The award-winning work of Stephen Cook and Richard Karp

Meena Mahajan

The Institute of Mathematical Sciences, Chennai.

7 November, 2017
Stephen Arthur Cook

Citation

For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, "The Complexity of Theorem Proving Procedures," presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, laid the foundations for the theory of NP-Completeness. The ensuing exploration of the boundaries and nature of NP-complete class of problems has been one of the most active and important research activities in computer science for the last decade.
For his continuing contributions to the theory of algorithms including the development of efficient algorithms for network flow and other combinatorial optimization problems, the identification of polynomial-time computability with the intuitive notion of algorithmic efficiency, and, most notably, contributions to the theory of NP-completeness. Karp introduced the now standard methodology for proving problems to be NP-complete which has led to the identification of many theoretical and practical problems as being computationally difficult.
THE influential papers

- Stephen A. Cook.  
The Complexity of Theorem-Proving Procedures.  

- Richard M. Karp.  
Reducibility Among Combinatorial Problems.  
Other early influential papers


Stephen Cook

- Grew up in New York State, USA.
- 1957-61: Undergraduate studies at University of Michigan.
- Doctoral studies at Harvard University. Influenced by:
  - Early work in computational complexity by Michael Rabin (Turing award winner with Dana Scott, 1976),
  - Alan Cobham’s work characterizing polynomial-time computable functions,
  - Hao Wang’s work on automated theorem proving.
- Since 1970: at Univ of Toronto, Canada.
Richard Karp

- Grew up in Massachusetts State, USA.
- Higher education at Harvard University; PhD in 1959.
- 1959-1968: research staff at IBM Watson Research Center in Yorktown Heights, NY.
- Founding director of the Simons Institute for the Theory of Computing at the University of California, Berkeley.
The computing world before 1970

- Some form of computing devices had been around for quite a while. Famously: mechanical code-breaking devices designed by Turing during World War II.
- Turing machines as an abstract computation model had already been much studied. Recursion theory told us that there are decidable problems, and undecidable problems, and there are “degrees” of undecidability.
The computing world before 1970

- Some form of computing devices had been around for quite a while. Famously: mechanical code-breaking devices designed by Turing during World War II.
- Turing machines as an abstract computation model had already been much studied. Recursion theory told us that there are decidable problems, and undecidable problems, and there are “degrees” of undecidability.
- By the 60s, electronic universal computers were a reality.
- Efficiency of computation was becoming an issue. How to quantify efficiency? What is efficient enough?
Alan Cobham’s work

- Alan Cobham. The intrinsic computational difficulty of functions. 1964 International Congress for Logic, Methodology and Philosophy of Science.

- For computing numerical functions, this paper proposes a criterion for “simple functions”: number of steps bounded by a polynomial in the lengths of the input numbers. Such functions are in the class Cobham called $\mathcal{L}$. 
The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?
The subject of my talk is perhaps most directly indicated by simply asking two questions:
first, is it harder to multiply than to add? and second, why?

This may be a good point to mention that although I have so far been tacitly equating computational difficulty with time and storage requirements, I don’t mean to commit myself to either of these measures.
The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?

This may be a good point to mention that although I have so far been tacitly equating computational difficulty with time and storage requirements, I don’t mean to commit myself to either of these measures.

The problem is reminiscent of, and obviously closely related to, that of the formalization of the notion of effectiveness. But the emphasis is different in that the physical aspects of the computation process are here of predominant concern.
The perfect matching problem

- There’s a bunch of people. Some get along with each other, some don’t.
- The task: Pair them up into teams of 2.
- The conditions:
  - Pair up everyone with someone.
  - Pair together only people who get along well.

Devise an algorithm to find such a pairing, or to report (and prove) that no such pairing exists.
First efficient algorithm for the perfect matching problem:

*Jack Edmonds.*

*Paths, Trees and Flowers.*


What’s relevant here is not the algorithm, but Edmonds’ remarks in Section 2.
Section 6 presents a certain invariance property of the dual to maximum matching.

In paper (4), the algorithm is extended from maximizing the cardinality of a matching to maximizing for matchings the sum of weights attached to the edges. At another time, the algorithm will be extended from a capacity of one edge at each vertex to a capacity of $d_i$ edges at vertex $v_i$.

This paper is based on investigations begun with G. B. Dantzig while at the RAND Combinatorial Symposium during the summer of 1961. I am indebted to many people, at the Symposium and at the National Bureau of Standards, who have taken an interest in the matching problem. There has been much animated discussion on possible versions of an algorithm.

2. Digression. An explanation is due on the use of the words “efficient algorithm.” First, what I present is a conceptual description of an algorithm and not a particular formalized algorithm or “code.”

For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, “efficient” means “adequate in operation or performance.” This is roughly the meaning I want—in the sense that it is conceivable for maximum matching to have no efficient algorithm. Perhaps a better word is “good.”

I am claiming, as a mathematical result, the existence of a good algorithm for finding a maximum cardinality matching in a graph.

There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph.

The mathematical significance of this paper rests largely on the assumption that the two preceding sentences have mathematical meaning. I am not prepared to set up the machinery necessary to give them formal meaning, nor
Quotes from Edmonds’ “digression”

- There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph.
Quotes from Edmonds’ “digression”

- There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph.
- It may be that since one is customarily concerned with existence, convergence, finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.
Quotes from Edmonds’ “digression”

- There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph.
- It may be that since one is customarily concerned with existence, convergence, finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.
- For practical purposes the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite.
Quantifying used resources

  On the Computational Complexity of Algorithms.

- This work defined “complexity classes” like DTIME($T(n)$).
  It proved hierarchy theorems: given sufficiently more time, you can provably do more!
Quantifying used resources

- This work defined “complexity classes” like DTIME($T(n)$). It proved hierarchy theorems: given sufficiently more time, you can provably do more!
- It fetched them the Turing award in 1993, some years after Cook and Karp!
- From the 1993 citation:
  ... in recognition of their seminal paper which established the foundations for the field of computational complexity theory.
Cook’s contribution, against this backdrop

“... give evidence that (tautologies) is a difficult set to recognise, since many apparently difficult problems can be reduced to determining tautologyhood.”
Cook’s contribution, against this backdrop

“... give evidence that (tautologies) is a difficult set to recognise, since many apparently difficult problems can be reduced to determining tautologyhood.”

To provide the evidence, a definition of resource-bounded reducibility via query machines (machines with oracles).
Cook’s contribution, against this backdrop

“... give evidence that (tautologies) is a difficult set to recognise, since many apparently difficult problems can be reduced to determining tautologyhood.”

To provide the evidence, a definition of resource-bounded reducibility via query machines (machines with oracles).

The theorem - as stated by Cook:

*If a set $S$ of strings is accepted by some nondeterministic Turing machine within polynomial time, then $S$ is P-reducible to DNF tautologies.*

“... If $\mathcal{L}^+$ is the analog of the class of r.e. sets, then determining tautologyhood is the analog of the halting problem.”
What are these difficult problems?

Boolean formulas:
variables $a, b, c, \ldots$ taking values \texttt{True}, \texttt{False},
connected by \underline{AND} $\wedge$, \underline{OR} $\lor$, and \underline{NOT} $\neg$.

\[ \neg((a \lor b) \wedge \neg(\neg a \lor (b \land c))) \]
What are these difficult problems?

Boolean formulas:
variables $a$, $b$, $c$, ... taking values $\text{True}$, $\text{False}$, connected by $\text{AND}$ $\land$, $\text{OR}$ $\lor$, and $\text{NOT}$ $\neg$.

$$\neg((a \lor b) \land \neg(a \lor (b \land c)))$$

- **TAUT**: Tautologies. Formulas that are always true.
What are these difficult problems?

Boolean formulas:
variables $a$, $b$, $c$, … taking values True, False, connected by AND $\land$, OR $\lor$, and NOT $\neg$.

$$\neg((a \lor b) \land \neg(\neg a \lor (b \land c)))$$

- **TAUT**: Tautologies. Formulas that are always true.
- **DNF TAUT**: Tautologies in Disjunctive Normal Form

$$(a \land \neg b \land c) \lor (\neg a \land d \land \neg f) \lor \ldots \lor (\neg b \land f)$$
What are these difficult problems?

Boolean formulas:
variables $a, b, c, \ldots$ taking values $\text{True}$, $\text{False}$, connected by $\text{AND}$ $\wedge$, $\text{OR}$ $\vee$, and $\text{NOT}$ $\neg$.

$$\neg((a \lor b) \land \neg(a \lor (b \land c)))$$

- **TAUT**: Tautologies. Formulas that are always true.
- **DNF TAUT**: Tautologies in Disjunctive Normal Form
  $$(a \land \neg b \land c) \lor (\neg a \land d \land \neg f) \lor \ldots \lor (\neg b \land f)$$

- **D$_3$**: DNF tautologies with term size at most 3.
What are these difficult problems?

Boolean formulas:
variables \( a, b, c, \ldots \) taking values \( \text{True}, \text{False}, \)
connected by \( \text{AND} \ \& \), \( \text{OR} \ \vee \), and \( \text{NOT} \ \neg \).

\[
\neg ((a \lor b) \land \neg(a \lor (b \land c)))
\]

- **TAUT**: Tautologies. Formulas that are always true.
- **DNF TAUT**: Tautologies in Disjunctive Normal Form

\[
(a \land \neg b \land c) \lor (\neg a \land d \land \neg f) \lor \ldots \lor (\neg b \land f)
\]

- **D_3**: DNF tautologies with term size at most 3.
- **SAT**: Formulas that can evaluate to True.
  \( F \) is a Tautology if and only if \( \neg F \) is not in SAT.
What are these difficult problems?

Boolean formulas:
variables $a, b, c, \ldots$ taking values True, False, connected by AND $\land$, OR $\lor$, and NOT $\neg$.

$\neg((a \lor b) \land \neg(a \lor (b \land c)))$

- **TAUT**: Tautologies. Formulas that are always true.
- **DNF TAUT**: Tautologies in Disjunctive Normal Form

$$(a \land \neg b \land c) \lor (\neg a \land d \land \neg f) \lor \ldots \lor (\neg b \land f)$$

- **D$_3$**: DNF tautologies with term size at most 3.
- **SAT**: Formulas that can evaluate to True.
  $F$ is a Tautology if and only if $\neg F$ is not in SAT.
- **Subgraph Pairs**: Pairs of graphs $(G, H)$ where $G$ is isomorphic to some subgraph of $H$. 
The notation Cook used

Since Cobham denoted the class of \( P \)-computable functions by \( \mathcal{L} \),

- \( \mathcal{L}^* \): the class of sets recognisable in polynomial time. What we now call \( P \).
- \( \mathcal{L}^+ \): the class of sets accepted in polynomial time by some nondeterministic Turing machine. What we now call \( \text{NP} \): \textbf{Not Polynomial Time} \hfill \textbf{Nondeterministic Polynomial Time}

\( P \): solution easy to find; \( \text{NP} \): solution easy to check.

- The notion of resource-bounded reducibilities, via query machines using oracles. Modern notation: \( A \leq^p_T B \).
Cook’s evidence of hardness

- Cook’s Theorem 1: If a set $S$ of strings is accepted by some nondeterministic Turing machine within polynomial time, then $S$ is $P$-reducible to DNF tautologies.
Cook’s evidence of hardness

- **Cook’s Theorem 1:** If a set $S$ of strings is accepted by some nondeterministic Turing machine within polynomial time, then $S$ is $P$-reducible to DNF tautologies.

- **Cook’s Theorem 2:** The following sets are $P$-reducible to each other in pairs: TAUT, DNF TAUT, $D_3$, Subgraph Pairs.
Cook’s evidence of hardness

- Cook’s Theorem 1: If a set $S$ of strings is accepted by some nondeterministic Turing machine within polynomial time, then $S$ is $P$-reducible to DNF tautologies.
- Cook’s Theorem 2: The following sets are $P$-reducible to each other in pairs: TAUT, DNF TAUT, D$_3$, Subgraph Pairs.
- Cook’s Remark: “We have not been able to add either Primes or Graph Isomorphism to the above list”
Cook’s results, rephrased by Karp

- What Cook actually proved: If $S$ has a non-deterministic polynomial time algorithm, then $S$ can be reduced in deterministic polynomial time to solving an instance of SAT and reporting that answer as is.
- In fact, $P = NP$ if and only if SAT is in P.
What Cook actually proved: If $S$ has a non-deterministic polynomial time algorithm, then $S$ can be reduced in deterministic polynomial time to solving an instance of SAT and reporting that answer as is.

In fact, $P = NP$ if and only if SAT is in $P$.

Karp defined a more sensitive notion of reducibility reflecting this "make a single query, report answer as is". Modern notation: $A \leq_{p_m} B$.

Cook’s theorem, as rephrased by Karp: SAT is NP-complete (with respect to $\leq_{p_m}$ reductions).
NP-completeness of SAT: The significance

- A problem easily seen to be NP-complete:

\[ \{(M, x, 1^t) \mid M \text{ is a nondeterministic algorithm, } M \text{ accepts } x \text{ along some path in } \leq t \text{ steps} \} \]
NP-completeness of SAT: The significance

- A problem easily seen to be NP-complete:

\[ \left\{ (M, x, 1^t) \mid M \text{ is a nondeterministic algorithm, } M \text{ accepts } x \text{ along some path in } \leq t \text{ steps} \right\} \]

- SAT defined independent of TMs. Yet, it captures the inherent hardness of every problem solvable by a NPTM. Remarkable!

Cook’s genius: showing every abstract **NPTM computation** efficiently reduces to a SAT instance.
NP-completeness of SAT: The significance

A problem easily seen to be NP-complete:

\[ \{ (M, x, 1^t) \mid M \text{ is a nondeterministic algorithm, } M \text{ accepts } x \text{ along some path in } \leq t \text{ steps} \} \]

SAT defined independent of TMs. Yet, it captures the inherent hardness of every problem solvable by a NPTM. Remarkable!
Cook’s genius: showing every abstract NPTM computation efficiently reduces to a SAT instance.

How / Why did Cook zero in on SAT and TAUT?
Automatic theorem-proving ...
Tautologies are Theorems. How can we mechanically prove such theorems?
... the interplay of logic and computational complexity ...
NP-completeness: The significance ...

- All NP-complete problems are equally hard (or equally easy). Solve one, solve them all!
NP-completeness: The significance ...

- All NP-complete problems are equally hard (or equally easy). Solve one, solve them all!
- Turns out, there are many such problems. SAT provides the **starting point** to show that they are as hard.
- Subsequently, showing some other problem \( \Pi \) is NP-hard is much more easy: suffice to show SAT reduces to \( \Pi \).
NP-completeness: The significance ...

- All NP-complete problems are equally hard (or equally easy). Solve one, solve them all!
- Turns out, there are many such problems. SAT provides the starting point to show that they are as hard.
- Subsequently, showing some other problem \( \Pi \) is NP-hard is much more easy: suffice to show SAT reduces to \( \Pi \).
- That’s what Karp did, in a big way!
Karp’s web of reductions

FIGURE 1 - Complete Problems
Knapsack

**MY HOBBY:**
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

**CHOTCHKIES RESTAURANT**

- **Appetizers**
  - Mixed Fruit: 2.15
  - French Fries: 2.75
  - Side Salad: 3.35
  - Hot Wings: 3.55
  - Mozzarella Sticks: 4.20
  - Sampler Plate: 5.80

- **Sandwiches**
  - Barbecue: 6.55

---

WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.

EXACTLY? UHH...

HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.

LISTEN, I HAVE SIX OTHER TABLES TO GET TO—

—AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

---

General solutions get you a 50% tip.

xkcd.com
What's the complexity class of the best linear programming cutting-plane techniques? I couldn't find it anywhere. Man, the Garfield guy doesn't have these problems ...

xkcd.com
With so many natural problems from multiple diverse contexts shown to be equivalent and NP-complete, the P vs NP problem shot to the spotlight.
And computational complexity emerged convincingly from the shadow of recursion theory.
P vs NP

With so many natural problems from multiple diverse contexts shown to be equivalent and NP-complete, the P vs NP problem shot to the spotlight.

And computational complexity emerged convincingly from the shadow of recursion theory.

NP-completeness was immortalised in the cartoon in the Garey-and-Johnson book (1979): I can’t solve this problem, but neither can all these famous people.
With so many natural problems from multiple diverse contexts shown to be equivalent and NP-complete, the P vs NP problem shot to the spotlight.

And computational complexity emerged convincingly from the shadow of recursion theory.

NP-completeness was immortalised in the cartoon in the Garey-and-Johnson book (1979): I can’t solve this problem, but neither can all these famous people.

45 years down the line, “Is P=NP?” remains the canonical challenge and open question of the field.
A Clay Math Institute’s Millenium Problem

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker’s list also appears on the list from the Dean’s office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Image credit: on the left, Stephen Cook by Jiří Janiček (cropped). CC BY-SA 3.0

2 of 2 07/11/17, 3:04 PM
Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer.

Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.
Leonid Levin

Early 1960s: Boarding school for high-school math/physics prodigies in Kiev.

1970: Masters degree from Moscow University.
PhD denied officially for political reasons.

1978: Emigrated to US.

1979: PhD from MIT.

1980 onwards: Boston University.
The PhD denied

As a graduate student: Levin produced two major lines of work:

1. universal search algorithm
2. the notion of a universal search problem (equivalent to NP-completeness), and a list of six universal search problems, including SAT. The others are versions of Set Cover, minimal DNFs, subgraph homomorphism, subgraph isomorphism, bounded tiling
The PhD denied

As a graduate student: Levin produced two major lines of work:

1. universal search algorithm
2. the notion of a universal search problem (equivalent to NP-completeness), and a list of six universal search problems, including SAT. The others are versions of Set Cover, minimal DNFs, subgraph homomorphism, subgraph isomorphism, bounded tiling

Told to combine the two, otherwise not publishable!
The PhD denied

As a graduate student: Levin produced two major lines of work:

1. universal search algorithm
2. the notion of a universal search problem (equivalent to NP-completeness), and a list of six universal search problems, including SAT. The others are versions of Set Cover, minimal DNFs, subgraph homomorphism, subgraph isomorphism, bounded tiling

Told to combine the two, otherwise not publishable!

One two-page paper:

Leonid Levin.

Not known in the West until much later.
Across the iron curtain

- Different people, different cultures.
- Same time period.
- Same scientific notions developed.
- Staggering coincidence?
Across the iron curtain

- Different people, different cultures.
- Same time period.
- Same scientific notions developed.
- Staggering coincidence?
  An idea whose time had come: **NP-completeness.**
Across the iron curtain

- Different people, different cultures.
- Same time period.
- Same scientific notions developed.
- Staggering coincidence?
  An idea whose time had come: **NP-completeness**.
- Today,

  “SAT is NP-complete”: the **Cook-Levin theorem**

  Polynomial-time
  many-one reductions:

  Polynomial-time
  many-one reductions
  that retrieve witnesses:

- Karp reductions
- Levin reductions
Across the iron curtain ...

- Karp’s work too had parallels in the then USSR.

- Karp’s Turing award citation refers to this work:
  
  *Jack Edmonds; Richard M. Karp.*
  
  *Theoretical improvements in algorithmic efficiency for network flow problems.*
  

This paper describes a strongly polynomial-time algorithm to find the maximum flow that can be sent through a network.
Karp’s work too had parallels in the then USSR.

Karp’s Turing award citation refers to this work:

*Jack Edmonds; Richard M. Karp.
*Theoretical improvements in algorithmic efficiency for network flow problems.

This paper describes a strongly polynomial-time algorithm to find the maximum flow that can be sent through a network.

At the same time, a different algorithm for the same problem:

*Algorithm for solution of a problem of maximum flow in a network with power estimation.
Other work by Stephen Cook

- foundations of propositional proof complexity, along with Robert Reckhow
- logical system to characterise feasible reasoning
- parallel computation - an early result was that DCFLs can be recognised in polynomial time with just $O(\log^2 n)$ space.

(the name NC for the class of efficiently parallelizable problems was given by Cook; Nick’s classes, named after Nicolas Pippenger. Later, the class of problems simultaneously efficient in time and space was called SC - Steve’s classes.)

...
Other work by Richard Karp

- models of parallel computation
- heuristics for hard problems
- randomization in the design and analysis of algorithms
- computational biology, using techniques from combinatorial optimization and stochastic approaches from machine learning
...
Thank you