Good but still Exp Algorithms for 3-SAT

Exposition by William Gasarch

Credit Where Credit is Due

This talk is based on parts of the following AWESOME books:

The Satisfiability Problem SAT, Algorithms and Analyzes
by
Uwe Schoning and Jacobo Torán

Exact Exponential Algorithms
by
Fedor Formin and Dieter Kratsch

This Lecture is Out of Order!

Typical Order of topics:

1. Define P, NP, NP-complete.
2. NP-complete means Probably Hard (see next slide).
3. Prove SAT is NP-complete
4. Show some other problems NP-complete
5. Boo :-(These NP-complete problems are hard!
6. OH- there are some things you can do about that: Approximations, clever techniques to make brute force a bit better (this talk).

Usually the last item is an afterthought in a course like this.
So why am I talking about this at the beginning of the NP-complete section?

This Lecture is Out of Order!

Typical Order of topics:

1. Define P, NP, NP-complete.
2. NP-complete means Probably Hard (see next slide).
3. Prove SAT is NP-complete
4. Show some other problems NP-complete
5. Boo :-(These NP-complete problems are hard!
6. OH- there are some things you can do about that: Approximations, clever techniques to make brute force a bit better (this talk).

Usually the last item is an afterthought in a course like this.
So why am I talking about this at the beginning of the NP-complete section?
1) Its the basis for Project 2
2) NP-completeness is often presented as the end of the story, I want to counter that.

PET Problems

One of the early names proposed for NP-complete problems (before NP-complete became standard) was *PET*-problems. Why? Now it stands for

Probably Exponential time

PET Problems

One of the early names proposed for NP-complete problems (before NP-complete became standard) was *PET*-problems. Why? Now it stands for

Probably Exponential time

If $P \neq NP$ is proven then it stands for

Provably Exponential time

PET Problems

One of the early names proposed for NP-complete problems (before NP-complete became standard) was *PET*-problems. Why? Now it stands for

Probably Exponential time

If $P \neq NP$ is proven then it stands for

Provably Exponential time

If $P = NP$ is proven then it stands for

Previously Exponential time

OUR GOAL

We will show algorithms for 3SAT that

1. Run in time $O(\alpha^n)$ for various $\alpha > 1$. Some will be randomized algorithms.
   NOTE: By $O(\alpha^n)$ we really mean $O(p(n)\alpha^n)$ where p is a poly. We ignore such factors.
2. Quite likely run even better in practice, or modifications of them do.

TRUE and FALSE in Formulas

Note: In terms of being satisfied:

$$(x_1 \vee x_2 \vee \mathit{FALSE}) \wedge (\neg x_1 \vee x_3) \equiv (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$$

Rule: *FALSE* can be removed. But see next example for caveat.

$$(\mathit{FALSE} \vee \mathit{FALSE} \vee \mathit{FALSE}) \wedge (\neg x_1 \vee x_3) \equiv \mathit{FALSE}$$

Rule: If all literals in a clause are *FALSE* then FALSE, so NOT satisfiable.

$$(x_1 \vee x_2 \vee \mathit{TRUE}) \wedge (\neg x_1 \vee x_3) \equiv (\neg x_1 \vee x_3)$$

Rule: If *TRUE* is in a claue the entire clause can be removed.

2SAT

2SAT is in P:

Look this up yourself

Convention For All of our Algorithms

Example:

$$(x_1) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_3)$$

Definition:

1. A *Unit Clause* is a clause with only one literal in it.
   Examples: (x_1) and $(\neg x_3)$.
2. A *Pure Literal* is a literal that only shows up as non negated or only shows up as negated.
   Examples: x_2 and $\neg x_4$
3. A *POS-Pure Literal* is a pure literal that is a variable.
   Example: x_2
4. A *NEG-Pure Literal* is a pure literal that is a negation of a var.
   Example: $\neg x_4$

STAND Alg

Input(ϕ, z) where z is a partial assignment. Output is either YES or NO or an easier equiv problem.

1. If every clause has ≤ 2 literals then run 2-SAT algorithm.
2. If ϕ has a unit clause $C = \{L\}$ then extend z by setting L to TRUE and output resulting formula and extended z.
3. If ϕ has POS-pure literal L then extend z by setting to L to TRUE and output resulting formula and extended z.
4. If ϕ has NEG-pure literal $\neg L$ then extend z by setting to L to FALSE and output resulting formula and extended z.
5. If every clause has a literal in it that is set to TRUE then output YES.
6. If there is some clause where every literal in it is set to FALSE then output NO.

We will use algorithm STAND in all of our algorithms.

DPLL ALGORITHM

DPLL (Davis-Putnam-Logemann-Loveland) ALGORITHM

ALG(F: 3CNF fml; z: Partial Assignment)

STAND(F, z) (Base case of the recursive algorithm.)
Pick a variable x (VERY CLEVERLY!)
ALG($F; z \cup \{x = T\}$) If outputs YES then output YES.
ALG($F; z \cup \{x = F\}$) If outputs YES then output YES,
otherwise output NO

Note: Variants will involve setting more than one variable.

Key Idea ONE Behind Recursive 7-ALG

Example: Given formula ϕ that has as one of its clauses

$$(x_1)$$

Then we KNOW that in a satisfying assignment cannot have

$$x_1 = F$$

So even brute force can be a bit clever by NOT trying any assignment that has $x_1 = F$
(This case will never come up since STAND will take care of it.)

Key Idea TWO Behind Recursive 7-ALG

Example: Given formula ϕ that has as one of its clauses

$$(x_1 \vee x_2)$$

Then we KNOW that in a satisfying assignment cannot have

$$x_1 = F, x_2 = F$$

So even brute force can be a bit clever by NOT trying any assignment that has $x_1 = F, x_2 = F$

Key Idea THREE Behind Recursive 7-ALG

Example: Given formula ϕ that has as one of its clauses

$$(x_1 \vee x_2 \vee \neg x_3)$$

Then we KNOW that in a satisfying assignment cannot have

$$x_1 = F, x_2 = F, x_3 = T$$

So even brute force can be a bit clever by NOT trying any assignment that has $x_1 = F, x_2 = F, x_3 = T$

Key Idea Behind Recursive 7-ALG: One Shortcut

Example: Given formula ϕ and a partial assignment z. We want to extend z to a satisfying assignment (or show we can't). If ϕ has a 2-clause:

$$(x_1 \vee \neg x_2)$$

So we will extend z by setting (x_1, x_2) to all possibilities EXCEPT

$$x_1 = F, x_2 = T$$

If there is a 2-clause then better to use it.

Recursive-7 ALG

ALG(F: 3CNF fml; z: Partial Assignment)

STAND
Two Cases:
    (1) Exists a 2-clause: Case 1, next slide.
    (2) All 3-clauses: Case 2, nextnext slide
Next Two slides.

Recursive-7 ALG: Case 1

There is a clause $C = (L_1 \vee L_2)$
Let z_1, z_2, z_3 be the 3 ways
to set (L_1, L_2) so that C is true
ALG$(F; z_1)$ If returns YES, then YES.
ALG$(F; z_2)$ If returns YES, then YES.
ALG$(F; z_3)$ If returns YES, then YES,
else NO.

Note: In this case get $T(n) = 3T(n-2)$.

Bounding the Recurrence

$T(1) = 1$ if only one var then easy to check if SAT or not

$$T(n) = 3T(n-3)$$

GUESS that $T(n) = \alpha^n$ for some α

$$\alpha^n = 3\alpha^{n-3}$$

$$\alpha^3 = 3$$

$\alpha = \sqrt{3} \sim 1.73$
SO

$$T(n) = O((\sqrt{3})^n) \sim O((1.73)^n).$$

But only if always find a 2-clause. Unlikely.

Recursive-7 ALG: Case 2

There is a clause $C = (L_1 \vee L_2 \vee L_3)$
Let $z_1, \ldots, z_7$ be the 7 ways
to set (L_1, L_2, L_3) so that C is true

ALG$(F; z_1)$ If returns YES, then YES.
ALG$(F; z_2)$ If returns YES, then YES.
ALG$(F; z_3)$ If returns YES, then YES.
ALG$(F; z_4)$ If returns YES, then YES.
ALG$(F; z_5)$ If returns YES, then YES.
ALG$(F; z_6)$ If returns YES, then YES.
ALG$(F; z_7)$ If returns YES, then YES,
else NO.

Note: In this case get $T(n) = 7T(n-3)$. If always did this $T(n) = (7^{1/3})^n \sim (1.91)^n$. Leave it to you to derive that. It might be on the final.

GOOD NEWS/BAD NEWS

1. Good News: BROKE the 2^n barrier. Hope for the future!
2. Bad News: Still not that good a bound.
3. Good News: Similar ideas gets time to $O((1.84)^n)$.
4. Bad News: Still not that good a bound.
5. Good News: The above algorithm is basis for your project 2.
6. Bad News: The rest of this talk is not used for your project 2.

COOL Partial Assignments

Definition: If F is a fml and z is a partial assignment then z is COOL if every clause that z affects is made TRUE.

Example:

$$(x_1) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3)$$

$(x_2, x_4) = (F, F)$ is COOL: clauses that have any of $\{x_2, x_4\}$ are TRUE.

Lemma One about Coolness

Lemma One: Let F be a 3CNF fml and z be a partial assignment. If z is COOL then $F \in 3SAT$ iff $F(z) \in 3SAT$.

Example:

$$(x_1) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3)$$

$(x_2, x_4) = (F, F)$ is COOL: clauses that have any of $\{x_2, x_4\}$ are TRUE.

Plug in $(x_2, x_4) = (F, F)$ to get

$$(x_1) \wedge (TRUE) \wedge (TRUE) \wedge (\neg x_3)$$

$$= x_1 \wedge \neg x_3$$

Now need this to be in SAT.

Point of Lemma One

Assume z is COOL.

If ϕ is satisfiable then there is a satisfying assignment that is an extension of z.

Lemma Two about Coolness

Lemma Two: Let F be a 3CNF fml and z be a partial assignment. If z is NOT COOL then $F(z)$ will have a clause of length 2.

Example:

$$(x_1) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3)$$

$(x_2, x_4) = (F, T)$ is NOT COOL: The clause $\neg x_1 \vee x_2 \vee x_4$ is affected and not made TRUE (also not made FALSE)

Plug in $(x_2, x_4) = (F, T)$ to get

$$(x_1) \wedge (\neg x_1 \vee \mathit{FALSE} \vee \neg \mathit{TRUE}) \wedge (\neg \mathit{FALSE} \vee x_3 \vee \mathit{TRUE}) \wedge (\neg x_3)$$

$$(x_1) \wedge (\neg x_1) \wedge (\mathit{TRUE} \vee x_3 \vee \mathit{TRUE}) \wedge (\neg x_3)$$

$$(x_1) \wedge (\neg x_1) \wedge (\neg x_3)$$

KEY: The clause that z did not make true now has LESS literals.

Point of Lemma Two

Assume z is NOT COOL.

If use z then some of the clauses become 2-clauses, making formula simpler.

Recursive-3 ALG MODIFIED MORE

ALG(F: 3CNF fml, z: partial assignment)

STAND
if $(\exists C = (L_1 \vee L_2)$ then
$z1 = z \cup \{L_1 = T\})$
if $z1$ is COOL then ALG$(F; z1)$
else
$z01 = z \cup \{L_1 = F, L_2 = T\})$
if $z01$ is COOL then ALG$(F; z01)$
else
ALG$(F; z1)$ (will have 2−clauses)
ALG$(F; z01)$ (will have 2−clauses)
else (COMMENT: The ELSE is on next slide.)

Recursive-3 ALG MODIFIED MORE

(COMMENT: This slide is when a 3CNF clause not is satisfied.)

if $(\exists C = (L_1 \vee L_2 \vee L_3)$ then
$z1 = z \cup \{L_1 = T\})$
if $z1$ is COOL then ALG$(F; z1)$

else
$z01 = z \cup \{L_1 = F, L_2 = T\})$
if $z01$ is COOL then ALG$(F; z01)$

else
$z001 = z \cup \{L_1 = F, L_2 = F, L_3 = T\})$
if $z001$ is COOL then ALG$(F; z001)$

else
ALG$(F; z1)$
ALG$(F; z01)$
ALG$(F; z001)$

IS IT BETTER?

VOTE: IS THIS BETTER THAN $O((1.84)^n)$?

IS IT BETTER?

VOTE: IS THIS BETTER THAN $O((1.84)^n)$?
IT IS!

IT IS BETTER!

KEY1: If any of $z1$, $z01$, $z001$ are COOL then only ONE recursion: $T(n) = T(n-1) + O(1)$.

KEY2: If NONE of the $z1$, $z01$ $z001$ are COOL then ALL of the recurrences are on fml's with a 2CNF clause in it.

$T(n)$= Time alg takes on 3CNF formulas.

$T'(n)$= Time alg takes on 3CNF formulas that have a 2CNF in them.

$T(n) = \max\{T(n-1), T'(n-1) + T'(n-2) + T'(n-3)\}$.

$T'(n) = \max\{T(n-1), T'(n-1) + T'(n-2)\}$.

Can show that worst case is:

$T(n) = T'(n-1) + T'(n-2) + T'(n-3)$.

$T'(n) = T'(n-1) + T'(n-2)$.

The Analysis

$T'(0) = O(1)$
$T'(n) = T'(n-1) + T'(n-2).$

$$T'(n) = O((1.618)^n).$$

So

$$T(n) = O(T'(n)) = O((1.618)^n).$$

VOTE: Is better known?
VOTE: Is there a proof that *these techniques* cannot do any better?

The Analysis

$T'(0) = O(1)$
$T'(n) = T'(n-1) + T'(n-2).$

$$T'(n) = O((1.618)^n).$$

So

$$T(n) = O(T'(n)) = O((1.618)^n).$$

VOTE: Is better known?
VOTE: Is there a proof that *these techniques* cannot do any better?
We will do better, but not clear if using same techniques

Hamming Distances

Definition If x, y are assignments then $d(x, y)$ is the number of bits they differ on.

KEY TO NEXT ALGORITHM: If F is a fml on n variables and F is satisfiable then either

1. F has a satisfying assignment z with $d(z, 0^n) \leq n/2$, or
2. F has a satisfying assignment z with $d(z, 1^n) \leq n/2$.

HAM ALG

HAMALG(F: 3CNF fml, z: full assignment, h: number) h bounds $d(z, s)$ where s is SATisfying assignment

STAND

if $\exists C = (L_1 \vee L_2)$ not satisfied then

$\quad$ ALG$(F; z \oplus \{L_1 = T\}; h-1\}$

$\quad$ ALG$(F; z \oplus \{L_1 = F, L_2 = T\}; h-2)$

if $\exists C = (L_1 \vee L_2 \vee L_3)$ not satisfied then

$\quad$ ALG$(F; z \oplus \{L_1 = T\}; h-1)$

$\quad$ ALG$(F; z \oplus \{L_1 = F, L_2 = T\}; h-2)$

$\quad$ ALG$(F; z \oplus \{L_1 = F, L_2 = F, L_3 = T\}; h-3)$

REAL ALG

HAMALG(F; 0^n; $n/2$)
If returned NO then HAMALG(F; 1^n; $n/2$)

VOTE: IS THIS BETTER THAN $O((1.61)^n)$?

REAL ALG

HAMALG(F; 0^n; $n/2$)
If returned NO then HAMALG(F; 1^n; $n/2$)

VOTE: IS THIS BETTER THAN $O((1.61)^n)$?
IT IS NOT! It is $O((1.73)^n)$.

KEY TO HAM

KEY TO HAM ALGORITHM: Every element of $\{0,1\}^n$ is within $n/2$ of either 0^n or 1^n

Definition: A *covering code of* $\{0,1\}^n$ *of SIZE* s *with RADIUS* h is a set $S \subseteq \{0,1\}^n$ of size s such that

$$(\forall x \in \{0,1\}^n)(\exists y \in S)[d(x,y) \leq h].$$

Example: $\{0^n, 1^n\}$ is a covering code of SIZE 2 of RADIUS $n/2$.

ASSUME ALG

Assume we have a Covering code of $\{0,1\}^n$ of size s and radius h. Let Covering code be $S = \{v_1, \ldots, v_s\}$.

$i = 1$
FOUND=FALSE
while (FOUND=FALSE) and ($i \leq s$)
    HAMALG($F; v_i; h$)
    If returned YES then FOUND=TRUE
        else
            $i = i + 1$
end while

ANALYSIS OF ALG

Each iteration satisfies recurrence
$T(0) = 1$
$T(h) = 3T(h-1)$
$T(h) = 3^h$.
And we do this s times.
ANALYSIS: $O(s3^h)$.
Need covering codes with small value of $O(s3^h)$.

IN SEARCH OF A GOOD COVERING CODE

RECAP: Need covering codes of size s, radius h, with small value of $O(s3^h)$.

IN SEARCH OF A GOOD COVERING CODE

RECAP: Need covering codes of size s, radius h, with small value of $O(s3^h)$.
THAT'S NOT ENOUGH: We need to actually CONSTRUCT the covering code in good time.

IN SEARCH OF A GOOD COVERING CODE

RECAP: Need covering codes of size s, radius h, with small value of $O(s3^h)$.
THAT'S NOT ENOUGH: We need to actually CONSTRUCT the covering code in good time.
YOU'VE BEEN PUNKED: We'll just pick a RANDOM subset of $\{0,1\}^n$ and hope that it works.

IN SEARCH OF A GOOD COVERING CODE- RANDOM!

CAN find with high prob a covering code with

- Size $s = n^2 2^{.4063n}$
- Distance $h = 0.25n$.

Can use to get SAT in $O((1.5)^n)$.
Note: Best known: $O((1.306)^n)$.

Summary

1. There is an $O((1.913)^n)$ alg for 3-SAT.
2. There is an $O((1.84)^n)$ alg for 3-SAT.
3. There is an $O((1.618)^n)$ alg for 3-SAT.
4. There is an $O((1.306)^n)$ alg for 3-SAT (randomized).

1. These algorithms are for 3-SAT so not really used.
2. Similar ones ARE used in the real world.
3. There are some AWESOME SAT-Solvers in the real world.
4. Confronted with an NP-complete problem one strategy is to reduce it to a SAT problem and use a SAT-solver.

Relevant to Ontologix?

(I gave this talk to a SAT-solving company, Ontologix.)
Relevant: These algorithms work better in practice then their worst case run-times.

Not Relevant: The real world is k-Sat, not 3-Sat.

Relevant: Good to get new ideas and see how other people think about things (kind of the whole purpose of my visit!)