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?[0]

— Martin Gardner, *Aha! Insight* (1978)

BMO: *I finished with my computations! As I thought, it's a real language they are speaking based on intonations! [speaking to Chips and Ice Cream] Chips? Chips chips chips! Ice cream?*
Chips: *Chips chips chips!*
Ice Cream: *Ice cream ice cream!*

— "Chips and Ice Cream", *Adventure Time* season 6, episode 34 (April 30, 2015)

For every polynomial-time algorithm you have,
there is an exponential algorithm that I would rather run.

— Alan Perlis (first Turing Award winner)

B
Faster Exponential Algorithms

[Read Chapters 2, 3, and 12 first.]

Status: Unfinished.

There are many useful problems—including but not limited to the set of all NP-hard problems—for which the fastest algorithm known runs in exponential time. Moreover, there is *very* strong evidence (but alas, no proof) that it is *impossible* to solve these problems in less than exponential time—it's not that we're all stupid; the problems really are that hard!

But sometimes we have to solve these problems anyway. In these circumstances, it is important to ask: *Which* exponential? An algorithm that runs in $O(2^n)$ time, while still

[0]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'?

© 2018 Jeff Erickson http://algorithms.wtf

unusable for large instances, is still significantly better than an algorithm that runs in $O(4^n)$ time; an algorithm that runs in $O(1.5^n)$ or $O(1.25^n)$ is better still.

The most straightforward method to find an optimal solution to an NP-hard problem to recursively generate *all* possible solutions via recursive backtracking and check each one: all satisfying assignments, or all vertex colorings, or all subsets, or all permutations, or whatever. However, many (perhaps even 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.

B.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 two variables: n is the number of variables in the input formula, and m is the number of clauses. There are 2^n possible assignments, and we can evaluate each assignment in $O(m)$ time, so the overall running time is $O(2^n m)$. Assuming no clause appears in the input formula more than once—a condition that we can easily enforce in polynomial time—then $m = O(n^3)$, so the running time is at most $O(2^n n^3)$.

Polynomial factors like n^3 are essentially noise when the overall running time is exponential, so from now on I'll write $\operatorname{poly}(n)$ to represent any polynomial function of n; in other words, $\operatorname{poly}(n) := n^{O(1)}$. For example, the trivial algorithm for 3Sat runs in time $O(2^n \operatorname{poly}(n))$.

One Clause

Recall that all 3CNF formulas have the following recursive structure

> A 3CNF formula is either nothing
> or (a clause with three literals) ∧ (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 $\boldsymbol{\Psi|x}$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by setting x to TRUE and simplifying as much as possible. If Ψ has n variables, then $\boldsymbol{\Psi|x}$ has at most $n-1$ variables.

It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge (\Psi|x)$, and therefore

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

Thus, for 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 $\Phi'|x$, $\Phi'|y$, and $\Phi'|z$ has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

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

whose solution is $O(3^n \operatorname{poly}(n))$. 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 $\Phi'|\bar{x}y = (\Phi'|\bar{x})|y$. 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 $\Phi = \varnothing$
      return TRUE
  $(x \vee y \vee z) \wedge \Phi' \leftarrow \Phi$
  if 3SAT($\Phi'|x$)
      return TRUE
  if 3SAT($\Phi'|\bar{x}y$)
      return TRUE
  return 3SAT($\Phi'|\bar{x}\bar{y}z$)
```

The running time off this algorithm obeys the recurrence

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

where $\operatorname{poly}(n)$ denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on.

How do we solve this recurrence? It isn't simple enough to let us guess an accurate solution and check it by induction. Fortunately, there is a standard mechanical solution for all backtracking recurrences of the form

$$T(n) \leq \operatorname{poly}(n) + \sum_{i=1}^{k} T(n - a_i),$$

where $k, a_1, a_2, \ldots, a_k$ are constants. Assume without loss of generality that the constants a_i are indexed in non-increasing order: $a_1 \leq a_2 \leq \cdots \leq a_k$. The fundamental theorem of algebra implies that the ***characteristic polynomial***

$$r^{a_k} - \sum_{i=1}^{k} r^{a_k - a_i}$$

(where r is a formal variable) has exactly k complex roots, some of which may be equal. Let λ be the root with largest magnitude. Then the solution to the recurrence is $\boldsymbol{T(n) = O(\lambda^n \operatorname{poly}(n))}$. This technique is described in more detail in the appendix; in particular, we can actually derived *exact closed form* solutions to backtracking-style recurrences from the complete sequence of characteristic roots and base cases, if the polynomial terms and base cases are specified exactly.

The recurrence $T(n) = T(n-1) + T(n-2) + T(n-3) + \operatorname{poly}(n)$ has characteristic polynomial $r^3 - r^2 - r - 1$, whose roots are approximately

$$1.83928676, \quad -0.419643337 + 0.60629073i, \quad -0.419643337 - 0.60629073i.$$

(Thank you, Wolfram Alpha!) The first root has larger magnitude than the other two, so we get the solution

$$T(n) = O(\lambda^n \operatorname{poly}(n)) = \boldsymbol{O(1.83929^n)}.$$

(Notice that we cleverly eliminated the polynomial noise by ever so slightly increasing the base of the exponent.)

Pure Literals

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 clause 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 $\Phi|x$ contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we eliminated the variables y and x in the second and third cases of our previous algorithm:

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

Our new faster algorithm is shown in Figure **??**. The running time $T(n)$ of this algorithm satisfies the recurrence

$$T(n) = \operatorname{poly}(n) + \max \begin{Bmatrix} T(n-1) \\ 2T(n-2) + 2T(n-3) \end{Bmatrix}$$

```
FASTER3SAT($\Phi$):
  if $\Phi = \varnothing$
      return TRUE
  if $\Phi$ has a pure literal $x$
      return FASTER3SAT($\Phi|x$)
  $(x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi' \leftarrow \Phi$
  if FASTER3SAT($\Phi|xu$)
      return TRUE
  if FASTER3SAT($\Phi|x\bar{u}v$)
      return TRUE
  if FASTER3SAT($\Phi|\bar{x}y$)
      return TRUE
  return FASTER3SAT($\Phi|\bar{x}\bar{y}z$)
```

Figure B.1. A faster backtracking algorithm for 3SAT.

To solve this recurrence, we solve each case separately and take the larger of the two solutions. Here, the comparison is easy; the first case yields a *polynomial* bound, while the second case clearly yields an exponential bound, so for purposes of worst-case analysis, we can simplify the recurrence to $T(n) = O(\text{poly}(n)) + 2T(n-2) + 2T(n-3)$. The solution to this simpler recurrence is

$$T(n) = O(\lambda^n \text{ poly}(n)) = \boldsymbol{O(1.76930^n)},$$

where $\lambda \approx 1.76929235$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Local Search

We can get an even faster algorithm using a different backtracking strategy, called ***local search***. Instead of constructing a satisfying assignment recursively, we search for a path through the set of all assignments that ends at a satisfying assignment.

The ***Hamming distance*** between two different assignments is the number of variables where the two assignments differ.[2] Suppose we could somehow magically find an assignment that is *almost* satisfying, meaning that its Hamming distance from some satisfying assignment is small. Then instead of searching over the entire space of 2^n assignments, we only need to look at the assignments that are close to the assignment we already have. Moreover, we can guide the search by considering only the clauses that are *not* satisfied by our current assignment.

To make this concrete, let A be our current assignment, and suppose there is a satisfying assignment A^* that has Hamming distance r from A. Consider any clause of Φ

[2]Richard Hamming was an early giant of computer and information science, who earned PhD in mathematics at Illinois in 1942, programmed IBM calculating machines for the Manhattan project in 1945, and then went on to create the first error-correcting codes. Arguably the greatest recognition of his achievements is that his name is attached to the number of indices where two bit vectors differ.

that is *not* satisfied by A; at least one the three variables in that clause has a different value in A^*. Thus, if we change that particular variable's value, the resulting assignment has Hamming distance $r-1$ from A^*. Of course, we don't know which variable to change, so we have to try all three of them. And we don't know the target assignment A^*, so we just check at every stage of recursion whether our current assignment is satisfying. The resulting backtracking algorithm, shown in Figure **??**, runs in $O(3^r \operatorname{poly}(n))$ time.

```
LOCAL3SAT(Φ, A, r):
  if A satisfies Φ
      return TRUE
  if r = 0
      return FALSE
  C ← any clause of Φ that A does not satisfy
  x, y, z ← variables of C
  A[x] ← ¬A[x]
  if LOCAL3SAT(Φ, A, r − 1) = TRUE
      return TRUE
  A[x] ← ¬A[x]
  A[y] ← ¬A[y]
  if LOCAL3SAT(Φ, A_y, r − 1) = TRUE
      return TRUE
  A[y] ← ¬A[y]
  A[z] ← ¬A[z]
  return LOCAL3SAT(Φ, A_z, r − 1)
```

Figure B.2. A local search algorithm for 3SAT.

It must be emphasized that the local search algorithm is *not* explicitly considering every assignment within Hamming distance r of the starting assignment A.

Now we apply the following trivial observation: In any satisfying truth assignment, either at least half the variables are TRUE or at least half the variables are FALSE. Thus, *every* satisfying assignment has Hamming distance at most $n/2$ from either the all-TRUE assignment or the all-FALSE assignment. So using the local search strategy, we can solve 3SAT in ***$O(3^{n/2} \operatorname{poly}(n)) = O(1.73206^n)$ time***.

♥Random Starting Assignments

We can improve the local search algorithm further, at the expense of a small probability of a false negative, by repeatedly choosing *random* starting assignments and searching within a small radius of each.

First let's do some preliminary math. Let $\boldsymbol{V(r)}$ denote the number of assignments within Hamming distance r of any fixed assignment. Suppose we run the local search algorithm starting with a random assignment; let $P(r)$ denote the probability that the algorithm finds a satisfying assignment (assuming one exists). A simple counting argument implies that $\boldsymbol{P(r) \geq V(r)/2^n}$. If we choose $N(r)$ random assignments and run the local search algorithm starting at each one, the probability of *not* finding a satisfying assignment (if one exists) is *at most* $(1-P(r))^{N(r)}$. Thus, if we set $\boldsymbol{N(r) = n2^n/V(r)}$, this failure probability is at most

$$(1-P(r))^{n2^n/V(r)} \leq \left(\left(1-\frac{V(r)}{2^n}\right)^{2^n/V(r)}\right)^n \leq e^{-n}.$$

The overall running time of our algorithm is

$$O(N(r)\,3^r \operatorname{poly}(n)) \;=\; O\!\left(\frac{n}{P(r)}\cdot 3^r \operatorname{poly}(n)\right) \;=\; O\!\left(\frac{2^n 3^r}{V(r)}\operatorname{poly}(n)\right).$$

To complete the analysis, we need to estimate the volume $V(r)$ and choose an appropriate value of r. For any constant $0 < \alpha < 1$, Stirling's approximation $n! \sim \sqrt{2\pi n}\,(n/e)^n$ implies the lower bound

$$V(\alpha n) \;=\; \sum_{i=0}^{r}\binom{n}{i} \;=\; \Omega\big(H(\alpha)^n/\sqrt{n}\big)$$

where $H(\alpha)$ is the *binary entropy* function:

$$H(\alpha) = \left(\frac{1}{\alpha}\right)^{\alpha}\left(\frac{1}{1-\alpha}\right)^{1-\alpha}.$$

Thus, if we set $r = \alpha n$, the running time of our algorithm becomes

$$O\!\left(\frac{2^n 3^{\alpha n}}{H(\alpha)^n}\operatorname{poly}(n)\right) \;=\; O\big(\big(2(3\alpha)^{\alpha}(1-\alpha)^{1-\alpha}\big)^n \operatorname{poly}(n)\big)$$

We can find the value of α that minimizes the base of the exponential part of this upper bound using a bit of calculus.

$$\begin{aligned}
\frac{d}{d\alpha}\ln\big((3\alpha)^{\alpha}(1-\alpha)^{1-\alpha}\big) &= \frac{d}{d\alpha}\big(\alpha\ln(3\alpha) + (1-\alpha)\ln(1-\alpha)\big)\\
&= \ln(3\alpha) - \ln(1-\alpha) = 0\\
\implies \alpha &= 1/4.
\end{aligned}$$

Finally, if we set $\alpha = 1/4$ (or equivalently, $r = n/4$), the algorithm runs in

$$O\big(\big(2(3/4)^{1/4}(3/4)^{3/4}\big)^n \operatorname{poly}(n)\big) \;=\; \boldsymbol{O\big((3/2)^n \operatorname{poly}(n)\big)}\ \textbf{\textit{time}}.$$

Further Extensions

Naturally, these approaches 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; while a few of these improvements use pure backtracking, most of them are based on some variant of local search. For example, in 1999, Uwe Schöning the running time of the randomized local search algorithm to $O((4/3)^n \operatorname{poly}(n))$, still with an exponentially small error probability, by replacing the backtracking search with a random walk of length $3n$. As of 2016, the fastest deterministic algorithm for 3Sat runs in $O(1.3303^n)$ time[3], and the fastest *randomized* algorithm runs in $O(1.30704^n)$ expected time[4], but there is good reason to believe that these are *not* the best possible.

There is a thriving industry of practical Sat solvers, which combine backtracking, local search, random restarts, and other more sophisticated algorithmic techniques with domain-specific heuristics. With these tools in hand, most real-world instances of Sat are actually easy to solve in practice; modern Sat solvers routinely resolve instances with millions of variables and clauses. Sat solvers have become an indispensable tool in several areas of computing, including hardware and software verification, operations research, and computational biology. The practically of Sat does not contradict the fact that the fastest algorithms require exponential time *in the worst case*; it just means that *the worst case* is rare in practice. Unfortunately, a proper discussion of practical techniques for Sat is beyond both the scope of this text and the expertise of its author.

B.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, we obtain the following algorithm, where $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors.

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

[3]Kazuhisa Makino, Suguru Tamaki, Masaki Yamamoto. Derandomizing the HSSW algorithm for 3-SAT. *Algorithmica* 67(2):112–124 (2013)

[4]Timon Hertli. 3-SAT faster and simpler: Unique-SAT Bounds for PPSZ gold in general. *SIAM J. Comput.* 43(2):718–729 (2014).

Our algorithm exploits 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 $M(n) \le 2M(n-1)$, with base case $M(1) = 1$. We can easily guess and confirm the solution $M(n) \le 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

Smarter Case Analysis

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

Degree 0. If any vertex 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.

$$\begin{aligned} T(n) &\le \text{poly}(n) + \max \left\{ \begin{matrix} T(n-1) \\ \boldsymbol{T(n-1) + T(n-2)} \end{matrix} \right\} \\ &= O(1.61804^n) \end{aligned}$$

As before, the upper bound is derived by solving each case separately using characteristic polynomials and taking the larger of the two solutions. The first case gives us $T(n) = \text{poly}(n)$; the second case yields our old friends the Fibonacci numbers. Here and in later multi-case recurrences, I've typeset the worst case of the recurrence in **bold**.

Degree 1. We can improve this bound even more by examining the new worst case: some vertex v has exactly one neighbor w. In this case, either v or w appears in every maximal independent set. 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.

$$\begin{aligned} T(n) &\le \text{poly}(n) + \max \begin{Bmatrix} T(n-1) \\ T(n-2) \\ \boldsymbol{T(n-1)+T(n-3)} \end{Bmatrix} \\ &= O(1.46558^n) \end{aligned}$$

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

Degree $\geq$ 3. 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.

$$\begin{aligned} T(n) &\le \text{poly}(n) + \max \begin{Bmatrix} T(n-1) \\ T(n-2) \\ T(n-1)+T(n-4) \\ \boldsymbol{3T(n-3)} \end{Bmatrix} \\ &= O(3^{n/3}\,\text{poly}(n)) \\ &= O(1.44225^n) \end{aligned}$$

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 $O(1.38028^n)$, where the base is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

Degree 2. 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. 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, with no recursion at all!

$$\begin{aligned} T(n) &\le \text{poly}(n) + \max \begin{Bmatrix} T(n-1) \\ T(n-2) \\ \boldsymbol{T(n-1)+T(n-4)} \end{Bmatrix} \\ &= O(1.38028^n) \end{aligned}$$

Again, the base of the exponential running time is the largest root of the characteristic polynomial $r^4 - r^3 - 1$. The final algorithm after all these improvements is shown in Figure **??**.

```
MaximumIndSetSize(G):
  if $G = \varnothing$
      return 0
  else if $G$ has a node $v$ with degree 0 or 1
      return $1 + \text{MaximumIndSetSize}(G \setminus N(v))$    ⟨⟨$\leq n-1$⟩⟩
  else if $G$ has a node $v$ with degree greater than 2
      $withv \leftarrow 1 + \text{MaximumIndSetSize}(G \setminus N(v))$    ⟨⟨$\leq n-4$⟩⟩
      $withoutv \leftarrow \text{MaximumIndSetSize}(G \setminus \{v\})$    ⟨⟨$\leq n-1$⟩⟩
      return $\max\{withv, withoutv\}$
  else ⟨⟨every node in $G$ has degree 2⟩⟩
      $total \leftarrow 0$
      for each component of $G$
          $k \leftarrow$ number of vertices in the component
          $total \leftarrow total + \lfloor k/2 \rfloor$
      return $total$
```

Figure B.3. A fast backtracking algorithm for MaxIndependentSet

Further Improvements

As with 3Sat, there are faster but increasingly complex algorithms for MaxIndependent-Set. For example, in 1977, Bob Tarjan and Anthony Trojanowski published a complex backtracking algorithm that considers several dozen cases, each with its own running time recurrence. Most of these cases are clearly dominated by others; for example, even without solving the recurrences, it's easy to see that the recurrence $T(n) \leq \text{poly}(n) + T(n-4) + T(n-5)$ has a smaller solution than $T(n) \leq \text{poly}(n) + T(n-3) + T(n-5)$. After discarding all such dominated cases, Tarjan and Trojanowski are left with the rather impressive recurrence shown in Figure **??**. The solution to this recurrence is $T(n) = O(1.25603^2) = O(2^{n/3})$, where the base of the exponent is the largest root of the characteristic polynomial $r^{10} - r^8 - r^2 - 2$ of the third case (in bold).

As of 2016, the fastest published algorithm for computing maximum independent sets runs in $O(1.2114^n)$ time[5]. However, in an unpublished technical report, Mike 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.[6]

B.3 Dynamic Programming

Warmup: Bellman-Held-Karp TSP in $O(2^n\,\text{poly}(n))$ time. (Leave as exercise?)

★★★

★★★

[5]Nicolas Bourgeois, Bruno Escoffier, Vangelis Paschos, and Johan M. M. can Rooij. Fast algorithms for max independent set. *Algorithmica* 62(1):382–415, 2010.

[6]John Michael 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⟩.

$$T(n) \le \operatorname{poly}(n) + \max \left\{ \begin{array}{c} T(n-2)+T(n-6) \\ T(n-2)+2T(n-8) \\ \boldsymbol{T(n-2)+T(n-8)+2T(n-10)} \\ T(n-3)+T(n-5) \\ T(n-4)+T(n-6)+T(n-8) \\ T(n-4)+2T(n-7) \\ T(n-4)+T(n-8)+3T(n-11) \\ T(n-4)+2T(n-8)+T(n-10) \\ T(n-4)+T(n-9)+2T(n-12) \\ T(n-4)+4T(n-10) \\ T(n-5)+4T(n-9) \\ T(n-5)+6T(n-11)+4T(n-14)+T(n-17) \end{array} \right\}$$

Figure B.4. The (simplified!) running time recurrence for Tarjan and Trojanowski's independent set algorithm.

3-coloring in $O(2^n \operatorname{poly}(n))$ time. Let $OPT(X) = \text{True}$ if there is a 3-coloring in which every vertex in X is red. Then

$$OPT(X) = \begin{cases} \text{False} & \text{if } X \text{ is not an independent set} \\ \text{True} & \text{if } G[V \setminus X] \text{ is bipartite} \\ \bigvee_{v \in V \setminus X} OPT(X \cup \{v\}) & \text{otherwise} \end{cases}$$

Lawlar improved the running time to $O(3^{n/3} \operatorname{poly}(n))$ by observing that we only need to care about $OPT(X)$ when X is a *maximal* independent set. There are $O(3^{n/3})$ maximal independent sets, which can be listed in $O(3^{n/3} \operatorname{poly}(n))$ time.

Alternatively, 3-coloring in $O(2^n \operatorname{poly}(n))$ time by backtracking through the vertices in some fixed whatever-first order. Each vertex must be colored differently from is parent (and possibly other vertices) and without loss of generality, the first tow nodes are red and green, respectively. So we consider only 2^{n-2} possible colorings, but memoization doesn't help. (Can we do better by looking in max-adjacency order?)

★★★

Lawler's algorithm for chromatic number in $O((1+3^{1/3})^n \operatorname{poly} n)$ time: Let $OPT(X)$ denote the chromatic number of the subgraph $G[X]$ induced by the vertex subset $X \subseteq V$. Then

$$OPT(X) = \begin{cases} 0 & \text{if } X = \varnothing \\ 1 + \min \left\{ OPT(X \setminus I) \;\middle|\; \text{maximal ind. set } I \text{ in } G[X] \right\} & \text{otherwise} \end{cases}$$

There are $O(3^{k/3})$ maximal independent sets of size k, which can be listed in $O(3^{k/3} \operatorname{poly}(n))$ time.

★★★

Other techniques to include?

- Horowitz-Sahni partitioning to solve SubsetSum in $O(2^{n/2}\operatorname{poly}(n))$ time? (See also: 3SUM)
- Decision via counting via Inclusion-Exclusion?

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. (a) Describe an algorithm for kSat whose running time satisfies the recurrence

$$T(n) = 2T(n-1) - T(n-k+1) + \operatorname{poly}(n).$$

   (b) Describe an algorithm for kSat whose running time satisfies the recurrence

$$T(n) = T(n-1) + 2T(n-2) - 2T(n-k+1) + \operatorname{poly}(n).$$

   [Hint: Modify the backtracking algorithms for 3Sat described in the text. No, your algorithm will not make negative recursive calls.]

♥3. Describe an algorithm that solves 3Sat in $O(\phi^n \operatorname{poly}(n))$ time, 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. Describe an algorithm to determine whether a graph is 4-colorable in $O(2^n \operatorname{poly}(n))$ time. *[Hint: Consider red and green together, but separately from blue and yellow.]*

5. (a) Prove the following upper bound for all constant $0 < \alpha < 1$:

$$\binom{n}{\alpha n} \le \sum_{i=0}^{\alpha n} \binom{n}{i} < \left(\left(\frac{1}{\alpha}\right)^{\alpha}\left(\frac{1}{1-\alpha}\right)^{1-\alpha}\right)^{n}$$

   [Hint: Expand the expression $(\alpha + (1-\alpha))^n = 1$ using the binomial theorem.]

   (b) Prove the following lower bound for all constant $0 < \alpha < 1$:

$$\binom{n}{\alpha n} \ge \left(\left(\frac{1}{\alpha}\right)^{\alpha}\left(\frac{1}{1-\alpha}\right)^{1-\alpha}\right)^{n} \cdot \Omega(1/\sqrt{n})$$

   [Hint: Use Stirling's approximation: $\sqrt{2\pi n}\,(n/e)^n \le n! \le 2\sqrt{\pi n}\,(n/e)^n$.]

Need a few more exercises.

★★★

© 2018 Jeff Erickson http://algorithms.wtf