Exact algorithms for NP-hard problems: A survey

Gerhard J. Woeginger

Department of Mathematics
University of Twente, P.O. Box 217
7500 AE Enschede, The Netherlands

Abstract. We discuss fast exponential time solutions for NP-complete problems. We survey known results and approaches, we provide pointers to the literature, and we discuss several open problems in this area. The list of discussed NP-complete problems includes the travelling salesman problem, scheduling under precedence constraints, satisfiability, knapsack, graph coloring, independent sets in graphs, bandwidth of a graph, and many more.

1 Introduction

Every NP-complete problem can be solved by exhaustive search. Unfortunately, when the size of the instances grows the running time for exhaustive search soon becomes forbiddingly large, even for instances of fairly small size. For some problems it is possible to design algorithms that are significantly faster than exhaustive search, though still not polynomial time. This survey deals with such fast, super-polynomial time algorithms that solve NP-complete problems to optimality. In recent years there has been growing interest in the design and analysis of such super-polynomial time algorithms. This interest has many causes.

- It is now commonly believed that P$\neq$NP, and that super-polynomial time algorithms are the best we can hope for when we are dealing with an NP-complete problem. There is a handful of isolated results scattered across the literature, but we are far from developing a general theory. In fact, we have not even started a systematic investigation of the worst case behavior of such super-polynomial time algorithms.
- Some NP-complete problems have better and faster exact algorithms than others. There is a wide variation in the worst case complexities of known exact (super-polynomial time) algorithms. Classical complexity theory can not explain these differences. Do there exist any relationships among the worst case behaviors of various problems? Is progress on the different problems connected? Can we somehow classify NP-complete problems to see how close we are to the best possible algorithms?
- With the increased speed of modern computers, large instances of NP-complete problems can be solved effectively. For example it is nowadays routine to solve travelling salesman (TSP) instances with up to 2000 cities.

And if the data is nicely structured, then instances with up to 13000 cities can be handled in practice (Applegate, Bixby, Chvátal & Cook [2]). There is a huge gap between the empirical results from testing implementations and the known theoretical results on exact algorithms.
- Fast algorithms with exponential running times may actually lead to practical algorithms, at least for moderate instance sizes. For small instances, an algorithm with an exponential time complexity of $O(1.01^n)$ should usually run much faster than an algorithm with a polynomial time complexity of $O(n^4)$.

In this article we survey known results and approaches to the worst case analysis of exact algorithms for NP-hard problems, and we provide pointers to the literature. Throughout the survey, we will also formulate many exercises and open problems. Open problems refer to unsolved research problems, while exercises pose smaller questions and puzzles that should be fairly easy to solve.

Organization of this survey. Section 2 collects some technical preliminaries and some basic definitions that will be used in this article. Sections 3–6 introduce and explain the four main techniques for designing fast exact algorithms: Section 3 deals with dynamic programming across the subsets, Section 4 discusses pruning of search trees, Section 5 illustrates the power of preprocessing the data, and Section 6 considers approaches based on local search. Section 7 discusses methods for proving *negative* results on the worst case behavior of exact algorithms. Section 8 gives some concluding remarks.

2 Technical preliminaries

How do we measure the quality of an exact algorithm for an NP-hard problem? Exact algorithms for NP-complete problems are sometimes hard to compare, since their analysis is done in terms of different parameters. For instance, for an optimization problem on graphs the analysis could be done in terms of the number n of vertices, or possibly in the number m of edges. Since the standard reductions between NP-complete problems may increase the instance sizes, many questions in computational complexity theory depend delicately on the choice of parameters. The right approach seems to be to include an explicit complexity parameter in the problem specification (Impagliazzo, Paturi & Zane [21]). Recall that the decision version of every problem in NP can be formulated in the following way:

Given x, decide whether there exists y so that $|y| \leq m(x)$ and $R(x, y)$.

Here x is an instance of the problem; y is a short YES-certificate for this instance; $R(x, y)$ is a polynomial time decidable relation that verifies certificate y for instance x; and $m(x)$ is a polynomial time computable and polynomially bounded *complexity parameter* that bounds the length of the certificate y. A trivial exact algorithm for solving x would be to enumerate all possible strings with lengths

up to $m(x)$, and to check whether any of them yields a YES-certificate. Up to polynomial factors that depend on the evaluation time of $R(x,y)$, this would yield a running time of $2^{m(x)}$. The first goal in exact algorithms always is to break the triviality barrier, and to improve on the time complexity of this trivial enumerative algorithm.

Throughout this survey, we will measure the running times of algorithms *only* with respect to the complexity parameter $m(x)$. We will use a modified big-Oh notation that suppresses all other (polynomially bounded) terms that depend on the instance x and the relation $R(x,y)$. We write $O^*(T(m(x)))$ for a time complexity of the form $O(T(m(x)) \cdot \text{poly}(|x|))$. This modification may be justified by the exponential growth of $T(m(x))$. Note that for instance for simple graphs with $m(x) = n$ vertices and m edges, the running time $1.7344^n \cdot n^2 m^5$ is sandwiched between the running times 1.7344^n and 1.7345^n.

We stress, however, the fact that the complexity parameter $m(x)$ in general is not unique, and that it heavily depends on the representation of the input. For an input in the form of an undirected graph, for instance, the complexity parameter might be the number n of vertices or the number m of edges.

Time complexities and complexity classes. Consider a problem in NP as defined above, with instances x and with complexity parameter $m(x)$. An algorithm for this problem has *sub-exponential* time complexity, if the running time depends polynomially on $|x|$ and if the logarithm of the running time depends sub-linearly on $m(x)$. For instance, a running time of $|x|^5 \cdot 2^{\sqrt{m(x)}}$ would be sub-exponential. A problem in NP is contained in the complexity class SUBEXP (the class of SUB-EXPonentially solvable problems) if for every fixed $\varepsilon > 0$, it can be solved in $\text{poly}(|x|) \cdot 2^{\varepsilon \cdot m(x)}$ time.

The complexity class SNP (the class Strict NP) was introduced by Papadimitriou & Yannakakis [32] for studying the approximability of optimization problems. SNP constitutes a subclass of NP, and it contains all problems that can be formulated in a certain way by a logical formula that starts with a series of second order existential quantifiers, followed by a series of first order universal quantifiers, followed by a first-order quantifier-free formula (a Boolean combination of input and quantifier relations applied to the quantified element variables). In this survey, the class SNP will only show up in Section 7. As far as this survey is concerned, all we need to know about SNP is that it is a fairly broad complexity class that contains many of the natural combinatorial optimization problems.

Downey & Fellows [7] introduced parameterized complexity theory for investigating the complexity of problems that involve a parameter. This parameter may for instance be the treewidth or the genus of an underlying graph, or an upper bound on the objective value, or in our case the complexity parameter $m(x)$. A whole theory has evolved around such parameterizations, and this has lead to the so-called W-hierarchy, an infinite hierarchy of complexity classes:

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[k] \subseteq \cdots \subseteq W[P].$$

We refer the reader to Downey & Fellows [8] for the exact definitions of all these classes. It is commonly believed that all W-classes are pairwise distinct, and that hence all displayed inclusions are strict.

Some classes of optimization problems. Let us briefly discuss some basic classes of optimization problems that contain many classical problems: the class of subset problems, the class of permutation problems, and the class of partition problems. In a *subset problem*, every feasible solution can be specified as a subset of an underlying ground set. For instance, fixing a truth-assignment in the satisfiability problem corresponds to selecting a subset of TRUE variables. In the independent set problem, every subset of the vertex set is a solution candidate. In a *permutation problem*, every feasible solution can be specified as a total ordering of an underlying ground set. For instance, in the TSP every tour corresponds to a permutation of the cities. In single machine scheduling problems, feasible schedules are often specified as permutations of the jobs. In a *partition problem*, every feasible solution can be specified as a partition of an underlying ground set. For instance, a graph coloring is a partition of the vertex set into color classes. In parallel machine scheduling problems, feasible schedules are often specified by partitioning the job set and assigning every part to another machine.

As we observed above, all NP-complete problems possess trivial algorithms that simply enumerate and check all feasible solutions. For a ground set with cardinality n, subset problems can be trivially solved in $O^*(2^n)$ time, permutation problems can be trivially solved in $O^*(n!)$ time, and partition problems are trivial to solve in $O^*(c^{n \log n})$ time; here $c > 1$ denotes a constant that does not depend on the instance. These time complexities form the triviality barriers for the corresponding classes of optimization problems.

More technical remarks. All optimization problems considered in this survey are known to be NP-complete. We refer the reader to the book [14] by Garey & Johnson for (references to) the NP-completeness proofs. We denote the base two logarithm of a real number z by $\log(z)$.

3 Technique: Dynamic programming across the subsets

A standard approach for getting fast exact algorithms for NP-complete problems is to do dynamic programming across the subsets. For every 'interesting' subset of the ground set, there is a polynomial number of corresponding states in the state space of the dynamic program. In the cases where all these corresponding states can be computed in reasonable time, this approach usually yields a time complexity of $O^*(2^n)$. We will illustrate these benefits of dynamic programming by developping algorithms for the travelling salesman problem and for total completion time scheduling on a single machine under precedence constraints. Sometimes, the number of 'interesting' subsets is fairly small, and then an even better time complexity might be possible. This will be illustrated by discussing the graph 3-colorability problem.

The travelling salesman problem (TSP). A travelling salesman has to visit the cities 1 to n. He starts in city 1, runs through the remaining $n-1$ cities in arbitrary order, and in the very end returns to his starting point in city 1. The distance from city i to city j is denoted by $d(i,j)$. The goal is to minimize the total travel length of the salesman. A trivial algorithm for the TSP checks all $O(n!)$ permutations.

We now sketch the exact TSP algorithm of Held & Karp [16] that is based on dynamic programming across the subsets. For every non-empty subset $S \subseteq \{2,\ldots,n\}$ and for every city $i \in S$, we denote by $\textsc{Opt}[S;i]$ the length of the shortest path that starts in city 1, then visits all cities in $S-\{i\}$ in arbitrary order, and finally stops in city i. Clearly, $\textsc{Opt}[\{i\};i] = d(1,i)$ and

$$\textsc{Opt}[S;i] \;=\; \min\{\textsc{Opt}[S-\{i\};j] + d(j,i) :\; j \in S-\{i\}\}\,.$$

By working through the subsets S in order of increasing cardinality, we can compute the value $\textsc{Opt}[S;i]$ in time proportional to $|S|$. The optimal travel length is given as the minimum value of $\textsc{Opt}[\{2,\ldots,n\};j] + d(j,1)$ over all j with $2 \le j \le n$. This yields an overall time complexity of $O(n^2 2^n)$ and hence $O^*(2^n)$.

This result was published in 1962, and from nowadays point of view almost looks trivial. Still, it yields the best time complexity that is known today.

Open problem 3.1 *Construct an exact algorithm for the travelling salesman problem with time complexity $O^*(c^n)$ for some $c < 2$. In fact, it even would be interesting to reach such a time complexity $O^*(c^n)$ with $c < 2$ for the closely related, but slightly simpler Hamiltonian cycle problem (given a graph G on n vertices, does it contain a spanning cycle).*

Hwang, Chang & Lee [19] describe a sub-exponential time $O(c^{\sqrt{n}\log n})$ exact algorithm with some constant $c > 1$ for the Euclidean TSP. The Euclidean TSP is a special case of the TSP where the cities are points in the Euclidean plane and where the distance between two cities is the Euclidean distance. The approach in [19] is heavily based on planar separator structures, and it cannot be carried over to the general TSP. The approach can be used to yield similar time bounds for various NP-complete geometric optimization problems, as the Euclidean p-center problem and the Euclidean p-median problem.

Total completion time scheduling under precedence constraints. There is a single machine, and there are n jobs $1,\ldots,n$ that are specified by their length p_j and their weight w_j $(j = 1,\ldots,n)$. Precedence constraints are given by a partial order on the jobs; if job i precedes job j in the partial order (denoted by $i \to j$), then i must be processed to completion before j can begin its processing. All jobs are available at time 0. We only consider non-preemptive schedules, in which all p_j time units of job J_j must be scheduled consecutively. The goal is to schedule the jobs on the single machine such that all precedence constraints are obeyed and such that the total completion time $\sum_{j=1}^{n} w_j C_j$ is minimized; here C_j is the time at which job j is completed in the given schedule. A trivial algorithm checks all $O(n!)$ permutations of the jobs.

Dynamic programming across the subsets yields a time complexity of $O^*(2^n)$. A subset $S \subseteq \{1,\ldots,n\}$ of the jobs is called an *ideal*, if $j \in S$ and $i \to j$ always implies $i \in S$. In other words, for every job $j \in S$ the ideal S also contains all jobs that have to be processed before j. For an ideal S, we denote by $\textsc{first}(S)$ all jobs in S without predecessors, by $\textsc{last}(S)$ all jobs in S without successors, and by $p(S) = \sum_{i \in S} p_i$ the total processing time of the jobs in S. For an ideal S, we denote by $\textsc{Opt}[S]$ the smallest possible total completion time for the jobs in S. Clearly, for any $j \in \textsc{first}(\{1,\ldots,n\})$ we have $\textsc{Opt}[\{j\}] = w_j p_j$. Moreover, for $|S| \geq 2$ we have

$$\textsc{Opt}[S] \;=\; \min \{\textsc{Opt}[S - \{j\}] + w_j\, p(S) :\; j \in \textsc{last}(S)\}\,.$$

This DP recurrence is justified by the observation that some job $j \in \textsc{last}(S)$ has to be processed last, and thus is completed at time $p(S)$. By working through the ideals S in order of increasing cardinality, we can compute all values $\textsc{Opt}[S]$ in time proportional to $|S|$. The optimal objective value can be read from $\textsc{Opt}[\{1,\ldots,n\}]$. This yields an overall time complexity of $O^*(2^n)$.

Similar approaches yield $O^*(c^n)$ time exact algorithms for many other single machine scheduling problems.

Exercise 3.2 *Use dynamic programming across the subsets to get exact algorithms with time complexity $O^*(2^n)$ for the following two scheduling problems.*

(a) Minimizing the weighted number of late jobs. There are n jobs that are specified by a length p_j, a penalty w_j, and a due date d_j. If a job j is completed after its due date d_j, one has to pay a penalty p_j. The goal is to sequence the jobs on a single machine such that the total penalty for the late jobs is minimized.

(b) Minimizing the total tardiness. There are n jobs that are specified by a length p_j and a due date d_j. If a job j is completed at time C_j in some fixed schedule, then its tardiness is $T_j = \max\{0, C_j - d_j\}$. The goal is to sequence the jobs on a single machine such that the total tardiness of the jobs is minimized.

Exercise 3.3 *Total completion time scheduling under precedence constraints and job release dates. That is the problem that we have solved above, but with the additional restriction that every job j cannot be processed before its release r_j. As a consequence, their might be gaps in the middle of the schedule where the machine is idle.*

Use dynamic programming across the subsets to get an exact algorithm with time complexity $O^*(3^n)$ for this problem.

Graph coloring. Given a graph $G = (V, E)$ with n vertices, color the vertices with the smallest possible number of colors such that adjacent vertices never receive the same color. This smallest possible number is the *chromatic* number $\chi(G)$ of the graph. Every color class is a vertex set without induced edges; such a vertex set is called an *independent* set. An independent set is *maximal*, if none of its proper supersets is also independent. For any graph G, there exists a feasible coloring with $\chi(G)$ colors in which at least one color class is a maximal independent set. Moon & Moser [29] have shown that a graph with n vertices

contains at most $3^{n/3} \approx 1.4422^n$ maximal independent sets. By considering a collection of $n/3$ independent triangles, we see that this bound is best possible. Paull & Unger [36] designed a procedure that generates all maximal independent sets in a graph in $O(n^2)$ time per generated set.

Based on the ideas introduced by Lawler [26], we present a dynamic program across the subsets with a time complexity of $O^*(2.4422^n)$. For a subset $S \subseteq V$ of the vertices, we denote by $G[S]$ the subgraph of G that is induced by the vertices in S, and we denote by $\textsc{Opt}[S]$ the chromatic number of $G[S]$. If S is empty, then clearly $\textsc{Opt}[S] = 0$. Moreover, for $S \neq \emptyset$ we have

$$\textsc{Opt}[S] \;=\; 1 + \min\{\textsc{Opt}[S-T] :\; T \text{ maximal indep. set in } G[S]\}\,.$$

We work through the sets S in order of increasing cardinality, such that when we are handling S, all its subsets have already been handled. Then the time needed to compute the value $\textsc{Opt}[S]$ is dominated by the time needed to generate all maximal independent subsets T of $G[S]$. By the above discussion, this can be done in $k^2 3^{k/3}$ time where k is the number of vertices in $G[S]$. This leads to an overall time complexity of

$$\sum_{k=0}^{n} \binom{n}{k} k^2 3^{k/3} \;\leq\; n^2 \sum_{k=0}^{n} \binom{n}{k} 3^{k/3} \;=\; n^2 (1+3^{1/3})^n.$$

Since $1 + 3^{1/3} \approx 2.4422$, this yields the claimed time complexity $O^*(2.4422^n)$. Very recently, Eppstein [11] managed to improve this time complexity to $O^*(2.4150^n)$ where $2.4150 \approx 4/3 + 3^{4/3}/4$. His improvement is based on carefully counting the *small* maximal independent sets in a graph.

Finally, we turn to the (much easier) special case of deciding whether $\chi(G) = 3$. Lawler [26] gives a simple $O^*(1.4422^n)$ algorithm: Generate all maximal independent sets S, and check whether their complement graph $G[V-S]$ is bipartite. Schiermeyer [42] describes a rather complicated modification of this idea that improves the time complexity to $O^*(1.415^n)$. The first major progress is due to Beigel & Eppstein [4] who get a running time of $O^*(1.3446^n)$ by applying the technique of pruning the search tree; see Section 4 of this survey. The current champion algorithm has a time complexity of $O^*(1.3289^n)$ and is due to Eppstein [10]. This algorithm combines pruning of the search tree with several tricks based on network flows and matching.

Exercise 3.4 *(Nielsen [30])*
Find an $O^*(1.7851^n)$ exact algorithm that decides for a graph on n vertices whether $\chi(G) = 4$. Hint: Generate all maximal independent sets of cardinality at least $n/4$ (why?), and use the algorithm from [10] to check their complement graphs.

Eppstein [10] also shows that for $n/4 \leq k \leq n/3$, a graph on n vertices contains at most $O(3^{4k-n}4^{n-3k})$ maximal independent sets. Apply this result to improve the time complexity for 4-coloring further to $O^*(1.7504^n)$.

Open problem 3.5 *Design fast algorithms for k-colorability where k is small, say for $k \leq 6$. Design faster exact algorithms for the general graph coloring problem. Can we reach running times around $O^*(2^n)$?*

4 Technique: Pruning the search tree

Every NP-complete problem can be solved by enumerating and checking all feasible solutions. An organized way for doing this is to (1) concentrate on some piece of the feasible solution, to (2) determine all the possible values this piece can take, and to (3) branch into several subcases according to these possible values. This naturally defines a search tree: Every branching in (3) corresponds to a branching of the search tree into subtrees. Sometimes, it can be argued that certain values for a certain piece can never lead to an optimal solution. In these cases we may simply ignore all these values, kill the corresponding subtrees, and speed-up the search procedure. Every Branch-and-Bound algorithm is based on this idea, and we can also get exact algorithms with good worst case behavior out of this idea. However, to get the worst case analysis through, we need a good mathematical understanding of the evolution of the search tree, and we need good estimates on the sizes of the killed subtrees and on the number and on the sizes of the surviving cases.

We will illustrate the technique of pruning the search tree by developping algorithms for the satisfiability problem, for the independent set problem in graphs, and for the bandwidth problem in graphs.

The satisfiability problem. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of logical variables. A variable or a negated variable from X is called a *literal*. A *clause* over X is the disjunction of literals from X. A Boolean formula is in *conjunctive normal form (CNF)*, if it is the conjunction of clauses over X. A formula in CNF is in *k-CNF*, if all clauses contain at most k literals. A formula is *satisfiable*, if there is a truth assignment from X to $\{0,1\}$ which assigns to each variable a Boolean value (0=false, 1=true) such that the entire formula evaluates to true. The k-satisfiability problem is the problem of deciding whether a formula F in k-CNF is satisfiable. It is well-known that 2-satisfiability is polynomially solvable, whereas k-satisfiability with $k \geq 3$ is NP-complete. A trivial algorithm checks all possible truth assignments in $O^*(2^n)$ time.

We will now describe an exact $O^*(1.8393^n)$ algorithm for 3-satisfiability that is based on the technique of pruning the search tree. Let F be a Boolean formula in 3-CNF with m clauses ($m \leq n^3$). The idea is to branch on one of the clauses c with three literals ℓ_1, ℓ_2, ℓ_3. Every satisfying truth assignment for F must fall into one of the following three classes:

(a) literal ℓ_1 is true;
(b) literal ℓ_1 is false, and literal ℓ_2 is true;
(c) literals ℓ_1 and ℓ_2 are false, and literal ℓ_3 is true.

We fix the values of the corresponding one, two, three variables appropriately, and we branch into three subtrees according to these cases (a), (b), and (c) with $n-1$, $n-2$, and $n-3$ unfixed variables, respectively. By doing this, we cut away the subtree where the literals ℓ_1, ℓ_2, ℓ_3 all are false. The formulas in the three subtrees are handled recursively. The stopping criterion is when we reach a formula in 2-CNF, which can be resolved in polynomial time. Denote by

$T(n)$ the worst case time that this algorithm needs on a 3-CNF formula with n variables. Then

$$T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(n+m).$$

Here the terms $T(n-1)$, $T(n-2)$, and $T(n-3)$ measure the time for solving the subcase with $n-1$, $n-2$, and $n-3$ unfixed variables, respectively. Standard calculations yield that $T(n)$ is within a polynomial factor of α^n where α is the largest real root of $\alpha^3 = \alpha^2+\alpha+1$. Since $\alpha \approx 1.8393$, this gives a time complexity of $O^*(1.8393^n)$.

In a milestone paper in this area, Monien & Speckenmeyer [28] improve the branching step of the above approach. They either detect a clause that can be handled without any branching, or they detect a clause for which the branching only creates formulas that contain one clause with at most $k-1$ literals. A careful analysis yields a time complexity of $O^*(\beta^n)$ for k-satisfiability, where β is the largest real root of $\beta = 2 - 1/\beta^{k-1}$. For 3-satisfiability, this time complexity is $O^*(1.6181^n)$. Schiermeyer [41] refines these ideas for 3-satisfiability even further, and performs a quantitative analysis of the number of 2-clauses in the resulting subtrees. This yields a time complexity of $O^*(1.5783^n)$. Kullmann [24, 25] writes half a book on the analysis of this approach, and gets time complexities of $O^*(1.5045^n)$ and $O^*(1.4963^n)$ for 3-satisfiability. The current champion algorithms for satisfiability are, however, not based on pruning the search tree, but on local search ideas; see Section 6 of this survey.

Exercise 4.1 *For a formula F in CNF, consider the following bipartite graph G_F: For every logical variable in X, there is a corresponding variable-vertex in G_F, and for every clause in F, there is a corresponding clause-vertex in G_F. There is an edge from a variable-vertex to a clause-vertex if and only if the corresponding variable is contained (in negated or un-negated form) in the corresponding clause. The planar satisfiability problem is the special case of the satisfiability problem that contains all instances with formulas F in CNF for which the graph G_F is planar.*

Design a sub-exponential time exact algorithm for the planar 3-satisfiability problem! Hint: Use the planar separator theorem of Lipton & Tarjan [27] to break the formula F into two smaller, independent pieces. Running times of roughly $O^*(c^{\sqrt{n}})$ are possible.

The independent set problem. Given a graph $G = (V, E)$ with n vertices, the goal is to find an independent set of maximum cardinality. An *independent* set $S \subseteq V$ is a set of vertices that does not induce any edges. Moon & Moser [29] have shown that a graph contains at most $3^{n/3} \approx 1.4422^n$ maximal (with respect to inclusion) independent sets. Hence the first goal is to beat the time complexity $O^*(1.4422^n)$.

We describe an exact $O^*(1.3803^n)$ algorithm for independent set that is based on the technique of pruning the search tree. Let G be a graph with m edges. The idea is to branch on a high-degree vertex: If all vertices have degree at most two, then the graph is a collection of cycles and paths. It is straightforward to

determine a maximum independent set in such a graph. Otherwise, G contains a vertex v of degree $d \geq 3$; let $v_1, \ldots, v_d$ be the neighbors of v in G. Every independent set I for G must fall into one of the following two classes:

(a) I does not contain v.
(b) I does contain v; then I cannot contain any neighbor of v.

We dive into two subtrees. The first subtree deals with the graph that results from removing vertex v from G. The second subtree deals with the graph that results from removing v together with $v_1, \ldots, v_d$ from G. We recursively compute the maximum independent set in both subtrees, and update it to a solution for the original graph G. Denote by $T(n)$ the worst case time that this algorithm needs on a graph with n vertices. Then

$$T(n) \leq T(n-1) + T(n-4) + O(n+m).$$

Standard calculations yield that $T(n)$ is within a polynomial factor of γ^n where $\gamma \approx 1.3803$ is the largest real root of $\gamma^4 = \gamma^3 + 1$. This yields the time complexity $O^*(1.3803^n)$.

The first published paper that deals with exact algorithms for maximum independent set is Tarjan & Trojanowski [46]. They give an algorithm with running time $O^*(1.2599^n)$. This algorithm follows essentially the above approach, but performs a smarter (and pretty tedious) structural case analysis of the neighborhood around the high-degree vertex v. The algorithm of Jian [22] has a time complexity of $O^*(1.2346^n)$. Robson [38] further refines the approach. A combinatorial argument about connected regular graphs helps to get the running time down to $O^*(1.2108^n)$. Robson's algorithm uses exponential space. Beigel [3] presents another algorithm with a weaker time complexity of $O^*(1.2227^n)$, but polynomial space complexity. Robson [39] is currently working on a new algorithm which is supposed to run in time $O^*(1.1844^n)$. This new algorithm is based on a detailed computer generated subcase analysis where the number of subcases is in the tens of thousands.

Open problem 4.2 *(a) Construct an exact algorithm for the maximum independent set problem with time complexity $O^*(c^n)$ for some $c \leq 1.1$. If this really is doable, it will be very tedious to do.*

(b) Prove a lower bound on the time complexity of any exact algorithm for maximum independent set that is based on the technique of pruning the search tree and that makes its branching decision by solely considering the subgraphs around a fixed chosen vertex.

Exercise 4.3 *(a) Design an algorithm with time complexity $O^*(1.1602^n)$ for the restriction of the maximum independent set problem to graphs with maximum degree three! Warning: This is not an easy exercise. See Chen, Kanj & Jia [5] for a solution.*

(b) Design a sub-exponential time exact algorithm for the restriction of the maximum independent set problem to planar graphs! Hint: Use the planar separator theorem of Lipton & Tarjan [27].

Open problem 4.4 *An input to the Max-Cut problem consists of a graph* $G = (V, E)$ *on* n *vertices. The goal is to find a partition of* V *into two sets* V_1 *and* V_2 *that maximizes the number of edges between* V_1 *and* V_2 *in* E.

(a) Design an exact algorithm for the Max-Cut problem with time complexity $O^*(c^n)$ *for some* $c < 2$.

(b) Design an exact algorithm for the restriction of the Max-Cut problem to graphs with maximum degree three that has a time complexity $O^*(c^n)$ *for some* $c < 1.5$. *Gramm & Niedermeier [15] state an algorithm with time complexity* $O^*(1.5160^n)$.

The bandwidth problem. Given a graph $G = (V, E)$ with n vertices, a *linear arrangement* is a bijective numbering $f : V \to \{1, \ldots, n\}$ of the vertices from 1 to n (which can be viewed as a layout of the graph vertices on a line). In some fixed linear arrangement, the *stretch* of an edge $[u, v] \in E$ is the distance $|f(u) - f(v)|$ of its endpoints, and the *bandwidth* of the linear arrangement is the maximum stretch over all edges. In the bandwidth problem, the goal is to find a linear arrangement of minimum bandwidth for G. A trivial algorithm checks all possible linear arrangements in $O^*(n!)$ time.

We will sketch an exact $O^*(20^n)$ algorithm for the bandwidth problem that is based on the technique of pruning the search tree. This beautiful algorithm is due to Feige & Kilian [13]. The algorithm checks for every integer b with $1 \leq b \leq n$ in $O^*(20^n)$ time whether the bandwidth of the input graph G is less or equal to b. To simplify the presentation, we assume that both n and b are powers of two (and otherwise analogous but more messy arguments go through). Moreover, we assume that G is connected. The algorithm proceeds in two phases. In the first phase, it generates an initial piece of the search tree that branches into up to 5^n subtrees. In the second phase, each of these subtrees is handled in $O(4^n)$ time per subtree.

The goal of the first phase is to break the set of 'reasonable' linear arrangements into up to $n\,5^{n-1}$ subsets; in each of these subsets the approximate position of every single vertex is known. More precisely, we partition the interval $[1, n]$ into $2n/b$ segments of length $b/2$, and we will assign every vertex to one of these segments. We start with an arbitrary vertex $v \in V$, and we check all $2n/b$ possibilities for assigning v to some segment. Then we iteratively select a yet unassigned vertex x that has a neighbor y that has already been assigned to some segment. In any linear arrangement with bandwidth b, vertex x can not be placed more than two segments away from vertex y; hence, vertex x can only be assigned to five possible segments. There are $n - 1$ vertices to assign, and we end up with $O^*(5^n)$ assignments.

In the second phase, we check which of these $O^*(5^n)$ assignments can be extended to a linear arrangement of bandwidth at most b. All assignments are handled in the same way: If an assignment stretches some edge from a segment to another segment with at least two other segments in between, then this assignment can never lead to a linear arrangement with bandwidth b; therefore, we may kill such an assignment right away. If an edge goes from a segment to the same segment or to one of the adjacent segments, then it will have stretch at

most b regardless of the exact positions of vertices in segments; therefore, such an edge may be removed. Hence, in the end we are only left with edges that either connect consecutive even numbered segments or consecutive odd numbered segments. The problem decomposes into two independent subproblems, one within the even numbered segments and one within the odd numbered segments.

All these subproblems now are solved recursively. We break every segment into two subsegments of equal length. We try all possibilities for assigning every vertex from every segment into the corresponding left and right subsegments. Some of these refined assignments can be killed right away since they overstretch some edge; other edges are automatically short, and hence can be removed. In any case, we end up with two independent subproblems (one within the right subsegments and one within the left subsegments) that both can be solved recursively. Denote by $T(k)$ the time needed for solving a subproblem with k vertices. Then

$$T(k) \leq 2^k \cdot (T(k/2) + T(k/2)).$$

Standard calculations yield that $T(k)$ is in $O(4^k)$. Therefore, in the second phase we check $O^*(5^n)$ assignments in $O^*(4^n)$ time per assignment. This yields an overall time complexity of $O^*(20^n)$. Feige & Kilian [13] do a more careful analysis and improve the time complexity below $O^*(10^n)$.

Open problem 4.5 *(Feige & Kilian [13])*
Does the bandwidth problem have considerably faster exact algorithms? For instance, can it be solved in $O^*(2^n)$ time?

Exercise 4.6 *In the minimum sum linear arrangement problem, the input is a graph $G = (V, E)$ with n vertices. The goal is to find a linear arrangement of G that minimizes the sum of the stretches of all edges. Design an exact algorithm with time complexity $O^*(2^n)$ for this problem. Hint: Do not use the technique of pruning the search tree.*

5 Technique: Preprocessing the data

Preprocessing is an initial phase of computation, where one analyzes and restructures the given data, such that later on certain queries to the data can be answered quickly. By preprocessing an exponentially large data set or part of this data in an appropriate way, we may sometimes gain an exponentially large factor in the running time. In this section we will use the technique of preprocessing the data to get fast algorithms for the subset sum problem and for the binary knapsack problem. We start this section by discussing two very simple, polynomially solvable toy problems where preprocessing helps a lot.

In the first toy problem, we are given two integer sequences $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ and an integer S. We want to decide whether there exist an x_i and a y_j that sum up to S. A trivial approach would be to check all possible pairs in $O(k^2)$ overall time. A better approach is to first preprocess the data and to sort

the x_i in $O(k \log k)$ time. After that, we may repeatedly use bisection search in this sorted array, and search for the k values $S - y_j$ in $O(\log k)$ time per value. The overall time complexity becomes $O(k \log k)$, and we save a factor of $k/\log k$. By applying the same preprocessing, we can also decide in $O(k \log k)$ time, whether the sequences $\langle x_i \rangle$ and $\langle y_j \rangle$ are disjoint, or whether every value x_i also occurs in the sequence $\langle y_j \rangle$.

In the second toy problem, we are given k points (x_i, y_i) in two-dimensional space, together with the n numbers $z_1, \ldots, z_k$, and a number W. The goal is to determine for every z_j the largest value y_i, subject to the condition that $x_i + z_j \leq W$. The trivial solution needs $O(k^2)$ time, and by applying preprocessing this can be brought down to $O(k \log k)$: If there are two points (x_i, y_i) and (x_j, y_j) with $x_i \leq x_j$ and $y_i \geq y_j$, then the point (x_j, y_j) may be disregarded since it is always dominated by (x_i, y_i). The subset of non-dominated points can be computed in $O(k \log k)$ time by standard methods from computational geometry. We sort the non-dominated points by increasing x-coordinates and store this sequence in an array. This completes the preprocessing. To handle a value z_j, we simply search in $O(\log k)$ time through the sorted array for the largest value x_i less or equal to $W - z_j$.

In both toy problems preprocessing improved the time complexity from $O(k^2)$ to $O(k \log k)$. Of course, when dealing with exponential time algorithms an improvement by a factor of $k/\log k$ is not impressive at all. The right intuition is to think of k as roughly $2^{n/2}$. Then preprocessing the data yields a speedup from $k^2 = 2^n$ to $k \log k = n 2^{n/2}$, and such a speedup of $2^{n/2}$ indeed *is* impressive!

The subset sum problem. In this problem, the input consists of positive integers $a_1, \ldots, a_n$ and S. The question is whether there exists a subset of the a_i that sums up to S. The subset sum problem belongs to the class of subset problems, and can be solved (trivially) in $O^*(2^n)$ time. By splitting the problem into two halves and by preprocessing the first half, the time complexity can be brought down to $O^*(\sqrt{2}^n) \approx O^*(1.4145^n)$.

Let X denote the set of all integers of the form $\sum_{i \in I} a_i$ with $I \subseteq \{1, \ldots, \lfloor n/2 \rfloor\}$, and let Y denote the set of all integers of the form $\sum_{i \in I} a_i$ with $I \subseteq \{\lfloor n/2 \rfloor + 1, \ldots, n\}$. Note that $0 \in X$ and $0 \in Y$. It is straightforward to compute X and Y in $O^*(2^{n/2})$ time by complete enumeration. The subset sum instance has a solution if and only if there exists an $x_i \in X$ and a $y_j \in Y$ with $x_i + y_j = S$. But now we are back at our first toy problem that we discussed at the beginning of this section! By preprocessing X and by searching for all $S - y_j$ in the sorted structure, we can solve this problem in $O(n 2^{n/2})$ time. This yields an overall time of $O^*(2^{n/2})$.

Exercise 5.1 *An input to the Exact-Hitting-Set problem consists of a ground set X with n elements, and a collection $\mathcal{S}$ of subsets over X. The question is whether there exists a subset $Y \subseteq X$ such that $|Y \cap T| = 1$ for all $T \in \mathcal{S}$.*

Use the technique of preprocessing the data to get an exact algorithm with time complexity $O^*(2^{n/2}) \approx O^*(1.4145^n)$.

Drori & Peleg [9] use the technique of pruning the search tree to get a time complexity of $O^*(1.2494^n)$ for the Exact-Hitting-Set problem.

Exercise 5.2 *(Van Vliet [47])*
In the Three-Partition problem, the input consists of $3n$ positive integers $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, and $c_1, \ldots, c_n$, together with an integer D. The question is to determine whether there exist three permutations π, ψ, ϕ of $\{1, \ldots, n\}$ such that $a_{\pi(i)} + b_{\psi(i)} + c_{\phi(i)} = D$ holds for all $i = 1, \ldots, n$. By checking all possible triples (π, ψ, ϕ) of permutations, this problem can be solved trivially in $O^*(n!^3)$ time.

Use the technique of preprocessing the data to improve the time complexity to $O^*(n!)$.

The binary knapsack problem. Here the input consists of n items that are specified by a positive integer value a_i and a positive integer weight w_i $(i = 1, \ldots, n)$, together with a bound W. The goal is to find a subset of the items with the maximum total value subject to the condition that the total weight does not exceed W. The binary knapsack problem is closely related to the subset sum problem, and it can be solved (trivially) in $O^*(2^n)$ time. In 1974, Horowitz & Sahni [18] used a preprocessing trick to improve the time complexity to $O^*(2^{n/2})$.

For every $I \subseteq \{1, \ldots, \lfloor n/2 \rfloor\}$ we create a compound item x_I with value $a_I = \sum_{i \in I} a_i$ and weight $w_I = \sum_{i \in I} w_i$, and we put this item into the set X. For every $J \subseteq \{\lfloor n/2 \rfloor + 1, \ldots, n\}$ we put a corresponding compound item y_J into the set Y. The sets X and Y can be determined in $O^*(2^{n/2})$ time. The solution of the knapsack instance now reduces to the following: Find a compound item x_I in X and a compound item y_J in Y, such that $w_I + w_J \leq W$ and such that $a_I + a_J$ becomes maximum. But this can be handled by preprocessing as in our second toy problem, and we end up with an overall time complexity and an overall space complexity of $O^*(2^{n/2})$.

In 1981, Schroeppel & Shamir [45] improved the *space* complexity of this approach to $O^*(2^{n/4})$, while leaving its time complexity unchanged. The main trick is to split the instance into four pieces with $n/4$ items each, instead of two pieces with $n/2$ items. Apart from this, there has been no progress on exact algorithms for the knapsack problem since 1974.

Open problem 5.3 *Construct an exact algorithm for the subset sum problem or the knapsack problem with time complexity $O^*(c^n)$ for some $c < \sqrt{2}$, or prove that no such algorithm can exist under some reasonable complexity assumptions.*

6 Technique: Local search

The idea of using local search methods in designing exact exponential time algorithms is relatively new. A *local search* algorithm is a search algorithm that wanders through the space of feasible solutions. At each step, this search algorithm moves from one feasible solution to another one nearby. In order to express

the word 'nearby' mathematically, we need some notion of distance or neighborhood on the space of feasible solutions. For instance in the satisfiability problem, the feasible solutions are the truth assignments from the set X of logical variables to $\{0,1\}$. A natural distance between truth assignments is the *Hamming* distance, that is, the number of bits where two truth assignments differ.

In this section we will concentrate on the 3-satisfiability problem where the input is a Boolean formula F in 3-CNF over the n logical variables in $X = \{x_1, x_2, \ldots, x_n\}$; see Section 4 for definitions and notations for this problem. We will describe three exact algorithms for 3-satisfiability that all are based on local search ideas. All three algorithms are centered around the Hamming neighborhood of truth assignments: For a truth assignment t and a non-negative integer d, we denote by $\mathcal{H}(t,d)$ the set of all truth assignments that have Hamming distance at most d from assignment t. It is easy to see that $\mathcal{H}(t,d)$ contains exactly $\sum_{k=0}^{d} \binom{n}{k}$ elements.

Exercise 6.1 *For a given truth assignment t and a given non-negative integer d, use the technique of pruning the search tree to check in $O^*(3^d)$ time whether the Hamming neighborhood $\mathcal{H}(t,d)$ contains a satisfying truth assignment for the 3-CNF formula F.*

In other words, the Hamming neighborhood $\mathcal{H}(t,d)$ can be searched quickly for the 3-satisfiability problem. For the k-satisfiability problem, the corresponding time complexity would be $O^*(k^d)$.

First local search approach to 3-satisfiability. We denote by 0^n (respectively, 1^n) the truth assignment that sets all variables to 0 (respectively, to 1). Any truth assignment is in $\mathcal{H}(0^n, n/2)$ or in $\mathcal{H}(1^n, n/2)$. Therefore by applying the search algorithm from Exercise 6.1 twice, we get an exact algorithm with running time $O^*(\sqrt{3}^n) \approx O^*(1.7321^n)$ for 3-satisfiability. It is debatable whether this algorithm should be classified under pruning the search tree or under local search. In any case, it is due to Schöning [44].

Second local search approach to 3-satisfiability. In the first approach, we essentially covered the whole solution space by two balls of radius $d = n/2$ centered at 0^n and 1^n. The second approach works with balls of radius $d = n/4$. The crucial idea is to *randomly* choose the center of a ball, and to search this ball with the algorithm from Exercise 6.1. If we only do this once, then we ignore most of the solution space, and the probability for answering correctly is pretty small. But by repeating this procedure a huge number α of times, we can boost the probability arbitrarily close to 1. A good choice for α is $\alpha = 100 \cdot 2^n / \sum_{k=0}^{n/4} \binom{n}{k}$. The algorithm now works as follows: Choose α times a truth assignment t uniformly at random, and search for a satisfying truth assignment in $\mathcal{H}(t, n/4)$. If in the end no satisfying truth assignment has been found, then answer that the formula F is not satisfiable.

We will now discuss the running time and the error probability of this algorithm. By Exercise 6.1, the running time can be bounded by roughly $\alpha \cdot 3^{n/4}$. By

applying Stirling's approximation, one can show that up to a polynomial factor the expression $\sum_{k=0}^{n/4} \binom{n}{k}$ behaves asymptotically like $(256/27)^{n/4}$. Therefore, the upper bound $\alpha \cdot 3^{n/4}$ on the running time is in $O^*((3/2)^n) = O^*(1.5^n)$.

Now let us analyze the error probability of the algorithm. The only possible error occurs, if the formula F is satisfiable, whereas the algorithm does not manage to find a good ball $\mathcal{H}(t, n/4)$ that contains some satisfying truth assignment for F. For a single ball, the probability of containing a satisfying truth assignment equals $\sum_{k=0}^{n/4} \binom{n}{k}/2^n$, that is the number of elements in $\mathcal{H}(t, n/4)$ divided by the overall number of possible truth assignments. This probability equals $100/\alpha$. Therefore the probability of selecting a ball that does *not* contain any satisfying truth assignment is $1 - 100/\alpha$. The probability of α times *not* selecting such a ball equals $(1 - 100/\alpha)^\alpha$, which is bounded by the negligible value e^{-100}.

In fact, the whole algorithm can be derandomized without substantially increasing the running time. Dantsin, Goerdt, Hirsch, Kannan, Kleinberg, Papadimitriou, Raghavan & Schöning [6] do not choose the centers of the balls at random, but they take all centers from a so-called *covering code* so that the resulting balls cover the whole solution space. They show that such covering codes can be computed within reasonable amounts of time. The approach in [6] yields deterministic exact algorithms for k-satisfiability with running time $O^*((2-\frac{2}{k+1})^n)$. For 3-satisfiability, [6] improve the time complexity further down to $O^*(1.4802^n)$ by using a smart idea for an underlying branching step. This is currently the fastest known deterministic algorithm for 3-satisfiability.

Third local search approach to 3-satisfiability. The first approach was based on selecting the center of a ball deterministically, and then searching through the whole ball. The second approach was based on selecting the center of a ball randomly, and then searching through the whole ball. The third approach now is based on selecting the center of a ball randomly, and then doing a short random walk within the ball. More precisely, the algorithm repeats the following procedure roughly $200 \cdot (4/3)^n$ times: Choose a truth assignment t uniformly at random, and perform $2n$ steps of a random walk starting in t. In each step, first select a violated clause at random, then select a literal in the selected clause at random, and finally flip the truth value of the corresponding variable. If in the very end no satisfying truth assignment has been found, then answer that the formula F is not satisfiable.

The intuition behind this algorithm is as follows. If we start far away from a satisfying truth assignment, then the random walk has little chance of stumbling towards a satisfying truth assignment. Hence, it is a good idea to terminate it quite early after $2n$ steps, without wasting time. But if the starting point is very close to a satisfying truth assignment, then the probability is high that the random walk will be dragged closer and closer towards this satisfying truth assignment. And if the random walk indeed is dragged into a satisfying truth assignment, then with high probability this happens within the first $2n$ steps of the random walk. The underlying mathematical structure is a Markov chain that can be analyzed by standard methods. Clearly, the error probability can

be made negligibly small by sufficiently often restarting the random walk. And up to a polynomial factor, the running time of the algorithm is proportional to the number of performed random walks. This implies that the time complexity is $O^*((4/3)^n) \approx O^*(1.3334^n)$.

This algorithm and its analysis are due to Schöning [43]. Some of the underlying ideas go back to Papadimitriou [31] who showed that 2-SAT can be solved in polynomial time by a randomized local search procedure. The algorithm easily generalizes to the k-satisfiability problem, and yields a randomized exact algorithm with time complexity $O^*((2(k-1)/k)^n)$. The fastest known randomized exact algorithm for 3-satisfiability is due to Hofmeister, Schöning, Schuler & Watanabe [17], and has a running time of $O^*(1.3302^n)$. It is based on a refinement of the above random walk algorithm.

Open problem 6.2 *Design better deterministic and/or randomized algorithms for the k-satisfiability problem.*

More resuls on exact algorithms for k-satisfiability and related problems can be found in the work of Paturi, Pudlak & Zane [34], Paturi, Pudlak, Saks & Zane [35], Pudlak [37], Rodošek [40], and Williams [48].

7 How can we prove that a problem has no sub-exponential time exact algorithm?

All the problems discussed in this paper are NP-complete, and almost all of the developped algorithms use exponential time. Of course we cannot expect to find polynomial time algorithms for NP-complete problems, but maybe there exist better, sub-exponential, super-polynomial algorithms? How can we settle such questions?

Since our understanding of the landscape around the complexity classes P and NP still is fairly poor, the only available way of proving negative results on exact algorithms is by arguing relative to some widely believed conjectures. For instance, an NP-hardness proof establishes that some problem does not have a polynomial time algorithm, *given that the widely believed conjecture P≠NP holds true.* The right conjecture for disproving the existence of sub-exponential time exact algorithms seems to be the following.

Widely believed conjecture 7.1 *SNP $\not\subseteq$ SUBEXP.*

We already mentioned in Section 2 that the class SNP is a broad complexity class that contains many important combinatorial optimization problems. Therefore, if the widely believed Conjecture 7.1 is false, then quite unexpectedly all these important problems would possess relatively fast, sub-exponential time algorithms. However, the exact relationship between the P versus NP question and Conjecture 7.1 is unclear.

Open problem 7.2 *Does SNP $\subseteq$ SUBEXP imply P = NP?*

Impagliazzo, Paturi & Zane [21] introduce the concept of *SERF-reduction* (Sub-Exponential Reduction Family) that preserves sub-exponential time complexities. Consider two problems A_1 and A_2 in NP with complexity parameters m_1 and m_2, respectively. A SERF-reduction from A_1 to A_2 is a family T_ε of Turing-reductions from A_1 to A_2 over all $\varepsilon > 0$ with the following two properties:

- The reduction $T_\varepsilon(x)$ can be done in time $\text{poly}(|x|) \cdot 2^{\varepsilon \cdot m_1(x)}$.
- If the reduction $T_\varepsilon(x)$ queries A_2 with input x', then $m_2(x')$ is linearly bounded in $m_1(x)$ and the length of x' is polynomially bounded in the length of x.

SERF-reducibility is transitive. Moreover, if problem A_1 is SERF-reducible to problem A_2 and if problem A_2 has a sub-exponential time algorithm, then also problem A_1 has a sub-exponential time algorithm. Consider some problem A that is hard for the complexity class SNP under SERF-reductions. If problem A had a sub-exponential time algorithm, then *all* the problems in SNP had sub-exponential time algorithms, and this would contradict the widely believed Conjecture 7.1 that SNP $\not\subseteq$ SUBEXP.

The k-satisfiability problem plays a central role for sub-exponential time algorithms, the same central role that it plays everywhere else in computational complexity theory. There are two natural complexity parameters for k-satisfiability, the number of logical variables and the number of clauses. Impagliazzo, Paturi & Zane [21] prove that the two variants of k-satisfiability with these two complexity parameters are SERF-reducible to each other, and hence are equivalent under SERF-reductions. This indicates that for k-satisfiability the exact parameterization is not very important, and that all natural parameterizations of k-satisfiability should be SERF-reducible to each other. Most important, the paper [21] shows that for any fixed $k \geq 3$ the k-satisfiability problem is SNP-complete under SERF-reductions. As we discussed above, this implies that for any fixed $k \geq 3$ the k-satisfiability problem cannot have a sub-exponential time algorithm, unless SNP $\subseteq$ SUBEXP. Therefore, the widely believed Conjecture 7.1 could also be formulated in the following way.

Widely believed conjecture 7.3 *(Exponential Time Hypothesis, ETH)*
For any fixed $k \geq 3$, k-satisfiability does not have a sub-exponential time algorithm.

Now let s_k denote the infimum of all real numbers δ with the property that there exists an $O^*(2^{\delta n})$ exact algorithm for solving the k-satisfiability problem. Observe that $s_k \leq s_{k+1}$ and $0 \leq s_k \leq 1$ hold trivially for all $k \geq 3$. The exponential time hypothesis conjectures $s_k > 0$ for all $k \geq 3$, and that the numbers s_k converge to some limit $s_\infty > 0$. Impagliazzo & Paturi [20] prove that under ETH, $s_k \leq (1 - \alpha/k) \cdot s_\infty$ holds, where α is some small positive constant. Consequently, under ETH we can never have $s_k = s_\infty$ and the time complexities for k-satisfiability must increase more and more as k increases.

Open problem 7.4 *(Impagliazzo & Paturi [20])*
Assuming the exponential time hypothesis for k-satisfiability, obtain evidence for the hypothesis that $s_\infty = 1$.

Now let us discuss the behavior of some other problems in NP. Impagliazzo, Paturi & Zane [21] show that for any fixed $k \geq 3$ the k-colorability problem is SNP-complete under SERF-reductions. Hence 3-colorability can not be solved in sub-exponential time, unless SNP $\subseteq$ SUBEXP. The paper [21] also shows that the Hamiltonian cycle problem and the independent set problem (both with the number of vertices as complexity parameter) can not be solved in sub-exponential time, unless SNP $\subseteq$ SUBEXP. Johnson & Szegedy [23] strengthen the result on the independent set problem by showing that the independent set problem in arbitrary graphs is equally difficult as in graphs with maximum degree three: Either both of these problems have a sub-exponential time algorithm, or neither of them does. Feige & Kilian [13] prove that also the bandwidth problem can not be solved in sub-exponential time, unless SNP $\subseteq$ SUBEXP. For all results listed in this paragraph, the proofs are done by translating classical NP-hardness proofs from the 1970s into SERF-reductions. The main technical problem is to keep the complexity parameters $m(x)$ under control.

In another line of research, Feige & Kilian [12] show that if in graphs with n vertices independent sets of size $O(\log n)$ can be found in polynomial time, then the 3-satisfiability problem can be solved in sub-exponential time. This result probably does not speak against the ETH, but indicates that finding small independent sets is difficult.

The W-hierarchy gives rise to yet another widely believed conjecture that can be used for disproving the existence of sub-exponential time exact algorithms. As we already mentioned in Section 2, the general belief is that all the W-classes are pairwise distinct. The following (cautious) conjecture only states that the W-hierarchy does not collapse completely.

Widely believed conjecture 7.5 *$FPT \neq W[P]$.*

Abrahamson, Downey & Fellows [1] proved that Conjecture 7.5 is false if and only if the satisfiability problem for Boolean circuits can be solved in sub-exponential time $\text{poly}(|x|) \cdot 2^{o(n)}$. Here $|x|$ denotes the size of the Boolean circuit that is given as an input, n denotes the number of input variables of the circuit, and $o(n)$ denotes some sub-linear function in n. Since the k-satisfiability problem is a special case of the Boolean circuit satisfiability problem, the exponential time hypothesis ETH implies Conjecture 7.5. It is not known whether the reverse implication also holds.

Most optimization problems that are mentioned in this survey possess exact algorithms with time complexity $O^*(c^{m(x)})$, i.e., exponential time where the exponent grows linearly in the complexity parameter $m(x)$. The quadratic assignment problem is a candidate for a natural problem that does not possess such an exact algorithm.

Open problem 7.6 *In the quadratic assignment problem (QAP) the input consists of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ $(1 \leq i, j \leq n)$ with real*

entries. The objective is to find a permutation π that minimizes the cost function $\sum_{i=1}^{n} \sum_{j=1}^{n} a_{\pi(i)\pi(j)} b_{ij}$. The QAP can be solved in $O^*(n!)$ time. The QAP is a notoriously hard problem, and no essentially faster algorithms are known (Pardalos, Rendl & Wolkowicz [33]).

Prove that (under some reasonable complexity assumptions) the QAP can not be solved in $O^*(c^n)$ time, for any fixed value c.

8 Concluding remarks

Currently, when we are dealing with an optimization problem, we are used to look at its computational complexity, its approximability behavior, its online behavior (with respect to competitive analysis), its polyhedral structure. Exact algorithms with good worst case behavior should probably become another standard item on this list, and we feel that the known techniques and results as described in Sections 3–6 deserve to be taught in our introductory algorithms courses.

There remain many open problems and challenging questions around the worst case analysis of exact algorithms for NP-hard problems. This seems to be a rich and promising area. We only have a handful of techniques available, and there is ample space for improvements and for new results.

Acknowledgement. I thank David Eppstein, Jesper Makholm Nielsen, and Ryan Williams for several helpful comments on preliminary versions of this paper, and for providing some pointers to the literature. Furthermore, I thank an unknown referee for many suggestions how to improve the structure, the English, the style, and the contents of this paper.

References

1. K.A. Abrahamson, R.G. Downey, and M.R. Fellows [1995]. Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE analogues. *Annals of Pure and Applied Logic 73*, 235–276.
2. D. Applegate, R. Bixby, V. Chvátal, and W. Cook [1998]. On the solution of travelling salesman problems. *Documenta Mathematica 3*, 645–656.
3. R. Beigel [1999]. Finding maximum independent sets in sparse and general graphs. *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA'1999)*, 856–857.
4. R. Beigel and D. Eppstein [1995]. 3-Coloring in time $O(1.3446^n)$: A no-MIS algorithm. *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'1995)*, 444–453.
5. J. Chen, I.A. Kanj, and W. Jia [1999]. Vertex cover: Further observations and further improvements. *Proceedings of the 25th Workshop on Graph Theoretic Concepts in Computer Science (WG'1999)*, Springer, LNCS 1665, 313–324.
6. E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C.H. Papadimitriou, P. Raghavan, and U. Schöning [2001]. A deterministic $(2-\frac{2}{k+1})^n$ algorithm for k-SAT based on local search. To appear in *Theoretical Computer Science.*

7. R.G. Downey and M.R. Fellows [1992]. Fixed parameter intractability. *Proceedings of the 7th Annual IEEE Conference on Structure in Complexity Theory (SCT'1992)*, 36–49.
8. R.G. Downey and M.R. Fellows [1999]. *Parameterized complexity.* Springer Monographs in Computer Science.
9. L. Drori and D. Peleg [1999]. Faster exact solutions for some NP-hard problems. *Proceedings of the 7th European Symposium on Algorithms (ESA'1999)*, Springer, LNCS 1643, 450–461.
10. D. Eppstein [2001]. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA'2001)*, 329–337.
11. D. Eppstein [2001]. Small maximal independent sets and faster exact graph coloring. *Proceedings of the 7th Workshop on Algorithms and Data Structures (WADS'2001)*, Springer, LNCS 2125, 462–470.
12. U. Feige and J. Kilian [1997]. On limited versus polynomial nondeterminism. *Chicago Journal of Theoretical Computer Science* (`http://cjtcs.cs.uchicago.edu/`).
13. U. Feige and J. Kilian [2000]. Exponential time algorithms for computing the bandwidth of a graph. Manuscript.
14. M.R. Garey and D.S. Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco.
15. J. Gramm and R. Niedermeier [2000]. Faster exact solutions for Max2Sat. *Proceedings of the 4th Italian Conference on Algorithms and Complexity (CIAC'2000)*, Springer, LNCS 1767, 174–186.
16. M. Held and R.M. Karp [1962]. A dynamic programming approach to sequencing problems. *Journal of SIAM 10*, 196–210.
17. T. Hofmeister, U. Schöning, R. Schuler, and O. Watanabe [2001]. A probabilistic 3-SAT algorithm further improved. Manuscript.
18. E. Horowitz and S. Sahni [1974]. Computing partitions with applications to the knapsack problem. *Journal of the ACM 21*, 277–292.
19. R.Z. Hwang, R.C. Chang, and R.C.T. Lee [1993]. The searching over separators strategy to solve some NP-hard problems in subexponential time. *Algorithmica 9*, 398–423.
20. R. Impagliazzo and R. Paturi [2001]. Complexity of k-SAT. *Journal of Computer and System Sciences 62*, 367–375.
21. R. Impagliazzo, R. Paturi, and F. Zane [1998]. Which problems have strongly exponential complexity? *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS'1998)*, 653–663.
22. T. Jian [1986]. An $O(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Transactions on Computers 35*, 847–851.
23. D.S. Johnson and M. Szegedy [1999]. What are the least tractable instances of max independent set? *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA'1999)*, 927–928.
24. O. Kullmann [1997]. Worst-case analysis, 3-SAT decisions, and lower bounds: Approaches for improved SAT algorithms. In: *The Satisfiability Problem: Theory and Applications*, D. Du, J. Gu, P.M. Pardalos (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science 35, 261–313.
25. O. Kullmann [1999]. New methods for 3-SAT decision and worst case analysis. *Theoretical Computer Science 223*, 1–72.
26. E.L. Lawler [1976]. A note on the complexity of the chromatic number problem. *Information Processing Letters 5*, 66–67.

27. R.J. Lipton and R.E. Tarjan [1979]. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics 36*, 177–189.
28. B. Monien and E. Speckenmeyer [1985]. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics 10*, 287–295.
29. J.W. Moon and L. Moser [1965]. On cliques in graphs. *Israel Journal of Mathematics 3*, 23–28.
30. J.M. Nielsen [2001]. Personal communication.
31. C.H. Papadimitriou [1991]. On selecting a satisfying truth assignment. *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS'1991)*, 163–169.
32. C.H. Papadimitriou and M. Yannakakis [1991]. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences 43*, 425–440.
33. P. Pardalos, F. Rendl, and H. Wolkowicz [1994]. The quadratic assignment problem: A survey and recent developments. In: *Proceedings of the DIMACS Workshop on Quadratic Assignment Problems*, P. Pardalos and H. Wolkowicz (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science 16, 1–42.
34. R. Paturi, P. Pudlak, and F. Zane [1997]. Satisfiability coding lemma. *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS'1997)*, 566–574.
35. R. Paturi, P. Pudlak, M.E. Saks, and F. Zane [1998]. An improved exponential time algorithm for k-SAT. *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS'1998)*, 628–637.
36. M. Paull and S. Unger [1959]. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers 8*, 356–367.
37. P. Pudlak [1998]. Satisfiability – algorithms and logic. *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'1998)*, Springer, LNCS 1450, 129–141.
38. J.M. Robson [1986]. Algorithms for maximum independent sets. *Journal of Algorithms 7*, 425–440.
39. J.M. Robson [2001]. Finding a maximum independent set in time $O(2^{n/4})$? Manuscript.
40. R. Rodošek [1996]. A new approach on solving 3-satisfiability. *Proceedings of the 3rd International Conference on Artificial Intelligence and Symbolic Mathematical Computation* Springer, LNCS 1138, 197–212.
41. I. Schiermeyer [1992]. Solving 3-satisfiability in less than $O(1.579^n)$ steps. *Selected papers from Computer Science Logic (CSL'1992)*, Springer, LNCS 702, 379–394.
42. I. Schiermeyer [1993]. Deciding 3-colorability in less than $O(1.415^n)$ steps. *Proceedings of the 19th Workshop on Graph Theoretic Concepts in Computer Science (WG'1993)*, Springer, LNCS 790, 177–182.
43. U. Schöning [1999]. A probabilistic algorithm for k-SAT and constraint satisfaction problems. *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'1999)*, 410–414.
44. U. Schöning [2001]. New algorithms for k-SAT based on the local search principle. *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS'2001)*, Springer, LNCS 2136, 87–95.
45. R. Schroeppel and A. Shamir [1981]. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing 10*, 456–464.

46. R.E. Tarjan and A.E. Trojanowski [1977]. Finding a maximum independent set. *SIAM Journal on Computing 6*, 537–546.
47. A. van Vliet [1995]. Personal communication.
48. R. Williams [2002]. Algorithms for quantified Boolean formulas. *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA'2002)*.