CHAPTER 11

Having a hard time? Explore parameterized complexity!

Britta Dorn and Ildikó Schlotter

11.1 Motivation

More often than not, life teems with difficult problems. This is not less true if you happen to be a researcher in computational social choice; however, in this case you can spend considerable time focusing only on computational hardness.

Collective decision making has been studied from various aspects: political science, economy, mathematics, logic, and philosophy have all contributed to the area of social choice. With the advance of computer science, computational issues have become more and more important. Taking a casual look at the landscape of computational problems in social choice, we find an abundance of hard problems. Within the theory of voting, already winner determination is $NP$-hard for several electoral systems like Dodgson, Young, or Kemeny voting. Considering certain forms of manipulation, control, or bribery in elections, or dealing with partial information results in computationally hard problems as well. But we can find examples in every area of social choice, let it be judgment aggregation, auctions, fair division of goods, or matching under preferences.

Computational complexity: the classical approach. When considering the computational tractability of a given problem, we focus on the time and space necessary for an algorithm to solve it. In most cases, however, space is not a scarce resource, and therefore whether an algorithm is considered tractable or not depends on its running time. Of course, running times depend on the actual input, and to overcome this rather cumbersome difficulty, classical complexity theory teaches us to view the running time of an algorithm as a function of the length of its input. More precisely, the running time $T(n)$ of a given algorithm $A$ is defined as the maximum number of computational steps performed by $A$ on any input of length $n$. Using this notion, a broadly accepted rule of thumb is to consider $A$ (and the problem solved by $A$) tractable if $T(n)$ is a polynomial of $n$.

To grasp the notion of computational intractability, classical complexity theory offers a hierarchy of complexity classes, among which the most important is the class of $NP$-hard problems. Instead of repeating the formal definition here, we only would like to recall its most vital property. Namely, there is strong evidence indicating that $NP$-hard problems are not solvable in polynomial time. From a
practical point of view, this means that we cannot expect an algorithm solving an NP-hard problem that runs in reasonable time for large inputs.

Over the years, researchers facing NP-hard problems have come up with numerous strategies to deal with intractability. Sometimes, focusing on easy special cases might be enough. Randomization and parallel computing might also help us reduce the running time. In many areas, approximation algorithms turned out to be extremely useful. Recently, the evergrowing computational capacities made exponential-time (exact) algorithms a viable choice in some cases. And when theory does not seem to offer any help, heuristics still play an important role.

All of these strategies might be useful in computational social choice too. However, there is one crucial aspect shared by these approaches which dooms them inefficient in a certain way: they are all one-dimensional in the sense that they regard the running time merely as a function of the input length. However, in reality, there may be several properties of the input, explicit or implicit, that heavily influence the complexity of the problem, and to neglect these properties is a deep source of inefficiency.

Parameterized complexity. So far the only well-developed framework that uses a multi-dimensional approach to deal with computationally hard problems is parameterized complexity. This approach, developed first by (Downey and Fellows, 1999), considers the complexity of a given problem with respect to several so-called parameters (usually integers), and views the running time of a given algorithm as the function of both the input length and the parameters. This simple idea allows us to draw a much more detailed map of the complexity of any given problem.

Each instance of a parameterized problem $P$ consists of an input $I$ and a parameter $k$, which is usually an integer (we will explain later how to handle multiple parameters within this framework). We consider the running time of an algorithm solving $P$ as a function of $|I|$ and $k$. Since we are mostly dealing with NP-hard problems, we cannot expect this function to be polynomial. Instead, what we are interested in is whether the exponential explosion in the running time can be, in a sense, attributed to the parameter. More presicely, we ask whether $P$ admits an algorithm that runs in time

$$f(k)|I|^{O(1)}$$

for some computable function $f$. We call such an algorithm fixed-parameter tractable or FPT for short. Usually, the function $f$ is exponential (or worse), but observe that the dependence of the running time on the input length $|I|$ is a polynomial of constant degree. Hence the essential property of a fixed-parameter tractable algorithm: it works fast whenever the parameter value $k$ is a small integer. Intuitively, this indicates that the source of the computational hardness of $P$ is the parameter: if $k$ is small, our instance is tractable, but as $k$ grows, it quickly becomes intractable.

Using the notion of fixed-parameter tractability, we can examine the computational complexity of our problem from many different aspects by choosing different parameters and searching for FPT algorithms with each parameterization—we hence exploit the structure of the problem that is given in the input.
This approach has great potential from a practical perspective: if some parameter is likely to be small in typical real-world instances, then an FPT algorithm can be highly efficient in practice; we will illustrate this more in detail in Section 11.2.

**Why use parameterized complexity in social choice?** Apart from the general advantages of using the parameterized framework, there are two additional reasons why it might be extremely helpful in the field of computational social choice.

First, a typical problem in collective decision making contains a handful of natural parameters that, in certain realistic scenarios, are likely to have small values. The most basic examples are the number of agents or alternatives present, but for a typical problem we can easily detect several natural possibilities for parameterization that could lead to efficient fixed-parameter algorithms. This phenomenon can be explained by the fact that most problems in social choice model some real-world situation, and such models tend to have a composite nature, involving various entities and relations in between them. Examples include the diversity or amount of variety in a voting profile, the budget in a bribery scenario, or the ‘distance’ from an instance with a certain desirable property, such as single-peakedness for voting profiles, stability for a matching, or envy-freeness of an allocation.

Second, certain problems in the area of social choice have the curious property that their computational hardness might be, in fact, desirable. Such situations often arise when the computational problem models actions of a malicious agent; to name some examples, we can think about bribery, manipulation, or control of some decision making process. For such a problem, computational hardness means that the given process (e.g., a voting system) is *safe* in the sense that a malicious agent necessarily faces a computationally intractable situation.

Note however, that simple \( \text{NP} \)-hardness might not prevent malicious acts in reality: as we have argued earlier, even \( \text{NP} \)-hard problems might admit efficient algorithms that are applicable in practice. Thus, in such cases a more detailed complexity analysis can become crucial—and this is exactly what we can accomplish by studying our problem from the parameterized aspect.

Similarly to the classical theory of computational complexity, parameterized complexity also offers an intractability theory with a whole hierarchy of hardness-classes that can be used to provide evidence that certain problems are not fixed-parameter tractable. Parameterized complexity can hence contribute to a better valuation of the hardness of the problem in two ways: on the one hand, fixed-parameter tractability with respect to a parameter shows that \( \text{NP} \)-hardness might only constitute a theoretical barrier, in particular in applications where the value of this parameter is small. On the other hand, parameterized complexity theory may help to justify the shield provided by computational complexity: if a problem can be shown to belong to one of the hardness classes with respect to a parameter \( k \), it is unlikely that an efficient algorithm can be found to solve it, even for small values of \( k \).

**Relation to existing literature and goal of this chapter.** In the last decade, parameterized complexity has been applied with great success to many problems
in computational social choice. We refer to several surveys and overview papers relating to its use in computational social choice, starting with the work by Lindner and Rothe (2008), followed by the work by Betzler et al. (2012) concerning voting problems, the article by Bredereck et al. (2014) presenting nine research challenges in parameterized algorithmics for computational social choice, and the recent article by Faliszewski and Niedermeier (2015) on parameterization in computational social choice. The goal of this chapter is not to add on another survey of current results, trends and challenges, but to provide a comprehensive introduction for everyone who might be interested to get into this attractive area of research, tailored to applicability in computational social choice, and illustrated with helpful examples.

For a thorough introduction to parameterized algorithmics and complexity in general, we refer to the books by Downey and Fellows (1999, 2013), Flum and Grohe (2006), Niedermeier (2006), and Cygan et al. (2015).

**Organization.** We will first present in Section 11.2 some of the most basic algorithmic techniques for obtaining fixed-parameter tractability results, such as bounded search trees, data reduction and problem kernels, integer linear programming, and color-coding. We will also explain how to handle multiple parameters. We then turn to parameterized intractability in Section 11.3, where we deal with FPT reductions and the most common parameterized complexity classes. Some more advanced techniques like lower bounds for kernelization and the relation between approximation and parameterized algorithms are presented in Section 11.4. We finish with our conclusions in Section 11.5.

### 11.2 Basic Algorithmic Techniques

To illustrate the most basic techniques for designing fixed-parameter tractable algorithms, we will use the classical VERTEX COVER problem. Given a graph $G$, a **vertex cover** is a set $S$ of vertices in $G$ such that each edge of $G$ has at least one endpoint in $S$.

**VERTEX COVER:**

**Input:** An undirected graph $G$, and an integer $k$.

**Question:** Does $G$ contain a vertex cover of size at most $k$?

Although this problem itself is not about collective decision making, we believe that its importance as a graph problem renders VERTEX COVER essential also to the researchers of this area: graphs are a great and flexible tool for anyone interested in networks, multi-agent systems, or social connections of any kind. Therefore, it is not a surprise that graphs are ubiquitous in computational social choice problems, and that they are crucial for understanding how and why certain structures underlying our given computational problem affect its tractability.

VERTEX COVER is a graph problem belonging to the 21 problems proved to be NP-complete by Karp in his seminal paper (Karp, 1972). Thus, we obviously
cannot hope to solve this problem by a polynomial-time algorithm. Given this intractability result and the central role of VERTEX COVER in graph theory, several researchers have attempted to design algorithms for it that would perform well in practical situations. In the last decades, VERTEX COVER became one of the most prominent problems in parameterized complexity, showing how successfully the parameterized framework can be applied in practice.

**Brute force approach.** Let \((G, k)\) be an instance of VERTEX COVER with \(G\) having \(n\) vertices. The most simple, brute force approach would be the following: simply try every possible set \(S\) of at most \(k\) vertices, and check if \(S\) is indeed a vertex cover. Since this latter condition for a given set \(S\) can be checked in \(O(|E(G)|)\) time, the whole process can be performed in \(\binom{n}{k} O(|E(G)|)\) time. \(^1\)

Of course, one can easily observe that \(k\) vertices can cover (i.e., be adjacent to) at most \(k(n - 1)\) vertices, so if \(G\) has more than \(k(n - 1)\) edges, then \((G, k)\) must be a 'no'-instance. Hence, we may assume that \(|E(G)| \leq k(n - 1) = O(nk)\). Using this, the brute force algorithm described above has running time

\[
\binom{n}{k} O(nk) = O(kn^{k+1}).
\]

Such a running time becomes intractable already for relatively small graphs: it cannot even deal with an instance where \(n = 100\) and \(k = 10\).

In what follows, we shall see some basic techniques in parameterized complexity that can be used to design much more efficient algorithms. Currently the fastest algorithm for VERTEX COVER, developed by (Chen et al., 2010), runs in time \(O(1.2738^k + kn)\). This renders VERTEX COVER solvable even for instances as large as \(n = 10^6\) and \(k = 40\).

### 11.2.1 Bounded Search Tree

Example: VC.
Example: Minimax Approval Voting.

### 11.2.2 Kernelization

Example: VC.
Example: Coalitional Manipulation for Copeland
Example: EEF Allocation (trivial kernel)
Mention also partial kernels. [see also Section 3.5 of Bredereck et al. (2014) (partial kernel for Kemeny score)]

### 11.2.3 Integer Linear Programming

Many problems can be formulated in terms of an optimization task of a linear objective function which has to fulfill several constraints given by linear

\(^1\)Here and later on, we will rely on the standard notation in graph theory, as used for example in the book by Diestel (Diestel, 2005).
(in)equalities. These *linear programs* can be described in their canonical form as follows:

<table>
<thead>
<tr>
<th>Linear Programming (LP):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Matrix $A \in \mathbb{R}^{m,n}$, two vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$.</td>
</tr>
<tr>
<td><strong>Task:</strong> Find a vector $x \in \mathbb{R}^n$ with $x \geq 0$ that fulfills $Ax \leq b$ and, among all such vectors, maximizes $c^T x$.</td>
</tr>
</tbody>
</table>

The problem can equivalently be formulated in other variants, e.g., as a minimization problem, or with equalities.

LP problems are known to be solvable in polynomial time. If the variables can only take integral values, one speaks of an *ILP (Integer Linear Program)*. This makes the problem more difficult in general: the corresponding decision problem is NP-complete (Karp, 1972). However, an ILP formulation of a problem can help us obtain an FPT result: A famous theorem by Lenstra (1983) states that solving an ILP is fixed-parameter tractable if the parameter is the number of variables or the number of constraints. Lenstra’s running time was later improved by (Kannan, 1987) and (Frank and Tardos, 1987), yielding that an ILP with $p$ variables can be solved in $O(p^{2.5p+o(p)} \cdot |I|)$ time, where $|I|$ is the input size.

However, we shall remark that the combinatorial explosion of the running time shown by Lenstra is terrible, rendering it impractical. Lenstra’s result should therefore be seen as a classification theorem in the first place. We refer to a more detailed discussion about ILP-based fixed-parameter tractability by Bredereck et al. (2014, Section 3.1).

**Example: ILP for Vertex Cover**

We start by giving a negative example for VERTEX COVER. To solve this problem by an ILP, we proceed as follows. For each vertex $v \in V$, we create a variable $x_v$ which can take a value in $\{0, 1\}$: including some vertex $v$ in the vertex cover corresponds to setting the value of variable $x_v$ to 1. Then the following ILP clearly solves the problem:

\[
\begin{align*}
\text{Minimize} & \quad \sum_{v \in V} x_v \\
\text{subject to} & \quad x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\
& \quad x_v \in \{0, 1\} \quad \forall v \in V
\end{align*}
\]

However, the number of variables used in this ILP is equal to $|V| = n$ and therefore depends not only on the parameter $k$, but on the input size, so Lenstra’s result is not applicable here.

**Example: ILP for EEF Allocation**

**Example: ILP for CRG**

**11.2.4 Color-coding**

Example: Minimax Approval Voting
11.2.5 Multiple Parameters

How can we handle it (taking the sum or the max of the parameters). Multidimensional study: each "variant" can be a) constant, b) parameter, c) unbounded. We could cite some paper by Ronald’s, and my paper with Matthias Mnich (only on arXiv now).

11.3 Parameterized intractability

Intro: Independent Set and its relation to VC, their difference.

11.3.1 FPT reduction

Definition, trivial example (IS \(\rightarrow\) Clique), non-example (VC \(\rightarrow\) IS).

11.3.2 Parameterized complexity classes

Example: Coalitional Resource Game is W[1]-hard, reduction from Clique
Example: EEF is W[1]-complete, reduction from Unary Bin Packing

11.4 Advanced Techniques

Just mentioning some further topics/techniques, mostly without formal definitions and all examples without proofs.

11.4.1 Lower bounds for kernelization

Examples: Coalitional Manipulation for Copeland, Efficient Envy-Free Allocation

11.4.2 Approximation and parameterized algorithms

Example: Minimax Approval Voting

11.4.3 Lower Bounds Assuming ETH?

11.5 Conclusion

Bibliography


