An Introduction to Boolean Satisfiability

Ian Blumenfeld

CyberPoint International

14 May 2013

Motivation, AKA Who Cares?

What is SAT used for? Some important applications are:

- Circuit synthesis: SAT is built into Xilinx toolchain for FPGAs
- Hardware verification: SAT is an integral part of verifying the Centaur Technologies low power x86 chips
- Software verification: Galois Inc., has used SAT in verification complicated crytogrphic algorithms including a fast Java implementation of elliptic curve Diffie-Hellman encryption
- Malware analysis: CyberPoint has used SAT to help deobfuscate malware, making manual analysis easier

Motivation, AKA Who Cares?

What is SAT used for? Some important applications are:

- Circuit synthesis: SAT is built into Xilinx toolchain for FPGAs
- Hardware verification: SAT is an integral part of verifying the Centaur Technologies low power x86 chips
- Software verification: Galois Inc., has used SAT in verification complicated crytogrphic algorithms including a fast Java implementation of elliptic curve Diffie-Hellman encryption
- Malware analysis: CyberPoint has used SAT to help deobfuscate malware, making manual analysis easier

Motivation, AKA Who Cares?

What is SAT used for? Some important applications are:

- Circuit synthesis: SAT is built into Xilinx toolchain for FPGAs
- Hardware verification: SAT is an integral part of verifying the Centaur Technologies low power x86 chips
- Software verification: Galois Inc., has used SAT in verification complicated crytogrphic algorithms including a fast Java implementation of elliptic curve Diffie-Hellman encryption
- Malware analysis: CyberPoint has used SAT to help deobfuscate malware, making manual analysis easier

Motivation, AKA Who Cares?

What is SAT used for? Some important applications are:

- Circuit synthesis: SAT is built into Xilinx toolchain for FPGAs
- Hardware verification: SAT is an integral part of verifying the Centaur Technologies low power x86 chips
- Software verification: Galois Inc., has used SAT in verification complicated crytogrphic algorithms including a fast Java implementation of elliptic curve Diffie-Hellman encryption
- Malware analysis: CyberPoint has used SAT to help deobfuscate malware, making manual analysis easier

Motivation, AKA Who Cares?

What is SAT used for? Some important applications are:

- Circuit synthesis: SAT is built into Xilinx toolchain for FPGAs
- Hardware verification: SAT is an integral part of verifying the Centaur Technologies low power x86 chips
- Software verification: Galois Inc., has used SAT in verification complicated crytogrphic algorithms including a fast Java implementation of elliptic curve Diffie-Hellman encryption
- Malware analysis: CyberPoint has used SAT to help deobfuscate malware, making manual analysis easier

Motivation, AKA Who Cares?

What is SAT used for? Some important applications are:

- Circuit synthesis: SAT is built into Xilinx toolchain for FPGAs
- Hardware verification: SAT is an integral part of verifying the Centaur Technologies low power x86 chips
- Software verification: Galois Inc., has used SAT in verification complicated crytogrphic algorithms including a fast Java implementation of elliptic curve Diffie-Hellman encryption
- Malware analysis: CyberPoint has used SAT to help deobfuscate malware, making manual analysis easier

What is SAT?

- Let $f : \{0,1\}^n \to \{0,1\}$, i.e. $f(x_0, \ldots, x_{n-1}) \in \{0,1\}$.
- Is there an assignment of 0s and 1s, $a_0, \ldots, a_{n-1}$, to variables $x_0, \ldots, x_{n-1}$ such that $f(a_0, \ldots, a_{n-1}) = 1$?
- If there is, f is satisfiable and $a_0, \ldots, a_{n-1}$ is called a satisfying assignment.
- If not f is unsatisfiable.

What is SAT?

- Let $f : \{0,1\}^n \to \{0,1\}$, i.e. $f(x_0, \ldots, x_{n-1}) \in \{0,1\}$.
- Is there an assignment of 0s and 1s, $a_0, \ldots, a_{n-1}$, to variables $x_0, \ldots, x_{n-1}$ such that $f(a_0, \ldots, a_{n-1}) = 1$?
- If there is, f is satisfiable and $a_0, \ldots, a_{n-1}$ is called a satisfying assignment.
- If not f is unsatisfiable.

What is SAT?

- Let $f : \{0,1\}^n \to \{0,1\}$, i.e. $f(x_0, \ldots, x_{n-1}) \in \{0,1\}$.
- Is there an assignment of 0s and 1s, $a_0, \ldots, a_{n-1}$, to variables $x_0, \ldots, x_{n-1}$ such that $f(a_0, \ldots, a_{n-1}) = 1$?
- If there is, f is satisfiable and $a_0, \ldots, a_{n-1}$ is called a satisfying assignment.
- If not f is unsatisfiable.

What is SAT?

- Let $f : \{0,1\}^n \to \{0,1\}$, i.e. $f(x_0, \ldots, x_{n-1}) \in \{0,1\}$.
- Is there an assignment of 0s and 1s, $a_0, \ldots, a_{n-1}$, to variables $x_0, \ldots, x_{n-1}$ such that $f(a_0, \ldots, a_{n-1}) = 1$?
- If there is, f is satisfiable and $a_0, \ldots, a_{n-1}$ is called a satisfying assignment.
- If not f is unsatisfiable.

What is SAT?

- Let $f : \{0,1\}^n \to \{0,1\}$, i.e. $f(x_0, \ldots, x_{n-1}) \in \{0,1\}$.
- Is there an assignment of 0s and 1s, $a_0, \ldots, a_{n-1}$, to variables $x_0, \ldots, x_{n-1}$ such that $f(a_0, \ldots, a_{n-1}) = 1$?
- If there is, f is satisfiable and $a_0, \ldots, a_{n-1}$ is called a satisfying assignment.
- If not f is unsatisfiable.

Rephrasing in terms of SAT

Many questions can be framed in terms of satisfiability, for example asking if

$$\forall \vec{x} \in \{0,1\}^n.\ P(\vec{x}) = 1$$

is equivalent to asking if

$$\neg P(\vec{x})$$

is satisfiable and taking the opposite answer.

SAT and $P = NP$

SAT is the canonical NP-complete problem.

- This means that if you can find a polynomial time algorithm for solving SAT you win $1000000.
- You still win the cash if you show that it's impossible to find one.
- Good luck.

Despite being in NP, SAT problems that arise naturally can often be solved by good computer programs called SAT solvers.
Most SAT solvers attack something called CNF.

SAT and $P = NP$

SAT is the canonical NP-complete problem.

- This means that if you can find a polynomial time algorithm for solving SAT you win $1000000.
- You still win the cash if you show that it's impossible to find one.
- Good luck.

Despite being in NP, SAT problems that arise naturally can often be solved by good computer programs called SAT solvers.
Most SAT solvers attack something called CNF.

SAT and $P = NP$

SAT is the canonical NP-complete problem.

- This means that if you can find a polynomial time algorithm for solving SAT you win $1000000.
- You still win the cash if you show that it's impossible to find one.
- Good luck.

Despite being in NP, SAT problems that arise naturally can often be solved by good computer programs called SAT solvers.
Most SAT solvers attack something called CNF.

SAT and $P = NP$

SAT is the canonical NP-complete problem.

- This means that if you can find a polynomial time algorithm for solving SAT you win $1000000.
- You still win the cash if you show that it's impossible to find one.
- Good luck.

Despite being in NP, SAT problems that arise naturally can often be solved by good computer programs called SAT solvers.
Most SAT solvers attack something called CNF.

SAT and $P = NP$

SAT is the canonical NP-complete problem.

- This means that if you can find a polynomial time algorithm for solving SAT you win $1000000.
- You still win the cash if you show that it's impossible to find one.
- Good luck.

Despite being in NP, SAT problems that arise naturally can often be solved by good computer programs called SAT solvers.
Most SAT solvers attack something called CNF.

SAT and $P = NP$

SAT is the canonical NP-complete problem.

- This means that if you can find a polynomial time algorithm for solving SAT you win $1000000.
- You still win the cash if you show that it's impossible to find one.
- Good luck.

Despite being in NP, SAT problems that arise naturally can often be solved by good computer programs called SAT solvers.
Most SAT solvers attack something called CNF.

Boolean Functions in CNF

- It is a theorem that any function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be expressed in conjunctive normal form or CNF.
- A CNF is a conjuction of clauses

$$F = c_0 \wedge \ldots \wedge c_{m-1}$$

- A clause is a disjunction of literals

$$c_i = l_{i_0} \vee \ldots \vee l_{i_{s-1}}$$

- A literal is either a variable or a negated variable

$$l_{i_j} = v_{i_j} \text{ or } \overline{v_{i_j}}$$

Boolean Functions in CNF

- It is a theorem that any function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be expressed in conjunctive normal form or CNF.
- A CNF is a conjuction of clauses

$$F = c_0 \wedge \ldots \wedge c_{m-1}$$

- A clause is a disjunction of literals

$$c_i = l_{i_0} \vee \ldots \vee l_{i_{s-1}}$$

- A literal is either a variable or a negated variable

$$l_{i_j} = v_{i_j} \text{ or } \overline{v_{i_j}}$$

Boolean Functions in CNF

- It is a theorem that any function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be expressed in conjunctive normal form or CNF.
- A CNF is a conjuction of clauses

$$F = c_0 \wedge \ldots \wedge c_{m-1}$$

- A clause is a disjunction of literals

$$c_i = l_{i_0} \vee \ldots \vee l_{i_{s-1}}$$

- A literal is either a variable or a negated variable

$$l_{i_j} = v_{i_j} \text{ or } \overline{v_{i_j}}$$

Boolean Functions in CNF

- It is a theorem that any function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be expressed in conjunctive normal form or CNF.
- A CNF is a conjuction of clauses

$$F = c_0 \wedge \ldots \wedge c_{m-1}$$

- A clause is a disjunction of literals

$$c_i = l_{i_0} \vee \ldots \vee l_{i_{s-1}}$$

- A literal is either a variable or a negated variable

$$l_{i_j} = v_{i_j} \text{ or } \overline{v_{i_j}}$$

Boolean Functions in CNF

- It is a theorem that any function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be expressed in conjunctive normal form or CNF.
- A CNF is a conjuction of clauses

$$F = c_0 \wedge \ldots \wedge c_{m-1}$$

- A clause is a disjunction of literals

$$c_i = l_{i_0} \vee \ldots \vee l_{i_{s-1}}$$

- A literal is either a variable or a negated variable

$$l_{i_j} = v_{i_j} \text{ or } \overline{v_{i_j}}$$

The Davis-Putnam Algorithm: Part 1

- First (somewhat) efficient attack on SAT through CNF
- Based on resolution

$$\frac{\Gamma \vee p; \overline{p} \vee \Delta}{\Gamma \vee \Delta}$$

The Davis-Putnam Algorithm: Part 1

- First (somewhat) efficient attack on SAT through CNF
- Based on resolution

$$\frac{\Gamma \vee p; \overline{p} \vee \Delta}{\Gamma \vee \Delta}$$

The Davis-Putnam Algorithm: Part 1

- First (somewhat) efficient attack on SAT through CNF
- Based on resolution

$$\frac{\Gamma \vee p; \overline{p} \vee \Delta}{\Gamma \vee \Delta}$$

The Davis-Putnam Algorithm: Part 1

- First (somewhat) efficient attack on SAT through CNF
- Based on resolution

$$\frac{\Gamma \vee p; \overline{p} \vee \Delta}{\Gamma \vee \Delta}$$

The Davis-Putnam Algorithm: Part 2

The steps of the algorithm are:

- Remove clauses that contain both a variable and its negation (tautologies)
- "Randomly" select a variable
- Add all resolutions on that variable, then delete any clause with that variable remaining.
- Repeat, selecting a new variable, until none remains.

If in the end you are left with an empty clause, the formula is UNSAT. If you are left with no clauses the formula is SAT.

The Davis-Putnam Algorithm: Part 2

The steps of the algorithm are:

- Remove clauses that contain both a variable and its negation (tautologies)
- "Randomly" select a variable
- Add all resolutions on that variable, then delete any clause with that variable remaining.
- Repeat, selecting a new variable, until none remains.

If in the end you are left with an empty clause, the formula is UNSAT. If you are left with no clauses the formula is SAT.

The Davis-Putnam Algorithm: Part 2

The steps of the algorithm are:

- Remove clauses that contain both a variable and its negation (tautologies)
- "Randomly" select a variable
- Add all resolutions on that variable, then delete any clause with that variable remaining.
- Repeat, selecting a new variable, until none remains.

If in the end you are left with an empty clause, the formula is UNSAT. If you are left with no clauses the formula is SAT.

The Davis-Putnam Algorithm: Part 2

The steps of the algorithm are:

- Remove clauses that contain both a variable and its negation (tautologies)
- "Randomly" select a variable
- Add all resolutions on that variable, then delete any clause with that variable remaining.
- Repeat, selecting a new variable, until none remains.

If in the end you are left with an empty clause, the formula is UNSAT. If you are left with no clauses the formula is SAT.

The Davis-Putnam Algorithm: Part 2

The steps of the algorithm are:

- Remove clauses that contain both a variable and its negation (tautologies)
- "Randomly" select a variable
- Add all resolutions on that variable, then delete any clause with that variable remaining.
- Repeat, selecting a new variable, until none remains.

If in the end you are left with an empty clause, the formula is UNSAT. If you are left with no clauses the formula is SAT.

The Davis-Putnam Algorithm: Part 2

The steps of the algorithm are:

- Remove clauses that contain both a variable and its negation (tautologies)
- "Randomly" select a variable
- Add all resolutions on that variable, then delete any clause with that variable remaining.
- Repeat, selecting a new variable, until none remains.

If in the end you are left with an empty clause, the formula is UNSAT. If you are left with no clauses the formula is SAT.

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Improvements to DP

- The DPLL algorithm is an incremental improvement to Davis-Putnam.
- Uses branching and backtracking instead of resolution to potential memory blowups.
- This means, after selecting a variable, just guess its value. Backtrack if you get UNSAT and try the other value.
- Makes use of a few other techniques, such as unit propogation and pure literals.

Modern SAT solvers are all based on DPLL... well, really its direct descendant CDCL

Unit Propogation

- A clause with only a single literal is called a unit clause
- For the formula to be satisfied, the assignment to the variable in a unit clause must match its sign
- If other clauses have the same variable, we can propogate this assignment to them, either satisfying the clause, or reducing the number of variables therein.

Unit Propogation

- A clause with only a single literal is called a unit clause
- For the formula to be satisfied, the assignment to the variable in a unit clause must match its sign
- If other clauses have the same variable, we can propogate this assignment to them, either satisfying the clause, or reducing the number of variables therein.

Unit Propogation

- A clause with only a single literal is called a unit clause
- For the formula to be satisfied, the assignment to the variable in a unit clause must match its sign
- If other clauses have the same variable, we can propogate this assignment to them, either satisfying the clause, or reducing the number of variables therein.

Unit Propogation

- A clause with only a single literal is called a unit clause
- For the formula to be satisfied, the assignment to the variable in a unit clause must match its sign
- If other clauses have the same variable, we can propogate this assignment to them, either satisfying the clause, or reducing the number of variables therein.

Pure Literals

- A pure literal is one whose negation never appears in the formula being considered.
- $$\begin{array}{lll} v, & w, & \overline{z} \\ x, & \overline{w}, & \\ v, & \overline{y}, & \overline{z} \\ \overline{v}, & \overline{x}, & y \end{array}$$
- If a literal is pure, we may assign it its sign, and eliminate clauses containing it from the CNF.

Pure Literals

- A pure literal is one whose negation never appears in the formula being considered.
- $$\begin{array}{lll} v, & w, & \overline{z} \\ x, & \overline{w}, & \\ v, & \overline{y}, & \overline{z} \\ \overline{v}, & \overline{x}, & y \end{array}$$
- If a literal is pure, we may assign it its sign, and eliminate clauses containing it from the CNF.

Pure Literals

- A pure literal is one whose negation never appears in the formula being considered.
- $$\begin{array}{lll} v, & w, & \overline{z} \\ x, & \overline{w}, & \\ v, & \overline{y}, & \overline{z} \\ \overline{v}, & \overline{x}, & y \end{array}$$
- If a literal is pure, we may assign it its sign, and eliminate clauses containing it from the CNF.

Pure Literals

- A pure literal is one whose negation never appears in the formula being considered.
- $$\begin{array}{llll} v, & w, & \overline{z} \\ x, & \overline{w}, & \\ v, & \overline{y}, & \overline{z} \\ \overline{v}, & \overline{x}, & y \end{array}$$
- If a literal is pure, we may assign it its sign, and eliminate clauses containing it from the CNF.

Pure Literals

- A pure literal is one whose negation never appears in the formula being considered.
- $$\begin{array}{lll} v, & w, & \overline{z} \\ x, & \overline{w}, & \\ v, & \overline{y}, & \overline{z} \\ \overline{v}, & \overline{x}, & y \end{array}$$
- If a literal is pure, we may assign it its sign, and eliminate clauses containing it from the CNF.

An Example

Is the following forumla satisfiable? If so, what is an assignment?

$$
\begin{array}{lll}
p & q & r \\
p & \overline{q} & \overline{r} \\
p & \overline{w} & \\
\overline{q} & \overline{r} & \overline{w} \\
\overline{p} & \overline{q} & r \\
u & x & \\
u & \overline{x} & \\
q & \overline{u} & \\
\overline{r} & \overline{u} &
\end{array}
$$

An Example

Is the following forumla satisfiable? If so, what is an assignment?

$$
\begin{array}{lll}
p & q & r \\
p & \overline{q} & \overline{r} \\
p & \overline{w} & \\
\overline{q} & \overline{r} & \overline{w} \\
\overline{p} & \overline{q} & r \\
u & x & \\
u & \overline{x} & \\
q & \overline{u} & \\
\overline{r} & \overline{u} &
\end{array}
$$

Hint: Start by looking for a pure literal.

SAT Meets the Real World

- It turns out that in the real world identification of pure literals is too slow, and not worth doing.
- Identification of unit clauses is extremely important, and there are really clever pointer tricks to do it efficiently.
- In addition the solver may be able to learn lemma clauses as it makes incorrect branching choices. This is the basis of CDCL.

SAT Meets the Real World

- It turns out that in the real world identification of pure literals is too slow, and not worth doing.
- Identification of unit clauses is extremely important, and there are really clever pointer tricks to do it efficiently.
- In addition the solver may be able to learn lemma clauses as it makes incorrect branching choices. This is the basis of CDCL.

SAT Meets the Real World

- It turns out that in the real world identification of pure literals is too slow, and not worth doing.
- Identification of unit clauses is extremely important, and there are really clever pointer tricks to do it efficiently.
- In addition the solver may be able to learn lemma clauses as it makes incorrect branching choices. This is the basis of CDCL.

SAT Meets the Real World

- It turns out that in the real world identification of pure literals is too slow, and not worth doing.
- Identification of unit clauses is extremely important, and there are really clever pointer tricks to do it efficiently.
- In addition the solver may be able to learn lemma clauses as it makes incorrect branching choices. This is the basis of CDCL.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_4 made false. Pointer must move to some unassigned, non-pointed-to literal.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_4 made false. Pointer must move to some unassigned, non-pointed-to literal.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_4 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_5 made false.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_4 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_5 made false.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_4 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_5 made false.
l_0 made false. Pointer cannot move to anywhere not already pointed to.

The Two Pointer Trick

When looking at a clause, we add two pointers to arbitrary literals therein:

l_2 made false by branch choice or unit propogotion
l_1 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_4 made false. Pointer must move to some unassigned, non-pointed-to literal.
l_5 made false.
l_0 made false. Pointer cannot move to anywhere not already pointed to.
We know that we have a unit clause.

Conflict Driven Clause Learning

- State of the art solvers use a variant of DPLL that can learn conflict clauses.
- If branching decisions result in an UNSAT result, keep track of the relevant choices.
- Construct a new clause expressing that these choices will cause a conlfict.
- Backjump in the decision procedure to the first relevant choice point and add the conflict clause. Then proceed with branching.
- This addition to DPLL has resulted in dramatic improvement in the speed of SAT solvers since the first version in 1996.

Conflict Driven Clause Learning

- State of the art solvers use a variant of DPLL that can learn conflict clauses.
- If branching decisions result in an UNSAT result, keep track of the relevant choices.
- Construct a new clause expressing that these choices will cause a conlfict.
- Backjump in the decision procedure to the first relevant choice point and add the conflict clause. Then proceed with branching.
- This addition to DPLL has resulted in dramatic improvement in the speed of SAT solvers since the first version in 1996.

Conflict Driven Clause Learning

- State of the art solvers use a variant of DPLL that can learn conflict clauses.
- If branching decisions result in an UNSAT result, keep track of the relevant choices.
- Construct a new clause expressing that these choices will cause a conlfict.
- Backjump in the decision procedure to the first relevant choice point and add the conflict clause. Then proceed with branching.
- This addition to DPLL has resulted in dramatic improvement in the speed of SAT solvers since the first version in 1996.

Conflict Driven Clause Learning

- State of the art solvers use a variant of DPLL that can learn conflict clauses.
- If branching decisions result in an UNSAT result, keep track of the relevant choices.
- Construct a new clause expressing that these choices will cause a conlfict.
- Backjump in the decision procedure to the first relevant choice point and add the conflict clause. Then proceed with branching.
- This addition to DPLL has resulted in dramatic improvement in the speed of SAT solvers since the first version in 1996.

Conflict Driven Clause Learning

- State of the art solvers use a variant of DPLL that can learn conflict clauses.
- If branching decisions result in an UNSAT result, keep track of the relevant choices.
- Construct a new clause expressing that these choices will cause a conlfict.
- Backjump in the decision procedure to the first relevant choice point and add the conflict clause. Then proceed with branching.
- This addition to DPLL has resulted in dramatic improvement in the speed of SAT solvers since the first version in 1996.

Conflict Driven Clause Learning

- State of the art solvers use a variant of DPLL that can learn conflict clauses.
- If branching decisions result in an UNSAT result, keep track of the relevant choices.
- Construct a new clause expressing that these choices will cause a conlfict.
- Backjump in the decision procedure to the first relevant choice point and add the conflict clause. Then proceed with branching.
- This addition to DPLL has resulted in dramatic improvement in the speed of SAT solvers since the first version in 1996.

Non-CNF Based Techniques: BDDs

- Binary decision diagrams are an alternate way to CNF to express Boolean formulae.
- BDDs are binary trees with variables as nodes and true-false choices as edges. The leaves are True or False, giving the final output of the formula.
- Given an input order, there is a canonical way to reduce a BDD to a simpler form.
- Once one has a reduced BDD, determing satisfiability is easy. Just answer, "is there a path to the True leaf?"
- So why is it that BDD's don't give us a polynomial-time solution to SAT?

Non-CNF Based Techniques: BDDs

- Binary decision diagrams are an alternate way to CNF to express Boolean formulae.
- BDDs are binary trees with variables as nodes and true-false choices as edges. The leaves are True or False, giving the final output of the formula.
- Given an input order, there is a canonical way to reduce a BDD to a simpler form.
- Once one has a reduced BDD, determing satisfiability is easy. Just answer, "is there a path to the True leaf?"
- So why is it that BDD's don't give us a polynomial-time solution to SAT?

Non-CNF Based Techniques: BDDs

- Binary decision diagrams are an alternate way to CNF to express Boolean formulae.
- BDDs are binary trees with variables as nodes and true-false choices as edges. The leaves are True or False, giving the final output of the formula.
- Given an input order, there is a canonical way to reduce a BDD to a simpler form.
- Once one has a reduced BDD, determing satisfiability is easy. Just answer, "is there a path to the True leaf?"
- So why is it that BDD's don't give us a polynomial-time solution to SAT?

Non-CNF Based Techniques: BDDs

- Binary decision diagrams are an alternate way to CNF to express Boolean formulae.
- BDDs are binary trees with variables as nodes and true-false choices as edges. The leaves are True or False, giving the final output of the formula.
- Given an input order, there is a canonical way to reduce a BDD to a simpler form.
- Once one has a reduced BDD, determing satisfiability is easy. Just answer, "is there a path to the True leaf?"
- So why is it that BDD's don't give us a polynomial-time solution to SAT?

Non-CNF Based Techniques: BDDs

- Binary decision diagrams are an alternate way to CNF to express Boolean formulae.
- BDDs are binary trees with variables as nodes and true-false choices as edges. The leaves are True or False, giving the final output of the formula.
- Given an input order, there is a canonical way to reduce a BDD to a simpler form.
- Once one has a reduced BDD, determing satisfiability is easy. Just answer, "is there a path to the True leaf?"
- So why is it that BDD's don't give us a polynomial-time solution to SAT?

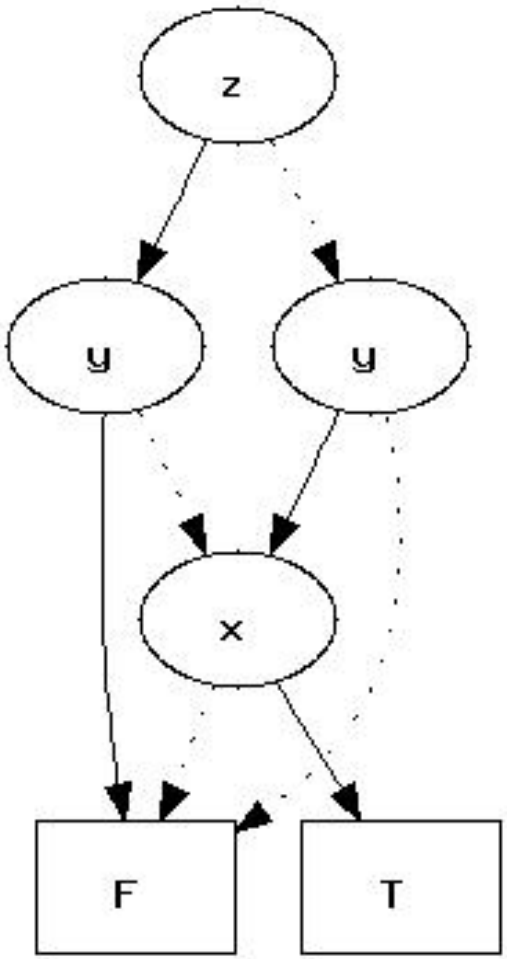
Non-CNF Based Techniques: BDDs

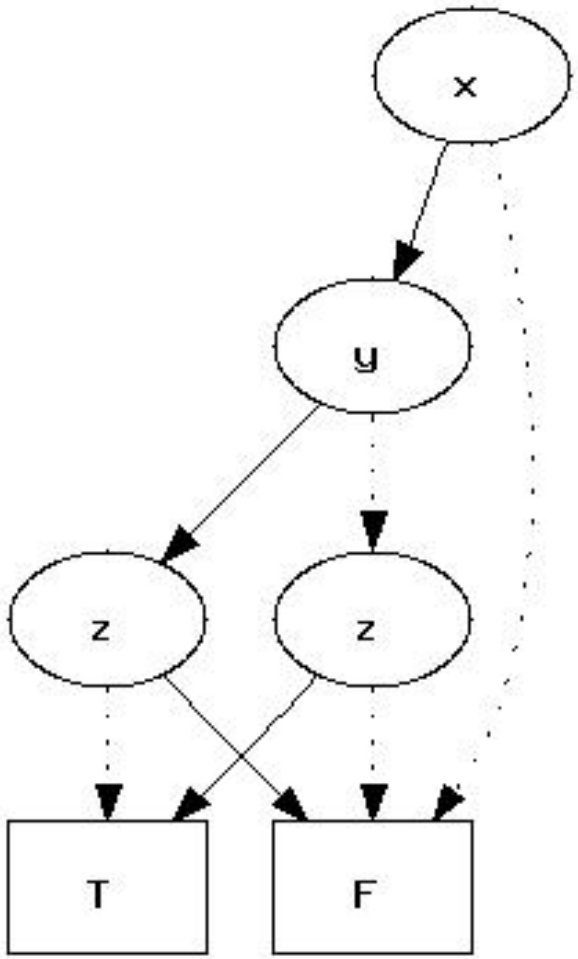
- Binary decision diagrams are an alternate way to CNF to express Boolean formulae.
- BDDs are binary trees with variables as nodes and true-false choices as edges. The leaves are True or False, giving the final output of the formula.
- Given an input order, there is a canonical way to reduce a BDD to a simpler form.
- Once one has a reduced BDD, determing satisfiability is easy. Just answer, "is there a path to the True leaf?"
- So why is it that BDD's don't give us a polynomial-time solution to SAT?

BDD Examples

You can construct your own examples using the BDD visualizer at:
`http://www.cs.uc.edu/~weaversa/BDD_Visualizer.html`

Two different BDD's for the forumla $x \wedge (y \oplus z)$





SBSAT: A BDD-based SAT Solver

- SBSAT (State-Based Satisfiability) is a non-clausal SAT solver based on state-machines and BDDs
- SBSAT is nice because it has an input language that does not require writing CNF.
- SBSAT's approach is markedly faster than a CDCL solver on some problems, but slower on others.
- SBSAT can translate its more intuitive language to CNF if you want to use a more standard solver like PicoSAT or Lingeling.
- SBSAT is availible for download at `http://www.cs.uc.edu/~weaversa/SBSAT.html`.

SBSAT: A BDD-based SAT Solver

- SBSAT (State-Based Satisfiability) is a non-clausal SAT solver based on state-machines and BDDs
- SBSAT is nice because it has an input language that does not require writing CNF.
- SBSAT's approach is markedly faster than a CDCL solver on some problems, but slower on others.
- SBSAT can translate its more intuitive language to CNF if you want to use a more standard solver like PicoSAT or Lingeling.
- SBSAT is availible for download at `http://www.cs.uc.edu/~weaversa/SBSAT.html`.

SBSAT: A BDD-based SAT Solver

- SBSAT (State-Based Satisfiability) is a non-clausal SAT solver based on state-machines and BDDs
- SBSAT is nice because it has an input language that does not require writing CNF.
- SBSAT's approach is markedly faster than a CDCL solver on some problems, but slower on others.
- SBSAT can translate its more intuitive language to CNF if you want to use a more standard solver like PicoSAT or Lingeling.
- SBSAT is availible for download at `http://www.cs.uc.edu/~weaversa/SBSAT.html`.

SBSAT: A BDD-based SAT Solver

- SBSAT (State-Based Satisfiability) is a non-clausal SAT solver based on state-machines and BDDs
- SBSAT is nice because it has an input language that does not require writing CNF.
- SBSAT's approach is markedly faster than a CDCL solver on some problems, but slower on others.
- SBSAT can translate its more intuitive language to CNF if you want to use a more standard solver like PicoSAT or Lingeling.
- SBSAT is availible for download at `http://www.cs.uc.edu/~weaversa/SBSAT.html`.

SBSAT: A BDD-based SAT Solver

- SBSAT (State-Based Satisfiability) is a non-clausal SAT solver based on state-machines and BDDs
- SBSAT is nice because it has an input language that does not require writing CNF.
- SBSAT's approach is markedly faster than a CDCL solver on some problems, but slower on others.
- SBSAT can translate its more intuitive language to CNF if you want to use a more standard solver like PicoSAT or Lingeling.
- SBSAT is availible for download at `http://www.cs.uc.edu/~weaversa/SBSAT.html`.

SBSAT: A BDD-based SAT Solver

- SBSAT (State-Based Satisfiability) is a non-clausal SAT solver based on state-machines and BDDs
- SBSAT is nice because it has an input language that does not require writing CNF.
- SBSAT's approach is markedly faster than a CDCL solver on some problems, but slower on others.
- SBSAT can translate its more intuitive language to CNF if you want to use a more standard solver like PicoSAT or Lingeling.
- SBSAT is availible for download at `http://www.cs.uc.edu/~weaversa/SBSAT.html`.

Solving Sudoku with SBSAT

- Instead of just CNF, SBSAT allows you to use lots of different logical operations: `and`, `or`, `eq`, `imp`, `xor`, `...`
- One great function in SBSAT is `minmax`.
- `minmax(m, n, x0, x1, ..., xn)` means a most `n` and at least `m` of the literals `x1, ...xn` are true.
- Using `minmax` it is easy to encode the rules of Sudoku puzzles.

Solving Sudoku with SBSAT

- Instead of just CNF, SBSAT allows you to use lots of different logical operations: `and`, `or`, `eq`, `imp`, `xor`, `...`
- One great function in SBSAT is `minmax`.
- `minmax(m, n, x0, x1, ..., xn)` means a most `n` and at least `m` of the literals `x1, ...xn` are true.
- Using `minmax` it is easy to encode the rules of Sudoku puzzles.

Solving Sudoku with SBSAT

- Instead of just CNF, SBSAT allows you to use lots of different logical operations: `and`, `or`, `eq`, `imp`, `xor`, `...`
- One great function in SBSAT is `minmax`.
- `minmax(m, n, x0, x1, ..., xn)` means a most `n` and at least `m` of the literals `x1, ...xn` are true.
- Using `minmax` it is easy to encode the rules of Sudoku puzzles.

Solving Sudoku with SBSAT

- Instead of just CNF, SBSAT allows you to use lots of different logical operations: `and, or, eq, imp, xor, ...`
- One great function in SBSAT is `minmax`.
- `minmax(m, n, x0, x1, ..., xn)` means a most `n` and at least `m` of the literals `x1, ...xn` are true.
- Using `minmax` it is easy to encode the rules of Sudoku puzzles.

Solving Sudoku with SBSAT

- Instead of just CNF, SBSAT allows you to use lots of different logical operations: `and, or, eq, imp, xor, ...`
- One great function in SBSAT is `minmax`.
- `minmax(m, n, x0, x1, ..., xn)` means a most `n` and at least `m` of the literals `x1, ...xn` are true.
- Using `minmax` it is easy to encode the rules of Sudoku puzzles.

A Ludicrously Difficult Sudoku: AI Escargot

This is the AI Escargot Sudoku puzzle:

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

To do this by hand requires an 8-deep backtrack. How fast can SBSAT solve it?

Extending SAT: SMT

- Sometimes we have contraint problems that involve non-Boolean notions
- Efficient decision procedures for these theories may exist
- The merger of these theory solvers with an underlying SAT solver is call SMT for SAT Modulo Theories
- Theories with such solvers include:
  - Bit vectors
  - Linear arithmetic
  - Arrays
  - Uninterpreted function
  - ...

SMT is a very active area of current research.

Extending SAT: SMT

- Sometimes we have contraint problems that involve non-Boolean notions
- Efficient decision procedures for these theories may exist
- The merger of these theory solvers with an underlying SAT solver is call SMT for SAT Modulo Theories
- Theories with such solvers include:
  - Bit vectors
  - Linear arithmetic
  - Arrays
  - Uninterpreted function
  - ...

SMT is a very active area of current research.

Extending SAT: SMT

- Sometimes we have contraint problems that involve non-Boolean notions
- Efficient decision procedures for these theories may exist
- The merger of these theory solvers with an underlying SAT solver is call SMT for SAT Modulo Theories
- Theories with such solvers include:
  - Bit vectors
  - Linear arithmetic
  - Arrays
  - Uninterpreted function
  - ...

SMT is a very active area of current research.

Extending SAT: SMT

- Sometimes we have contraint problems that involve non-Boolean notions
- Efficient decision procedures for these theories may exist
- The merger of these theory solvers with an underlying SAT solver is call SMT for SAT Modulo Theories
- Theories with such solvers include:
  - Bit vectors
  - Linear arithmetic
  - Arrays
  - Uninterpreted function
  - ...

SMT is a very active area of current research.

Extending SAT: SMT

- Sometimes we have contraint problems that involve non-Boolean notions
- Efficient decision procedures for these theories may exist
- The merger of these theory solvers with an underlying SAT solver is call SMT for SAT Modulo Theories
- Theories with such solvers include:
  - Bit vectors
  - Linear arithmetic
  - Arrays
  - Uninterpreted function
  - ...

SMT is a very active area of current research.

Extending SAT: SMT

- Sometimes we have contraint problems that involve non-Boolean notions
- Efficient decision procedures for these theories may exist
- The merger of these theory solvers with an underlying SAT solver is call SMT for SAT Modulo Theories
- Theories with such solvers include:
  - Bit vectors
  - Linear arithmetic
  - Arrays
  - Uninterpreted function
  - ...

SMT is a very active area of current research.

SMT Solver Examples: Z3

- Z3 is a very good SMT solver created at Microsoft Research.
- It is available for academic use for free at `z3.codeplex.com`.
- There is an online Z3 tutorial at `http://rise4fun.com/Z3Py/tutorial/guide`.

Other Front Ends

We have seen that SAT and SMT solvers can be quite difficult to write for directly. Nice front ends or APIs are very useful to have. Some are:

- SBV (SMT-Based Verification) is an open-source Haskell library that acts as a generic API for various SMT solvers
- ABC is a circuit synthesis tool out of Berkeley that uses AIGs (And-Inverter Graphs) instead of CNF to specifically solve certain types of SAT problems. It is excellent for proving functional equivalence.
- Cryptol is a domain-specific language for Cryptography that works as a front end for solvers to prove properties about crypto

I'm happy to demonstrate applications of each of these if there is time and interest.

Other Front Ends

We have seen that SAT and SMT solvers can be quite difficult to write for directly. Nice front ends or APIs are very useful to have. Some are:

- SBV (SMT-Based Verification) is an open-source Haskell library that acts as a generic API for various SMT solvers
- ABC is a circuit synthesis tool out of Berkeley that uses AIGs (And-Inverter Graphs) instead of CNF to specifically solve certain types of SAT problems. It is excellent for proving functional equivalence.
- Cryptol is a domain-specific language for Cryptography that works as a front end for solvers to prove properties about crypto

I'm happy to demonstrate applications of each of these if there is time and interest.

Other Front Ends

We have seen that SAT and SMT solvers can be quite difficult to write for directly. Nice front ends or APIs are very useful to have. Some are:

- SBV (SMT-Based Verification) is an open-source Haskell library that acts as a generic API for various SMT solvers
- ABC is a circuit synthesis tool out of Berkeley that uses AIGs (And-Inverter Graphs) instead of CNF to specifically solve certain types of SAT problems. It is excellent for proving functional equivalence.
- Cryptol is a domain-specific language for Cryptography that works as a front end for solvers to prove properties about crypto

I'm happy to demonstrate applications of each of these if there is time and interest.

Other Front Ends

We have seen that SAT and SMT solvers can be quite difficult to write for directly. Nice front ends or APIs are very useful to have. Some are:

- SBV (SMT-Based Verification) is an open-source Haskell library that acts as a generic API for various SMT solvers
- ABC is a circuit synthesis tool out of Berkeley that uses AIGs (And-Inverter Graphs) instead of CNF to specifically solve certain types of SAT problems. It is excellent for proving functional equivalence.
- Cryptol is a domain-specific language for Cryptography that works as a front end for solvers to prove properties about crypto

I'm happy to demonstrate applications of each of these if there is time and interest.

Other Front Ends

We have seen that SAT and SMT solvers can be quite difficult to write for directly. Nice front ends or APIs are very useful to have. Some are:

- SBV (SMT-Based Verification) is an open-source Haskell library that acts as a generic API for various SMT solvers
- ABC is a circuit synthesis tool out of Berkeley that uses AIGs (And-Inverter Graphs) instead of CNF to specifically solve certain types of SAT problems. It is excellent for proving functional equivalence.
- Cryptol is a domain-specific language for Cryptography that works as a front end for solvers to prove properties about crypto

I'm happy to demonstrate applications of each of these if there is time and interest.

The End

Questions?

iblumenfeld@cyberpointllc.com