CSCC63 – Week 6

This week: We've now finished the section of the course dealing with computability, so let's move on to the topic of complexity.

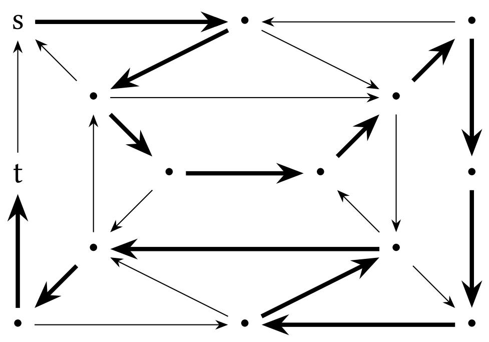
A Motivating Example

Let's look at two somewhat similar problems: HAM-PATH and MST.

Definition: HAM-PATH

> **Instance:** A directed graph G with specified vertices s and t.
>
> **Question:** Does there exist a Hamiltonian path in G from s to t?
> *(i.e., a path from s to t that passes through each vertex exactly once)*

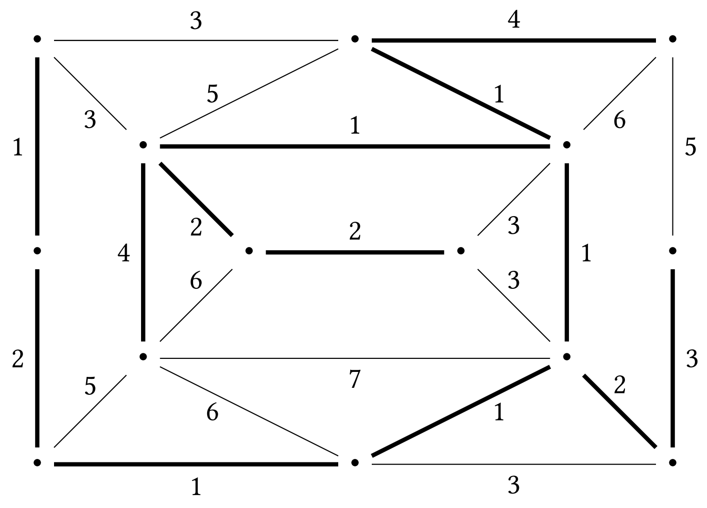
Example:



Definition: MST

> **Instance:** A graph G with weights on its edges, a positive number k.
>
> **Question:** Does G have a minimum spanning tree with weight at most k?

Example:



What We'd Like to Know: *How difficult are these programs to solve?*

Are these problems decidable?

Yes - if a graph G has m edges and n vertices, we can solve HAM-PATH by doing a brute-force search over the $(n-2)!$ orderings of the vertices other than s and t and seeing if any of them is a Hamiltonian path. We can solve MST by doing a brute-force search over the $\mathcal{O}(n^{n-2})$ possible spanning trees and seeing which of them is the minimum one. ?

We'll be using $\mathcal{O}$ notation in this section of the course. Recall that we say that $f(n) = \mathcal{O}(g(n))$ if

$$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathcal{N}, n \geq n_0 \rightarrow f(n) \leq cg(n).$$

We'll say $f(n) = o(g(n))$ if

$$\forall c \in \mathbb{R}^+, \exists n_0 \in \mathcal{N}, n \geq n_0 \rightarrow f(n) \leq cg(n).$$

You've seen this before — we often talk about programs running in $\mathcal{O}(n)$ or $\mathcal{O}(n^2)$ time, and so on.

So our problem is solved — these problems are decidable! Once we write a program that solves them we're done, right?

...But we'd probably like to do a bit better than $\mathcal{O}(n!)$ or $\mathcal{O}(n^{n-2})$ time.

The MST problem can be decided by several much better algorithms, such as Kruskal's algorithm. This algorithm lets us solve the problem in $\mathcal{O}(mn)$ time. Since $m < n^2$, this gives us a $\mathcal{O}(n^3)$ time algorithm. In fact, if we're clever in our implementation we can even get a $\mathcal{O}(n^2 \log(n))$ time implementation.

We can also do better for the HAM-PATH problem, but not much better: we've only been able to get the worst-case running time down to about $\mathcal{O}(n^2 2^n)$.

Question: *Why do we get such a large difference in running time for these two problems?*

Question: *Can we ever hope to build an algorithm for* HAM-PATH *that is even close to the efficiency that we can achieve for the* MST *problem?*

Conjecture: We believe that there is no such algorithm — that there is something fundamental about HAM-PATH such that the laws of logic will not allow us to solve it efficiently.

This is a *conjecture*, not a *theorem*: we do not yet know if the statement is true, but it is widely believed to be so.

If you can figure it out (with a valid proof) one way or the other, you can be awarded \$1M. Even better, you'll get 100% on this course!

Computational Complexity

In order to even start answering the questions posed above — that is, what are the most efficient ways of solving these problems, and why do they have those efficiencies — we'll have to define exactly what we mean when we say a problem is efficiently solvable.

We'll approach this by looking at the programs that solve these problems.

Question: *How can we measure the complexity of a program?*

There are a number of ways of doing this, but in this course we'll mostly look at two:

- **Time complexity:** How many steps does a program take in its computation?
- **Space complexity:** How much memory does a program use in its computation?

We'll spend the next several lectures looking at time complexity.

Recall that a program M has a *worst-case time* of $\mathcal{O}(f(n))$ if $t_M(n) = \mathcal{O}(f(n))$, where $t_M(n)$ is the longest running time M can take for any input of size n.

But we're not interested in a single program — remember that we're asking whether a problem can be computed efficiently.

When we talked about computability, we started by looking at yes/no questions (and their associated languages). We'll do the same thing here.

So we want to know, for a (decidable) language L, how efficiently we can decide L.

Definition: Let's define the class $\text{TIME}\big(t(n)\big)$, for any function $t : \mathbb{N} \to \mathbb{R}_+$ to be

$$\text{TIME}\big(t(n)\big) = \left\{ L \,\middle|\, \begin{array}{c} L \text{ is a language decided by some TM} \\ \text{that has a worst-case time of } \mathcal{O}(t(n)) \end{array} \right\}$$

Now we can argue that functions that grow relatively slowly, such as linear functions, are efficient. So the class $\text{TIME}\big(n\big)$, or even $\text{TIME}\big(n^2\big)$ would seem to be a reasonable description of the problems that can be efficiently solved.

But we should make sure our definition of efficiently solvable programs is robust — it should not vary depending on which TM formalism we use.

Question: *Does our TM model matter when determining which class a problem lies in?*

It turns out that it does: Consider the language $L = \left\{0^n 1^n \,\middle|\, n \in \mathbb{N}\right\}$.

If we use a single-tape TM, we can use the following algorithm (described at a high level — a couple of extra steps, such as marking the left side of the tape, will be needed):

- Move the head left-to-right on the tape and back.
- Each pass, cross off one 0 and one 1.
- If all of the 0s and 1s can be crossed off this way, accept. Otherwise, reject.

Time taken: $\mathcal{O}(n^2)$ (to recognize $0^n 1^n$ — an input length of $2n$ — you need n passes, each taking $4n$ steps).

We can do better, e.g., by using the code:

- Move the head left-to-right on the tape and back.
- Each pass, cross off every other 0 and every other 1.
- If, during any pass, you cross off an even number of 0s and an odd number of 1s or an odd number of 0s and an odd number of 1s, reject.
- If you can cross off every number this way, accept.

Time taken: $\mathcal{O}(n \log_n)$ (this time you cross off half the numbers in each step).

If we use a multi-tape TM, though, we can do even better:

- Move the head left-to-right on the top tape once. Copy each 1 to the lower tape.
- Move both heads back to the start.
- Run another left-to-right pass, this time on both heads. If the top head runs out of $0s$ at the same time as the bottom runs out of 1s, accept. Otherwise, reject.

Time taken by this TM: $\mathcal{O}(n)$.

So we get a difference in time, but the difference is not more than a power of two.

> *We can show that these are the best times we can get for these languages — problem 7.49 in the text shows that any non-regular language will need at least $\Omega(n \log(n))$ time on a single-tape deterministic TM, and so the single-tape algorithm above is asymptotically optimal. Similarly, we'll need at least $\Omega(n)$ time to even read the input, so we can't do better using a multitape TM either.*

To recap from earlier in the course: this is the case for any algorithm we run on a multi-tape TM: if a multi-tape TM M has a worst-case time of $\mathcal{O}(t(n))$, we can simulate it with a single-tape TM M' that has a worst-case time of at worst $\mathcal{O}(t^2(n))$.

In fact, any deterministic TM-equivalent formalism that we know how to build can be simulated by a single-tape TM with a possible blow-up of a constant power.

This means that when we are asking about computation times, TMs formalisms matter, but only up to a constant power.

So when we ask which problems can be solved efficiently:

- Don't ask if they can be solved in, e.g., linear time (depends on formalism).
- Ask if they can be solved in some constant power of linear time: i.e., in polynomial time

Definition: We say the class P is the set of languages that can be decided in polynomial time on a deterministic single tape TM:

$$\mathcal{P} = \bigcup_{k \in \mathbb{N}} \text{TIME}\left(n^k\right).$$

This gives us a class of languages that can be decided efficiently, and which is robust across different TM formalisms.

But… There is one TM formalism that we have talked about that we can't build: the non-deterministic TM.

Remember that we can simulate any NTM M with a single-tape deterministic TM M' with an exponential slowdown. Determining whether we need this slowdown is one of the biggest questions in Computer Science. We'll see more about this soon.

Note: When we looked at the complexity of the algorithms for HAM-PATH and MST, we used the parameters m and n, where m was the number of edges and n was the number of vertices.

Question: *If we define the time complexity of our algorithms based on $|I|$, where I is the input, how does this change the complexity of our algorithms?*

Remember that we have to list both n and m in our input. So $n \leqslant |I|$ and $m \leqslant |I|$. This means we get:

$$n^2 2^n \leqslant |I|^2 2^{|I|}$$

and

$$m \log(n) \leqslant |I| \log |I|$$

So we can use these "natural inputs" to our problem without changing the question of whether the resulting algorithm is in $\mathcal{P}$.

Question: What if the input is an integer written out in n digits?

Every time we add a digit to the end of a number, it grows by a factor of about 10 *(with a small difference depending on the digit added). So if I have an n-digit number x, and I loop from* 1 *to x, this takes* $10^{\mathcal{O}(n)}$ *time.*

Note: Since $\mathcal{P}$ is robust across realizable TM formalisms, we'll allow ourselves to use the running times of those formalisms when we calculate running times, so long as we're clear that we're off by a polynomial factor. So if a problem can be solved in $\mathcal{O}(n^2)$ time in a RAM machine, we'll treat it as an $\mathcal{O}(n^2)$ algorithm unless we really need to expand it out in a single-tape TM. This means we'll be able to write code and then do the usual time analysis on that code.

Complexity and Nondeterministic TMs

So we've seen the class $\mathcal{P}$ and argued that it gives us a robust model of the problems that can be computed efficiently.

However, we also saw that this class doesn't cover all models of computation: we don't consider nondeterministic TMs when deciding which languages are in $\mathcal{P}$, since nondeterministic TMs are not possible to build.

It is worth talking about nondeterministic TMs, though: it turns out that many real-world problems are more naturally expressed in terms of nondeterminism than in terms of determinism.

The language HAM-PATH *that we discussed earler will turn out to be decidable by an "efficient" nondeterministic TM, but we don't believe that it is in $\mathcal{P}$.*

Question: *So how do we measure time complexity for nondeterministic TMs?*

Suppose that N is a nondeterministic TM that decides some language L.

The running time $f(n)$ of N is the maximum number of steps taken on *any branch* of the computation tree that N takes for any input of size n.

Definition: We can define the class $\text{NTIME}\big(t(n)\big)$ using the same approach we used in the deterministic setting: For any function $t : \mathbb{N} \to \mathbb{R}_+$, we say that

$$\text{NTIME}\big(t(n)\big) = \left\{ L \;\middle|\; \begin{array}{l} L \text{ is a language decided by some nondeterministic TM} \\ \text{that has a running time time of } \mathcal{O}(t(n)). \end{array} \right\}$$

As we discussed in the last class, the problems that can be solved in polynomial time are the ones that we consider to be efficient. If we are using nondeterministic TMs, this gives us the following class of languages.

Definition: We say the class $\mathcal{NP}$ is the set of languages that can be decided in polynomial time on a nondeterministic TM:

$$\mathcal{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}\big(n^k\big).$$

Question: *What is the relationship between $\mathcal{P}$ and $\mathcal{NP}$?*

Answer: We don't actually know.

We do have a few details:

- Since every deterministic TM is also a nondeterministic TM, every language in $\mathcal{P}$ is also in $\mathcal{NP}$:
$$\mathcal{P} \subseteq \mathcal{NP}.$$

- Since every nondeterministic TM can be simulated by a deterministic TM with an exponential slowdown, every language in $\mathcal{NP}$ has a deterministic $2^{n^{\mathcal{O}(1)}}$ time algorithm:

Any $L \in \mathcal{NP}$ can be decided in exponential time.

We believe, however, that the classes are not equal: that $\mathcal{P}$ is a proper subset of $\mathcal{NP}$.

Conjecture: $\mathcal{P} \neq \mathcal{NP}$

This is one of the most important open problems in Computer Science.

Question: *How does this relate to what we've already done?*

Let's have a look at what sort of problems we expect to be in $\mathcal{NP}$ (such as HAM-PATH).

We don't expect that HAM-PATH can be solved efficiently: Suppose I give you a graph G and a pair s and t of nodes, and then tell you that G has a Hamiltonian path from s to t. You can't check this very easily, and might well decide not to believe me.

But there is something more I can do: I can tell you what the Hamiltonian path in G actually is. You can check whether I've given you a Hamiltonian path very easily.

Is this starting to sound a bit like the difference between decidability and recognizability?

This idea is common to all problems in $\mathcal{NP}$.

Remember that, since an NTM makes nondeterministic choices as it passes through its computation, its computation follows a tree structure:

$$\begin{tikzcd}
 & & & q_0\$01 \arrow[d] & & & \\
 & & & \$q_1 01 \arrow[dll] \arrow[drr] & & & \\
 & \$xq_2 1 \arrow[dl] \arrow[d] \arrow[dr] & & & & \$0q_4 1 \arrow[dl] \arrow[d] \arrow[dr] & \\
 {} & {} & {} & & {} & {} & {} \\
 & \dots & & & & \dots &
\end{tikzcd}$$

We can make our nondeterministic choices when we face a branch in the computation, or we can decide beforehand what choices we'll make, and then search that path of the tree:

Before we start the calculation, we decide we'll take the central path, then the left, then the right:

$$\begin{tikzcd}
 & & & q_0\$01 \arrow[d, thick] & & & \\
 & & & \$q_101 \arrow[dll, thick] \arrow[drr] & & & \\
 & \$xq_21 \arrow[dl] \arrow[d] \arrow[dr, thick] & & & & \$0q_41 \arrow[dl] \arrow[d] \arrow[dr] & \\
 {} & {} & {} & & {} & {} & {} \\
 & \cdots & & & & \cdots &
\end{tikzcd}$$

This doesn't change the complexity of the computation – it's just an approach that lets us make our decisions sooner rather than later when doing our search of the computation tree. We still have to actually find an accepting computation.

What we're doing is a bit like doing a DFS on the tree, which is allowable since we know the tree has finite depth.

So we can search a single path of the computation tree of a nondeterministic TM by choosing that path to start off, then checking that path on the tree.

And following a single path of the tree is a deterministic process.

Now, if I tell you that a nondeterministic TM N accepts some input x, I'm really saying that there's some branch of its computation tree rooted at x that accepts.

Since we can choose that branch before we run through the tree, I can tell you which branch to choose, and then you can check that branch deterministically.

Note: *If $N \in \mathcal{NP}$, then this branch has a length that's polynomial in $|x|$. So I can tell you which branch to take with a description that's polynomial in $|x|$.*

Alternative Description of $\mathcal{NP}$:
A language $L \in \mathcal{NP}$ if there is a deterministic polytime TM V such that $x \in L$ iff $\exists$ a string C_x that is polynomial in the length of x such that $V(x, C_x)$ accepts.

The TM V is a verifier, and the string C_x is a certificate.

To show a language L is in $\mathcal{NP}$, we follow five steps:

1. Show that L is a decision problem.
2. Give a certificate for L.
3. Show the certificate is polynomial in the input size.
4. Give a verifier for the certificate.
5. Show the verifier is polytime in the input size.

This may be a bit of a confusing description, so let's look at a few examples:

HAM-PATH:

Input: A graph G with specified vertices s and t.

Question: Does there exist a Hamiltonian path in G from s to t?

We covered this in the last class, but you can see that if I give you a G, s, and t, and then describe an actual Hamiltonian path from s to t, you can easily say that the graph has a Hamiltonian path – it's just finding that path that's hard.

So to show HAM-PATH is in $\mathcal{NP}$, we can use the following certificate and verifier:

Certificate: Let the certificate C be an ordered list $v_1, v_2, \dots, v_k$ of nodes in G.

Is this polynomial in the size of G?

Verifier: On input $\langle G, s, t, C\rangle$:

```
1. Check that $v_1 = s$.
2. Check that $v_k = t$.
3. Check that $k = |V|$.
4. For $i = 1$ to $k - 1$:
5.     Check that there is an edge from $v_i$ to $v_{i+1}$.
6. Check that every vertex $v$ in $G$ occurs exactly once in the list.
7. If every one of these checks passes, *accept*. Otherwise *reject*.
```

Is this polynomial in the size of G?

CLIQUE:

Input: A graph $G$and a number $k \in \mathbb{N}$.

Question: Does G have a clique of size k?

A clique S is a collection of vertices in G such that for any $u, v \in S$, there is an edge between u and v.

It turns out that this is also a hard problem – but again, checking it is easy: Suppose I give you a set S of nodes in G. You can easily verify whether it's a clique and whether it has size k.

Earlier we said that an integer k can be exponentially larger than its input size. Why doesn't it bother us here that we're asking for a list of nodes of size k?

Because we're listing out all n nodes anyway, and $k \leqslant n$.

So to show CLIQUE is in $\mathcal{NP}$, we can use the following certificate and verifier:

Certificate: Let the certificate C be a set $v_1, v_2, \ldots, v_r$ of nodes in G.

Is this polynomial in the size of G?

Verifier: On input $\langle G, k, C\rangle$:

```
1. Check that $r \geq k$.
2. For every pair $u, v$ of vertices in $C$:
3.     Check that there is an edge from $u$ to $v$.
4. If every one of these checks passes, accept. Otherwise reject.
```

Is this polynomial in the size of G?

PATH:

Input: A graph G with specified vertices s and t.

Question: Does there exist a path in G from s to t?

Notice that this time we're just asking for a path, not a Hamiltonian path – it doesn't have to pass through every vertex.

It turns out that this problem is not only in $\mathcal{NP}$, but is also in $\mathcal{P}$! But just for fun, let's show it's in $\mathcal{NP}$ anyway.

So to show PATH is in $\mathcal{NP}$ we can use the following certificate and verifier:

Certificate: Let the certificate C be an ordered list $v_1, v_2, \ldots, v_k$ of nodes in G.

Is this polynomial in the size of G?

Verifier: On input $\langle G, s, t, C\rangle$:

```
1. Check that $v_1 = s$.
2. Check that $v_k = t$.
3. For $i = 1$ to $k - 1$:
4.     Check that there is an edge from $v_i$ to $v_{i+1}$.
5. If every one of these checks passes, accept. Otherwise reject.
```

Is this polynomial in the size of G?

3-SAT:

Input: A boolean formula ϕ in 3CNF.
3CNF just means that ϕ looks something like:

$$(x_1 \lor x_2 \lor \overline{x_3}) \land (\overline{x_4} \lor x_3 \lor \overline{x_1}) \land \ldots \land (x_k \lor \overline{x_{k-1}} \lor x_2).$$

It's a conjunction of several clauses, each of which is a disjunction of exactly three boolean variables

Question: Does ϕ have a satisfying assignment?

This is a very important problem. We'll see more of it later.

So to show 3SAT is in $\mathcal{NP}$, we can use the following certificate and verifier:

Certificate: Let the certificate τ be a truth assignment $x_1 = T/F, x_2 = T/F, \ldots, x_k = T/F$ to the variables in ϕ.

Is this polynomial in the size of ϕ?

Verifier: On input $\langle \phi, \tau \rangle$:

1. Set every variable in ϕ according to the truth assignment τ and evaluate the resulting expression.
2. If ϕ evaluates to True with the truth assignment τ, *accept*. Otherwise, *reject*.

Is this polynomial in the size of ϕ?

So these are the sorts of problems that are in $\mathcal{NP}$. We'll be introduced to more later.

What we can see from these examples is that the problems in $\mathcal{NP}$ are the problems that are easy to *check*. This is in comparison to the problems in $\mathcal{P}$, which are easy to *solve*.

Since the problems in $\mathcal{NP}$ are the ones whose solutions can be checked easily, what can make them hard is the difficulty of finding a solution. Also, if there is no solution, there may be no way of showing it quickly.

You may be noticing a symmetry in the complexity classes we're using here and the complexity classes we've talked about before:

The **decidable** languages are the languages that can be solved, while $\mathcal{P}$ is the class of languages that can be solved efficiently.

The **recognizable** languages are the languages that can be verified (proven) with a certificate, while $\mathcal{NP}$ is the class of languages that can be verified (proven) efficiently with a certificate.

This is a useful analogy, and we'll be seeing more similarities in the future (think about question 1 from the week 4 tutorial…).

The Take-Away from this Lesson:

- We started talking about computational complexity.
- We Introduced the idea of the time complexity of algorithms.
- We introduced the classes $\mathcal{P}$ and $\mathcal{NP}$.
- We've introduced several important languages in $\mathcal{NP}$.

Glossary:

$\mathcal{NP}$, $\mathit{NTIME}\big(t(n)\big)$, $\mathcal{P}$, $\mathit{TIME}\big(t(n)\big)$

Languages:

3SAT, CLIQUE, HAM-PATH, PATH, MST