innermost problems are called NP-complete
They are all interreducible with SAT
A “pretty good” solver for any of them would help the world equally
Many practically important examples: see course homepage for pointers
You may well discover some yourself - most things you want are NP-complete
Local register allocation
(an NP-complete planning problem from systems)

- Computer has N fast registers, and unlimited slow memory
- Given a straight sequence of computations using M > N vars:
  - ... x = y*z; w = x+y; ...
- Generate assembly-language code:
  - ##### x = y*z #####
  - LOAD R2, (address of y)
  - LOAD R3, (address of z)
  - MUL R1 R2 R3
  - STORE (address of x), R1
  - ##### w = x+y #####
  - LOAD R1, (address of x)
  - LOAD R2, (address of y)
  - ADD R3 R1 R2
  - STORE (address of w), R3

- Can we eliminate or minimize loads/stores? Reorder code?

Do we need these? Not if y, z are already in registers

Do we need this? Not if we can arrange to keep x in R1 until we need it again

x is already in a register (R1)
y is already in a register (R2)
Clever to pick R3? Is z the best thing to overwrite?
Register allocation as a SAT problem

- Tempting to state everything a reasonable person would know:
  - Don’t have the same variable in two registers
  - Don’t store something unless you’ll need it again ("liveness")
  - Don’t store something that’s already in memory
  - Only ever store the result of the computation you just did
  - ...

- Yes, looks like an optimal solution will observe these constraints

- But you don’t have to state them!

- They are part of the solution, not part of the problem
  - (violating them is allowed, merely dumb)

- Stating these extra constraints might speed up the SAT solver
  - Helps the solver prune away stupid possibilities
  - But that’s just an optimization you can add later
  - Might slow things down if the overhead exceeds the benefit
  - A great SAT solver might discover these constraints for itself
Assumptions about the input code

- Assume all statements look like \( x = y \times z \)
  - Preprocess input to eliminate more complex statements:
    \[
    \begin{align*}
    a &= b \times c + d \times e \\
    \text{temp1} &= b \times c \\
    \text{temp2} &= d \times e \\
    a &= \text{temp1} + \text{temp2}
    \end{align*}
    \]
  - Now all intermediate quantities are associated with variables
  - So if needed, we can store them to memory and reload them later

- Assume variables are constant once computed
  - Preprocess: If \( z \) takes on two different values, split it into \( z_1 \) and \( z_2 \)
    \[
    \begin{align*}
    x &= y \times z \\
    z &= x + 1 \\
    a &= a + z
    \end{align*}
    \]
    \[
    \begin{align*}
    x &= y \times z_1 \\
    z_2 &= x + 1 \\
    a_2 &= a_1 + z_2
    \end{align*}
    \]
  - When \( z_2 \) is in a register or memory, we know which version we have
    \textbf{“static single assignment (SSA) form”}
How to set up SAT variables? (one possibility)

- K time steps, M program vars, N registers
  - SAT variable $2y_3$ is true iff at time 2, $y$ is in R3
  - SAT variable $2y$ is true iff at time 2, $y$ is in memory
  - An assignment to all variables defines a "configuration" of the machine at each time $t$
  - How many variables total?

- How will we decode the SAT solution? (which only tells us what the configs are)
  - For each time $t$, decoder will generate a load-compute-store sequence of machine code to get to $t$'s configuration from $t-1$'s
    1. To put some new values in registers, generate loads
    2. To put other new values in registers, generate computes
       (might use newly loaded registers; order multiple computes topologically)
    3. To put new values in memory, generate stores
       (might use newly computed registers)

Other planning problems:
- Get dressed
- Solve Rubik's cube

Typical when encoding a planning problem

- Time 0 config
  - load-compute-store
- Time 1 config
  - load-compute-store
- Time 2 config
  - load-compute-store
  - ... (other configurations)
- Time K-1 config
  - load-compute-store
- Time K config
  - load-compute-store

preferred (faster than loads)
How to set up SAT variables?

(One possibility)

- K time steps
  - How do we pick K?
  - Large enough to compile input
  - Not too large, or solver is slow
  - Solution: Let K = # of input instructions; we know that’s large enough ...

- How will we decode the SAT solution?
  - For each time t, will generate a load-compute-store sequence of machine code to get to t’s configuration from t-1’s
    1. To put some new values in registers, generate loads
    2. To put other new values in registers, generate computes
       (might use newly loaded registers; order multiple computes topologically)
    3. To put new values in memory, generate stores
       (might use newly computed registers)

\[
\begin{align*}
x &= y \times z \\
w &= x + y \\
t &= w + x
\end{align*}
\]
Example: How do we get from t-1 to t?

- Time 0 –
  \( x = y \times z \)
- Time 1 –
  \( w = x + y \)
- Time 2 –
  \( \ldots \)
- Time K-1 –
  \( t = w + x \)
- Time K –

<table>
<thead>
<tr>
<th>Time</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time t-1</td>
<td>y</td>
<td></td>
<td></td>
<td>y,z</td>
</tr>
<tr>
<td>Time t</td>
<td>x</td>
<td></td>
<td>z</td>
<td>x,y,z</td>
</tr>
</tbody>
</table>

- **LOAD** R3, (address of z)
- **MUL** R1 R1 R3  // compute x into R1, overwriting y
- **STORE** (address of x), R1

For each time t, generate a **load-compute-store** sequence to get to t’s configuration from t-1’s

1. To put some new values in registers, generate **loads**
2. To put other new values in registers, generate **computes**
   
   *(might use newly loaded registers; order multiple computes topologically)*
3. To put new values in memory, generate **stores**
   
   *(might use newly computed registers)*
What is the SAT formula?

- Many constraint clauses combined with “and”:
  - At time K, desired output vars are in mem/registers
  - At time 0, input vars are in mem/certain registers
    - ... but otherwise, no vars are held anywhere!
    - E.g., ~0y ^ 0y1 ^ ~0y2 ^ ... ^ ~0x ^ ~0x1 ^ ~0x2 ...
  - No two vars can be in same register at same time
  - If w is in a register at time t but not t-1, where w=x*y, then
    - either w was in memory at time t-1 (load)
    - or (better) x, y were also in registers at time t (compute)
  - If w is in memory at time t but not t-1, then
    - w was also in a register at time t (store)

What’s missing?

This setup may even reorder or eliminate computations!

This box turns into lots of constraints (why?)

If these hold, we can find an adequate load-compute-store sequence for t-1 \rightarrow t
Where did we say to minimize loads/stores???

- **Additional constraints:**
  - No loads
  - No stores

- **But what if SAT then fails?**
  - Retry with a weaker constraint: “at most 1 load/store” …
  - Continue by linear search or binary chop

If $w$ is in a register at time $t$ but not $t-1$, then
- either $w$ was in memory at time $t-1$
- or (better) $x$, $y$ were also in registers at time $t$

If $w$ is in memory at time $t$ but not $t-1$, then $w$ was also in a register at time $t$
Alternatively, use MAX-SAT

- Specify our input formula as a conjunction of several subformulas (known as “clauses” or “constraints”)
- A MAX-SAT solver seeks an assignment that satisfies as many clauses as possible
- Such conjunctive formulas are very common:
  - each clause encodes a fact about the input, or a constraint

- Can specify a weight for each clause (not all equally important)
- Weighted MAX-SAT: seek an assignment that satisfies a subset of clauses with the maximum total weight possible
  - How does this relate to unweighted MAX-SAT?
  - Can you implement either version by wrapping the other?
- How do we encode our register problem?
  - Specifically, how do we implement “hard” vs. “soft” constraints?
If \( w \) is in a register at time \( t \) but not \( t-1 \), then
- either \( w \) was in memory at time \( t-1 \)
- or (better) \( x, y \) were also in registers at time \( t \)

If \( w \) is in memory at time \( t \) but not \( t-1 \), then \( w \) was also in a register at time \( t \)

We require certain clauses to be satisfied (e.g., so that we'll be able to decode the satisfying assignment into a load-compute-store sequence).

We give these **hard constraints** a weight of \( \infty \). The MAX-SAT solver must respect them.

It's okay to violate “don't load” and “don't store,” but each violation costs 150 cycles of runtime.

We give each of these **soft constraints** a weight of 150. The MAX-SAT solver will try to satisfy as many as it can.

Include both kinds ...

**Hard constraints:**
- weight \( \infty \): \( A \rightarrow (B \lor C) \)
- weight 150: \( A \rightarrow C \)
- weight \( \infty \): \( D \rightarrow E \)
- weight 150: \( \neg D \)

**Soft constraints:**
- A \rightarrow (B \lor C)
- A \rightarrow C
- D \rightarrow E
- \neg D

Strengthen our constraints:
- \( A \rightarrow (B \lor C) \)
- to \( A \rightarrow C \)
- and \( D \rightarrow E \)
- to \( \neg D \)
Eliminating infinite-weight clauses …

- Can specify a weight for each clause (not all equally important)
- Weighted MAX-SAT: seek an assignment that satisfies a subset of clauses with the maximum total weight possible

- What if the solver doesn’t allow $\infty$ as a clause weight?

  **Hard constraints**
  
  weight $\infty$: $A \rightarrow (B \lor C)$
  weight $\infty$: $D \rightarrow E$
  …

  **Soft constraints**
  
  weight 20: $A \rightarrow C$
  weight 14: $\neg D$
  …

- Just give weight $999+1 = 1000$ to each hard constraint in this case
- This acts like weight $\infty$, since solver would rather violate all soft constraints than violate even one hard constraint!
- So solver will only violate hard constraints if it can’t avoid doing so
  - Check the solution. If any hard constraints are violated, then the set of hard constraints must be UNSAT.
Simpler formulas, simpler solvers

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F and G are formulas, then so are
  - F ^ G (“F and G”)
  - F v G (“F or G”)
  - F \rightarrow G (“If F then G”; “F implies G”)
  - F \leftrightarrow G (“F if and only if G”; “F is equivalent to G”)
  - F \oplus G (“F or G but not both”; “F differs from G”)
  - ~F (“not F”)

- Is all of this fancy notation really necessary?
  - Or is some of it just syntactic sugar?
  - Can we write a front-end preprocessor that reduces the user’s formula to a simpler notation? That would simplify solver’s job.
Simpler formulas, simpler solvers

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F and G are formulas, then so are
  - F ^ G
  - F v G
  - F \rightarrow G \rightarrow (~F v G)
  - F \leftrightarrow G \rightarrow (F \rightarrow G) \land (G \rightarrow F) \quad \text{Alternatively: } (F \land G) v (~F \land ~G)
  - F \oplus G \rightarrow (F \land ~G) v (~F \land G)
  - ~F
Step 1: Eliminate $\leftrightarrow$, xor, then $\rightarrow$
Step 1: Eliminate $\leftrightarrow$, xor, then $\rightarrow$
Simpler formulas, simpler solvers

- If A is a [boolean] variable, then A and ~A are “literal” formulas.
- If F and G are formulas, then so are
  - F ^ G
  - F v G
  - F -> G
  - F <-> G
  - F ⊕ G
  - ~F

- If not already a literal, must have form ~(G ^ H) or ~(G v H) or ~G

Additionally:
- (F ^ G) v (~F ^ ~G)
- (F ^ ~G) v (~F ^ G)
- (F v G) v (~F ^ ~G)
Step 2: Push $\sim$ down to leaves

$\sim(G \wedge H)$  $\sim(G \vee H)$  $\sim\sim G$

$\downarrow$

$\sim G \vee \sim H$  $\sim G \wedge \sim H$  $G$

$\downarrow$

$\downarrow$

$\Downarrow$
Step 2: Push ~ down to leaves

\[(\sim G \lor \sim H) \sim (G \land \sim H) \equiv \sim G \land \sim (G \lor H)\]

Before:

After:
Step 2: Push ~ down to leaves

\[ \sim(G \land H) \quad \sim(G \lor H) \]
\[ \sim G \lor \sim H \quad \sim G \land \sim H \]

after
Step 2: Push ~ down to leaves

\[ \neg(G \lor H) \quad \neg(G \land H) \quad \neg\neg G \quad \neg G \lor \neg H \quad \neg G \land \neg H \quad G \]
Step 2: Push \( \sim \) down to leaves

\[
\begin{align*}
\sim(G \land H) & \quad \sim(G \lor H) & \quad \sim \sim G \\
\downarrow & \quad \downarrow & \quad \downarrow \\
\sim G \lor \sim H & \quad \sim G \land \sim H & \quad G
\end{align*}
\]

\[
\begin{align*}
V & \quad (V \lor H) & \quad (G \lor H) \\
\downarrow & \quad \downarrow & \quad \downarrow \\
\sim \sim & \quad \sim \sim & \quad \sim \sim
\end{align*}
\]
Step 2: Push \( \sim \) down to leaves

\[
\begin{align*}
\sim (G \wedge H) & \quad \sim (G \vee H) & \quad \sim \sim G \\
\downarrow & \quad \downarrow & \quad \downarrow \\
\sim G \vee \sim H & \quad \sim G \wedge \sim H & \quad G
\end{align*}
\]
Step 2: Push $\sim$ down to leaves

\[
\begin{align*}
\sim(G \land H) & \quad \sim(G \lor H) & \quad \sim \sim G \\
\downarrow & & \downarrow \\
\sim G \lor \sim H & \quad \sim G \land \sim H & \quad G
\end{align*}
\]
Step 2: Push \(~\) down to leaves
Step 2: Push \( \sim \) down to leaves
Step 2: Push $\sim$ down to leaves

\[ \begin{align*}
\neg(G \wedge H) & \quad \neg(G \vee H) \\
\neg G \vee \neg H & \quad \neg G \wedge \neg H \\
G &
\end{align*} \]
Step 2: Push ~ down to leaves

\[ \sim(G \land H) \quad \sim(G \lor H) \quad \sim\sim G \]

\[ \vdash \quad \vdash \quad \downarrow \]

\[ \sim G \lor \sim H \quad \sim G \land \sim H \quad G \]

Diagram:

```
  V
 / \  
|   |   
H   V  
/   /    
F   |   V   
     |   /     
     |  /      
     | /       
     |/        
   ~G
```

After

```
  \sim H
 /   /
|   | 
V   V
|   | 
\sim \sim \sim
|   |   
J   K
```

Declarative Methods - J. Eisner
Step 2: Push $\sim$ down to leaves

\[
\sim(G \lor H) \quad \sim(G \lor H) \\
\downarrow \\
\sim G \lor \sim H \quad \sim G \lor \sim H
\]

\[
\sim \sim G \\
\downarrow \\
G
\]
Step 2: Push $\sim$ down to leaves

\[
\sim(G \lor H) \quad \sim(G \land H) \\
\downarrow \quad \downarrow \\
\sim G \lor \sim H \quad \sim G \land \sim H
\]

after
Step 2: Push $\sim$ down to leaves

same tree redrawn

(For simplicity, we’re now drawing root as $\lor$ with 4 children. Used to be $(F \lor H) \lor (G \lor \ldots)$, but $F \lor H \lor G \lor \ldots$ is okay since $\lor$ is associative.)

- Now everything is $\land$ and $\lor$ over literals. Yay!
**Duality** ...

So essentially the same: just swap the meanings of “true” and “false” (in both input and output).

Reduce green to brown: encode by negating inputs, decode by negating outputs.

### Truth table for OR

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>F v G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Truth table for AND (listed upside-down from usual order)

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>F ^ G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Reduction

<table>
<thead>
<tr>
<th>“Primal”</th>
<th>“Dual”</th>
<th>Reduction</th>
<th>Reverse reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>^</td>
<td>F ^ G = ~(~F v ~G)</td>
<td>F v G = ~(~F ^ ~G)</td>
</tr>
<tr>
<td>(\exists)</td>
<td>(\forall)</td>
<td>(\forall x \phi(x) = \neg \exists x \neg \phi(x))</td>
<td>(\exists x \phi(x) = \neg \forall x \neg \phi(x))</td>
</tr>
<tr>
<td>SAT</td>
<td>TAUT</td>
<td>TAUT((\phi)) = \neg SAT(\neg \phi)</td>
<td>SAT((\phi)) = \neg TAUT(\neg \phi)</td>
</tr>
<tr>
<td>NP</td>
<td>co-NP</td>
<td>(\forall y f(x,y) = \neg \exists y \neg f(x,y))</td>
<td>(\exists y f(x,y) = \neg \forall y \neg f(x,y))</td>
</tr>
</tbody>
</table>

**“de Morgan’s laws”**
Duality ...

So essentially the same: just swap the meanings of “true” and “false” (in both input and output)

Reduce green to brown: encode by negating inputs, decode by negating outputs

At least one (not empty): like \( \lor \)

- \( \exists x = \text{“there exists } x \text{ such that”} \)
- \( \text{SAT} = \text{is the formula satisfiable? (there exists a satisfying assignment)} \)
- \( \text{NP} = \text{generalization of SAT (problems that return } \text{true} \text{ iff } \exists y f(x,y), \text{ where } f \text{ can be computed in polynomial time } O(|x|^k)) \)

All (complement is not empty): like \( \land \)

- \( \forall x = \text{“all } x \text{ are such that”} \)
- \( \text{TAUT} = \text{is the formula a tautology? (all assignments are satisfying)} \)
- \( \text{co-NP} = \text{generalization of TAUT (problems that return } \text{true} \text{ iff } \forall y f(x,y), \text{ where } f \text{ can be computed in polynomial time } O(|x|^k)) \)
Now everything is ^ and v over literals. Yay!

Can we make it even simpler??

- Yes. We can also push v down, to just above the leaves.
- Then the whole tree will be ^ at the root, over v, over ~ at the leaves.
- That’s called “conjunctive normal form” (CNF).
Step 3: Push v down below ^

Now everything is ^ and v over literals. Yay!

Can we make it even simpler??

- Yes. We can also push v down, to just above the leaves.
- Then the whole tree will be ^ at the root, over v, over ~ at the leaves.
- That’s called “conjunctive normal form” (CNF).
Step 3: Push v down below ^

Now everything is ^ and v over literals. Yay!

Can we make it even simpler??
- Yes. We can also push v down, to just above the leaves.
- Then the whole tree will be ^ at the root, over v, over ~ at the leaves.
- That’s called “conjunctive normal form” (CNF).
Step 3: Push v down below ^

Now everything is ^ and v over literals. Yay!

Can we make it even simpler??

- Yes. We can also push v down, to just above the leaves.
- Then the whole tree will be ^ at the root, over v, over ~ at the leaves.
- That’s called “conjunctive normal form” (CNF).
Step 3: Push \( v \) down below \( ^\wedge \)

- Now everything is \( ^\wedge \) and \( v \) over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push \( v \) down, to just above the leaves.
  - Then the whole tree will be \( ^\wedge \) at the root, over \( v \), over \( \sim \) at the leaves.
  - That’s called “conjunctive normal form” (CNF).
Step 3: Push $v$ down below $^$ 

Now everything is $^$ and $v$ over literals. Yay!

Can we make it even simpler??

- Yes. We can also push $v$ down, to just above the leaves.
- Then the whole tree will be $^$ at the root, over $v$, over $\sim$ at the leaves.
- That’s called “conjunctive normal form” (CNF).
Simpler formulas, simpler solvers

- **Conjunctive normal form (CNF):**
  - Conjunction of disjunctions of literals
  - \((A \lor B \lor \neg C) \land (\neg B \lor D \lor E) \land (\neg A \lor \neg D) \land \neg F \ldots\)
  - Can turn any formula into CNF
  - So all we need is a CNF solver (still NP-complete: no free lunch)

- **Disjunctive normal form (DNF):**
  - Disjunction of conjunctions of literals
  - \((A \land B \land \neg C) \lor (\neg B \land D \land E) \lor (\neg A \land \neg D) \lor \neg F \ldots\)
  - Can turn any formula into DNF
  - So all we need is a DNF solver (still NP-complete: no free lunch)
What goes wrong with DNF

- Satisfiability on a DNF formula takes only linear time:
  \[(A \land B \land \neg C) \lor (\neg B \land D \land E) \lor (\neg A \land \neg D) \lor \neg F \ldots\]

- Proof that we can convert any formula to DNF
  - First convert to \(\lor/\land\) form, then do it recursively:
  - **Base case:** Literals don’t need to be converted
  - **To convert** \(A \lor B\), first convert \(A\) and \(B\) individually
    - \((A_1 \lor A_2 \lor \ldots \lor A_n) \lor (B_1 \lor B_2 \lor \ldots \lor B_m)\)
    - \(= A_1 \lor A_2 \lor \ldots \lor A_n \lor B_1 \lor B_2 \lor \ldots \lor B_m\)
  - **To convert** \(A \land B\), first convert \(A\) and \(B\) individually
    - \((A_1 \land A_2 \land \ldots \land A_n) \land (B_1 \land B_2 \land \ldots \land B_m)\)
    - \(= (A_1 \land B_1) \lor (A_1 \land B_2) \lor \ldots \lor (A_2 \land B_1) \lor (A_2 \land B_2) \lor \ldots \lor (A_n \land B_m)\)
    - Hmm, this is quadratic blowup. What happens to \(A \land B \land C \ldots\) ?
    - \((A_1 \land A_2) \land (B_1 \land B_2) \land (C_1 \land C_2) \land \ldots = \text{what?}\)
    - Exponential blowup – really just generate and test of all combinations
    - So it doesn’t help that we have a linear DNF solver
  - It’s like having a “linear-time” factoring algorithm where the input has to be in unary notation: 2010 encoded as “11111111111111111”
  - Or more precisely, a SAT algorithm where input is a truth table! (* allowed)
Just the same thing goes wrong with CNF

- DNF and CNF seem totally symmetric: just swap v, ^
  - You can convert to CNF or DNF by essentially same algorithm
  - DNF blows up when you try to convert A ^ B ^ C …
  - CNF blows up when you try to convert A v B v C …

- But there is another way to convert to CNF! (Tseitin 1970)
  - Only quadratic blowup after we’ve reduced to v, ^, ~
  - Idea: introduce a new “switching variable” for each v

  \[(B_1 \land B_2 \land B_3) \lor (C_1 \land C_2 \land C_3)\] this is satisfiable iff …
  \[= (Z \rightarrow (B_1 \land B_2 \land B_3)) \land (\lnot Z \rightarrow (C_1 \land C_2 \land C_3))\] … this is satisfiable (solver picks Z)
  \[= (\lnot Z \lor (B_1 \land B_2 \land B_3)) \land (Z \lor (C_1 \land C_2 \land C_3))\]
  distribute v over ^ as before, but gives 3+3 clauses, not 3*3
  \[= (\lnot Z \lor B_1) \land (\lnot Z \lor B_2) \land (\lnot Z \lor B_3) \land (Z \lor C_1) \land (Z \lor C_2) \land (Z \lor C_3)\]
Efficiently encode any formula as CNF

- We rewrote \((B_1 \land B_2 \land B_3) \lor (C_1 \land C_2 \land C_3)\)
  
  \[
  = (\neg Z \lor B_1) \land (\neg Z \lor B_2) \land (\neg Z \lor B_3) \land (Z \lor C_1) \land (Z \lor C_2) \land (Z \lor C_3)
  \]

- Recursively eliminate \(v\) from \((A_1 \land A_2) \lor ((B_1 \land B_2) \lor (C_1 \land C_2))\):
  
  \[
  = (A_1 \land A_2) \lor ((\neg Z \lor B_1) \land (\neg Z \lor B_2) \land (Z \lor C_1) \land (Z \lor C_2))
  \]

- Input formula suffers at worst quadratic blowup. Why? Each of its literals simply becomes a clause with the switching variables that affect it:

  - For example, we start with a copy of \(B_1\) that falls in the 2nd arg of the \(Y \lor\) and the 1st arg of the \(Z \lor\). So it turns into a clause \((Y \lor \neg Z \lor B_1)\).
Efficiently encode any formula as CNF

- We rewrote \((B_1 \land B_2 \land B_3) \lor (C_1 \land C_2 \land C_3)\)
  
  \[= (\neg Z \lor B_1) \land (\neg Z \lor B_2) \land (\neg Z \lor B_3) \land (Z \lor C_1) \land (Z \lor C_2) \land (Z \lor C_3)\]

- Recursively eliminate \(v\) from \((A_1 \land A_2) \lor (\neg (B_1 \lor B_2) \lor (C_1 \lor C_2)) \land D\):

  \[= (A_1 \land A_2) \lor (\neg Z \lor B_1) \land (\neg Z \lor B_2) \land (Z \lor C_1) \land (Z \lor C_2) \land D\]

- Input formula suffers at worst quadratic blowup. Why? Each of its literals simply becomes a clause with the switching variables that affect it:

- For example, we start with a copy of \(B_1\) that falls in the 2nd arg of the \(Y \lor\) and the 1st arg of the \(Z \lor\). So it turns into a clause \((Y \lor \neg Z \lor B_1)\).
So why can’t we use this trick for DNF?

- We rewrote \((B_1 \land B_2 \land B_3) \lor (C_1 \land C_2 \land C_3)\) as a “short” CNF formula:
  
  \[= (~Z \lor B_1) \land (~Z \lor B_2) \land (~Z \lor B_3) \land (Z \lor C_1) \land (Z \lor C_2) \land (Z \lor C_3)\]

- But we can just switch \(\lor\) and \(\land\) – they’re symmetric!

- So why not rewrite \((B_1 \lor B_2 \lor B_3) \land (C_1 \lor C_2 \lor C_3)\) as this “short” DNF?
  
  \[= (~Z \land B_1) \lor (~Z \land B_2) \lor (~Z \land B_3) \lor (Z \land C_1) \lor (Z \land C_2) \lor (Z \land C_3)\]

  - Because we’d get a polytime SAT solver and win the Turing Award.
  - And because the rewrite is clearly wrong. It yields something easy to satisfy.

- The CNF/DNF difference is because we introduced extra variables.

- In CNF, original formula was satisfiable if new formula is satisfiable for either \(Z=\text{true}\) or \(Z=\text{false}\). But wait! We’re trying to switch \(\lor\) and \(\land\):

- In DNF, original formula is satisfiable if new formula is satisfiable for both \(Z=\text{true}\) and \(Z=\text{false}\). Look at the formulas above and see it’s so!
  
  - Alas, that’s not what a SAT checker checks. We’d have to call it many times (once per assignment to the switching variables \(Y, Z, \ldots\)).
## CNF and DNF are duals …

<table>
<thead>
<tr>
<th>“Primal”\</th>
<th>“Dual” \</th>
<th>Reduction \</th>
<th>Reverse reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lor)\</td>
<td>(^\land)\</td>
<td>(F \land G = \neg(\neg F \lor \neg G))</td>
<td>(F \lor G = \neg(\neg F \land \neg G))</td>
</tr>
<tr>
<td>(\exists)\</td>
<td>(\forall)\</td>
<td>(\forall x , \phi(x) = \neg \exists x , \neg \phi(x))</td>
<td>(\exists x , \phi(x) = \neg \forall x , \neg \phi(x))</td>
</tr>
<tr>
<td>SAT \</td>
<td>TAUT \</td>
<td>TAUT((\phi)) = \neg SAT(\neg \phi) \</td>
<td>SAT((\phi)) = \neg TAUT(\neg \phi) \</td>
</tr>
<tr>
<td>NP \</td>
<td>co-NP \</td>
<td>(\forall y , f(x,y) = \neg \exists y , \neg f(x,y))</td>
<td>(\exists y , f(x,y) = \neg \forall y , \neg f(x,y))</td>
</tr>
<tr>
<td>CNF-SAT \</td>
<td>DNF-TAUT \</td>
<td>can reduce problem efficiently to this form, using switching vars, <strong>but</strong> of course it’s still hard to solve</td>
<td></td>
</tr>
<tr>
<td>DNF-SAT \</td>
<td>CNF-TAUT \</td>
<td>problem in this form can be solved in linear time, <strong>but</strong> reducing to this form can blow up the size exponentially since switching variable trick can no longer be used</td>
<td></td>
</tr>
</tbody>
</table>
So we’ve reduced SAT to CNF-SAT

Most SAT solvers are actually CNF-SAT solvers.
- They make you enter a CNF formula.
- To use an arbitrary formula, you have to convert it to CNF yourself.
- Fortunately, many practical problems like the LSAT puzzles are naturally expressed as something close to CNF. (Convert each fact or constraint to CNF and conjoin them all: the result is still in CNF.)
SAT solvers could be even more annoyingly restrictive:

- They could require each CNF clause to have at most 3 literals.
- When converting your input formula to CNF, you’d have to get rid of “long” clauses like \((A_1 \lor A_2 \lor A_3 \lor A_4)\).
- This conversion is easy, so 3-CNF-SAT is still hard (NP-complete).
From CNF-SAT to 3-CNF-SAT

- How do we convert CNF to 3-CNF?
- Again replace “v” with a switching variable.
  - Formerly did that to fix non-CNF: \((A_1 \lor A_2) \lor (B_1 \lor B_2)\)
  - Now do it to fix long clause: \(A_1 \lor A_2 \lor A_3 \lor A_4\)

\[
A_1 \lor A_2 \lor \ldots \lor A_{n-2} \lor A_{n-1} \lor A_n
\]

\[
(Z \rightarrow A_1 \lor A_2 \lor \ldots \lor A_{n-2}) \lor (~Z \rightarrow A_{n-1} \lor A_n)
\]

\[
(A_1 \lor A_2 \lor \ldots \lor A_{n-2} \lor \sim Z) \lor (A_{n-1} \lor A_n \lor Z)
\]

- reduced from length n to length n-1; now recurse!
- length 3 as desired
From CNF-SAT to 3-CNF-SAT

Why can’t we get down to 2-CNF-SAT?

- Actually 2-CNF-SAT can be solved in polynomial time.
- So if we could convert any formula to 2-CNF-SAT in polynomial time, we’d have a proof that $P = NP$.

So it would be surprising if we could use switching variables to get to 2-CNF-SAT:

$$A_1 \lor A_2 \lor A_3$$

$$(Z \rightarrow A_1) \land (\sim Z \rightarrow A_2 \lor A_3)$$

$$(A_1 \lor \sim Z) \land (A_2 \lor A_3 \lor Z)$$

Yay, down to length 2

Oops, still has length 3
The Tseitin Transformation (1968)

- Alternative way to convert any formula directly to 3-CNF.

Disadvantages:
- Introduces even more extra variables, and sometimes more clauses.
- If the original formula is already in 3-CNF, will make it bigger.

Advantages:
- Simple, general, elegant.
- Gets only linear blowup.
  - That is, the CNF formula has length $O(k)$ where $k$ is the length of the original formula.
- Linear even if the original formula contains $\leftrightarrow$, xor.
  - Our old method of eliminating those could get exponential blowup.
  - Why? Converted $F \leftrightarrow G$ to $(F \rightarrow G) \land (G \rightarrow F)$, doubling the length.
  - And $F$, $G$ could contain $\leftrightarrow$ themselves, hence repeated doubling.
The Tseitin Transformation (1968)

Associate a new variable with each internal node (operator) in the formula tree

Constrain each new variable to have the appropriate value given its children

Require the whole formula to have value true!

Now separately convert each of these short clauses to 3-CNF.
E.g., \((X_1 \iff (X_2 \lor X_4))\) becomes 
\((\lnot X_1 \lor X_2 \lor X_4) \land (X_1 \lor X_2) \land (X_1 \lor X_4)\)
The Tseitin Transformation (1968)

Associate a new variable with each internal node (operator) in the formula tree.

You can build such a DAG out of logic gates. So it’s called a “Boolean circuit”:

\[ \neg F \rightarrow (H \lor \neg((J \land K) \lor L)) \lor \neg(G \land (H \lor \neg((J \land K) \lor L))) \]

- The circuit allows shared subformulas, so can be much smaller than the formula written out without sharing.
- Tseitin transformation will reuse the work on each subformula!
- Each node in the DAG gives rise to at most four 3-clauses.
- So the final 3-CNF formula has size that’s linear in the size of the input DAG.
Quantified Satisfiability (QSAT)

- SAT asks whether \( (\exists a) \ F(a) \)  
a is an assignment to all vars
- TAUT asks whether \( (\forall a) \ F(a) \)  
a is an assignment to all vars
- They’re the standard NP-complete and co-NP-complete problems.

- QSAT lets you ask, for example, whether  
\( (\exists a) \ (\forall b) \ (\exists c) \ (\forall d) \ F(a,b,c,d) \)  
a,b,c,d are assignments to non-overlapping subsets of the vars

- Harder!  Worse than NP-complete (outside both NP and co-NP).

- QSAT problems are the standard complete problems for the problem classes higher up in the polynomial hierarchy.

- Example: Can White force a win in 4 moves?  That is: Is there an opening move for you \( (\exists a) \) such that for whatever response Kasparov makes \( (\forall b) \), you will have some reply \( (\exists c) \) so that however Kasparov moves next \( (\forall d) \), you’ve checkmated him in a legal game \( (F(a,b,c,d)) \)?