> *Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign." have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and And, and And and and, and and and And, and And and and, and and and Chips, as well as after Chips?*[1]
>
> — Martin Gardner, *Aha! Insight* (1978)

*4 Efficient Exponential-Time Algorithms

In another lecture note, we discuss the class of ***NP-hard*** problems. For every problem in this class, the fastest algorithm anyone knows has an exponential running time. Moreover, there is *very* strong evidence (but alas, no proof) that it is *impossible* to solve any NP-hard problem in less than exponential time—it's not that we're all stupid; the problems really are that hard! Unfortunately, an enormous number of problems that arise in practice are NP-hard; for some of these problems, even *approximating* the right answer is NP-hard.

Suppose we absolutely have to find the exact solution to some NP-hard problem. A polynomial-time algorithm is almost certainly out of the question; the best running time we can hope for is exponential. But *which* exponential? An algorithm that runs in $O(1.5^n)$ time, while still unusable for large problems, is still significantly better than an algorithm that runs in $O(2^n)$ time!

For most NP-hard problems, the only approach that is guaranteed to find an optimal solution is recursive backtracking. The most straightforward version of this approach is to recursively generate *all* possible solutions and check each one: all satisfying assignments, or all vertex colorings, or all subsets, or all permutations, or whatever. However, most NP-hard problems have some additional structure that allows us to prune away most of the branches of the recursion tree, thereby drastically reducing the running time.

4.1 3SAT

Let's consider the mother of all NP-hard problems: 3SAT. Given a boolean formula in conjunctive normal form, with at most three literals in each clause, our task is to determine whether any assignment of values of the variables makes the formula true. Yes, this problem is NP-hard, which means that a polynomial algorithm is almost certainly impossible. Too bad; we have to solve the problem anyway.

The trivial solution is to try every possible assignment. We'll evaluate the running time of our 3SAT algorithms in terms of the number of variables in the formula, so let's call that n. Provided any clause appears in our input formula at most once—a condition that we can easily enforce in polynomial time—the overall input size is $O(n^3)$. There are 2^n possible assignments, and we can evaluate each assignment in $O(n^3)$ time, so the overall running time is $O(2^n n^3)$.

[1] If you ever decide to read this sentence out loud, be sure to pause briefly between 'Fish and and' and 'and and and And', 'and and and And' and 'and And and and', 'and And and and' and 'and and and And', 'and and and And' and 'and And and and', and 'and And and and' and 'and and and Chips'!

Did you notice the punctuation I carefully inserted between 'Fish and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', and 'and' and 'and and and Chips'?

Since polynomial factors like n^3 are essentially noise when the overall running time is exponential, from now on I'll use $\text{poly}(n)$ to represent some arbitrary polynomial in n; in other words, $\text{poly}(n) = n^{O(1)}$. For example, the trivial algorithm for 3SAT runs in time $O(2^n \text{poly}(n))$.

We can make this algorithm smarter by exploiting the special recursive structure of 3CNF formulas:

> A 3CNF formula is either nothing
> or a clause with three literals $\wedge$ a 3CNF formula

Suppose we want to decide whether some 3CNF formula Φ with n variables is satisfiable. Of course this is trivial if Φ is the empty formula, so suppose

$$\Phi = (x \vee y \vee z) \wedge \Phi'$$

for some literals x, y, z and some 3CNF formula Φ'. By distributing the $\wedge$ across the $\vee$s, we can rewrite Φ as follows:

$$\Phi = (x \wedge \Phi') \vee (y \wedge \Phi') \vee (z \wedge \Phi')$$

For any boolean formula Ψ and any literal x, let $\Psi|x$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by assuming x is true. It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge \Psi|x$, which implies that

$$\Phi = (x \wedge \Phi'|x) \vee (y \wedge \Phi'|y) \vee (z \wedge \Phi'|z).$$

Thus, in any satisfying assignment for Φ, either x is true and $\Phi'|x$ is satisfiable, or y is true and $\Phi'|y$ is satisfiable, or z is true and $\Phi'|z$ is satisfiable. Each of the smaller formulas has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

$$T(n) \leq 3T(n-1) + \text{poly}(n),$$

whose solution is $O(3^n \text{poly}(n))$. So we've actually done *worse!*

But these three recursive cases are not mutually exclusive! If $\Phi'|x$ is *not* satisfiable, then x *must* be false in any satisfying assignment for Φ. So instead of recursively checking $\Phi'|y$ in the second step, we can check the even simpler formula $\Phi'|\bar{x}y$. Similarly, if $\Phi'|\bar{x}y$ is not satisfiable, then we know that y must be false in any satisfying assignment, so we can recursively check $\Phi'|\bar{x}\bar{y}z$ in the third step.

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  (x ∨ y ∨ z) ∧ Φ' ← Φ
  if 3SAT(Φ|x)
      return TRUE
  if 3SAT(Φ|x̄y)
      return TRUE
  return 3SAT(Φ|x̄ȳz)
```

The running time off this algorithm obeys the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \text{poly}(n),$$

where poly(n) denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on. The annihilator method gives us the solution

$$T(n) = O(\lambda^n \operatorname{poly}(n)) = \boxed{O(1.83928675522^n)}$$

where $\lambda \approx 1.83928675521\ldots$ is the largest root of the characteristic polynomial $r^3 - r^2 - r - 1$. (Notice that we cleverly eliminated the polynomial noise by increasing the base of the exponent ever so slightly.)

We can improve this algorithm further by eliminating *pure* literals from the formula before recursing. A literal x is *pure* in if it appears in the formula Φ but its negation $\bar{x}$ does not. It's not hard to prove (hint, hint) that if Φ has a satisfying assignment, then it has a satisfying assignment where every pure literal is true. If $\Phi = (x \vee y \vee z) \wedge \Phi'$ has no pure literals, then some in Φ contains the literal $\bar{x}$, so we can write

$$\Phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi'$$

for some literals u and v (each of which might be y, $\bar{y}$, z, or $\bar{z}$). It follows that the first recursive formula $\Phi|x$ has contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we tested the variables y and x in the second and third cases of our previous algorithm:

$$\Phi|x = (u \vee v) \wedge \Phi'|x = (u \wedge \Phi'|xu) \vee (v \wedge \Phi'|x\bar{u}v).$$

Here is our new faster algorithm:

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  if Φ has a pure literal x
      return 3SAT(Φ|x)
  (x ∨ y ∨ z) ∧ (x̄ ∨ u ∨ v) ∧ Φ' ← Φ
  if 3SAT(Φ|xu)
      return TRUE
  if 3SAT(Φ|xūv)
      return TRUE
  if 3SAT(Φ|x̄y)
      return TRUE
  return 3SAT(Φ|x̄ȳz)
```

The running time $T(n)$ of this new algorithm satisfies the recurrence

$$T(n) = 2T(n-2) + 2T(n-3) + \operatorname{poly}(n),$$

and the annihilator method implies that

$$T(n) = O(\mu^n \operatorname{poly}(n)) = \boxed{O(1.76929235425^n)}$$

where $\mu \approx 1.76929235424\ldots$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Naturally, this approach can be extended much further; since 1998, at least fifteen different 3SAT algorithms have been published, each improving the running time by a small amount. As of 2010, the fastest deterministic algorithm for 3SAT runs in $O(1.33334^n)$ time[2], and the fastest

[2]Robin A. Moser and Dominik Scheder. A full derandomization of Schöning's k-SAT algorithm. ArXiv:1008.4067, 2010.

randomized algorithm runs in $O(1.32113^n)$ expected time[3], but there is good reason to believe that these are *not* the best possible.

4.2 Maximum Independent Set

Now suppose we are given an undirected graph G and are asked to find the size of the *largest independent set*, that is, the largest subset of the vertices of G with no edges between them. Once again, we have an obvious recursive algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, the algorithm might look like this.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0
  else
      v ← any node in G
      withv ← 1 + MaximumIndSetSize(G \ N(v))
      withoutv ← MaximumIndSetSize(G \ {v})
      return max{withv, withoutv}.
```

Here, $N(v)$ denotes the *neighborhood* of v: The set containing v and all of its neighbors. Our algorithm is exploiting the fact that if an independent set contains v, then by definition it contains none of v's neighbors. In the worst case, v has no neighbors, so $G \setminus \{v\} = G \setminus N(v)$. Thus, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n-1)+\text{poly}(n) = O(2^n \text{ poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph; an independent set is maximal if every vertex in G is either already in the set or a neighbor of a vertex in the set. If the graph is non-empty, then every maximal independent set either includes or excludes each vertex. Thus, the number of maximal independent sets satisfies the recurrence $M(n) \leq 2M(n-1)$, with base case $M(1) = 1$. The annihilator method gives us $M(n) \leq 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

We can speed up our algorithm by making several careful modifications to avoid the worst case of the running-time recurrence.

- If v has no neighbors, then $N(v) = \{v\}$, and both recursive calls consider a graph with $n-1$ nodes. But in this case, v is in *every* maximal independent set, so one of the recursive calls is redundant. On the other hand, if v has at least one neighbor, then $G \setminus N(v)$ has at most $n-2$ nodes. So now we have the following recurrence.

$$T(n) \leq O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-1) + T(n-2) \end{array} \right\} = O(1.61803398875^n)$$

  The upper bound is derived by solving each case separately using the annihilator method and taking the larger of the two solutions. The first case gives us $T(n) = O(\text{poly}(n))$; the second case yields our old friends the Fibonacci numbers.

- We can improve this bound even more by examining the new worst case: v has exactly one neighbor w. In this case, either v or w appears in every maximal independent set.

[3] Kazuo Iwama, Kazuhisa Seto, Tadashi Takai, and Suguru Tamaki. Improved randomized algorithms for 3-SAT. To appear in *Proc. STACS*, 2010.

However, given any independent set that includes w, removing w and adding v creates another independent set of the same size. It follows that *some maximum independent set includes v*, so we don't need to search the graph $G \setminus \{v\}$, and the $G \setminus N(v)$ has at most $n-2$ nodes. On the other hand, if the degree of v is at least 2, then $G \setminus N(v)$ has at most $n-3$ nodes.

$$T(n) \leq O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-3) \end{array} \right\} = O(1.46557123188^n)$$

The base of the exponent is the largest root of the characteristic polynomial $r^3 - r^2 - 1$.

- Now the worst-case is a graph where every node has degree at least 2; we split this worst case into two subcases. If G has a node v with degree 3 or more, then $G \setminus N(v)$ has at most $n-4$ nodes. Otherwise (since we have already considered nodes of degree 0 and 1), *every* node in G has degree 2. Let u, v, w be a path of three nodes in G (possibly with u adjacent to w). In any maximal independent set, either v is present and u, w are absent, or u is present and its two neighbors are absent, or w is present and its two neighbors are absent. In all three cases, we recursively count maximal independent sets in a graph with $n-3$ nodes.

$$T(n) \leq O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \\ 3T(n-3) \end{array} \right\} = O(3^{n/3}\,\text{poly}(n)) = O(1.44224957031^n)$$

The base of the exponent is $\sqrt[3]{3}$, the largest root of the characteristic polynomial $r^3 - 3$. The third case would give us a bound of $O(1.3802775691^n)$, where the base is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

- Now the worst case for our algorithm is a graph with an extraordinarily special structure: *Every node has degree 2*. In other words, every component of G is a cycle. But it is easy to prove that the largest independent set in a cycle of length k has size $\lfloor k/2 \rfloor$. So we can handle this case directly in polynomial time, without no recursion at all!

$$T(n) \leq O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \end{array} \right\} = O(1.3802775691^n)$$

Again, the base of the exponential running time is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0

  else if G has a node v with degree 0 or 1
      return 1 + MaximumIndSetSize(G \ N(v))        ⟨⟨≤ n − 1⟩⟩

  else if G has a node v with degree greater than 2
      withv ← 1 + MaximumIndSetSize(G \ N(v))       ⟨⟨≤ n − 4⟩⟩
      withoutv ← MaximumIndSetSize(G \ {v})         ⟨⟨≤ n − 1⟩⟩
      return max{withv, withoutv}

  else ⟨⟨every node in G has degree 2⟩⟩
      total ← 0
      for each component of G
          k ← number of vertices in the component
          total ← total + ⌊k/2⌋
      return total
```

As with 3SAT, further improvements are possible but increasingly complex. As of 2010, the fastest published algorithm for computing maximum independent sets runs in $O(1.2210^n)$ time[4]. However, in an unpublished technical report, Robson describes a *computer-generated* algorithm that runs in $O(2^{n/4}\,\text{poly}(n)) = O(1.1889^n)$ time; just the description of this algorithm requires more than 15 pages.[5]

Exercises

1. (a) Prove that any n-vertex graph has at most $3^{n/3}$ maximal independent sets. *[Hint: Modify the MaximumIndSetSize algorithm so that it lists all maximal independent sets.]*

   (b) Describe an n-vertex graph with exactly $3^{n/3}$ maximal independent sets, for every integer n that is a multiple of 3.

*2. Describe an algorithm to solve 3SAT in time $O(\phi^n\,\text{poly}(n))$, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

[4]Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Measure and conquer: A simple $O(2^{0.288n})$ independent set algorithm. *Proc. SODA*, 18–25, 2006.

[5]Mike Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical report 1251-01, LaBRI, 2001. ⟨http://www.labri.fr/perso/robson/mis/techrep.ps⟩.