Chapter 2

Preliminaries

2.1 Basic Graph Theory

A graph is a tuple (V, E), where V is a finite set of *vertices* and E is a set of *edges*, which are size-two subsets of V, that is $E \subseteq \{\{u, v\} \mid \{u, v\} \subseteq V\}$. Note that by this definition the edges are not directed and there are no multiple edges or self-loops, that is, the graphs considered in this thesis are simple and undirected. The *complement* of a graph $G = (V, E)$ is the graph $\bar{G} := (V, \bar{E})$, where $\bar{E} := \{\{u, v\} \mid \{u, v\} \subseteq V\} \setminus E$. For a graph $G = (V, E)$, we write $V(G)$ to denote its vertex set and $E(G)$ to denote its edge set. By default, we use n and m to denote the number of vertices and edges, respectively, of a given graph. Two vertices $u, v \in V$ are *adjacent* if $\{u, v\} \in E$. A vertex $v \in V$ and an edge $e \in E$ are *incident* if $v \in e$.

For a vertex $v \in V(G)$, the set $N_G(v) := \{u \in V \mid \{u, v\} \in E\}$ is the set of *neighbors* of v. The *closed* neighborhood of v is defined as $N_G[v] := N_G(v) \cup \{v\}$. For $S \subseteq V$, the set $N_G(S) := \bigcup_{v \in S} N(v) \setminus S$ is the *neighborhood* of S. The *closed* neighborhood is denoted as $N_G[S] := N_G(S) \cup S$. If the graph G is clear from the context, then we also write $N(v)$, $N[v]$, $N(S)$, and $N[S]$ instead of $N_G(v)$, $N_G[v]$, $N_G(S)$, and $N_G[S]$, respectively. The *degree* of a vertex v is the number of its neighbors $|N(v)|$. If every vertex in G has degree at most d, then we say that G has *maximum degree* d. For a vertex set $S \subseteq V$, we write $G[S]$ to denote the graph induced by S in G, that is, $G[S] := (S, \{e \in E \mid e \subseteq S\})$. For a vertex $v \in V$, we also write $G - v$ instead of $G[V \setminus \{v\}]$ and for a vertex set $S \subseteq V$ we also write $G - S$ instead of $G[V \setminus S]$.

A *path* is a sequence of vertices $v_1, \ldots, v_p$ with $\{v_i, v_{i+1}\} \in E$ for all $1 \leq i < p$, where all the vertices v_i are distinct. The number of edges of a path is its *length*. A *cycle* is a path with $\{v_p, v_1\} \in E$. The *girth* of a graph is the length of a shortest cycle in it. A *clique* is a complete graph, that is, a graph in which all vertices are pairwise adjacent. A K_n is a clique of n vertices. The graph K_3 is also called *triangle*. A P_n is a path of n vertices, and C_n is a cycle of n vertices. A *wheel* is a graph W that has a vertex $v \in V(W)$ that is adjacent to all other vertices such that $W - v$ is a cycle. For $s \geq 1$, the graph $K_{1,s} := (\{u, v_1, \ldots, v_s\}, \{\{u, v_1\}, \ldots, \{u, v_s\}\})$ is an *s-star*, or simply *star*. The vertex u is the *center* of the star and the vertices $v_1, \ldots, v_s$ are the *leaves* of the star. A *$\leq s$-star* is an s'-star with $s' \leq s$ and a *$<s$-star* is an s'-star with $s' < s$. A graph is *connected* if there is a path between any two vertices. For a connected graph G, a *cut-vertex* is a vertex $v \in V$ such that $G - v$ is not connected. The *distance* between two vertices u, v is the length of a shortest path between u and v. The *distance* between two edges e_1, e_2 is the smallest distance between any two vertices $u \in e_1$ and $v \in e_2$.

Given an undirected graph $G = (V, E)$ and an edge subset $E' \subseteq E$, to *subdivide* the edges E' in G means to remove from G all edges in E', and then to add for each edge $\{u, v\} \in E'$ a vertex $x_{u,v}$, making it adjacent to u and v. The vertices in $\{x_{u,v} \mid \{u, v\} \in E'\}$ are called *subdivision vertices*.

For a family of graphs $\mathcal{H}$ we define $V(\mathcal{H}) := \bigcup_{H \in \mathcal{H}} V(H)$ and $E(\mathcal{H}) := \bigcup_{H \in \mathcal{H}} E(H)$. We say that a graph H' is a *copy of H* if H' is isomorphic to H. For a graph G and a graph H, we say that H' is a *copy of H in G* if H' is a subgraph of G and H' is a copy of H. Given two graphs H_1 and H_2, the *intersection* of H_1 and H_2 is defined as $V(H_1) \cap V(H_2)$. A *packing* P of a graph H in a graph G is a set of pairwise vertex-disjoint copies of H in G.

Matching Basics. Given an undirected graph $G = (V, E)$, an edge subset $M \subseteq E$ is called a *matching* if the edges in M are pairwise disjoint. A matching M is *maximal* if there exists no edge $e \in (E \setminus M)$ such that $M \cup \{e\}$ is a matching. A matching M is *maximum* if there exists no larger matching. A vertex $v \in V$ is *matched* if there exists an edge in M that is incident to v. A vertex $v \in V$ is *unmatched* if it is not matched. An *M-alternating path* is a path in G that starts with an unmatched vertex, and then contains, alternately, edges from $E \setminus M$ and M. If an M-alternating path ends with an unmatched vertex, then it is called *M-augmenting path*.

Graph Properties. A *graph property* Π is a (possibly infinite) set of graphs. We also write that a graph G *satisfies* Π if $G \in \Pi$. A graph property Π is *hereditary* if it is closed under deleting vertices, that is, if $G \in \Pi$, then for any induced subgraph G' of G, $G' \in \Pi$. A hereditary graph property is *non-trivial* if it is satisfied by infinitely many graphs and it is not satisfied by infinitely many graphs. A hereditary graph property is *determined by the components* if a graph G satisfies Π whenever every connected component of G satisfies Π. For any hereditary property Π there exists a set of "minimal" forbidden induced subgraphs, that is, forbidden graphs for which every induced subgraph satisfies Π [GHK73]. If Π is a hereditary property that is determined by the components, then the corresponding set of forbidden induced subgraphs only contains *connected* graphs.

A graph is *planar* if it can be embedded in the plane, that is, it can be drawn in a plane such that the edges only intersect in their endpoints. Every planar graph contains a vertex of degree at most five, which is a consequence of Euler's formula.

For more about graph theory, we refer to the books by Diestel [Die05] and West [Wes01].

2.2 Parameterized Complexity and Fixed-Parameter Algorithms

Since many graph problem are NP-hard, it seems hopeless to solve them exactly in polynomial time. However, NP-hardness expresses the computational hardness of a problem in the *worst case*, and there often exist even large instances of NP-hard problems that can be solved in reasonable time. The reason is that such instances might contain some structure that can be exploited by an algorithm. Such structure can often be expressed by a parameter (usually a nonnegative integer), and then one can do a two-dimensional worst-case analysis that measures the growth of the running time depending on the input size *and* the parameter. The hope is that the seemingly unavoidable combinatorial explosion of the running time can be restricted to the parameter. Then, if the parameter is small, which is often a reasonable assumption, the problem can be solved efficiently even on large instances.

For instance, for the Vertex Cover application sketched in Chapter 1 (sequence alignment), it is reasonable to assume that the solution is small; otherwise, one would have to remove too many sequences from the sample, which is an indication that the sample contains too many errors in order to derive meaningful results. Other types of parameters restrict the structure of the input graph; for instance, there are problems where the input typically has a tree-like structure which can be exploited to find an optimal solution (see also Section 2.3.4).

Downey and Fellows [DF99] first describe a formal framework for such a two-dimensional analysis of problems.

Definition 2.1. *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The second component is called the* parameter *of the problem.*

Throughout this thesis the parameter is a nonnegative integer, and therefore we assume that $L \subseteq \Sigma^* \times \mathbb{N}$. For $(I, k) \in L$, the two dimensions of the parameterized complexity analysis are then the input size $n := |(I, k)|$ and the parameter k. Since in our applications all parameter values are upper-bounded by $|I|$, we can simply assume $n := |I|$ in our asymptotic considerations. The following notion expresses that a parameterized problem can be solved efficiently for small parameter values.

Definition 2.2. *A parameterized problem L is* fixed-parameter tractable *with respect to the parameter k if there exists an algorithm that decides in $f(k) \cdot \text{poly}(n)$*

time whether $(I, k) \in L$, where f is a computable function only depending on k. The complexity class containing all fixed-parameter tractable problems is called FPT.

In other words, a parameterized problem is fixed-parameter tractable if it can be solved in time that is exponential in the parameter, but only polynomial in the input size. There are several techniques to show that a parameterized problem is fixed-parameter tractable. In the next section, we introduce some of the most important ones used in this thesis, like problem kernelization (Section 2.3.1), bounded search trees (Section 2.3.2), iterative compression (Section 2.3.3), and dynamic programming on tree decompositions (Section 2.3.4). There also exist parameterized problems that are likely to be not fixed-parameter tractable. Analogously to the concept of NP-hardness, Downey and Fellows [DF99] developed a framework containing reduction and completeness notions in order to show hardness of parameterized problems. See Section 2.3.5 for more details.

For a more detailed introduction to parameterized algorithmics and parameterized complexity theory we refer to the books by Downey and Fellows [DF99], Flum and Grohe [FG06], and Niedermeier [Nie06].

2.3 Basic Fixed-Parameter Techniques

In this section, we outline some of the most important techniques in the field of fixed-parameter algorithmics that are applied in this thesis. Concerning the first three techniques, see also a recent survey by Hüffner et al. [HNW08] for a more detailed description with many examples.

2.3.1 Problem Kernelization

To solve NP-hard problems, polynomial-time preprocessing is a natural approach. The main idea is to use preprocessing to remove the "easy" parts of the input in order to obtain the computationally hard "core" of the instance. One important requirement of such preprocessing in our context is that they preserve the ability to solve the problem to optimality, that is, that an optimal solution for the reduced instance can be used to derive an optimal solution for the input instance.

In the classic one-dimensional analysis of algorithms, it is difficult to measure the quality of such an "exact" polynomial-time preprocessing, since any preprocessing step with provable effectiveness (that is, a guarantee that the preprocessing step will reduce the instance) could be applied repeatedly until the remaining instance is empty, which would imply P = NP. The picture changes completely if we consider parameterized problems. Here, the parameter can be used to show provable size bounds of the instance after applying the preprocessing algorithm. Such a reduced instance is called *problem kernel*.

Definition 2.3. *Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. A* reduction to a problem kernel *or* kernelization *is a polynomial-time transformation of an instance (I, k) to an instance (I', k') such that $(I, k) \in L$ if any only if $(I', k') \in L$, $|I'| \leq g(k)$ for some arbitrary computable function g depending only on k, and $k' \leq k$.*

Thus, a problem kernelization is an algorithm that replaces the input instance by an equivalent instance whose size depends only on k and not on the input size anymore. The *size* of the problem kernel is $|I'|$. However, for many graph problems, the kernel size is often stated with respect to the number of vertices only. Moreover, a problem kernel with $O(k)$ vertices is often called "linear problem kernel", although it might contain $O(k^2)$ edges. In this thesis, most of the kernelization results are stated with respect to the number of vertices.

A problem kernelization is often described via data reduction rules. A *data reduction rule* is an algorithm that replaces in polynomial time an instance (I, k) with an instance (I', k'), where $|I'| < |I|$, such that $(I, k) \in L$ if and only if $(I', k') \in L$. A problem instance to which none of a given set of reduction rules applies is called *reduced* with respect to these rules.

For an example of a problem kernel, consider the parameterized version of Vertex Cover, where the size of a vertex cover is bounded by the parameter k. If there is a vertex v of degree at least $k + 1$, then one may assume that v is in the vertex cover, since otherwise all neighbors of v have to be in the cover, which would be more than k. Therefore, as a first data reduction rule, we add all vertices with at least $k+1$ neighbors to the vertex cover, and for each vertex that is added, we decrease the parameter k by one. After that, a second data reduction rule deletes all degree-0 vertices, which is obviously correct. If the remaining graph is a yes-instance, that is, there exists a vertex cover S of size at most k, then the remaining graph contains at most k^2 edges and at most $k + k^2$ vertices, since each vertex in S has a most k neighbors, and there are no edges between vertices in $N(S)$. Therefore, a last data reduction rule returns the reduced graph if it contains at most k^2 edges and at most $k+k^2$ vertices; otherwise, it returns a trivial no-instance. The resulting instance is a $O(k^2)$-size problem kernel for Vertex Cover with respect to the parameter k. The currently best problem kernel for Vertex Cover has at most $2k$ vertices [NT75, CKJ01]. This kernelization has found practical applications in computational biology, where it helps to make problem instances small enough such that they can be solved exactly [AFLS07].

It is not difficult to see that any parameterized problem that admits a problem kernel is fixed-parameter tractable. The corresponding fixed-parameter algorithm simply solves the problem by brute force on the problem kernel. The contrary is also true:

Theorem 2.1 ([CCDF97]). *For every parameterized problem that is fixed-parameter tractable there exists a problem kernel and vice versa.*

Unfortunately, the theorem cannot be used to get an efficient fixed-parameter algorithm from a problem kernel or a small (e.g., polynomial size) problem kernel from a fixed-parameter algorithm. It is mainly used to establish fixed-parameter tractability or the existence of a problem kernel.

Problem kernelization is a very powerful tool to show the effectiveness of data reduction rules. Moreover, since it preserves the ability to solve the problem exactly, virtually any method can be used to solve the problem on the kernel (like fixed-parameter algorithms, but also approximation and heuristic algorithms). However, problem kernelization is not restricted to serve as a pure preprocessing step. There is theoretical [NR00] and practical [ALSS06] evidence that it can be efficiently interleaved with the main solving algorithm (in particular bounded search trees, see Section 2.3.2). For instance, our experimental results in Chapter 9 are heavily based on such methods, achieving speedups of several orders of magnitude in practice.

A "success story" for kernelization is Cluster Editing, the problem of adding and deleting at most k edges of a graph such that every connected component becomes a clique. Here, a first problem kernel had $O(k^2)$ vertices [GGHN05], where k is the number of allowed editing operations. The kernelization has been gradually improved [FLRS07, PdSS09], and the best-known kernel size is now $4k$ vertices [Guo09]. Kernelization algorithms for Cluster Editing have also found applications in practice [BBK09, DLL$^+$06]. As another example, for the (undirected) Feedback Vertex Set problem, a kernel of $O(k^3)$ vertices [Bod07] has recently been improved to $O(k^2)$ vertices [Tho09], where k is the feedback vertex set number of the given graph.

For more about kernelization refer to a survey by Guo and Niedermeier [GN07a].

2.3.2 Bounded Search Trees

It is usually inevitable to use some exponential-time method in order to solve an NP-hard problem to optimality. A standard way to do so is a systematic exhaustive search, which can be organized in a tree-like fashion. The basic idea is to find in polynomial-time a small part of the input such that at least one element of that part has to be in an optimal solution. We then branch into several cases of choosing an element of the small part to be in the solution, and then proceed recursively until a solution is found. A search tree corresponds to the recursive calls of such an algorithm. If we can bound the number of cases in each search tree node as well as the height of the tree (the maximum number of nested recursive calls), then we obtain a fixed-parameter algorithm. The number of recursive calls is the number of nodes in the according tree. This number is governed by linear recurrences with constant coefficients. These can be solved by standard mathematical methods [Nie06]. If the algorithm solves a problem instance of size s and calls itself recursively for problem instances of sizes $s - d_1, \dots, s - d_i$, then $(d_1, \dots, d_i)$ is called the *branching vector* of this recursion. It corresponds to the recurrence $T_s = T_{s-d_1} + \dots + T_{s-d_i}$ for the asymptotic size T_s of the overall search tree.