Good but still Exp Algorithms for 3SAT

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 Unusual!

Typical 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 Unusual!

Typical 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?
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 $1 < \alpha < 2$. 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:

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

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

$$(FALSE \vee FALSE \vee FALSE) \wedge (\neg x_1 \vee x_3) \equiv FALSE$$

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

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

Rule: If *TRUE* is in a clause 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 2SAT 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 L to TRUE and output resulting formula and extended z.
4. If ϕ has NEG-Pure literal $\neg L$ then extend z by setting 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(ϕ: 3-CNF fml; z: Partial Assignment)

```
STAND(φ, z) (Base case of the recursive algorithm.)
 Pick a variable x (VERY CLEVERLY!)
ALG(φ; z ∪ {x = T}) If outputs YES then output YES.
ALG(φ; z ∪ {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(ϕ: 3-CNF 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 = (L1 ∨ L2)
Let z1, z2, z3 be the 3 ways
          to set (L1, L2) so that C is true
    ALG(φ; z1) If returns YES, then YES.
    ALG(φ; z2) If returns YES, then YES.
    ALG(φ; z3) 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

$$T(n) = 3T(n-2)$$

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

$$\alpha^n = 3\alpha^{n-2}$$

$$\alpha^2 = 3$$

$\alpha = \sqrt{3} \sim 1.73$
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 ∨ L_2 ∨ L_3)
Let z_1, ..., z_7 be the 7 ways
          to set (L_1, L_2, L_3) so that C is true
     ALG(φ; z_1) If returns YES, then YES.
     ALG(φ; z_2) If returns YES, then YES.
     ALG(φ; z_3) If returns YES, then YES.
     ALG(φ; z_4) If returns YES, then YES.
     ALG(φ; z_5) If returns YES, then YES.
     ALG(φ; z_6) If returns YES, then YES.
     ALG(φ; 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 get time to $O((1.84)^n)$.
4. Bad News: Still not that good a bound.

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 ϕ is a fml on n variables and ϕ is satisfiable then either

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

HAM ALG

HAMALG(ϕ: 3-CNF 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$(\phi; z \oplus \{L_1 = T\}; h - 1\}$

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

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

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

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

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

REAL ALG

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

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

REAL ALG

HAMALG(ϕ ; 0^n ; $n/2$)
If returned NO then HAMALG(ϕ ; 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 ≤ s)
    HAMALG(φ; 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 3SAT.
2. There is an $O((1.84)^n)$ alg for 3SAT.
3. There is an $O((1.618)^n)$ alg for 3SAT.
4. There is an $O((1.306)^n)$ alg for 3SAT (randomized).

1. These algorithms are for 3SAT 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 than their worst case run-times.

Not Relevant: The real world is kSAT, not 3SAT.

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

SATisfiable?

The AND of the following:

1. $x_{11} \vee x_{12}$
2. $x_{21} \vee x_{22}$
3. $x_{31} \vee x_{32}$
4. $\neg x_{11} \vee \neg x_{21}$
5. $\neg x_{11} \vee \neg x_{31}$
6. $\neg x_{21} \vee \neg x_{31}$
7. $\neg x_{12} \vee \neg x_{22}$
8. $\neg x_{12} \vee \neg x_{32}$
9. $\neg x_{22} \vee \neg x_{32}$

SATisfiable?

The AND of the following:

1. $x_{11} \vee x_{12}$
2. $x_{21} \vee x_{22}$
3. $x_{31} \vee x_{32}$
4. $\neg x_{11} \vee \neg x_{21}$
5. $\neg x_{11} \vee \neg x_{31}$
6. $\neg x_{21} \vee \neg x_{31}$
7. $\neg x_{12} \vee \neg x_{22}$
8. $\neg x_{12} \vee \neg x_{32}$
9. $\neg x_{22} \vee \neg x_{32}$

This is Pigeonhole Principle: x_{ij} is putting ith pigeon in j hole!

SATisfiable?

The AND of the following:

1. $x_{11} \vee x_{12}$
2. $x_{21} \vee x_{22}$
3. $x_{31} \vee x_{32}$
4. $\neg x_{11} \vee \neg x_{21}$
5. $\neg x_{11} \vee \neg x_{31}$
6. $\neg x_{21} \vee \neg x_{31}$
7. $\neg x_{12} \vee \neg x_{22}$
8. $\neg x_{12} \vee \neg x_{32}$
9. $\neg x_{22} \vee \neg x_{32}$

This is Pigeonhole Principle: x_{ij} is putting ith pigeon in j hole!
Can't put 3 pigeons into 2 holes!