Complexity theory, proofs and approximation

Johan Håstad
Royal Institute of Technology
Stockholm, Sweden

January 5, 2011

Abstract

We give a short introduction to some questions in complexity theory and proceed to give some recent developments. In particular, we discuss probabilistically checkable proofs and their applications in establishing inapproximability results. In a traditional proof the proof-checker reads the entire proof and decides deterministically whether the proof is correct. In a probabilistically checkable proof the proof-checker randomly verifies only a very small portion of the proof but still cannot be fooled into accepting a false claim except with small probability.

1 Introduction

The question of what can be done in a completely mechanical way has now been studied for at least 70 years. The early studies led to the invention of the Turing machine [29], a formal model of computation in the form of a primitive computer and the definition that a task can be solved mechanically iff it can be solved by the Turing machine. Many other definitions of mechanically computability were proposed but as they were all proved to be equivalent this led to the consensus that indeed the correct model had been found. The (non-mathematical) statement that computability by the Turing machine indeed captures the true spirit of the intuitive notion of "computable by a mechanical procedure" is usually called "Church's thesis".

With the invention of modern computers it was realized that in practice it does not make a big difference whether a problem cannot be solved at all by a computer or if any general solution requires 10^{200} elementary computational steps. In the latter case, even if every atom in the universe is turned into a super-fast computer, we would not see the end of the computation before our sun has long ago ceased to exist. This realization lead to the development of complexity theory, where we do not only care that the problem can be solved mechanically but where we also study how many elementary computational steps are needed.

One basic parameter of complexity theory is the length of the input. Clearly it is reasonable to expect that more operations are needed to factor a 1024-bit number than a 128-bit number. The variable n is usually used to denote the size of the input. Sometimes it is simply the number of bits needed to specify the input but it is also commonly used to denote a more natural size-parameter closely related to the number of bits needed to specify the input. In particular if one studies graphs, n is usually the number of nodes in the graph, while the number of bits to fully specify a graph on n nodes is $\binom{n}{2}$.

The most studied computational problems are problems where "reasonable" size instances can be solved "reasonably" quickly. This informally defined set of problems in practice coincides well with a class that can be defined formally in a simple way; the class P, polynomial time.

A computational problem belongs to P if the number of elementary steps needed to solve it on instances of size n can be bounded by a polynomial, i.e. as $O(n^k)$.

Many computational problems could be put into the class P, some by straightforward algorithms and some by more sophisticated algorithms, but some natural problems resisted all attempts. Many such problems had the additional property that if indeed the answer was found then it could be verified in polynomial time. An example would be integer factorization. It might be difficult to find the factors but once they are found it is easy to verify that indeed we have a correct solution. This gave birth to the complexity class NP and the most famous problem of complexity theory is whether all problems in NP also belong to P. This problem is still open and is one of the seven million-dollar millennium problems of the Clay institute. In spite of it's importance both in theory and practice it is our belief that this is the Clay problem with the fewest number of active attackers. It seems like many people in the area are waiting for a new idea to surface before it will be possible to fruitfully devote time to this problem. Hopefully that new idea will come soon.

For most people, from an intuitive standpoint, it would seem obvious that $NP \neq P$. If indeed any problem in NP also was in P, this would mean that whenever it is "easy" to verify a found solution it is also "easy" to find the solution. One can think of an NP-statement as a "theorem" and the fact that it is easy to verify a solution would translate into having a short proof. The conclusion would now be that it is also easy to find the proof and for mathematicians spending their lives missing short proofs this would seem especially surprising.

The feeling among almost all people working in computational complexity is that the basic intuition is correct and indeed that $NP \neq P$. The state of complexity theory is such, however, that we currently have no idea how to prove this. The machinery to prove lower bounds is simply too primitive. To prove that a problem does not lie in P we have to prove that any fast

algorithm, no matter how crazy, makes a mistake on some input, and this quantification over all algorithms is problematic.

The fact that we cannot handle this basic question of complexity theory is a major stumbling block to the continued development of the theory as if we cannot tell whether $NP \neq P$ there are many question that we cannot answer. One major technique is to relate other questions to the NP/P-question. One prominent example of this is to prove that a given computational problem, X, is NP-complete or NP-hard. In either case if X belongs to P then $NP = P$. This is in many cases the best available evidence that a problem cannot be solved efficiently.

The notion of NP-completeness was put forth in 1971 by Cook and soon extended by Karp [20]. Somewhat surprisingly, we have since then seen that very many problems are NP-complete. In fact, there are very few, a handful some would argue, natural problems that are not known to lie either in P or to be NP-hard. Some famous examples of such problems would be integer factorization, discrete logarithms and graph isomorphism.

One of the most famous problems in NP is the Traveling Salesperson Problem, TSP, in which we have n cities and are given distances between the cities. The goal is to find a tour that visits all cities once and is of minimal total length. This was one of the first problems showed to be NP-hard in 1972 [20]. Thus if we believe that $NP \neq P$ then we cannot solve this efficiently and optimally for all instances of the problem. This does not prevent us from looking for algorithms with interesting properties. We can study algorithms that find the optimum on random instances, algorithms that finds a reasonably good solution on all instances or even algorithms that finds reasonably good solutions on random instances.

To study random instances one defines a probability distribution on the inputs. If one puts no condition on the probability distribution this is not a very interesting notion in that, in this case, one can make sure that random instances behave like worst case instances. If, however, one demands that the probability distribution is simple, this could possibly change the situation. One definition of simple is that instances with the given distribution can be generated by a probabilistic algorithm running in polynomial time. With this notion Levin [25] proved that complete problems exist but they have proved to be rare and rather special. On the other hand, even for difficult problems, it is often easy to come up with some notion of random instances that makes the problem is easy on the average. Whether such problems capture some real property of the computational problem or are simply a consequence of a "friendly" distribution is sometimes mostly a matter of taste.

A mathematically more appealing notion is that of an approximation algorithm with a guaranteed approximation ratio. Consider TSP discussed above and suppose that we have the triangle-inequality. In this case there is a very efficient algorithms that finds a tour that is at most twice as long

as the optimal tour. There is also a less efficient, but still polynomial time, algorithm by Christofides [10] that finds a tour that is at most a factor 1.5 longer than the optimum. This ratio of approximation is true for *any* input. This raises the question whether this is the best achievable factor and what can be said for other problems. Do all NP-hard optimization problems allow some nontrivial approximation algorithms? We discuss some of the most famous such results in the technical part of this paper.

Positive results that prove that some problems can be solved within specified factors are complemented by results that show, based on some assumption usually that $NP \neq P$, there is no polynomial time algorithm achieving a given factor. Many of these results are based on the very interesting notion called probabilistically checkable proofs. To describe these let us first describe NP as a proof system.

The most typical NP-complete problem is satisfiability of Boolean formulas. We are given a formula φ with logical connectives $\wedge$ (and), $\vee$ (or) and negation and Boolean variables $x_1, x_2, \ldots, x_n$. The question is whether there is an assignment that makes the formula evaluate to true. If there is such an assignment, once it is found it can be checked quickly. We can view this assignment as a proof that φ is satisfiable. It is checked by a proof-checker, which in computer science traditionally is called a verifier and denoted by V, that simply evaluates the formula on this assignment and verifies that it evaluates to true.

This is an excellent proof-system, each true statement of the given type, i.e. "φ is satisfiable" has a proof that is fairly short and can be checked efficiently by V. The proof-system is perfectly sound in that V is never convinced of an incorrect statement. It seems like there is little more we can hope for but it turns out to be profitable to ask how much of the proof V has to read. Naively one would think that V would have to read the entire proof but this is in fact not true in general.

In a PCP we have a statement, still on the form "φ is satisfiable" and a written proof. The verifier, V is, however, probabilistic and reads only a very small portion of the proof. In fact, it might decide to look at as little as three bits of the proof. Completeness is as before in that V accepts a correct proof for a correct theorem with probability 1. Soundness has to be relaxed and in fact all that can be achieved is that any proof for an incorrect statement is found to be incorrect with a constant probability $s < 1$. It is a remarkable theorem by Arora et al. [5], the PCP-theorem that this is in fact possible.

We elaborate on the connection to approximability in the technical part of the paper but let us at least give a hint.

Given a statement φ, which we do not know if it is satisfiable or not we can consider the optimization problem of finding the "best" proof for it. The quality of the proof is defined as the probability that V accepts it. If φ is satisfiable we know that there is a proof that makes V accept with

probability 1. On the other if φ is not satisfiable no proof makes V accept with probability greater than s. This implies that if we could approximate the optimum of the proof optimization problem within a factor better than $1/s$ we could in fact determine whether φ is satisfiable. Since the latter is an NP-hard problem so is the former and thus "all" we have to do is to construct the PCP in such a way that the proof optimization problem is in fact the optimization problem we care about.

For the record let us note that there are other ways to use PCPs to get inapproximability results but the proof optimization problem is the most basic.

In the rest of this paper we essentially retell the story told in the introduction but in a more technical way. We give essentially no proofs and some of the definitions are not totally rigorous but the aim is to give the interested reader enough detail to convey a feeling for the area.

2 Basic definitions

We are studying efficient algorithm for computational problems and thus we should define "algorithm" and "computational problem".

The standard formal definition of an algorithms goes through the notion of a Turing machine which is a bit cumbersome and we choose not to do this. It is also the case that almost any intuitive notion of an algorithm can be formalized in a suitable manner leading to an equivalent notion.

Anybody that has written a computer program and is comfortable with formal definitions should easily be able to abstract the notion. The only crucial point is that we want the computer words to be bounded in size. If we bound the size by a constant this makes the model slightly awkward in that indirect addressing does not allow us to access all of memory in a straightforward manner. If we allow ourselves words that are of a length that is logarithmic in the amount of memory we use, indirect addressing works without problems and we get a robust, simple and intuitive model of computation. The number of operations is defined as the number of machine steps performed.

Those who have not programmed have done calculation by hand. There is no essential difference to machine calculation. The fact that the word size of the computer is limited is reflected in the fact that we have a finite number of symbols. Now you can simply count the number of symbols written and erased. The details of the model does effect the number of operations but not on the level of detail we discuss in this paper.

The formal definition of a computational problem is simple but maybe not very informative. We use the binary alphabet i.e. $\Sigma = \{0, 1\}$ and inputs and outputs of our algorithms are nonempty finite strings over Σ and this is denoted by Σ^*. A computational problem is now simply a mapping from

Σ^* to Σ^*. To make informal sense of a computational problem we need a more intuitive way of thinking of the input and the output. Most of the time, this is easy; integers are specified by their representation in the binary number system, text as the ASCII-value of their characters, etc. Sometimes the situation is more complicated and in particular if we want an algorithm to deal with more or less arbitrary real numbers we have to be careful but this goes beyond the scope of this paper.

For each computational problem we have a parameter n which nicely measures the size of the instance.

Let us take an example. Consider the problem of adding and multiplying two integers each with n bits. It is not difficult to convince oneself that the standard grade-school algorithm for adding two numbers runs in $O(n)$ time and this is optimal since any algorithm must read its input. The grade-school algorithm for multiplying two numbers multiplies each digit of one number with each digit of the other resulting in an $O(n^2)$ time algorithm. There are many ways to improve this and the fastest algorithm designed by Schönhage and Strassen already in 1971 [28] runs in time $O(n \log n \log\log n)$.

A fundamental question that now arises is whether this is optimal and the honest answer to this question is that we have no idea. There is in fact no lower bound for the complexity of multiplication that goes essentially beyond the fact that we have to read the input and write the output. In particular it is possible, but in many peoples eyes unlikely, that multiplication can be done in time $O(n)$. Thus in one sense, complexity theory has not left first grade as we do not understand multiplication. Note, however, that it is not obvious that resolving this question is simpler than deciding the NP/P-question. Proving a lower bound larger than cn for multiplication for any constant c is a fairly subtle issue when we know that the true bound is a most $O(n \log n \log\log n)$. To prove that $NP \neq P$ we need to prove the lower bound n^k for any k on the number of operations to solve Satisfiability and as we believe the true bound is more like 2^n the margins here appears larger and more crude methods might apply. To paraphrase, the lower bound for multiplication will probably need something like a very sharp knife while proving $NP \neq P$ might need something closer to a nuclear bomb.

3 NP and P

Let us give a formal definition of NP, or at least what would have been a formal definition, had we defined our computational model and running time of an algorithm formally.

To make formal sense of NP we focus on decision problems. A decision problem is a computational problem where we limit the output to a single bit. The standard terminology in this case would be that inputs that map to 1 are "accepted" and inputs that map to 0 are "rejected". Many times

one calls the elements of NP "languages" where a language is the subset of Σ^* given by the accepted inputs.

Definition 1 *Let $L \subseteq \Sigma^*$. $L \in NP$ iff there is a Turing machine M that runs in time polynomial in the length of its first input such that $x \in L$ iff there exists y such that $M(x, y) = 1$.*

We could require that the length of y is polynomial in the length of x but this is assured by the fact that M can only read a polynomial number of bits in polynomial time.

Satisfiability is the standard NP-problem. It is the language of (codings of) satisfiable Boolean formulas. The input y is an assignment to the variable occurring in the formula coded by x and M checks whether this assignment satisfies the formula.

Note further that TSP as described in the introduction does not belong to NP as it is not a decision problem. To make it a decision problem we can introduce a parameter K, and ask whether there exists a tour of length at most K. The problem now belongs to NP. It is not always important to make the distinction between the optimization problem and the decision problem but on the formal level this might cause some confusion.

As we want to make $P \subseteq NP$ we define also P as a set of decision problems.

Definition 2 *Let $L \subseteq \Sigma^*$. $L \in P$ iff there is a Turing machine M that runs in time polynomial in the length of its input such that $x \in L$ iff $M(x) = 1$.*

We proceed to make a formal definition of the property of being NP-complete. We want to capture the idea of having a subroutine that decides a language L. Such a machine, traditionally denoted by M^L, is given the ability to ask questions of the type "$x \in L$?" which are answered correctly in one elementary step. Such machines are called "oracle Turing machines" and L is the called the "oracle language".

Definition 3 *Let $L \subseteq \Sigma^*$. L is NP-*complete *iff $L \in NP$ and for any language $L' \in NP$ there is an oracle Turing machine M^L that runs in time polynomial in the length of its input such that $x \in L'$ iff $M^L(x) = 1$.*

Note that if L is NP-complete and L belongs to P then so does any language in NP as we can replace calls to the oracle with a polynomial time machine deciding L. A language is NP-hard if we drop the requirement that it belongs to NP.

Definition 4 *Let $L \subseteq \Sigma^*$. L is NP-*hard *iff for any language $L' \in NP$ there is an oracle Turing machine M^L that runs in time polynomial in the length of its input such that $x \in L'$ iff $M^L(x) = 1$.*

We extend the above notion to non-decision problems by saying that giving a subroutine that solves the given problem we can decide an arbitrary language in NP in polynomial time.

There are thousands of known NP-hard and NP-complete problems. Satisfiability is NP-complete and TSP in its decision form is NP-complete and in its optimization form it is NP-hard. Thus we expect that none of these problems can be solved in polynomial time.

Problems in NP can now be classified to be of three types. They can be NP-complete, belong to P or neither. Surprisingly the third category is very rare for natural problems and with few exceptions, already by early 1980's most problems were known to be either NP-complete or to belong to P. The main progress on this set of problems in the last decade has been on a more refined measure of hardness.

4 Approximation algorithms

Given an NP-hard optimization problem we can study polynomial time heuristics that return good but possibly not optimal solutions. For our model problem TSP, a large number of heuristics are known and many are discussed by Johnsson and McGeoch in [18]. Many heuristics are hard to analyze and best evaluated experimentally but for some strong and precise statements can be made. Let O be an optimization problem with instances x and solutions y where the objective value is $Val(x, y)$. For TSP x is thus a set of distances, y is a proposed order in which to visit the cities and $Val(x, y)$ is the total length of the tour given by y with distances x. The optimal value for a minimization problem is defined as

$$Opt(x) = \min_y Val(x, y).$$

Definition 5 *An algorithm A is a C-approximation for an minimization problem O if it for each instance x, $Val(x, A(x)) \leq C \cdot Opt(x)$.*

The approximation ratio for maximization problems is defined in an analogous. We let

$$Opt(x) = \max_y Val(x, y).$$

Definition 6 *An algorithm A is a C-approximation for an maximization problem O if it for each instance x, $Val(x, A(x)) \geq Opt(x)/C$.*

Sometimes one requires an approximation algorithm not to output a solution but only an estimate for the optimal value. It is interesting that almost all lower bounds apply to this weaker model while the almost all known upper bounds are given by an algorithm in the stronger model.

Sometimes we allow A to be a randomized algorithm. We then study $E[Val(x, A(x))]$ where the expectation is taken only over the random choices of A and we emphasize that there is no randomization over the input and the bound is true for worst-case inputs. We now turn to our main example which is of both practical and theoretical interest.

5 Linear systems of equations

Systems of linear equations over different fields appear in many situations. We are given a set of equations

$$\sum_{i=1}^{n} a_{ij} x_i = b_j, \quad 1 \leq j \leq m$$

and we want to find values of x_i to satisfy these equations in an as good way as possible. If way can satisfy all equations then such an assignment can be found in polynomial time by Gaussian elimination, or even more efficient algorithms in some situations. The most interesting situation for us now is the case when the system is inconsistent.

If we cannot satisfy all equation there are sometimes several possible definitions of "best solution". If the field in question is the rational numbers one common definition of best is the least squares approximation i.e. to minimize

$$\sum_{j=1}^{m} \left(\sum_{i=1}^{n} a_{ij} x_i - b_j \right)^2$$

and also in this case it is possible to find the best solution in polynomial time. Another extreme is when the field is the field with two elements, $GF[2]$, where the two elements are 0 and 1 and addition is performed modulo 2. In this situation the only possible measure is to maximize the number of satisfied equations and this is the measure we adopt for any field.

Definition 7 *For a field F let Max-Lin-F be the optimization problem that given a set of linear equations to simultaneously satisfy the maximal number of equations. If F is the field of p elements we call the problem Max-Lin-p.*

It is not difficult to classify these problems on the NP-hardness scale and the following theorem is a possible exercise in a basic complexity class.

Theorem 5.1 *For any prime p, Max-Lin-p in its decision form is NP-hard and this is also true for Max-Lin-$\mathbf{Q}$, where $\mathbf{Q}$ is the field of rational numbers.*

Let us turn to the approximability of Max-Lin-p. Suppose we have m equations. If we pick an assignment to the variables uniformly at random then we satisfy each equation with probability $1/p$ and thus we expect to satisfy, on the average, m/p equations. This leads to a randomized

p-approximation algorithm but it is not difficult to make a deterministic algorithm that finds a solution that satisfies at least m/p equations. We have the following theorem:

Theorem 5.2 *For any prime p one can, in deterministic polynomial time, approximate Max-Lin-p within a factor of p.*

This is complemented by the following theorem by Håstad [16].

Theorem 5.3 *For any prime p and any $\epsilon > 0$, it is NP-hard to approximate Max-Lin-p within $p - \epsilon$.*

Thus in particular, even if we know that there is an assignment that satisfies almost all equations there is no efficient way to find an assignment that does significantly better than a random assignment.

The result applies as long as we allow three variables in each equation and has been extended by Engebretsen et al. [12] to apply to any group. On the other hand, if we only allow two variables in each equation we do get non-trivial approximation for any p [14, 3].

Over the rational numbers our knowledge is not quite as complete. We can pick a maximal set of linearly independent equations and satisfy these equations disregarding the remaining equations. This does not yield a very good approximation ratio but we should not hope for too much in view of the following lower bound by Amaldi and Kann [2]:

Theorem 5.4 *There is a $\delta > 0$ such that it is NP-hard to to approximate Max-Lin-$\mathbf{Q}$ within n^δ.*

The proof is Theorem 5.3 is, in principle, simple. We start with a Boolean formula φ and any $\delta > 0$. We construct, in polynomial time, a linear system L of m equations. The make sure that if φ is satisfiable then there is an assignment that satisfies $(1 - \delta)m$ of the equations of L while if φ is not satisfiable, no assignment satisfies more than a fraction $(\frac{1}{p} + \delta)m$ of the equations. It follows that any algorithm that determines the maximal number of simultaneously satisfiable solution within a factor smaller than

$$\frac{1 - \delta}{\frac{1}{p} + \delta}$$

can be used to determine whether φ is satisfiable or not and hence it must be an NP-hard task to achieve this approximation ratio. Choosing δ a suitable function of ϵ now establishes the result.

This reduction of creating L from φ is just a computational procedure and could be described by a combinatorial algorithm. It has, however, been profitable to think in terms of proof systems and we turn to probabilistically checkable proof.

6 Probabilistically Checkable Proofs

First let us phrase NP as a proof system.

Definition 8 *A Turing machine V running in polynomial time in the length of its first input is a verifier in an NP- proof system for a language L iff*

- *For $x \in L$ there exists a π such that $V(x, \pi) = 1$.*
- *For $x \notin L$, for all π, $V(x, \pi) = 0$.*

The machine V is called the verifier and it is the same as the machine M in Definition 1. We are interested in discussing verifiers that read a very small portion of the proof. It is most convenient to use the concept of an oracle Turing machine as already used in Definition 3. This time we let V access the proof by asking questions "i?" which is answered by π_i, the i'th bit of the proof. We also assume that V is probabilistic and this is achieved by having a source of "random coins" which are bits each taking the value 0 with probability $\frac{1}{2}$ independently of each other and the input. We denote the random string by r.

Definition 9 *Let c and s be real numbers such that $1 \geq c > s \geq 0$. A probabilistic polynomial time Turing machine V is a verifier in a* Probabilistically Checkable Proof (PCP) *with soundness s and completeness c for a language L iff*

- *For $x \in L$ there exists an oracle π such that $Pr_r[V^\pi(x, r) = 1] \geq c$.*
- *For $x \notin L$, for all π $Pr_r[V^\pi(x, r) = 1] \leq s$.*

In many circumstances one would expect a good verifier to always accept a correct proof of a correct statement and $c = 1$ is also the most common value but values slightly below 1 for c are also useful.

The famous PCP-theorem [5] can now be stated as follows:

Theorem 6.1 *Any $L \in NP$ allows a PCP with perfect completeness ($c = 1$), constant soundness $s < 1$, where V only accesses three bits of π and uses $O(\log n)$ random coins on inputs of length n. The size of π is polynomial in n.*

Even a sketch of the proof of this theorem would take us too far. One key idea is to code the satisfying assignment as the outputs of a low degree polynomial over a finite field, a second is to use proof-composition, a type of recursive proof technique. Both were introduced prior to [5] and we refer to that paper for a discussion of the history.

To see the connection to inapproximability we consider the proof optimization problem.

Definition 10 *Let V be a verifier in a PCP for a language L.* The proof optimization *problem is that given an input x to determine the maximal probability with which V accepts x.*

We have the following trivial observation.

Theorem 6.2 *If the verifier V has soundness s and completeness c then if we can determine the optimum of the proof optimization problem within a factor smaller than c/s then we can decide membership in L with the same amount of resources.*

Proof: Suppose that we have an algorithm A that determines the value of the proof optimization problem within a factor $k < \frac{c}{s}$. Then, on input x, run A and if the value of the obtained solution is greater than s accept the output and otherwise reject.

By the soundness condition of the proof-system whenever we accept the input this is the correct decision. The fact that we always accept elements of L is implied by the completeness condition and the assumed approximation ratio. ∎

The key now to getting interesting in-approximability results is to design a PCP for an NP-complete problem with the property that the proof optimization problem is in fact equivalent to an optimization problem we care about. Let us describe the properties of the PCP that underlies the proof of Theorem 5.3 in the case of $p = 2$.

Given a parameter δ, the proof consists of a polynomial number of bits $(\pi_j)_{j=1}^{n^k}$ and is verified as follows. V flips $O(\log n)$ random coins to determine three addresses j_1, j_2 and j_3 and a bit b. The verifier now accepts if the exclusive-or of π_{j_1}, π_{j_2} and π_{j_3} equals b. The completeness is $1 - \delta$ and the soundness is $\frac{1}{2} + \delta$.

Now we can see that proof optimization problem is just Max-Lin-2 in disguise. Optimizing over the proof is the same as thinking of the i'th bit of the proof as a variable x_i and then to optimize over these variables. Suppose that V flips R coins. Each possible outcome of the random coins leads to a linear equation which determines whether V accepts on this particular set of coin flips. We end up with 2^R equations and the maximum fraction of simultaneously satisfiable equations is exactly the maximum probability to convince the verifier.

Note that it is important that the verifier does not use too many random coins as the number of different sets of coinflips is the number of resulting equations. Also it is important that the proof is small in that each bit of the proof directly corresponds to a variable in the linear system of equations.

To describe in detail how to construct this PCP is not feasible in these notes and we refer to the original paper [16]. On the very high level, the proof utilizes Theorem 6.1 as a black box and then improves the parameters.

This is done by repeating the proof in parallel and then condensing the answers using an interesting binary code called the long code and proposed by Bellare et al. [6]. The long code of input $v \in \{0,1\}^t$ is indexed by functions $f : \{0,1\}^t \mapsto \{0,1\}$ and the value at position f is $f(v)$. Thus 2^{2^t} bits are used to code t bits and it is the longest binary code, disallowing coordinates that are equal for each pair of inputs. This code is extremely long but as it is used for constant size inputs it length does not affect the results except in that the implicit constants are rather weak.

Let us now consider some other problems.

7 Independent set and Coloring

Given a graph G, the independent set problem is to find the largest number of nodes of which no two are connected. A related problem is "clique" where we ask for the largest number of nodes all of which are pairwise connected. These two problems are clearly equivalent as can be seen from changing edges to non-edges.

Independent set initially sounds like an innocent problem and for a while it was somewhat surprising that, for graphs with n nodes, the best approximation ratio achieved by any polynomial time algorithm was as poor as $O(\frac{n}{(\log n)^2})$ [8]. This implies that even for a graph which has an independent set of size linear in the number of nodes the algorithm can guarantee only that we find an independent set of size $\Omega((\log n)^2)$. For graphs with an independent set as large $O(n/(\log n)^2)$ the algorithm gives no guarantee.

This poor performance was explained by subsequent lower bounds. Based on the assumption that NP cannot be solved in probabilistic polynomial time Håstad [15] proved that for any $\epsilon > 0$ one cannot approximate independent set within a factor $n^{1-\epsilon}$. Making stronger, but still almost universally believed assumptions Khot [22] showed that it is possible make ϵ decrease as $(\log n)^{-\gamma}$ for some $\gamma > 0$. Thus what seemed to be trivial upper bounds pointed very much in the correct direction namely that independent set is indeed a very difficult problem.

To get this inapproximability results, very strong PCPs are needed and the required properties have very natural parameters also when formulated as proof systems. Suppose we restrict V to use $O(\log n)$ random coins and to read q bits of the proof, require (almost) perfect completeness and we are looking to minimize the soundness. It was established by Samorodnitsky and Trevisan [27] that if we allow non-perfect completeness one could achieve soundness $2^{-q+O\sqrt{q}}$. This was later extended to perfect completeness by Håstad and Khot [17]. It is amazing that the probability of being cheated essentially decreases by a factor of 2 for each bit read. Through a sequence of reductions this gives the desired bound for independent set.

A very related problem is graph coloring. In this case we want to color

the nodes in a graph in order that any two adjacent nodes are of different colors. The objective function to be minimized is given by the number of colors. Note that each color class is an independent set and using this it is possible to prove that a good approximation algorithm for independent set would have yielded an almost as good approximation algorithm for coloring, but no direct reduction is known in the other direction. Feige and Kilian [30] showed, however, that it is possible to extend the lower bounds of independent set to coloring and thus also this problem is very difficult to approximate.

Of special interest are graph which can be colored with very few colors, the first interesting case being three-colorable graphs. This is one of the major open problems of the area of approximability. By a result of Blum and Karger [7] it is known how to color such a graph in polynomial time with roughly $O(n^{3/14})$ colors while the best lower bound by Khanna et el. [21] is that unless $P = NP$ it cannot be done with 4 colors. Most people in the area seem to expect the true answer to be of the form $O(n^{\delta})$ for some positive δ but this conjecture must be considered highly uncertain.

8 Maximum cut

Maximum cut is the following problem. Given a graph, divide the nodes into two groups V_1 and V_2 in order that the maximum number of edges that are cut, i.e. go between the two parts.

For a long time, the best approximation algorithm for this problem was a random assignment, giving an approximation ratio of 2 as a random assignment cuts half the edges on the average.

A leap forward was made by Goemans and Williamson [14] when semi-definite programming was introduced as a tool to achieve good provable approximation ratios. Linear programming had long been used as a tool for designing heuristics and semi-definite programming is an extension. In a semi-definite program, we have a set of variables organized in a matrix. Apart from linear conditions on the variables we also have the constraint that the matrix is positive semi-definite. Assuming a linear objective function, the optimum can, by a result of Alizadeh [1], be found to any desired accuracy. One reason one could hoped for semi-definite programming to be solved efficiently is that the set of semi-definite matrices form a convex set and hence there is no problem with local extrema.

Using this method for maximum cut, Goemans and Williamson [14] found a polynomial time approximation algorithm with approximation ratio

$$\max_{\theta} \frac{\pi}{2} \cdot \frac{1 - \cos\theta}{\theta} \approx 1.138. \tag{1}$$

This algorithm remains the champion while the lower bound on approximability is $17/16 - \epsilon$ for any $\epsilon > 0$ [16]. There has recently been work by

Khot et al. [23] indicating the the upper bound might be the correct answer. Given two strong, but not unrealistic conjectures, one can prove upto an arbitrary $\epsilon > 0$ matching lower bounds.

9 Set cover

In set cover we are given a sequence of subsets $(S_i)_{i=1}^m$ of a universe X of cardinality n. The goal is to find a minimal size sub-collection that covers X.

There is a straightforward greedy algorithm for this problem. Keep picking the set that covers the maximal number of uncovered elements. If the optimal covering contains k elements then it is not difficult to see that at each iteration we cover at least a fraction $\frac{1}{k}$ of the uncovered elements. The number of remaining uncovered elements after t sets have been picked is thus at most

$$(1 - \frac{1}{k})^t n$$

and it follows that after at most $k \ln n$ sets have been picked, all elements are covered. We conclude that we get an $\ln n$ approximation algorithm which was first described by Johnson [19]. This is complemented by a lower bound that says that if NP is not contained in deterministic time $n^{O(\log \log n)}$ then no polynomial time algorithm can approximate set cover within a factor $(1 - o(1)) \ln n$. Slightly weaker results are known if we are only willing to assume $NP \neq P$.

10 Vertex cover

Vertex cover is the special case of set cover where each element only appears in two sets. This is mostly easily visualized as a graph. The edges of the graph corresponds to the elements while each node gives a set defined by the edges incident to that node. The task now is to find the minimal number of nodes such that each edge has at least one endpoint in the picked set.

There are many ways to approximate this problem within a factor 2 and one is to relax it to linear programming. Introduce a variable x_i for each node and minimize

$$\sum_{i=1}^{n} x_i$$

given the constraint

$$x_i + x_j \geq 1$$

for any edge (i, j) as well as $x_i \geq 0$ for any i. Clearly any legitimate solution to the vertex cover gives a solution to the linear program by making $x_i = 1$ when i is included in the solution and setting $x_i = 0$ otherwise.

Thus we know that the optimum to the linear program is at most the value of the optimal solution to vertex cover. The optimal solution to any linear program can be found in polynomial time but the optimal solution probably takes values outside $\{0, 1\}$ and hence do correspond directly to a vertex cover. To recover a correct solution to vertex from a general solution to the linear program one can proceed as follows.

For any i with $x_i \geq 1/2$ increase x_i to 1 while otherwise set $x_i = 0$. It is not difficult to see that the cost increases by at most a factor 2 and we get a solution for vertex cover giving an efficient 2-approximation algorithm.

The strongest known lower bound on approximability for vertex cover by Dinur and Safra [11], is that it is NP-hard to approximate within $10\sqrt{5} - 21 - \epsilon \approx 1.36$ for any $\epsilon > 0$. Khot and Regev [24] have proved that, again subject to an unproven and slightly speculative conjecture, that the upper bound is the correct value.

11 Traveling salesperson problem

Let us finally return to TSP. In most reasonable circumstances, instances obey the triangle inequality so let us concentrate on this case.

If we only assume the triangle inequality the algorithm by Christofides [10] with the best approximation ratio has been known for over 20 years and it gives a factor 1.5. Here we have a lower bound but much weaker than for other problems. The best lower bound with a fully published proof is 3813/3812 by Böckenhauer and Seibert [9], but stronger results are in the process of being verified. It seems, however, that a ratio of 1.01 is not achievable by the current methods.

One interesting subcase is that the cities are points in the two-dimensional plane and the distances are Euclidean distances. To find the optimal solution is this case was early on proved to be NP-hard by Papadimitriou [26]. For a long time, the algorithm of Christofides remained the best also in this case but eventually a celebrated result by Arora [4] showed that the Euclidean structure can be used and if fact for any $\epsilon > 0$ it is possible to find an approximation within a factor $(1 + \epsilon)$ in polynomial time. Thus the Euclidean case is provably simpler than the general case with the triangle inequality.

An interesting extension is that of non-symmetric TSP, i.e. where it is possible that $d(i, j) \neq d(j, i)$ which is quite possible in many models of reality, even for a modern salesperson with prevailing western winds playing a factor at long distance flights.

Clearly any lower bound for the symmetric model also applies to asymmetric case and in fact the bounds can be strengthened slightly but no bound beyond 1.01 is currently claimed. More interestingly all approximation algorithms that give a constant approximation factor relies on the

distance-function being symmetric and the smallest achievable approximation ratio in polynomial time is currently $O(\log n)$, the first such algorithm was given by Frieze et al. [13]. It is difficult to guess what the true bound might be and we end with this totally open question.

12 Final words

If most problems were classified as either in P or as NP-hard by the 1980'ies we are now closing in on knowing approximability of most NP-hard optimization problems. Clearly many problems do remain but progress since the beginning 1990'ies when this research started has been spectacular. One cannot help to be amazed that it keeps being the case that either problems are solvable in polynomial time or turn out to be NP-hard. The in-between case, that one can prove must occur by constructing artificial problems, continues to be rare for natural problems. What this is so, we can only speculate.

References

[1] F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5:13–51, 1995.

[2] E. Amaldi and V. Kann. The complexity and approximability of finding feasible subsystems of linear relations. *Theoretical Computer Science*, 147:181–210, 1995.

[3] G. Andersson, L. Engebretsen, and J. Håstad. A new way to use semidefinite programming with applications to linear equations mod p. *Journal of Algorithms*, 39:162–204, 2001.

[4] S. Arora. Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. *Journal of the ACM*, 45:753–782, 1998.

[5] S. Arora, C. Lund, R. Motwani, M. Sudan, and M.Szegedy. Proof verification and intractability of approximation problems. *Journal of the ACM*, 45:501–555, 1998.

[6] M. Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs and non-approximability—towards tight results. *SIAM Journal on Computing*, 27:804–915, 1998.

[7] A. Blum and D. Karger. An $\tilde{O}(n^{3/14})$-coloring algorithm for 3-colorable graphs. *Information processing letters*, 61:49–53, 1997.

[8] R. Boppana and M. Haldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32:180–196, 1992.

[9] H.-J. Böckenhauer and Sebastian Seibert. Improved lower bounds on the approximability of the traveling salesman problem. *RAIRO Theoretical Informatics and Applications*, 34:213–255, 2000.

[10] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.

[11] I. Dinur and S. Safra. On the importance of being biased. In *Proceedings of 34th Annual ACM symposium on Theory of Computing*, pages 33–42, 2002.

[12] L. Engebretsen, J. Holmerin, and A. Russell. Inapproximability results for equations over finite groups. *Theoretical Computer Science*, 312:17–45, 2004.

[13] A. Frieze, G. Galbiati, and F. Maffioli. On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12:23–39, 1982.

[14] M. Goemans and D. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.

[15] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[16] J. Håstad. Some optimal inapproximability results. *Journal of ACM*, 48:798–859, 2001.

[17] J. Håstad and S. Khot. Query efficient PCPs with perfect completeness. In *Proceedings of 42nd Annual IEEE Symposium of Foundations of Computer Science*, pages 610–619, 2001.

[18] D. Johnson and L. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Ltd., 1997.

[19] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal Computer and System Sciences*, 1974:256–278, 9.

[20] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[21] S. Khanna, M. Linial, and S. Safra. On the hardness of approximating the chromatic number. In *Proceedings of the 2nd Isreal Symposium on Theory of Computing*, pages 250–260. IEEE Computer Society, 1993.

[22] S. Khot. Improved inapproximability results for maxclique and chromatic number. In *Proceedings of 42nd Annual IEEE Symposium of Foundations of Computer Science*, pages 600–609, 2001.

[23] S. Khot, E. Mossel G. Kindler, and R. O'Donnell. Optimal inapproximability results for max-cut and other 2-variable CSPs. to appear at FOCS 2004.

[24] S. Khot and O. Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. In *Proc. of 18th IEEE Annual Conference on Computational Complexity (CCC)*, pages 379–386, 2003.

[25] L. Levin. Average case complete problems. *SIAM Journal on Computing*, 15:285–286, 1986.

[26] C. Papadimitriou. Euclidean TSP is NP-complete. *Theoretical computer science*, 4:237–244, 1977.

[27] A. Samorodnitsky and L. Trevisan. A PCP characterization of NP with optimal amortized query complexity. In *In proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 191–199, 2000.

[28] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[29] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. Ser 2*, 42:230–265, 1936.

[30] U.Feige and J. Kilian. Zero-knowledge and the chromatic number. *Journal of Computer and System Sciences*, 57:187–200, 1998.