CSCC63 – Week 11

This week: We'll another topic in TM functions: counting the number of certificates.

The Class $\#\mathcal{P}$

Consider the following problem:

> **INPUT:** A 3CNF ϕ.
>
> Suppose we flip a fair coin for every variable. If it comes down *heads*, we set that variable to *True*, otherwise we set it to *False*.
>
> **QUESTION:** What is the probability that ϕ is set to *True*?

You can see that this probability is just

$$\frac{\text{The \# of satisfying truth assignments to } \phi}{\text{The \# of truth assignments to } \phi}.$$

If ϕ has k variables, the total number of truth assignments is 2^k. So the question boils down to counting the number of truth assignments that satisfy ϕ. So we really want to answer the question:

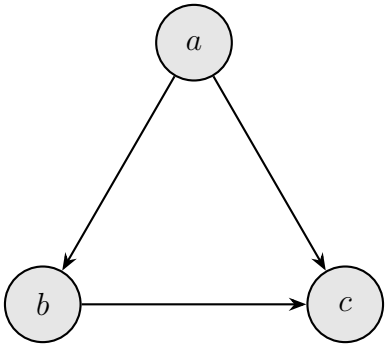
#3SAT (Sharp-3SAT)

> **INPUT:** A 3CNF ϕ.
>
> **QUESTION:** How many satisfying assignments does ϕ have?

This is actually a very important question to us, since it comes up when we try to do statistics on problems that involve, for example, 3CNF formulas. *And a lot of artificial intelligence and machine learning is built around statistical models!*

As an example, here is a very similar type of question that actually shows up in AI:

A *Bayesian Network* is a graphical model we use to estimate correlated probabilities. It is a DAG, e.g.,



Each node is set to either *true* of *false*, with some probability.

(These nodes can represent, for example, the probability that it is raining today, or that it is wet outside)

Some nodes are hidden from us — we might not be able to observe the node a, for example. some nodes are observed — we might be able to see the node c, for example. The problem would then be to infer the probability that a is *true*, given our observation of c.

Of course to get this probability we'd need to know how the nodes are correlated — the the graph comes with a set of probability tables, e.g.,

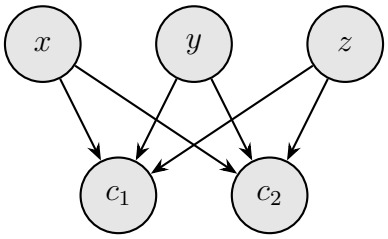
Table for b:

a	*true*	*false*
true	0.1	0.9
false	0.7	0.3

Table for c:

a	b	*true*	*false*
false	*false*	0.1	0.9
false	*true*	0.999	0.001
true	*false*	0.85	0.15
true	*true*	1	0

You can see that we can encode 3SAT into this type of model — we hide the variables and show the clauses, so, for example the formula $(x \vee \overline{y} \vee z) \wedge (\overline{x} \vee y \vee z)$ would be



We'll set c_1 to *true* with probability 1 if x or z is *true*, or if y is *false*. We'll set c_1 to *false* otherwise. Similarly, we'll set c_2 to *true* with probability 1 iff y or z is *true*, or if x is *false*.
You can see that the above problems are fairly similar — solving one would seem to allow us to solve the other. And of course, once we start using one problem to solve another we're just doing reductions.

But what sort of complexity class would characterize these functions?

We've defined the class of functions $\mathcal{FP}$, which can be solved in polynomial time, the class of functions $\mathcal{FNP}$, which take an instance for a problem in $\mathcal{NP}$ as input and return a certificate, and this is a third type of problem. In this type of problem we again take as an input an instance of a problem in $\mathcal{NP}$, but this time we output the *number* of certificates for that instance — e.g., the number of satisfying truth assignments for a 3CNF formula ϕ.

If $L = \left\{x \,\middle|\, \exists c, V(x,c) = \textit{True}\right\}$ is a problem in $\mathcal{NP}$, then $\#L$ is the corresponding function that counts the number of these certificates:

$$\begin{aligned} \#L(x) &= \text{Number of certificates } c \text{ that verify } x \\ &= \left|\left\{c \,\middle|\, V(x,c) = \textit{True}\right\}\right|. \end{aligned}$$

The class $\#\mathcal{P}$ (sharp-$\mathcal{P}$) is the class of these functions:

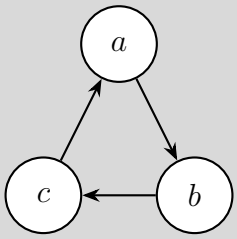
$$\#\mathcal{P} = \left\{ f : \Sigma^* \to \mathbb{N} \,\middle|\, \exists L \in \mathcal{NP}, f = \#L \right\}.$$

Actually, we'll say here that $\#\mathcal{P}$ is the class of functions that are equivalent to some problem of this type - so the probabilistic 3SAT *and Bayesian network examples would also quality.*

But Wait...

There's a small problem with this definition: when I ask you how many certificates a problem instance has, your answer may depend on what you mean by a certificate!

E.g., suppose I give you the graph:



And I ask you how many Hamiltonian cycles this has.

One person might reasonably state that the answer is three: (a, b, c), (b, c, a), and (c, a, b). On the other hand, another might reasonably argue that these are all the same cycle, just rotated around.

...So who's right?

In one sense, there is just the one cycle in this graph, and so the answer should be one. On the other hand, if you look at our verifier for HAM-CYCLE in the first Polytime Reductions handout, the certificate we use is just a list of vertices that form a cycle – all three of the cycles above would be different strings encoding a cycle, and our verifier would accept all three. So there would be three strings (certificates) that our verifier accepts.

On yet another hand, if our verifier took an adjacency matrix encoding the cycle instead of the list of vertices, there would only be one string accepted.

When we say that #3SAT is $\#\mathcal{P}$-complete (which we'll cover in the next page), it's because we can modify the proof of the Cook-Levin theorem from a couple of weeks back so that the number of truth assignments that satisfy our ϕ is equal to the number of certificate strings accepted by our verifier. So we can't reasonably ignore the three-certificates argument.

This means that the number of certificates can depend on the exact encoding of those certificates.

Rather than try to define the "right" certificate for every language, we'll try to be clear in our problem descriptions about exactly what type of certificates we're counting.

How Does $\#\mathcal{P}$ Relate to the Other Classes We Know?

Clearly, $\#\mathcal{P}$ is hard for $\mathcal{NP}$ — if we could count the number of satisfying truth assignments for a 3CNF ϕ, we could also tell if ϕ is satisfiable. Generally, if we could show that $\mathcal{FP}$ were equal to $\#\mathcal{P}$, it would imply that $\mathcal{P}$ was equal to $\mathcal{NP}$.

> *We could also tell if ϕ were unsatisfiable, too — so $\#\mathcal{P}$ is hard for* co-$\mathcal{NP}$*, as well. In fact, it is hard for a much larger set of problems — counting would let us deal with multiple layers of non-determinism. We won't cover the details here, but the relevant theorem is called* Toda's Theorem.

On the other hand, we can see that we can solve problems in $\#\mathcal{P}$ using only a polynomial amount of space: just loop over the possible certificates and count the number of them that accept. So $\mathcal{NP}$ is hard for $\#\mathcal{P}$.

We've been working with the idea of $\mathcal{NP}$-completeness for a while, though, so we can reasonably ask:

Is there such a thing as a $\#\mathcal{P}$-complete problem?

It turns out that there is — a function f is $\#\mathcal{P}$-complete iff:

- $f \in \#\mathcal{P}$, and
- Any polytime solution to f would yield a polytime solution for all problems in $\#\mathcal{P}$.

But how can we show how to use f to solve any problem in $\#\mathcal{P}$?

The trick we'll use here is something called a *parsimonious* reduction…

> *You've seen this before, actually: the modifications you'done to the* Subset-Sum *reduction in week 10's tutorial made it parsimonious.*

A polytime reduction is *parsimonious* if, in addition to being a regular polytime reduction, it preserves the *number* of certificates to the problem.

> *Now, technically a parsimonious mapping gives us a one-to-one relationship between the certificates for the language we're reducing from and the language we're reducing to. But there are interesting examples of languages where this is impossible to achieve: e.g., we can permute the colours of any 3-colouring of a graph G to get another 3-colouring: the number of colourings is always divisible by 3. There are 3CNF formulas with only one satisfying assignment, and so there is no one-to-one mapping from* 3SAT *to* 3COL*. But there is a one-to-six mapping: for every satisfying truth assignment for ϕ there would be exactly six 3-colourings of G.*
>
> It's slightly non-standard terminology, but in this course we'll refer to such a one-to-six mapping as (1-to-6) parsimonious, or as 6-parsimonious. A (1-to-k) parsimonious reduction, or a k-parsimonious reduction, will be defined analogously.

The idea is that, if you have a parsimonious reduction from A to B, you can then use a solution to the counting problem for B to solve the counting problem for A.

> *This means that if we have a parsimonious reduction from, say,* #3SAT *to* #CLIQUE*, we can use* #CLIQUE *to solve* #3SAT*, and so also to do inference over Bayesian networks and other graphical models.*

To start, we can show that #3SAT is $\#\mathcal{P}$-complete — to see why, let's have a brief recap of the proof of Cook's theorem: remember that we spent a class working with a TM operation given in terms of a configuration table:

TM Operation:

#	q_0	w_0	w_1	...	w_{n-1}	␣	␣	...	␣	#
#	$t_{1,0}$	q_1	w_1	...	w_{n-1}	␣	␣	...	␣	#
#										#
#										#
⋮	⋮	⋮			⋮					⋮
#	$t_{n^k-1,0}$	$t_{n^k-1,1}$			...				t_{n^k-1,n^k-1}	#

We used this table to build a 3CNF ϕ whose variables were of the form $x_{i,j,s}$, where

$$x_{i,j,s} = \begin{cases} \textit{true}, & \text{the cell at } (i,j) \text{ contains the character } s, \\ \textit{false}, & \text{otherwise.} \end{cases}$$

Now, you may also recall that we've decided to use a very specific type of NTM N in this reduction. Recall that if $L \in \mathcal{NP}$ then it has polytime verifier V. We'll use this V to write the following NTM:

Let $N =$ "On input $\langle x \rangle$:

1. Nondeterministically choose certificate c for x.
2. Deterministically run V on $\langle x, c \rangle$.
3. If it accepts, *accept*. Otherwise, *reject*.

If we use a resonable methed for non-deterministically chooing the certificte c, there will be exactly one satisfying computation history of N for any one accepted certificate c. The relationship is one-to-one.

This will give us a reduction in which there is exactly one satisfying truth assignment of the certificate we choose is accepted, and no satisfying truth assignment otherwise...

So the reduction in the the proof of Cook's theorem is basically parsimonious — #SAT, at least, is $\#\mathcal{P}$-complete!

Wait a minute — we just said #SAT*, not* #3SAT*. Why didn't we say* #3SAT*?*

Well, remember that up to this point we've just used the variables $x_{i,j,s}$. But at the end of the Cook-Levin reduction we introduced extra variables when changing the ϕ into a 3CNF — for example, we rewrote

$$(a \vee b \vee c \vee d)$$

as

$$(a \vee b \vee x) \wedge (\overline{x} \vee c \vee d),$$

where x is a new variable. Certainly a satisfying truth assignment for one clause can be changed into a satisfying truth assignment for the other — switching between the two subformulas doesn't change the *satisfiability* of a formula ϕ. But it may change then *number* of satisfying assignments: If a, b, c, and d are all *true*, then we can choose any value we want for x, while if a and b are both *false* then we know x must be true.

We can add a bit of overhead, though, to make the same basic reduction work out parsimoniously: once we've split the larger clauses to produce the 3-literal clauses, e.g.,

$$(a \vee b \vee x) \wedge (\overline{x} \vee c \vee d)$$

we can add the new clauses

$$(a \vee b \vee x) \wedge (\overline{a} \vee \overline{a} \vee \overline{x}) \wedge (\overline{b} \vee \overline{b} \vee \overline{x}) \wedge (\overline{x} \vee c \vee d).$$

Note that we make this modification after breaking the clauses into 3*-clauses, not as we do it.*

This has the same overall effect, but now there's one possible truth assignment for x.

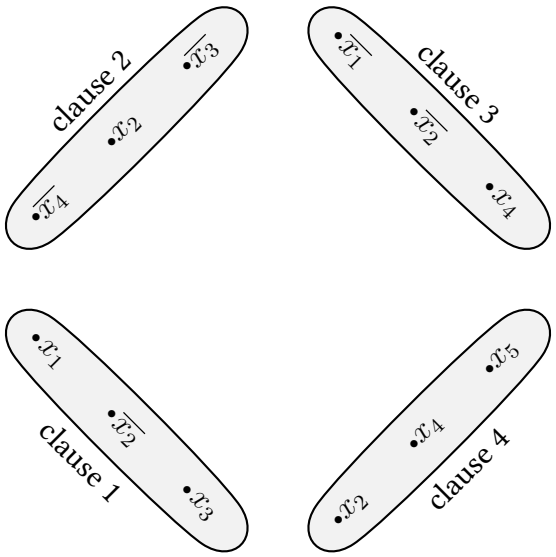
We had a similar problem for ϕ_{move}, but we can just turn those subformulas into CNF form, then use the same approach here.

Overall, it takes a bit more time to describe, but the the takeaway is this: with just a little bit of extra overhead, we can use the proof of the Cook-Levin theorem to show that #3SAT is $\#\mathcal{P}$-complete.

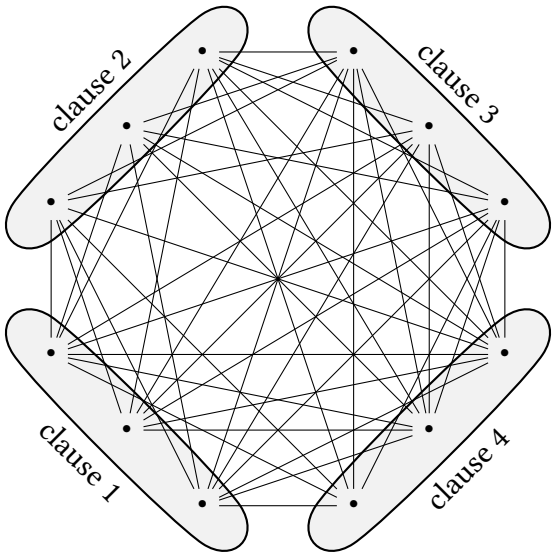
> *This lets us analyze other $\mathcal{NP}$-complete problems as well. You've seen* #SUBSET-SUM *in this week 10's tutorial, but let's see a parsimonious reduction from* #3SAT *to* #CLIQUE *as well.*

That is, let's modify the CLIQUE reduction we saw earlier in the course so that, if we could count the number of k-cliques in the output graph, we could also count the number of satisfying assignments in the input ϕ.

Recall the CLIQUE reduction we've already done in class — given an input 3CNF ϕ, we build a graph with three nodes per clause. So if $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_4} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (x_2 \vee x_4 \vee x_5)$, the graph will have the nodes



We connect nodes from different clauses iff they are compatible (i.e., we can connect an x_1 node to other x_1 nodes, to x_2 and $\overline{x_2}$ nodes, but not to any $\overline{x_1}$ node). So we get a graph like:



Now let's look at some of the truth assignments that we can feed into ϕ. Clearly, a non-satisfying assignment like $x_1 = x_2 = x_3 = x_4 = x_5 = F$ can't be matched up with any 4-clique in this graph. That's why the reduction works.

But how many 4-cliques are consistent with the truth assignment $x_1 = x_2 = x_3 = x_4 = x_5 = T$?

Any such clique will need exactly one node from each of the clause groups, and there are:

- Two ways of choosing a true literal from clause 1 (x_1 or x_3).
- One way of choosing a true node from clause 2 (x_2).

- One way of choosing a true node from clause 3 (x_4).
- Three ways of choosing a true node from clause 4 (x_2, x_4, and x_5).

So there are six cliques consistent with this truth assignment.
On the other hand, if we look at the satisfying truth assignment $x_1 = x_2 = x_3 = x_4 = F, x_5 = T$, then there is:

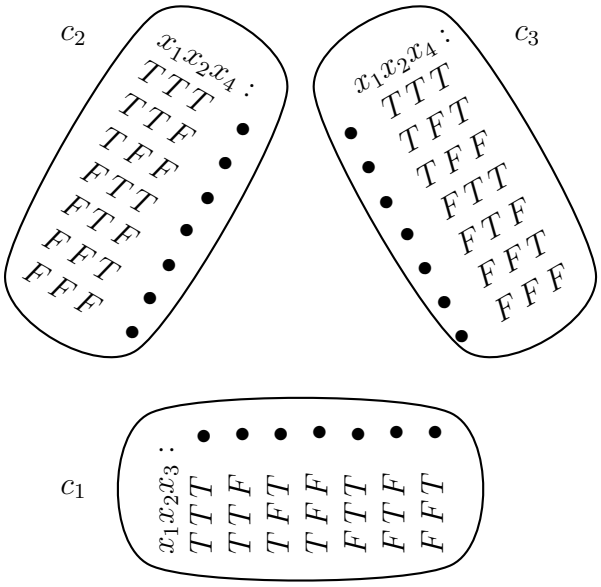
- One way of choosing a true literal from clause 1 ($\overline{x_2}$).
- Two ways of choosing a true node from clause 2 ($\overline{x_1}$ or $\overline{x_3}$).
- Two ways of choosing a true node from clause 3 ($\overline{x_1}$ or $\overline{x_2}$).
- One way of choosing a true node from clause 4 (x_5).

So there are four cliques consistent with this truth assignment.

What this means is that if I were to tell you that I've got some other ϕ', that I've run it through this reduction, and that the resulting G' has 36 k'-cliques, can you tell me how many satisfying truth assignments the 3CNF ϕ' had?

> *It's actually worse than that — some cliques may be consistent with multiple truth assignments, too! So we really can't use one count to predict the other.*

But we can make a simple change to the reduction to get around this issue: instead of having three nodes per clause (one per literal), we can have seven nodes (one per satisfying truth assignment to those literals). E.g., with the 3CNF $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4)$, we'd get the following nodes:



As before, we'll let the resulting graph be G and we'll let k be the number of clauses in ϕ. We'll return $\langle G, k \rangle$.
We can see that any satisfying assignment to our input ϕ will be associated with exactly one node per clause group. Since all of these nodes will be consistent, they will form a k-clique. We can also see that non-satisfying truth assignments will not give us any k-clique.

On the other hand, every k-clique must take one node from every clause group. Since these nodes must be consistent with each other, the variable assignment can be extended to form a satisfying truth assignment for ϕ. Since a variable only shows up in G if it shows up in ϕ, there is exactly one such assignment.

So there is a one-to-one relationship between the satisfying truth assignments to ϕ and the k-cliques of G. This sort of linear relationship between certificates is what we're looking for in a *parsimonious* reduction.

So there are $\#\mathcal{P}$-complete problems, and now we know what a few of them are.

Well, if we're looking for problems that are $\#\mathcal{P}$-complete, it's not too much of a surprise that the counting versions $\mathcal{NP}$-complete problems would fit the bill. But here's something that might surprise you:

> *There are problems in $\mathcal{P}$ whose counting versions are $\#\mathcal{P}$-complete!*

This shouldn't actually be too surprising, since counting can give us information about lots of related problems. Consider this: if I give you a 3CNF ϕ, e.g.,

$$\phi = (x \vee \overline{y} \vee z) \wedge (x \vee \overline{x} \vee y),$$

you'll generally have a hard time telling if it's satisfiable…

But it's pretty easy to tell if it's *falsifiable*. If you manage to set any clause to *false*, the whole thing is set to *false*, as well.

Now, the second clause above is not falsifiable, since no matter what value x takes, one of x or $\overline{x}$ will hold. But the first clause can definitely be set to *false*: just set $x = z =$*false* and $y =$*true*.

To check if ϕ is falsifiable, we really just have to check every clause independently.

On the other hand we can see that, for any k-variable ϕ, we know that

$$\begin{array}{rr} & \text{\# of falsifying truth assignments} \\ + & \text{\# of satisfying truth assignments} \\ \hline = & 2^k \end{array}$$

This means that if we can tell the number of falsifying assignments to ϕ, we can also count its number of satisfying truth assignments.

There are lots of other examples — the perfect matching problem (which we haven't covered, but which you can look up) is also a problem in $\mathcal{P}$ whose counting variant is $\#\mathcal{P}$-complete, as is the language PATH.

So is the language CYCLE:

> **INPUT:** A directed graph $G = (V, E)$.
>
> **QUESTION:** Is there a cycle in G?

You can certainly answer this question in polytime: just use a DFS algorithm on every vertex, as explained in CSCB63.

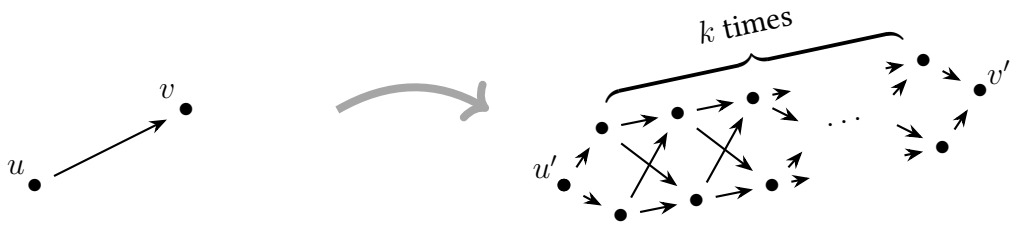
But we can reduce to the counting variant #CYCLE of this problem from HAM-CYCLE, and so show that it is $\mathcal{NP}$-hard.

> *In fact, since you can show that* #HAM-CYCLE *is $\#\mathcal{P}$-complete, we know that* #CYCLE *is $\#\mathcal{P}$-complete, as well. The reduction isn't parsimonious, though.*

Here's the idea: suppose you give me a directed graph G, and want to know if it has a Hamiltonian cycle.

I'm going to take G, and build from it a new graph G' as an input for #CYCLE.

I'll initially set G' to have the same vertices as G. Now, for every edge uv in G, I'll build the following widget:



So we have k pairs of extra nodes per edge. You can see that, where we originally had one path through this edge from u to v, we now have 2^k — we have to choose which node to pass through in each pair, and this choice is independent.

If we repeat this same construction for every edge in the graph, and keep k constant as we do so, then the exponential blowup in the single edge path that we see above extends to longer paths and cycles: if we want to trace a path of length ℓ in G, we have $2^{\ell k}$ choices about how to do so in G'.

On the other hand, you can see that every cycle in G' corresponds to a cycle in G — take any cycle in G' and "flatten" it back into the edges from G, and you'll get a cycle.

Suppose that we set n to be $|V|$. Then, if you have a cycle of length n in G, you'll get at least 2^{kn} cycles in G'.

What if you don't have a cycle of length n?

Well, in that case the longest cycle you can have is of length $n-1$. So how many cycle of this form can we find?

If we take any cycle and start listing its vertices (start anywhere you want), you have up to n possibilities for the first, $n-1$ possibilities for the second, and so on, until we see a repeat, at which point we're done the cycle.

If we take an upper bound for this number, we get

$$\underbrace{n \times n \times n \times \cdots \times n}_{n-1 \text{ times}} = n^{n-1}.$$

We could also use the factorial, too, but this makes the calculations easier.

So by choosing a good value for k, say $k = \lceil n^2 \log_2(n) \rceil$, you can set things up so that a single cycle of length n in G will add more cycles to G' than all n^{n-1} possible cycles of length at most $n-1$. If $k = \lceil n^2 \log_2(n) \rceil$, then the number of cycles in G' that we get from cycles of length at most $n-1$ in G is no more than:

$$\begin{aligned}
n^{n-1} \times 2^{k(n-1)} &= n^{n-1} \times 2^{\lceil n^2 \log_2(n) \rceil (n-1)} \\
&< n^{n-1} \times 2^{(n^2 \log_2(n)+1)(n-1)} \\
&= n^{n-1} \times 2^{(n^2 \log_2(n))(n-1)} \times 2^{n-1} \\
&= n^{n-1} \times n^{(n^2)(n-1)} \times 2^{n-1} \\
&= n^{n-1} \times n^{(n^3-n^2)} \times 2^{n-1} \\
&= n^{n^3-n^2+n-1} \times 2^{n-1} \\
&= n^{n^3-1} \times n^{-n^2+n} \times 2^{n-1} \\
&= n^{n^3-1} \times (1/n)^{n(n-1)} \times 2^{n-1} \\
&= n^{n^3-1} \times \left(\frac{2}{n^n}\right)^{n-1}
\end{aligned}$$

If $n \geqslant 2$ we see that $2/n^n \leqslant 1/2$, and so this whole value is strictly less than n^{n^3-1}.

On the other hand, the number of cycles contributed to G' by every Hamiltonian cycle in G is at least

$$\begin{aligned}
2^{kn} &= 2^{\lceil n^2 \log_2(n) \rceil n} \\
&\geqslant 2^{(n^2 \log_2(n)) \times n} \\
&= n^{(n^2) \times n} \\
&= n^{n^3} \\
&> n^{n^3-1}.
\end{aligned}$$

So G' has at least n^{n^3} cycles iff G has a Hamiltonian cycle (in fact, if we're careful we can even use this approach to count the number of Hamiltonian cycles in G).

There is one more function we should cover while talking about $\#\mathcal{P}$-completeness: finding the *permanent* of a matrix.

INPUT: An $n \times n$ matrix A.

QUESTION: What is the permanent of A?

OK, so what is the permanent of a matrix?

Remember the idea of the *determinant* of a matrix from linear algebra — e.g.,

$$\begin{aligned} det\left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right) &= \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} \\ &= 1 \times 4 - 2 \times 3 \\ &= -2 \end{aligned}$$

and

$$\begin{aligned} \begin{vmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{vmatrix} &= 1 \begin{vmatrix} 1 & 2 \\ 3 & 1 \end{vmatrix} - 2 \begin{vmatrix} 3 & 2 \\ 2 & 1 \end{vmatrix} + 3 \begin{vmatrix} 3 & 1 \\ 2 & 3 \end{vmatrix} \\ &= 1 \times (1 \times 1 - 2 \times 3) - 2(3 \times 1 - 2 \times 2) + 3 \times (3 \times 3 - 1 \times 2) \\ &= 18 \end{aligned}$$

The permanent is the same thing, but we don't flip the signs of the terms for odd permutations:

$$\begin{aligned} perm\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \\ &= 1 \times 4 + 2 \times 3 \\ &= 10 \end{aligned}$$

and

$$\begin{aligned} & perm\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{pmatrix} \\ &= 1 \times perm\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} + 2 \times perm\begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix} + 3 \times perm\begin{pmatrix} 3 & 1 \\ 2 & 3 \end{pmatrix} \\ &= 1 \times (1 \times 1 + 2 \times 3) + 2(3 \times 1 + 2 \times 2) + 3 \times (3 \times 3 + 1 \times 2) \\ &= 54 \end{aligned}$$

> *The determinant has a nice geometric interpretation in terms of the area of a particular n-dimensional object, and it is useful for problems in linear algebra. The permanent has neither property. But it is useful for counting problems in combinatorics.*

Now, we can calculate the determinant for a matrix in a reasonable amount of time — you learned how to do this in linear algebra. But but it turns out that calculating the permanent is $\#\mathcal{P}$-complete: you can do a reduction from #3SAT into the permanent problem for $(0, 1)$-matrices. We won't go over that reduction, but you should know it exists.

A Small Extra Idea. . .

Actually, there are several "normal" problems in mathematics whose exact solutions are $\#\mathcal{P}$-hard to solve. Many of these problems involve implicit or explicit polynomials (the permanent

problem is one of these). Complex implicit polynomials can also show up when calculating partial derivitives, or when calculating integrals.

Suppose we are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of non-negative integers.

If we look at the integral

$$\int_{x=0}^{2\pi} \frac{1}{2\pi} \Big[\prod_{i=1}^{n} 2\cos(a_i x) \Big] dx,$$

then by using the identity $\cos\theta = \frac{e^{i\theta}+e^{-i\theta}}{2}$, we can evaluate this integral to get the number of solutions to the PARTITION instance $\langle S \rangle$!

We don't assume knowledge about complex analysis in this course, so if you don't know this identity don't worry about it. But if you are familiar with it, you may find it interesting to go through the calculation to see why this happens.

How Hard is it to Calculate Functions in $\#\mathcal{P}$?

So that's what counting problems look like. As we've said, actually calculating these functions is $\#\mathcal{P}$-hard, and so also $\mathcal{NP}$-hard. So we expect that most of the calculations can't be done in better than exponential time. But what about finding approximations, or even significant digits?

First, let's look at the issue of significant digits. If I have a problem in $\#\mathcal{P}$, for example, #3SAT, and I'm looking at an input, say, a 3CNF ϕ, I might ask, "What is the most significant digit of the number of satisfying assignments to ϕ?"

> *That is, ϕ has 2^k truth assignments, where k is the number of variables. Is the number of satisfying assignments at greater than 2^{k-1}?*

You'll see that the question above is a *yes/no* question: given an input ϕ, are at more than of its truth assignments accepting?

This is the idea of the closest language class to $\#\mathcal{P}$: a language L is in the class $\mathcal{PP}$ (Probabilistic Polynomial Time) iff L has a polytime certificate/verifier pair such that

$$L = \big\{ \langle x \rangle \,\big|\, \text{more than half of the certificates for } x \text{ are accepted.} \big\}$$

> *That is, if the most significant digit of $\#L(\langle x \rangle)$ is a 1.*
>
> *Alternatively, we can read this as saying that L can be decided by a probabilistic TM whose probability of error is strictly less than $1/2$ for all inputs x. The probability is not bounded away from $1/2$ like it is with other classes (there's a class called $\mathcal{BPP}$ – Bounded Probabilistic Polynomial Time – that does this), though, so this is actually harder.*

We won't spend too much time on the class $\mathcal{PP}$, but we'll show that it's related to $\#\mathcal{P}$: You can see that if I can calculate the number of accepting certificates for an input to a problem, then I can tell if this is at least half of the total possible certificates: if I know the number of satisfying truth assignments for a 3CNF ϕ, I can tell if this number is at least 2^{k-1}.

But suppose we can do the converse: suppose we can answer any problem in $\mathcal{PP}$. I want to determine the number of satisfying truth assignments to, say, a k-variable 3CNF ϕ.

What I'll do is try to guess whether the number of satisfying truth assignments is greater than or equal to some number i, where $0 \leqslant i \leqslant 2^k$. If I can do this, I can do a binary search over i to find the actual number of satisfying truth assignments.

So I take as a certificate a triple made up of:

- A truth assignment τ for ϕ,
- a boolean variable b, and
- a number j such that $0 \leqslant j < 2^k$.

I'll write a verifier that will return true iff:

- $b = 0$ and ϕ does not accept τ, or
- $b = 1$ and $j < i$.

You can see that the total number of certificates is $2^k \times 2 \times 2^k = 2^{2k+1}$, and if we can solve any problem in $\mathcal{PP}$, we can tell if more than 2^{2k} of these certificates accept.

Now, suppose that there are c accepting truth assignments to ϕ. You can see that there are two types of accepting certificates to this problem:

- Certificates in which ϕ is not satisfied, and $b = 0$. If this is the case we don't care what j is, so we have $(2^k - c) \times 1 \times 2^k = 2^{2k} - c2^k$ accepting certificates.
- Certificates in which $j < i$, and $b = 1$. If this is the case we don't care whether ϕ is satisfied, so we have $2^k \times 1 \times i$ accepting certificates.

If we put these together, we get $2^{2k} + 2^k(i - c)$ accepting certificates. You'll see that this is greater than 2^{2k}, that is, that more than half of them accept, iff $i > c$. So this lets me test whether the number of satisfying truth assignments is at least i.

You can also use this idea to cast any problem in $\mathcal{NP}$ as a problem in $\mathcal{PP}$. So $\mathcal{NP} \subseteq \mathcal{NP}$. The same thing applies to co-$\mathcal{NP}$.

Since $\mathcal{PS}$ is hard for $\#\mathcal{P}$, it is hard for $\mathcal{PP}$ as well. So we can also say that $\mathcal{PP} \subseteq \mathcal{PS}$. So:

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PP} \subseteq \mathcal{PS}.$$

We suspect that all of these inequalities are strict.

Fun Fact:

You'll notice (and are likely disppointed) that we haven't talked about quantum computing in this course. There is a class of problems called $\mathcal{BQP}$ that represents what we can reasonably do in polynomial time with quantum computers, though. $\mathcal{BQP}$ is a quantum generalization of a probabilistic polynomial time class $\mathcal{BPP}$, and so to talk about it we'd need to work with TMs, probability, and a bit of quantum physics. We know it's contained in $\mathcal{PS}$, but we haven't yet determined how it related to $\mathcal{NP}$ — we have yet to determine if either $\mathcal{BQP}$ or $\mathcal{NP}$ contain the other.

It turns out, though, that $\mathcal{PP}$ is equal to a larger quantum computing class called $\mathcal{P}\wr\int\sqcup\mathcal{BQP}$. This class is what we could calculate if we could condition on unlikely events (that is, if we could post-select) using a quantum program that would otherwise give a language in $\mathcal{BQP}$.

The problem is that conditioning on unlikely events is not likely to be possible to do efficiently, even for quantum computers (if something is exponentially unlikely, you'll need an exponential expected number of samples to see it). So this class is likely much more powerful than we could manage to compute, even with a quantum computer.

All told, this means that we suspect that calculating even the largest significant digit of a function in $\#\mathcal{P}$ is, in general, hard. The story is similar for approximation, but there's a little more to say in this area. But before we talk about that, we'll want to look at approximations of intractable problems in general. So we'll continue next week by looking at approximation algorithms.

The Take-Away from this Lesson:

- We've briefly described the major function class $\#\mathcal{P}$.
- We've introduced $\#\mathcal{P}$-completeness and showed how to use parsimonious reductions to do reductions between $\#\mathcal{P}$ problems.
- We've revisited the proof of the Cook-Levin theorem and argued that #SAT and #3SAT are $\#\mathcal{P}$-complete.
- We've argued that there are decision problems in $\mathcal{P}$ whose counting problems are $\#\mathcal{P}$-complete.
- We've given an argument that #CYCLE is $\#\mathcal{P}$-complete.
- We have introduced the permanent problem, and argued without proof that it is $\#\mathcal{P}$-complete.
- We have introduced the language class $\mathcal{PP}$, and argued that it is of similar difficulty to $\#\mathcal{P}$. We have also described how this class relates to the other language classes we've already seen.

Glossary:

$\#\mathcal{P}$, $\#\mathcal{P}$-complete, Bayesian Networks, Parsimonious reduction, $\mathcal{PP}$

***Languages*:**

CYCLE

***Problems*:**

#3SAT, CYCLE, *#L, Permanent*