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

Other Sources

Sources on SAT Solvers

1. The Satisfiability Problem SAT, Algorithms and Analyzes by Uwe Schoning and Jacobo Torán. Available from UMCP Library as ebook.
2. Exact Exponential Algorithms by Fedor Formin and Dieter Kratsch. See here:
   `ExactExpAlg.pdf`
3. Algorithms for the Satisfiability Problem, PhD thesis by Rolf:
   `SATRolf.pdf`
4. Algorithms for the Satisfiability Problem, Book by Franco and Weaver:
   `SATFW.pdf`

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.

T and F in Formulas

Note In terms of being satisfied:

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

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

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

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

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

Rule: If T is in a clause the entire clause can be removed.

Standard Format for Formulas and Many Examples

DIMACS has run several SAT SOLVING competitions.
They use a standard format for formulas and have LOTS of formulas for benchmarks.
If you are coding up SAT SOLVERS, use their format and try your algorithms on their examples.
See
`https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

2SAT

2SAT is in P:

Look this up yourself

1. `https://cp-algorithms.com/graph/2SAT.html`
2. Look up Kosaraju's Algorithm on Wikipedia.

Convention For All of our Algorithms

Example

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

Def

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 $\neg x_4$

Setting Variables

If you set some x to T then also do the following.

1. Set all occurrence of x to T.
2. Set all occurrence of $\neg x$ to F.

Setting Variables

If you set some x to T then also do the following.

1. Set all occurrence of x to T.
2. Set all occurrence of $\neg x$ to F.

If you set some x to F then also do the following.

1. Set all occurrence of x to F.
2. Set all occurrence of $\neg x$ to T.

Program CHECK

CHECK

Program CHECK

CHECK

1. Input is a formula which may have T and F in it. So for example

$$(x \vee y \vee F) \wedge (\neg x \vee w \vee T)$$

could be an input.

Program CHECK

CHECK

1. Input is a formula which may have T and F in it. So for example

$$(x \vee y \vee F) \wedge (\neg x \vee w \vee T)$$

   could be an input.
2. If a clause has ALL F's then return F, the entire formula is not satisfied. Note that this only indicates that this particular way to satisfy the formula failed, there may be another way.

Program CHECK

CHECK

1. Input is a formula which may have T and F in it. So for example

$$(x \vee y \vee F) \wedge (\neg x \vee w \vee T)$$

could be an input.

2. If a clause has ALL F's then return F, the entire formula is not satisfied. Note that this only indicates that this particular way to satisfy the formula failed, there may be another way.
3. If a clause has a literal set to T then GET RID of that clause, it is already satisfied. If there are now no more clauses left then return T, The formula IS satisfiable!

Program CHECK

CHECK

1. Input is a formula which may have T and F in it. So for example

$$(x \vee y \vee F) \wedge (\neg x \vee w \vee T)$$

could be an input.
2. If a clause has ALL F's then return F, the entire formula is not satisfied. Note that this only indicates that this particular way to satisfy the formula failed, there may be another way.
3. If a clause has a literal set to T then GET RID of that clause, it is already satisfied. If there are now no more clauses left then return T, The formula IS satisfiable!

Next slides is examples.

Examples of CHECK

Example: $(T \vee y \vee z) \wedge (\neg x \vee w)$

Examples of CHECK

Example: $(T \vee y \vee z) \wedge (\neg x \vee w)$
Output $(\neg x \vee w)$

Examples of CHECK

Example: $(T \vee y \vee z) \wedge (\neg x \vee w)$
Output $(\neg x \vee w)$

Tricky Example: $(F \vee F \vee \neg F) \wedge (T \vee y \vee w)$

Examples of CHECK

Example: $(T \vee y \vee z) \wedge (\neg x \vee w)$
Output $(\neg x \vee w)$

Tricky Example: $(F \vee F \vee \neg F) \wedge (T \vee y \vee w)$
Return T. The first clause has $\neg F \equiv T$. The second clause has a T.

Warning About CHECK

CHECK will return either

- T
- F
- A formula (shorter than the original).

This can be tricky that there are two diff kinds of outputs.

STAND Alg

Input(ϕ, z) where z is a partial assignment. Output is either T or F or a equiv simplified formula.

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 EVERY occurrence of L to T.
3. If ϕ has POS-Pure literal L then extend z by setting EVERY occurrence of L to T.
4. If ϕ has NEG-Pure literal $\neg L$ then extend z by setting EVERY occurrence of L to F.
5. Run CHECK. If CHECK returned T or F then return that value.
6. (CHECK did not return T or F) If the formula we have now is DIFF from the input, then run STAND on the simplified formula. (So keep reducing by STAND until you can't.)

We will use algorithm STAND in all of our algorithms.

Warning About STAND

STAND will return either

- T
- F
- A formula (usually shorter than the original).

Note that there are two diff kinds of outputs so your program needs to take that into account.

DPLL Alg Template

The next slide does not have an algorithm.

DPLL Alg Template

The next slide does not have an algorithm.

It has an Algorithm Template.

DPLL Alg Template

The next slide does not have an algorithm.

It has an Algorithm Template.

There is a curious line in it:

Pick a variable x (VERY CLEVERLY!)

DPLL Alg Template

The next slide does not have an algorithm.

It has an Algorithm Template.

There is a curious line in it:

Pick a variable x (VERY CLEVERLY!)

Roughly speaking
All SAT Solving algorithms find different ways to be clever

DPLL Alg Template

DPLL (Davis-Putnam-Logemann-Loveland) Alg Template

ALG(ϕ: 3-CNF fml; z: Partial Assignment)

STAND(ϕ, z) (Base case of the recursive algorithm.)
Pick a variable x (VERY CLEVERLY!)
ALG($\phi; z \cup \{x = T\}$) If outputs T then output T.
ALG($\phi; z \cup \{x = F\}$) If outputs T then output T,
otherwise output F

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 T, then T.
     ALG(φ; z2) If returns T, then T.
     ALG(φ; z3) If returns T, then T,
                    else F.
```

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

Bounding the Recurrence

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

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

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 = (L1 ∨ L2 ∨ L3)
Let z1,...,z7 be the 7 ways
          to set (L1, L2, L3) so that C is true
    ALG(φ; z1) If returns T, then T.
    ALG(φ; z2) If returns T, then T.
    ALG(φ; z3) If returns T, then T.
    ALG(φ; z4) If returns T, then T.
    ALG(φ; z5) If returns T, then T.
    ALG(φ; z6) If returns T, then T.
    ALG(φ; z7) If returns T, then T,
                    else F.
```

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

Def 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 ∃C = (L1 ∨ L2 ∨ L3) not satisfied then
        HAMALG(φ; z ⊕ {L1 = T}; h − 1)
        HAMALG(φ; z ⊕ {L2 = T}; h − 1)
        HAMALG(φ; z ⊕ {L3 = T}; h − 1)
```

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

Def 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

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

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=F
while (FOUND=F) and (i ≤ s)
    HAMALG(φ; v_i; h)
    If returned T then FOUND=T
        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!