DOI:10.1145/2428556.2428575

Discovering surprises in the face of intractability.

BY FEDOR V. FOMIN AND PETTERI KASKI

Exact Exponential Algorithms

MANY COMPUTATIONAL PROBLEMS have been shown to be intractable, either in the strong sense that no algorithm exists at all—the canonical example being the undecidability of the Halting Problem—or that no *efficient* algorithm exists. From a theoretical perspective perhaps the most intriguing case occurs with the family of *NP*-complete problems, for which *it is not known* whether the problems are intractable. That is, despite extensive research, neither is an efficient algorithm known, nor has the existence of one been rigorously ruled out.[16]

To cope with intractability, advanced techniques such as *parameterized algorithms*[10,13,31] (that isolate the exponential complexity to a specific structural parameter of a problem instance) and *approximation algorithms*[34] (that produce a solution whose value is guaranteed to be within a known factor of the value of an optimum solution) have been developed. But what can we say about finding exact solutions of non-parameterized instances of intractable problems? At first glance, the general case of an *NP*-complete problem is a formidable opponent: when faced with a problem whose instances

» key insights

- **While it remains open whether or not *P* equals *NP*, significant progress in the area of exhaustive search has been made in the last few years. In particular, many *NP*-complete problems can now be solved significantly faster by exhaustive search. The area of exact exponential algorithms studies the design of such techniques.**
- **While many exact exponential algorithms date back to the early days of computing, a number of beautiful surprises have emerged recently.**

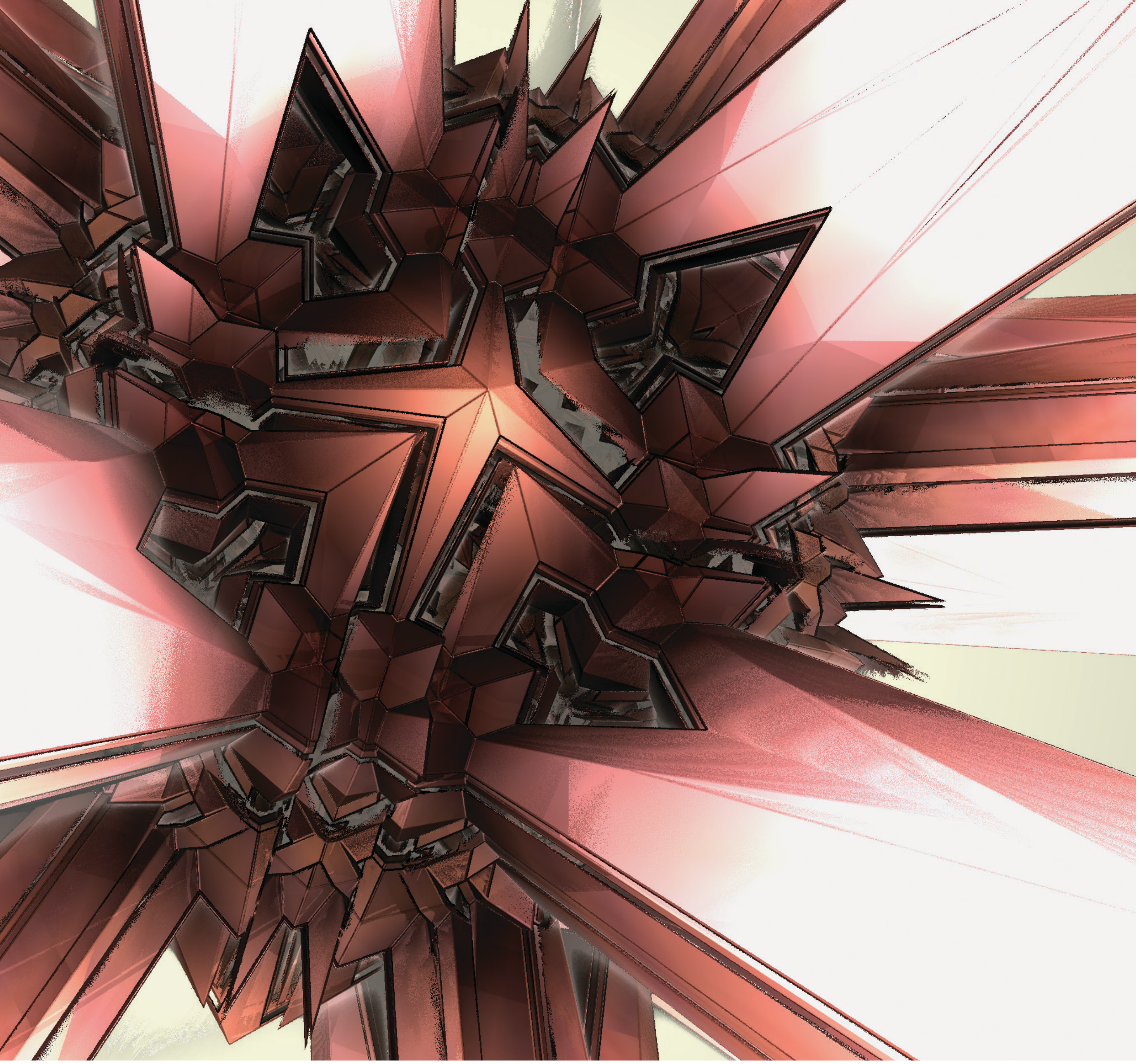


ILLUSTRATION BY MIKAEL HVIDTFELDT CHRISTENSEN

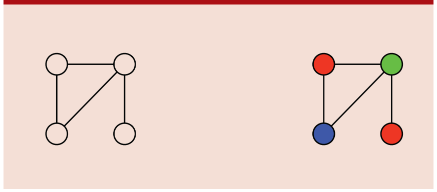
can express arbitrary nondeterministic computation, how is one to proceed at solving a given instance, apart from the obvious exhaustive search that "tries out all the possibilities"?

Fortunately, the study of algorithms knows many positive surprises. Computation is malleable in nontrivial ways, and subtle algorithmically exploitable structure has been discovered where none was thought to exist. Furthermore, the more generous a time budget the algorithm designer has, the more techniques become available. Especially so if the budget is exponential in the size of the input. Thus, absent complexity-theoretic obstacles, *one should be able to do better than exhaustive search*. This is the objective of *exact exponential algorithms*.[15]

Arguably, the oldest design technique to improve upon exhaustive search is *branching* or *backtrack search*,[18,35] which recursively splits the exhaustive search space, attempting to infer in the process that parts of the space need not be visited. For recent applications of branching techniques, we refer to Eppstein[12] and Fomin et al.[14] Another classical design technique is *dynamic programming*,[2] which derives a solution from the bottom up by storing solutions of smaller subproblems and combining them via a recurrence relation to progressively yield solutions of larger subproblems. These two techniques in many cases give significant improvements over plain exhaustive search, but in other cases, no improvement at all upon exhaustive search has been available, and many problems remain with this status.

In what follows, we do not try to give a comprehensive survey of exact exponential algorithms. Indeed, even listing the most significant results would require a format different from this review. Instead, we have chosen to review the area by highlighting three recent results. In each case, research had been essentially stuck for an extended period of time—in one case for almost 50 years!—and it was conceivable that perhaps no improvement could be obtained over the known algorithms. But computation has the power

Figure 1. Graph coloring.



to surprise, and in this article we hope to convey some of the excitement surrounding each result. We also find these results particularly appealing because they are *a posteriori* quite accessible compared with many of the deep results in theoretical computer science, and yet they illustrate the subtle ways in which computation can be orchestrated to solve a problem.

Three *NP*-Complete Problems

The three problems we discuss in more detail are *Maximum 2-Satisfiability*, *Graph Coloring*, and *Hamiltonian Path*. We start by giving an overview of previous approaches to attack each problem, and then in the subsequent sections discuss the novel algorithms.

MAX-2-SAT. The satisfiability problem takes as input a logical expression built from n variables $x_1, x_2, \dots, x_n$ and the Boolean connectives $\neg$ (NOT), $\vee$ (OR), and $\wedge$ (AND). The task is to decide whether the expression can be *satisfied* by assigning a truth value, either 0 (false) or 1 (true), to each variable such that the expression evaluates to 1. For example, the expression

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \qquad (1)$$

can be satisfied by setting $x_1 = 1$ and $x_2 = 0$, whereas the expression

$$\begin{aligned} &(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \\ &\wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \\ &\wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ &\wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \end{aligned} \qquad (2)$$

is not satisfiable.

It is customary to assume that the input expression is in *conjunctive normal form*, where it is required that the expression is the AND of *clauses*, each of which is an OR of *literals*, which are variables or negations of variables. If all clauses have k literals, then the expression is in *k-conjunctive normal form*, or *k-CNF*. For example, (1) is in 2-CNF and (2) is in 3-CNF. The satisfiability problem for an expression in k-CNF is called the *k-CNF satisfiability* or *k-SAT* problem. It is polynomial-time solvable for $k \le 2$ and *NP*-complete for $k \ge 3$.[17]

A stronger variant of the problem, *maximum k-CNF satisfiability* or *MAX-k-SAT*, gives a threshold t as additional input, and the task is to decide whether there is an assignment of truth values to the variables such that at least t clauses evaluate to 1. This variant is *NP*-complete for all $k \ge 2$.[17]

MAX-k-SAT is trivially solvable by trying all possible truth assignments. When a formula has n variables, it has 2^n possible assignments and for each assignment we can compute in polynomial time how many clauses are satisfied. Thus, the total running time, up to a factor polynomial in n, is dominated by 2^n. A special case of the problem, known as MAX-CUT, can be obtained by formulating MAX-2-SAT as a problem of partitioning the vertices of an n-vertex graph into two subsets such that at least t edges cross between subsets. However, even in the special cases of MAX-2-SAT and MAX-CUT, no better algorithm than the trivial exhaustive search was known until the work of Williams.[36]

Graph Coloring. In the graph coloring problem, we are given as input a graph G with n vertices and a palette of k colors. The task is to decide whether it is possible to assign to each vertex a color from the palette so that the coloring is *proper*, that is, every edge has distinct colors at its ends. For example, the graph in Figure 1 admits a proper coloring of its vertices using three colors.

The graph coloring problem is polynomial-time solvable for $k \le 2$ and *NP*-complete for $k \ge 3$.[17] The minimum number of colors for which a graph G has a proper coloring is the *chromatic number* $\chi(G)$ of G.

The first algorithmic approaches to compute the chromatic number of a graph can be traced back to the work of Zykov.[41] The idea is based on a branching procedure. The base case of the branching occurs when all pairs of vertices of G are adjacent, that is, G is a complete graph, in which case the chromatic number is equal to the number of vertices in G. Otherwise, G contains a pair u, v of vertices that are not joined by an edge. In every proper coloring of G it holds that u and v either have distinct colors (in which case we construct a new graph by joining u and v with an edge), or have the same color (in which case we construct a new graph by identifying u and v). This enables us to recursively branch on the two cases and return the best of the two solutions obtained. In terms of running time, however, this approach is in general no better than plain exhaustive search, which involves iterating through the k^n distinct ways to color the n vertices of G using the k available colors, and for each coloring testing whether it is proper.

After Zykov's seminal work, the history of algorithms for graph coloring benefits from a digression to the study of independent sets in graphs. In particular, every proper coloring of G has the property that no two vertices of the same color are joined by an edge. Such a set of vertices is an *independent set* of G. An independent set of G is *maximal* if it is not a proper subset of a larger independent set of G. In 1976, Lawler[27] observed that dynamic programming and advances in the study of independent sets can be used to drastically improve upon the k^n exhaustive search. Let us first develop a basic version of the algorithm. Since each color class in a proper coloring of G is an independent set of G, we have that G is k-colorable if and only if the vertex set V of G decomposes into a union of k independent sets of G. Stated in terms of the chromatic number, we have $\chi(G) = 0$ if G has no vertices; otherwise, we have

$$\chi(G) = 1 + \min\{\chi(G \setminus I) : I \in \mathcal{I}(G)\}, \qquad (3)$$

where $\mathcal{I}(G)$ is the family of all nonempty independent sets of G, and $G\setminus I$ denotes the graph obtained from G by deleting the vertices in I. For every subset $X \subseteq V$, we can thus compute the chromatic number $\chi(G[X])$ of the subgraph of G induced by X as follows. When X is empty, we set $\chi(G[X]) = 0$. When X is nonempty, we compute the value $\chi(G[X])$ from the already computed values of proper subsets of X by making use of (3).

What is the running time of this algorithm? The algorithm considers all subsets $X \subseteq V$, and for each such X, it considers all $I \subseteq X$ that are independent in $G[X]$. The number of such I is at most $2^{|X|}$. Thus, the number of steps of the algorithm is, up to a factor polynomial in n, at most $\sum_{i=0}^{n}\binom{n}{i}2^i = 3^n$.

Lawler also observed that the basic 3^n-algorithm can be improved. Namely, instead of going through all subsets $I \subseteq X$ that are independent in $G[X]$, it suffices to consider only maximal independent sets of $G[X]$. It was known[29] already in the 1960s that the number of maximal independent sets in a graph with i vertices is at most $3^{i/3}$, and that these sets can be listed in time $\mathcal{O}(3^{i/3}n)$. Thus, the exponential part of the running time of the algorithm is bounded by

$$\sum_{i=0}^{n}\binom{n}{i}3^{i/3} = \left(1+\sqrt[3]{3}\right)^n < 2.45^n.$$

It is possible to make even further improvements of this idea by more accurate counting of large and small maximal independent sets.[11] But in all these improvements the following common pattern seemed unavoidable: we have to go through all vertex subsets of the graph, and for each subset, we have to enumerate an exponential number of subsets, resulting in time C^n, for a constant $C > 2$.

Hamiltonian Path. In the *NP*-complete *Hamiltonian cycle* problem, we are given a graph on n vertices and the task is to decide whether the graph has a Hamiltonian cycle, which is a cycle visiting every vertex of the graph exactly once. For example, the graph in Figure 2 has a Hamiltonian cycle, outlined in bold edges.

This is a special case of the famous *Traveling Salesman Problem*, where the task is to, given an $n \times n$ matrix of travel costs between n cities, design a travel schedule that visits each city exactly once and returns back to the starting point so that the total cost is minimized.

A stronger variant, the *Hamiltonian path* problem, constrains one of the vertices as the first vertex s and another vertex t as the last vertex, and asks us to decide whether the graph has a path that starts at s, ends at t, and visits all the vertices exactly once. (By trying all the pairs $\{s, t\}$ joined by an edge, we can solve the Hamiltonian cycle problem if we can solve the Hamiltonian path problem.)

For the Hamiltonian path problem, exhaustive search iterates through the $(n - 2)!$ ways to arrange the n vertices into a sequence that starts at s and ends at t, testing for each sequence whether it forms a path (of the minimum cost).

Bellman[3] and Held and Karp[19] used dynamic programming to solve the problem in time $\mathcal{O}(2^n n^2)$, by keeping track for every vertex v and vertex subset S, the existence (or the minimum cost) of a path from s to v that visits exactly the vertices in $S \subseteq V$. This algorithm, however, requires also space 2^n.

It is possible to solve the problem within the same running time but within polynomial space by making use of the principle of inclusion and exclusion. It seems that essentially the same approach was rediscovered several times.[1,23,25] To illustrate the design, Figure 3 displays a graph with $n = 8$ vertices $\{a, b, c, d, e, f, g, h\}$.

Let us assume that $s = a$ and $t = h$. A walk of length $n - 1$ that starts from s and ends at t can be viewed as a string of length $2n - 1$ with alternating and possibly repeating vertices and edges, such as

$$aAeCbDfFcGgIdJh \qquad (4)$$

or

$$aBfDbEgGcFfFcHh. \qquad (5)$$

We observe that each such walk makes exactly n visits to vertices and contains, possibly with repetitions, $n - 1$ edges. Moreover, the walk is a Hamiltonian path if and only if the walk visits n distinct vertices; indeed, otherwise there is at least one vertex that is visited more than once. For example, (4) is a path and (5) is a non-path because it repeatedly visits f (and c).

Although finding a Hamiltonian path is a challenging computational problem, one can compute in polynomial time the number of walks of length k from s to t. Indeed, let A be the adjacency matrix of G with rows and columns indexed by vertices of G, such that the (x, y)-entry of A is set to 1 if there is an edge from x to y in G, and set to 0 otherwise. By induction on k we observe that the (s, t)-entry of the kth matrix power A^k counts the number of walks of length k in G that start at s and end at t. Therefore, the number of walks of length $n - 1$ can be read from the matrix A^{n-1}, which can be computed in time polynomial in n.

One approach to isolate the paths among the walks is to employ the *principle of inclusion and exclusion*. Consider a finite set X and three subsets A_1, A_2, and A_3 (see Figure 4).

To obtain $|A_1 \cup A_2 \cup A_3|$, we can use the following formula

Figure 3. Example for Hamiltonian path.

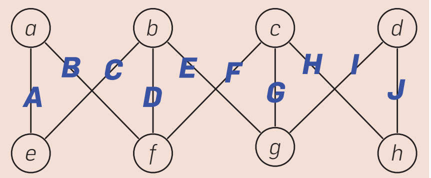


Figure 2. Hamiltonian cycle.

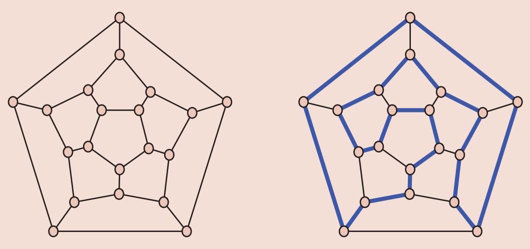


Figure 4. A Venn diagram for three subsets.

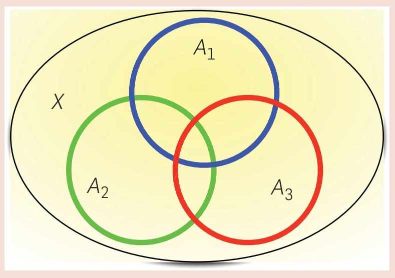
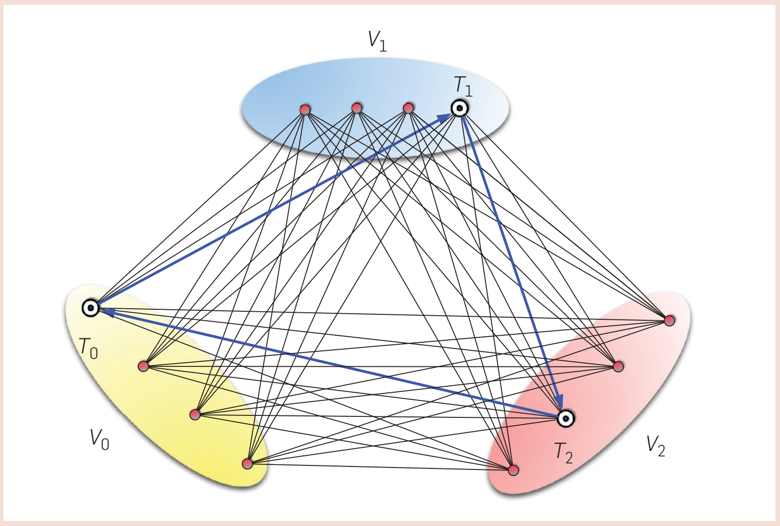


Figure 5. The directed graph D with one triangle $T_0T_1T_2$ highlighted.



$$\begin{aligned}|A_1\cup A_2\cup A_3| = {} & |A_1|+|A_2|+|A_3| \\ & -|A_1\cap A_2|-|A_1\cap A_3| \\ & -|A_2\cap A_3|+|A_1\cap A_2\cap A_3|,\end{aligned}$$

or, equivalently,

$$\begin{aligned}|X\setminus(A_1\cup A_2\cup A_3)| = {} & |X|-|A_1|-|A_2|-|A_3| \\ & +|A_1\cap A_2|+|A_1\cap A_3| \\ & +|A_2\cap A_3| \\ & -|A_1\cap A_2\cap A_3|.\end{aligned}$$

The principle of inclusion and exclusion generalizes the last formula to the case when there are q subsets $A_1, A_2, \ldots, A_q$ of X by

$$\left|X\setminus\bigcup_{i=1}^{q} A_i\right| = \sum_{J\subseteq\{1,2,\ldots,q\}} (-1)^{|J|}\left|\bigcap_{j\in J} A_j\right|. \quad (6)$$

Let us come back to the Hamiltonian path problem. Take $q = n - 2$ and suppose that the vertices other than s and t are labeled with integers 1, 2, ..., $n - 2$. Let X be the set of all walks of length $n - 1$ from s to t and, for each $i = 1, 2, \ldots, n - 2$, let A_i be the set of walks in X that avoid the vertex i. Then, $X\setminus\bigcup_{i=1}^{q} A_i$ is the set of Hamiltonian paths, and we can use (6) to count their number. In particular, for each fixed $J \subseteq \{1, 2, \ldots, q\}$, the right-hand side of (6) can be computed time polynomial in n by counting the number of walks of length $n - 1$ from s to t in the graph with the vertices in J deleted.

This approach can be used to compute the number of Hamiltonian paths in an n-vertex graph in time $\mathcal{O}(2^n n)$. It is also possible to obtain similar running time by making use of dynamic programming. But in both approaches, it seemed that the most time consuming part of the procedure, going through all possible vertex subsets, was unavoidable. This situation was particularly frustrating because the 2^n barrier had withstood attacks since the early 1960s.

Surprise 1: MAX-2-SAT

Let us recall that for MAX-2-SAT the challenge was to break the 2^n barrier in running time. The following approach for doing this is due to Williams.[36] An alternative approach via sum-product algorithms is due to Koivisto.[26]

Let us start with a seemingly unrelated task, namely that of deciding whether a given directed graph D contains a *triangle*, that is, a triple x, y, z of vertices such that the arcs xy, yz, and zx occur in D. While the immediate combinatorial approach to find a triangle in a ν-vertex graph is to try all possible triples of vertices, which would require $\mathcal{O}(\nu^3)$ steps, there is a faster algorithm of Itai and Rodeh.[22] The algorithm relies on formulating the problem in terms of linear algebra. Let A be the adjacency matrix of D, and recall that the (s, t)-entry of the kth power A^k counts the number of walks of length k from s to t. In particular, every walk of length 3 that starts and ends at a vertex x must pass through three distinct vertices, and thus form a triangle, enabling us to extract the number of triangles in D from the diagonal entries of the matrix A^3. Thus, it suffices to compute the matrix A^3. The immediate algorithm for computing the product of two $\nu \times \nu$ matrices requires $\mathcal{O}(\nu^3)$ steps. However, this product can be computed in time $\mathcal{O}(\nu^\omega)$, where $\omega < 2.376$ is the so-called square matrix multiplication exponent; see Coppersmith and Winograd[7] and Strassen.[32] Very recently, it has been shown that $\omega < 2.3727$.[33]

The key insight is now to exploit the fact that triangles can be found quickly to arrive at a nontrivial algorithm for MAX-2-SAT. Toward this end, suppose we are given as input a 2-CNF formula F over n variables. We may assume that n is divisible by 3 by inserting dummy variables as necessary. Let X be the set of variables of F and let X_0, X_1, X_2 be an arbitrary partition of X into sets of size $n/3$.

Let us transform the instance F into a directed graph D as follows. For every $i = 0, 1, 2$ and every subset $T_i \subseteq X_i$, the graph D has a vertex T_i. The meaning of T_i is that it corresponds to an assignment that sets all variables in T_i to the value 1 and all variables in $X_i\setminus T_i$ to the value 0. Let us write V_i for the set of all subsets $T_i \subseteq X_i$. The arcs of D are all possible pairs of the form (T_i, T_j), where $T_i \subseteq X_i$, $T_j \subseteq X_j$, and $j \equiv i + 1 \pmod 3$. We observe that D has $\nu = 3 \times 2^{n/3}$ vertices and $3 \times 2^{2n/3}$ arcs. For $i = 0, 1, 2$, let the set C_i consist of all clauses of F that either (a) contain variables only from X_i; or (b) contain one variable from X_i and one variable from X_j, with $j \equiv i + 1 \pmod 3$. Now observe that every clause of F has at most two variables. In particular, either both these variables belong to some set X_i, or one variable is in X_i and the other is in X_j with $j \equiv i + 1 \pmod 3$. Thus, the sets C_0, C_1, C_2 partition the clauses in F. We still require weights on the arcs of D. Let us set the weight $w(T_i, T_j)$ of the arc from $T_i \subseteq X_i$ to $T_j \subseteq X_j$ to be equal to the number of clauses in C_i satisfied by assigning the value 1 to all variables in $T_i \cup T_j$ and the value 0 to all remaining variables in $(X_i \cup X_j) \setminus (T_i \cup T_j)$.

To illustrate the construction, let us assume F is the following formula

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (x_3 \vee x_4) \wedge (x_1 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6)$$

and partition the variables so that $X_0 = \{x_1, x_2\}$, $X_1 = \{x_3, x_4\}$, and $X_2 = \{x_5, x_6\}$. Then, $C_0 = \{(x_1 \vee x_2), (\neg x_2 \vee x_3), (x_1 \vee x_3), (\neg x_2 \vee x_4)\}$, $C_1 = \{(x_3 \vee x_4), (\neg x_4 \vee \neg x_6)\}$, and $C_2 = \{(x_1 \vee \neg x_5)\}$. Figure 5 illustrates the underlying graph D, where each set V_0, V_1, V_2 has size 4. For example, $V_0 = \{\emptyset, \{x_1\}, \{x_2\}, \{x_1, x_2\}\}$. For sets $T_0 = \emptyset$, $T_1 = \{x_3, x_4\}$, and $T_2 = \{x_6\}$, the corresponding assignment, viz. $x_1 = x_2 = 0$, $x_3 = 1$, $x_4 = 1$, $x_5 = 0$, $x_6 = 1$, satisfies five clauses. Accordingly, the weight of the triangle $T_0T_1T_2$ in D is also five.

The equivalence of the following statements follows from the construction of D: (i) There is a subset of variables $T \subseteq X$ such that exactly t clauses are satisfied by assigning the value 1 to variables in T and the value 0 to the variables in $X \backslash T$. (ii) The graph D contains a triangle $T_0T_1T_2$ with $T_i \subseteq X_i$ for each $i = 0, 1, 2$ such that

$$t = w(T_0, T_1) + w(T_1, T_2) + w(T_2, T_0).$$

Thus, to find an assignment that satisfies most clauses, it suffices to find a heaviest triangle in D.

We are almost done. Indeed, every formula with n variables has at most $4n^2$ clauses of length 2, and hence to find a heaviest triangle, it suffices to test for the existence of a triangle of weight t for each $0 \leq t \leq 4n^2$ in turn. To test for a triangle of weight t, we go through all possible $\mathcal{O}(t^3)$ partitions $t = t_0 + t_1 + t_2$ into nonnegative parts, and for each partition, we construct a subgraph D_{t_0,t_1,t_2} of D by leaving only arcs of weight t_i for arcs going from subsets of X_i to subsets of X_j with $j \equiv i + 1 \pmod 3$. Finally, it suffices to decide whether D_{t_0,t_1,t_2} has a triangle. The subgraph D_{t_0,t_1,t_2} can be constructed in time $\mathcal{O}(2^{2n/3}n)$ by going through all arcs of D. The total running time is thus

$$\mathcal{O}\left(\sum_{t=0}^{4n^2} t^3\left(2^{\omega n/3} + 2^{2n/3}\right)\right).$$

Because $\omega < 2.376$, we conclude that the running time of the algorithm is $\mathcal{O}(1.74^n)$.

Surprise 2: Graph Coloring

The next surprise is due to Björklund et al.[6] To explain the idea of the algorithm, it will again be convenient to start with a task that may appear at first completely unrelated, namely the multiplication of polynomials. To multiply two given polynomials, the elementary algorithm is to cross-multiply the monomials pairwise and then collect to obtain the result:

$$\begin{aligned}(1 + 3x + x^2)(2 - x + x^2) &= 2 - x + x^2 + 6x - 3x^2 + 3x^3 + 2x^2 - x^3 + x^4 \\ &= 2 + 5x + 2x^3 + x^4.\end{aligned}$$

In particular, if we are multiplying two polynomials of degree d (that is, the highest degree of a monomial with a nonzero coefficient is d), we require $\mathcal{O}(d^2)$ steps to get the result via the elementary algorithm due to the cross-multiplication of monomials. Fortunately, we can drastically improve upon the elementary algorithm by deploying the fast Fourier transform (FFT) to evaluate both input polynomials (given as two lists of $d + 1$ coefficients, one coefficient for each monomial) at $2d + 1$ distinct points, $x_0, x_1, \ldots, x_{2d}$, then multiplying the evaluations pointwise, and finally employing the inverse FFT to recover the list of coefficients for the product polynomial. With such an algorithm, the number of operations is reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(d \log d)$.

But what about graph coloring? Could we formulate the task of decomposing the vertex set into a union of independent sets of G as a task *analogous* to polynomial multiplication? Let us try to find the solution incrementally for $j = 1, 2, \ldots, k$. Suppose we have a list of all the sets of vertices that decompose into a union of j independent sets of G, and would like to determine such a list for $j + 1$.

Let us consider an example. Figure 6 displays a graph with $n = 4$ whose independent sets are

$$\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, d\}, \{c, d\}.$$

For $j = 2$, the sets of vertices that decompose into a union of j independent sets are

$$\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}.$$

Given the family of independent sets and the family of solutions for j, we would like to determine the family of solution for $j + 1$. Pursuing an analogy with polynomial multiplication, we can view the sets in both set families as "monomials" and multiply these "monomials" using set union. For example:

$$\begin{aligned}(\emptyset + \{a\} + \{a, b\}) \cup (\emptyset + \{b, c\} + \{c\}) &= \emptyset + \{b, c\} + \{c\} \\ &\quad + \{a\} + \{a, b, c\} + \{a, c\} \\ &\quad + \{a, b\} + \{a, b, c\} + \{a, b, c\} \\ &= \emptyset + \{a\} + \{a, b\} + \{a, c\} \\ &\quad + 3\{a, b, c\} + \{b, c\} + \{c\}.\end{aligned}$$

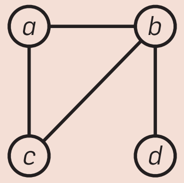
In general, both set families being multiplied may have up to 2^n members, and the same holds for the product. Again the elementary algorithm will consider the monomials pairwise, which requires consideration of $2^n \times 2^n = 4^n$ pairs in the worst case. But analogous to polynomial multiplication, it turns out that we can do considerably better.

Suppose the input set families are f and g. We can view f (and similarly g) as a function that takes an integer value $f(S)$ for each subset $S \subseteq V$ of our n-element vertex set V. (Indeed, let us assume that we have $f(S) = 1$ if and only if the set S is in the family, and $f(S) = 0$ otherwise.) The product, $e = f \cup g$, is then a similar function defined for each $S \subseteq V$ by the rule

$$e(S) = \sum_{A, B \subseteq V : A \cup B = S} f(A)g(B).$$

Since each pair (A, B) contributes by $f(A)g(B)$ to the value of e at exactly $S = A \cup B$, we observe that $\mathcal{O}(4^n)$ multiplications and additions suffice to compute the function e from the given functions f and g, which corresponds to the elementary multiplication algorithm. Now, the analogy to the FFT algorithm

Figure 6. Example for graph coloring.



for multiplying polynomials suggests a different approach, namely to transform the inputs f and g somehow, then multiply pointwise, and finally transform back to the original representation to recover $f \cup g$. The relevant transform turns out to be the *zeta transform* $f\zeta$ of f, defined for all $Y \subseteq V$ by

$$f\zeta(Y) = \sum_{X \subseteq Y} f(X),$$

and its inverse, the *Möbius transform* $f\mu$ of f, defined for all $Y \subseteq V$ by

$$f\mu(Y) = (-1)^{|Y|} \sum_{X \subseteq Y} (-1)^{|X|} f(X).$$

Indeed, the product $f \cup g$ can be computed using the expression

$$f \cup g = ((f\zeta) \times (g\zeta))\mu.$$

Both the zeta transform $f \mapsto f\zeta$ and the Möbius transform $f \mapsto f\mu$ admit fast algorithms analogous to the FFT. Indeed, it follows from the work of Yates[40] (see Knuth[24]) that given f as input, we can compute $f\zeta$ (and similarly $f\mu$) using $\mathcal{O}(2^n n)$ additions and subtractions. This algorithm is perhaps best illustrated in arithmetic circuit form, which Figure 7 illustrates in the case $n = 3$. Observe that each of the n dashed cubes takes the sum along one of the n "dimensions" so that each output $f\zeta(Y)$ ends up taking the sum of all the inputs $f(X)$ with $X \subseteq Y$.

We can thus compute $e = f \cup g$ from f and g given as input using $\mathcal{O}(2^n n)$ additions, negations, and multiplications.

It now follows that we can decide in $\mathcal{O}(2^n nk)$ steps whether a given n-vertex graph G is k-colorable. Indeed, we first compute the characteristic function f of the independent sets of G, that is, for each $S \subseteq V$ we set $f(S) = 1$ if S is independent in G, and $f(S) = 0$ otherwise. Next, we compute the functions e_j for $j = 1, 2, \ldots, k$ by starting with $e_1 = f$ and taking the product $e_j = f \cup e_{j-1}$ for $j \geq 2$. We have that G is k-colorable if and only if $e_k(V) > 0$.

Surprise 3: Hamiltonian Path

Here we illustrate the third surprise, namely a randomized algorithm for the Hamiltonian path problem that runs in time $\mathcal{O}(1.66^n)$. This algorithm is due to Björklund.[4] For ease of exposition, we restrict our consideration to bipartite graphs and obtain running time $\mathcal{O}(1.42^n)$. (The algorithm design here is also slightly different from Björklund's original design; here we rely on reversal of a closed subwalk for cancellation of non-paths[5] and, inspired by Cygan et al.,[8] use the Isolation Lemma in place of polynomial identity testing.)

Let us return to the example in Figure 3. We observe that the graph is bipartite with $n = 8$, $V_1 = \{a, b, c, d\}$, and $V_2 = \{e, f, g, h\}$. As before, our task is to decide whether there exists a Hamiltonian path from vertex s to vertex t. Let us assume that $s = a$ and $t = h$.

Every walk of length $n - 1$ makes exactly n visits to vertices, where exactly $n/2$ visits are to vertices in V_1 because the graph is bipartite. Let us now *label* each of the $n/2$ visits to V_1 using an integer from $L = \{1, 2, \ldots, n/2\}$. In particular, each walk has $(n/2)^{n/2}$ possible labelings, exactly $(n/2)!$ of which are *bijective*, that is, each label is used exactly once. For example, let us consider the labeled walk

$$\underset{1}{a}AeC\underset{3}{b}DfB\underset{4}{a}BfF\underset{2}{c}Hh. \quad (7)$$

We observe that (7) is a bijectively labeled non-path.

Let us now partition the set of all labeled walks into two disjoint classes, the "good" class and the "bad" class. A labeled walk is *good* if the labeling is bijective and the walk is a path. Otherwise a labeled walk is *bad*. We observe that the good class is nonempty if and only if the graph has a Hamiltonian path from s to t.

We now develop a randomized algorithm that decides whether the good class is nonempty. The key idea is to build a sieve for filtering labeled walks so that (a) the bad class is always filtered out and (b) a "witness" from the good class remains with fair probability whenever the good class is nonempty. Conceptually, it will be convenient to regard the sieve as a "bag" (multiset) to which we "hash" labeled walks so that upon termination each "bad" hash value will occur in the bag an even number of times, and each "good" hash value will occur exactly once.

Define the *hash* of a labeled walk to be the multiset that consists of all the elements visited by a walk, together with their labels (if any). For example, the hash value of (7) is

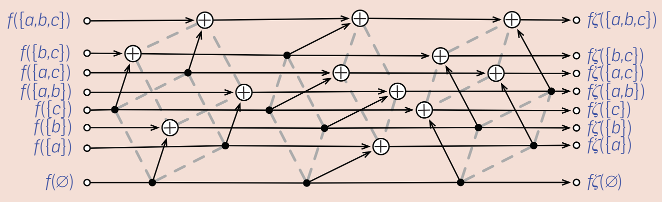
$$\{A, B, B, C, D, F, H, \underset{1}{a}, \underset{4}{a}, \underset{3}{b}, \underset{2}{c}, e, f, f, h\}. \quad (8)$$

In general, we cannot reconstruct a labeled walk from its hash value. However, every bijectively labeled path—that is, every good labeled walk—can be reconstructed from its hash value. Indeed, the vertices in a path are distinct, and the set of edges of a path determines the ordering of the vertices, which we know must start with s and end with t. Thus, each good labeled walk has a unique hash value.

Our next objective is to make sure that each hash value arising from a bad labeled walk gets inserted an even number of times into the sieve. Toward this end, there are two disjoint types of bad labeled walks, namely (a) bijectively labeled non-paths and (b) non-bijectively labeled walks.

Let us consider a bijectively labeled non-path W. We show that W can be paired with a bijectively labeled non-path W' with the same hash value. If we view W as a string, there is a minimal string prefix that contains a repeated vertex. Let us call the last vertex v in such a prefix the *first* repeated vertex in W. Let v be the first repeated vertex in W, and call the subwalk between the

Figure 7. Fast zeta transform for $n = 3$.



first two occurrences of v in W the *first closed subwalk* in W. For example, in (7) the first closed subwalk is $\underset{1}{a}AeC\underset{3}{b}DfB\underset{4}{a}$. There are two cases to consider in setting up the pairing, depending on whether the first repeated vertex in W is in V_1 or in V_2.

If the first repeated vertex is in V_1, let us define W' by transposing the labels of the first and last vertex in the first closed subwalk (that is, the first two occurrences of the first repeated vertex in W). For example, in the case of (7) we obtain

$$\underset{4}{a}AeC\underset{3}{b}DfB\underset{1}{a}BfF\underset{2}{c}Hh. \tag{9}$$

Clearly, W and W' have the same hash value. Furthermore, because W is bijectively labeled, $W' \neq W$. Since $W'' = W$, we have a bijective pairing of bijectively labeled non-walks where the first repeated vertex is in V_1.

If the first repeated vertex is in V_2, let us *reverse* the first closed subwalk (also reversing the labels) in W to obtain the bijectively labeled non-path W'. For example,

$$\underset{1}{a}BfD\underset{3}{b}EgG\underset{4}{c}FfF\underset{2}{c}Hh \tag{10}$$

gets paired with

$$\underset{1}{a}BfF\underset{4}{c}GgE\underset{3}{b}DfF\underset{2}{c}Hh. \tag{11}$$

It is immediate that W and W' have the same hash value. We also observe that $W'' = W$ since two reversals restore the original bijectively labeled non-path. It remains to conclude that $W \neq W'$. Here it is not immediate that reversing the first closed subwalk will result in a different labeled walk. Indeed, the first closed subwalk may be a palindrome, such as $eC\underset{2}{b}Ce$ in

$$\underset{1}{a}AeC\underset{2}{b}CeA\underset{3}{a}BfF\underset{4}{c}Hh. \tag{12}$$

Fortunately, because of bijective labeling, the only possible pitfall is a palindrome of length 5 that starts at V_2, visits a vertex in V_1, and returns to the same vertex in V_2. We can avoid such palindromes by keeping track of the last vertices visited by a partial walk, and hence assume that our labeled walks do not contain such palindromes, and consequently $W' \neq W$. Thus, the set of bijectively labeled non-paths partitions into disjoint pairs $\{W, W'\}$, where each pair has the same hash value.

Next, let us consider a non-bijectively labeled walk W. Each such W *avoids at least one label* from the set of all labels L. In particular, if W avoids exactly a labels, there are exactly 2^a sets $A \subseteq L$ such that W avoids every label in A (and possibly some other labels outside A).

From the previous observations we now obtain the following high-level algorithm. For each subset $A \subseteq L$ in turn, we insert into the sieve the hash value of each labeled walk that avoids every label in A. After all subsets A have been considered, a hash value occurs with odd multiplicity in the sieve if and only if it originates from a good labeled walk.

A second key idea is now to implement the sieve at low level using what is essentially a layer of hashing so that the hash values—such as (8)—are not considered explicitly, but rather *by weight only*. That is, instead of sieving hash values explicitly, we sieve only their weights. In particular, at the start of the algorithm, let us associate an integer weight in the interval $1, 2, \ldots, n(n+1)$ independently and uniformly at random to each of the $(n+1)n/2$ elements that may occur in a hash value. The *weight* of a hash value is the sum of the weights of its elements. When running the sieve, instead of tracking the (partial) walks and their (partial) hash values by dynamic programming, we only track the number of hash values of each weight. This enables us to process each fixed $A \subseteq L$ in time polynomial in n. The number of all sets $A \subseteq L$ is $2^{|L|} \leq 2^{n/2} < 1.42^n$. Thus, the total running time of the above procedure is $\mathcal{O}(1.42^n)$. When the sieve terminates, we assert that the input graph has a Hamiltonian path if the counter for the number of hash values of at least one weight is odd; otherwise we assert that the graph has no Hamiltonian path.

To see that the presence of an odd counter implies the existence of a Hamiltonian path, observe that by our careful design, each bad hash value gets inserted into the sieve an even number of times, and in particular contributes an even increment to the counter corresponding to the weight of the hash value. Thus, an odd counter can arise only if a good hash value was inserted into the sieve, that is, the graph has a Hamiltonian path.

Next, let us study the probability of a false negative, that is, all counters are even although the graph has a Hamiltonian path. Here it suffices to invoke the "Isolation Lemma" of Mulmuley et al.[30] which states that for any set family over a base set of m elements, if we assign a weight independently and uniformly at random from $1, 2, \ldots, r$ to each element of the base set, there will be a unique set of the minimum weight in the family with probability at least $1 - m/r$. In particular, if we consider the set family of good hash values—indeed, each good hash value is a set—there is a unique such hash value of the minimum weight—and hence an odd counter in the sieve—with probability at least 1/2.

We thus have a randomized algorithm for detecting Hamiltonian paths in bipartite graphs that runs in time $\mathcal{O}(1.42^n)$, gives no false positives, and gives a false negative with probability at most 1/2. (The algorithm could now be extended to graphs that are not bipartite with running time $\mathcal{O}(1.66^n)$ by partitioning the vertices randomly into V_1 and V_2 and employing a bijective labeling also for the edges with both ends in V_2.)

Conclusion

This article has highlighted three recent results in exact exponential algorithms, with the aim of illustrating the range of techniques that can be employed and the element of surprise in each case. In this regard, it is perhaps safe to say that the area is still in a state of flux, and with more research one can expect more positive surprises. Certainly, the authors do not mind to be labeled as optimists in this sense. We also hope the three highlighted results have illustrated perhaps the main reason why one wants to study algorithms that run in exponential time. That is, the study of exponential time algorithms is really a quest for understanding computation and the structure of computational problems, including pursuing the sometimes surprising connections uncovered in such a quest.

We conclude with three challenge problems, each of which at first sight appears quite similar to one of the three surprises we have covered in this article. Frustratingly enough, however, there has been no progress at all on these problems.

MAX-3-SAT. We have seen that MAX-2-SAT can be solved in time $\mathcal{O}(2^{\omega n/3})$ essentially because of the existence of nontrivial algorithms for matrix multiplication. But no such tools are available when one considers instances with clauses of length 3 instead of length 2. The challenge is to find an algorithm that runs in time $\mathcal{O}((2-\epsilon)^n)$ for MAX-3-SAT, where n is the number of variables and $\epsilon > 0$ is a constant independent of n.

Edge Coloring. The *edge-coloring* problem asks us to color the edges of a graph using the minimum number of colors such that the coloring is *proper*, that is, any two edges that share an endvertex must receive different colors. It is known that the number of colors required is either Δ or $\Delta + 1$, where Δ is the maximum degree of a vertex, and it is *NP*-complete to decide which of the two cases occurs.[20] For a graph G, the edge-coloring of G is equivalent to deciding whether the chromatic number of the line graph $L(G)$ of G is Δ or $\Delta + 1$, which implies that edge-coloring can be solved in time $2^m m^{\mathcal{O}(1)}$, where m is the number of edges in G. The challenge is to find an algorithm that runs in time $\mathcal{O}((2-\epsilon)^m)$ where $\epsilon > 0$ is independent of m.

Traveling Salesman. While the Hamiltonian cycle problem can be solved in randomized time $\mathcal{O}(1.66^n)$, no such algorithm is known for the Traveling Salesman Problem with n cities and travel costs between cities that are nonnegative integers whose binary representation is bounded in length by a polynomial in n. The challenge is to find an algorithm that runs in time $\mathcal{O}((2-\epsilon)^n)$ where $\epsilon > 0$ is independent of n.

Further Reading

Beyond the highlighted results in this article, the recent book of Fomin and Kratsch[15] and the surveys of Woeginger[38, 39] provide a more in-depth introduction to exact exponential algorithms. Dantsin and Hirsch[9] survey algorithms for SAT, while Malik and Zhang[28] discuss the deployment of SAT solvers in practical applications. Husfeldt[21] gives an introduction to applications of the principle of inclusion and exclusion in algorithmics. Flum and Grohe[13] give an introduction to parameterized complexity theory and its connections to subexponential and exponential time complexity. Williams[37] relates improvements to exhaustive search with superpolynomial lower bounds in circuit complexity.

Acknowledgments
The authors would like to thank Andreas Björklund, Thore Husfeldt, Mikko Koivisto, and Dieter Kratsch for their comments that greatly helped to improve the exposition in this review. F.V.F. acknowledges the support of the European Research Council (ERC), grant Rigorous Theory of Preprocessing, reference 267959. P.K. acknowledges the support of the Academy of Finland, Grants 252083 and 256287. C

References

1. Bax, E.T. Inclusion and exclusion algorithm for the Hamiltonian path problem. *Inf. Process. Lett. 47*, 4 (1993), 203–207.
2. Bellman, R. Dynamic Programming, Princeton University Press, 1957.
3. Bellman, R. Dynamic programming treatment of the travelling salesman problem. *J. ACM 9* (1962), 61–63.
4. Björklund, A. Determinant sums for undirected hamiltonicity. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)* (2010), IEEE, 173–182.
5. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M. Narrow sieves for parameterized paths and packings. arXiv:1007.1161 (2010).
6. Björklund, A., Husfeldt, T., Koivisto, M. Set partitioning via inclusion–exclusion. *SIAM J. Comput. 39*, 2 (2009), 546–563.
7. Coppersmith, D., Winograd, S. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput. 9*, 3 (1990), 251–280.
8. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J.M.M., Wojtaszczyk, J.O. Solving connectivity problems parameterized by treewidth in single exponential time. In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science* (2011), IEEE, 150–159.
9. Dantsin, E., Hirsch, E.A. Worst-case upper bounds. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, 403–424.
10. Downey, R.G., Fellows, M.R. Parameterized Complexity, Springer, 1999.
11. Eppstein, D. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms Appl. 7*, 2 (2003), 131–140.
12. Eppstein, D. Quasiconvex analysis of multivariate recurrence equations for backtracking algorithms. *ACM Trans. Algorithms 2*, 4 (2006), 492–509.
13. Flum, J., Grohe, M. Parameterized Complexity Theory, Springer, 2006.
14. Fomin, F.V., Grandoni, F., Kratsch, D. A measure & conquer approach for the analysis of exact algorithms. *J. ACM 56*, 5 (2009).
15. Fomin, F.V., Kratsch, D. Exact Exponential Algorithms, Springer, 2010.
16. Fortnow, L. The status of the *P* versus *NP* problem. *Commun. ACM 52*, 9 (2009), 78–86.
17. Garey, M.R., Johnson, D.S. Computers and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.
18. Golomb, S.W., Baumert, L.D. Backtrack programming. *J. ACM 12* (1965), 516–524.
19. Held, M., Karp, R.M. A dynamic programming approach to sequencing problems. *J. Soc. Indust. Appl. Math. 10* (1962), 196–210.
20. Holyer, I. The NP-completeness of edge-coloring. *SIAM J. Comput. 10*, 4 (1981), 718–720.
21. Husfeldt, T. Invitation to algorithmic uses of inclusion–exclusion. arXiv:1105.2942 (2011).
22. Itai, A., Rodeh, M. Finding a minimum circuit in a graph. *SIAM J. Comput. 7*, 4 (1978), 413–423.
23. Karp, R.M. Dynamic programming meets the principle of inclusion and exclusion. *Oper. Res. Lett. 1*, 2 (1982), 49–51.
24. Knuth, D.E. The Art of Computer Programming, vol. 2: Seminumerical Algorithms, 3rd edn, Addison-Wesley, 1998.
25. Kohn, S., Gottlieb, A., Kohn, M. A generating function approach to the traveling salesman problem. In *Proceedings of the ACM Annual Conference (ACM 1977)* (1977), ACM Press, 294–300.
26. Koivisto, M. Optimal 2-constraint satisfaction via sum-product algorithms. *Inform. Process. Lett. 98*, 1 (2006), 24–28.
27. Lawler, E.L. A note on the complexity of the chromatic number problem. Inf. Process. Lett. 5, 3 (1976), 66–67.
28. Malik, S., Zhang, L. Boolean satisfiability: From theoretical hardness to practical success. *Commun. ACM 52*, 8 (2009), 76–82.
29. Moon, J.W., Moser, L. On cliques in graphs. *Israel J. Math. 3* (1965), 23–28.
30. Mulmuley, K., Vazirani, U.V., Vazirani, V.V. Matching is as easy as matrix inversion. *Combinatorica 7*, 1 (1987), 105–113.
31. Niedermeier, R. Invitation to Fixed-Parameter Algorithms, Oxford University Press, 2006.
32. Strassen, V. Gaussian elimination is not optimal. *Numer. Math. 13* (1969), 354–356.
33. Vassilevska Williams, V. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of 44th ACM Symposium on Theory of Computing (STOC 2012)* (2012), ACM, 887–898.
34. Vazirani, V.V. Approximation Algorithms, Springer, 2001.
35. Walker, R.J. An enumerative technique for a class of combinatorial problems. In *Proceedings of Symposia in Applied Mathematics*, vol. 10, American Mathematical Society, 1960, 91–94.
36. Williams, R. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoret. Comput. Sci. 348*, 2–3 (2005), 357–365.
37. Williams, R. Improving exhaustive search implies superpolynomial lower bounds. In *Proceedings of 42nd ACM Symposium on Theory of Computing* (2010), ACM, 231–240.
38. Woeginger, G. Exact algorithms for NP-hard problems: a survey. In *Combinatorial Optimization – Eureka, You Shrink!* (2003), volume 2570 of *Lecture Notes in Computer Science*, Springer, 185–207.
39. Woeginger, G. Space and time complexity of exact algorithms: some open problems. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation* (2004), volume 3162 of *Lecture Notes in Computer Science*, Springer, 281–290.
40. Yates, F. The Design and Analysis of Factorial Experiments, Imperial Bureau of Soil Science, 1937.
41. Zykov, A.A. On some properties of linear complexes. *Mat. Sbornik N.S. 24*, 66 (1949), 163–188.

Fedor V. Fomin (fomin@ii.uib.no) is a professor in the Institutt for Informatikk, University of Bergen, Norway.

Petteri Kaski (petteri.kaski@aalto.fi) is an Academic Research Fellow in the Department of Information and Computer Science at Aalto University, Aalto, Finland.