Chapter 24

Heuristics: Algorithms we don't know how to analyze

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.

Though the heuristics apply to many problems, 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 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 more clever 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.

24.1 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.

One of you asked why adding clauses (ie more constraints) simplifies the problem instead of making it harder. The answer is that the clauses can be seen as guidance towards a satisfying assignment (if one exists). The clauses can be used in making the crucial decision in DPLL procedures about which variable to set, and how to set it (T or F). The wrong decision may cause you to potentially incur huge cost. So anything that lowers the probability of wrong decision by even a bit could drastically change your running time.

24.2 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 more paradigm called *local search*, which can be described as follows.

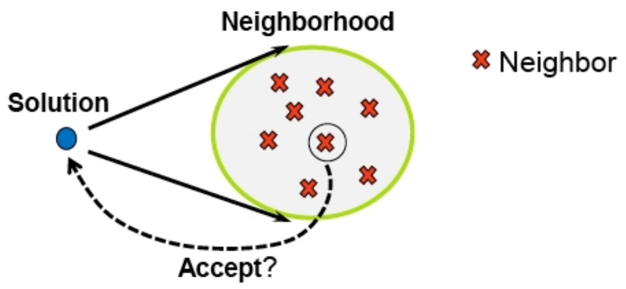

Figure 24.1: Local search algorithms try to improve the solution by looking for small changes that improve it.

> *Maintain a solution at each step. If the current solution is x, look for a solution y in a neighborhood $Ball(x, r)$ of radius r around x (that is, all solutions that differ from x up to some amount small r). If you find such a y that improves over x (in terms of the objective being optimized) then replace x by y. Stop if no such y was found.*

Clearly, when the algorithm stops, the current solution is optimal in its neighborhood (i.e., locally optimal). One can think of this as a discrete analog of *gradient descent*. An example of nonlocal change is any of the global optimization algorithms like Ellipsoid method.

Thus local search is a formalization of improvement strategies that we come up with intuitively, e.g., change ourselves by making small continuous changes. The Japanese have a name for it: *kaizen*[1]

Example 52 Local search is a popular and effective heuristic for many other problems including traveling salesman and graph partitioning. For instance, one local search strategy (which even students in my freshman seminar were able to quickly invent) is to start with a tour, and at each step try to improve it by changing up to two edges (2-OPT) or k edges (k-OPT). We can find the best local improvement in polynomial time (there are only $\binom{n}{2}$ ways to choose 2 edges in a tour) but the number of local improvement steps may be exponential in n. So the overall running time may be exponential.

These procedures often do well in practice, though theoretical results are few and far between. One definitive study is
The traveling salesman problem: A case study in local optimization, by D. Johnson and C. McGeoch. 1997

Example 53 *Evolution* a la Darwin can be seen as a local search procedure. Mutations occur spontaneously and can be seen as exploring a small neighborhood of the organism's genome. The environment gives feedback over the quality of mutations. If the mutation is good, the descendents thrive and the mutation becomes more common in the gene pool. (Thus the mutated genome becomes the new solution y in the local search). If the mutation is harmful the descendents die out and the mutation is thus removed from the gene pool.

[1]I would like to know if Japanese magazines have cover stories on new kaizen ideas just as cover stories in US magazines promote radical makeovers.

24.3 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.

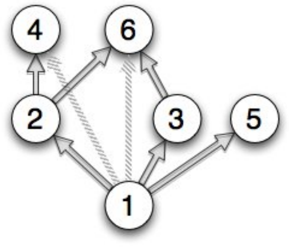

Figure 24.2: The relation "is a divisor of"is a partial order among integers.

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.

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

24.4 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.

24.5 Metropolis-Hastings and Computational statistics

Now we turn to counting problems and statistical estimation, discussed earlier in Lecture 21. Recall the Monte Carlo method for estimating the area of a region: through darts and see what fraction land in the region.

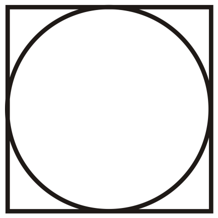

Figure 24.3: Monte Carlo (dart throwing) method to estimate the area of a circle. The fraction of darts that fall inside the disk is $\pi/4$.

Now suppose we are trying to integrate a nonnegative valued function f over the region. Then we should throw a dart which lands at x with probability $f(x)$. We'll examine how to throw such a dart.