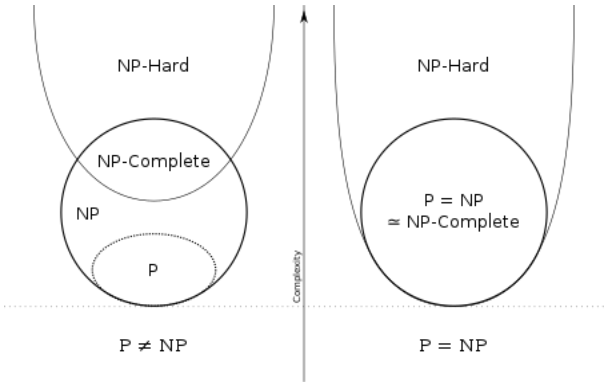
NP-hardness

From Wikipedia, the free encyclopedia

In computational complexity theory, **NP-hardness** (non-deterministic polynomial-time hardness) is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP". A simple example of an NP-hard problem is the subset sum problem.

A more precise specification is: a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H; that is, assuming a solution for H takes 1 unit time, H's solution can be used to solve L in polynomial time.[1][2] As a consequence, finding a polynomial time algorithm to solve any NP-hard problem would give polynomial time algorithms for all the problems in NP. As it is suspected that P≠NP, it is unlikely that such an algorithm exists.[3]

It is suspected that there are no polynomial-time algorithms for NP-hard problems, but that has not been proven.[4] Moreover, the class P, in which all problems can be solved in polynomial time, is contained in the NP class.[5]



Euler diagram for P, NP, NP-complete, and NP-hard set of problems. The left side is valid under the assumption that P≠NP, while the right side is valid under the assumption that P=NP (except that the empty language and its complement are never NP-complete)

Contents

Definition

A decision problem H is NP-hard when for every problem L in NP, there is a polynomial-time many-one reduction from L to H.[1]:80 An equivalent definition is to require that every problem L in NP can be solved in polynomial time by an oracle machine with an oracle for H.[6] Informally, an algorithm can be thought of that calls such an oracle machine as a subroutine for solving H and solves L in polynomial time if the subroutine call takes only one step to compute.

Another definition is to require that there be a polynomial-time reduction from an NP-complete problem G to H.[1]:91 As any problem L in NP reduces in polynomial time to G, L reduces in turn to H in polynomial time so this new definition implies the previous one. Awkwardly, it does not restrict the class NP-hard to decision problems, and it also includes search problems or optimization problems.

Consequences

If P ≠ NP, then NP-hard problems could not be solved in polynomial time.

Some NP-hard optimization problems can be polynomial-time approximated up to some constant approximation ratio (in particular, those in APX) or even up to any approximation ratio (those in PTAS or FPTAS).

Examples

An example of an NP-hard problem is the decision subset sum problem: given a set of integers, does any non-empty subset of them add up to zero? That is a decision problem and happens to be NP-complete. Another example of an NP-hard problem is the optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph. This is commonly known as the travelling salesman problem.[7]

There are decision problems that are *NP-hard* but not *NP-complete* such as the halting problem. That is the problem which asks "given a program and its input, will it run forever?" That is a *yes*/*no* question and so is a decision problem. It is easy to prove that the halting problem is NP-hard but not NP-complete. For example, the Boolean satisfiability problem can be reduced to the halting problem by transforming it to the description of a Turing machine that tries all truth value assignments and when it finds one that satisfies the formula it halts and otherwise it goes into an infinite loop. It is also easy to see that the halting problem is not in *NP* since all problems in NP are decidable in a finite number of operations, but the halting problem, in general, is undecidable. There are also NP-hard problems that are neither *NP-complete* nor *Undecidable*. For instance, the language of true quantified Boolean formulas is decidable in polynomial space, but not in non-deterministic polynomial time (unless NP = PSPACE).[8]

NP-naming convention

NP-hard problems do not have to be elements of the complexity class NP. As NP plays a central role in computational complexity, it is used as the basis of several classes:

NP
: Class of computational decision problems for which any given *yes*-solution can be verified as a solution in polynomial time by a deterministic Turing machine (or *solvable* by a *non-deterministic* Turing machine in polynomial time).

NP-hard
: Class of problems which are at least as hard as the hardest problems in NP. Problems that are NP-hard do not have to be elements of NP; indeed, they may not even be decidable.

NP-complete
: Class of decision problems which contains the hardest problems in NP. Each NP-complete problem has to be in NP.

NP-easy
: At most as hard as NP, but not necessarily in NP.

NP-equivalent
: Decision problems that are both NP-hard and NP-easy, but not necessarily in NP.

NP-intermediate
: If P and NP are different, then there exist decision problems in the region of NP that fall between P and the NP-complete problems. (If P and NP are the same class, then NP-intermediate problems do not exist because in this case every NP-complete problem would fall in P, and by definition, every problem in NP can be reduced to an NP-complete problem.)

Application areas

NP-hard problems are often tackled with rules-based languages in areas including:

- Approximate computing
- Configuration
- Cryptography
- Data mining
- Decision support
- Phylogenetics
- Planning
- Process monitoring and control
- Rosters or schedules
- Routing/vehicle routing

- Scheduling

References

1. Leeuwen, Jan van, ed. (1998). *Handbook of Theoretical Computer Science*. Vol. A, Algorithms and complexity. Amsterdam: Elsevier. ISBN 0262720140. OCLC 247934368 (https://www.worldcat.org/oclc/247934368).
2. Knuth, Donald (1974). "Postscript about NP-hard problems". *ACM SIGACT News*. **6** (2): 15–16. doi:10.1145/1008304.1008305 (https://doi.org/10.1145%2F1008304.1008305). S2CID 46480926 (https://api.semanticscholar.org/CorpusID:46480926).
3. Daniel Pierre Bovet; Pierluigi Crescenzi (1994). *Introduction to the Theory of Complexity*. Prentice Hall. p. 69. ISBN 0-13-915380-2.
4. "Shtetl-Optimized » Blog Archive » The Scientific Case for P≠NP" (http://www.scottaaronson.com/blog/?p=1720). *www.scottaaronson.com*. Retrieved 2016-09-25.
5. "PHYS771 Lecture 6: P, NP, and Friends" (http://www.scottaaronson.com/democritus/lec6.html). *www.scottaaronson.com*. Retrieved 2016-09-25.
6. V. J. Rayward-Smith (1986). *A First Course in Computability*. Blackwell. p. 159. ISBN 0-632-01307-9.
7. Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G.; Shmoys, D. B. (1985), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (https://archive.org/details/travelingsalesma00lawl), John Wiley & Sons, ISBN 0-471-90413-9.
8. More precisely, this language is PSPACE-complete; see, for example, Wegener, Ingo (2005), *Complexity Theory: Exploring the Limits of Efficient Algorithms* (https://books.google.com/books?id=1fo7_KoFUPsC&pg=PA189), Springer, p. 189, ISBN 9783540210450.

- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5.