Parameterized Complexity: Theory and Algorithms

Sam Boardman, Zhan Shi, Yuxin Xue

December 6, 2023

Abstract

We present a survey of parameterized complexity theory and parameterized algorithms. The complexity classes FPT, XP, and para-NP are formally stated, in addition to the W-hierarchy. Classical graph problems are related to these classes and the W-hierarchy, and analogous notions of hardness and reductions are developed for them. Paradigms for designing algorithms corresponding to these complexity classes are reviewed, including kernelization, dynamic programming, and bounded search tree, and are applied to NP-hard problems. Analysis and further directions of inquiry are provided.

1 Introduction

1.1 Motivation

A fundamental task in complexity theory is the classification of computational problems according to the resources they require, most notably the time it takes to execute an algorithm that solves a problem. Such runtime is typically measured as a function of the size of the input to the algorithm, with traditional complexity classes distinguishing between, e.g., problems solvable in polynomial-time and ones solvable in exponential-time. There are both practical and theoretical reasons for doing this. For many applications, especially given the scale of the Internet and the advent of big data, problems must be solved for very large instances, so the viability of an algorithm depend on how quickly its runtime increases with input size; in an asymptotic sense, polynomial growth is arbitrarily better than exponential growth. On the other hand, this difference speaks to the apparent disparity in P and NP, one of the most important open questions in computer science. Notwithstanding, this approach for measuring runtime may be further refined by interpreting the input in a more nuanced manner, reflecting that data are not just exchangeable bits and often consist of discrete parts that may behave very differently as the input size increases, which garners new practical results and theoretical implications alike. More concretely, we may regard instances of certain languages as a pair, comprised of an object and a numerical parameter, where the language is formed of those for which the object satisfies some property depending on the parameter. Then, we may consider algorithms whose runtime is a function of both the object's size and the parameter; this is the focus of parameterized complexity, also called two-dimensional complexity theory due to the preceding. Parameterized complexity gives surprising hope that many problems generally thought of as hard can be practically and exactly solved in certain contexts; a key goal of the survey is to demonstrate this.

1.2 Outline of Paper

Section 2 introduces the notion of parameterized problems, the fundamental complexity classes for parameterized complexity theory, and the corresponding hardness concepts. Section 3 places many

well-known NP-hard graph problems in one of these preceding classes—demonstrating some type of tractability—by presenting algorithms for them; the algorithms are emblematic of broader themes in parameterized algorithms. Section 4 offers further insights and concluding remarks.

2 Parameterized Complexity Classes

We start by describing parametric complexity classes by formalizing the notion of a parameterized problem.

Definition 2.1. *A parameterized problem is a language $L \subset \{0,1\}^* \times \mathbb{N}$. Given an instance (x,k), we call x the input and k the parameter.*

Often times, the parameter of a given problem depends on the context, and a parameterized algorithm refers to one where the run-time depends on the parameter (and input size).

2.1 Complexity Classes FPT, XP and para-NP

We know that the language of Independent Set is a language of complexity NP. Consider the following parameterized version.

Definition 2.2. *The language k-Independent Set consists of graphs $G = (V,E)$ where there exists a subset $S \subseteq V$, where $|S| \geq k$, that are pairwise not adjacent.*

To decide this language, consider the following algorithm [1]: given $G = (V,E)$, where $n = |V|$, generate all size k subsets of vertices and check whether they are pairwise disjoint. This algorithm has run time:

$$O(\binom{n}{k}k^2) = O(n^k k^2),$$

where $\binom{n}{k}$ is the number of subsets of size k. Notice the first term n^k depends on both n and k, while the second term only depends on k. We categorize languages admitting an algorithm with this property as follows.

Definition 2.3. *The complexity class XP, namely slice-wise polynomial tractable, consists of parameterized languages L for which there exists an algorithm $\mathcal{A}$ and two computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that $\mathcal{A}$ runs in time*

$$|(x,k)|^{g(k)} f(k)$$

on input (x,k) and decides L.

Now we consider another parameterized NP-hard problem.

Definition 2.4. *The language k-Vertex-Cover describes a graph $G = (V,E)$ where there exists a subset $S \subseteq V$ where $|S| \leq k$, such that for each edge $e \in E$, one of its end points is in S.*

There is a recursive algorithm that can decide this language in time [1]:

$$O((m+n) * 2^k), m = |E|, n = |V|$$

Notice the first term $(m+n)$ was not raised to the power of k. We will classify such languages as follows.

Definition 2.5. *The complexity class* FPT*, namely fixed-parameter-tractable, consists of parameterized languages L for which there exists an algorithm $\mathcal{A}$ and a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathcal{A}$ runs in time*

$$|(x,k)|^c f(k)$$

on input (x,k) and decides L.

We can clearly see that FPT $\subseteq$ XP since FPT is a more restrictive version of XP. Correspondingly, people have been trying to reduce problems in XP to FPT. A current ongoing debate is whether k-clique, the same complexity as k-independent set, is in FPT. One hypothesis is that k-clique is not in FPT.

We have shown that some problems are efficient to solve if given a parameter k. However, other problems are still NP-hard. For example:

Definition 2.6. *The language k-*Vertex-Coloring *is the collection of graphs that can be colored by at most k colors such that there are no adjacent vertices with the same color.*

Unfortunately for any $k > 3$, this is an NP-hard problem [1]. We call such a problem para-NP-hard [2].

Definition 2.7. *The complexity class* para-NP *consists of parameterized languages L having nondeterministic algorithms that can decide if an instance x is in L with parameter k in time*

$$p(|x|)f(k),$$

where p is a polynomial.

2.2 Parametric Reductions

One can define a lot of parameterized languages with respect to many different parameters. It would be daunting to come up with algorithms for each language with every possible parameter. Some of the languages might not be fixed-parameter-solvable, like k-Vertex-Coloring. Therefore, we want to provide some lower bounds on the complexity of certain languages by presenting the so-called W-Hierarchy. To do this, we need to define a relationship between languages analogous to Karp reductions, which we could use to efficiently reduce one language to another language and so forth. This is what is called a parameterized reduction, and we follow the exposition of [3].

Definition 2.8. *Let A, B be two parameterized languages. We say that A can be parametrically reduced to B if there exists an algorithm $\mathcal{A}$ that given $(x,k) \in A$ as input, outputs (x',k') that satisfies the following condition:*

1. *$(x,k) \in A$ if and only if $(x',k') \in B$*
2. *$k' < g(k)$ for some computable function g (k' cannot be too big)*
3. *The run-time of $\mathcal{A}$ is $f(k) * |x|^{O(1)}$*

Here is an introductory example [3].

Theorem 2.9. *The language* Independent Set *can be parametrically reduced to* Clique.

Proof. Given $(G = (V, E), k) \in$ Independent Set, consider the complement graph $(\bar{G} = (V, E'), k)$, where $e \in E$ if and only if $e \notin E'$. We argue that $(\bar{G}, k) \in$ Clique.

We now inspect whether this reduction satisfies the requirements of a parameterized reduction.

1. If $(G, k) \in$ Independent Set, that means there is an independent set H of size at least k in the graph G, where for each pair of vertices $(a, b) \in H$, there is no edge between them. When we take the complement graph $\bar{G}$ and the same vertex set H, we see that there will be an edge for each pair of vertices $(a, b), a, b \in H$ in $\bar{G}$. Thus $(\bar{G}, k) \in$ Clique.

2. The new parameter k is the same as the old parameter, which is also k. We can just let $g(x) = 2 * x$, which is clearly a computable function, and it satisfies $k' = k < g(k)$.

3. Since we only flipped the existence for every edge, in the worst case scenario, the run-time is $O(n^2)$, which clearly is of the form $f(k) * |x|^{O(1)}$.

□

We must be careful, however; some Karp reductions might not make sense if we try to convert them to a parametric reduction. For example:

Theorem 2.10. *Given $(G, k) \in$ Independent Set, $(G, n - k)$, where $n = |V(G)|$, is in* Vertex Cover. *However, this is not a parametric reduction.*

Proof. The new parameter $k' = n - k$ includes a term n in it. Since $k' < g(k)$ for some computable k, then $n - k < g(k)$. However, n could be unbounded. So this reduction violates requirement 2. □

Notice that the reduction algorithm could run for $f(k) * |x|^{O(1)}$ amount of time, which is exponential. This means that parametric reductions might not be a polynomial-time reduction. The other significance of parametric reductions is that they imply the parametric complexity of certain languages.

Theorem 2.11. *If a language A can be parametrically reduced to a language $B \in$ FPT, then $A \in$ FPT.*

Proof. Usually we assume that the functions f, g are non-decreasing since for any f, g we could make new functions f', g' such that $f'(x) = \max_{i=0}^{x} f(i)$ [3].

If this is the case, then we have an algorithm to transform input $(x, k) \in A$ to $(x', k') \in B$, where $k' < g(k)$ for computable g, and this reduction algorithm takes $f(k) * |x|^l$ to compute for some computable f, where l is a large enough constant. Now because $B \in$ FPT, given the result of the reduction algorithm, (x', k'), we can decide whether it belongs to B in run-time $|x'|^c f'(k')$ for computable f'.

To decide language A, we can first run the reduction algorithm and then use the parametric decider for B. Observe $|x'| < f(k) * |x|^l$ since the reduction algorithm can only run for this amount of time. We also know that since $k' < g(k)$, then $f'(k') < f' \circ g(k)$. Substitute these values into the run-time of the decider algorithm for B to get:

$$|f(k) * |x|^l|^c * f' \circ g(k)$$

Rearrange the terms and add the time taken for the first step to obtain:

$$(|x|^{lc}) * (f(k)^c * f' \circ g(k)) + f(k) * |x|^l$$

We know that $(f(k)^c * f' \circ g(k))$ is computable since it is a finite composition of computable functions, and lc is a constant. Moreover we know that $f(k) * |x|^l$ is the time bound required by FPT. Hence $A \in$ FPT. □

Considering Theorem 2.11, since Vertex Cover $\in$ FPT, this would imply Independent Set is in FPT as well. However, we do not currently know whether there is a parametric reduction exhibiting this.

Additionally, parametric reductions obey transitivity.

Theorem 2.12. *If language A can be parametrically reduced to language B, and B can be parametrically reduced to language C, then A can be parametrically reduced to C.*

Proof. Again, let (x', k') be the output of (x, k) from the reduction algorithm from A to B, with $k' < g(k)$ and run-time $f(k)|x|^c$ for computable f, g. We also let (x'', k'') be the output from B to C, with $k'' < g'(k')$ and run-time $f'(k)|x'|^d$. From Theorem 2.11 we already know that $|x'| < f(k)|x|^c$. We first reduce an instance (x, k) of A to an instance of B and then from that instance of B to an instance of C. The time required would be at most

$$f' \circ g(k)(f(k)|x|^c)^d + f(k)|x|^c$$

and

$$k'' < g' \circ g(k)$$

Rearranging the first equation, we have:

$$f' \circ g(k) f(k)^d |x|^{cd} + f(k)|x|^c$$

This term satisfies the requirement of time constraint of parametric reduction. $g' \circ g$ is also computable so it satisfies the third constraint. Thus A can be parametrically reduced to C. □

With this in mind, we can perform a chain of reductions to reduce the starting language to a series of languages that are at least as hard as the starting language. We present a critical reduction from a variant of Independent Set to the language of Dominating Set [3] in this spirit.

Definition 2.13. *An instance of* Multicolored Indset *is similar to an instance of* Independent Set *but with the vertices partitioned into sets $V_1, ..., V_k$, where k is the size of the independent set. A graph $G = (V, E)$ is in* Multicolored Indset *if G has an independent set $I \subseteq V$ with each vertex in a different V_i.*

We now show how this reduction works.

Theorem 2.14. *$G = (V, E, C = (V_1, ..., V_k)) \in$ Multicolored Indset if and only if $(G', k) \in$ Dominating Set.*

Proof.

Algorithm 1 Reduce MULTICOLORED INDSET to DOMINATING SET, $G = (V, E, C = (V_1, ..., V_k))$

Require: G is a graph, $(V_1, ..., V_k)$ are collection of vertices in G

Construct a new graph G' as follows:
for each $v \in V$ **do**
    Add vertex v to G'.
for each $V_i \in C$ **do**
    make V_i in corresponding G' a clique
    create two new vertices x_i, y_i, connect them with every $v \in V_i$ in G' (do not connect x_i, y_i with an edge)
for $e = (u, v) \in E$, $u \in V_i, v \in V_j$ **do**
    Introduce vertex w_e, connect it with every vertex in both V_i and V_j except u and v.

- If $G = (V, E, C = (V_1, ..., V_k)) \in$ MULTICOLORED INDSET, then G does have an independent set I that includes one vertex from each $V_i \in C$, denoted v_i. Now consider graph G' and the same vertex of I in G'. I would certainly dominate all $V_i \in C$ and all x_i, y_i added already, since in G', any $v \in V_i$ connects to any other vertex in V_i as well as x_i, y_i. We just need to consider the newly added vertex w_e. Since w_e comes from $e = (u, v)$ in the original graph G, I could only contain at most one of u or v to be an independent set. This means I contains at least one vertex from $V_i - u$ or $V_j - v$ or both. By construction any vertex in these two category will dominate w_e. In all I can dominate every vertex in G'.

- Now we consider if $(G', k) \in$ DOMINATING SET. For D of size k to dominate every x_i and y_i where $i = 1, .., k$, D must choose one and exactly one vertex from each set of $V_i \cup \{x_i, y_i\}$. Consider that x_i and y_i are not connected with each other, we could not choose x_i or y_i. Therefore D contains one and only one vertex from each V_i. We will argue that D is the multicolored independent set. We do this by taking contra-positives. Suppose D contains $v_i \in V_i$ and $v_j \in V_j$ and $e = (v_i, v_j) \in E$ (D is not a multicolored independent set), then D will not dominate w_e since w_e are connect to vertices in V_i and V_j except v_i, v_j. This means D must be multicolored independent.

□

There are a series of reductions of parameterized languages such as from CLIQUE to INDEPENDENT SET on regular graphs and from CLIQUE to MULTICOLORED CLIQUE and so on [3]. It is an involved process to reduce from CLIQUE to MULTICOLORED INDSET and to DOMINATING SET as well as its counterparts. We show only the most useful reduction as it relates to the contents in the next section.

2.3 W-Hierarchy

Unlike NP-hard problems, we do not necessarily know whether there is some parametric language, for example INDEPENDENT SET, that is parametrically reducible to another language, for example VERTEX COVER. This indicates the likely existence of some difficulty hierarchy within hard parametric languages, which we will call the W-hierarchy [3]. To proceed, we need knowledge of Boolean circuits.

Definition 2.15. *A Boolean circuit is a directly acyclic graph where the nodes are interpreted as logical gates or input or output nodes. The following conditions also hold.*

1. *The input nodes do not have incoming edges*
2. *Each* NOT *gate has a fan-in of 1*
3. *Any node that has fan-in at least 2 is either an* AND *gate or an* OR *gate.*
4. *There is exactly one output node, which has no outgoing edges.*

The following definition is drawn from [4].

Definition 2.16. *The circuit satsifiability problem* Ckt-Sat *is the language of circuits that have a satisfying assignment.*

Moreover, it is also known that circuit satisfiability is NP-complete. We attempt to parameterize the number of 1s in the circuit assignment [3]:

Definition 2.17. *The weight of an assignment is the number of* 1*s in the input.*

Definition 2.18. *The* Weighted Circuit Satisfiability *problem* WCS($\mathcal{C}$) *is a collection of circuits* $\mathcal{C}$*, where there is an input of weight* k *that satisfies the circuit (i.e. results in an output bit of 1).*

Given a fixed value of k, there are only $\binom{n}{k}$, i.e. $O(n^k)$, different assignments to check. This means that for a fixed k we may solve this problem in polynomial-time.

It is natural to ask what kind of circuits would capture the nature of hierarchy in parameterized languages. We will consider circuits of certain depth and what is called the *weft* of a circuit [3].

Definition 2.19. *The depth of the circuit is the number of gates in the longest path from input node to output node of the circuit.*

Definition 2.20. *The weft of a circuit is the maximum number of nodes with in-degree at least 3 on a path from an input node to the output node.*

For example, if there is a path that has 2 gates (most likely includes the output node) of fan-in of more than 3, while any other path has only one such gate, then we call this circuit a weft-2 circuit. Given a circuit and its depth b and weft a, we call it a "weft-a and depth b" circuit.

We use $\mathcal{C}_{a,b}$ to denote any circuit that has weft $\leq a$ and depth $\leq b$. Therefore the language $WCS(\mathcal{C}_{t,d})$ would simply includes all circuits of weft $\leq t$ and depth $\leq d$ that have a satisfying assignment of weight k.

Definition 2.21. *A language* L *is in the complexity class* $W[t]$*,* $t \in \{1, 2, 3, \dots\}$*, if* L *can be parametrically reduced to* WCS($\mathcal{C}_{t,d}$) *for some* $d \geq 1$*.*

In essence, we define WCS($\mathcal{C}_{t,d}$) to be a $W[t]$-complete language. Additionally, the language k-Independent Set is in $W[1]$ and k-Domset is in $W[2]$, and they are complete languages for $W[1]$ and $W[2]$, respectively. The proof of these statements is non-trivial; we skip them for brevity, but they are available in [3].

Definition 2.22. *A language* L *is* $W[t]$*-hard if any language* $L' \in W[t]$ *can be parametrically reduced to* L*. A language* L *is* $W[t]$*-complete if* $L \in W[t]$ *and* L *is* $W[t]$*-hard.*

Previously we have proven that Multicolored Indset can be reduced to Dominating Set (Theorem 2.14), which essentially shows that that $W[1]$-hard language is in $W[2]$. However, we do not know whether we can reduce in the opposite direction. If so, then $W[1] = W[2]$.

Furthermore, $\text{WCS}(\mathcal{C}_{t,d})$ could be reduced to specific structures resembling an alternating circuit. Namely, the parent of AND gates are OR gates and vice versa, with NOT gates incident to input nodes. This is referred to as canonical form. Specifically, the canonical form of $\text{WCS}(\mathcal{C}_{t,d})$ is a circuit of the same output with depth t and where the gates alternate. We call this procedure t-normalization. For example, consider gates at depth i as follows: denoting C_i as the circuit having depth i and gate AND in the final layer and D_i as the circuit having depth i but with gate OR in the final layer, then C_i should be the AND of all D_{i-1}, while D_i should be the OR of all C_{i-1}. In the base case, D_0 and C_0 are symbols of only one element. This grants us the following definition [3].

Definition 2.23. *The* Weighted-t-Normalized Satisfiability *problem is the* Weighted-Satisfiability *problem with the circuit t-normalized.*

Definition 2.24. *Consider the special cases of a circuit with every input literal positive, and every input literal negative (in the sense of Boolean negation). The corresponding circuit satisfiability problems are* Weighted-Monotone-t-Satisfiability *and* Weighted-Anti-Monotone-t-Satisfiability*, respectively.*

For any $t \in \mathbb{N}$ odd, weighted Weighted-Monotone-t-Satisfiability and Weighted-Monotone-$t+1$-Satisfiability are $W[t]$-complete, and for any $t \in \mathbb{N}$ even, Weighted-Anti-Monotone-t-Satisfiability and Weighted-Anti-Monotone-$t+1$-Satisfiability are $W[t]$-complete. We omit the proof, which appears in [3].

3 Algorithmic Paradigms for Parameterized Complexity

As we have seen in the previous section, brute-force algorithms for parameterized problems sometimes face combinatorial explosion in a way that entangles the parameter k and the size of the other part of the instance n, especially those having runtime $\Omega(n^k)$. In designing algorithms conscious of parameterized complexity—in particular to place a problem in FPT—the goal is often to confine the aforementioned explosion to the parameter k. This may take the form of brute-force search *after* applying a clever reduction to decrease the size of the instance, as in *kernelization*, or in randomly restricting the space of solutions via independent trials so that a solution may be found with high probability. We consider these approaches, and others, through the lens of well-known problems.

3.1 Kernelization

The premise of kernelization is to preprocess the input to a problem in order to efficiently identify whether the instance is an easy special case of the more general problem, and, if not, apply an algorithm that exploits the structure of the problem to reduce the instance to one of smaller size. The new instance is called the kernel of the original instance; it represents the core structure of the instance and the hardness thereof, but generally with a more manageable "input" size. Then, brute-force methods can be applied to the kernel, but the preprocessing step stands to significantly diminish the amount of computation required, even though expending more time upfront. To illustrate the utility of this method, we apply an elementary kernelization technique to establish

VERTEX COVER $\in$ FPT, where the vertex cover size k is the parameter (in the proceeding, given a graph $G = (V, E)$, denote $n = |V|$, $m = |E|$) [3].

Theorem 3.1. VERTEX COVER $\in \mathsf{DTIME}(2^{2k^2} + n + m)$.

Proof. Consider the following kernelization procedure. Let $\langle G, k\rangle$ be an instance of VERTEX COVER. Initialize $G' = G$, $k' = k$. First, delete all isolated vertices from G'. Then, while G' has a vertex of degree greater than k', delete the vertex and all edges incident to it from G' and decrease k' by 1.

We claim that $\langle G, k\rangle \in$ VERTEX COVER if and only if $\langle G', k'\rangle \in$ VERTEX COVER, where $\langle G', k'\rangle$ is the resultant instance at the end of the reduction. To see this, note that isolated vertices do not cover any edges, so choosing one only decreases the budget for the vertex cover and hence does not affect the existence of a vertex cover of a given size. Also, if a vertex has degree greater than k', that means in a vertex cover of size k', in order to cover the edges incident to it, either that vertex must be selected or all of its neighbors must be, but the latter violates the budget of k'. Thus the former must hold, meaning that such a vertex cover is characterized by choosing $k' - 1$ remaining vertices that cover all edges (not incident to that vertex). Therefore, each step of the reduction does not affect whether there is a vertex cover of the contextually prescribed size, so by induction the claim holds.

Now, if $E(G') > k'^2$, since each vertex has degree at most k' by construction, it follows that k' vertices in G' will cover at most k'^2 edges, in which case $\langle G', k'\rangle \notin$ VERTEX COVER. Otherwise, $E(G') \leq k'^2$, so $V(G') \leq 2k'^2$ since each edge has two endpoints and there are no isolated vertices. Now we may simply test each of the $2^{2k'^2} \leq 2^{2k^2}$ subsets of the vertices in G' and check if one of them witnesses that $\langle G', k'\rangle \in$ VERTEX COVER. □

It is worth noting that VERTEX COVER is NP-hard, and so that must be true when restricting to the set of possible kernels for the problem in the proof of Theorem 3.1 to which we reduce. Indeed, k is still exponential in the size of k, so the above algorithm for vertex cover is superexponential, yet still polynomial in $|G|$. While the latter is often large in applications, that may not be the case for k, meaning this could still be practical. Relatedly, the algorithm actually identifies a vertex cover, since the kernel is obtained by fixing elements of the vertex cover and the kernel is solved by enumeration, so the reduction does not just constitute abstract knowledge.

Moreover, the dependence on k may be improved as follows, also demonstrating the versatility of kernelization and the opportunity for more creative approaches.

Theorem 3.2. VERTEX COVER $\in \mathsf{DTIME}(2^k(n + m))$.

Proof. Consider an edge (u, v) in the original graph; we could pick either u or v for our vertex set. Afterwards we split this graph into two parts $G - u$ or $G - v$ depending on our choice. Then we compute vertex covers of size $k - 1$ on both graph $G - u$ and $G - v$ (if they exist). Apparently this is a recursive algorithm and its time complexity is the number of "nodes" in the "tree structure". First, the depth of this tree is k since we reduce the parameter by 1 each time we branch. Therefore we have in total $O(2^k)$ amount of cases to search. And then for each "node" in the "tree" given the result of $G - u$ and $G - v$, we need to check that, if the algorithm chooses u to be in vertex cover (without loss of generality), all edge between v and $G - v$ get covered. This quantity depends on both $|V|$ and $|E|$. Therefore we need $O(|V|+|E|)$ amount of time at each node.

In all, this algorithm takes time $O((m + n) * 2^k)$ to execute (using the above notation). □

3.2 Dynamic Programming on Parameter-Dependent Subsets

In this section, we present an algorithm for Set Cover [3], which exemplifies the use of dynamic programming in finding parameterized algorithms by considering subsets described by a problem parameter, so that iterating through all of them only incurs exponentially many steps in the *parameter*.

Definition 3.3. *An instance of the* Set Cover *problem is given by* $\langle U, \mathcal{F}, k\rangle$*, where* U *is a set,* $\mathcal{F} \subset 2^U$*, and* $k \in \mathbb{N}$*.* $\langle U, \mathcal{F}, k\rangle \in$ Set Cover *exactly when there exists* $\mathcal{S} \subset \mathcal{F}$ *such that* $\cup_{S\in\mathcal{S}} S = U$ *and* $|\mathcal{S}| \leq k$*.*

In this case, our parameter is $|U|$, the size of the underlying set; this is indeed part of the size of the problem instance, but in general we can only guarantee $|\mathcal{F}|$ is $O(2^{|U|})$, which is much larger than $|U|$. Therefore, establishing fixed-parameter tractability of Set Cover in this context is still worthwhile.

Theorem 3.4. *There exists a deterministic algorithm with runtime* $2^{|U|}(|U| + |\mathcal{F}|)^{O(1)}$ *that, given an instance of* Set Cover*, returns a set cover of minimum size (if any exist) and hence can decide whether there is a set cover of size at most* k*.*

Proof. Denote $\mathcal{F} = \{F_1, \ldots, F_{|\mathcal{F}|}\}$. For $X \subset U$ and $1 \leq i \leq |\mathcal{F}|$, let $M[X, i]$ be the minimum size of a subset $\mathcal{F}' \subset \{F_1, \ldots, F_i\}$ covering X, i.e. $\cup_{F_j\in\mathcal{F}'} F_j \supset X$ (if no such subset exists, set $M[X, i] = \infty$). Tautologically, $M[\emptyset, 0] = 0$ and $M[X, 0] = \infty$ for $X \neq \emptyset$. Now for $i \geq 1$ and $X \subset U$, we claim that

$$M[X, i] = \min(M[X, i-1], 1 + M[X\backslash F_i, i-1]).$$

To see this, note that any subset of $\{F_1, \ldots, F_{i-1}\}$ covering X is a subset of $\{F_1, \ldots, F_i\}$ covering X, so $M[X, i] \leq M[X, i-1]$. Also, any subset of $\{F_1, \ldots, F_{i-1}\}$ covering $X\backslash F_i$ together with F_i forms a subset of $\{F_1, \ldots, F_i\}$ covering $X\backslash F_i \cup F_i \supset X$, so $M[X, i] \leq 1 + M[X\backslash F_i, i-1]$. Hence $M[X, i] \leq \min(M[X, i-1], 1 + M[X\backslash F_i, i-1])$.

For the reverse inequality, note that any subset S of $\{F_1, \ldots, F_i\}$ covering X either excludes F_i, in which case its size is at least $M[X, i-1]$, or it includes F_i, in which case its size is at least $1 + M[X\backslash F_i, i-1]$ (because it must also include a subset $\{F_1, \ldots, F_{i-1}\}$ covering all the elements of X not covered by F_i, and $M[X\backslash F_i, i-1]$ is the size of the smallest such subset; we add 1 for F_i). Since S was arbitrary, this statement still holds for such a set S of minimum size, so $M[X, i] \geq \min(M[X, i-1], 1 + M[X\backslash F_i, i-1])$. We have now established both inequalities, so $M[X, i] = \min(M[X, i-1], 1 + M[X\backslash F_i, i-1])$ as claimed.

Moreover, by construction, $M[U, |\mathcal{F}|]$ is the minimum size of a set cover (consisting of a subset of $\mathcal{F}$) for U, so $\langle U, \mathcal{F}, k\rangle \in$ Set Cover if and only if $M[U, |\mathcal{F}|] \leq k$. Therefore, the claimed algorithm can iterate through $X \subset U$ in increasing order of size and $1 \leq i \leq |\mathcal{F}|$ in increasing order to compute each value of $M[X, i]$, and it can keep track of whether each value is exhibited by including F_i or not to determine a set cover of minimum size. There are $2^{|U|}$ subsets of U, so the algorithm runs in time $2^{|U|}(|U| + |\mathcal{F}|)^{O(1)}$. □

Next, we see similar brute-force dynamic programming on subsets, but where the computation is affected by preceding random choices, and later we see a more intricate application of dynamic programming on a tree structure.

3.3 Randomized Methods

Despite conjectures that the true computational power of randomized algorithms is the same as deterministic algorithms (e.g. [5]), randomized algorithms are regularly employed to avoid structure and adequately sample from a set of possibilities in a way that is difficult to do a priori with deterministic decisions. Here, we handle a parameter by expressing its combinatorial richness via a palette of colors—randomly assigned to an object—that result in identifying a solution with constant probability; this is but one instance of a broader technique called *color coding*. First, we state our problem of interest, which has applications to scheduling problems and computational biology while also being an interesting NP-complete problem in its own right [3].

Definition 3.5. *An instance of the* Longest Path *problem is given by* $\langle G, k\rangle$*, where* $\langle G, k\rangle \in$ Longest Path *exactly when there exists a (simple) path on* k *vertices in* G *(i.e. there is a path whose edges are incident to exactly* k *vertices, and where each vertex is incident with exactly one edge or two adjacent edges). Here,* G *is a directed or undirected graph, and* k *is a natural number.*

For a graph G, denote $n = |V(G)|$. Our goal is to prove the following.

Theorem 3.6. *There exists a probabilistic algorithm with runtime* $(2e)^k n^{O(1)}$ *that, given an instance of* Longest Path*, (correctly) computes a path on* k *vertices in* G *with probability at least some fixed positive constant or else (possibly incorrectly) declares there is no such path.*

For each fixed value of k, then, the corresponding longest path problem is in RP; the runtime of the algorithm is exponential only in the parameter k.

To achieve this, we temporarily make the problem harder by coloring each vertex with one of k colors and only looking for *colorful paths* on k vertices (that is, for each of the k colors there is a vertex in the path with that color), but with the benefit of having fewer paths to look for. That way, we only have to keep track of subsets of colors used on the prospective path, at most 2^k, instead of subsets of vertices used in a naive brute-force solution, at most n^k. First, we show that an arbitrary subset of size k—including paths of the form we are trying to find—will be colorful with sufficiently high probability. The approach is as in [3].

Lemma 3.7. *Let* $X \subset U$ *be sets of size* k *and* n*, respectively. Let* $\chi : U \to [k]$ *be a random coloring of* U *by the "colors"* $[k]$ *obtained by coloring each element uniformly at random and independently; denote by* P *the resultant probability measure. Then* $P(\chi(X) = [k]) \geq e^{-k}$*.*

Proof. We use a counting argument. There are k^n possible colorings χ. The ones where $\chi(X) = U$ are characterized by each element of X having a different color, and the elements in $U \backslash X$ also have some color, so there are $k!k^{n-k}$ such colorings. By construction, each coloring is equally likely, so

$$P(\chi(X) = [k]) = \frac{k!k^{n-k}}{k^n} \geq e^{-k},$$

where we have used the standard estimate $k! \geq (\frac{k}{e})^k$ arising from Stirling's approximation. □

Next, we formally show how we may keep track of colors, as alluded to above, to find paths consisting of actual vertices.

Lemma 3.8. *There exists a deterministic algorithm with runtime* $2^k n^{O(1)}$ *that, given a graph* G *with vertex coloring* $\chi : V(G) \to [k]$*, outputs a colorful path on* k *vertices or (correctly) declares that there is no such path.*

Proof. We seek to construct a path by iterating over subsets of colors used on the path thus far, employing a recursive way to relate these to each other in order to progressively augment candidate paths with new colors/vertices. This goal suggests a dynamic programming approach. For $S \subset [k]$ and $u \in V(G)$, define $\text{PATH}(S, u)$ as true if there exists a (simple) path on $|S|$ vertices such that u is the last vertex traversed (not unique if G is undirected), where each vertex (under χ) has a different color in S and they exhaust the colors in S (false otherwise). For $|S| = 1$, $\text{PATH}(S, u)$ is true exactly when $S = \{\chi(u)\}$. For $|S| > 1$, if $\chi(u) \notin S$, then $\text{PATH}(S, u)$ is false; if $\chi(u) \in S$, then we have the recurrence

$$\text{PATH}(S, u) = \bigvee_{v \in V} [\text{PATH}(S \backslash \{\chi(u)\}, v) \wedge (v, u) \in E(G)].$$

To see this, note that if $\text{PATH}(S, u)$ holds, by removing u from the corresponding path, we obtain a path on $k - 1$ vertices assuming all the colors in $S \backslash \{\chi(u)\}$, where the last vertex traversed is now some $v \in V$ for which there exists an edge from v to u, and clearly this is sufficient as well. Finally, we may compute all values of PATH, $2^k n$ many, by first iterating through $1 \leq |S| \leq k$; for each subset S of that size, iterate through $V(G)$ to compute the corresponding values of PATH, which takes time polynomial in n. By construction, there exists a colorful path in G on k vertices exactly when $\text{PATH}([k], v)$ is true for some $v \in V$, and, if so, standard use of backlinks in dynamic programming may be used to obtain the path without changing the above time complexity. □

We now prove Theorem 3.6.

Proof of Theorem 3.6. The algorithm is as follows. For $i = 1, \ldots, \lceil e^k \rceil$ (independent) iterations, choose a coloring $\chi_i : V(G) \to [k]$ uniformly at random and, given this coloring, perform the algorithm described in the proof of Lemma 3.8. If any execution of this algorithm returns a (colorful) path on k vertices, return it; otherwise, declare that there is no path on k vertices.

To establish correctness, first suppose there is a path in G on k vertices consisting of vertices $V' \subset V(G)$. By Lemma 3.7, $P(\chi_i(V') = [k])$ for each i, so by independence,

$$P(\exists i : \chi_i(V') = [k]) = 1 - P(\forall i : \chi_i(V') \neq [k]) \geq 1 - (1 - e^{-k})^{e^k} \geq 1 - \frac{1}{e}.$$

In other words, with probability at least $1 - \frac{1}{e}$, one of the colorings will result in the vertices of V' constituting a *colorful* path on k vertices in G, in which case, by Lemma 3.8, our algorithm returns that colorful path (or another one on k vertices). On the other hand, by Lemma 3.8, our algorithm will only return a path on k vertices if one exists, so we need not consider the output for other iterations of the deterministic algorithm, and, if instead there is no path on k vertices in G, our algorithm correctly reports that. □

3.4 Counting-to-Search via Inclusion-Exclusion

Many combinatorial quantities can be computed in exponential-time via the principle of inclusion-exclusion. In some cases, this counting algorithm may be used as a subroutine to find such an object, leading to a parameterized algorithm for a search problem and the corresponding decision problem. We first recall the inclusion-exclusion principle.

Theorem 3.9 (Inclusion-Exclusion)**.** *Let $A_1, \ldots, A_n$ be finite sets. Then*

$$|\bigcup_{i=1}^{n} A_i| = \sum_{i=1}^{n} |A_i| - \sum_{1 \leq i < j \leq n} |A_i \cap A_j| + \sum_{1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n+1} |A_1 \cap \cdots \cap A_n|. \quad (1)$$

Corollary 3.10. *Let $A_1, \ldots, A_n \subset U$ be finite sets. Then*

$$|\bigcap_{i=1}^{n} \bar{A}_i| = |U| - \sum_{i=1}^{n} |A_i| + \sum_{1 \le i < j \le n} |A_i \cap A_j| - \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| + \cdots - (-1)^{n+1} |A_1 \cap \cdots \cap A_n|. \quad (2)$$

Note that the right-hand sides of equation (1) and equation (2) each consist of $O(2^n)$ summands, so these sums may be evaluated in a fixed-parameter tractable algorithm if n is the parameter. We leverage this fact to solve the Hamiltonian path problem, an important special case of Longest Path.

Definition 3.11. *The language* Hamiltonian Path *consists of all directed graphs $G = (V, E)$ for which there exists a (simple) path of length $|V|$, i.e. one that visits each vertex exactly once (called a Hamiltonian path)*

As alluded to above, we start by counting Hamiltonian paths (as performed in [6]).

Theorem 3.12. *There exists a deterministic algorithm with runtime $O(2^n n^{O(1)})$ that, given a graph $G = (V, E)$, $V = |n|$, counts the number of Hamiltonian paths it has.*

Proof. Let G be as in the theorem statement. The strategy is to count walks on n vertices, then use inclusion-exclusion to count only those that are simple paths. Given a graph H with adjacency matrix A, the number of walks on n vertices, W_H, derives from the well-known expression for the number of walks of length $n-1$ from vertex i to vertex j; $W_H = \sum_{i \neq j} A_{ij}^{n-1}$ [7], which clearly may be calculated in polynomial-time. Observe that for any $V' \subset V$, the walks on n vertices in G that do not visit any $v \in V'$ are precisely the walks on n vertices in $G \backslash V'$, where $G \backslash V'$ is obtained from G by deleting each $v \in V'$ and all remaining edges incident to it. Now for $i \in [n]$, let S_i be the set of walks on n vertices in G that visit the ith node of G (in some enumeration). By inspection, $\cap_{i=1}^{n} S_i$ is the set of Hamiltonian paths in G. Regarding the S_i as subsets of the set of walks on n vertices in G, by Corollary 3.10 and our above observation,

$$\begin{aligned} |\bigcap_{i=1}^{n} S_i| &= W_G - \sum_{i=1}^{n} |\bar{S}_i| + \sum_{1 \le i < j \le n} |\bar{S}_i \cap \bar{S}_j| - \sum_{1 \le i < j < k \le n} |\bar{S}_i \cap \bar{S}_j \cap \bar{S}_k| + \cdots - (-1)^{n+1} |\bar{S}_1 \cap \cdots \cap \bar{S}_n| \\ &= W_G - \sum_{i=1}^{n} W_{G \backslash \{i\}} + \sum_{1 \le i < j \le n} W_{G \backslash \{i,j\}} - \sum_{1 \le i < j < k \le n} W_{G \backslash \{i,j,k\}} + \cdots - (-1)^{n+1} W_{G \backslash \{1,\ldots,n\}}. \end{aligned}$$

The right-hand side has $O(2^n)$ summands, each of which can be calculated in $\mathsf{poly}(n)$-time using the above characterization, so the desired quantity $|\cap_{i=1}^{n} S_i|$ may be found in time $O(2^n n^{O(1)})$. □

We conclude by using this algorithm to actually find a Hamiltonian path [8].

Theorem 3.13. *There exists a deterministic algorithm with runtime $O(2^n n^{O(1)})$ that, given a graph $G = (V, E)$, $V = |n|$, computes a Hamiltonian path in G or (correctly) declares that there is no Hamiltonian path in G.*

Proof. Denote by $G' \backslash \{e'\}$ the graph obtained from G' by deleting the edge e'. The algorithm is as follows. First calculate the number of Hamiltonian paths in G. If 0, declare there is no Hamiltonian path in G. Otherwise, initialize $H = G$. While H contains more than $n-1$ edges, iterate through the edges $e \in E(H)$ that have not been considered in any preceding time step until finding an edge

e^* such that the graph $H \backslash \{e^*\}$ contains at least one Hamiltonian path, then update $H \leftarrow H \backslash \{e^*\}$. Upon termination, the remaining edges in H are declared as forming a Hamiltonian path in G.

Since each edge G is only considered at most once for deletion, the algorithm only requires $O(|E|) = O(n^2)$ calls to the algorithm in the proof of Theorem 3.12. As for correctness, note that a Hamiltonian path in G or H (at any iteration) consists of $n-1$ edges (so that each vertex is visited exactly once), so while the number of edges in H is greater than this, an edge may be deleted while preserving a Hamiltonian path in H. Lastly, such an edge will not be one already considered because then even in a graph with additional edges that edge is always part of a Hamiltonian path, so the same is true in H at the given iteration. □

A desirable property of this algorithm is that the exponential-time computation only requires storing one value at a time—the preexisting sum (and index values), and so the algorithm takes polynomial space. This is in contrast to the exponential space required by dynamic programming approaches to maintain the table of look-up values, including one possible algorithm for Hamiltonian Path, which is not something exhibited even by the naive brute-force search algorithm [6].

3.5 Bounded Search Tree

Another one of the most basic technique we use to get fixed-parameter tractability is the method called bounded search tree. The bounded search tree is used to solve problems by breaking them down into a sequence of decisions, which forms a search tree. Each step of the algorithm considers various choices and then split the problem into smaller sub-problems that are addressed individually. The execution of such a "branching" algorithm can be represented as traversing a search tree until a solution is found in one of the leaves. Since the tree we considered here is bounded by the height and the number of children nodes, we call it a bounded search tree. We will illustrate this technique through modeling the hitting set problem.

Definition 3.14. *A hypergraph $\mathcal{G}$ is a pair $\mathcal{G} = (V, E)$, where V is the set of vertices. E is the set of hyperedges, each of which is a non-empty subset of V.*

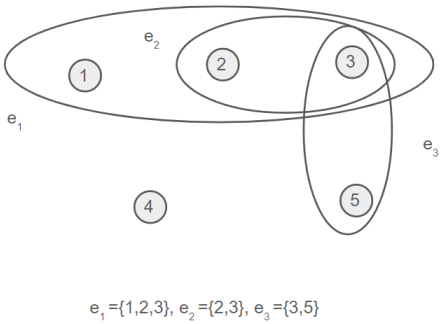

Figure 1: Hypergraph $\mathcal{G}$ with hyperedges $E = \{e_1, e_2, e_3\}$, $V = \{1, 2, 3, 4, 5\}$

Definition 3.15. *A hitting set of the hypergraph $\mathcal{G} = (V, E)$ is a set $S \subseteq V$ such that for all $e \in E$, $S \cap e \neq \emptyset$. The* Hitting Set *problem is defined as the problem: given a hypergraph $\mathcal{H}$ and cardinality $k \in \mathbb{N}$, decide whether $\mathcal{H}$ has a hitting set of k elements.*

The parameterized Hitting Set problem has a parameter: $k + d$, where k is the maximum cardinality, $d := \max\{|e|, e \in E\}$. We will prove the following theorem for the parameterized Hitting Set problem. Notice that for this parameterization, we indeed want to view the parameters k and d separately. However, to simplify the analysis, we give only one parameter instead. We prove the theorem using induction, as performed in [2].

Theorem 3.16. *There exists an algorithm such that given hypergraph $\mathcal{G}$, cardinality k for the hitting set and cardinality d for the hyperedge, it can compute a list of all minimal hitting sets of $\mathcal{G}$ in time $O(d^k \cdot k \cdot |H|)$, where $|H| := |V| + \sum_{e \in E} |e|$.*

Proof. To prove this theorem, we aim to design an algorithm that returns a set S of hitting sets for $\mathcal{H}$, each with cardinality less than or equal to k. First, we need to demonstrate that this algorithm yields the correct output. We prove this by induction on the cardinality of the hyperedges of $\mathcal{H}$.

Algorithm 2 Algorithm enumerating hitting sets, EnuHS($\mathcal{H}, k, X$)

Require: $\mathcal{H} = (V, E)$ hypergraph, $k \geq 0$, set X of vertices (not of $\mathcal{H}$)
1: **if** $E = \emptyset$ **then return** $\{A\}$
2: **else if** $k = 0$ **then return** $\emptyset$
3: **else**
4: choose an edge $e \in E$
5: $S \leftarrow \emptyset$
6: **for** each vertex $v \in e$ **do**
7: $V_v \leftarrow V \setminus \{v\}$
8: $E_v \leftarrow \{e \in E \mid v \notin e\}$
9: $\mathcal{H}_v \leftarrow (V_v, E_v)$
10: $S \leftarrow S \cup \text{EnuHS}(\mathcal{H}_v, k - 1, X \cup \{v\})$
return S

Base case: When $|E| = 0$, there are no hyperedges in the hypergraph. Therefore, the only minimal hitting set is the empty set, which is trivially enumerated by the algorithm.
Induction step: Assume the statement is true for all hypergraphs with fewer than $|E|$ hyperedges. Now consider a hypergraph with $|E|$ hyperedges.

- If $E = \emptyset$ then the algorithm returns $\{X\}$, which is a hitting set by definition since there are no edges to hit.
- If $k = 0$ then the algorithm returns $\emptyset$, indicating that no hitting set of size up to k is possible.
- Otherwise, the algorithm chooses an edge $e \in E$ and for each vertex $v \in e$, it computes $V_v = V \setminus \{v\}$, $E_v = \{e \in E | v \notin e\}$, and $\mathcal{H}_v = (V_v, E_v)$. It then recursively calls itself with the smaller hypergraph $\mathcal{H}_v$ and $k - 1$, adding v to the set X.

The proof must then show two things:

1. For all sets S' in S, we have $X \subseteq S'$ and $S' \backslash X$ is a hitting set of $\mathcal{H}$ with cardinality $\leq k$.
2. For each minimal hitting set S' of $\mathcal{H}$ with cardinality $\leq k$, $S' \cup X$ is contained in S.

We prove these two claims separately:

1. Consider a non-empty hypergraph $\mathcal{H}$ and a set X disjoint from V. Let S be the set returned by EnuHS$(\mathcal{H}, k, X)$. Given $S' \in S$, consider the edge e chosen in the algorithm 1. Then choose a $v \in e$, let S_v be the set returned by EnuEHS$(\mathcal{H}_v, k-1, X \cup \{v\})$. Then S' is included in S during the execution of line 10, implying $S' \in S_v$. By the induction hypothesis, $X \cup \{v\} \subseteq S'$ and $S' \backslash (X \cup \{v\})$ is a hitting set of $\mathcal{H}_v$. We must show that $S' \backslash X$ is a hitting set of $\mathcal{H}$. Consider an edge e' in E. We know that if $e' = e$, then $v \in S'$, also implies $e' \cap (S' \backslash X) \neq \emptyset$. Otherwise if $e' \neq e$, then $e' \cap (S' \backslash (X \cup \{v\})) \neq \emptyset$ by the induction hypothesis. Therefore $e' \cap (S' \backslash X) \neq \emptyset$. This proves that $S' \backslash X$ is a hitting set for $\mathcal{H}$. Thus, it has cardinality at most k.

2. Let S' be any minimal hitting set of $\mathcal{H}$ with $|S'| \leq k$. Since S' is minimal, for every $v \in S'$, $S' \backslash \{v\}$ is not a hitting set of $\mathcal{H}$. Therefore, exists an edge $e_v \in E$ such that $e_v \cap (S' \backslash \{v\}) = \emptyset$. Thus v is essential for S' to hit e_v. Now, consider EnuHS$(\mathcal{H}, k, X)$. For each $v \in S'$, because v is part of a minimal hitting set, the algorithm will eventually choose the hyperedge e_v in line 4 and include v in the hitting set being constructed. This inclusion happens in the recursive call EnuHS$(\mathcal{H}_v, k-1, X \cup \{v\})$, where $\mathcal{H}_v$ is the hypergraph obtained from $\mathcal{H}$ by removing v from its hyperedges. By the induction hypothesis, since $S' \backslash \{v\}$ is a hitting set for $\mathcal{H}_v$ and it is minimal for $\mathcal{H}_v$, the set $(S' \backslash \{v\}) \cup X$ is included in the set S_v generated. Thus $S' \cup X$ is contained in the cumulative set S that is the union of all sets S_v for each $v \in S'$. Therefore, $S' \cup X$ is guaranteed to be in S, which is constructed by the union of all sets S_v generated by the recursive calls of the algorithm for each vertex $v \in S'$.

We analyze the running time of EnuHS$(\mathcal{H}, k, X)$ by establishing a recurrence relation. We let $T(k, n, d)$ denote the maximum running time of EnuHS$(\mathcal{H}, k, X)$ for a hypergraph H' with $|H'| \leq n$, and each hyperedge e in E' has $|e| \leq d$, and we enumerates hitting sets of size at most k.
The base case of the recurrence is straightforward. When $k = 0$ or $|E'| = 0$, the algorithm terminates immediately, thus $T'(k, n, d) = O(1)$ for these cases. For $k > 0$ and $|E'| > 0$, the algorithm chooses a hyperedge e and recursively run for each vertex v in e. Each recursive call attempts to construct a hitting set by including v and reducing the problem size by excluding v from the vertex set and reducing k by 1. Therefore we have the recurrence:

$$T(k, n, d) \leq d \cdot T(k-1, n, d) + O(n)$$

We claim that the running time is bounded by $T(k, n, d) \leq (d^k - 1) \cdot c \cdot n$, where c is a constant. We prove this by induction on k.

1. Base case: For $k = 0$, the claim holds trivially as $T'(0, n, d) = O(1)$.

2. Assume the claim holds for $k-1$. Then for $k > 0$,

$$\begin{aligned} T'(k, n, d) &\leq d \cdot T'(k-1, n, d) + c \cdot n \\ &\leq d \cdot ((d^{k-1} - 1) \cdot c \cdot n) + c \cdot n \\ &= (d^k - d) \cdot c \cdot n + c \cdot n \\ &= (d^k - d) \cdot c \cdot n + c \cdot n \\ &= (d^k - 1) \cdot c \cdot n \end{aligned}$$

   Thus, by induction, the running time $T'(k, n, d)$ is $O(d^k \cdot |H|)$.

Therefore, since we also know the search tree, which is corresponds to the recursion tree produced by the recursive algorithm, traversed by the algorithm has parameters d and k, and the minimal hitting set with cardinality $\leq k$ appears in $\mathcal{S}$, we can check the minimality in time $O(k \cdot n)$. Therefore, the total running time is $O(d^k \cdot k \cdot |H|)$. □

Therefore, given the previous theorem, we may conclude that the parameterized Hitting Set problem is in FPT.

3.6 Dynamic Programming on Tree Decomposition

In the area of parameterized complexity, besides kernelization, randomization and bounded search tree, the method of dynamic programming on tree decomposition has been introduced as a similarly efficient approach for tackling parameterized problems. When analyzing similarities of graphs, tree width is a useful parameter. A tree decomposition of the graph G is a pair $(\mathcal{T}, (B_t))$, where $\mathcal{T}$ is a tree, and B_t is a family of subsets of V, which we call bags. These bags ensure that every node and edge in G is included in at least one bag, and any node that appears in two bags must be contained in all bags between these two bags. Then, the width of the decomposition $(\mathcal{T}, (B_t))$ is defined as $\max |B_t| - 1$, which is the size of the largest bag minus one. The tree width, denoted as $tw(G)$, is the minimum width among all tree decompositions of G. By restricting the input to graphs with bounded tree width, many hard algorithmic problems can be solved in linear time using dynamic programming on tree decompositions. We will illustrate this technique by considering the 3-Colorability problem. We define the parameterized 3-Colorability as follows: given instance graph G, and the parameter $tw(G)$, decide whether G is 3-colorable.

Theorem 3.17. *Parameterized* 3-Colorability $\in$ FPT.

Before presenting the proof for Theorem 3.17, we first introduce Bodlaender's Theorem [9], which is useful for the computation of a tree decomposition. A detailed proof of the Bodlaender's Theorem will be skipped here. With the Bodlaender's Theorem, we use a similar approach as shown in [2].

Theorem 3.18 (Bodlaender's Theorem)**.** *There exists a polynomial p and an algorithm that, for a given graph G, computes a tree decomposition of G with tree width k in time at most $2^{p(k)} \cdot n$.*

Proof of Theorem 3.17. We aim to show that the 3-Colorability problem, given a graph $G = (V, E)$ with tree width at most k, is solvable in linear time. We start by finding a tree decomposition $(T, \{B_t\}_{t \in T})$ of G of width $\leq k$. Before proving the theorem, we firstly need to define a small tree decomposition. A tree decomposition $(T, \{B_t\}_{t \in T})$ of a graph is considered small if the subsets of vertices associated with any two different nodes $(t, t' \in T)$ in the tree are $B_t, B_{t'}$, and we have $B_t \not\subseteq B_{t'}$. We notice that for any two nodes t and t', and the set of vertices of t is the subset of the set of vertices of t', then this subset relationship also holds for any node on the undirected path from t to t'. Therefore, we can also define a small tree decomposition as: the decomposition is small if and only if for every edge in the tree that connects two nodes, neither node's subset is a subset of the other.

Claim 3.19. *Every graph G has a small tree decomposition with width $tw(G)$*

Proof. Consider a tree decomposition $(T, \{B_t\}_{t \in T})$ of G. Begin with the leaves of the tree T. If a leaf node l has a parent node p, such that $B_l \subseteq B_p$, we contract the edge connecting l to p, and remove l from the tree. Let $B'_p = B_p$ be the new bag for the parent node. Notice that this contraction does not increase the width of tree decomposition. For internal nodes, if there exists an edge (t, u) such that either $B_t \subseteq B_u$ or $B_u \subseteq B_t$, we contract this edge and take the union of

the bags, repeating this process for all internal edges. At each contraction step, we can assume the new bag does not exceed the original width. It is a valid assumption because we only take unions of bags where one is a subset of the other. Therefore the width of the union doesn't increase since each contraction involves only local operations on the tree and its bags, and the number of contractions is at most $|V| - 1$. The resulting tree decomposition $(T', (B'_t)_{t \in T'})$ is small, because no bag is a proper subset of another, and it has the same width as the original decomposition. □

Therefore, combining Claim 3.19 and Theorem 3.18, we can compute a small tree decomposition $(T, \{B_t\}_{t \in T})$ with width $\leq k$. We denote the leaves of T by L and proceed by inductive computation of partial solutions for the subgraphs induced on vertex sets B_t.

- Let $Col(t)$ denote the set of all proper 3-colorings of the subgraph of G induced by B_t. These colorings can be represented by mappings $f : B_t \to \{1, 2, 3\}$.
- Let $Extcol(t)$ be the set of all colorings in $Col(t)$ that can be extended to a 3-coloring of the entire graph G.

Notice that a graph G is 3-colorable if and only if $Extcol(r) \neq \emptyset$, where r is the root of T.

Computational Details: For a node t in T, $Col(t)$ is computed in $O(3^{k+1} \cdot k^2)$ time by enumerating all mappings and filtering out non-proper colorings. For leaves $l \in L$, $Extcol(l) = Col(l)$. For internal nodes, we compute $Extcol(t)$ by ensuring consistency with colorings of adjacent nodes. Let $t_1, .., t_m$ be the children of T. Then

$$Extcol(t) = \bigcap_{i=1}^{m} \{f \in Col(t) : \exists f_i \in Extcol(t_i) \text{ with } f|_{B_t \cap B_{t_i}} = f_i|_{B_t \cap B_{t_i}}\}. \tag{3}$$

The running time for $Extcol(t)$ is $O(3^{2(k+1)} \cdot k \cdot m)$ per edge of the tree. Therefore, we proved that the parameterized problem is in FPT. □

4 Discussion

It is worth noting that there is a disparity between the parameterized 3-Colorability problem we defined in Section 3.6 and a true parameterized problem in the sense of the definition in Section 2. The reason is that $tw(G)$ cannot be computed in polynomial-time unless P = NP. Arnborg, Corneil, and Proskurowski proved in 1987 that determining whether a graph G has a tree width of at most k is NP-complete [10]. One of the most recent papers by Korhonen and Lokshtanov shows that, given a graph G with n vertices and an integer k, there is an algorithm that runs in $2^{O(k^2)}n^{O(1)}$ time to output a tree decomposition of G with width at most k [11]. Therefore, since the problem is fixed-parameter tractable with respect to tree width itself, we weakened our assumption to be able to talk about more NP-hard questions.

One interesting feature of the parameterized algorithms we have presented is that permitting exponential dependence on the parameter (e.g. in the sense of FPT) lends itself to brute-force solutions, just of a subtler nature than enumerating *every* possible solution. As a result, in many of the algorithms presented above, including for Vertex Cover and Hamiltonian Path, not only the decision problem at hand is solved but also the search version of the problem. This gives parameterized complexity applications to combinatorial optimization, and is non-trivial in that efficient decision-to-search reductions are not always known to exist. For example, the problem of deciding

if a number is composite, COMPOSITE, is solvable in polynomial-time, whereas the problem of finding a non-trivial factor of a composite number, COMPOSITESEARCH, is believed unlikely to have a polynomial-time algorithm [12].

In that vein, parameterized complexity speaks to the core difficulty of computational problems, ones for which we do not know how to do much better than enumeration in the worst case. This is similar to our interpretation of NP as "non-deterministic" polynomial-time and our understanding of various NP-hard problems. However, parameterized complexity, especially FPT, provides a more precise, quantitative description of the algorithmic intractability of a problem that causes exponential blow-up: a parameter—something that just might be well-understood or manageable. In some sense, then, parameterized complexity offers possibilities in the face of (complexity theoretic statements of) improbability.

5 Appendix

To see a video presentation based on this manuscript, refer to the EECS 574, Fall 2023 Piazza page.

References

[1] Ignasi Sau. *Introduction to parameterized complexity - LIRMM*. 2018. URL: https://www.lirmm.fr/~sau/talks/Intro-PC-UFMG.pdf.

[2] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 3540299521.

[3] Marek Cygan et al. *Parameterized Algorithms*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 3319212745.

[4] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521424267.

[5] Russell Impagliazzo and Avi Wigderson. "P = BPP If E Requires Exponential Circuits: Derandomizing the XOR Lemma". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 220–229. ISBN: 0897918886. DOI: 10.1145/258533.258590. URL: https://doi.org/10.1145/258533.258590.

[6] Ryan Williams. *Algorithms for Finding Long Paths*. URL: https://people.csail.mit.edu/virgi/6.s078/lecture17.pdf.

[7] Andrew Duncan. "Powers of the adjacency matrix and the walk matrix". In: (2004).

[8] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. 1st. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN: 364216532X.

[9] H. L. Bodlaender. "A linear-time algorithm for finding tree-decompositions of small treewidth". In: *SIAM Journal on Computing* (1996).

[10] A. Proskurowski S. Arnborg D. Corneil. "Complexity of finding embeddings in a k -tree". In: *Siam Journal on Algebraic and Discrete Methods* (1987).

[11] Daniel Lokshtanov Tuukka Korhonen. "An Improved Parameterized Algorithm for Treewidth". In: (2023). URL: https://arxiv.org/abs/2211.07154.

[12] Valentine Kabanets. *P, NP, and "search to decision" reductions.* URL: https://www.sfu.ca/~kabanets/308/lectures/lec12.pdf.