Space and Time Complexity of Exact Algorithms: Some Open Problems

Gerhard J. Woeginger

TU Eindhoven, The Netherlands
gwoegi@win.tex.nl

Abstract. We discuss open questions around worst case time and space bounds for NP-hard problems. We are interested in exponential time solutions for these problems with a relatively good worst case behavior.

1 Introduction

Every problem in NP can be solved in exponential time by exhaustive search: Recall that a decision problem is in NP, if and only if there exists a polynomial time decidable relation $R(x, y)$ and a polynomial $m(|x|)$ such that for every YES-instance x, there exists a YES-certificate y with $|y| \leq m(x)$ and $R(x, y)$. A trivial exact algorithm for solving instance x enumerates all possible strings y with lengths up to $m(|x|)$, and checks whether any of them yields a YES-certificate. Up to polynomial factors that depend on the evaluation time of $R(x, y)$, this yields an exponential running time of $2^{m(x)}$.

A natural question is: Can we do better than this trivial enumerative algorithm? Interestingly, for many combinatorial optimization problems the answer is YES. Early examples include an $O^*(1.4422^n)$ algorithm for deciding 3-colorability of an n-vertex graph by Lawler [21]; an $O^*(1.2599^n)$ algorithm for finding a maximum independent set in an n-vertex graph by Tarjan & Trojanowski [24]; an $O^*(1.4142^n)$ algorithm for the SUBSET-SUM problems with n integers by Horowitz & Sahni [18]. (The notation $O^*(f(n))$ is explained at the end of this section.) Woeginger [26] surveys many results in this area.

For some optimization problems, we can reach an improved time complexity, but it seems that we have to pay for this with an *exponential* space complexity. Note that algorithms with exponential space complexities are absolutely useless for real life applications. In this paper, we discuss a number of results around fast exponential time algorithms that come with exponential space complexities. We present approaches, tricks, related polynomially solvable problems, and related open questions.

Notation. Throughout this paper, we will use a modified big-Oh notation that suppresses polynomially bounded terms. For a positive real constant c, we write $O^*(c^n)$ for a time complexity of the form $O(c^n \cdot \text{poly}(n))$. The notations $\Omega^*(c^n)$ and $\Theta^*(c^n)$ are defined analogously.

R. Downey, M. Fellows, and F. Dehne (Eds.): IWPEC 2004, LNCS 3162, pp. 281–290, 2004.

2 Integers and Their Sums

We start this section with a couple of polynomially solvable problems: An input to the first problem "k-SUM" consists of m integers $a_1, \ldots, a_m$ and a goal sum S. The problem is to decide whether there are k of these integers that add up to S. An input to the second problem "Table-k-SUM" consists of a $k \times m$ table and a goal sum S; the entries in row i of the table are denoted by $R_i(1), \ldots, R_i(m)$. The problem is to decide whether one can choose k integers from this table, exactly one from each row, that add up to S. In both problems, the number k is a fixed integer that is not part of the input. Both problems are closely related, and they can be reduced to each other in linear time (Erickson [12]). Both problems are trivially solvable in polynomial time $O(m^k)$.

Here is how to get a better time complexity for Table-2-SUM: Sort the entries in the first row. Then for $j = 1, \ldots, m$ perform a binary search for the value $S - R_2(j)$ in this sorted first row. If the search succeeds at $R_1(i)$, then $R_1(i) = S - R_2(j)$ and the answer is YES. If all searches fail, then the answer is NO.

Fact. *Table-2-SUM can be solved in $O(m \log m)$ time and $O(m)$ space.*

The same approach also yields fast algorithms for Table-k-SUM for all $k \geq 3$: Compute the sum of every $\lceil k/2 \rceil$-tuple of integers that has one entry in each of the first $\lceil k/2 \rceil$ rows; these sums form the first row in a new table. Compute the sum of every $\lfloor k/2 \rfloor$-tuple of integers that has one entry in each of the last $\lfloor k/2 \rfloor$ rows; these sums form the second row in the new table. Apply the above algorithm to this new instance of Table-2-SUM.

Fact. *Table-k-SUM can be solved in $O(m^{\lceil k/2 \rceil} \log m)$ time and $O(m^{\lceil k/2 \rceil})$ space.*

For odd k, the time complexity can be slightly improved to $O(m^{\lceil k/2 \rceil})$; see for instance Erickson [11]. In particular, the 3-SUM problem can be solved in $O(m^2)$ time. We will not go into details, since in this paper we really do not care about logarithmic factors. The main drawback of all these algorithms is their horrible space complexity.

Schroeppel & Shamir [23] improve the space complexity for Table-4-SUM by using a data structure that enumerates the m^2 sums $R_1(i) + R_2(j)$ with $1 \leq i, j \leq m$ in non-decreasing order. This data structure uses only $O(m)$ space. Every time we kick it, it starts working for $O(\log m)$ time steps, and then spits out the next larger sum $R_1(i) + R_2(j)$. The data structure is based on a balanced search tree that supports deletions, insertions, and extracting the minimum with logarithmic work per operation. It is built as follows: In a preprocessing step, we bring the entries in the second row into non-decreasing order. As a consequence, we have for every fixed index i that

$$R_1(i) + R_2(1) \;\leq\; R_1(i) + R_2(2) \;\leq\; \cdots \;\leq\; R_1(i) + R_2(m).$$

For every index i ($1 \leq i \leq m$), the data structure stores the pair (i, j) that corresponds to the first unvisited sum $R_1(i) + R_2(j)$ in this ordering. Whenever the data structure is kicked, it extracts and deletes the pair (i, j) with minimum sum, and inserts the pair $(i, j+1)$ instead. All in all, the enumeration of the m^2 sums costs $O(m^2 \log m)$ time.

Schroeppel & Shamir [23] use two such data structures; the first one generates the sums $x = R_1(i) + R_2(j)$ in non-decreasing order, whereas the second one generates the sums $y = R_3(s) + R_4(t)$ in non-increasing order. Whenever $x + y < S$ holds, the current value of x is too small for reaching the goal sum S; we replace it by the next larger sum $R_1(i) + R_2(j)$ from the first data structure. Whenever $x + y > S$ holds, the current value of y is too large for reaching the goal sum S; we replace it by the next smaller sum $R_3(s) + R_4(t)$ from the second data structure. These steps are repeated over and over again, until one data structure becomes empty (answer NO) or until we reach $x + y = S$ (answer YES).

Fact. *Table-4-SUM can be solved in $O(m^2 \log m)$ time and $O(m)$ space.*

Open problem 1

(a) Is there an $O(m^3 \log m)$ time and $O(m)$ space algorithm for Table-6-SUM?
(b) Is there an $O(m^{\lceil k/2 \rceil - \alpha})$ time algorithm for Table-k-SUM for some integer $k \geq 3$ and some real $\alpha > 0$?

Now let us turn to negative results around the k-SUM and the Table-k-SUM problem. The 3-SUM problem plays a notorious role in computational geometry. Gajentaan & Overmars [15] have put together a long list of geometric problems: All problems on this list can be solved in quadratic time, and for all of them nobody knows how to do better. All problems on this list contain 3-SUM as a special case (under linear time reductions), and for all of them this 3-SUM special case (intuitively) seems to be the main obstacle for breaking through the quadratic time barrier. One example problem on this list is: Given m (possibly overlapping) triangles in the Euclidean plane, compute the area of their union. Another one: Given m pairwise non-intersecting straight line segments in the Euclidean plane, is there a straight line that separates them into two non-empty subsets? And another one: Given m points in the Euclidean plane, are some three of them on a common line? For instance, the linear time reduction from 3-SUM to 3-POINTS-ON-A-COMMON-LINE is based on the following observation: The x-coordinates of the intersection points of the line $y = ax + b$ with the curve $y = f(x) = x^3 - Sx^2$ are the roots of $x^3 - Sx^2 - ax - b = 0$; for every line the sum of these roots equals S, the coefficient of the quadratic term. Consequently, the point set $(a_1, f(a_1)), (a_2, f(a_2)), \ldots, (a_m, f(a_m))$ contains three points $(a_x, f(a_x))$, $(a_y, f(a_y))$, $(a_z, f(a_z))$ on a common line, if and only if $a_x + a_y + a_z = S$. The bottom-line of this paragraph is that research on the 3-SUM problem is severely stuck at the threshold $O(m^2)$.

What about the general k-SUM problem with $k \geq 4$? Here we are stuck around the threshold $O(m^{\lceil k/2 \rceil})$. Erickson [11] proved an $\Omega(m^{\lceil k/2 \rceil})$ lower bound on k-SUM in a certain restricted variant of the linear decision tree model. The additional restriction in his model is that every decision step must be based on testing the sign of some affine linear combination of at most k elements of the input. At first sight, this model seems to be strange, and the lower bound result seems to be quite weak. However, given our general failure in proving reasonable lower bounds for algorithmic problems and given the lack of tools in this area, Erickson's lower bound result in fact is a major breakthrough.

Open problem 2 *Prove a non-trivial lower bound for the k-SUM problem in the algebraic decision tree model or in the algebraic computation tree model (see Ben-Or [4]).*

Downey & Fellows [7,8] have proved that the k-SUM problem with parameter k is W[1]-hard. All these negative results for k-SUM translate into analogous negative results for Table-k-SUM.

After this long polynomial time prelude, we will spend the rest of this section on NP-hard problems. In the NP-hard SUBSET-SUM problem, the input consists of n positive integers $b_1, \ldots, b_n$ and a goal sum B. The problem is to decide whether there exists some subset of the b_i that add up to B. The strongest known negative result for SUBSET-SUM is an $\Omega(n^2)$ lower bound in the algebraic computation tree model of computation [6,4].

On the positive side, Horowitz & Sahni [18] have come up with the following approach for SUBSET-SUM: They split the instance into two parts, one part with $b_1, \ldots, b_{\lfloor n/2 \rfloor}$ and another part with $b_{\lfloor n/2 \rfloor +1}, \ldots, b_n$. They construct a table with two rows, where the first row consists of all the subset sums for the first part, and where the second row consists of all the subset sums for the second part. The table can be computed in $O^*(2^{n/2})$ time. The SUBSET-SUM instance has answer YES, if and only if the constructed Table-2-SUM instance with $S = B$ has answer YES. Our above discussion of Table-2-SUM yields the following result.

Fact. *SUBSET-SUM can be solved in $O^*(2^{n/2})$ time and in $O^*(2^{n/2})$ space.*

Schroeppel & Shamir [23] follow essentially the same idea, but instead of splitting the SUBSET-SUM instance into two parts, they split it into four parts of size approximately $n/4$. This leads to a corresponding instance of Table-4-SUM, and to a substantially improved space complexity.

Fact. *SUBSET-SUM can be solved in $O^*(2^{n/2})$ time and in $O^*(2^{n/4})$ space.*

Generally, if we split the SUBSET-SUM instance into $k \geq 2$ parts, then we get a corresponding table with k rows and $O(2^{n/k})$ elements per row. Applying the fastest known algorithm to the corresponding instance of Table-k-SUM gives a time complexity of $O^*(2^{f(n,k)})$ with $f(n,k) = n \lceil k/2 \rceil / k \geq n/2$. Hence, this approach will not easily lead to an improvement over the time complexity $O^*(2^{n/2})$. Schroeppel & Shamir [23] also construct $t(n)$ time and $s(n)$ space algorithms for SUBSET-SUM for all $s(n)$ and $t(n)$ with $\Omega^*(2^{n/2}) \leq t(n) \leq O^*(2^n)$ and $s^2(n) \cdot t(n) = \Theta^*(2^n)$.

Open problem 3

(a) Construct an $O^*(1.4^n)$ time algorithm for SUBSET-SUM.
(b) Construct an $O^*(1.99^n)$ time and polynomial space algorithm for SUBSET-SUM.

(c) *We have seen that positive results for Table-k-SUM yield positive results for SUBSET-SUM. Can we establish some reverse statement? Do fast (exponential time) algorithms for SUBSET-SUM yield fast (polynomial time) algorithms for Table-k-SUM?*

Another NP-hard problem in this area is the EQUAL-SUBSET-SUM problem: Given n positive integers $b_1, \dots, b_n$, do there exist two disjoint non-empty subsets of the b_i that both have the same sum. A translation of EQUAL-SUBSET-SUM into a corresponding Table-4-SUM instance leads to an $O^*(2^n)$ algorithm for EQUAL-SUBSET-SUM. It might be interesting to design faster algorithms for EQUAL-SUBSET-SUM, and to get some understanding of the relationship between fast algorithms for SUBSET-SUM and fast algorithms for EQUAL-SUBSET-SUM.

3 Graphs and Their Cliques and Cuts

We start this section with the polynomially solvable k-CLIQUE problem: An input consists of an undirected, simple, loopless p-vertex graph $G = (V, E)$. The problem is to decide whether G contains a clique on k vertices. We stress that k is not part of the input. The k-CLIQUE problem is easily solved in polynomial time $O(p^k)$.

Itai & Rodeh [19] observed that fast matrix multiplication can be used to improve this time complexity for 3-CLIQUE: Recall that the product of two $p \times p$ matrices can be computed in $O(p^\omega)$ time, where $\omega < 2.376$ denotes the so-called *matrix multiplication exponent*; see Coppersmith & Winograd [5]. Recall that in the ℓth power A^ℓ of the adjacency matrix A of graph G, the entry at the intersection of row i and column j counts the number of walks with $\ell + 1$ vertices in G that start in vertex i and end in vertex j. Furthermore, a 3-clique $\{x, y, z\}$ yields a walk $x - y - z - x$ with four vertices from x to x, and vice versa, every walk with four vertices from vertex x to x corresponds to a 3-clique. Hence, G contains a 3-clique if and only if A^3 has a non-zero entry on its main-diagonal.

Fact. *The 3-CLIQUE problem for a p-vertex graph can be solved in $O(p^\omega)$ time (where $\omega < 2.376$ is the matrix multiplication exponent) and in $O(p^2)$ space.*

Nešetřil & Poljak [22] extend this idea to the $3k$-CLIQUE problem: For every k-clique C in G, create a corresponding vertex $v(C)$ in an auxiliary graph. Two vertices $v(C_1)$ and $v(C_2)$ are connected by an edge in the auxiliary graph, if and only if $C_1 \cup C_2$ forms a $2k$-clique in G. Note that the auxiliary graph has $O(p^k)$ vertices. Furthermore, graph G contains a $3k$-clique if and only if the auxiliary graph contains a 3-clique.

Fact. *The $3k$-CLIQUE problem for a p-vertex graph can be solved in $O(p^{\omega\, k})$ time and $O(p^{2k})$ space.*

This approach yields a time complexity of $O(p^{\omega\, k+1})$ for $(3k + 1)$-CLIQUE, and a time complexity of $O(p^{\omega\, k+2})$ for $(3k+2)$-CLIQUE. Eisenbrand & Grandoni [9] slightly improve on these bounds for $(3k + 2)$-CLIQUE (with $k \geq 2$) and for

$(3k+1)$-CLIQUE (with $1 \leq k \leq 5$). In particular, for 4-CLIQUE [9] gives a time complexity of $n^{3.334}$.

Open problem 4

(a) Design algorithms with better time and/or space complexities for k-CLIQUE!
(b) Is there an $O(p^{7.5})$ time algorithm for 10-CLIQUE?
(c) Is 3-CLIQUE as difficult as Boolean matrix multiplication?

On the negative side, we have that the k-CLIQUE problem with parameter k is W[1]-hard (Downey & Fellows [7,8]). For the variant where k is part of the input and $k \approx \log n$, Feige & Kilian [14] show that a polynomial time algorithm highly unlikely to exist.

Now let us turn to NP-hard problems. In the MAX-CUT problem, the input consists of an n-vertex graph $G = (V, E)$. The problem is to find a cut of maximum cardinality, that is, a subset $X \subseteq V$ of the vertices that maximizes the number of edges between X and $V - X$. The MAX-CUT problem can be solved easily in $O^*(2^n)$ time by enumerating all possible certificates X. Fedin & Kulikov [13] present an $O^*(2^{|E|/4})$ time algorithm for MAX-CUT; however, it seems a little bit strange to measure the time complexity for this problem in terms of $|E|$ and not in terms of $n = |V|$.

Williams [25] developed the following beautiful approach for MAX-CUT: We partition the vertex set V into three parts V_0, V_1, V_2 that are of roughly equal cardinality $n/3$. We introduce a complete tri-partite auxiliary graph that contains one vertex for every subset $X_0 \subseteq V_0$, one vertex for every subset $X_1 \subseteq V_1$, and one vertex for every subset $X_2 \subseteq V_2$. For every subset $X_i \subseteq V_i$ and every $X_j \subseteq V_j$ with $j = i+1 \pmod 3$, we introduce the directed edge from X_i to X_j. This edge receives a weight $w(X_i, X_j)$ that equals the number of edges in G between X_i and $V_i - X_i$ plus the number of edges between X_i and $V_j - X_j$ plus the number of edges between X_j and $V_i - X_i$. Note that for $X_i \subseteq V_i$ $(i = 0, 1, 2)$ the cut $X_0 \cup X_1 \cup X_2$ cuts exactly $w(X_0, X_1) + w(X_1, X_2) + w(X_2, X_0)$ edges in G. Consequently, the following three statements are equivalent:

– The graph G contains a cut with z edges.
– The auxiliary graph contains a 3-clique with total edge weight z.
– There exist non-negative integers z_{01}, z_{12}, z_{20} with $z_{01} + z_{12} + z_{20} = z$, such that the auxiliary graph contains a 3-clique on three vertices $X_i \subseteq V_i$ $(i = 0, 1, 2)$ with $w(X_0, X_1) = z_{01}$ and $w(X_1, X_2) = z_{12}$ and $w(X_2, X_0) = z_{20}$.

The condition in the third statement is easy to check: There are $O(|E|^3)$ possible triples (z_{01}, z_{12}, z_{20}) to consider. For each such triple, we compute a corresponding simplified version of the auxiliary graph that only contains the edges of weight z_{ij} between vertices $X_i \subseteq V_i$ and $X_j \subseteq V_j$. Then everything boils down to finding a 3-clique in the simplified auxiliary graph on $O(2^{n/3})$ vertices.

Fact. *The MAX-CUT problem can be solved in $O^*(2^{\omega\, n/3})$ time and $O^*(2^{\omega\, n/3})$ space. Note that $2^{\omega\, n/3} < 1.732^n$.*

Of course, William's algorithm could also be built around a partition of the vertex set V into four parts of roughly equal cardinality $n/4$, or around a partition of the vertex set V into k parts of roughly equal cardinality n/k. The problem then boils down to finding a k-clique in some simplified auxiliary graph on $O(2^{n/k})$ vertices. With the currently known k-CLIQUE algorithms, this will not give us an improved time complexity.

Open problem 5

(a) Design a faster exact algorithm for MAX-CUT.
(b) Construct an $O^*(1.99^n)$ time and polynomial space algorithm for MAX-CUT.

An input of the NP-hard BISECTION problem consists of an n-vertex graph $G = (V, E)$. The problem is to find a subset $X \subseteq V$ with $|X| = n/2$ that minimizes the number of edges between X and $V - X$. The approach of Williamson yields an $O^*(2^{\omega\, n/3})$ time algorithm for BISECTION. Can you do better?

4 Sets and Their Subsets

There is a number of exact algorithms in the literature that attack an NP-hard problem by running through all the subsets of an underlying n-element ground set, while generating and storing useful auxiliary information. Since an n-element ground set has 2^n subsets, the time complexities of these approaches are typically $\Omega^*(2^n)$. And also the space complexities of these approaches are typically $\Omega^*(2^n)$, since they store and remember auxiliary information for every subset.

A good example for this approach is the famous dynamic programming algorithm of Held & Karp [17] for the travelling salesman problem (TSP): A travelling salesman has to visit the cities 1 to n. He starts in city 1, runs through the cities $2, 3, \ldots, n-1$ in arbitrary order, and finally stops in city n. The distance $d(i, j)$ from city i to city j is specified as part of the input. The goal is to find a path that minimizes the total travel length of the salesman. The dynamic program of Held & Karp [17] introduces for every non-empty subset $S \subseteq \{2, \ldots, n-1\}$ of the cities and for every city $i \in S$ a corresponding state $[S; i]$. By $\textsc{Length}[S; i]$ we denote 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, $\textsc{Length}[\{i\}; i] = d(1, i)$ holds for every $i \in \{2, \ldots, n-1\}$. And for every $S \subseteq \{2, \ldots, n-1\}$ with $|S| \geq 2$ we have

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

By processing the subsets S in order of increasing cardinality, we can compute the value $\textsc{Length}[S; i]$ in time proportional to $|S|$. In the end, the optimal travel length is given as the minimum $\min_{2 \leq k \leq n-1} \textsc{Length}[\{2, \ldots, n-1\}; k] + d(k, n)$.

Fact. *The TSP can be solved in $O^*(2^n)$ time and $O^*(2^n)$ space.*

Open problem 6

(a) *Construct an exact algorithm for the n-city TSP with $O^*(1.99^n)$ time complexity!*

(b) *Construct an exact algorithm for the n-city TSP with $O^*(2^n)$ time complexity and polynomial space complexity!*

In the Hamiltonian path problem, we have to decide for a given graph $G = (V, E)$ with vertices $1, \ldots, n$ whether it contains a spanning path starting in vertex 1 and ending in vertex n. The Hamiltonian path problem forms a simpler special case of the TSP. Karp [20] (and independently Bax [1]) provided a cute solution for the restriction of Problem 6.(b) to this Hamiltonian special case. We use the following definitions. A *walk* in a graph is a sequence $v_1, \ldots, v_k$ of vertices such that every pair of consecutive vertices is connected by an edge; vertices and edges may show up repeatedly in a walk. For a subset $S \subseteq V$ we denote by $\textsc{Walk}(S)$ the set of all walks with n vertices in G that start in vertex 1, end in vertex n, and avoid all the vertices in S. Let A be the adjacency matrix of $G - S$. Recall that in the kth power A^k of A, the entry at the intersection of row i and column j counts the number of walks with $k + 1$ vertices in $G - S$ that start in vertex i and end in vertex j. Therefore, the number of walks in $\textsc{Walk}(S)$ can be read from matrix A^{n-1}:

Fact. *For every subset $S \subseteq V$, the cardinality $|\textsc{Walk}(S)|$ can be determined in polynomial time.*

If a walk through n vertices in G does not avoid any vertex k, then it must visit all the vertices, and hence must form a Hamiltonian path. Consequently, the number of Hamiltonian paths from 1 to n in G equals

$$|\textsc{Walk}(\emptyset)| \; - \; |\bigcup_{k=2}^{n-1} \textsc{Walk}(\{k\})| \quad = \quad \sum_{S \subseteq V} (-1)^{|S|} \cdot |\textsc{Walk}(S)|.$$

Here we have used the inclusion-exclusion principle. The sum in the right hand side of the displayed equation is straightforward to compute by applying the fact discussed above. We only need to remember the partial sum of all the terms evaluated so far, and the space used for evaluating one term can be reused in evaluating the later terms. All in all, evaluating and adding up the values of $O(2^n)$ terms yields an $O^*(2^n)$ time and polynomial space algorithm for *counting* the number of Hamiltonian paths. The following fact is a trivial consequence of this:

Fact. *The Hamiltonian path problem in an n-vertex graph can be solved in $O^*(2^n)$ time and polynomial space.*

Eppstein [10] improves on this polynomial space result for Hamiltonian path in the special case of *cubic* graphs: He presents an algorithm that uses $O^*(1.297^n)$ time and linear space. Bax [2] and Bax & Franklin [3] have extended the inclusion-exclusion approach to a number of counting problems around paths and cycles in n-vertex graphs. For all these problems, the time complexity is $O^*(2^n)$ and the space complexity is polynomial.

Open problem 7 *Construct $O^*(1.99^n)$ time exact algorithms for the following counting problems in n-vertex graphs G:*

(a) Count the number of paths between a given pair of vertices in G.
(b) Count the number of cycles in G.
(c) Count the number of cycles through a given vertex in G.
(d) Count the number of cycles of a given length ℓ in G.

Now let us turn to some relatives of the n-city TSP. For a fixed Hamiltonian path from city 1 to city n and for a fixed city k, we denote by the *delay* of city k the length of the subpath between city 1 and city k. In the travelling repairman problem (TRP), the goal is to find a Hamiltonian path from city 1 to city n that minimizes the sum of delays over all cities. In the precedence constrained travelling repairman problem (prec-TRP), the input additionally specifies a partial order on the cities. A Hamiltonian path is feasible, if it obeys the partial order constraints.

Here is a related scheduling problem SCHED: There are n jobs $1, \ldots, n$ with processing times $p_1, \ldots, p_n$. The jobs are partially ordered (precedence constrained), and if job i precedes job j in the partial order, then i must be processed to completion before j can begin its processing. All jobs are available at time 0, and job preemption is not allowed. The goal is to schedule the jobs on a single machine such that all precedence constraints are obeyed and such that the total job completion time $\sum_{j=1}^{n} C_j$ is minimized; here C_j is the time at which job j completes in the given schedule. SCHED is the special case of prec-TRP where the distances between cities $i \neq j$ are given by $d(i, j) = p_j$. It is quite straightforward to design an $O^*(2^n)$ time and $O^*(2^n)$ space dynamic programming algorithm for prec-TRP (and for its special cases TRP and SCHED).

Open problem 8

(a) Construct an $O^*(1.99^n)$ time exact algorithm for TRP or for SCHED or for prec-TSP.
(b) Provide evidence in favor of or against the following claim: If there exists an $O^*(c^n)$ time exact algorithm with $c < 2$ for one of the four problems TSP, TRP, SCHED, prec-TSP, then there exist $O^*(c^n)$ time exact algorithms for all four problems.

References

1. E.T. Bax (1993). Inclusion and exclusion algorithm for the Hamiltonian path problem. *Information Processing Letters 47*, 203–207.
2. E.T. Bax (1994). Algorithms to count paths and cycles. *Information Processing Letters 52*, 249–252.
3. E.T. Bax and J. Franklin (1996). A finite-difference sieve to count paths and cycles by length. *Information Processing Letters 60*, 171–176.
4. M. Ben-Or (1983). Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing (STOC'1983)*, 80–86.

5. D. Coppersmith and S. Winograd (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation 9*, 251–280.
6. D. Dobkin and R.J. Lipton (1978). A lower bound of $\frac{1}{2}n^2$ on linear search programs for the knapsack problem. *Journal of Computer and System Sciences 16*, 413–417.
7. R.G. Downey and M.R. Fellows (1995). Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theoretical Computer Science 141*, 109–131.
8. R.G. Downey and M.R. Fellows (1999). *Parameterized complexity.* Springer Monographs in Computer Science.
9. F. Eisenbrand and F. Grandoni (2004). On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, to appear.
10. D. Eppstein (2003). The traveling salesman problem for cubic graphs. In *Proceedings of the 8th International Workshop on Algorithms and Data Structures (WADS'2003)*, Springer-Verlag, LNCS 2748, 307–318.
11. J. Erickson (1999). Lower bounds for linear satisfiability problems. *Chicago Journal of Theoretical Computer Science 1999(8).*
12. J. Erickson (2004). Private communication.
13. S.S. Fedin and A.S. Kulikov (2002). Solution of the maximum cut problem in time $2^{|E|/4}$. (In Russian). *Zapiski Nauchnykh Seminarov Sankt-Peterburgskoe Otdeleniya Matematicheskiĭ Institut imeni V.A. Steklova 293*, 129–138.
14. U. Feige and J. Kilian (1997). On limited versus polynomial nondeterminism. *Chicago Journal of Theoretical Computer Science 1997.*
15. A. Gajentaan and M.H. Overmars (1995). On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry 5*, 165–185.
16. M.R. Garey and D.S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco.
17. M. Held and R.M. Karp (1962). A dynamic programming approach to sequencing problems. *Journal of SIAM 10*, 196–210.
18. E. Horowitz and S. Sahni (1974). Computing partitions with applications to the knapsack problem. *Journal of the ACM 21*, 277–292.
19. A. Itai and M. Rodeh (1978). Finding a minimum circuit in a graph. *SIAM Journal on Computing 7*, 413–423.
20. R.M. Karp (1982). Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters 1*, 49–51.
21. E.L. Lawler (1976). A note on the complexity of the chromatic number problem. *Information Processing Letters 5*, 66–67.
22. J. Nešetřil and S. Poljak (1985). On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae 26*, 415–419.
23. 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.
24. R.E. Tarjan and A.E. Trojanowski (1977). Finding a maximum independent set. *SIAM Journal on Computing 6*, 537–546.
25. R. Williams (2004). A new algorithm for optimal constraint satisfaction and its implications. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP'2004)*, Springer Verlag, 2004.
26. G.J. Woeginger (2003). Exact algorithms for NP-hard problems: A survey. In *Combinatorial Combinatorial Optimization – Eureka, you shrink!"*, LNCS 2570, Springer Verlag, 185–207.