CSCC63 – Week 12

This week: Let's have a look at approximation algorithms.

Approximation Algorithms

We've seen the function classes $\mathcal{FP}$, $\mathcal{FNP}$, and $\#\mathcal{P}$. The classes $\mathcal{FNP}$, and $\#\mathcal{P}$, are at least as hard as $\mathcal{NP}$, and so they contain problems that we don't believe are solvable in polynomial time.

But there's something that we might be able to do with functions that we can't do for Boolean decision problems: we might be able to approximate their solutions, even if we can't solve them exactly.

Which is to say, we should also expect these functions to be harder to approximate than optimization versions of problems in $\mathcal{NP}$ (i.e., the problems in $\mathcal{FNP}$). Let's work with the easier class first.

Now, what do we mean when we say we want to approximate the answer to a problem in $\mathcal{FNP}$ or $\#\mathcal{P}$? How do you approximate a truth assignment to a 3CNF ϕ?

Well, we can't really, but we can find a related problem that we can approximate:

- *Given a graph G, what is the size of its smallest vertex cover?*
- *Given a 3CNF ϕ, what is the maximum number of clauses we can satisfy if we input any truth assignment?*
- *Given a graph G, what is the size of its largest clique?*
- *Given a finite multiset S of natural numbers and a target number t, what is the what is the largest sum $s \leqslant t$ that I can find, where $s = \sum_{x \in X}$ for some $X \subseteq S$?*

Let's start by looking at these problems.

First of all, let's see a VERTEX-COVER approximation — we'll try to guess the size of the smallest vertex cover (alternatively, we'll find the smallest vertex cover we can).

Here's one way of finding an approximation:

```
"On the input $\langle G = (V, E) \rangle$:
  1. Let $G' = (V', E') = G$.
  2. Set $S = \varnothing$.
  3. While $E' \neq \varnothing$:
  4.     Pick an edge $e \in E'$ and place both of its endpoints into $S$.
  5.     Remove both of these endpoints from $G'$, as well as all edges adjacent to them.
  6. Return $S$."
```

Now, you can see that what you get back is a vertex cover of G — you can see that we remove edges from G' once they're covered by some vertex of S, and that we keep running the loop until we've removed (i.e., covered) all of the edges.

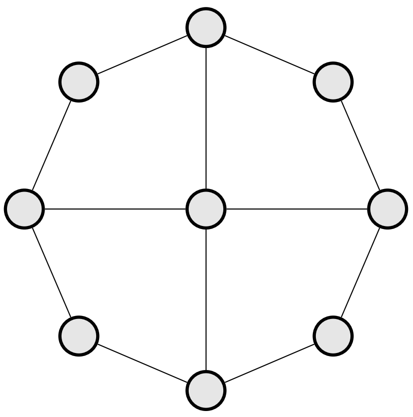
So how good (or bad) is this approximation?

What we really want is the smallest cover. This algorithm won't generally give us the smallest cover. But we can argue that it gives us a set that's no more that twice the size of the best one:

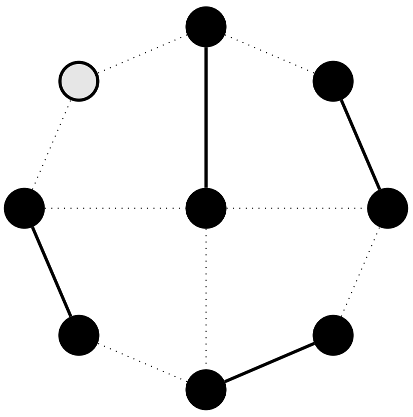
- Let $\hat{S}$ be some optimal (smallest) vertex cover of G.
- Consider any edge e picked in some iteration of line 4 of our approximation.

  Since $\hat{S}$ is a vertex cover of G, it must cover e — that is, at least one of the endpoints of e is in $\hat{S}$.
- By adding the second of the two endpoints (if needed), we can't do more than double the size of $\hat{S}$. So $|S| \leqslant 2 \times |\hat{S}|$.

Now, with a bit of work you can find graphs that give you this blowup, e.g., the graph



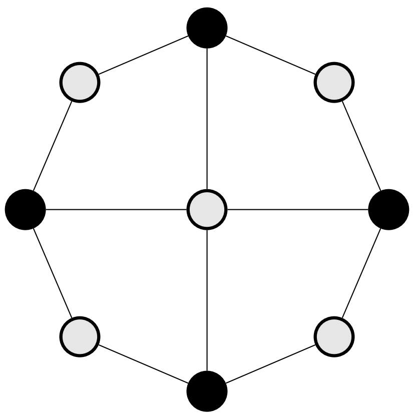
can give the following approximation:



Here, the black nodes are the ones in S, and the dark lines are the ones picked in line 4 of the algorithm.

You can see that our approximate vertex cover is of size 8.

But here's an optimal covering:



That is, our optimal size is exactly half of the size of our approximation.

What we're really interested in is the ratio of these sizes — the number $|S|/|\hat{S}|$ (or sometimes $|\hat{S}|/|S|$). We call this the approximation factor (or approximation ratio). So our algorithm has an approximation factor of 2 (if we were interested in making it as large as possible we might say $1/2$, instead — it's just a matter of how you set up the fraction).

> *It's believed that* 2 *is the best we can do in polytime, as well, unless* $\mathcal{P}$ *equals* $\mathcal{NP}$. *So in some sense, this is optimal.*

So this is one type of behaviour we can see — we can actually approximate some problems to within some constant factor.

> *But can we do this with every problem?*

It turns out that we can't. But to see why, let's look at how we can argue that approximation is hard in the first place. To do that, let's move on to the 3CNF problem above.

In this problem, I give you a 3CNF ϕ, and you want to tell me how many of its clauses I can satisfy.

For an example, let's use the formula

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

Now, this formula clearly isn't satisfiable — no matter how we set x, one of the first two clauses will fail to be true.

But here's a way we can come up with an approximate solution:

- If we set $x =$*true*, we can satisfy one clause. If we set it to *false*, we can satisfy two. So we set $x =$*false*.
- When we do this, we have satisfied the second two clauses, so we remove them from ϕ. We also remove the first clause, since there's no way we can satisfy it now.
- Now, we look at the variable y. If we set it to *true*, we can satisfy the remaining clause. So we set y to *true* and remove the last clause.
- There is no clause left, so we can set z to anything we like.

Once we've done this you can see that we've satisfied $3/4$ of the clauses - and that's actually the best we can do, since there are only four clauses, and we can't satisfy all of them.

In general, we can use this idea for an approximation algorithm — at every step, we take a variable and set it so that it satisfies the largest number of remaining clauses. Once we've done this, we remove the satisfied clauses.

If we do this we won't generally get the optimal answer — for example,

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

would give us the same approximation, but this formula is satisfiable.

On the other hand, every time we set a variable, we satisfy at least half of the remaining clauses that contain it, and so this gives us a $1/2$-approximation algorithm.

> *But can we do better? what's the limit on how well we can do?*
>
> *We can often do a little better — if we don't allow clauses with repeated variables, such as the $(x \vee x \vee x)$ clauses above, we can bump the factor up to $7/8$.*

Here's an idea — if ϕ has ℓ clauses, then we certainly can't approximate the number of satisfiable clauses and reliably be off by less than one, otherwise we could just round off and get the exact number of satisfied clauses — we'd know the answer!

So we can't get a factor that's better than $1 - 1/\ell$.

> *It's known that, as long as $\mathcal{P} \neq \mathcal{NP}$, $7/8$ is actually the best we can do – there's a slightly better algorithm than the simple one we just showed that always gives $7/8$.*

Now, while it's hard to show that this 3SAT variant is hard to approximate to within a factor that's better than $7/8$, we can revisit some of our old reductions and see what happens when we feed a reduction through them.

As an example, we'll look at the language CLIQUE. Given a graph G, how well can we approximate its largest clique? The idea we'll use here is to run through the reduction we did earlier in the course. Recall that when we embed a ϕ into a clique problem, we build a graph with three nodes per clause, and connect nodes iff they are in different clauses and could possibly be consistent.

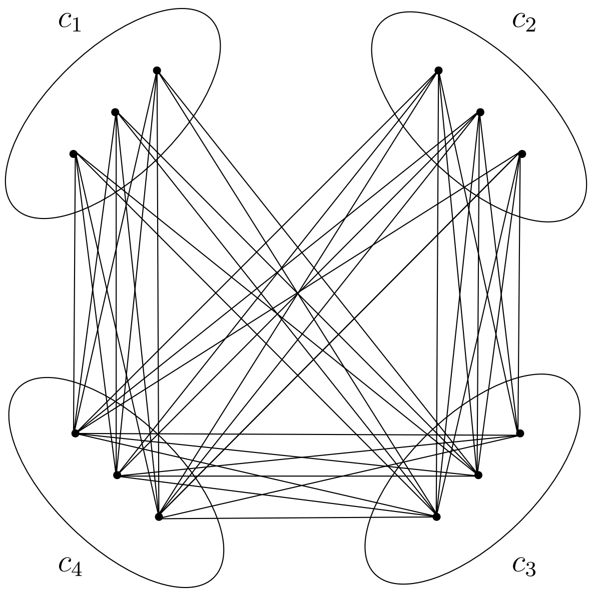
In our $\mathcal{NP}$-completeness proof, we argued that if we took a satisfying assignment and picked the node representing a true literal from each clause, we'd have a k-clique.

But if we start with a non-satisfying assignment, we can still get a clique by picking a true literal from each clause.

Let's see an example. If we start with the 3CNF

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

would give us the graph *(I'll leave out the literal names because it's cluttered enough already — read the literals clockwise)*:



Now, if we were doing the full reduction, we'd look for a 4-clique in this graph. But we already know that our ϕ isn't satisfiable. So the best we can look for in this graph is a 3-clique.

What you'll notice is that the $1 - 1/\ell$ gap from the 3CNF problem has carried over — the small approximation gap from our 3SAT problem gives us another small gap in the clique problem.

> *That is, if we had ℓ clauses in ϕ, then we'd have $n = 3\ell$ vertices in G. So $\ell n/3$. We can't get an approximation factor of better than $1 - 1/\ell$, so this means right away that we can't get a better approximation than $1 - 1/3n$ for* CLIQUE.
>
> *The $7/8$ approximation bound for* 3SAT *gives us a $7/8$ approximation bound for* CLIQUE, *for similar reasons.*

You can see that $1 - 1/3n$ seems to be closer to 1 than $1 - 1/\ell$, if ℓ and n are similar in size. In general, some reductions can magnify the size of an approximation gap, some will reduce it. This gives us different levels of approximability in our problems.

But there's a reason we're using clique as an example…it turns out that if $\mathcal{P} \neq \mathcal{NP}$, then CLIQUE can't be reliably approximated to any constant!

Let's see why:

Basically, we use a self-reduction from CLIQUE to itself to "Pump up" the size of the approximation constant. Suppose we start with an approximation bound of $1/2$ for clique — remember that we know there's a $7/8$ bound that carries over from the 3SAT bound, which we're not proving, but we'll pretend that we've pumped this constant up to $1/2$ and go on from there (for clarity reasons).

So unless $\mathcal{P} = \mathcal{NP}$, which we consider unlikely, there would be no algorithm that can reliably give us an approximation for CLIQUE with a ratio better than $1/2$.

But suppose I had some polytime algorithm M_{Clique} that gave a CLIQUE-approximation with a ratio better than $1/4$. Nothing we've assumed so far says that this is impossible.

You give me a graph G, and I'll try to come up with the best approximation I can for it (i.e., find the largest clique that I can). I'll do it by running the following algorithm:

```
**APPROXIMATE-CLIQUE** = "On input $\langle G = (V, E)\rangle$:
    1. Construct a graph $G' = G[G]$.
       That is, the ndoes in $G'$ are of the form $(u, v)$, where $u, v \in V$. Two nodes $(a, b)$ and $(c, d)$ in $G'$ will be connected iff:
         • $a = c$ and $\{b, d\} \in E$,
         • $\{a, c\} \in E$ and $b = d$, or
         • if both $\{a, c\}$ and $\{b, d\} \in E$.
    2. Let $C'$ be the result of running $M_{Clique}$ on $G'$.
    3. Let $C_1$ be the prjection of $C'$ onto its first variable, and $C_2$ be its projection onto its second variable.
       That is, $C_1 = \{u \mid \exists v, (u, v) \in C'\}$. $C_2$ is analogous.
    4. Let $C$ be the larger of $C_1$ and $C_2$.
    5. Return $C$."
```

So what just happened, and what does it prove?

The key insight here is that taking $G[G]$ blows up size differences in cliques: if I have a clique C of size k in G, then the set C^2 is a size k^2 clique in $G[G]$ (*).

- The size of C^2 is clearly k^2, and
- for any (a, b) and $(c, d) \in C^2$, (a, b) and (c, d) will be connected.

On the other hand, if I have a size k^2 clique in $G[G]$, then at least one of its projections has to be a clique of size at least k (**).

- If both projections were smaller than k, their product would be smaller than k^2.
- Both projection are cliques: if any edge was missing, there would also be a missing edge in C'.

All of this means that my return value from **APPROXIMATE-CLIQUE** is most certainly a clique. But how good an approximation is it?

Well, let's say that $\hat{C}$ is an optimal (largest) clique in G. By (*), $(\hat{C})^2$ is a clique in $G[G]$, and by (**), there cannot be a larger clique. So $(\hat{C})^2$ is an optimal clique in $G[G]$.

When we run line 2 of **APPROXIMATE-CLIQUE**, we get a clique C' that is larger than $1/4$ of the size of the optimum of G' : $|C'| > |(\hat{C})^2|/4 = |\hat{C}|^2/4$ (this is by our assumption on M_{Clique}).

But by (**), this means that our return value C has a size larger than $\sqrt{|\hat{C}|^2/4} = |\hat{C}|/2$.

We've already said that this isn't possible to guarantee if $\mathcal{P} \neq \mathcal{NP}$. We we started by assuming that there isn't any polytime algorithm that'll always let us beat a factor of $1/2$.

On the other hand, **APPROXIMATE-CLIQUE** is polytime: Finding $G[G]$ and the projections C_1 and C_2 are $\mathcal{O}(n^4)$ (as an upper bound), and the running time of M_{Clique} is polytime in the size of $G[G]$, and so also in the size of G. So the whole thing is polytime.

This means that there has to be some reason we can't implement **APPROXIMATE-CLIQUE**, and that reason is our assumption: the assumption that M_{Clique} would give us a polytime 1/4-approximation for CLIQUE.

So not only can we not approximate CLIQUE to within a factor of $1/2$, but we can't even do it to within a factor of $1/4$!

In fact, unless $\mathcal{P} = \mathcal{NP}$ we can't approximate CLIQUE to within any constant factor, or even to within a factor of n^c for any constant c. The proof is an extension of the above argument.

> *Actually, the proof that* 3SAT *can't be approximated to within a factor of* $7/8$ *is similar: instead of a square construction, though, it's a repeated self-reduction that blows up the approximation factor and the clause size by an exponential amount. The reduction breaks if you start with an approximation factor of less than* $7/8$*, though — so it starts with the one-clause inapproximibility result from earlier in the lecture, blows it up to* $7/8$*, then stops. The proof involves error-correcting codes and something called probabilistic proofs. We won't go into detail about it in this course, though.*

Finally, we'll look at the SUBSET-SUM approximation problem above: Given a finite multiset S of natural numbers and a target number t, we want to find a subset $X \subseteq S$ whose sum is as large as possible without going over t.

Due to time constraints we won't go over the details of the approximation algorithms for this problem, but we will note that it is much easier to approximate than the CLIQUE problem, or even the VERTEX-COVER problem…possible to find an algorithm that, given any $\varepsilon > 0$, will give us a $1 - \varepsilon$ approximation of the optimal solution!

So we can approximate this problem arbitrarily well in polynomial time — the catch is that the algorithm is polynomial in n, but the running time of the algorithm also depends on ε. So setting ε low enough to get the actual optimal solution still gives you an exponential algorithm.

This sort of algorithm is called a *Polynomial Time Approximation Scheme* (a PTAS, for short). If the algorithm is polynomial in both n and $1/\varepsilon$, if is a *Fully Polynomial Time Approximation Scheme* (e.g., the algorithm for SUBSET-SUM is polynomial in n and $1/\varepsilon$ — you just have to set $\varepsilon \approx 2^n$ to get the optimal answer). A polynomial randomized version of the same thing is generally referred to as a *Fully Polynomial Randomized Approximation Scheme* (an FPRAS).

So we've seen the types of behaviours we can expect when we try to approximate intractable problems:

- We might be able to approximate the answer we want with any degree of accuracy, but with a cost that depends on that accuracy.
- We might be able to approximate the answer to our problems up to some fixed accuracy, but no further, and
- we might not be able to reliably approximate the answer to our problems at all.

So we can see all of these behaviours in $\mathcal{FNP}$-complete problems. What about $\#\mathcal{P}$-complete problems?

It turns out that we can see some of the similar behaviours:

- Some problems, like the permanent problem, actually can be approximated. The permanent problem can be approximated by an FPRAS.

  This turns out to be related to the idea that we can efficiently take random samples from the certificates for the permanent problem. We won't go into too much detail about this here, though.

- Some problems, such as the #CYCLE problem, can't be approximated to within any accuracy. In particular, #CYCLE can't be approximated to within a factor of $1/2$ if $\mathcal{P} \neq \mathcal{NP}$.

Notably, though, we can see that many of these problems will be on the hard end of the scale: if A has a parsimonious reduction f to B, then we can map instances of $\#A$ to instances of $\#B$ in such a way that the solution $\#B(f(x))$ is always a constant multiple of $\#A(x)$.

This implies that a p-approximation of $\#B$ could be used to give us a p-approximation of $\#A$. So if $\#A$ is hard to approximate, so is $\#B$.

Overall, while we can efficiently approximate the solution to the permanent problem, we can't reasonably expect to be able to do so for most of the problems we face — like inferring probabilities in a Bayesian network.

> *Note: you'll see this is again related to the problem of sampling — if I try to infer a probability on a Bayesian network that's conditioned on a very unlikely event, it's very hard to determine the probability we'll end up with.*
>
> *That's why we can't just try lots of values and average them out. But we won't cover that in detail here.*

A final note on approximations: we believe that $\#\mathcal{P}$ is strictly harder than $\mathcal{NP}$ - in fact, we might not be able to calculate these functions even in a world in which $\mathcal{P}$ were equal to $\mathcal{NP}$.

But, we can approximate these functions quite well with an oracle to $\mathcal{NP}$ — so if it were true that $\mathcal{P}$ and $\mathcal{NP}$ were equal, we could build an FPRAS for every problem in $\#\mathcal{P}$.

And that covers what we need to know right now for approximation — and actually, it ends the material we need to cover for the course.

But let's finih off by asking why it's been so hard to solve the $\mathcal{P}$ versus $\mathcal{NP}$ problem. A lot of approaches have been tried, but we haven't gotten very far. We have, however, learned a few things along the way:

- The first thing we know is that a liar's paradox approach — like the one we used to show HALT is not decidable — won't work here.

  The problem we have is this: if you remember back to the bonus question of assignment 1 (if you did that question), we talked about languages like $\text{HALT}^{\text{HALT}}$ — that is, the halting problem for TMs with access to an oracle for HALT.

  Whether you did that question or not, there was an interesting observation to be made: *A TM with access to an oracle for* HALT *can solve the regular halting problem, but it can't solve the halting problem for other machines like itself!*

  More specifically, if I give a TM access to an oracle, all of the arguments I originally used to argue HALT is undecidable still apply. So no TM formalism is strong enough to solve its own halting problem.

  What this means is that the diagonalization proof we used for HALT *relativizes* — if we paste an oracle onto a machine it still applies.

  So how does this relate to $\mathcal{P}$ versus $\mathcal{NP}$?

  Well, we can show the existence of two different languages A and B such that, for TMs using these oracles, $\mathcal{P}^A = \mathcal{NP}^A$, while $\mathcal{P}^B \neq \mathcal{NP}^B$. So $\mathcal{P}$ versus $\mathcal{NP}$ depends on the fact that we're using TMs (or an equivalent formalism). No proof about these classes can relativize.

  > *Likewise, if you see a proof that claims to solve the issue either way, and that proof doesn't change under an oracle, you know it must be wrong.*
  >
  > *Ironically, the language B is itself defined using diagonalization.*

- When we realized this sort of proof wouldn't work, many researchers tried a different approach — they started to work on another area of theory called *circuit complexity*. If you follow this path, you'll see classes that look like $\mathcal{TC}^0$ and $\mathcal{P}/\mathcal{P}oly$. These are very interesting classes (especially $\mathcal{P}/\mathcal{P}oly$), but we won't cover them in depth here.

  All of our work led to another barrier — a very easy way to try to prove $\mathcal{P}$ versus $\mathcal{NP}$ would be to try to find some easy-to verify property of the languages in $\mathcal{P}$ that is not true for languages that are $\mathcal{NP}$-complete: we could ask if a language is sparse, for example.

  Unfortunately, we've shown that any such proof can be formalized as something we call a *natural proof.* And under reasonably strong assumptions (a slightly stronger version of the $\mathcal{P}$ versus $\mathcal{NP}$ conjecture involving pseudorandom number generators), no such proof can settle the issue of $\mathcal{P}$ versus $\mathcal{NP}$.

  Which is to say that the very difficulty of solving $\mathcal{NP}$-complete problems is also part of the reasons that it's hard to prove that they're hard.

- There are other barriers to the proof that we've found — there is also a barrier to something called an algebraic proof, as well, and we've even found an oracle relative to which the $\mathcal{P}$ versus $\mathcal{NP}$ problem is not solvable in ZFC.

You don't need to know the details of these arguments, though — but it does let you know at a high level what we've run into as we've tried to prove the $\mathcal{P}$ versus $\mathcal{NP}$ conjecture.

And that's the material for the course — as I've said, I'll be available today to answer questions.

The Take-Away from this Lesson:

- We've introduced the idea behind approximation algorithms. We've showed that:
  - Some hard problems can be approximated to within some constant factor (e.g., VERTEX-COVER),
  - Some cannot be approximated to within any constant factor (e.g., CLIQUE), and
  - Some hard problems can be approximated as closely as we like (e.g., SUBSET-SUM) These approximation algorithms have a cost that varies with the approximation factor.
- We've showed that the same range of difficulties applies to approximating functions in $\#\mathcal{P}$. Moreover, most of these functions are believed to be hard to approximate to within any constant factor, but if $\mathcal{P}$ were equal to $\mathcal{NP}$ we could approximate them as well as we liked.

Glossary:

Approximation Factor, FPRAS, PTAS