CS261: A Second Course in Algorithms
Lecture #19: Beating Brute-Force Search[*]

Tim Roughgarden[†]

March 8, 2016

A popular myth is that, for NP-hard problems, there are no algorithms with worst-case running time better than that of brute-force search. Reality is more nuanced, and for many natural NP-hard problems, there are algorithms with (worst-case) running time much better than the naive brute-force algorithm (albeit still exponential). This lecture proves this point by revisiting three problems studied in previous lectures: vertex cover, the traveling salesman problem, and 3-SAT.

1 Vertex Cover and Fixed-Parameter Tractability

This section studies the special case of the vertex cover problem (Lecture #18) in which every vertex has unit weight. That is, given an undirected graph $G = (V, E)$, the goal is to compute a minimum-cardinality subset $S \subseteq V$ that contains at least one endpoint of every edge.

We study the problem of checking whether or not a vertex cover instance admits a vertex cover of size at most k (for a given k). This problem is no easier than the general problem, since the latter reduces to the former by trying all possible values of k. Here, you should think of k as "small," for example between 10 and 20. The graph G can be arbitrarily large, but think of the number of vertices as somewhere between 100 and 1000. We'll show how to beat brute-force search for small k. This will be our only glimpse of "parameterized algorithms and complexity," which is a vibrant subfield of theoretical computer science.

The naive brute-force search algorithm for checking whether or not there is a vertex cover of size at most k is: for every subset $S \subseteq V$ of k vertices, check whether or not S is a vertex cover. The running time of this algorithm scales as $\binom{n}{k}$, which is $\Theta(n^k)$ when k is small. While technically polynomial for any constant k, there is no hope of running this algorithm unless k is extremely small (like 3 or 4).

If we aim to do better, what can we hope for? Better than $\Theta(n^k)$ would a running time of the form poly$(n) \cdot f(k)$, where the dependence on k and on n can be separated, with

[*]©2016, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

the latter dependence only polynomial. Even better would be a running time of the form poly(n) + $f(k)$ for some function k. Of course, we'd like the poly(n) term to be as close to linear as possible. We'd also like the function $f(k)$ to be as small as possible, but because the vertex cover problem is NP-hard for general k, we expect $f(k)$ to be at least exponential in k. An algorithm with such a running time is called *fixed-parameter tractable (FPT)* with respect to the parameter k.

We claim that the following is an FPT algorithm for the minimum-cardinality vertex cover problem (with budget k).

FPT Algorithm for Vertex Cover

```
set $S = \{v \in V : \deg(v) \geq k + 1\}$
set $G' = G \setminus S$
set $G''$ equal to $G'$ with all isolated vertices removed
**if** $G''$ has more than $k^2$ edges **then**
    return "no vertex cover with size $\leq k$"
**else**
    compute a minimum-size vertex cover $T$ of $G''$ by brute-force search
    return "yes" if and only if $|S| + |T| \leq k$
```

We next explain why the algorithm is correct. First, notice that if G has a set cover S of size at most k, then every vertex with degree at least $k + 1$ must be in S. For if such a vertex v is not in S, then the other endpoint of each of the (at least $k + 1$) edges incident to v must be in the vertex cover; but then $|S| \geq k + 1$. In the second step, G' is obtained from G by deleting S and all edges incident to a vertex in S. The edges that survive in G' are precisely the edges not already covered by S. Thus, the vertex covers of size at most k in G are precisely the sets of the form $S \cup T$, where T is a vertex cover of G' size at most $k - |S|$. Given that every vertex cover with size at most k contains the set S, there is no loss in discarding the isolated vertices of G' (all incident edges of such a vertex in G are already covered by vertices in S). Thus, G has a vertex cover of size at most k if and only if G'' has a vertex cover of size at most $k - |S|$. In the fourth step, if G'' has more than k^2 edges, then it cannot possibly have a vertex cover of size at most k (let alone $k - |S|$). The reason is that every vertex of G'' has degree at most k (all higher-degree vertices were placed in S), so each vertex of G'' can only cover k edges, so G'' has a vertex cover of size at most k only if it has at most k^2 edges. The final step computes the minimum-size vertex cover of G'' by brute force, and so is clearly correct.

Next, observe that in the final step (if reached), the graph G'' has at most k^2 edges (by assumption) and hence at most $2k^2$ vertices (since every vertex of G'' has degree at least 1). It follows that the brute-force search step can be implemented in $2^{O(k^2)}$ time. Steps 1–4 can be implemented in linear time, so the overall running time is $O(m) + 2^{O(k^2)}$, and hence the algorithm is fixed-parameter tractable. In FPT jargon, the graph G'' is called a *kernel* (of size $O(k^2)$), meaning that the original problem (on an arbitrarily large graph, with a given budget k) reduces to the same problem on a graph whose size depends only on k. Using linear programming techniques, it is possible to show that every unweighted vertex cover instance actually admits a kernel with size only $O(k)$, leading to a running time dependence on k of $2^{O(k)}$ rather than $2^{O(k^2)}$. Such singly-exponential dependence is pretty much the best-case scenario in fixed-parameter tractability.

Just as some problems admit good approximation algorithms and others do not (assuming $P \neq NP$), some problems (and parameters) admit fixed-parameter tractable algorithms while others do not (under appropriate complexity assumptions). This is made precise primarily via the theory of "$W[1]$-hardness," which parallels the familiar theory of NP-hardness. For example, the independent set problem, despite its close similarity to the vertex cover problem (the complement of a vertex cover is an independent set and vice versa), is $W[1]$-hard and hence does not seem to admit a fixed-parameter tractable algorithm (parameterized by the size of the largest independent set).

2 TSP and Dynamic Programming

Recall from Lecture #16 the traveling salesman problem (TSP): the input is a complete undirected graph with non-negative edge weights, and the goal to compute the minimum-cost TSP tour, meaning a simple cycle that visits every vertex exactly once. We saw in Lecture #16 that the TSP problem is hard to even approximate, and for this reason we focused on approximation algorithms for the (still NP-hard) special case of the metric TSP. Here, we'll give an exact algorithm for TSP, and we won't even assume that the edges satisfy the triangle inequality.

The naive brute-force search algorithm for TSP tries every possible tour, leading to a running time of roughly $n!$, where n is the number of vertices. Recall that $n!$ grows considerably faster than any function of the form c^n for a constant c (see also Section 3). Naive brute-force search is feasible with modern computers only for n in the range of 12 or 13. This section gives a dynamic programming algorithm for TSP that runs in $O(n^2 2^n)$ time. This extends the "tractability frontier" for n into the 20s. One drawback of the dynamic programming algorithm is that it also uses exponential space (unlike brute-force search). It is an open question whether or not there is an exact algorithm for TSP that has running time $O(c^n)$ for a constant $c > 1$ and also uses only a polynomial amount of space. Two take-aways from the following algorithm are: (i) TSP is another fundamental NP-hard problem for which algorithmic ingenuity beats brute-force search; and (ii) your algorithmic toolbox (here, dynamic programming) continues to be extremely useful for the design of exact algorithms for NP-hard problems.

Like any dynamic programming algorithm, the plan is to solve systematically a collection of subproblems, from "smallest" to "largest," and then read off the final answer from the biggest subproblems. Coming up with right subproblems is usually the hardest part of designing a dynamic programming algorithm. Here, in the interests of time, we'll just cut to the chase and state the relevant subproblems.

Let $V = \{1, 2, \ldots, n\}$ be the vertex set. The algorithm populates a two-dimensional array A, with one dimension indexed by a subset $S \subseteq V$ of vertices and the other dimension indexed by a single vertex j. At the end of the algorithm, the entry $A[S, j]$ will contain the cost of the minimum-cost path that:

(i) visits every vertex $v \in S$ exactly once (and no other vertices);

(ii) starts at the vertex 1 (so 1 better be in S);

(iii) ends at the vertex j (so j better be in S).

There are $O(n2^n)$ subproblems. Since the TSP is NP-hard, we should not be surprised to see an exponential number of subproblems.

After solving all of the subproblems, it is easy to compute the cost of an optimal tour in linear time. Since $A[\{1, 2, \ldots, n\}, j]$ contains the length of the shortest path from 1 to j that visits every vertex exactly once, we can just "guess" (i.e., do brute-force search over) the vertex preceding 1 on the tour:

$$OPT = \min_{j=2}^{n} \left(\underbrace{A[\{1, 2, \ldots, n\}, j]}_{\text{path from 1 to } j} + \underbrace{c_{j1}}_{\text{last hop}} \right).$$

Next, we need a principled way to solve all of the subproblems, using solutions to previously solved "smaller" subproblems to quickly solve "larger" subproblems. That is, we need a *recurrence* relating the solutions of different subproblems. So consider a subproblem $A[S, j]$, where the goal is to compute the minimum cost of a path subject to (i)–(iii) above. What must the optimal solution look like? If we only knew the penultimate vertex k on the path (right before j), then we would know what the path looks like: it would be the cheapest possible path visiting each of the vertices of $S \setminus \{j\}$ exactly once, starting at 1, and ending at k (why?), followed of course by the final hop from k to j. Our recurrence just executes brute-force search over all of the legitimate choices of k:

$$A[S, j] = \min_{k \in S \setminus \{1,j\}} \left(A[S \setminus \{j\}, k] + c_{kj} \right).$$

This recurrence assumes that $|S| \geq 3$. If $|S| = 1$ then $A[S, j]$ is 0 if $S = \{1\}$ and $j = 1$ and is $+\infty$ otherwise. If $|S| = 2$, then the only legitimate choice of k is 1.

The algorithm first solves all subproblems with $|S| = 1$, then all subproblems with $|S| = 2$, ..., and finally all subproblems with $|S| = n$ (i.e., $S = \{1, 2, \ldots, n\}$). When solving a subproblem, the solutions to all relevant smaller subproblems are available for constant-time lookup. Each subproblem can thus be solved in $O(n)$ time. Since there are $O(n2^n)$ subproblems, we obtain the claimed running time bound of $O(n^2 2^n)$.

3 3SAT and Random Search

3.1 Schöning's Algorithm

Recall from last lecture that a 3SAT formula involves n Boolean variables $x_1, \ldots, x_n$ and m clauses, where each clause is the disjunction of three literals (where a literal is a variable or its negation). Last lecture we studied MAX 3SAT, the optimization problem of satisfying as many of the clauses as possible. Here, we'll study the simpler decision problem, where the goal is to check whether or not there is a assignment that satisfies all m clauses. Recall that this is the canonical example of an NP-complete problem (cf., the Cook-Levin theorem).

Naive brute-force search would try all 2^n truth assignments. Can we do better than exhaustive search? Intriguingly, we can, with a simple algorithm and by a pretty wide margin. Specifically, we'll study Schöning's random search algorithm (from 1999). The parameter T will be determined later.

Random Search Algorithm for 3SAT (Version 1)

```
repeat $T$ times (or until a satisfying assignment is found):
    choose a truth assignment $\mathbf{a}$ uniformly at random
    repeat $n$ times (or until a satisfying assignment is found):
        choose a clause $C$ violated by the current assignment $\mathbf{a}$
        choose one the three literals from $C$ uniformly at random, and
          modify $\mathbf{a}$ by flipping the value of the corresponding variable
          (from "true" to "false" or vice versa)
if a satisfying assignment was found then
    return "satisfiable"
else
    return "unsatisfiable"
```

And that's it![1]

3.2 Analysis (Version 1)

We give three analyses of Schöning's algorithm (and a minor variant), each a bit more sophisticated and establishing a better running time bound than the last. The first observation is that the algorithm never makes a mistake when the formula is unsatisfiable — it will never find a satisfying assignment (no matter what its coin flips are), and hence reports "unsatisfiable." So what we're worried about is the algorithm failing to find a satisfying assignment when one exists. So for the rest of the lecture, we consider only satisfiable instances. We use $\mathbf{a}^*$ to denote a reference satisfying assignment (if there are many, we pick one arbitrarily). The high-level idea is to track the "Hamming distance" between $\mathbf{a}^*$ and our current truth assignment $\mathbf{a}$ (i.e., the number of variables with different values in $\mathbf{a}$ and $\mathbf{a}^*$). If this Hamming distance ever drops to 0, then $\mathbf{a} = \mathbf{a}^*$ and the algorithm has found a satisfying assignment.

[1]A little backstory: an analogous algorithm for 2SAT (2 literals per clause) was studied earlier by Papadimitriou. 2SAT is polynomial-time solvable — for example, it can be solved in linear time via a reduction to computing the strongly connected components of a suitable directed graph. Papadimitriou's random search algorithm is slower but still polynomial ($O(n^2)$), with the analysis being a nice exercise in random walks (covered in the instructor's Coursera videos).

A simple observation is that, if the current assignment $\mathbf{a}$ fails to satisfy a clause C, then $\mathbf{a}$ assigns at least one of the three variables in C a different value than $\mathbf{a}^*$ does (as $\mathbf{a}^*$ satisfies the clause). Thus, when the random search algorithm chooses a variable of a violated clause to flip, there is at least a 1/3 chance that the algorithm chooses a "good variable," the flipping of which decreases the Hamming distance between $\mathbf{a}$ and $\mathbf{a}^*$ by one. (If $\mathbf{a}$ and $\mathbf{a}^*$ differ on more than one variable of C, then the probability is higher.) In the other case, when the algorithm chooses a "bad variable," where $\mathbf{a}$ and $\mathbf{a}^*$ give it the same value, flipping the value of the variable in $\mathbf{a}$ increases the Hamming distance between $\mathbf{a}$ and $\mathbf{a}^*$ by 1. This happens with probability at most 2/3.[2]

All of the analyses proceed by identifying simple sufficient conditions for the random search algorithm to find a satisfying assignment, bounding below the probability that these sufficient conditions are met, and then choosing T large enough that the algorithm is guaranteed to succeed with high probability.

To begin, suppose that the initial random assignment $\mathbf{a}$ chosen in an iteration of the outer loop differs from the reference satisfying assignment $\mathbf{a}^*$ in k variables. A sufficient condition for the algorithm to succeed is that, in every one of the first k iterations of the inner loop, the algorithm gets lucky and flips the value of a variable on which $\mathbf{a}, \mathbf{a}^*$ differ. Since each inner loop iteration has a probability of at least 1/3 of choosing wisely, and the random choices are independent, this sufficient condition for correctness holds with probability at least 3^{-k}. (The algorithm might stop early if it stumbles on a satisfying assignment other than $\mathbf{a}^*$; this is obviously fine with us.)

For our first analysis, we'll use a sloppy argument to analyze the parameter k (the distance between $\mathbf{a}$ and $\mathbf{a}^*$ at the beginning of an outer loop iteration). By symmetry, $\mathbf{a}$ agrees with $\mathbf{a}^*$ on at least half the variables (i.e., $k \le n/2$) with probability at least 1/2. Conditioning on this event, we conclude that a single outer loop iteration successfully finds a satisfying assignment with probability at least $p = \frac{1}{2 \cdot 3^{n/2}}$. Hence, the algorithm finds a satisfying assignment in one of the T outer loop iterations except with probability at most $(1-p)^T \le e^{-pT}$.[3] If we take $T = \frac{d \ln n}{p}$ for a constant $d > 0$, then the algorithm succeeds except with inverse polynomial probability $\frac{1}{n^d}$. Substituting for p, we conclude that

$$T = \Theta\left((\sqrt{3})^n \log n\right)$$

outer loop iterations are enough to be correct with high probability. This gives us an algorithm with running time $O((1.74)^n)$, which is already significantly better than the 2^n dependence in brute-force search.

[2]The fact that the random process is biased toward moving farther away from $\mathbf{a}^*$ is what gives rise to the exponential running time. In the case of 2SAT, each random move is at least as likely to decrease the distance as increase the distance, which in turn leads to a polynomial running time.

[3]Recall the useful inequality $1 + x \le e^x$ for all $x \in \mathbb{R}$, used also in Lectures #11 (see the plot there) and #15.

3.3 Analysis (Version 2)

We next give a refined analysis of the same algorithm. The plan is to count the probability of success for all values of the initial distance k, not just when $k \leq n/2$ (and not assuming the worst case of $k = n/2$).

For a given choice of $k \in \{1, 2, \ldots, n\}$, what is the probability that the initial assignment $\mathbf{a}$ and $\mathbf{a}^*$ differ in their values to exactly k variables? There is one such assignment for each of the $\binom{n}{k}$ choices of a set S of k out of n variables. (The corresponding assignment $\mathbf{a}$ agrees with $\mathbf{a}^*$ on S and disagrees with $\mathbf{a}^*$ outside of S.) Since all truth assignments are equally likely (probability 2^{-n} each),

$$\mathbf{Pr}[\text{dist}(\mathbf{a}, \mathbf{a}^*) = k] = \binom{n}{k} 2^{-n}.$$

We can now lower bound the probability of success of an outer loop iteration by conditioning on k:

$$\begin{aligned}
\mathbf{Pr}[\text{success}] &= \sum_{k=0}^{n} \mathbf{Pr}[\text{dist}(\mathbf{a}, \mathbf{a}^*) = k] \cdot \mathbf{E}[\text{success} \mid \text{dist}(\mathbf{a}, \mathbf{a}^*) = k] \\
&\geq \sum_{k=0}^{n} \binom{n}{k} 2^{-n} \left(\frac{1}{3}\right)^k \\
&= 2^{-n}(1 + \tfrac{1}{3})^n \\
&= \left(\frac{2}{3}\right)^n,
\end{aligned}$$

where the penultimate equality follows from a slick application of the binomial formula.[4]

Thus, taking $T = \Theta((\frac{3}{2})^n \log n)$, the random search algorithm is correct with high probability.

3.4 Analysis (Version 3)

For the final analysis, we tweak the version of Schöning's algorithm above slightly, replacing "repeat n times" in the inner loop by "repeat $3n$ times." This only increases the running time by a constant factor.

Our two previous analyses only considered the cases where the random search algorithm made a beeline for the reference satisfying assignment $\mathbf{a}^*$, never making an incorrect choice of which variable to flip. There are also other cases where the algorithm will succeed. For example, if the algorithm chooses a bad variable once (increasing $\text{dist}(\mathbf{a}, \mathbf{a}^*)$ by 1), but then a good variable $k + 1$ times, then after these $k + 2$ iterations $\mathbf{a}$ is the same as the satisfying assignment $\mathbf{a}^*$ (unless the algorithm stopped early due to finding a different satisfying assignment).

[4]I.e., the formula $(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$.

For the analysis, we'll focus on the specific case where, in the first $3k$ inner loop iterations, the algorithm chooses a bad variable k times and a good variable $2k$ times. This idea leads to

$$\mathbf{Pr}[\text{success}] \geq \sum_{k=0}^{n} 2^{-n} \binom{n}{k} \binom{3k}{k} \left(\frac{1}{3}\right)^{2k} \left(\frac{2}{3}\right)^{k}, \tag{1}$$

since the probability that the random local search algorithm chooses a good variable $2k$ times in the first $3k$ inner loop iterations is at least $\binom{3k}{k}(\frac{1}{3})^{2k}(\frac{2}{3})^{k}$.

This inequality is pretty messy, with no less than two binomial coefficients complicating each summand. We'll be able to handle the $\binom{n}{k}$ terms using the same slick binomial expansion trick from the previous analysis, but the $\binom{3k}{k}$ terms are more annoying. To deal with them, recall *Stirling's approximation* for the factorial function:

$$n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^{n}\right).$$

(The hidden constant is $\sqrt{2\pi}$, but we won't need to worry about that.) Thus, in the grand scheme of things, $n!$ is not all that much smaller than n^n.

We can use Stirling's approximation to simplify $\binom{3k}{k}$:

$$\begin{aligned}\binom{3k}{k} &= \frac{(3k!)}{(2k)!k!}\\ &= \Theta\left(\frac{\sqrt{3k}}{\sqrt{k}\sqrt{2k}} \cdot \frac{(\frac{3k}{e})^{3k}}{(\frac{k}{e})^{k}(\frac{2k}{e})^{2k}}\right)\\ &= \Theta\left(\frac{1}{\sqrt{k}} \cdot \frac{3^{3k}}{2^{2k}}\right).\end{aligned}$$

Thus,

$$\underbrace{\binom{3k}{k}}_{=\Theta(3^{3k}/2^{2k}\sqrt{k})} \left(\frac{1}{3}\right)^{2k} \left(\frac{2}{3}\right)^{k} = \Theta\left(\frac{2^{-k}}{\sqrt{k}}\right).$$

Substituting back into (1), we find that for some constant $c > 0$ (hidden in the Θ notation),

$$\begin{aligned}
\mathbf{Pr}[\text{success}] &\geq \sum_{k=0}^{n} 2^{-n} \binom{n}{k} \binom{3k}{k} \left(\frac{1}{3}\right)^{2k} \left(\frac{2}{3}\right)^{k} \\
&\geq c 2^{-n} \sum_{k=0}^{n} \binom{n}{k} \frac{2^{-k}}{\sqrt{k}} \\
&\geq \frac{c}{\sqrt{n}} 2^{-n} \sum_{k=0}^{n} \binom{n}{k} 2^{-k} \\
&= \frac{c}{\sqrt{n}} 2^{-n} \left(1 + \frac{1}{2}\right)^{n} \\
&= \frac{c}{\sqrt{n}} \left(\frac{3}{4}\right)^{n}.
\end{aligned}$$

We conclude that with $T = \Theta\left(\left(\frac{4}{3}\right)^n \sqrt{n} \log n\right)$, the algorithm is correct with high probability.

This running time of $\approx (\frac{4}{3})^n$ has been improved somewhat since 1999, but this is still quite close to the state of the art, and it is an impressive improvement over the $\approx 2^n$ running time require by brute-force search. Can we do even better? This is an open question. The *exponential time hypothesis (ETH)* asserts that every correct algorithm for 3SAT has worst-case running time at least c^n for some constant $c > 1$. (For example, this rules out a "quasi-polynomial-time" algorithm, with running time $n^{\text{polylog}(n)}$.) The ETH is certainly a stronger assumption than $P \neq NP$, but most experts believe that it is true.

The random search idea can be extended from 3SAT to k-SAT for all constant values of k. For every constant k, the result is an algorithm that runs in time $O(c^n)$ for a constant $c < 2$. However, the constant c tends to 2 as k tends to infinity. The *strong exponential time hypothesis (SETH)* asserts that this is necessary — that there is no algorithm for the general SAT problem (with k arbitrary) that runs in worst-case running time $O(c^n)$ for some constant $c < 2$ (independent of k). Expert opinion is mixed on whether or not SETH holds. If it *does* hold, then there are interesting consequences for lots of different problems, ranging from the prospects of fixed-parameter tractable algorithms for NP-hard problems (Section 1) to lower bounds for classic algorithmic problems like computing the edit distance between two strings.