Fast Parameterized Preprocessing for Polynomial-Time Solvable Graph Problems

The challenge of transforming polynomial-time algorithms to really efficient ones

Anne-Sophie Himmel
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany

George B. Mertzios
Department of Computer Science, Durham University, UK
george.mertzios@durham.ac.uk

André Nichterlein
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany
andre.nichterlein@tu-berlin.de

Rolf Niedermeier
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany

Introduction and Motivation

A holy grail of theoretical computer science, with numerous fundamental implications to more applied areas of computing such as operations research and artificial intelligence, is the question of whether NP is equal to P. Much of modern technology is based on the so-called Cobham-Edmonds' thesis (named after Alan Cobham and Jack Edmonds) which states that algorithmic problems can be feasibly computed on a computational device only if they can be computed in polynomial time. In a nutshell: P means "feasible" while NP-hard means "infeasible" to compute.

Moving from theory to practice, and looking at problems more realistically, the size of the exponent in a polynomial-time algorithm matters a lot. In the era of big data, when n is, say, the number of users of Facebook or Twitter, an $\Theta(n^3)$- or even $\Theta(n^2)$-time algorithm might be too slow. In such applications where the input size n is very large, any practically efficient algorithm that solves a problem to optimality can only afford a linear or almost linear-time complexity. While the distinction between computationally infeasible and feasible problems has classically been "NP-hard vs. polynomial-time solvable", in the big data era it becomes "polynomial-time vs. (quasi-)linear-time solvable".

Parameterized algorithmics for polynomial problems.

When a fast algorithm (polynomial) is not fast enough (i. e., not linear) and when plausible relative lower bounds speak against further speedups for important algorithmic problems, what can one do?[1] For NP-hard (traditionally "computationally infeasible") problems, there are several established coping strategies, including heuristics (fingers crossed that a worst case does not appear), approximation (polynomial-time algorithms that provide approximate instead of optimal solutions), and parameterized algorithmics (be fast if the input instance carries a small parameter value); see Roughgarden [36] for an overview. With the advent of big data it has become apparent that we do not only face "speed issues" for NP-hard problems, but also for problems solvable in polynomial time. For instance, the classic Cocke-Younger-Kasami algorithm for parsing context-free languages is not used in practice due to its cubic running time, thus resorting to constrained context-free languages and corresponding linear-time parsing algorithms such as LL- and LR-parsing. Indeed, with the recent advent of fine-grained complexity analysis [10,38] we have clear indications that some of our beloved polynomial-time algorithms cannot be further improved without implying unlikely consequences in computational complexity theory. Prominent examples in this direction include all-pairs shortest paths [39], Fréchet distance [9], and longest common subsequence in strings [11].

Two recent trends in theoretical computer science towards addressing this challenge are again approximation algorithms and parameterized algorithms. For instance, Duan and Pettie [16] developed a linear-time algorithm for computing near-maximum weight matchings. Iwata et al. [26] developed an $O(k(m \log n + n \log^2 n))$-time[2] exact[3] parameterized algorithm for computing maximum-weight matchings, where k is the treewidth of the given graph. Notably, while no quasi-linear-time algorithm is known to compute maximum-weight matchings in the general case, the mentioned algorithms achieve such a running time in relaxed settings (approximate solutions or input graphs with small parameter values). In what follows, we want to review and discuss possibilities of a parameterized view on bypassing established complexity barriers for polynomial-time solvable problems. More specifically, we focus on the power of (linear-time) parameterized preprocessing in combination with a rigorous mathematical analysis, which may guide algorithm design. To this end, we present, illustrate, and discuss three basic concepts of parameterized preprocessing in the context of polynomial-time solvable problems.

Three directions of parameterized preprocessing.

Roughly speaking, our focus is on decreasing the algorithmic complexity by processing the input without giving up the possibility to find an optimal solution for the

[1] While all our examples will relate to graph algorithms, our general message certainly is not limited to these.

[2] Here and subsequently in this work, n denotes the number $|V|$ of vertices and m denotes the number $|E|$ of edges in the input graph $G = (V, E)$, respectively.

[3] By an exact algorithm we understand one that solves an optimization problem to optimality.

problem for any input instance. That is, we exclude approximation algorithms or considerations in the direction of approximation-preserving preprocessing. We identify and delineate three fundamental approaches for parameterized preprocessing which helped to gain significant (practical) improvements in algorithmic performance over previously existing approaches. The basic common aspects of these three preprocessing variants are, on the one hand, their focus on high efficiency of preprocessing (typically linear time) and, on the other hand, their use of appropriate problem-specific parameters (that is, some numbers measuring specific, algorithmically crucial, features of the input) to obtain an overall efficient algorithm with mathematically provable performance guarantees. With the vision of systematizing and unifying existing ideas and approaches, we use in the remainder of our article illustrative examples from graph algorithmics to delve deeper into the following three fundamental directions of parameterized preprocessing. In the following, we abstain from a discussion of the "art of problem parameterization"—while some parameterizations are more or less obvious, others (including above-guarantee parameterization or distance-from-triviality parameterization) leave quite some room for creative explorations [34, Chapter 5] [35].

- *Parameterized size reduction.* This approach originally stems from parameterized algorithmics for NP-hard problems and is known as *kernelization*. The point here is to efficiently preprocess the input by trying to identify and remove unnecessary or nonmeaningful parts, thus hopefully shrinking the input size considerably. To this end, one replaces the input instance by an "equivalent" one whose size can be upper-bounded by a hopefully small function of the chosen parameter only. In this way, one may obtain a rigorous way of provably effective preprocessing. Moreover, this is a pretty universal approach in the sense that the resulting reduced instance, also known as "(problem) kernel", can be typically attacked with any known solving strategy (such as exact, approximation, or heuristic algorithms). We exemplify this preprocessing approach in the context of computing a maximum-cardinality matching in undirected graphs, using the parameter feedback edge number of the graph.

- *Parameterized structure simplification.* This approach is perhaps the most natural alternative to kernelization: instead of shrinking the *size* of the input, simplify its *structure*. That is, we replace the original input with an equivalent one having a simpler structure, which we can then algorithmically exploit. Here "simpler" typically means that, apart from some few bad substructures, the rest of the input has nice properties that allow for an efficient algorithm to be applied. The fewer these bad substructures in this simplified (but equivalent) instance, the more efficient this algorithm will be. The chosen parameter comes naturally into play: we demand the number of bad substructures in the simplified instance (which is the only cause of inefficiency) to be upper-bounded by a function of the parameter only. We exemplify the structure simplification approach in the context of computing longest paths in interval graphs, using a very natural parameter.

- *Parameterized data structure design.* This approach is sort of closest to classic approaches of preprocessing and has been extensively used for "query-based" problems. The scenario is that we have an input on which many queries are expected to be asked later on. The approach here is to first design and implement an auxiliary data structure by preprocessing the given fixed input, and then to have an algorithm that quickly answers each query by operating exclusively on this auxiliary data structure. Here, the parameter comes into play in a more subtle way by measuring favorable properties of the data structure. We exemplify this preprocessing strategy in the context of computing shortest paths in edge-weighted, undirected graphs, using the parameter treewidth which measures how close an input graph is to being a tree.

In what follows, we discuss the above-described three directions in more detail.

Parameterized size reduction (kernelization)

It is fair to say that the method of kernelization is the practically most relevant method developed in parameterized algorithmics for NP-hard problems [17]. Here, we want to advocate kernelization in the context of polynomial-time solvable problems, an issue that has been rarely touched so far but opens a number of theoretically and practically promising research and application avenues.

Recall that a kernelization algorithm is a preprocessing algorithm that applies a set of *data reduction rules* to reduce a large problem instance to an equivalent smaller instance—a so-called *kernel*. We call an instance resulting from the application of a data reduction rule a *reduced instance*. The kernel can be seen as the final reduced instance, in which no data reduction rule can be applied any more. On this kernel, one can use any known exact, approximation, or heuristic algorithm to solve the problem. Recently, data reduction rules received a lot of attention in practice as well [21, 24].

Data reduction rules are evaluated in terms of *correctness*, *efficiency*, and *effectiveness*. A data reduction rule is correct if any optimal solution of the reduced instance can be extended to an optimal solution of the original input instance. Correctness is a necessary requirement for all data reduction rules used in kernelization. It is efficient if it can be applied much "faster" than the algorithm solving the problem instance, while it is effective if the size of the kernel is "smaller" than the original instance size. While correctness and efficiency are analyzed for each data reduction rule separately, the effectiveness is analyzed in the context of (typically) exhaustively applying all data reduction rules.

Our key illustrating example here is Maximum-Cardinality Matching: given an undirected graph $G = (V, E)$, the task is to find a *maximum matching*, that is, a maximum-cardinality edge subset $M \subseteq E$ such that no two edges in M share an endpoint. The (theoretically) fastest algorithm for this problem runs in $O(\sqrt{n}m)$ time (see Blum [6] and the discussion therein).

The kernelization algorithm for Maximum-Cardinality Matching we consider consists of two simple data reduction rules already considered by Karp and Sipser in the early 1980's [27]. The first rule removes degree-one vertices from the input graph (see also Fig. 1):

Data Reduction Rule 1. *Let v be a degree-one vertex with neighbor u. Then select the edge $\{u, v\}$ for the maximum matching and delete u and v.*

First, let us convince ourselves that the data reduction rule is correct. A maximum matching for the initial graph G

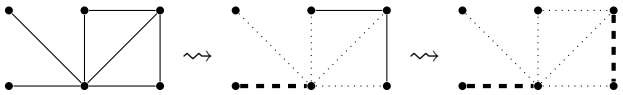


Figure 1: Left: Input graph. Middle/Right: The graph after applying Data Reduction Rule 1 once/twice. Bold dashed edges are remembered as contained in the matching. Dotted and dashed edges are removed.

Figure 2: Left: Input graph where the edge $\{u, w\}$ may or may not exist. Right: The graph after applying Data Reduction Rule 2 (folding v). Bold edges indicate one possible matching in each side.

can have at most one more edge than a maximum matching for the reduced graph G'. Given a maximum matching M' for G', we can construct a matching $M := M' \cup \{u, v\}$ for G. Thus, M is a maximum matching.

Second, as to efficiency, it is straightforward to implement the rule so that it can be exhaustively applied in linear time.

As mentioned above, the effectiveness shall be analyzed in combination with the second data reduction rule that removes degree-two vertices from the input graph. To this end, *folding* a degree-two vertex v means to delete v and its two neighbors and introduce a new vertex adjacent to all vertices at distance two from v (see Fig. 2).

Data Reduction Rule 2. *Let v be a degree-two vertex. Then fold v.*

The idea is as follows: in a maximum matching we can always match v with either u or w. Which of these two cases applies is not easy to decide in advance. However, if we have a matching where at least one of u and w is not matched, then the choice is trivial: match v with (one of) the non-matched neighbors. In any case, since Data Reduction Rule 2 deletes both vertices u and w, it ensures that at most one of the edges incident to u and w can be in a matching of the resulting graph. Exhaustively applying Data Reduction Rule 2 in linear time is actually non-trivial but doable [2].

Now we analyze the effectiveness of the two data reduction rules, that is, the size of the final reduced instance after applying them exhaustively. To this end, we make use of the kernelization concept from the toolbox of parameterized algorithmics and, to keep things simple, we only consider a very basic and typically large parameter: In the worst case, the graph does not have any degree-one or degree-two vertices and the reduction rules are not applicable. If, however, the graph is very tree-like in the beginning, namely, there are only a small number of edges to be removed such that the graph becomes a tree, then in the intermediate reduced instances there will be many degree-one and/or degree-two vertices and, thus, the number of vertices in the resulting kernel will be small. Next, we formalize the statement [33]; herein, the *feedback edge number* τ denotes the minimum number of edges one has to remove from a connected graph to obtain a tree. Exhaustively applying Data Reduction Rules 1 and 2 on graph G produces a reduced graph G' such that:

1. The graph G' contains $2 \cdot \tau$ vertices, where τ is the feedback edge number of G, and

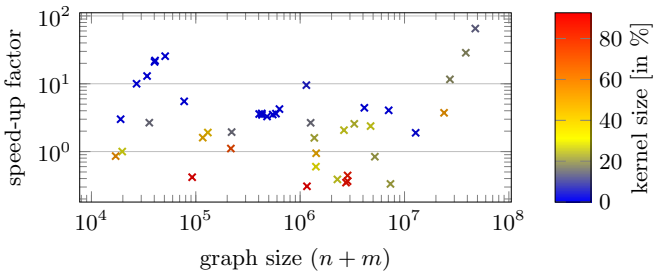


Figure 3: Speed-up factors obtained by using Data Reduction Rules 1 and 2 in a preprocessing phase before Kececioglu and Pecqueur's [28] algorithm on graphs from the SNAP [30] data set collection. A speed-up factor below one is a slowdown. The colors indicate the kernel size, that is, the percentage of vertices and edges of the orginal graph *not* removed by the data reduction rules.

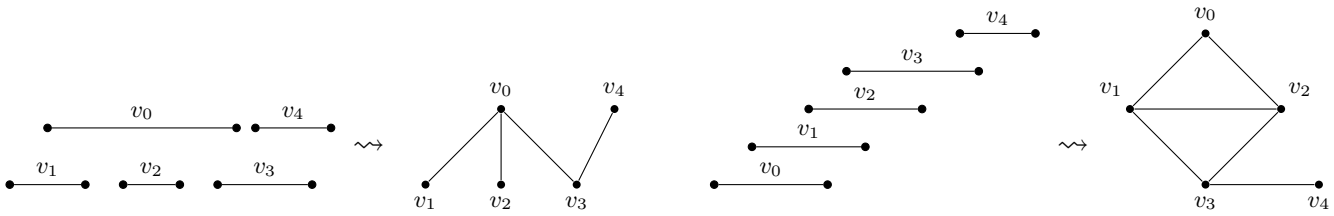
2. given a maximum-cardinality matching M' for G', one can compute in linear time a maximum-cardinality matching M for G.

The rough argument for the upper bound $O(\tau)$ on the number of vertices of the resulting graph G' is as follows: The minimum vertex degree in G' is three. As none of the two reduction rules introduces new cycles, G' has still feedback edge number at most τ. Thus, removing τ edges results in a tree $T_{G'}$ (or a forest) in which at most 2τ vertices of degree one and two exist. Since $T_{G'}$ is a tree, it follows by a simple counting argument that it has at most 2τ vertices of degree at least three. Hence, G' contains at most 4τ vertices.

Besides this theoretical measure for the effectiveness of Data Reduction Rules 1 and 2, there is also strong experimental evidence [29]: The probably still fastest practical algorithm, due to Kececioglu and Pecqueur [28], was tested on real-world graphs from the SNAP [30] large network data set collection with 10^4 to 10^8 edges, once with and once without exhaustively applying the two data reduction rules before running their algorithm. The effect of the preprocessing ranged from a slow down by a factor 3 to up to 60 times faster. On average the speed-up factor was 4.7 (see Fig. 3 for some details).

These experimental results reflect the above theoretical prediction as follows: For "bad" instances (all of these have high parameter value) the data reduction rules do not reduce the input size by much; however, since these rules are relatively easy, it does not take much time to apply them. In contrast, for "good" instances (with small parameter value) where most of the graph is reduced by the rules, the overall algorithmic speed-up is pronounced. As the subsequent algorithm that we apply to the kernel has usually a non-linear running time, the exhaustive application of the data reduction rules in linear time basically comes for free in terms of the overall algorithmic complexity, for both "good" and "bad" instances. Notably, for our above example, Kececioglu and Pecqueur [28] state a worst-case running time of $O(nm \cdot \alpha(n, m))$ for their algorithm, where α is the inverse of Ackermann's function. This is worse than the theoretically best bound of $O(\sqrt{n}m)$ [6], but in practice Kececioglu and Pecqueur's algorithm is still state of the art. Altogether, we conclude that even moderately shrinking the input has a significant impact on the running time.

To summarize, the above simple kernelization algorithm can significantly improve even a sophisticated matching im-



(a) An interval graph.

(b) A proper interval graph.

Figure 4: Interval and proper interval graphs.

plementation on real-world graphs with many low-degree vertices. Of course, reduction rules are used for many further problems, including linear programs or SAT solvers [5, 23]. Data reduction rules that can be exhaustively applied in time linear in the input size of the original instance are particularly attractive: even if there is no effect on the input instance, then there is no large penalty in the overall running time of solving the problem at hand.

Overall, kernelization is a very universal and flexible approach interesting for theory and practice: the parameterized framework allows us to get theoretical guarantees that yield good predictions on the efficiency and effectiveness of data reduction rules in practice.

Parameterized structure simplification

To gain fast algorithms, a key approach is to exploit some specific structural property of the input. However, in practice, the input often does not directly have such a special property that we can use, which makes this approach often not applicable in real-life scenarios. Indeed, detecting such useful properties can be an algorithmic challenge on its own. The idea we put forward here is similar to the kernelization approach in that we also rely on exhaustively applying the rules that modify the input instance. The difference is that, using a problem-specific parameterization, we *simplify* the structure of the input instead of shrinking its size; thus we call these rules *data simplification rules*. Once this preprocessing phase is done, we can then use another algorithm which is efficient on inputs having the desired special property. In this algorithmic scheme, the preprocessing approach helps to constrain the search space (using a function of the parameter) for a subsequent algorithm, thus leading to overall faster algorithms in the case of small parameter values.

Our key illustrating example for the structure simplification approach is Longest Path on Interval Graphs: given an undirected *interval* graph G, the task is to find a *longest path*, that is, a path in G whose length is as large as possible; if every vertex of G additionally has a positive weight assigned to it, then the task becomes to compute a path having the largest total weight on its vertices. Since this is a fundamental NP-hard problem on general graphs, which directly generalizes the famous Hamiltonian Path problem, researchers have tried to identify "islands of tractability", that is, appropriate graph families $\mathcal{F}$ such that, for every graph $G \in \mathcal{F}$ with n vertices, a longest path in G can be computed in time that is polynomial in n. One of the well-known graph families $\mathcal{F}$ for which this is possible is the class of *interval* graphs, where a longest path can be computed by a dynamic programming algorithm with running time $O(n^4)$ [25]. Although this running time is polynomial, the degree of the polynomial is too high for being practical when the number n of vertices becomes large; however, to date this remains the fastest known algorithm for the problem. We will see that some acceleration can be achieved if the interval graph is "close" to the special case of being a *proper* interval graph.

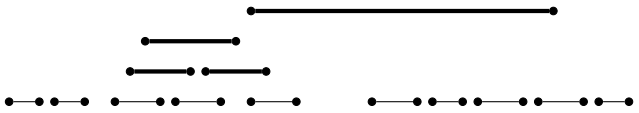


Figure 5: An interval graph G whose vertex cover number is four (see the four thick intervals).

A graph $G = (V, E)$ is an *interval* graph if one can assign to every vertex $v \in V$ exactly one closed interval I_v on the real line $\mathbb{R}$ such that u is adjacent to v if and only if $I_u \cap I_v \neq \emptyset$ (see Fig. 4a). If such a collection of intervals exists such that none of them is properly included in another one, then G is called a *proper interval* graph (see Fig. 4b). Interval graphs have received a lot of algorithmic attention due to their applicability in DNA physical mapping, genetics, molecular biology and scheduling problems [19] [20, Chapter 8.4].

By closely analyzing the above $O(n^4)$-time dynamic programming algorithm [25], it turns out that a very slight refinement of it runs in $O(\ell^3 n)$ time whenever in the input interval graph G has at least $n - \ell$ vertices that are *independent*, i. e., pairwise non-adjacent; that is, if the intervals associated with these $n - \ell$ vertices are pairwise non-intersecting [18]. In such a case, we say that the parameter *vertex cover number* of the graph is upper-bounded by ℓ (see Fig. 5): the vertex cover number of a graph $G = (V, E)$ is the size of the smallest vertex subset $S \subseteq V$ such that every edge is incident with at least one vertex of S. We can conclude the following.

Fact 1. *For an interval graph G with vertex cover number ℓ, a longest path in G can be computed in $O(\ell^3 n)$ time.*

Moreover, the same refined algorithm correctly computes a longest path of G, even if G has positive weights on its vertices. It is worth noting here that, since we always have that $\ell \leq n$, the running time $O(\ell^3 n)$ becomes $O(n^4)$ in the worst case, and thus it is asymptotically *tight* with respect to the currently best-known algorithm for general interval graphs.

The main idea for our illustrating example of structure simplification lies in the fact that, after appropriately preprocessing the input interval graph G (which has a potentially *very large* vertex cover number, e. g., $\Theta(n)$), we compute a new graph G' with *small* vertex cover number; then we can apply the $O(\ell^3 n)$-time algorithm to G' to compute

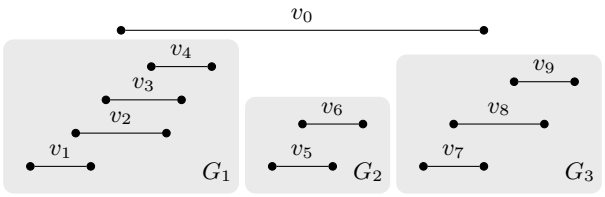


Figure 6: A (non-proper) interval graph G with only one long interval I_{v_0} which properly contains some other intervals. All other intervals (without I_{v_0}) induce three connected proper interval graphs G_1, G_2, G_3, each having a Hamiltonian path. To compute a longest path we just need to detect the two largest graphs among G_1, G_2, G_3 (here G_1 and G_3); then "glue" their Hamiltonian paths together using the long interval I_{v_0} as a "connector". In this case the longest path is $(v_1, v_2, v_3, v_4, v_0, v_7, v_8, v_9)$ and contains eight vertices.

a longest path in both G and G', as we describe below. That is, the desired "special property" after the preprocessing phase is "having small vertex cover number".

The prepocessing uses the fact that a longest path in a proper interval graph can be *trivially* computed in linear time. Indeed, due to the monotone arrangement of the intervals, every connected proper interval graph has a Hamiltonian path, i. e., a path that traverses each vertex exactly once [4]. Such a Hamiltonian path can be found by traversing all intervals from left to right, and thus a longest path of a proper interval graph can be computed by just finding its largest connected component.

Fact 2. *For a proper interval graph G with n vertices, a longest path in G can be computed in $O(n)$ time.*

Consequently, although interval and proper interval graphs are superficially similar, they appear to behave very differently when trying to compute a longest path in them.

What makes it substantially more difficult to compute a longest path in an interval graph? The intuition for this is illustrated by a simple example in Fig. 6.In this figure the only source of complication comes from the fact that there is one long interval I_{v_0} and we need to decide which of the smaller sub-graphs G_1, G_2, G_3 (each of them being proper interval) will connect their Hamiltonian paths using I_{v_0} as a "connector" interval. The situation becomes even more complicated when we have more "long" intervals and more proper interval sub-graphs, all of which can be interconnected in many different ways. The fastest known way to deal with such cases is to recursively check the appropriate points where these connector intervals (i. e., the long ones) connect the parts of the various proper interval sub-graphs in order to combine them appropriately to build a longest path.

The above discussion naturally leads to the appropriate parameter that measures the "backdoor" [40] or "distance to triviality" [22] for the problem Longest Path on Interval Graphs: the size k of a *minimum proper interval (vertex) deletion set*, i. e., the minimum number k of vertices that we have to delete from an interval graph G to obtain a proper interval graph [18]. As it turns out, given an interval graph G, a proper interval deletion set S of size at most $4k$ can be computed in linear time by appropriately traversing the intervals from left to right [18].

Based on this distance-to-triviality parameter, Giannopoulou et al. [18] provided a polynomial fixed-parameter algorithm for interval graphs that computes a

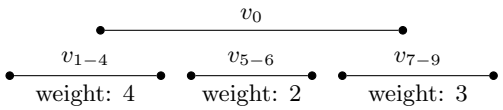


Figure 7: The weighted interval graph resulting from applying the first data simplification rule on the graph of Fig. 6. The new weighted intervals v_{1-4}, v_{5-6}, and v_{7-9} replaced the sets of intervals $\{v_1, v_2, v_3, v_4\}$, $\{v_5, v_6\}$, and $\{v_7, v_8, v_9\}$ and took weights 4, 2, and 3, respectively.

longest path in $O(k^9 n)$ time, implying linear time for constant values of k. In summary, the main idea of this algorithm is as follows:

1. Exhaustively apply two data simplification rules in total $O(kn)$ time to obtain an interval graph G' (with appropriate vertex weights) that has vertex cover size $\ell = O(k^3)$.
2. Apply to G' the algorithm behind Fact 1 with running time $O(\ell^3 n) = O(k^9 n)$ to compute a longest path of G', and thus at the same time also a longest path of G.

The details of the two data simplification rules of this preprocessing phase are quite delicate; therefore, we focus here on highlighting the main ideas behind them. The first simplification rule deals with specific sequences of intervals of G that have the "all-or-none" property; that is, a longest path contains either all intervals of the sequence or none of them; note that this property generalizes the idea behind Fact 2. In each of these sequences, the intervals form a proper interval sub-graph. Once we have identified these sequences, we replace each one of them with one single interval having an integer weight equal to the number of intervals of the sequence. The important structural property of these new weighted intervals is that they are mutually non-intersecting (i. e., they are independent), and thus they cannot be connected to each other directly in any path (see Fig. 7 for a visualization). That is, in contrast to the kernelization approach, here by exhaustively applying this first simplification rule we do not upper-bound the *number* of these new intervals (in fact, there can be $O(n)$ many), but we simplify their *structure* instead. Moreover, as these new intervals are mutually non-intersecting, all other intervals form a *vertex cover* of the graph, see Fig. 5.

The second data simplification rule proceeds as follows. The (at most) $4k$ intervals of the proper interval deletion set (which we have already computed in linear time) divide the real line $\mathbb{R}$ into at most $4k + 1$ disjoint parts. For each of the $O(k^2)$ pairs of these disjoint parts of $\mathbb{R}$, we consider the intervals of the graph that have their left endpoint in the one part and their right endpoint in the other one. As it turns out, we can just replace all of these (potentially $O(n)$ many) intervals with few new intervals (in fact, $O(k)$ many), each of them having an appropriate positive weight. Thus, we have in total $O(k^3)$ new weighted intervals after exhaustively applying the second simplification rule. This in turn implies that the vertex cover of the resulting (weighted) interval graph is $\ell = O(k^3)$. Thus we can now apply the refined dynamic-programming algorithm, obtaining overall an algorithm that computes a longest path of the input interval graph in $O(\ell^3 n) = O(k^9 n)$ time in total.

In summary, structure simplification is a natural "sister approach" to kernelization in the sense of focusing on cre-

ating nice structure instead of merely shrinking size in the reduced instance. Both however, make decisive use of appropriate problem-specific parameters (with hopefully instance-specific small values) in order to perform a rigorous mathematical analysis providing theoretical (worst-case) guarantees. As to structure simplification, we also point to the concept of treewidth reduction used in the context of NP-hard problems [32]; treewidth will also be a key concept in the main illustrating example for the third direction of parameterized preprocessing which comes next.

Parameterized data structure design

A classic approach to faster algorithms is to first build up appropriate clever data structures, and then to design algorithms that exploit these data structures to solve the target problem. Correspondingly, fundamental data structures such as heaps are discussed in every basic textbook on algorithms and usually have very well understood guarantees. Our goal here is to exhibit a strategy based on classic preprocessing (i. e., by first building up a helpful data structure) in conjunction with a parameterized analysis of the computational complexity.

Our key illustrating example here is the computation of shortest paths, known to be solvable in $O(n \log n + m)$ time using Dijkstra's classic algorithm. This running time is fast, however, sometimes not fast enough when used in modern applications such as real-time route planning, where n is huge (e. g., all cities in the US). We describe an approach that efficiently computes shortest-path queries using tree-decompositions—a data structure originating from algorithmic graph theory that is frequently used in parameterized algorithmics to cope with NP-hard problems. The approach to be described will not provide good worst-case guarantees in general; however, a parameterized analysis using treewidth [1, 12], combined with the fact that many infrastructure networks have small treewidth [31], shows its benefits. Indeed, the described ideas are used in practice on quite large data sets, for example in Microsoft's Bing Maps [3, 14].

In our setting, the input graph G is a static, edge-weighted graph representing, e. g., a road network. The task is to quickly report the *distance*—the length of a shortest path—between different pairs of vertices. To this end, we will *compile* G, that is, we build in a preprocessing step a data structure on which the subsequent algorithm can quickly answer distance queries between any pair of vertices of G. We remark that the approach allows for queries reporting the shortest path and fast weight updates without compiling G repeatedly [3, 14].

We first present three simple facts that are exploited in the approach. Afterwards, we give a high-level description of the concepts of tree-decomposition and treewidth. Finally, we explain how the overall approach works.

First, assume that the input graph is a tree. Then, computing the distance between s and t boils down to finding the lowest common ancestor of s and t. Consequently, the following holds:

FACT 3. *In a tree of height h, the distance between s and t can be computed in $O(h)$ time.*

Second, simply computing the distance between all possible pairs of vertices in the graph and storing the results in a table of size n^2 yields the following:

FACT 4. *Using a size-$O(n^2)$ table, one can answer every distance query in constant time.*

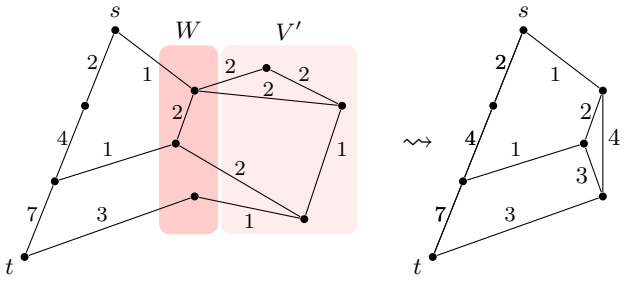


Figure 8: Left: Input graph G. Right: The graph G shrunk on its separator W. For computing the distance between s and t, it suffices to know the pairwise distances between the vertices in the separator W in the induced subgraph $G[W \cup V']$. The graph can be shrunk on the separator W by deleting V' and providing all edges within W with proper weights as shown on the right.

The table size of n^2, however, is prohibitively large for real-world applications. This leads us to our next and last fact. Assume that there is a vertex subset V' in the graph such that:

- V' is connected to the rest of the graph via a small separator[4] W and
- V' contains neither s nor t (see left side in Fig. 8).

We can simplify the graph as shown in Fig. 8 (right side). Our goal is to delete V' from G. Since a shortest path might pass through V', however, we need to keep the information about all possible paths through V'. To encode this information, we use that every sub-path of a shortest path is also a shortest path and combine this with Fact 4: Compute all pairwise distances for vertices in the separator W within the induced graph $G[W \cup V']$ and store the obtained distances using weighted edges with both endpoints in W, see Fig. 8 (left side). After that, one can remove V' without changing any distance between the vertices $s, t \in V \setminus V'$.

FACT 5. *Let $W \subseteq V \setminus V'$ separate $V' \subseteq V$ from $V \setminus V'$ with $s, t \in V \setminus V'$. Then remove V' and add edges representing shortest paths within $G[W \cup V']$ as displayed in Fig. 8.*

We now combine Facts 3 to 5 into an algorithmic strategy. Note that the usefulness of a separator (in Fact 5) depends on s and t. Moreover, our data structure shall support very fast distance queries for *all* pairs $s, t \in V$. But which separators should be used in the preprocessing (before s and t are known)? Considering *all* separators is clearly too expensive because there can be exponentially many of those. Instead, we need a relatively small set of not-too-large separators. A tree-decomposition provides such a set (given that the input graph has small treewidth).

A *tree-decomposition* of a graph $G = (V, E)$ is a tree T "containing" G. More precisely, each bag[5] b of T contains a vertex subset $V_b \subseteq V$ of G and, in particular, the following conditions are fulfilled (see Fig. 9 for an example):

(a) Every vertex of G appears in at least one bag of T.

(b) For any two bags b, b' that are adjacent in T, the intersection of their vertex sets $V_b \cap V_{b'}$ is a separator in G.

[4] That is, removing the vertex set W from the graph disconnects V' from the rest of the graph.
[5] The nodes of tree T are usually called bags.

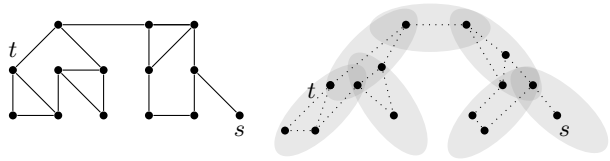


Figure 9: Left: Input graph G. Right: A tree-decomposition of G. The bags are represented by ellipses; each bag contains the vertices lying inside its corresponding ellipse. The edges of G are not part of the tree-decomposition and are thus only dotted in the right side. For the sake of simplicity, we omit the weights on the edges.

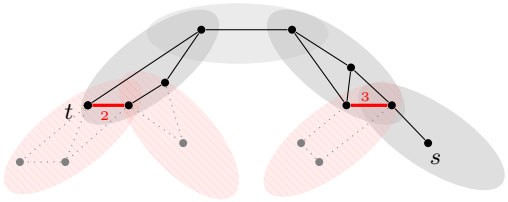


Figure 10: A tree-decomposition of a graph G (black vertices and black, dotted and solid edges). The thick red (weighted) edges encode a shortest path between the vertices within the sub-graph of G (dotted edges).

The *treewidth* of a graph G is k if there exists a tree-decomposition T such that each of the (at most $O(n)$) bags in T contains at most k vertices (thus the separators are small).

Now the algorithm for answering s-t-distance queries by using a (properly preprocessed) tree-decomposition and Facts 3 to 5 works as follows:

1. Find two bags containing s and t, respectively.
2. Find the shortest path P between these two bags in the tree T (see gray bags in Fig. 10).
3. Take the graph G' induced by the vertices in the bags of P. Based on Facts 4 and 5, add to G' the weighted edges encoding all paths in G that go via bags not in P (see thick red edges in Fig. 10).
4. Compute and return an s-t-distance in G' using Dijkstra's algorithm.

An extension of the preprocessing allows to return not only the distance but also a shortest path between s and t. The above algorithm requires a tree-decomposition, which can be computed once in the preprocessing. Computing an optimal tree-decomposition is NP-hard. However, there are practically useful heuristics computing tree-decompositions [13, 37]. Moreover, any tree-decomposition can be transformed efficiently into another tree-decomposition with depth $O(\log n)$ and roughly three times larger bags [7]. This ensures that the path Q found in Step 2 is of length $O(\log n)$.

For the computation of G' in Step 3, one needs the precomputed distances in the separators. More precisely, for each pair of adjacent bags b, b' one needs to compute for all vertex pairs in $V_b \cap V_{b'}$ the pairwise distances in both subgraphs that got disconnected by the separator. For each pair of bags, these pairwise distances will be stored in tables to allow constant-time access. Overall, these computations can be done using dynamic programming in $O(k^3 n)$ time [1], where k is the treewidth of the graph.

Next, we analyze the running time of the above algorithm. Since the depth of T is $O(\log n)$, it follows that the path P contains $O(\log n)$ bags and can, by Fact 3, be found in $O(\log n)$ time. By definition of treewidth, G' contains $O(k \log n)$ vertices and $O(k^2 \log n)$ edges. Thus, using Dijkstra's algorithm in G' requires $O(k \log n \log(k \log n) + k^2 \log n)$ time to obtain the distance between s and t. For moderately small k, this is a *sub-linear* running time. In infrastructure networks the treewidth is typically $O(\sqrt[3]{n})$ [31]. Hence, in such networks the query time is usually $O(n^{2/3} \log n)$.

Overall, the parameterized analysis yields a very good theoretical explanation and predictor for the practically observed efficiency. To the best of our knowledge, there are only few examples of similar data-structure-driven preprocessing algorithms combined with a parameterized analysis. One of these is given by a study with Gomory-Hu Trees to compute minimum cuts [8].

Notably, the presented preprocessing approach only works in conjunction with the algorithm answering the shortest path queries with the tree-decomposition. This is similar to structure simplification but in stark contrast to kernelization, which can be used as a preprocessing step before exact, approximate, or heuristic algorithms. We mention in passing, however, that the above discussed shortest path setting seems to be resistant to kernelization as any vertex in the graph can be the start or the end vertex of a query.

Summary and Outlook

Parameterization enjoyed great success in refining the computational complexity analysis of NP-hard problems, leading to an extremely rich area with a wealth of results and techniques. In this article we wanted to steer the view also to the area of polynomial-time solvable problems. More specifically, we focused on parameterized preprocessing, highlighting the potential of parameterized algorithm design for speeding up "slow" polynomial-time algorithms. The overall message we offer here is that looking through parameterized glasses, one may find promising avenues towards (provably) accelerating algorithms for fundamental computational problems.

Our studies are closely linked to the somewhat more general framework of studying fixed-parameter tractability for polynomial-time solvable problems [18] and the fine-grained complexity analysis framework [38]. The latter offers, based on assumptions such as, e. g., the Exponential Time Hypothesis, fresh ideas for proving the worst-case optimality of several known polynomial-time algorithms. Notably, parameterization again may break these fine-grained barriers by adding new dimensions of analysis, that is, problem-specific parameters that help to further refine the analysis and to design parameterized and multivariate algorithms.

Already in their path-breaking textbook on parameterized complexity in 1999, Downey and Fellows [15, Chapter 1.2] complained about the lack of predictive power of classic computational complexity analysis (and, correspondingly, algorithm design). Their parameterized complexity ideas enabled new views on the (worst-case) computational complexity landscape of mostly NP-hard problems. Taking up their ideas and trying to make them work in the world of polynomial-time problems is promising; indeed, in the era of big data, polynomial time frequently is just not enough to represent efficiency. Parameterized (quasi-)linear-time preprocessing, as we discussed, might contribute to remedying this situation.

Acknowledgment.

We devote this work to the memory of our dear colleague, friend, and mentor Rolf Niedermeier, who suddenly passed away during the revision process of this article.

We are grateful to two CACM reviewers for constructive feedback that helped to significantly improve our presentation. The research was supported by the DFG project FPT-inP, NI 369/16, and by the EPSRC grant EP/P020372/1.

References

[1] I. Abraham, S. Chechik, D. Delling, A. V. Goldberg, and R. F. Werneck. On dynamic approximate shortest paths for planar graphs with worst-case costs. In *Proceedings of SODA*, pages 740–753. SIAM, 2016.

[2] M. Bartha and M. Kresz. A depth-first algorithm to reduce graphs in linear time. In *Proceedings of SYNASC*, pages 273–281. IEEE, 2009.

[3] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *LNCS*, pages 19–80. Springer, 2016.

[4] A. A. Bertossi. Finding Hamiltonian circuits in proper interval graphs. *Information Processing Letters*, 17(2):97–101, 1983.

[5] R. E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[6] N. Blum. Maximum matching in general graphs without explicit consideration of blossoms revisited. Technical report, 2015. Available at http://arxiv.org/abs/1509.04927.

[7] H. L. Bodlaender. NC-algorithms for graphs with small treewidth. In *Proceedings of WG*, volume 344 of *LNCS*, pages 1–10. Springer, 1988.

[8] G. Borradaile, D. Eppstein, A. Nayyeri, and C. Wulff-Nilsen. All-pairs minimum cuts in near-linear time for surface-embedded graphs. In *Proceedings of SoCG*, volume 51 of *LIPIcs*, pages 22:1–22:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[9] K. Bringmann. Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. In *Proceedings of FOCS*, pages 661–670. IEEE, 2014.

[10] K. Bringmann. Fine-grained complexity theory (tutorial). In *Proceedings of STACS*, volume 126 of *LIPIcs*, pages 4:1–4:7, Dagstuhl, Germany, 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[11] K. Bringmann and M. Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of SODA*, pages 1216–1235. SIAM, 2018.

[12] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: sequential algorithms. *Algorithmica*, 27(3):212–226, 2000.

[13] H. Dell, C. Komusiewicz, N. Talmon, and M. Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *Proceedings of IPEC*, volume 89 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[14] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[15] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[16] R. Duan and S. Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM*, 61(1):1:1–1:23, 2014.

[17] F. V. Fomin, D. Lokshtanov, S. Saurabh, and M. Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2019.

[18] A. C. Giannopoulou, G. B. Mertzios, and R. Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. *Theoretical Computer Science*, 689:67–95, 2017.

[19] P. Goldberg, M. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *Journal of Computational Biology*, 2:139–152, 1995.

[20] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol. 57)*. North-Holland Publishing Co., 2nd edition, 2004.

[21] J. Gu, W. Zheng, Y. Cai, and P. Peng. Towards computing a near-maximum weighted independent set on massive graphs. In *Proceedings of KDD*, pages 467–477. ACM, 2021.

[22] J. Guo, F. Hüffner, and R. Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *Proceedings of IWPEC*, volume 3162 of *LNCS*, pages 162–173. Springer, 2004.

[23] J. Guo and R. Niedermeier. Invitation to data reduction and problem kernelization. *ACM SIGACT News*, 38(1):31–45, 2007.

[24] M. Henzinger, A. Noe, C. Schulz, and D. Strash. Finding all global minimum cuts in practice. In *Proceedings of ESA*, volume 173 of *LIPIcs*, pages 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[25] K. Ioannidou, G. B. Mertzios, and S. D. Nikolopoulos. The longest path problem has a polynomial solution on interval graphs. *Algorithmica*, 61(2):320–341, 2011.

[26] Y. Iwata, T. Ogasawara, and N. Ohsaka. On the power of tree-depth for fully polynomial FPT algorithms. In *Proceedings of STACS*, volume 96 of *LIPIcs*, pages 41:1–41:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[27] R. M. Karp and M. Sipser. Maximum matchings in sparse random graphs. In *Proceedings of FOCS*, pages 364–375. IEEE, 1981.

[28] J. D. Kececioglu and A. J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proceedings of WAE*, pages 121–132. Max-Planck-Institut für Informatik, 1998.

[29] T. Koana, V. Korenwein, A. Nichterlein, R. Niedermeier, and P. Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. *ACM Journal of Experimental Algorithmics*, 26:1–30, 2021.

[30] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[31] S. Maniu, P. Senellart, and S. Jog. An experimental study of the treewidth of real-world graph data. In

Proceedings of ICDT, volume 127 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[32] D. Marx, B. O'Sullivan, and I. Razgon. Finding small separators in linear time via treewidth reduction. *ACM Transactions on Algorithms*, 9(4):30:1–30:35, 2013.

[33] G. B. Mertzios, A. Nichterlein, and R. Niedermeier. The power of linear-time data reduction for maximum matching. *Algorithmica*, 82(12):3521–3565, 2020.

[34] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[35] R. Niedermeier. Reflections on multivariate algorithmics and problem parameterization. In *Proceedings of STACS*, volume 5 of *LIPIcs*, pages 17–32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

[36] T. Roughgarden. Beyond worst-case analysis. *Communications of the ACM*, 62(3):88–96, 2019.

[37] B. Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. Technical report, 2017. Available at http://arxiv.org/abs/1709.08949.

[38] V. Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of ICM*. World Scientific, 2018.

[39] V. Vassilevska Williams and R. R. Williams. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM*, 65(5):27:1–27:38, 2018.

[40] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI*, pages 1173–1178. Morgan Kaufmann, 2003.