







I Can't Get No (Boolean) Satisfaction

 Shairoz Sohail

Follow

 21 min read · 3 days ago

 76 

A deep dive into satisfiability with Python

These days, there is a lot of interest in math problems that are “easy to state, but hard to solve.” This is natural — such problems can be approached

without years of specialized education and gently introduce newcomers to the rich complexity that can arise from very simple structures. However, once the initial curiosity wears off, people usually start with the obvious question — “how is this applicable?”

Enter boolean satisfiability (SAT for short, pronounced as one word) — a problem that is simultaneously **easy to state, hard to solve, and enormously applicable**. In fact, the problem is so applicable that it lies at the heart of some of our biggest questions in mathematics, computer science, and philosophy. A world where someone has come up with a way to (efficiently) solve big SAT problems would be a very different world indeed.

Not only do SAT problems encode many difficult problems from seemingly unrelated domains (such as Sudoku, or vehicle routing, or protein folding), but they also seem to demonstrate some of the perceived limitations of computation— specifically in how efficiently a problem can be solved. SAT problems have led many researchers to believe that these limits are somehow fundamental— that certain problems have a lower bound on how difficult they are, no matter how clever we get.



In this article, we’re going to demonstrate why SAT matters and provide a (technical) deep dive into the world of SAT generators and solvers using the Python programming language.

SAT Problems in Disguise

A number of problems in the real world are, at their core, **constraint-satisfaction problems (CSPs)**. Often there are many possible candidates, and

a handful of difficult constraints that narrow down which of these candidates are viable. As this ratio increases (the number of potential candidates vs. the candidates satisfying the constraints), these problems become very difficult.

In some situations, we can easily reduce the search space to look through only a small set of candidates. Think of assigning people jobs in a factory — only people with a driver's license could be transporters. At a football try out, it only makes sense to consider fast runners for the wide receiver position. But without such qualifying information, the assignment task becomes an arduous game of trial and error. This “qualifying information” is usually in the form of some easy-to-compute rules (i.e yes/no has a driver's license, how fast can you run a quarter mile etc.). Sometimes we find such rules in numerical problems, but often times the problem setting just seems too general for such rules to exist.

Most problems of this kind (CSPs) can be encoded into SAT problems. The number of CSPs in the real world is immense, here is a sampling of some personal favorites:

- Sorting a disordered stack of pancakes so that a spatula can be inserted at any point in the stack and used to flip all pancakes above it (Bill Gates has worked on this problem!)
- Finding the remaining mines in a game of Minesweeper that is partially (>25%) complete
- Bitcoin mining
- Playing a perfect game of Super Mario Bros

Of course, these problems have something else in common besides just an ability to be stated in terms of a SAT problem, something we will discuss more in the section on P vs. NP. For now, I hope the title has at least been partially justified.

Now let's lift the veil on what this mysterious, all-encompassing SAT problem actually looks like.

Boolean Equations and Satisfiability

Boolean variables

Before having a reasonable discussion of Boolean equations, we must first define Boolean variables. Many kinds of variables can exist in mathematics, from those taking fixed (but unknown) values in the real or complex plane, to those taking on values that are themselves functions, or sets, or even probability distributions. However, boolean variables can only take on two values: true, or false. That's it. Every time a boolean variable is referenced here (with an uppercase letter), think of it taking a fixed, but unknown, value of 1 (true) or 0 (false).

Boolean Operators

Once we have variables, we need to define a set of operators between those variables and generate a relation table to show what the outcomes of those operators would be. Instead of the usual summation and multiplication operators we have in algebra, we instead consider the “and”, “or”, and “xor” operators. Unlike with real-valued variables (where the possible values they can take on is infinite), we can write out the entire table of outcomes for these operations on boolean variables:

A AND B	B = True	B = False
A = True	True	False
A = False	False	False

A OR B	B = True	B = False
A = True	True	True
A = False	True	False

Boolean Equations

Now that we have variables and operators, we can move on to some additional terminology. Consider two boolean variables (A,B) , joined by an “or” operator, and the result joined by an “and” operator against a third variable C:

$$(A \vee B) \wedge C$$

(A or B) and C

This entire statement is called a conjunction, since the statements on the left and right are joined by an “and” operator.

We can modify the above by switching the “and” to an “or” then adding an “and” statement to the C variable (and taking care to place parenthesis in the right place). Now, the statement on the right is called a “disjunction” since it uses an “or” operator, and the statement on the left is also a “disjunction.”

The equation as a whole is considered a “conjunction” since it is joined by an “and”:

$$(A \wedge B) \vee (C \wedge \neg B)$$

(A and B) or (C and not B)

Just remember, “and” = **conjunction** and “or” = **disjunction**.

Satisfiability

The equation above is satisfied when we pick true/false values for each of the variables (A, B, C) such that the entire statement is true. Let's take a random guess and see if it works — we will set

- A = True
- B = False
- C = False.

Then we will plug into the above equation and gradually reduce, first taking care of the negation signs, then taking care of the parenthesized statements (using the table above), then dealing with the conjunction.

$$\begin{aligned}
 & (True \wedge False) \vee (False \wedge not False) \\
 &= (True \wedge False) \vee (False \wedge True) \\
 &= (False) \vee (False) \\
 &= False
 \end{aligned}$$

So that assignment doesn't render the equation True, and doesn't "satisfy" it. One assignment that does is (A = True, B = False, C = True) — see if you could show (prove) this is the case using the same process as above. An additional bit of definition before we continue, a satisfying assignment of boolean variables is usually called a "**model**" for the statement.

CNF Form and K-SAT

Given the wealth of ways to arrange SAT statements, it is useful to consider a standard form to analyze and solve these statements. The CNF (**Conjunctive Normal Form**) is just the ticket, mostly.

In CNF, we have a set of disjunctions (usually on individual lines and with the same number of variables each), all joined as a disjunction. Here's an example:

$$\begin{aligned}
 & (A \vee B \vee C) \wedge \\
 & (\neg B \vee C \vee \neg A) \wedge \\
 & (\neg C \vee \neg A \vee B)
 \end{aligned}$$

The number of variables that appears in each conjunct is the “K” in K-SAT. So when you see a 3-SAT problem it means that (in CNF form) there is 3 variables that appear in each conjunct.

Two things to note before continuing — (1) some authors will use K-SAT to mean that **at most** K variables that appear in each conjunct and (2) it is important to remember that the “K” refers to the number of variables in each conjunct and **NOT** the total number of variables, which is often much higher. To specify a full set of boolean equations in CNF form, we can say:

“This is a K-SAT instance with N total variables and C conjuncts.”

Generating Random K-SAT Instances

The next step is to get our hands dirty and actually generate some K-SAT problems to solve. For this task, we’re going to be using the Python programming language along with the EasySAT library. (written specifically for this article).

Once you’re up and running with the library, you can use the following piece of code to generate a random 3-SAT problem with 5 literals/variables and 15 conjuncts:

```
from easysat.generators import KSAT_Generator

ksat_generator = KSAT_Generator()
ksat_generator.random_kcnf(k = 3, n_literals = 5, n_conjuncts=15)
```


which should return something that looks like the following (your numbers will be different):

```
[[3, -3, -5],  
 [5, 1, 4],  
 [5, -1],  
 [2, -3],  
 [-4, 4],  
 [3, -5],  
 [1, 5, -3],  
 [5, 1],  
 [1, 5, 2],  
 [5, -1],  
 [2, -1],  
 [5, 2],  
 [1, 2, -4],  
 [1, 2],  
 [1, -3, 4]]
```

This is the standard DIMACS form for representing a KSAT problem in CNF form in text format. The format is essentially a list of list, with each integer standing in for a boolean variable and the negative sign used for negation. Notice since we specified a $K = 3$, there are 3 literals per conjunct (per sublist in DIMACS format). Additionally, there are 5 unique boolean variables and 15 total sublists for the 15 conjuncts we specified.

While a 15 conjunct, 5 literal 3SAT problem is not impossible to solve by hand, remember there will be $2^5 = 32$ possibilities to check, and for each check you need to evaluate 15 expressions. This sounds more like a punishment than an exercise in learning. Luckily, we have a computer that excels at brute force tasks like this.

Brute Force Solver

In the code below, we generate a 3SAT problem just as above, but then import the brute force solver and use it to quickly solve the instance — the solver provides a True/False response to whether the instance is solvable, a “model” (satisfying assignment of the literals) if it is, and the number of assignments it tried before stopping.

```
from easysat.generators import KSAT_Generator
from easysat.solvers import BruteForce

ksat_generator = KSAT_Generator()
sample = ksat_generator.random_kcnf(k = 3, n_literals = 5, n_conjuncts=15)

BFS_solver = BruteForce()
BFS_solver.append_formula(sample)
sat, model, assigns = BFS_solver.solve()

print("Satisfiable? ", sat)
print("Total assignments tested: ", assigns)
print("Satisfying assignment: ", model)
```

Your results may vary, but for this particular run the response was:

```
Satisfiable? True
Total assignments tested: 10
Satisfying assignment: [-1, 2, 3, -4, 5]
```

Now, try the above code with the following settings for `ksat_generator.random_kcnf`:

- `k = 3, n_literals = 10, n_conjuncts = 40`

- $k = 3, n_{\text{literals}} = 15, n_{\text{conjuncts}} = 60$
- $k = 3, n_{\text{literals}} = 20, n_{\text{conjuncts}} = 90$

You may start to understand that brute force solving doesn't scale very well (that is, if the last case finishes for you in any reasonable amount of time). We need a more sophisticated solving strategy for even moderately sized cases.

Unit Propagation and Pure Literals

Suppose that instead of guessing a whole model (that is, a value for each variable), we instead guess the value of a single variable at a time, say $A = \text{True}$.

If we do this kind of sequential solving, we need a way to check the guess of our variables one at a time, which we haven't discussed. However, it's quite simple. Say we have the following conjunct formula:

$$(\neg A \vee B \vee \neg C)$$

After setting $A = \text{True}$ (the opposite polarity it appears with in the formula), we can eliminate that negative A from the equation since it is no longer an option to satisfy the conjunct with:

$$(B \vee \neg C)$$

If we now guess $B = \text{False}$, that also eliminates that from the formula:

$$\neg C$$

This leaves us with a single variable to satisfy the formula, C . Since C appears in the negative (and we have already guessed wrong for A and B), we know that the only way to satisfy this conjunct formula would be to set C to False.

If we instead set C to True, then we must also eliminate it from this conjunct and this results in an **empty clause**. An empty clause is bad, because it means either we did something wrong or the formula is unsatisfiable.

The technique above is called **unit propagation** — when there is only one variable left to satisfy a conjunct, we know how we must set it. Thus we usually take care of these first and foremost (of course, if there are two single variable conjuncts with opposite polarity left in a formula, say $(\neg C)$ and (C) , we know that the SAT problem is not satisfiable).

There is another situation where we essentially get an answer for free. Consider the below formula — what do you notice about the variable D ?

$$\begin{aligned} &(\neg A \vee B \vee C) \wedge \\ &(A \vee B \vee \neg D) \wedge \\ &(\neg B \vee \neg C \vee \neg D) \end{aligned}$$

The variable D only appears with one polarity — negative! This means that we don't need to think too hard about what polarity D would be in a model,

setting it to False eliminates the bottom two clauses with no conflicts. D is known as a **pure variable** — it's one that only shows up with one polarity in the whole formula.

Using these two concepts, we are now ready to dive into our first “intelligent” algorithm for solving boolean SAT problems: The Davis–Putnam–Logemann–Loveland algorithm.

The DPLL Solver

We will start with a pseudo code description of the DPLL solver:

For a SAT formula in CNF form:

- Perform unit propagation
- Perform pure literal assignment
- If there are no remaining clauses, return SAT
- If there are empty clauses, return UNSAT
- Select a variable that remains in the formula, and try setting it to True or False, and repeat if either case allows.

From this we can see that the algorithm terminates in one of two cases:

(1) There are no remaining clauses (i.e variables have been assigned such that all clauses are satisfied)

(2) An empty clause (i.e variables have been assigned that have eliminated all variables from a particular clause) has been created despite trying both True and False for a variable.

Furthermore, the last step of our pseudo code algorithm is a little vague. The selection of a variable to try is called a **branching variable**, and selecting a good one (instead of just a random one from the formula) has been the topic of a substantial body of work in the last three decades.

It's also important to note that in the worst case, DPLL needs the same amount of steps as brute force search (2^n). However, DPLL has the benefit of terminating before BFS on unsatisfiable instances (and often on satisfiable instances as well). We can try pitting them against each other using some random instances:

```
from easysat.solvers import DPLL, BruteForce

ksat_generator = KSAT_Generator()
sample = ksat_generator.random_kcnf(k = 3, n_literals = 5, n_conjuncts=20)

print("Brute force-----")
bfs = BruteForce()
bfs.append_formula(sample)
sat,model, assigns = bfs.solve()
print("Satisfiable? ", sat)
print("Total assignments tested: ", assigns)
print("Satisfying assignment: ", model)

print()
print("DPLL-----")
dpll = DPLL()
dpll.append_formula(sample)
sat,model, assigns = dpll.solve()
print("Satisfiable? ", sat)
print("Total assignments tested: ", assigns)
print("Satisfying assignment: ", model)
```

For my run, this results in the following:

```
Brute force-----  
Satisfiable? True  
Total assignments tested: 17  
Satisfying assignment: {('2', True), ('3', True), ('4', True), ('1', True), ('5', True)}  
  
DPLL-----  
Satisfiable? True  
Total assignments tested: 7  
Satisfying assignment: {'1': True, '2': True, '3': True, '5': False, '4': True}
```

While DPLL is often faster than brute force search, this is not always the case, as you can see by re-running the code above several times. To check understanding, it would be useful to think about how DPLL vs BFS behave when you have an unsatisfiable formula with a single conflicting variable.

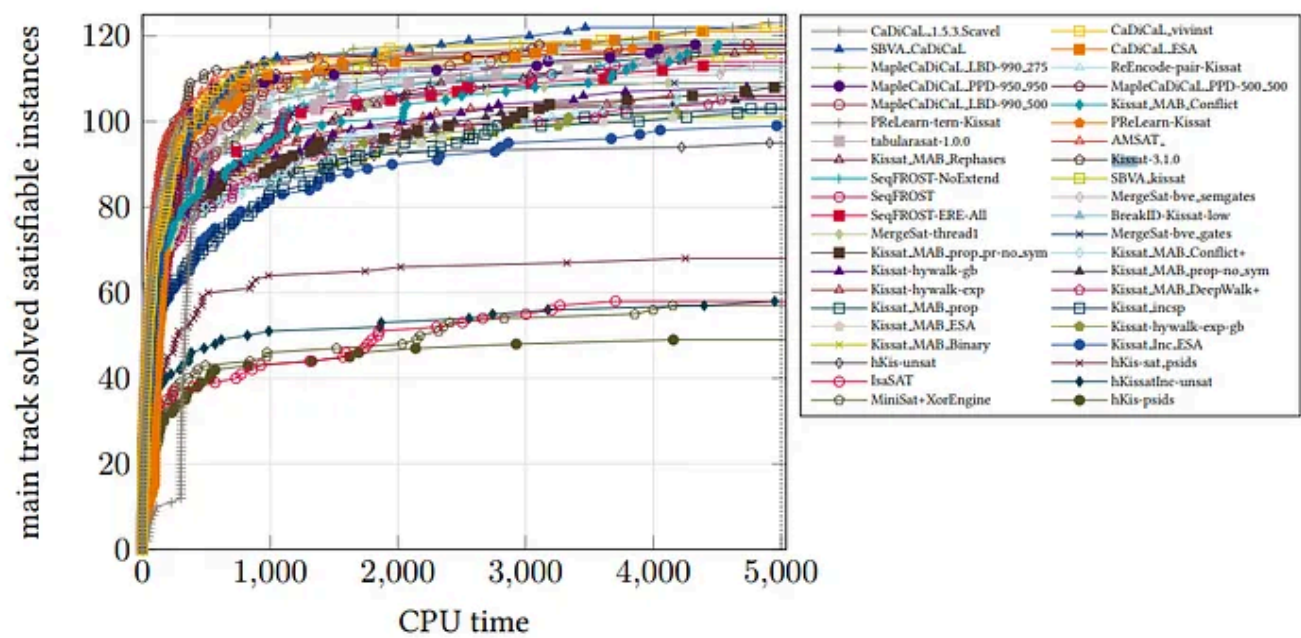
The Annual SAT Competition

Every year, a bunch of SAT fanatics get together to host the annual boolean satisfiability competition. Unfortunately named the “SAT competition,” it drives hundreds of submissions from independent researchers and major organizations energized by the prospect of being crowned as having the fastest SAT solver in history. Jokes aside, this title does mean something: SAT solvers are used extensively in industry, and even incremental speed ups can mean millions of dollars.

So what kinds of solvers are topping the charts in recent SAT competitions?

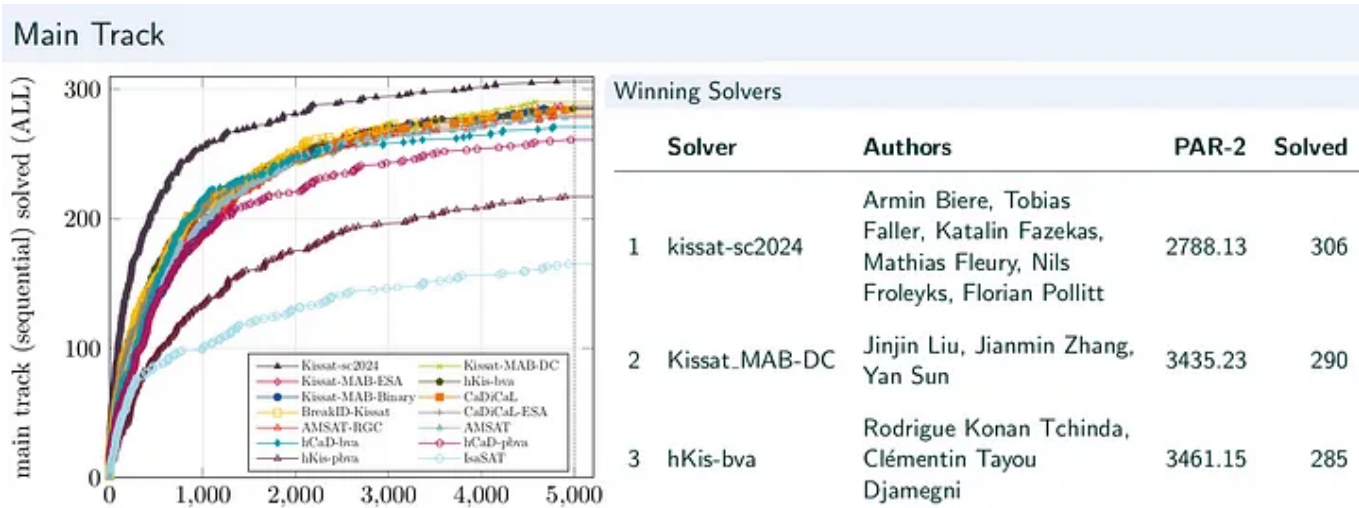
Here are the all the main track results for 2023:

Main (Sequential) Track SAT Plot



Main track results for SAT Competition 2023

Here are the top 3 results for 2024:



While most of the solver names might seem like gibberish, you’ll notice two terms that seem to show up pretty regularly – “CaDiCaL” and “Kissat”. Before we go on, it’s important to note that “Kissat” is just a re-

implementation of the CaDiCaL solver in the programming language C (with many optimizations). So this really raises the question — what exactly is CaDiCaL? How is it so consistently performing well?

Before we dive into CaDiCaL and **Conflict Driven Clause Learning**, we must first introduce the concept of an implication graph.

Implication Graphs

Consider a simple SAT instance with 2 literals and one (disjunctive) clause:

$$(A \vee B)$$

Now, think about the mental process of solving this SAT instance, variable-by-variable. Assigning either A or B to True obviously satisfies the instance. But if we assign A to False (for some reason), we know that we must then assign B to true to satisfy the instance. Expressed in logical terms (using the “implication” arrow):

$$\neg A \implies B$$

Now, if we instead set B to False, then we must do the opposite and set A to True.

$$\neg B \implies A$$

This is simple, but important. While predicting values for the variables, we can track our decisions and their implications through this process. Sometimes our hand will be forced (as in the two variable case above), but often times we'll find that a decision we made several turns ago turned out to be the wrong one. Ideally, we'd like to reverse that particular decision instead of trying to make it work with more guesses. Let's work through an example:

Consider the following formula with 5 literals and 4 conjuncts:

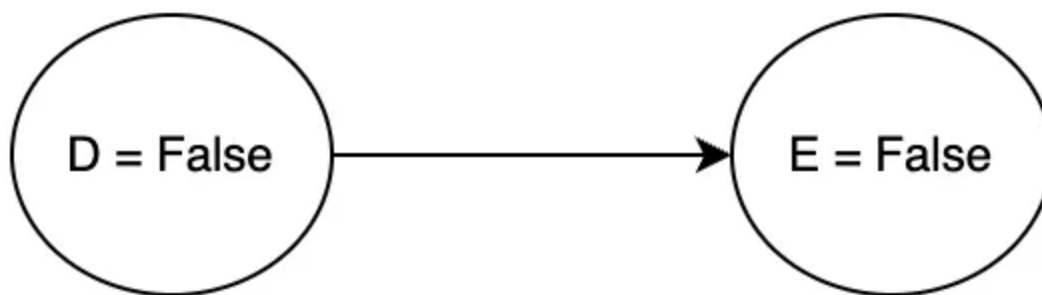
$$(A \vee B)$$

$$(A \vee \neg B \vee C)$$

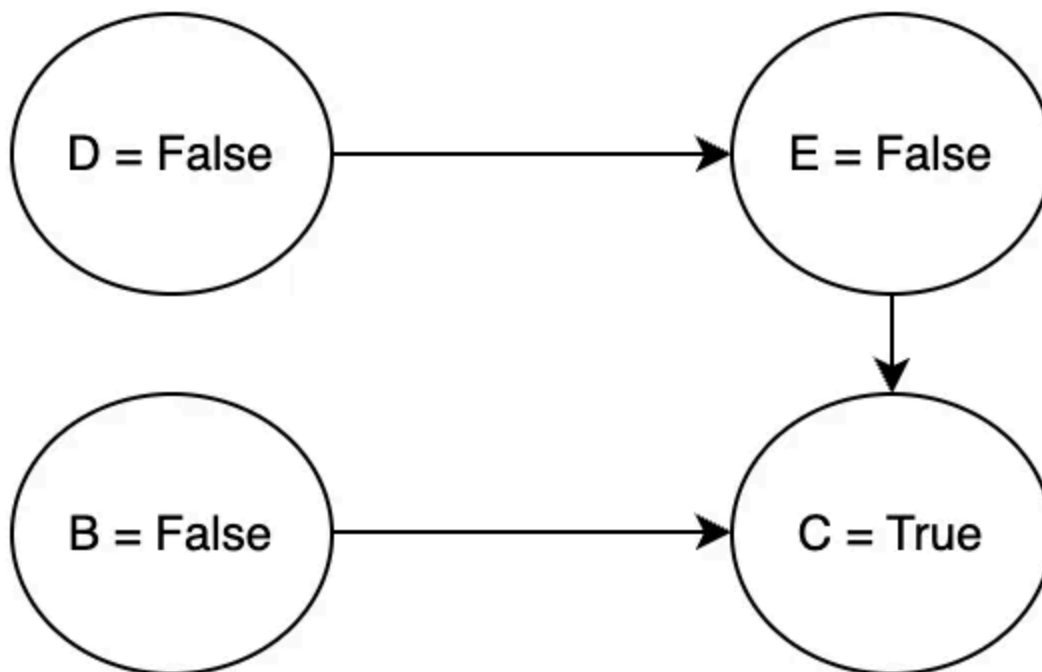
$$(B \vee \neg C \vee D)$$

$$(D \vee \neg E)$$

We'll start by guessing that $D = \text{False}$. This forces the outcome $E = \text{False}$, in order to satisfy the last clause. So far our implication graph looks like the following:



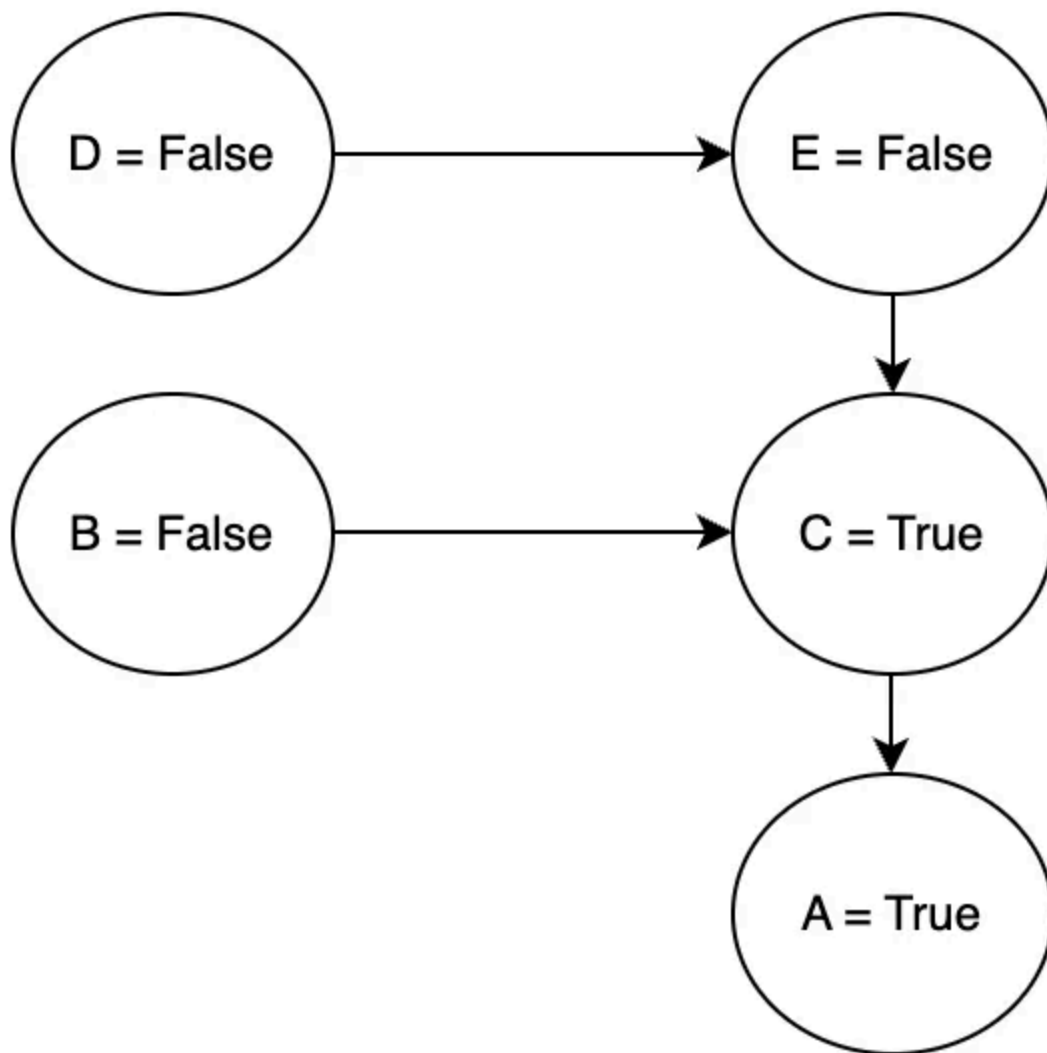
From here, we have three variables left to assign — A,B,C. Let's start with B. Assigning $B = \text{False}$, we notice that this forces $C = \text{True}$ to satisfy the second formula (since we already assigned $D = \text{False}$). Now, our implication graph looks like this:



Notice our guesses are on the left and any forced decisions on the right, with the right side connecting downward. This is one way to organize an

implication graph (though not the only way).

Finally, let's consider A. Because we've already assigned B and C in a way that makes the second conjunct true, we only need to consider the first one. The only way to make the first conjunct true is to set $A = \text{True}$, so this is a forced decision, and our final implication graph looks like this:



and we're done! We can easily read off the satisfying formula and see the full behavior of our "solver" from the implication graph.

We can also do this with longer SAT formula. An example 2-SAT formula with four conjuncts and four literals is given below, along with all of its implications.

$$(\neg A \vee B) \wedge (\neg B \vee \neg C) \wedge (C \vee \neg A) \wedge (A \vee D)$$

$$A \implies B$$

$$\neg B \implies \neg A$$

$$B \implies \neg C$$

$$C \implies \neg B$$

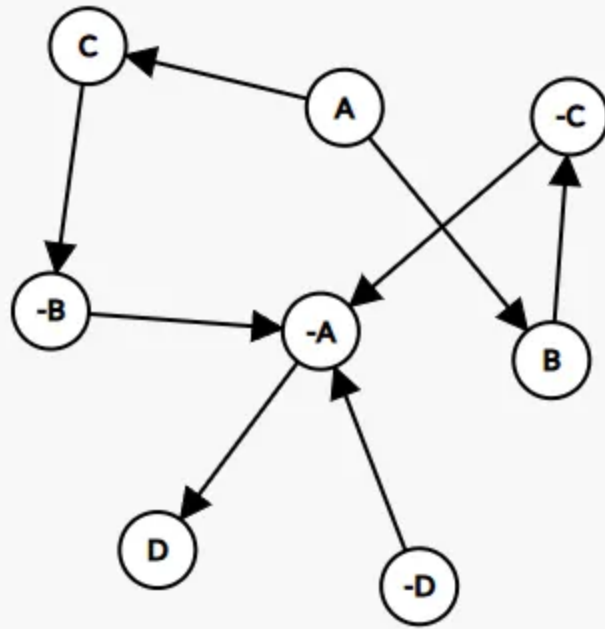
$$\neg C \implies \neg A$$

$$A \implies C$$

$$\neg A \implies D$$

$$\neg D \implies \neg A$$

Given that some of the literals are shared between implications, we can easily visualize this as a graph:



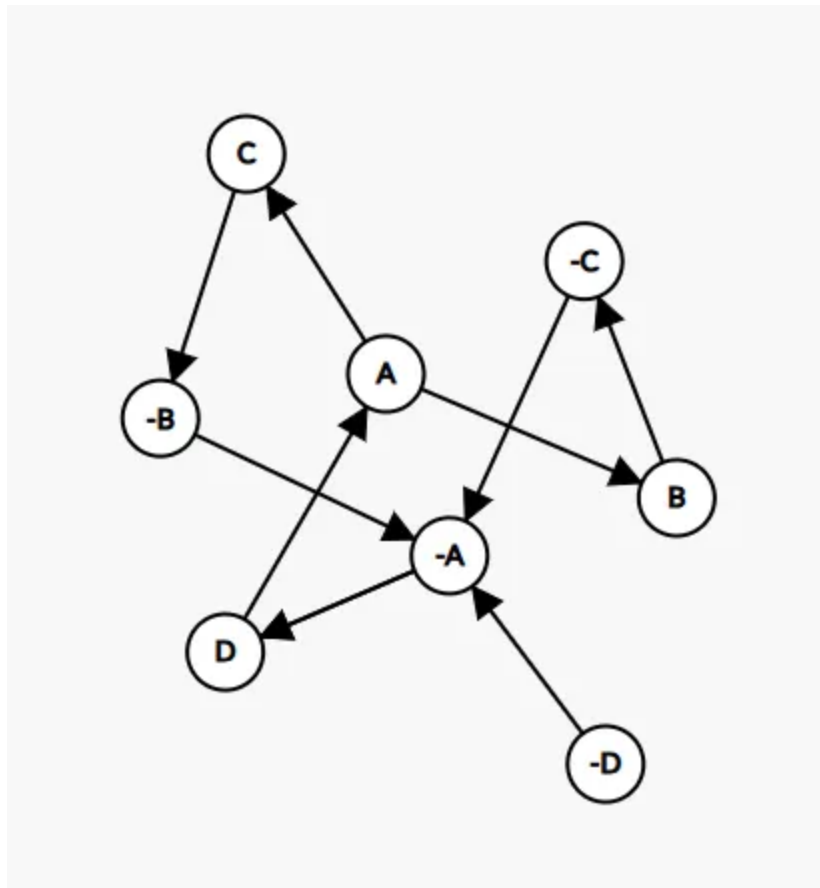
What does this get us beside a pretty picture? Well, consider adding the following conjunct (and associated implications) to our formula:

$$(\neg D \vee A)$$

$$D \implies A$$

$$\neg A \implies D$$

This transforms the implication graph into:



Now see what happens when you trace out the path starting at -D:

(-D, -A, D, A, B, -C, -A ...)

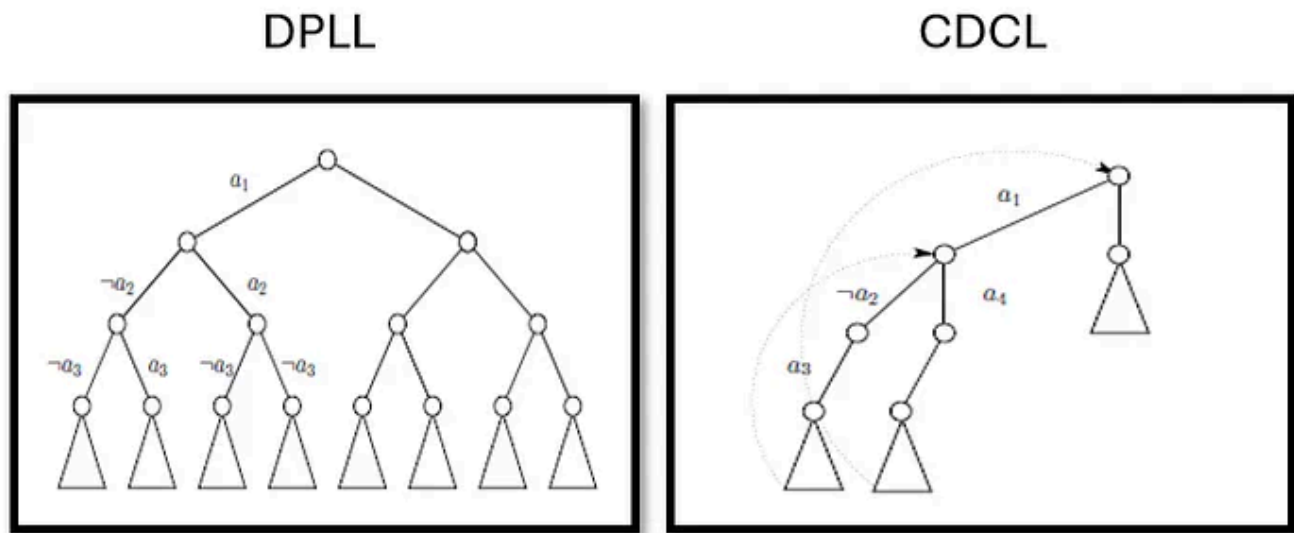
(1) The path enters you into an infinite loop

(2) The path contains a literal and its negation (actually several instances of this)

Because of these conditions in the implication graph, we know that -D was a poor choice. You can verify that this isn't the case for any of the paths in the original instance.

If you're guessing individual variables and updating the formula (using the DPLL approach above), at some point you may run into a looped implication

graph like the one above. This might not happen immediately though, and you might guess several variables before you realize that a guess you made many steps ago doomed you. Finding the root of the problem (the guess that doomed you), is called conflict analysis on implication graphs. Doing this automatically allows your solver to “backtrack” to the root and not make any more guesses down that branch. As you can imagine, this saves quite a bit of time compared to vanilla DPLL.



We are now ready to discuss the CDCL (Conflict Driven Clause Learning) algorithm.

Conflict Driven Clause Learning

We will start with the pseudo algorithm for CDCL (taken from Wikipedia, because it's a great description):

1. Select a variable and assign True or False. This is called decision state. Remember the assignment.
2. Apply unit propagation

3. Build the implication graph.
4. If there is any conflict, find the cut in the implication graph that led to the conflict
5. Derive a new clause which is the negation of the assignments that led to the conflict
6. Non-chronologically backtrack (“back jump”) to the appropriate decision level, where the first-assigned variable involved in the conflict was assigned
7. Otherwise continue from step 1 until all variable values are assigned.

There is a step here which may seem a bit confusing, and is different from DPLL:

(5): Derive a new clause which is the negation...

We haven't discussed what it means to “derive” a clause. Suppose you have made the following judgements and then reached a conflict: $A = \text{True}$, $B = \text{True}$, $C = \text{False}$.

The negation of these assignments is $A = \text{False}$, $B = \text{False}$, $C = \text{True}$.

So the new clause derived from this negation is simply

$$(\neg A \vee \neg B \vee C)$$

Adding this clause to the original formula will prevent this particular conflict from re-occurring. Often, this is referred to as an example of CDCL “learning” a way to prevent this (and other downstream) conflicts, but it is not learning in the traditional sense. CDCL is analytically backtracking to the earliest incorrect assignment, and modifying the formula to prevent this assignment from re-occurring. The process is entirely deterministic when the variable selection is fixed. Unlike BFS or DPLL, this prunes a whole section of the search space that is guaranteed to not work. The deeper this subspace of conflicting assignments, the more efficiency we gain by using CDCL instead of DPLL or BFS.

Benchmarking Different Algorithms

Now that we have a basic understanding of the satisfiability problem and some solvers, we can start discussing real world solver performance. There is a wealth of literature out there on random SAT problems, what makes them easy/difficult, and how different solvers stack up on different kinds of instances. In the interest of learning from practice, we will formulate an experiment here which will hopefully show the differences in solver performance in real time. To help with this, we will utilize one simple observation from theory:

The most difficult random SAT instances occur when the number of clauses is roughly 4.5x the number of variables. Less than this and instances generally have many solutions, while at higher ratios the instances generally have no solutions. One can look at this as the “edge of chaos” for SAT instances, though this is a topic for another time.

Now, here's the experimental setup:

- 1) Iterate through # of variables ranging between 3–20
- 2) For each variable count, multiply the number of variables by a random float between 4–5. This gives us a value in the critical region. Round this to get the number of propositions.
- 3) Generate a set of 50 random 3SAT instances with this variable and proposition count.
- 4) Run the set through each of the solvers, while keeping track of the worst clock time for each of the solvers over the set.

The code to run this experimental loop is rather long, so here it is as a gist:

```
1  import random
2  import time
3  import itertools
4  import numpy as np
5  import os
6  import numpy as np
7  import json
8  from tqdm import tqdm
9  from pysat.solvers import Glucose42, Minicard, Lingeling, Cadical153, Minisat22, MapleCh
10
11  from easysat.generators import KSAT_Generator
12  from easysat.solvers import BruteForce, DPLL
13
14  #####
15  %%time
16
17  ksg = KSAT_Generator()
18
19
20  brute_force_times = []
21  dpll_times = []
22  glucose_times = []
23  cadical_times = []
24  ms_times = []
25  lg_times = []
26  maple_times = []
27
28  MAX_LITERALS = 20 # Total number of literals to check up to
29  literals = list(range(3, MAX_LITERALS))
30  trials = 50 # Number of trials for each literal
31
32
33  hardest_problems = {}
34
35  for num_literals in tqdm(literals, total = len(literals)):
36
37      _brute_force_times = []
38      _dpll_times = []
39      _glucose_times = []
40      _cadical_times = []
41      _ms_times = []
42      _lg_times = []
43      _maple_times = []
44
45      # Sample trials from the critical region
```

```
45     # Sample a clause from the clause set
46     for num_conjuncts in [np.random.randint(num_literals*4,num_literals*5) for i in range(10000)]:
47
48         # Sample a random 3SAT clause
49         sample = ksg.random_kcnf(n_literals = num_literals, n_conjuncts=num_conjuncts)
50
51         # Initialize solver
52         g = Glucose42()
53         c = Minicard()
54         ms = Lingeling()
55         maple = Mergesat3()
56         dpll = DPLL()
57         bfs = BruteForce()
58
59         # Covert clauses to proper format
60         sample_cnf = ksg.kcnf_to_cnf(sample)
61
62         g.append_formula(sample_cnf)
63         c.append_formula(sample_cnf)
64         ms.append_formula(sample_cnf)
65         maple.append_formula(sample_cnf)
66         dpll.append_formula(sample_cnf)
67         bfs.append_formula(sample_cnf)
68
69         try:
70             ## Time BFS
71             if num_literals <= 15:
72                 start = time.time()
73                 _,answer, assigns = bfs.solve()
74                 runtime = time.time() - start
75                 _brute_force_times.append(assigns)
76
77
78             if num_literals <= 20:
79                 ## Time DPLL
80                 start = time.time()
81                 #_,answer,assigns = dpll.solve()
82                 runtime = time.time() - start
83                 _dpll_times.append(assigns)
84                 #_dpll_times.append(runtime)
85
86
87             ## Time Glucose
88             start = time.time()
89             result = g.solve()
```

```
90 runtime = time.time() - start
91 _glucose_times.append(g.accum_stats()['propagations'])
92 #_glucose_times.append(runtime)
93
94 props = g.accum_stats()['propagations']
95 if num_literals not in hardest_problems.keys():
96     hardest_problems[num_literals] = (sample_cnf, props, result, answer)
97 elif hardest_problems[num_literals][1] < props:
98     hardest_problems[num_literals] = (sample_cnf, props, result, answer)
99
100
101 ## Time Cadical
102 start = time.time()
103 c.solve()
104 runtime = time.time() - start
105 _cadical_times.append(c.accum_stats()['propagations'])
106 #_cadical_times.append(runtime)
107
108 ## Time Minisat
109 start = time.time()
110 ms.solve()
111 runtime = time.time() - start
112 _ms_times.append(ms.accum_stats()['propagations'])
113 #_ms_times.append(runtime)
114
115 ## Time Maple
116 start = time.time()
117 maple.solve()
118 runtime = time.time() - start
119 _maple_times.append(maple.accum_stats()['propagations'])
120 #_maple_times.append(runtime)
121
122 if num_literals <= 15:
123     brute_force_times.append(np.max(_brute_force_times))
124 if num_literals <= 20:
125     dpll_times.append(np.max(_dpll_times))
126 except Exception as e:
127     print(e)
128     pdb.set_trace()
129
130
131 glucose_times.append(np.max(_glucose_times))
132 cadical_times.append(np.max(_cadical_times))
133 ms_times.append(np.max(_ms_times))
134 maple_times.append(np.max(_maple_times))
```

```

135
136 #####
137 plt.style.use('tableau-colorblind10')
138 fig, axes = plt.subplots(figsize = (20,10))
139
140 axes.plot(literals[:len(brute_force_times)], brute_force_times, label = "Brute Force Sei
141 axes.plot(literals[:len(dpll_times)], dpll_times, label = "DPLL")
142 axes.plot(literals[:len(glucose_times)], glucose_times, label = "Glucose 3.4")
143 axes.plot(literals, cadical_times, label = "Minicard")
144 axes.plot(literals, ms_times, label = "Lingeling")
145 axes.plot(literals, maple_times, label = "Mergesat3")
146 axes.plot(literals, [1.15**n for n in literals], linestyle = '--', label = "F(x) = (1.1!
147 axes.set_ylim(0, 2000)
148 axes.set_xlabel("# of Literals")
149 axes.set_ylabel("Worst Case # of Propagations [out of n=50 runs per # of literals]")
150 axes.set_xticks(literals)
151 plt.suptitle("Performance of Modern Solvers on Random 3-SAT Formulas", fontsize = 16, y:
152 #plt.title("Worst Case out of n=100 runs per # of literals", fontsize = 10, y = 0.99)
153
154
155 axes.legend()

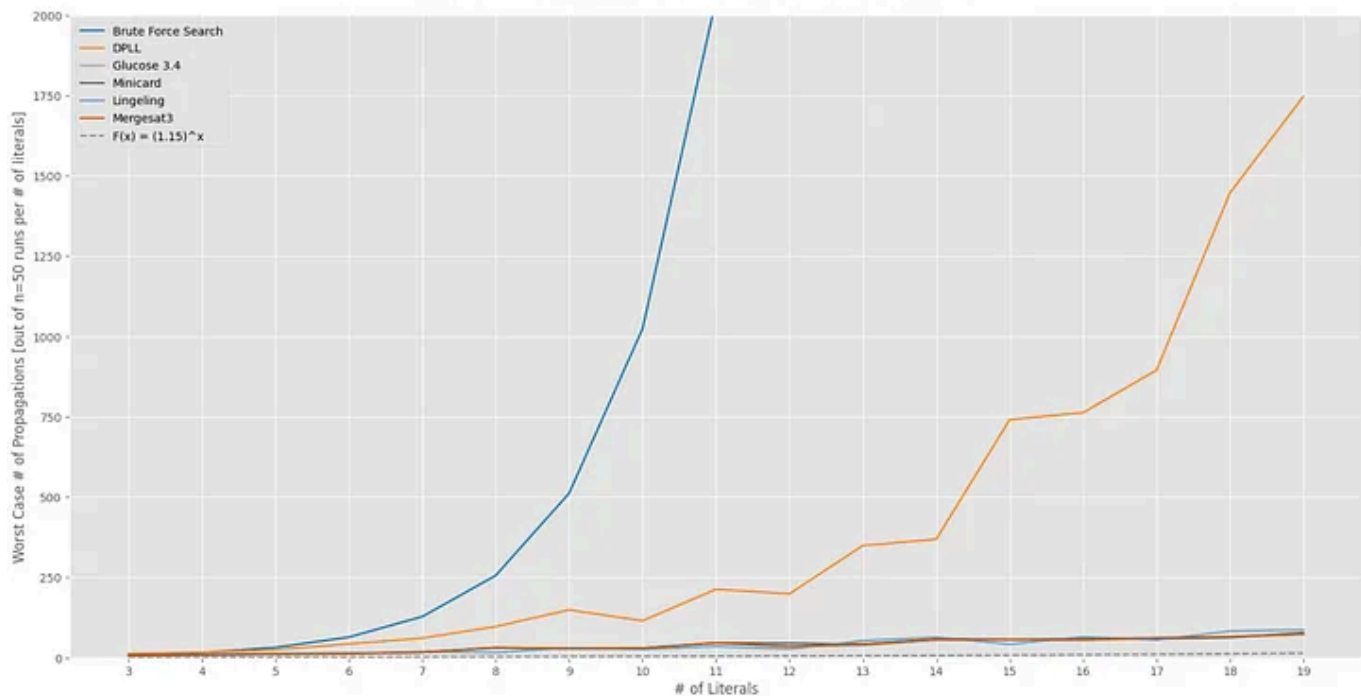
```

gistfile1.txt hosted with ❤ by GitHub

[view raw](#)

Note that we truncate the evaluation of BFS and DPLL to 15 and 20, respectively. This is to prevent drastic blowups of the run time for experiments with more literals. Here is the resulting graph, showing performance of some modern solvers against our random 3-SAT instances:

Performance of Modern Solvers on Random 3-SAT Formulas



Backbones and Instance Difficulty

Of course, the graph above makes the story look much simpler than it is. Given the richness of SAT instances, the simple relationship of more variables + critical clause-to-variable ratio = hard instance is less than the full picture.

It is true that very difficult SAT instances usually have a large search space. But if the large search space is accompanied by an equally large solution space, then the instance can still be easy to solve. The critical area favors instances with relatively small solution spaces compared to their search space, but there is a more explicit quantification of this fact, the **backbone size**.

The backbone of a SAT instance is the set of variables whose assignments are fixed in all solutions (if the instance is unsatisfiable, the definition of backbone varies). The larger this set, the less the number of solutions. In the

extreme case where an instance has only a single solution, the size of the backbone is equal to the number of variables — each of the variables can only be set in one way to solve the instance.

Generally, finding large SAT instances with unique or a small number of solutions is itself a difficult problem. Luckily for us, such instances have already been generated and are available openly on the web.

We will load these instances in and benchmark the performance of the best solver from above (Glucose) against different backbone sizes.

```
import random
import time
import itertools
import numpy as np
import os
import numpy as np
import json
from tqdm import tqdm
from pysat.solvers import Glucose42, Minicard, Lingeling, Cadical153, Minisat22,
import seaborn as sns
import matplotlib.pyplot as plt
from easysat.generators import KSAT_Generator
from easysat.solvers import BruteForce, DPLL

ksg = KSAT_Generator()

folder = r'./Data/CBS'
instances = [os.path.join(folder,x) for x in os.listdir(folder) if x.endswith('.

operations = {"Backbone10":[], "Backbone30":[], "Backbone50":[], "Backbone70":[]

for instance in tqdm(instances, total = len(instances)):

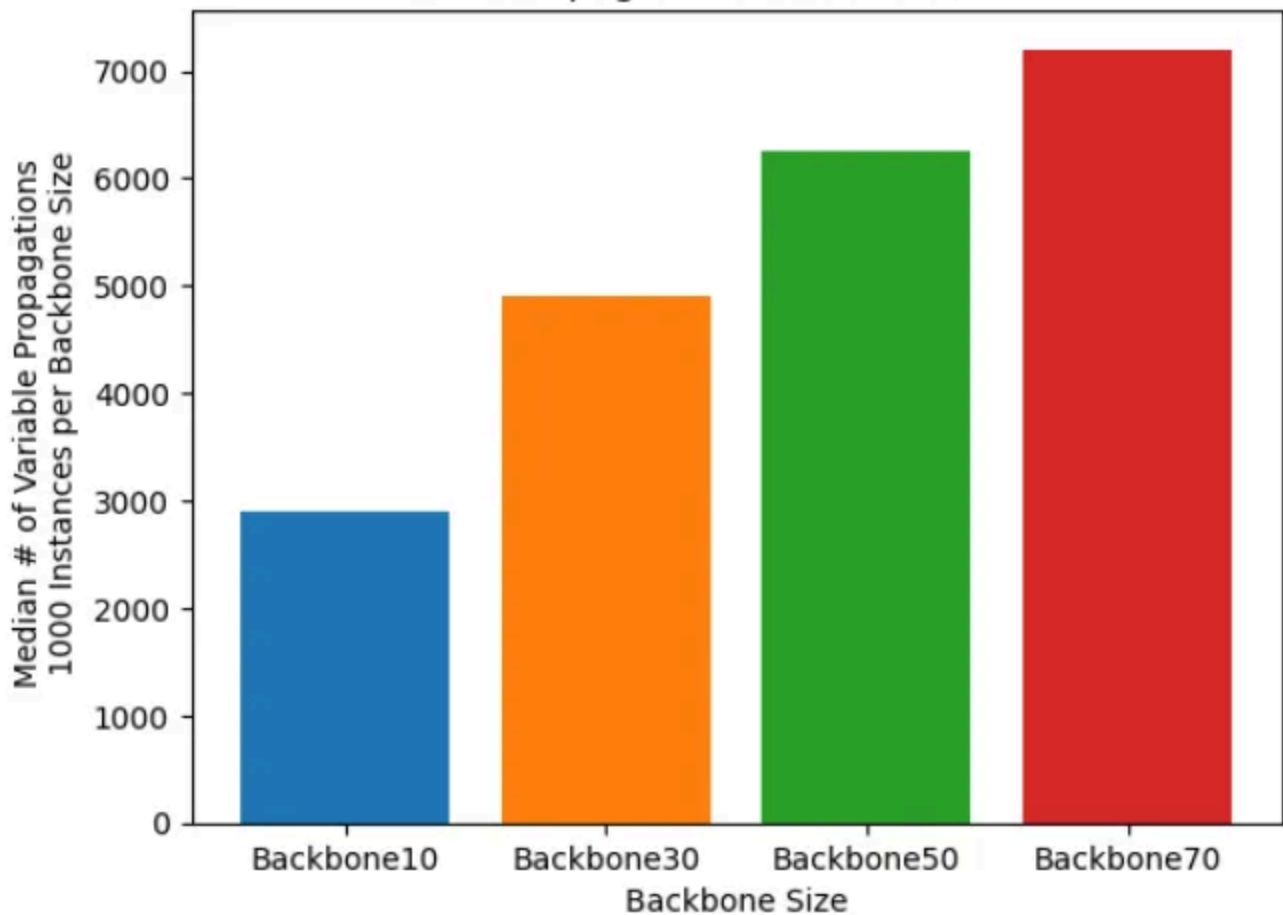
    cnf = ksg.from_dimacs_file(instance, 0)
    _, k, literals, clauses, backbone, _, = os.path.split(instance)[-1].split('_')
    backbone = int(backbone.replace('b', '').replace("_", ""))

    solver = Glucose42()
    solver.append_formula(cnf)
```

```
result = solver.solve()
runtime = solver.accum_stats()['propagations']

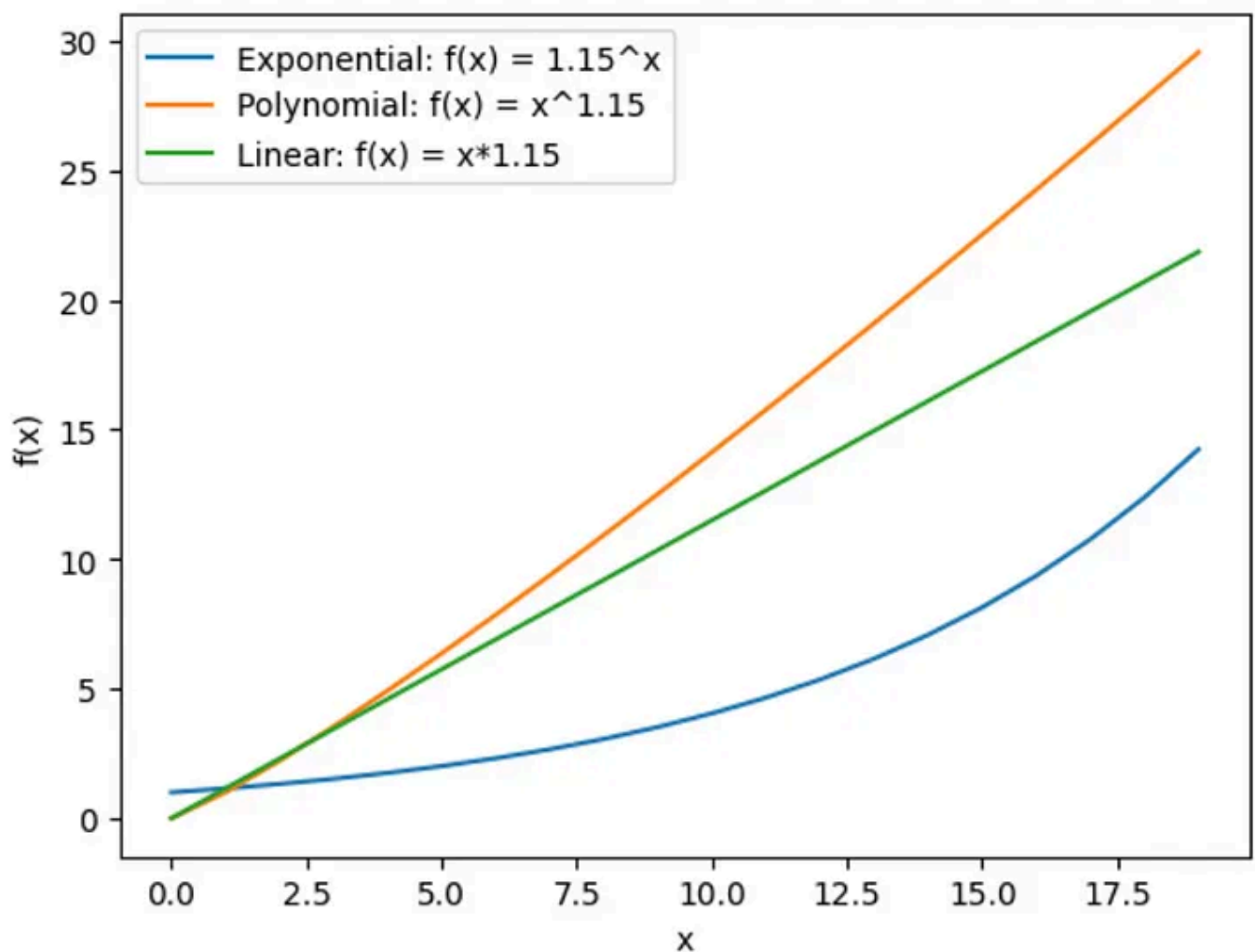
bkey = "Backbone" + str(backbone)
operations[bkey].append(runtime)

plt.bar("Backbone10", np.median(operations["Backbone10"]))
plt.bar("Backbone30", np.median(operations["Backbone30"]))
plt.bar("Backbone50", np.median(operations["Backbone50"]))
plt.bar("Backbone70", np.median(operations["Backbone70"]))
plt.title("Median # of Variable Propagations \n 1000 Instances per Backbone Size")
plt.title("Solver Propagations vs. Backbone Size")
plt.xlabel("Backbone Size")
plt.ylabel("Median # of Variable Propagations \n 1000 Instances per Backbone Siz
```

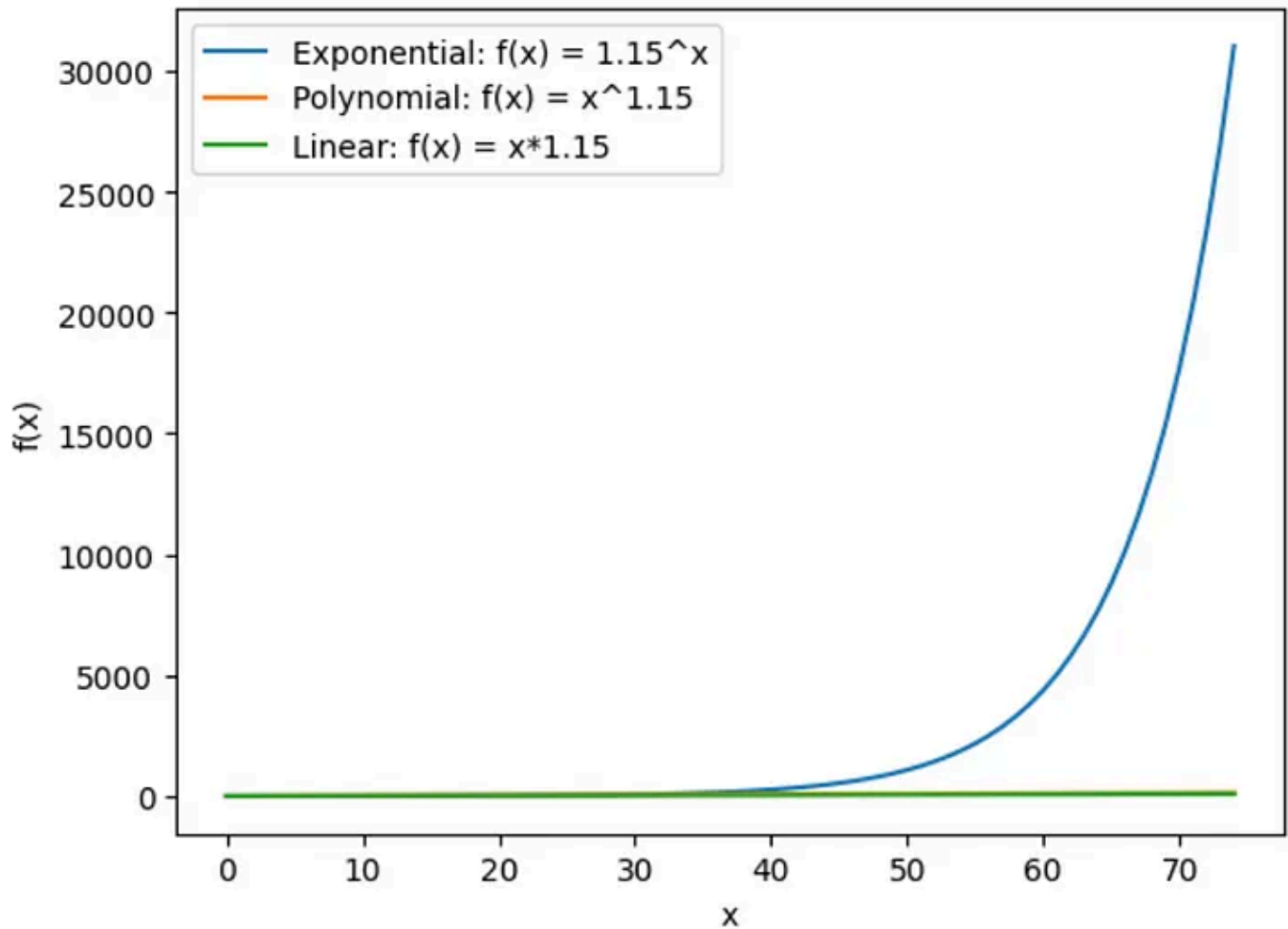
[Open in app ↗](#)**Medium** Search Write

Connections to the P=NP Problem

Look at the second to last graph again. Notice the dashed line that sits underneath all of the solvers. This dashed line indicates the scaling of the function $F(x) = 1.15^x$. The base 1.15 here is chosen arbitrarily, it can be any rational number > 1 . However the presence of the x in the exponent belies a stark reality — this function grows very quickly. It may not seem like it based on the performance plot, but here is the function next to a linear function and polynomial function utilizing the same constant:



Wait, this seems to contradict our point. Maybe if we extend the plot a little further...



Now things are more obvious. Here are the values at the end of the plot:

- $\text{Exponential}(2000) = 1.15^{75} \sim 31,019$
- $\text{Poly}(2000) = 75^{1.15} \sim 141$
- $\text{Lin}(2000) = 75 * 1.15 \sim 85$

Yeah, that's a big difference. Recall that many industrial grade instances can have millions of variables (in fact, the biggest application instance solved by at least one solver in the annual SAT competition contained 32 million clauses and 76 million literals).

The runtime of all of the algorithms we've looked at for SAT (with $k \geq 3$) is exponential (in the size of the instance). If an algorithm is found such that its scaling is polynomial for such instances, it would be a world changing discovery. This is for two reasons:

- 1) Many practical problems (of the sort discussed in the first section) result in very large SAT instances that are mostly out of reach of current day solvers.
- 2) SAT is something called an NP-Complete problem. This means that any problem in this complexity class can be translated into SAT, and a fast solver for SAT would result in a fast solver for any of the problems in this class.

Conclusion

After getting through this (admittedly long and exhausting) article, what are we left with? Yet another math problem that captures the real world but is frustratingly difficult to solve? Some interesting algorithms you can play with for an afternoon?

On the contrary, I think satisfiability problems are the easiest way to peek under the curtain of why the world may be the way it is. If even such a simple problem yields such immense complexity, it is not hard to imagine that the universe itself may not be as predictable as we believe. There may be some fundamental computational irreducibility, systems may exist whose evolution from one point to another cannot be further simplified.

Of course, this is a rabbit hole that goes much, much deeper. Hopefully this was a satisfying read for now.

References:

[1]: Understanding Implication Graphs, Mate Soos (2011).

[2]: Single Solution Random 3-SAT Instances, Marko Znidaric (2005).

[3]: Where the Really Hard Problems Are Cheeseman, P., Kanefsky, B., Taylor, W.M. (1991).

[4]: SAT Solver Etudes I, Philip Zucker (2025).

[5]: Backbones and Backdoors in Satisfiability, Kilby et al. (2005).

[Math](#)[Artificial Intelligence](#)[Computer Science](#)[Python](#)[Complexity](#)

Written by Shairoz Sohail

456 Followers · 14 Following

AI scientist @ JPL

Follow

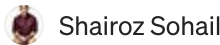
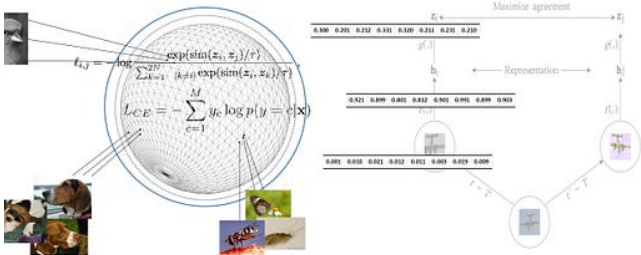
No responses yet





What are your thoughts?

More from Shairoz Sohail



Contrastive Representation Learning — A Comprehensive...

Turns out, telling things apart is a good way to understand the world

Dec 27, 2021

 87







AI For Good — Disaster Response

With all the discussions about the dangers and ethics around emerging artificial...

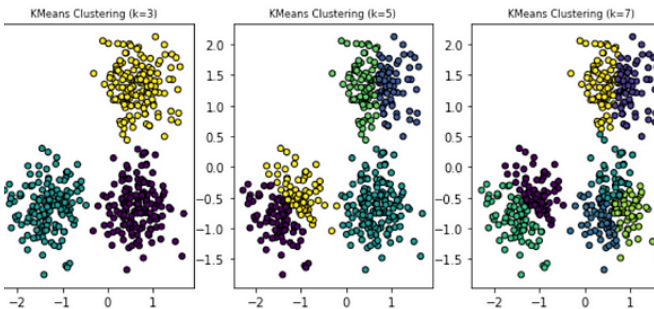
Jan 27, 2020

 201

 1









Shairoz Sohail

A Comprehensive Introduction to Clustering Methods

In terms of unsupervised learning methods, some of the most well researched and...

Mar 6, 2018



243



2



Shairoz Sohail

Generating 3D Models with Deep Learning (part 1)

i.e How the AI merchants of the metaverse will automatically create new stuff just for you

Feb 28, 2022



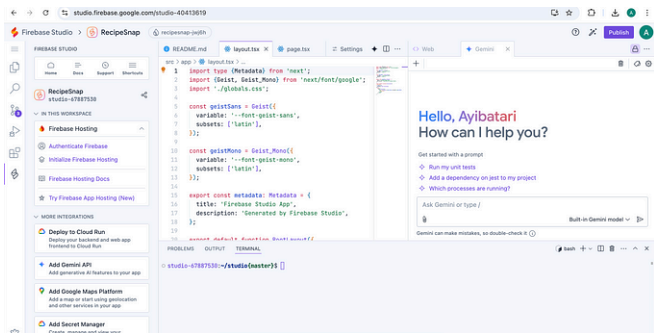
82




2

[See all from Shairoz Sohail](#)

Recommended from Medium



 In Coding Beauty by Tari Ibaba

Another amazing new IDE from Google—this destroys VS Code

Wow this is incredible—way better than Project IDX

★ 4d ago 🖱 401 💬 31  ⋮

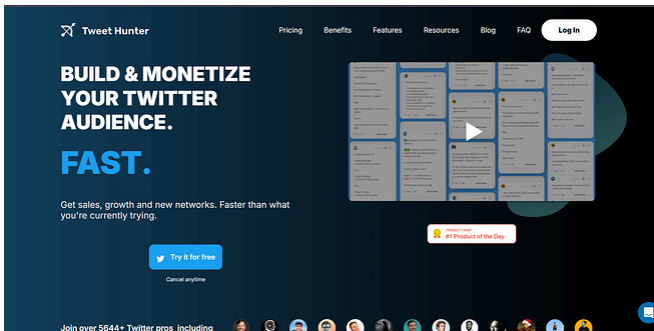


 Luke Skyward

OpenAI Might Have Just Killed Claude

AI Arms Race for Developers

★ 5d ago 🖱 278 💬 13  ⋮



 In Let's Code Future by Let's Code Future

I Replaced a Full Team With These 15 AI Tools (And Saved 20+...

If you're not using AI yet, you're not just behind—you're wasting time.

★ 4d ago 🖱 383 💬 14  ⋮




 Michael Swengel

Obsidian Offers Something Notion, Capacities and Craft NEVER Could

And that's why it's a clear winner. It's not even close.


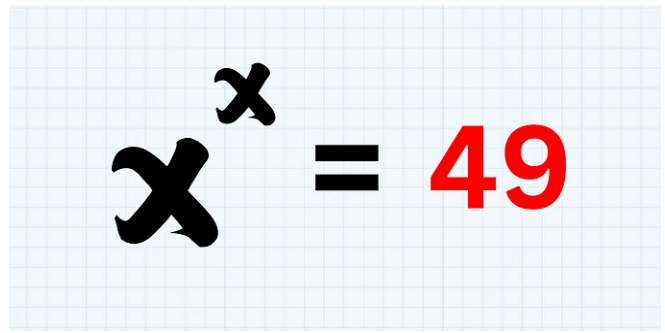
★ 4d ago 🖱 868 💬 22  ⋮

 Dhruv Ahuja

From Messy Scripts to Clean Code: How Python Classes Save the Day...

💡 Heads Up! Click here to unlock this article for free if you're not a Medium member!

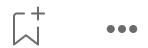
★ 4d ago 🖱 26

 In Think Art by Nnamdi Samuel

This Puzzle Breaks the Rules You're Used To

You'll Need More Than Just Trial and Error

★ 2d ago 🖱 88 💬 2



See more recommendations