Chapter 34 (II)
On $\mathcal{NP}$-completeness

Besides the Hanoi Tower problem, we also saw the following recursive implementation for the Fibonacci sequence (Page 27, Parallel algorithm), $\text{fib}(n)$, for all $n \geq 0$:

\[
\text{fib}(n)\{
    \quad \text{if } n=0 \text{ return } 0; \\
    \quad \text{if } n=1 \text{ return } 1; \\
    \quad \text{return } \text{fib}(n-1)+\text{fib}(n-2); \\
\}
\]

Let $F(n)$ be the time that it takes to run $\text{fib}(n)$, we saw earlier (Cf. Page 29, Parallel algorithm notes) that

\[
F(n) \geq \text{fib}(n) \approx 1.618^n.
\]

**Question:** Why is this bad?

**Answer:** It is hard, i.e., taking lots of time to get the result. 😞
How bad could it be?

If you send in $n = 200$, it takes at least $1.6^{200}$, about $2^{138}$, time units to calculate $F(200)$.

It would take the NEC Earth Simulator, with a clock speed of 40 trillion instructions per second, about $2^{92}$ seconds to finish, when the earth has already turned into a red giant star.

On the other hand, since $F(n+1) \approx 1.6^{n+1} = 1.6F(n)$, it takes a computer 1.6 times longer to calculate $fib(n + 1)$ as compared to $fib(n)$.

By Moore’s law, computers has been doubling their processing speed every 18 months, roughly 1.6 times every year. Let $n_0$ be the largest number the fastest machine can do this year for $fib(n)$, the fastest machine next year can only solve $fib(n)$ for $n_0 + 1$. 😁

Let’s wait for a year to get the next number... 😊

**Question:** What could be worse? Uncertainty.
We don’t know if it is bad. 😞

We also have seen problems that lie somewhere in between, such as the Traveling Salesman problem (TSP). Check out the course page ...

Hamiltonian problem is a “simpler” version of the TSP, i.e., whether, in a given graph $G$, there is a simple cycle that contains all the vertices exactly once. This problem is related to, but different from, the Könisburg problem: Is there a simple cycle that includes all the edges exactly once? (Check out Page 2 on, also the link under, the Graph Basics chapter)

There are actually quite some similar problems, SAT, Knapsack, Steiner minimum tree,....

For these problems, the best algorithm as we know of is exponential, but we have yet to prove that its complexity is indeed exponential.

The hardest ones are called $NP$-Complete problems: If it cracks, everything follows. 😊
Castle in the clouds...

It may be the case that efficient algorithms do exist for them, but none has been found yet.

It is also possible that the problems are intrinsically hard, but we don’t yet have the techniques to prove that is the case.

A remarkable result is that: if we find an efficient algorithm for any one of the $\mathcal{NP}$-Complete problems, all of them can be solved efficiently.

Since it is too good to be true, the common sense wisdom is that none of the $\mathcal{NP}$-Complete problems is easy, i.e., can be solved in polynomial time. 😞
Settle for less...

When it takes too much time to achieve a best result, we might settle for an approximate one with a significant reduction of the cost.

For example, the *Knapsack* problem that we went through in the greedy method chapter (Page 13) is $\mathcal{NP}$-complete, thus it is almost certain that it takes exponential time to find a best answer.

But it takes polynomial time to find a good answer. For example, as we saw on Page 16 of the *greedy* chapter, in an experiment of 600 cases, an approximation solution, generated with the profit density strategy, came up with a value within 10% of the optimal for 583 cases, and all 600 solutions fell within 25% of the optimal.

We should be happy for such a compromise... 😊
Let’s get serious...

Question: What should be considered easy, i.e., efficiently solvable?

Answer: If, for some polynomial $p(n)$, there exists an algorithm which can solve any instance of the problem of size $n$ in $p(n)$.

Thus, all the “easy” problems that we have been working on fall into this category.

Although it is reasonable to regard a program that requires $\Theta(n^{100})$ intractable, only few problems in reality needs this much.

It is also often the case that, once a polynomial time algorithm is found, a more efficient version will follow.

Once we got it done, an improvement is just an engineering problem: From Model T to Tesla. 😊
Why do we love them?

It is often the case that, if a problem can be solved in one computer model, e.g., one processor with RAM, in polynomial time, it can be so solved in another model.

Finally, the problems in $\mathcal{P}$ enjoy nice closure properties, since polynomials are closed under addition, multiplication, and composition. For example, if both $p(n)$ and $q(n)$ are polynomials, so is $p(n) + q(n), p(n) \times q(n), p(q(n))$, etc..

In particular, if the output of a polynomial-time algorithm is sent into another one as the input, then the composition of the two also takes polynomial time $p(q(n))$.

In this sense, most of the problems that we have been working on in the past, present, and future, can be quickly solved.

*This is why we choose to work on computer science in this computer age.* 😊
We are particularly interested in *decision problems*, for which the answer is either “yes” or “no”. For example, both “Is graph $G$ Hamiltonian?” and “Is the shortest path in graph $G$ between two vertices less than 10?” are decision problems.

A decision problem can be thought of defining a set $X$, of instances of the problems on which the correct answer is “yes”.

For example, the above decision problem for Hamiltonian cycles defines a set of graphs $H = \{G_1, G_2, \cdots, \}$, for each of such graphs some Hamiltonian cycles do exist. These instances are referred to as the “yes”-instances of this problem.

All the other graphs are the “no”-instances.
What is really $\mathcal{P}$?

We further say that a correct algorithm that solves a decision problem accepts “yes”-instance and rejects “no”-instances.

For example, an algorithm for the Hamiltonian cycles problem is correct, if it accepts all the graphs in $H$ and rejects all the other graphs.

**Definition:** $\mathcal{P}$ is the class of decision problems that can be solved by a polynomial-time algorithm.

A decision problem can thus only send out a binary answer of “yes” or “no”.

It might sound a bit too restrictive, but it is not when serving a specific purpose...

**Question:** What does Dr. Shen mean?
Let him clear up... 😊

The idea is that a problem can often be converted to a decision problem by setting up a bound.

For example, once we have a solution to the shortest path problem, we immediately have one for the corresponding decision problem, e.g., ““Is the shortest path in graph $G$ between $u$ and $v$ less than 10?”

$$\text{Find SP } P(u, v) \text{ in } G \Rightarrow |P(u, v)| < 10$$

Thus, if a problem can be easily solved, so is the associated decision problem. On the other hand, if the decision problem is “hard”, then the associated problem must be “hard”, too.

$$A \rightarrow B \equiv \neg B \rightarrow \neg A.$$ 

Thus, when we study how hard a problem is, it suffices to study the decision problems.
Some are, some aren’t.

The above decision problem for the shortest path belongs to $\mathcal{P}$, since, for any graph, we can run, e.g., the Bellman-Ford algorithm, to find out the shortest path between those two vertices, in $O(|V||E|)$, then compare its length with 10 in $\Theta(1)$, and their sum is also a polynomial. 😊

On the other hand, we don’t know yet if the decision problem for the Hamiltonian cycles belongs to $\mathcal{P}$. 😞 Thus, we have no idea about the original problem, either 😞.

But, if somebody claims that a given graph is Hamiltonian and offers a piece of evidence: a sequence of nodes, we can then trivially verify if it is indeed a Hamiltonian cycle for that graph. 😊

**Question:** How to do it?
Given such a cycle, we check first if the given input is a permutation of the vertices of the graph, i.e., if it contains exactly those vertices in the graph; and, if it is, proceed to check whether each of the consecutive edges along the cycle exists in the graph.

The following graph checks out. 😊

It takes $\Theta(|V| + |E|)$ to do the checking....
What does it mean?

Thus, we have a polynomial algorithm to verify whether a graph is Hamiltonian with a given evidence.

The \( \mathcal{NP} \) class is the collection of all the decision problems that can be so verified in polynomial time.

On the other hand, nobody has found a polynomial algorithm to construct a Hamiltonian cycle for a general graph.

**Question:** What is the difference between verification and construction?
What could we get out of it?

If the verification goes through, we are convinced that this graph is Hamiltonian. On the other hand, if it does not check out, we only know that this evidence does not work, but another one might. 😊 Thus, we are still unsure about whether this graph is indeed Hamiltonian. 😞

Intuitively, this sort of verification is much easier than the actual construction.

Otherwise, no body will be afraid of taking a final, where you have to construct, but not just verify, a solution. 😊

**Question:** Is verification more challenging than construction?

**Answer:** No body knows... 😞

There are a lot more out there for this very interesting, but mind boggling, subject, but why don’t we wrap up here... 😐
... with a summary

1. Although many problems can be efficiently solved, some of them are not solvable, while others can be solved, but it takes too long a time.

2. There exist a collection of problems for which all the existing solutions take exponential time (CS), but no one has proved that is the case (MA). On the other hand, it takes only polynomial time to verify whether something is a solution to such a problem. This set of problems is referred to as the $\mathcal{NP}$ class.

3. In contrast, the $\mathcal{P}$ class refers to the collection of problems that a solution can be constructed in polynomial time.

4. Although it is known that $\mathcal{P} \subseteq \mathcal{NP}$, whether $\mathcal{NP} \subseteq \mathcal{P}$, thus $\mathcal{NP} = \mathcal{P}$, is a major open problem, actually, a million dollar question (Check out the link on the course page).