Suppose $L \notin \mathcal{P}$. What does this mean?

Three ways to "get around" $\mathcal{NP}$-completeness:

- small inputs
- special cases
- near-optimal solutions

Let C^* be the value (or *cost*) of an optimal solution. Suppose we have a "near optimal" algorithm that produces a solution with value C. How good (or bad) is C? To quantify this, we would like to show that for some k,

$$\begin{aligned} C^* &\leq kC \quad \text{(for maximization)} \\ C &\leq kC^* \quad \text{(for minimization)} \end{aligned}$$

1. VertexCover. Here is a very simple algorithm. Given $G = (V, E)$, produce a cover C:

$C = \emptyset$
$E' = G.E$
while $E' =\neq \emptyset$
    pick $(u, v) \in E'$
    $C = C \cup \{(u, v)\}$
    remove from E' all edges incident on u or v

Claim: $|C| \leq 2\,|C^*|$.

Proof sketch. *Let A be the set of edges chosen from E' throughout the loop. C^* must include at least one endpoint of every edge in A by definition of vertex cover. Since no two edges in A share a vertex,*

$$|C^*| \geq |A|$$

Moreover,

$$|C| = 2\,|A| \leq 2\,|C^*| \quad \square$$

2. TravellingSalesman. First of all, restrict the problem to Euclidean graphs. Then we have a simple algorithm to compute a tour:

Pick one vertex as a root
Compute a MST (use, say, Prim's algorithm)
Do a preorder walk of the tree
Consider the order of vertices in that walk to be the tour

Let H^* by the real minimum Hamiltonian cycle and H be the computed one.
Claim: $c(H) \leq 2\ c(H^*)$. ($c(H)$ is the cost of H.)

> **Proof sketch.** *Any tour, including an optimal one, can be turned into a spanning tree by removing one edge. Let T be the MST computed by the algorithm.*
>
> $$c(T) \leq c(H^* - e) \leq c(H^*)$$
>
> *Now a full walk of T, call it W, would hit every edge twice, so ...*
>
> $$\frac{1}{2}c(W) \leq \ldots$$
>
> *By the triangle inequality, $c(H) \leq c(W)$, so $c(H) \leq 2\ c(H^*)$.* □

LP talks about three categorizations of approximability (pg 336).

1. *Fully approximable:* $\forall\ \epsilon > 0, \exists$ a polynomial algorithm such that for all x, the algorithm gives a solution C_x such that if C_x^* is the optimal solution,

$$\frac{|\text{value}(C_x^*) - \text{value}(C_x)|}{\text{value}(C_x^*)} \leq \epsilon$$

2. *Partly approximable:* $\exists\ \epsilon > 0$ such that the above is true, but also $\exists\ \delta > 0$ such that $\nexists \epsilon, \delta > \epsilon > 0$ such that the above is true, unless $\mathcal{P} = \mathcal{NP}$.

3. *Inapproximable:* There exists no such ϵ, unless $\mathcal{P} = \mathcal{NP}$.

While the $\mathcal{P}$ vs $\mathcal{NP}$ quandary is a central problem in computer science, we must remember that a resolution of the problem may have limited practical impact. It is conceivable that $\mathcal{P} = \mathcal{NP}$, but the polynomial-time algorithms yielded by a proof of the equality are completely impractical, due to a very large degree of the polynomial or a very large multiplicative constant. . . .

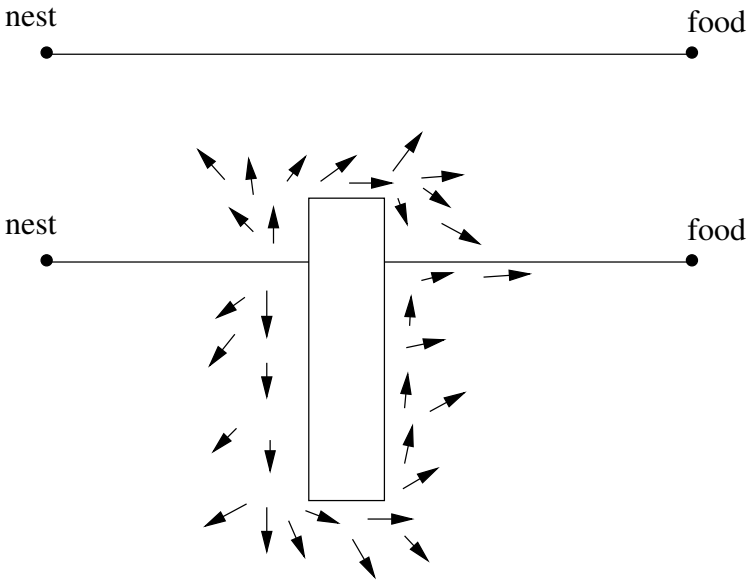
Even more significant, I believe, is the fact that computational complexity theory sheds limited light on behavior of algorithms in the real world. Take, for example, the Boolean Satisfiability Problem (SAT), which is the canonical $\mathcal{NP}$-complete problem. When I was a graduate student, SAT was a "scary" problem, not to be touched with a 10-foot pole.

Moshe Vardi, "On P, NP, and Computational Complexity." CACM Nov 2010

Garey and Johnson's classical textbook showed a long sad line of programmers who have failed to solve NP-complete problems. Guess what? These programmers have been busy! The August 2009 issue of Communications contained an article by Sharad Malik and Lintao Zhang in which they described SAT's journey from theoretical hardness to practical success. Today's SAT solvers, which enjoy wide industrial usage, routinely solve SAT instances with over one million variables. How can a scary $\mathcal{NP}$-complete problem be so easy? What is going on?

Important role of theory is to shed light on practice, and there we have large gaps. We need, I believe, a richer and broader complexity theory, a theory that would explain both the difficulty and the easiness of problems like SAT. More theory, please!

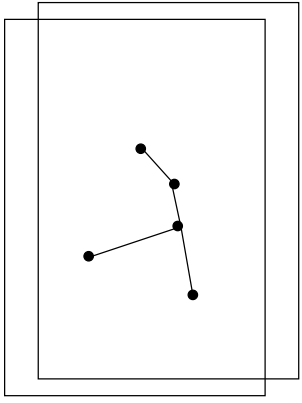
Moshe Vardi, "On P, NP, and Computational Complexity." CACM Nov 2010


nest
food
nest
food

To solve TSP with artificial ants: Put m artificial ants on random cities. Let the ants know the distance between cities and remember the cities they've visited. At each step, each ant chooses a new city,

- One it hasn't yet visited (this tour)
- preferring closer cities
- preferring pheromone-rich edges (explained in a moment)
- but with a bit of randomness

The SteinerTree problem is, given a weighted graph $G = (V, E)$ and a subset of vertices $S \subseteq V$, find a minimum-weight tree that's a subset of G a that contains S.



Let $\mathcal{EXP}$ be the class of languages that can be decided in exponential time by a deterministic TM, and let $\mathcal{NEXP}$ be the class of languages that can be decided in exponential time by a nondeterministic TM (or verified in exponential time by a deterministic TM).
We know

$$\text{If } \mathcal{P} = \mathcal{NP}, \text{then } \mathcal{EXP} = \mathcal{NEXP}.$$

Contrapositively,

$$\text{If } \mathcal{EXP} \neq \mathcal{NEXP}, \text{then } \mathcal{P} \neq \mathcal{NP}.$$

We don't know the converse. It is possible that

$$\mathcal{P} \neq \mathcal{NP}, \text{but } \mathcal{EXP} = \mathcal{NEXP}$$

$\mathcal{EXP}$ is TIME(2^{n^k}). Generally, let $2-\mathcal{EXP} = \text{TIME}(2^{2^{n^k}})$, $3-\mathcal{EXP} = \text{TIME}(2^{2^{2^{n^k}}})$, etc. ($\mathcal{P} = 0-\mathcal{EXP}$).

Result:

$$\text{If } k-\mathcal{EXP} = \mathcal{N}(k-\mathcal{EXP}), \text{ then } k+1-\mathcal{EXP} = \mathcal{N}(k+1-\mathcal{EXP})$$

Define the class of *elementary problems* to be $\{L \mid \exists k \in \mathbb{W} \mid L \in k-\mathcal{EXP}\}$.