P = NP – what does it mean?

essay for the course in

Theory of Science and Research Ethics.

Niklas Eén
Chalmers University of Technology, Sweden
een@cs.chalmers.se

May 15, 2001

Abstract

In today's technological society computers have become indispensable. We rely on them to routinely compute solutions for a vast amount of problems. Yet at the core of the theory of computation there is an unresolved question who's resolution may have a deep impact on what problems we can successfully apply computers to. The question can be phrased "Does P equal NP?". "P" and "NP" are technical definitions, but generalizing just a bit we can summarize "P" as a class of problems we can currently solve, and "NP" as a class of problems we very much would like to be able to solve. Literally thousands of industrial problems lie in the "NP" class, including problems such as scheduling, route planning, and automatic theorem proving for propositional logic. Proving "P=NP" might allow us to solve all these problems efficiently. It would be a gold rush for the computer scientist! However, there are also problems that we might *not* want to be able to solve, but which will suddenly be rendered tractable, such as breaking todays strongest encryptions. This will make todays secure bank transactions insecure overnight.

1 Introduction

The question for this exposition, "P = NP?", arose during the middle part of the twentieth century and is now considered to be one of the most important problems in contemporary mathematics and theoretical computer science. We will try to explain to the layman what the question means, why it is so interesting, and what possible consequences its resolution might have.

2 History

What does it mean to compute something? For a computer scientist, to *compute* is not restricted to manipulating numbers, but also encompass operations such as: from an unsorted list of integers, compute the sorted list of the same integers; or from a list of events that need to be scheduled, compute the optimal scheduling according to some criterion. A *computation* of these problems should be understood as a mechanical process following deterministically some unambiguous rules that we have specified—what we today would call an *algorithm*.

Problems of computation may also include more sophisticated tasks such as: from a given mathematical theorem, compute a proof for it, or state that no such proof exists. This is an example of a problem, known as the *Entscheidungsproblem*, that we today know is non-computable—that is to say there can be no general procedure to solve it for every instance. This was not known in the 1930s. One of history's greatest mathematician David Hilbert (1862-1943) strongly advocated the search of what he called a "finite method" for the Entscheidungsproblem. However, his dreams were soon to be shattered as a deeper understanding of the concept of a computation ensued.

In the first half of the twentieth century, forerunners of the programmable computer were beginning to appear. Mathematicians contemplated the capabilities and limitations of such devices. This embraced, as illustrated above, not only practical computations but also the foundations of mathematics. In 1936, Alan Turing (1912-1954) was the first person to formalize what earlier mathematicians had referred to as "mechanical computation" or "method". In his work he defines an extremely simple imaginary machine which still is strong enough to carry out "arbitrary" computations. The machine, which later became known as the Turing machine, basically consist of an infinite tape which it can read and write on and a list of rules defining where and what to write given what it currently says on the tape. Turing suggested that this machine indeed captured exactly what we mean by computable—or in his words "the formalism is sufficiently general to encompass anything that a human being could do when carrying out a definite method". Today we know that any reasonable model of computation, including (of course) actual computers, coincides with the Turing machine on computational power[1]. In this respect, a Turing machine can be thought of as just a standard computer, save that it should have an unlimited amount of memory. Likewise, the rules of the Turing machine is just the program software run by the computer.

Having established a formal definition of what it means for something to be computable, Turing and his fellow logicians plunged into the question of defining the boarder between computable and non-computable problems. The most notable result is the so called "halting problem", which says that it is impossible for a program, in general, to determine whether another (arbitrary) program will stop upon execution, or if it will run forever.

These theoretical results gave a framework for dealing with "strong" non-

[1] This is truly a remarkable fact. Its philosophical relevance cannot be overemphasized.

computability. No need to spend time looking for an efficient algorithm if the problem at hand is not even computable! However, already in the 50s in the early development of computer machines, the importance of *feasible* algorithms was exposed. For many interesting problem it seemed that the best solution could only be found by exhaustively testing all possible combinations, then selecting the best one found. This was highly unsatisfactory as already for toy examples, the algorithms would need weeks or even centuries to run. The problem is that the number of possible combinations grows so tremendously rapid, that no matter how fast computers we manage to build, we will still be unable to solve any typical real-life sized problems. Thus arose the need to categorize problems not only into computable and non-computable, but also into degrees of *efficiency*, that is how efficiently computable the particular problem is given a certain algorithm. This field of research is today known as *complexity theory*.

Regrettably enough, when we start to distinguish problems and algorithms with respect to the time it takes to solve or run them, we get into trouble. Suddenly it matters what computational model we use. The Turing machine is no longer equivalent to the computer machine, even with the hypothetically infinite amount of memory. In one model of computation we can solve a particular problem faster than in the other. Essentially we can make statements about efficiency on three levels of abstraction:

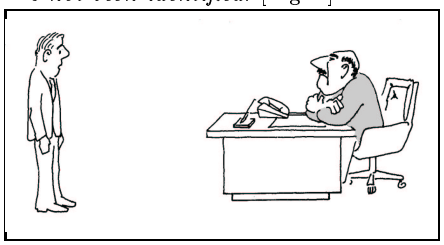
(i) On a specific machine we run a certain algorithm to solve a particular problem.

(ii) We run a certain algorithm to solve a particular problem.

(iii) We solve a particular problem.

Ideally we would like to make statements about feasibility on the third level, just as for computability, abstracting away the particular machine and algorithm we deploy on our problem. In 1971 the foundations for doing so was laid by Stephen Cook in his brief but elegant paper "The Complexity of Theorem Proving Procedures". Cook focused on algorithms solvable in *polynomial* time (more about this in the next section), and by this recovering the benefit of machine model independence. If an algorithm can be run in polynomial time on one machine, then so could it on any other reasonable machine. Further, he introduced the "NP-complete" class of problems (although he did not use the term himself), and showed that (a) many problems of interest belong to this class, and (b) if any problem in this class could be solved efficiently, then so could all of them. The latter is particular important as today literally thousands of real-life problems have been shown to belong to the NP-complete class, and yet no feasible algorithm has been found to solve any (and thereby all) of the problems. This establish a kind of weak infeasibility—even if we cannot yet prove a particular problem to be infeasible, we know that some of the smartest brains of this planet has not been able to find a feasible algorithm. The situation is illustrated by Figure 1-3.

Past situation. *NP-complete problems has not been identified.* [Fig. 1]



"I can't find an efficient algorithm, I guess I'm just too dumb."

Possible future. *NP-complete problems is proven to be hard.* [Fig. 2]

"I can't find an efficient algorithm, because no such algorithm is possible!"

3 What are P and NP?

When a complexity theoretician uses the word "problem" he means something more restricted than we normally do in everyday use. A *problem* to him is a description of *parameters* (the things we will vary) and a definition of what constitutes a *solution*. An *instance* of a problem is an assignment to the parameters.

Example. The *traveling salesman problem* (TSP). A salesman has to visit a number of cities in his line of work. He wishes to spend as little time as possible on the road, and as much time as possible selling his goods. Therefore he looks for the shortest route to visit all cities (and return back home).

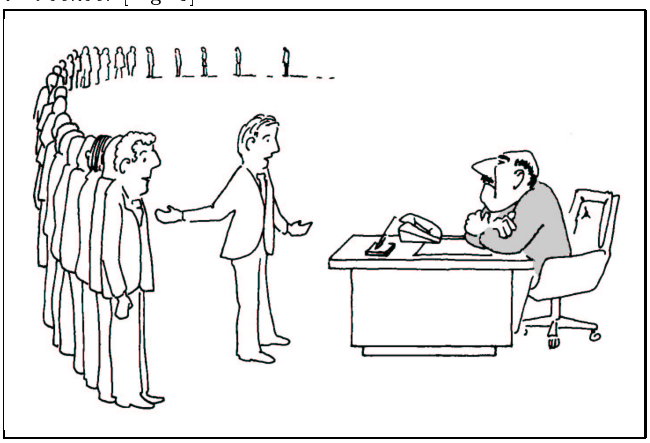
The parameters of this problem is the number of cities plus the pairwise distances between each two cities. A solution is a list, such that each city is listed exactly once and no other such list of cities will yield a shorter travel.

Given a problem we typically would like to predict how long our algorithm would take to produce a solution. This is critical for applications. The standard approach to this is to select some *size* measurement from the parameters and express an *upper bound* on the time consumption in this size. For the TSP problem we might choose the number of cities as the "size" of the problem. A statement about the time consumption might be something like "when the size is increased by one city, the solution takes twice as long to compute", or mathematically: $T(n) = 2^n$.

The class of polynomially solvable problems that Cook focused on constitutes problems for which there exists an algorithm whose time consumption is bounded by some polynomial—that is for the time T as a function of the size n it holds that $T(n) \approx n^k$ for some constant k. The $\approx$ can be given precise mathematical meaning using Ordo notation[2]. Examples of polynomial algorithms are sorting, matrix multiplication, or searching for a string in a text.

[2] A function T is Ordo f, denoted $O(f)$, if for some $r > 0$ and for all but finitely many n, $T(n) < rf(n)$. An algorithm is (time) polynomial if its time function $T(n)$ belong to $O(n^k)$ for some k.

Current status. *NP-complete problems hard in an empirical sense.* [Fig. 3]



"I can't find an efficient algorithm, but neither can all these famous people."

Stemming from automata theory, the concept of *non-determinism* comes very naturally into Turing machines and machine models in general. The idea is that the list of rules of what to do may contain ambiguities—sometimes more than one rule apply and the machine are allowed to do any of the actions dictated by those rules. For a computer program this would translate into using *random* choices in the algorithm. It is not immediately clear what it means for such a machine to solve a problem, since the outcome is not deterministic. A meaningful choice is to say that *if* the machine solves the problem for *some* possible run, *then* it solves the problem. In other words, if we are Mr Lucky Guy and always make the right choice to take us to the solution, then our algorithm is said to non-deterministically solve the problem.

The intuition says that such a machine, with the ability always to guess "right", should be much stronger than its deterministic counterpart. Take for instance a version of the TSP problem often considered: Given a number of cities and their pairwise distances, does there exist a path through all cities *shorter* than the distance D? With a non-deterministic machine we can just generate a sequence of cities at random, and check if this path is shorter than D. If such a path exist, then for *some* run of the algorithm the path will be produced. Generating a random sequence of cities and computing their total travelling distance takes time directly proportional to the number of cities, that is polynomial time with $k = 1$ (also called *linear* time). This is very efficient. On the contrary, it is not at all clear how to make an efficient algorithm for a deterministic machine. Indeed, no known algorithm exists. Despite this, no one has been able to prove that you actually "gain" computational power from the non-determinism. This is precisely the "P=NP?" question. Let us state this question a bit more formal:

Definition. P is the class of problems that can be solved by a polynomially

bounded deterministic algorithm.

Definition. NP is the class of problems that could be solved by a polynomially bounded *non*-deterministic algorithm.

The question is "Are these two classes the same?". This is still an open problem. The above variant of the TSP problem is an example of a problem that belongs to the NP class (by the algorithm presented) but for which no polynomial algorithm has yet been found. In fact, this particular problem has a property that it shares with thousands of NP problems, namely that *if* you can devise an algorithm to solve it in polynomial time by a deterministic machine, *then any* NP problem can be solved in polynomial time by a deterministic machine! This property is called *NP-completeness*. All we have to do in order to show P=NP is to find a polynomial algorithm for *one* of the vast amount of problems shown to be NP-complete. From this algorithm we can then construct polynomial algorithms for all the other problems.

During the thirty years that the "P=NP?" question has been around, many have of course attempted to produce such a polynomial algorithm for some NP-complete problem. Most often the author made some critical error in his analysis of the running time of his algorithm. Sometimes the algorithm actually was correct, but the NP-complete problem turned out not to be NP-complete after all—someone else made a mistake in *their* proof. There has also been several attempts to prove that P≠NP—that there can be no polynomial algorithm for any NP-complete problem, so far without success. So the question of whether such an algorithm can even exist remain open. It is fair to say that it has now become the official Holy Grail of computer science.

4 NP-complete problems

The first problem shown to be NP-complete was the so call *SAT problem* (Cook 1971). SAT is the problem of deciding whether a *propositional* formula is *satisfiable* or not[3]. Let us give an example of a propositional formula:

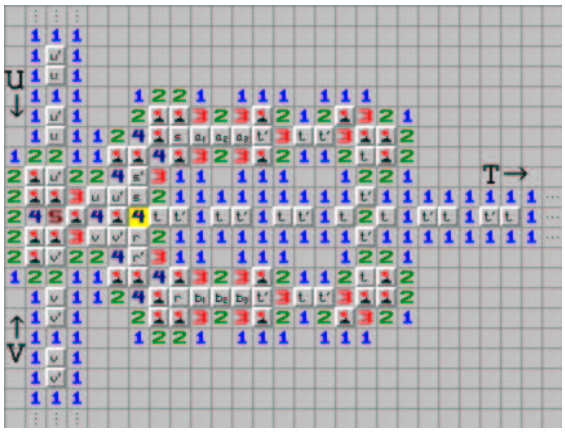
$$\neg\big(\ (A \wedge B \rightarrow C)\ \leftrightarrow\ (A \rightarrow B \rightarrow C)\ \big)$$

This formula essentially ask if the to following sentences has identical meaning:

- If I know A and B, then I know C.
- If I know A, then if I also know B I know C.

A program that solves the SAT problem can answer this question (positively). Since SAT is a NP complete problem, no one has been able to build a SAT solver than can cope with an arbitrary instance of the problem. Still, SAT

[3]A programmers formulation: Given a boolean expression, is there an assignment to the variables that makes the expression evaluate to true.



An AND gate translated into Minesweeper. A good Minesweeper player will never make a guess if there is a square that is certainly free and a guess is not required. Similarly he or she will mark up every mine that can be identified with certainty. [Fig. 4]

solvers for particular classes of formulas are extremely useful tools, as many problem translate naturally into a logical statement. It is an often quoted fact, that in 1994 Intel lost 475 million dollars due to a logical bug present in their Pentium processor. Millions of chips had to be retrieved and replaced. Proving the chips to conform to the specification before delivery would prevent this scenario, or at least diminish the likelihood of it. Today, SAT solvers and other techniques are routinely applied by all manufacturers of integrated circuits to prove logical correctness of their products. However, the advanced microprocessor designs of today are so complex that even for state-of-the-art SAT solvers it is not possible to prove 100% conformance. Bugs may still slip through, possibly costing billions of dollars in lost revenue. This demonstrates the practical importance of better algorithms for NP-complete problems.

In the previous section, we gave a one-line algorithm that showed TSP to be within NP. But how can we show it to be NP-*complete*? One way is to find an efficient (polynomial time) translation from SAT into TSP. If we can use TSP to solve a known NP-complete problem, then TSP must also be NP-complete, so the reasoning goes. This schema is commonly used for proofs of NP-completeness. Concerning practical importance, the TSP problem and variations of it has obvious application in scheduling and planning problems. There is no need to argue about the potential money that could be saved by better flight schedules, transportation planning etc.

Finally, just to give an example of how pervasive the concept of NP-completeness really is, consider the Windows game "Minesweeper" (Figure 4). Minesweeper has the property that if the computer provides you with a safe place to start, you can often play the game completely "rational"—you never have to chance it. However, Richard Kaye has demonstrated how you can translate SAT

into a Minesweeper instance. If you play the game rationally, you will have to solve an instance of an NP-complete problem. This means that to play rational may indeed be very hard. At least if the game is not set up at random, but by a vicious computer scientist.

To conclude: NP-complete problems occur naturally everywhere in programming (a list of some NP-complete problems can be found in the appendix). The exact nature of these problems has not yet been determined, even though everyone agrees that it would be of both philosophical as well as practical interest. In the next two sections we will discuss some possible consequences of answering the "P=NP?" question both positively and negatively.

5 What if P=NP?

Great! We have managed to prove that for each and every one of the interesting problems in the NP-complete class there exists an "efficient" (polynomial time) algorithm. But what does this mean?

First of all we need to understand that mathematical proofs come in two flavors: constructive and classic. A classic proof is a proof by contradiction. Such a proof shows that a statement must be true by establishing that the negation of the statement leads to absurdity. If we had such a proof for "P=NP", it would give us no clue as how to *find* an efficient algorithm—it would just establish the *existence* of such an algorithm. On the contrary, from a constructive proof you can actually *construct* the object whose existence has been proved. A constructive proof for "P=NP" would mean we have a polynomial time algorithm for some NP-complete problem.

At this point it should be stressed that a polynomial time algorithm might be just as useless as a super-polynomial (slower than polynomial) algorithm. If the degree of the polynomial is above 10 or so then the algorithm is not really feasible—hence the quotation marks around "efficient". For the algorithm to be truly efficient, we must push the degree down somehow. It is often the case though, that once a polynomial algorithm has been found for a problem, the insight brought by this will allow better algorithms to be devised.

An example that is often mentioned in the P vs NP discussion is *encryption*. These days, encryption is no longer something reserved solely for the military, but is becoming standard practice for information interchange on the Internet. Programs like PGP (Pretty Good Privacy), SSH (Secure Shell) and encryption schemes built into net browsers etc., ensures that no one can eavesdrop to your sensitive transactions on the net, compromising your privacy and security. However, the encryption schemes currently employed is based on the idea of *irreversible functions*. The purpose of such a function is to allow a person to give you his *public* key, which you can use to encrypt something so that it only can be read using his *private* key (which he keeps secret). The scheme relies on the "irreversible" part to be truly non-reversible. This can never be achieved theoretically, only in a computational sense. If no machine that human kind will ever build can have the computational power necessary to break the encryption,

then we might say it is safe. However, this is not the situation of the currently used irreversible functions, which are based on *prime factorization* being a hard problem. Prime factorization is a problem within NP, not even proved NP-complete. Should it be proved that P=NP, then these functions would probably not be computationally irreversible for much longer, and our transaction would be rendered unsafe. Should it be proved that P≠NP, the functions may be safe, but not necessarily so.

6 What if P≠NP?

This is by many regarded as the "expected", and to some extent also the "uninteresting", result. That NP problems could actually be solved efficiently is simply too good to be true, by their reasoning. In a sense, looking for efficient algorithms for NP-complete problems is a bit similar to the search for a solution of the Entscheidungsproblem. Most probably we are going to find it to be impossible. Still, not too infrequently serious attempts at the P=NP stand is published.

But would a proof P≠NP bring nothing good for society or the research community? Well, firstly it would bring a longed-for insight into complexity theory. The field has so far been unable to answer the most fundamental questions. Not only the P=NP question remains open, but also a number of other similar questions like "NP=co-NP?" and "NP=PSPACE?" remain unanswered. Finding a theory for reasoning about these questions would be considered a major step forward in theoretical computer science.

Secondly, should we know for certain that P≠NP, we could actually construct "provably" safe encryption schemes. If a message requires to solve an NP-complete problem in order do decrypt without the key, then it is safe—in a sense by the laws of mathematics. The only thing that might contravene this law is physical phenomenon not yet fully understood or exploited for computation, such as phenomenon of quantum physics.

7 Conclusions

The P=NP is an open problem of dignity and significance. As such, it has even been disputed if it *can* be solved. It is possible that there exists no polynomial algorithm for the SAT problem, but that it is impossible to prove this (using "conventional" mathematics); or it might be the case that there indeed exists such an algorithm, but that we can never prove its time bound. Most scientist however believe that the current state of the question is due to our lack of abilities rather than anything inherent in the mathematical foundation of the theory of computation.

Despite the importance of the issue of P=NP, funding for research projects aiming to resolve this question is running dry. The current attitude seems to be that the P=NP problem simply is too hard to be worth spending more money

on. Ph. D. students are given the advice to look into more fruitful topics, especially as everyone "knows" that P≠NP anyway. Resolution of the current P=NP "controversy" is thus deferred to the brilliant genius who manage to crack it independently. Let us hope his coming is close.

8 Appendix – A list of NP-complete problems

Graph Theory

Vertex Cover
Dominating Set
Domatic Number
Graph K-Colorability
Achromatic Number
Monochromatic Triangle
Feedback Vertex Set
Feedback Arc Set
Partial Feedback Edge Set
Minimum Maximal Matching
Partition Into Triangles
Partition Into Isomorphic Subgraphs
Partition Into Hamiltonian Subgraphs
Partition Into Forests
Partition Into Cliques
Partition Into Perfect Matchings
Covering By Cliques
Covering By Complete Bipartite Subgraphs

Clique
Independent Set
Induced Subgraph With Property Pi*
Induced Connected Subgraph W. Prop. Pi*
Induced Path
Balanced Complete Bipartite Subgraph
Bipartite Subgraph
Degree-Bounded Connected Subgraph
Planar Subgraph
Edge-Subgraph
Transitive Subgraph
Uniconnected Subgraph
Minimum K-Connected Subgraph
Cubic Subgraph
Minimum Equivalent Digraph
Hamiltonian Completion
Internal Graph Completion
Path Graph Completion

Hamiltonian Circuit
Directed Hamiltonian Circuit
Hamiltonian Path
Bandwidth
Directed Bandwidth
Optimal Linear Arrangement
Directed Optimal LInear Arrangement
Minimum Cut Linear Arrangement
Rooted Tree Arrangement
Directed Elimination Ordering
Elimination Degree Sequence

Subgraph Isomorphism
Largest Common Subgraph
Maximum Subgraph Matching

Graph Contractability
Graph Homomorphism
Digraph D-Morphism

Path With Forbidden Pairs
Multiple Choice Matching
Graph Grundy Numbering
Kernel
K-Closure
Intersection Graph Basis
Path Distinguishers
Matric Dimension
Nestril-Rdl Dimension
Threshold Number
Oriented Diameter
Weighted Diameter

Network Design

Degree Constrained Spanning Tree
Maximum Leaf Spanning Tree
Shortest Total Path Length Spanning Tree
Bounded Diameter Spanning Tree
Capacitated Spanning Tree
Geometric Capacitated Spanning Tree
Optimum Communication Spanning Tree
Isomorphic Spanning Tree
Kth Best Spanning Tree*
Bounded Component Spanning Forest
Multipple Choice Branching
Steiner Tree In Graphs
Geometric Steiner Tree

Graph Partitioning
Acyclic Partition
Max Cut
Minimum Cut Into Bounded Sets
Biconnectivity Augmentation
Strong Connectivity Augmentation
Network Reliability*
Network Survivabililty*

Traveling Salesman
Geometric Traveling Salesman
Bottleneck Traveling Salesman
Chinese Postman For Mixed Graphs
Stacker-Crane
Rural Postman
Longest Circuit
Longest Path
Integral Flow With Bundles
Undirected Flow With Lower Bounds
Directed Two-Commodity Integral Flow
Undirected Two-Commodity Integral Flow

Disjoint Connecting Paths
Maximum Length-Bounded Disjoint Paths
Maximum Fiexed-Length Disjoint Paths

Quadratic Assignment Problem
Minimizing Dummy Activities In Pert Networks
Constrained Triangulation
Intersection Graph For Segments On A Grid
Edge Embedding On A Grid
Geometric Connected Dominating Set
Minimum Broadcast Time
Min-Max Multicenter
Min-Sum Multicenter

Sets and Partitions

3-Dimensional Matching (3DM)
Exact Cover By 3-SETS (X3C)
Set Packing
Set Splitting
Minimum Cover
Minimum Test Set
Set Basis
Hitting Set
Inetersection Pattern
Comparative Containment
3-Matriod Intersection

Partition
Subset Sum
Subset Product
3-Partition
Numerical 3-Dimensional Matchining
Numerical Matchining With Target Sums
Expected Component Sum
Minimum Sum Of Squares
Kth Largest Subset*
Kth Largest m-Tuple*

Storage and Retrieval

Bin Packing
Dynamic Storage Allocation
Pruned Trie Space Minimization
Expected Retrieval Cost
Rooted Tree Storage Assignment
Multiple Copy File Allocation
Capacity Assignment

Shortest Common Supersequence
Shortest Common Superstring
Longest Common Subsequence
Bounded Post Correspondence Problem
Hitting String

Sparse Matrix Compression
Consecutive Ones Submatrix
Consecutive Ones Matrix Partition
Consecutive Ones Matrix Augmentation
Consecutive Block Minimization
Consecutive Sets
2-Dimensional Consecutive Sets
String-to-String Correction
Grouping By Swapping
External Macro Data Compression
Internal Macro Data Compression
Regular Expression Substitution
Rectilinear Picture Compression

Minimum Cardinality Key
Additional Key
Prime Attribute Name
Boyce-Codd Normal Form Violation
Conjunctive Query Foldabilty
Conjunctive Boolean Query
Tableau Equivalence
Serializability Of Database Hustories
Safety Of Database Transaction Systems*
Consistency Of Database Frequency Tables
Safety Of File Protection Systems*

Sequencing and Scheduling

Seq. With Release Times And Deadlines
Seq. To Minimize Tardy Tasks
Seq. To Minimize Tardy Task Weight
Seq. To Minimize Weighted Completion Time
Seq. To Minimize Weighted Tardiness
Seq. With Deadlines And Set-Up Times
Seq. Ti Minimize Maximum Cumulative Cost

Multiprocessor Scheduling
Precedence Constrained Scheduling
Resource Constrained Scheduling
Scheduling With Individual Deadlines
Preemptive Scheduling
Scheduling To Min. Weighted Compl. Time

Open-Shop Scheduling
Flow-Shop Scheduling
No-Wait Flow-Shop Scheduling
Two-Processor Flow-Shop W. Bounded Buf.
Job-Shop Scheduling

Timetable Design
Staff Scheduling
Prodcution Planning
Deadlock Aviodance

Mathematical Programming

Integer Programming
Quadratic Programming*
Cost-Parametric Linear Programming
Feasible Basis Extension
Minimum Weight Solution To Linear Eq.
Open Hemisphere
K-Relevancy
Traveling Salesman Polytope Non-Adjacency
Knapsack
Integer Knapsack
Continuous Multiple Choice Knapsack
Partially Orded Knapsack
Comparative Vector Inequalities

Algebra and Number Theory

Quadratic Congruences
Simultaneous Incongruences
Simultaneous Div. Of Linear Polynomials
Comparative Divisibility
Exponential Expression Divisibility*
Non-Divisibility Of A Product Polynomial
Non-Trivial Greatest Common Divisor*

Quadratic Diophantine Equations
Algebraic Equations Over GF[2]
Root Of Modulus 1*
Number Of Roots For A Product Polynomial*
Periodic Solution Recurrence Relation*

Permanent Evaluation*
Cosine Product Integration
Equilibrium Point
Unification With Commutative Operations
Unification For Finitely Presented Algebras
Integer Expression Membership

Games and Puzzles

Generalized Hex*
Generalized Geography*
Generalized Kayles*
Sequential Truth Assignment*
Variable Partition Truth Assignment*
Sift*
Alternating Hitting Set*
Alternating Maximum Weighted Matching*
Annihilation*
N×N Checkers*
N×N Go*
Left-R. Hackenbush For Redwood Furniture
Square-Tiling
Crossword Puzzle Construction
Generalized Instant Insanity

Propositional Logic

Satisfiability
3-Satisfiability (3SAT)
Not-All-Equal 3SAT
One-In-Three 3SAT
Maximum 2-Satisfiability
Generalized Satisfiability
Satisfiability Of Boolean Expressions
Non-Tautology
Minimum Disjunctive Normal Form
Truth-Functionally Complete Connectives

Quantified Boolean Formulas (QBF)*
First Order Theory Of Equality*
Modal Logic S5-Satisfiability
Modal Logic Provability*
Predicate Logic Without Negation
Conjunctive SatẆith Functions And Ineq.
Minimux Axiom Set
First Order Subsumption
Second Order Instantiation

Automata and Language Theory

Finite State Automaton Inequivalence*
Two-Way Finite State Aut. Non-Emptiness*
Linear Bounded Automaton Acceptance*
Quasi-Realtime Automaton Acceptance
Non-Erasing Stack Automaton Acceptance*
Finite State Automata Intersection*
Reduction Of Incompletely Specified Aut.
Minimum Inferred Finite State Automaton

Regular Expression Inequivalence*
Minimum Inferred Regular Expression
Reynold's Covering For Context-Free Gram.
Covering For Linear Grammars*
Strucural Inequivalence For Linear Gram.*
Regular Grammar Inequivalence*
Non-LR(K) Context-Free Grammar
ETOL Grammar Non-Emptiness*
Context-Free Prog. Lang. Membership
Quasi-Real-Time Language Membership
ETOL Language Membership*
Context-Sensitive Language Membership*
Tree Transducer Language Membership*

Program Optimization

Register Sufficiency
Feasible Register Assignment
Register Sufficiency For Loops
Code Generation On A One-Register Machine
Code Generation With Unlimited Registers
Code Generation For Parallel Assignments
Code Generation With Address Expressions
Code Gen. With Unfixed Variable Locations
Ensemble Computation
Microcode Bit Optimization

Inequivalence Of Programs With Arrays
Inequivalence Of Programs W. Assignments
Inequivalence Of Finite Memory Programs*
Inequivalence Of Loop Prog. W/O Nesting
Inequivalence Of Simple Functions
Strong Inequivalence Of Ianov Schemes
Strong Ineq. For Monadic Recursion Schemes
Non-Containment For Free B-Schemes
Non-Freedom For Loop-Free Prog. Schemes
Program With Formally Rec. Procedures

Miscellaneous

Betweenness
Cyclic Ordering
Non-Liveness Of Free Choice Petri Nets
Reachability For 1-Conservative Petri Nets*
Finite Function Generation*
Permutation Generation
Decoding Of Linear Codes
Shapley-Shubik Voting Power
Clustering
Randomization Test For Matched Pairs*
Maximum Likelihood Ranking
Matrix Domination
Matrix Cover
Simply Deviated Disjunction
Decision Tree
Minimum Weight And/Or Graph Solution
Fault Detection In Logic Circuits
Fault Detection In Directed Graphs
Fault Detection With Test Points

* An asterisk indicates that the problem is NP-hard, that is at least as hard as an NP-complete problem, but possibly harder.