SAT-Solvers: propositional logic in action

Russell Impagliazzo, with assistance from Cameron Held

October 22, 2013

1 Personal Information

A reminder that my office is 4248 CSE, my office hours for CSE 20 are Fridays at 2:30-3:30, and my email is russell@cs.ucsd.edu.

2 Overview

The purpose of these extra topics is to give you a sense of how what you are learning in CSE 20 fits into the bigger picture of the CSE curriculum and in the world of computing in general. Because the big picture is big, and will require the other three years of your undergrad program to understand fully, we are really only going to be able to give a superficial, whirlwind treatment of each topic.

SAT-Solvers automate propositional reasoning, and have become increasingly important for circuit and protocol verification. In discussing SAT-solvers, we need to brush on a bunch of themes that will come up again in your studies.

- When computers are so fast, why are fast algorithms important?
- How do we measure the speed of an algorithm?
- How much can we improve over exhaustive search algorithms? (The crux of the $P$ vs. $NP$ question).
- When is one problem a “special case” of another? (Reductions)
- Universal problem formats ($NP$-completeness)
- When are heuristics useful for hard problems?
- Are there any limits to what these heuristics can do?
3 The $P$ vs. $NP$ question

We say that an algorithmic problem is Easy if there is an efficient worst-case algorithms, an algorithm that always solves every instance and does it in a reasonable amount of time, i.e., one that scales well with the input. We usually use “polynomial in the input size amount of time” as a criterion for “easy”. A polynomial time has reasonably good scaling, in that if you double the input size, the time may double or be multiplied by some larger constant. This isn’t true for exponential time algorithms. $P$ stands for the class of problems that are easy in this sense, $NP$ stands for the class of problems where it is easy to verify solutions once they are found, but it may be hard to find solutions.

3.1 Example: Graph 3-coloring

- Can the vertices of a graph be given one of 3 colors (red, green, blue) in a way that doesn’t color any two adjacent nodes the same color?

- Originated from an abstraction of 19th century map coloring problems (want to distinctly color each region on a world map with minimum number of colors - answer is 4 colors).

- Trivial to verify the solution to graph coloring problem - but no simple efficient way to find the 3-coloring if it exists (doesn’t mean there aren’t easy instances, but many are much more difficult).

- So Graph 3-coloring is in $NP$, but we don’t know if it is in $P$.

3.1.1 Solving 3-coloring

The obvious way of solving 3 coloring is to use an Exhaustive search algorithm. Since there are 3 possible colors per node, we can try all combinations and check to see if a solution exists. With n nodes, this gives $3^n$ possible instances to check. Simple when n is small, but when n is large it becomes impossible.

Is exhaustive search good enough? I did a little more research just now. Answers.com suggests that a typical computer chip speed is around 7 gigaflops, about $2^{13}$ operations per second. The fastest super-computer is the 34 million petaflop Tianhe 2 (about $2^{55}$ operations each second), which uses about 3 million processors. $3^n$ is about $2^{1.6n}$. On our 7 gigaflop laptop, we could expect a $3^n$ time algorithm to take, for different n’s:

$n=20$ $2^{32}$ operations, about 1/000 of a second. Unnoticeable.

$n=30$ $2^{48}$ operations, about 32 seconds. Fast.

$n=40$ $2^{64}$ operations, about 3 weeks. Slow.

$n=50$ $2^{80}$ operations, about 3000 years. Ridiculous; don’t even think about it.
On the Tianhe-2, we could expect a $3^n$ time algorithm to take, for different $n$’s:

- **$n=20$**: $2^{32}$ operations. Unnoticeable.
- **$n=30$**: $2^{48}$ operations, about $1/128$’th of a second. Still unnoticeable.
- **$n=40$**: $2^{64}$ operations, about 500 seconds, or 8 minutes. Reasonably fast.
- **$n=50$**: $2^{80}$ operations, about a year. This had better be important.
- **$n=60$**: $2^{96}$ operations, about 64 thousand years. Ridiculous.

We can see that exponential growth becomes a huge problem. Some moderate sized instances, we might be able to wait for computers to get faster, but for larger instances, the solution becomes useless. Good performance on small inputs is no indication of good performance on large inputs, and getting faster computers only slightly increases the input size where we can hope the algorithm might terminate in our lifetimes.

### 3.1.2 Can we design a better algorithm?

What about if we find an algorithm that doesn’t need to do all of the steps? We can do significantly better. The best algorithm known for 3-coloring, by Beigel and Eppstein, runs in time about $2^{n/3}$. This is a big improvement: it multiplies the sizes for each category above by about 5, so that we could run it on inputs with $n$ up to 200 or so with hope of success. But it is still exponential, which means that as $n$ gets even slightly above that value, it soon becomes ridiculously slow. Might it be possible to get a much better algorithm?

### 3.2 NP-completeness.

Remember that a problem is in $NP$ if solutions can be verified easily. We say that a problem $\pi$ is $NP$-complete if

- $\pi \in NP$
- For every $\pi' \in NP$, instances of $\pi'$ can be "translated" into instances of $\pi$.

$NP$-Complete problems not only exist, but many natural problems are $NP$-Complete.

1. 3-coloring is an NP-Complete problem.
2. Sudoku is another example of an NP-Complete problem, if we generalize to $n^2 \times n^2$ sudoku puzzles rather than just a 9*9 puzzle.
3. SAT Problem
4. Many many others, involving scheduling, logic, computational biology, graphs, physics, chemistry....
3.3 The SAT Problem

An instance to the SAT problem specifies:

1. $x_1, \ldots, x_n$ Boolean variables

2. A set of clauses, where each clause is an OR of variables and their negations, e.g., $x_1 \lor \neg x_2 \lor x_8$.

Such an AND of OR’s is called a formula in Conjunctive Normal Form (CNF).

3.4 Reductions

If a problem is $NP$-complete, there should be a way to translate instances of any other $NP$ problem to instances of that problem. We won’t formally define what such a reduction is, but give an example, showing how to translate 3-coloring into SAT.

- What kind of boolean variables can we associate to the graph? One way is to introduce, for each node $a$, three boolean variables, $x_{ar} =$ “$a$ is red”, $x_{ab} =$ “$a$ is blue”, $x_{ag} =$ “$a$ is green”. But there are many ways to translate a problem into logic.

- Need clauses to represent constraints on the problems. Since we can’t have adjacent colors the same, for each edge between nodes $a$ and $b$, we introduce the clause $\neg x_{ar} \lor \neg x_{br}$, and similarly for the other two colors.

  We can introduce clauses such as $\neg x_{ar} \lor x_{ag}$ to represent that we cannot color a node both red and green. Finally we need to make sure that each node is colored, so we add the clause $x_{ar} \lor x_{ag} \lor x_{ab}$.

4 SAT Solvers

$NP$-completeness has two interpretations for problems: efficient algorithms are unlikely to exist (unless $P = NP$) but efficient algorithms would be extremely useful (since all search problems can be reduced to them.) One possible way of balancing this conundrum is with heuristics for $NP$-complete problems. Heuristics are algorithms that take exponential time in the worst-case, but often do much better.

$SAT$ is particularly useful, because the reductions to $SAT$ are much more direct than to other $NP$-complete problems and there are a variety of useful heuristics that have been implemented. Some well-known $SAT$-solvers are MiniSAT and zchaff. $SAT$ solvers have been used to verify complicated circuit designs and protocols, often involving solving instances of $SAT$ with hundreds of thousands of variables and millions of clauses. Almost all $SAT$ utilize a simple form of case-based propositional reasoning called resolution. The resolution rule is amazingly simple, but the best way of using it becomes increasingly sophisticated.
The main step is to pick a variable $x$ and consider the two cases when $x = T$ and $x = F$. When we set $x = T$, all clauses that contain $x$ are satisfied and can be (temporarily) removed. All clauses that contain $\neg x$ become shorter by 1. We then proceed recursively, for both restricted formulas (although we wait until one search terminates before trying the other: if we find a satisfying assignment in the first search, there is no need to search the other case).

If we falsify any clause, we can terminate the search. When we create a clause of size 1, that is called a unit clause propagation and forces the value of its remaining variable. Unit clause propagation is the main way the SAT solvers save over exhaustive search, especially once unit clauses start to chain, with one forced variable creating new unit clauses, which in turn create new unit clauses. But we can try to pick which variables to branch on to maximize unit clause propagation, and the exact rule used can affect time performance remarkably. In fact, one of the key ideas of the nineties was to start using restarts: if some order of variable choices was taking a long time, the algorithm permuted the variables and started over from scratch. Modern SAT solvers use clever variations like clause learning, where logical constraints deduced during the search in one case are stored for use in other cases. It is amazing how large an impact these variations can make in algorithm performance.

Some of my own research is in propositional proof complexity. If the formula has no solutions, a run of a SAT solver above produces a formal proof by contradiction that the formula is not satisfiable. By proving lower bounds on the size of these proofs, we can bound the time any such algorithm must take, no matter what variants it uses. With Princeton graduate student Chris Beck, I recently gave the best such lower bound: $2^{n/2}$ time is needed by any resolution-based algorithm in the worst-case for CNF’s with $n$ variables. So we know that the methods of solving SAT used in practice are exponential in the worst-case. But this does not explain at all their ability to solve very large instances that arise from applications. I would very much like to be able to explain the tremendous utility of SAT-solving as well as give limitations, but I have to admit that right now, it is still a mystery.

4.1 Extra Credit

Extra credit homework problem: Create a translation from Sudoku to SAT.