princeton univ. F'13 cos 521: Advanced Algorithm Design

Lecture 23: Heuristics: Algorithms we don't know how to analyze

Lecturer: *Sanjeev Arora* Scribe:

Any smart teenager who knows how to program can come up with a new algorithm. Analysing algorithms, by contrast, is not easy and usually beyond the teenager's skillset. In fact, if the algorithm is complicated enough, proving things about it (i.e., whether or not it works) becomes very difficult for even the best experts. Thus not all algorithms that have been designed have been analyzed. The algorithms we study today are called *heuristics*: for most of them we know that they do *not* work on worst-case instances, but there is good evidence that they work very well on many instances of practical interest. Explaining this discrepancy theoretically is an interesting and challenging open problem.

For pedagogical reasons, throughout the lecture we use the same problem as an example: 3SAT. Recall that the input to this problem consists of *clauses* which are $\vee$ (i.e., logical OR) of three literals, where a literal is one of n variables $x_1, x_2, \ldots, x_n$, or its negation. For example: $(x_1 \vee \neg x_4 \vee x_7) \wedge (x_2 \vee x_3 \vee \neg x_4)$. The goal is to find an assignment to the variables that makes them all clauses evaluate to true.

This is the canonical NP-complete problem: every other NP problem can be reduced to 3SAT (Cook-Levin Theorem, early 1970s). More importantly, problems in a host of areas are actually solved this way: convert the instance to an instance of 3SAT, and use an algorithm for 3SAT. In AI this is done for problems such as constraint satisfaction and motion planning. In hardware and software verification, the job of *verifying* some property of a piece of code or circuit is also reduced to 3SAT.

Let's get the simplest algorithm for 3SAT out of the way: try all assignments. This has the disadvantage that it takes 2^n time on instances that have few (or none) satisfying assignments. But there are cleverer algorithms, which run very fast and often solve 3SAT instances arising in practice, even on hundreds of thousand variables. The codes for these are publicly available, and whenever faced with a difficult problem you should try to represent it as 3SAT and use these solvers.

1 Davis-Putnam procedure

The Davis-Putnam procedure from the 1950s is very simple. It involves assigning values to variables one by one, and *simplifying* the formula at each step. For instance, if it contains a clause $x_3 \vee \neg x_5$ and we have just assigned x_5 to T (i.e., true) then the clause becomes true and can be removed. Conversely, if we assign it F the then the only way the remaining variables can satisfy the formula is if $x_3 = T$. Thus $x_5 = F$ *forces* $x_3 = T$. We call these effects the *simplification* of the formula.

Say the input is φ. Pick a variable, say x_i. Substitute $x_i = T$ in φ and simplify it. Recursively check the simplified formula for satisfiability. If it turns out to be unsatisfiable, then substitute $x_i = F$ in φ, simplify it, and recursively check that formula for satisfiability. If that also turns out unsatisfiable, then declare φ unsatisfiable.

When implementing this algorithm schema one has various choices. For instance, which variable to pick? Random, or one which appears in the most clauses, etc. Similarly, whether to try the value T first or F? What data structure to use to keep track of the variables and clauses? Many such variants have been studied and surprisingly, they do very well in practice. Hardware and software verification today relies upon the ability to solve instances with hundreds of thousands of variables.

CLAUSE LEARNING. The most successful variants of this algorithm involves learning from experience. Suppose the formula had clauses $(x_1 \vee x_7 \vee x_9)$ and $(x_1 \vee \neg x_9 \vee \neg x_6)$ and along some branch the algorithm tried $x_1 = F, x_7 = F, x_6 = T$, which led to a contradiction since x_9 is being forced to both T and F. Then the algorithm has learnt that this combination is forbidden, not only at this point but on every other branch it will explore in future. This knowledge can be added in the form of a new clause $x_1 \vee x_7 \vee \neg x_6$, since every satisfying assignment has to satisfy it. As can be imagined, clause learning comes in myriad variants, depending upon what rule is used to infer and add new clauses.

2 Local search

The above procedures set variables one by one. There is a different family of algorithms that does this in a different way. A typical is Papadimitriou's **Walksat** algorithm: *Start with a random assignment. At each step, pick a random variable and switch its value. If this increases the number of satisfied clauses, make this the new assignment. Continue this way until the number of satisfied clauses cannot be increased.*

Papadimitriou showed that this algorithm solves $2SAT$ with high probability.

Such algorithms fit in a paradigm called *local search*: maintain a solution at each step, and examine *neighboring* solutions in a bid to find one that improves the objective. Stop when the current solution is optimal in its neighborhood (i.e., locally optimal). One can think of this as a discrete analog of *gradient descent*.

Local search is a popular and effective heuristic for many other problems including traveling salesman and graph partitioning. Again, the theoretical results range from weak to nonexistent.

3 Difficult instances of 3SAT

We do know of hard instances for 3SAT for such heuristics. A simple family of examples uses the fact that there are small logical circuits (i.e., acyclic digraphs using nodes labeled with the gates $\vee, \wedge, \neg$)for integer multiplication. The circuit for multiplying two n-bit numbers has size about $O(n \log^2 n)$. So take a circuit C that multiplies two 1000 bit numbers. Input two random prime numbers p, q in it and evaluate it to get a result r. Now construct a boolean formula with $2n + O(|C|)$ variables corresponding to the input bits and the internal gates of C, and where the clauses capture the computation of each gate that results in the output r. (Note that the bits of r are "hardcoded "into the formula, but the bits of p, q as well as the values of all the internal gates correspond to variables.) Thus finding a satisfying assignment for this formula would also give the factors of r. (Recall that factoring a product of two random primes is the hard problem underlying public-key cryptosystems.)

The above SAT solvers have difficulty with such instances.

Other families of difficult formulae correspond to simple math theorems. A simple one is: *Every partial order on a finite set has a maximal element.* A *partial order* on n elements is a relation $\prec$ satisfying: (a) $x_i \not\prec x_i \quad \forall i$. (b) $x_i \prec x_j$ and $x_j \prec x_k$ implies $x_i \prec x_k$ (transitivity) (c) $x_i \prec x_j$ implies $x_j \not\prec x_i$. (Anti-symmetry).

For example, the relationship "is a divisor of"is a partial order among integers. We can represent a partial order by a directed acyclic graph.

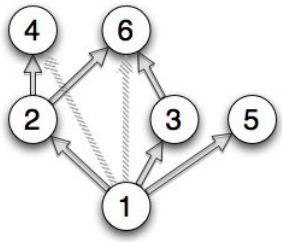

Figure 1: The relation "is a divisor of"is a partial order among integers.

Clearly, for every partial order on a finite set, there is a *maximal* element i such that $i \not\prec j$ for all j (namely, any leaf of the directed acyclic graph.) This simple mathematical statement can be represented as an unsatisfiable formula. However, the above heuristics seem to have difficulty detecting that it is unsatisfiable.

This formula has a variables x_{ij} for every pair of elements i, j. There is a family of clauses representing the properties of a partial order.

$$\begin{aligned} &\neg x_{ii} \quad \forall i \\ &\neg x_{ij} \vee \neg x_{jk} \vee x_{ik} \quad \forall i, j, k \\ &\neg x_{ij} \vee \neg x_{ji} \quad \forall i, j \end{aligned}$$

Finally, there is a family of clauses saying that no i is a maximal element. These clauses don't have size 3 but can be rewritten as clauses of size 3 using new variables.

$$x_{i1} \vee x_{i2} \vee \cdots \vee x_{in} \quad \forall i$$

4 Random SAT

One popular test-bed for 3SAT algorithms are *random* instances. A random formula with m clauses is picked by picking each clauses independently as follows: pick three variables randomly, and then toss a coin for each to decide whether it appears negated or unnegated.

Turns out if $m < 3.9n$ or so, then Davis-Putnal type procedures usually find a satisfying assignment. If $m > 4.3n$ these procedures usually fail. There is a different algorithm called *Survey propagation* that finds algorithms up to m close to $4.3n$. It is conjectured that there is a *phase transition* around $m = 4.3n$ whereby the formula goes from being satisfiable with

probability close to 1 to being unsatisfable with probability close to 1. But this conjecture is unproven, as is the conjecture that survey propagation works up to this threshold.

Now we show that if $m > 5.2m$ then the formula is unsatisfiable with high probability. This follows since the expected number of satisfying assignments in such a formula is $2^n(\frac{7}{8})^m$ (this follows by linearity of expectation since there are 2^n possible assignments, and any fixed assignment satisfies all the m independently chosen clauses with probability $(\frac{7}{8})^m$). For $m > 5.2n$ this number is very tiny, so by Markov's inequality the probability it is ≥ 1 is tiny.

Note that we do not know how to prove in polynomial time, given such a formula with $m > 5.2n$, that it is unsatisfiable. In fact it is known that known that for $m > Cn$ for some large constant C, the simple DP-style algorithms take exponential time.

5 Metropolis-Hastings and Gibbs Sampling

There is another family of heuristics that solves a more general problem: sampling from a probability distribution for which only the *density* function is known. Say the distribution is defined on $\{0,1\}^n$ and we have a *goodness* function $f(x)$ that is nonnegative and computable in polynomial time given $x \in \{0,1\}^n$. Then we wish to sample from the distribution where probability of getting x is *proportional* to $f(x)$. Since probabilities must sum to 1, we conclude that this probability is $f(x)/N$ where $N = \sum_{x\in\{0,1\}^n} f(x)$ is the so-called *partition function*. The main problem problem here is that N is in general hard to compute; it is complete for the class $\sharp P$ mentioned in an earlier lecture.

Lets note that if one could sample from such a distribution in general, then 3SAT becomes easy. For any assignment x define $f(x) = 2^{2nf_x}$ where f_x = number of clauses satisfied by x. Then if the formula has a satisfiable assignment, then $N > 2^{2n^2}$ whereas if the formula is unsatisfiable then $N < 2^n \times 2^{2n(n-1)} < 2^{2n^2-n}$. In particular, for a satisfiable formula, most of the probability is focused on the satisfying assingments, so the ability to sample from the distribution would yield a satisfying assignment with high probability.

This situation arises often in physics and machine learning, and there is a popular heuristic for sampling from such a distribution. Define the following random walk on $\{0,1\}^n$. *At every step there is a current assignment, say x. (Initialize arbitrarily.) At every step, toss a coin to stay at x with probability* $1/2$. *If you decide to move, randomly pick a neighbor x' of x. Move to x' with probability* $\min 1, \frac{f(x')}{f(x)}$. *(In other words, if $f(x') \geq f(x)$, definitely move. Otherwise move with probability given by their ratio.)*

CLAIM: *If all $f(x) > 0$ then the stationary distribution of this Markov chain is exactly $p(x)/N$, the desired distribution*

PROOF: The markov chain defined by this random walk is ergodic since $f(x) > 0$ implies it is connected, and the self-loops imply it mixes. Thus it suffices to show that the (unique) stationary distribution has the form $f(x)/K$ for some scale factor K, and then it follows that K is the partition function N. To do so it suffices to verify that such a distribution is stationary, i.e., in one step the probability flowing out of a vertex equals its inflow. For any x, lets L be the neighbors with a *lower* f value and H be the neighbors with value at least

as high. Then the outflow of probability per step is

$$\frac{f(x)}{2K}(\sum_{x' \in L} \frac{f(x')}{f(x)} + \sum_{x' \in H} 1),$$

whereas the inflow is

$$\frac{1}{2}(\sum_{x' \in L} \frac{f(x')}{K} \cdot 1 + \sum_{x' \in H} \frac{f(x')}{K} \frac{f(x)}{f(x')}),$$

and the two are the same. □

Simulated Annealing. If we use the suggested goodness function for 3SAT $f(x) = 2^{2nf_x}$ then this Markov chain can be shown to have poor mixing. So a variant is to use a markov chain that updates itself. The goodness function is initialized to say $2^{\gamma f_x}$ for $\gamma = 1$, then allowed to mix. This stationary distribution may put too little weight on the satisfying assignments. So then slowly increase γ from 1 to $2n$, allowing the chain to mix for a while at each step. This family of algorithms is called *simulated annealing*, named after the physical process of annealing.

For further information see this survey and its list of references.

Satisfiability Solvers, by C.P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. *Handbook of Knowledge Representation*, Elsevier 2008.