arXiv:2606.02492v1 [cs.CC] 1 Jun 2026

$O(n) + f(k)$: Truly Linear FPT

Benjamin Merlin Bumpus
Instituto de Matemática e Estatística, Universidade de São Paulo. Rua do Matão, 1010 — 05508–090, São Paulo, SP, Brasil.

Rod Downey
Victoria University of Wellington, School of Mathematics and Statistics, PO Box 600 Wellington, New Zealand

Tala Eagling-Vose
Department of Computer Science, Durham University, Durham, UK

Jessica Enright
School of Computing Science, University of Glasgow

Michael R. Fellows
Institute of Informatics, University of Bergen (Emeritus), Bergen, Norway

David C. Kutner
Department of Computer Science, Durham University, Durham, UK

Laura Larios-Jones
School of Computing Science, University of Glasgow, Glasgow, UK

Barnaby Martin
Department of Computer Science, Durham University, Durham, UK

Frances Rosamond
Institute of Informatics, University of Bergen (Emerita), Bergen, Norway

Ella Yates
School of Computing Science, University of Glasgow, Glasgow, UK

Abstract

Parameterized complexity has always been concerned with practical computing: by confining combinatorial explosion to a secondary parameter k, one can uncover why and how many NP-hard problems are effectively tackled in practice. Today, however, the scale of data has changed: scientists study Big Data, which is so large that even quadratic dependence in the total input size n is unaffordable. Therefore, what constitutes a practical algorithm has also changed. Classically, parameterized complexity is blind to the difference between defining fixed parameter tractability multiplicatively (i.e. $f(k) \cdot n^c$) or additively (i.e. $f(k) + n^c$). But what if the constant c is one and we require true linearity, is this distinction still inconsequential? Here, we define and explore Truly Linear FPT (TLFPT) – that is $O(n) + f(k)$ – and show that it is a strict subset of Linear FPT (LFPT) – that is $O(n) \cdot f(k)$ – via diagonalization.

Populating TLFPT requires careful consideration of linear-time algorithmics and data structures. We meet many inhabitants of TLFPT: SAT, Vertex Cover, Min-Max Matching, $(n-k)$-Coloring, Diverse Pair of Matchings, k-Path, and H-Coloring. Our parameterizations are equally varied. Beyond classical parameters like solution size, we leverage two parameters, treedepth and BFS-width, which are particularly well-suited to the TLFPT regime. We do so by developing techniques based on depth- and breadth-first search.

For parameterized complexity to be of service to the scientific community, we need to contend with Big Data. For sufficiently large inputs, FPT beyond linear may not suffice. Thus, there is a practical and theoretical need for more ambitious goals. TLFPT is a first step forward.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Theory of computation → Complexity classes; Theory of computation → Parameterized complexity and exact algorithms; Mathematics of computing → Graph algorithms; Theory of computation → Dynamic programming; Theory of computation → Streaming, sublinear and near linear time algorithms



© Benjamin Bumpus, Rod Downey, Tala Eagling-Vose, Jessica Enright, Michael R. Fellows, David C. Kutner, Laura Larios-Jones, Barnaby Martin, Frances Rosamond, and Ella Yates;
licensed under Creative Commons License CC-BY 4.0



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Truly Linear FPT, fixed-parameter tractability, parameterized complexity, graph algorithms, kernelization, dynamic programming.

Digital Object Identifier 10.4230/LIPIcs...

Funding *Jessica Enright*: Partly supported by EPSRC grants EP/T004878/1 and EP/V032305/1
David C. Kutner: Partly supported by EPSRC grant EP/T004878/1, Multilayer Algorithmics to Leverage Graph Structure (MultilayerALGS).
Barnaby Martin: supported by Leverhulme Trust Research Project Grant RPG-2024-182.

For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising from this submission. No data were created or analyzed in this work.

1 Introduction

There is a simple premise at the core of parameterized complexity: it is unrealistic to quantify the running time of algorithms solely in terms of the total input size n. In many cases there is a smaller, problem-relevant quantity k to which combinatorial explosion can be confined. From this perspective, an efficient algorithm is one running in time $f(k) \cdot n^c$, where f is a function independent of n and the constant c is independent of both n and k. This is *fixed-parameter tractability* (FPT).

This guiding philosophy is far from a mere mathematical curiosity; it bears genuine practical weight. Even for modest values of n and k, the difference between an n^k running time and a $2^k n$ running time can span dozens of orders of magnitude [10]. In other words, isolating the exponential dependence within k is not just elegant theory, but is often the only hope for obtaining an algorithm that has guarantees of both correctness and a running time that terminates within the remaining thermodynamic lifetime of the universe.

At this point, an observer might reasonably ask: if the goal is to design provably efficient algorithms, why insist on a running time *multiplicative* in n of the form $n^c \cdot f(k)$? Would the seemingly stronger bound $n^c + f(k)$ be even better, especially when n is expected to dwarf k? This instinct is sound, but in the classical framing of parameterized complexity, the distinction barely matters: the additive and multiplicative notions of FPT coincide up to polynomial factors[1] in n.

However, today inputs can be so large that even a *single* superlinear pass over one is infeasible. Applications to giant instances were the subject of some discussion at the recent Dagstuhl seminar "Recent Trends in Graph Decomposition" [28]), where the present paper's foundations were conceived. When linear-time processing is the limit of feasibility, there is a natural question that classical parameterized complexity did not originally need to confront:

> *Forced to incur only* ***linear*** *dependence on n, do the additive and multiplicative definitions of FPT remain equivalent?*

Our answer is a definitive **no**: we draw a sharp distinction between *linear FPT* (LFPT)[2] – algorithms running in time $n \cdot f(k)$ – and *truly linear FPT* (TLFPT) – algorithms running in time $O(n) + f(k)$. Furthermore, we show that TLFPT is a strict subset of LFPT, by diagonalization, yielding the refined hierarchy

$$\text{TLFPT} \subset \text{LFPT} \subset \text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \dots$$

Populating TLFPT requires careful consideration of the assumed model of computation (cf. [22]). As is standard in many fine-grained contexts, we work in a word-RAM model. Furthermore, mirroring the classical FPT/kernelization equivalence, we show that being in TLFPT corresponds exactly to admitting kernelization running in time $O(n + k)$[3]. This insight lets us identify problems that are already known to be in TLFPT (e.g. Max Leaf Spanning Tree on weighted graphs [25] and d-Hitting Set on hypergraphs [1]). Linear time kernelization[4] has also been applied to problems in P. For example, Maximum

[1] A problem admits a $\text{poly}(n) + f(k)$ algorithm if and only if it admits a $\text{poly}(n) \cdot f(k)$ algorithm.
[2] Sometimes referred to as fixed-parameter linear (FPL) [26].
[3] As we shall see, any problem for which a kernel is computable in time $O(n + f(k))$ for some $f : \mathbb{N} \to \mathbb{N}$ also admits a truly linear-time kernelization, i.e. one running in time $O(n + k)$.
[4] In the literature, "linear time" has varyingly been used to refer to running times of form $O(n \cdot f(k))$ where f is linear [33], polynomial but nonlinear [21] (there called "polynomial-linear FPT"), or superpolynomial [2]. In the formalism of [10], "kernelization" entails at most polynomial dependency on k.

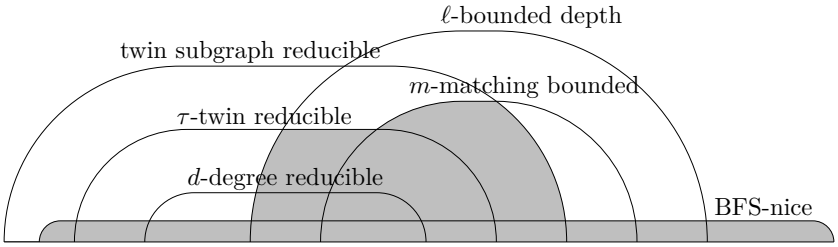

Figure 1 A graphical summary of our results. Containment of the properties we will use to categorize problems within in this paper. The shaded areas show TLFPT results.

Matching has been shown to have such a kernelization [33] and this result is bolstered by experimental evidence of its applicability [29].

Here, we identify further problems that inhabit TLFPT by establishing appropriate meta-theorems. We summarize the problems, parameters and methods we use below. These examples illustrate that truly linear FPT

is not prohibitively restrictive,

but a robust and attainable algorithmic goal.

Problems	**Parameter**	**Meta-theorem applied**
Vertex Cover	size of the vertex cover	Theorem 18 (also [10])
Dominating Set	vertex cover number	Theorem 26
Min–Max Matching	size of the matching	Theorem 26
Diverse Pair of Matchings	$\lvert M_1 \Delta M_2 \rvert$	Theorem 26
$(n-k)$-Coloring	k	Theorem 26
SAT	incidence treedepth	Theorem 42
k-Path	k	Theorem 42
k-Vertex Ranking (Treedepth)	k	Theorem 42
H-Coloring	BFS-width	Theorem 52

TLFPT is useful: Big Data and Vertex Cover We continue in the tradition of illustrating the value of FPT with a concrete example using Vertex Cover, which asks, given a graph $G = (V, E)$ and integer k, whether there exist k vertices in the graph which together cover all edges [8, 10, 11]. In those textbooks' motivating examples, values of n ranged from 1000 to 10000, with of k between 10 and 70. In 2026, major scientific endeavors are supported by the processing of petabytes of data [4]: for our example, we take n equal to a hundred billion and $k = 100$. The state-of-the art algorithm for Vertex Cover [5] runs in TLFPT time[5]: $O(1.27^k + |V| + |E|)$. Our example values of n and k make stark the distinction between TLFPT and LFPT: an algorithm running in time $1.2738^k \cdot n$ would take 10^{21} operations to terminate; an algorithm running in time $1.2738^k + n$ would take only 10^{11} operations to terminate. For an average laptop, this is the difference between minutes and millennia.

[5] The runtime of the algorithm as presented in [5] is $O(1.2738^k + kn)$, but n denotes the number of vertices; in the definition of TLFPT, n denotes the input size (i.e. $|V| + |E|$, for a graph). The term kn in the expression comes entirely from Buss kernelization, which takes truly linear time $O(|V| + |E|)$ [10].

Technical preliminaries Let $[n] := \{1, 2, \ldots, n\}$ and $[i, j] := \{i, i+1, \ldots, j\}$. Unless stated otherwise graphs $G = (V, E)$ are simple, loopless and undirected. The *union* of graphs (V, E_1) and (V, E_2) is $(V, E_1 \cup E_2)$. We refer the reader to Garey and Johnson [20] for the standard definitions of a maximal matching and dominating set, and the corresponding decision problems. Given a loopless graph H, an *H-coloring* of a graph G is a graph homomorphism $G \to H$. When H is the complete graph on n vertices, we speak of a *proper n-coloring.* The H-coloring problem asks if a given graph G admits an H-coloring. For convenience, we include the definition of a parameterized decision problem below (see [11] for background in parameterized complexity).

▶ **Definition 1** ((Parameterized) decision problem). *A parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$. Given an instance $I = (x, k)$, we call k the parameter. The kth slice Π_k of a parameterized problem Π is a language $L \subseteq \Sigma^*$ with the property that $(x, k) \in \Pi \iff x \in \Pi_k$.*

Related work Many works in LFPT are directly relevant to the problems considered here [1, 16, 29, 33, 34, 36]; further examples include [25, 21, 24, 30]. Many problems that cannot be solved in linear time can be solved in *almost linear* time, for example, in $O(n \cdot \mathit{polylog}(n))$. This is not only true of Sorting but also, e.g., for deciding first-order properties in graph classes with locally bounded expansion [12]. Note that in our context, since $\log(n)$ dominates k, $f(k) + n \cdot \mathit{polylog}(n)$ and $f(k) \cdot n \cdot \mathit{polylog}(n)$ define the same class.

2 Technical Foundations for TLFPT

We start our technical contributions by defining a class of algorithms and a related class of problems, and some important foundational notions for dealing with these classes.

▶ **Definition 2** (TLFPT). *An algorithm is Truly Linear Fixed Parameter Tractable (TLFPT) if, on input (I, k) with $|I| = n$, it runs in time $O(n) + f(k)$. TLFPT denotes the class of all parameterized problems admitting a TLFPT algorithm.*

▶ **Definition 3** (TLFPT-reduction). *Let Π and Γ be two parameterized decision problems, a map $r : \Pi \to \Gamma$ is a TLFPT-reduction if there exists a function f such that:*

- *$r(x, k)$ terminates in time $O(|x| + f(k))$ for each (x, k) in the set of instances of Π, and returns an instance (x', k') of Γ,*
- *$(x, k) \in \mathrm{YES}(\Pi) \iff (x', k') \in \mathrm{YES}(\Gamma)$, and*
- *$k' \leq f(k)$.*

We say that Π is TLFPT-reducible to Γ if there exists a TLFPT-reduction from Π to Γ.

In the next lemma, we show that TLFPT-reductions are transitive, and that reducing to a destination problem that is in TLFPT implies that that source problem is also in TLFPT.

▶ **Lemma 4.** *TLPFT-reductions are transitive. In addition, if parameterized problem Π is TLFPT-reducible to parameterized problem Γ and $\Gamma \in$ TLFPT, then $\Pi \in$ TLFPT.*

Proof. First we show transitivity, suppose a parameterized problem Π is TLPFT-reducible to parameterized problem Γ and Γ is TLFPT-reducible to parameterized problem Σ; we will show that Π is TLFPT-reducible to Σ. Let r be a TLFPT-reduction from Π to Γ and q be a TLFPT-reduction from Γ to Σ. Then we define a function $s : \Pi \to \Sigma$ by $s((x, k)) = q(r((x, k)))$ and show that our conditions hold. Let f be a function such that:

- $r((x, k))$ terminates in time $O(|x| + f(k))$ and has output (x', k') satisfying $k' \leq f(k)$, and

- $q((x', k'))$ terminates in time $O(|x'| + f(k'))$ and has output (x'', k'') satisfying $k'' \leq f(k')$

Then $s((x, k)) = q(r((x, k)))$ terminates in time $O(|x| + f(k) + f(f(k))$ and has output (x'', k'') satisfying $k'' \leq f(k') \leq f(f(k))$. Also, $(x'', k'') \in \text{YES}(\Sigma) \iff (x', k') \in \text{YES}(\Gamma) \iff (x, k) \in \text{YES}(\Pi)$ and thus s is TLFPT-reduction from Π to Σ.

Now we show that a TLFPT-reduction from Π to $\Gamma \in$ TLFPT gives us $\Pi \in$ TLFPT. Let r be a TLFPT-reduction from Π to Γ and $\mathcal{A}$ be a TLFPT algorithm solving Γ. Let f be a function such that both $\mathcal{A}$ and r run in time $O(n + f(k))$ and the instance (x', k') returned by $r((x, k))$ satisfies $k' \leq f(k)$.

For an instance (x, k) of Π, it is possible to compute $r((x, k)) = (x', k')$ and then to compute $\mathcal{A}(x', k')$. The runtime of $r((x, k))$ is clearly TLFPT in the size of the input instance; the runtime of $\mathcal{A}(x', k')$ is TLFPT in (x', k') and therefore is $O(|x'| + f(k')) \leq O(f(k) + |x| + f(f(k)))$ which is also TLFPT. ◀

The word-RAM architecture Results in the truly linear paradigm are highly sensitive to the choice of model. We use the word-RAM model [18, 19], as it is the canonical model for the analysis and design of fine-grained algorithms [14, 22]. More limited models of computation in this context are neither as widely studied nor as practically relevant, since they cannot necessarily support even depth-first search in linear time. Briefly, word-RAM is a unit-cost RAM with the additional property that the maximum number of bits stored in any single register – called the *word size* – is $\log n$, where n is length of the input. Thus word-RAM prohibits unbounded parallelism wherein constant-time operations are applied to words of length polynomial in the size of the input.

On the other hand, "word-level parallelism" may be exploited [23].

Such parallelism includes, for example, taking the bitwise AND of two bitstrings each of length ℓ in time $O(1)$ rather than time $O(\ell)$. A more powerful example: let $f : \{0,1\}^{\ell_1} \to \{0,1\}^{\ell_2}$ be some function stored as a lookup table with 2^{ℓ_1} entries, with ℓ_1, ℓ_2 at most than the word size of our RAM. Then for any $x \in \{0,1\}^{\ell_1}$ we may retrieve $f(x)$ in $O(1)$ time.

As is standard in parameterized complexity, we can assume that n is arbitrarily larger than k, and in particular that $\log n > f(k)$ for any function f (this statement is made rigorous in the appendix). This allows us, for example, to represent a graph of size k with a single word, and operate on it in constant time using bitwise operations.

▶ **Lemma 5.** *When considering the (non-)existence of a word-RAM algorithm for a parameterized problem* Π*, it suffices to fix any function* f *and consider only those instances* (x, k) *of* Π *satisfying that* $|x| > 2^{f(k)}$*.*

Proof. We shall show that for any function f, a word-RAM algorithm $\mathcal{A}$ deciding problem Π in TLFPT time exists if and only if there exists a RAM algorithm $\mathcal{B}$ with $\max(f(k), \log n)$-bit registers deciding Π in TLFPT time. The statement of the lemma follows directly from this. ($\Longleftarrow$): for any input (x, k) of the algorithm, we distinguish between two cases:

- if $f(k) \leq \log n$, then $\mathcal{B}$ can directly be run on a word-RAM with no changes to its behavior or runtime.
- if $f(k) > \log n$, then $\mathcal{B}$ can be simulated in a word-RAM (possibly much more slowly). The size of n is bounded by a function of k (specifically, $n < 2^{f(k)}$), so the runtime of the simulation is also bounded by a function of k.

Note that this does not require that f is computable; simply attempt to run $\mathcal{B}$ in the word-RAM model, and if ever a register of length greater than $\log n$ is needed then this

witnesses that $f(k) > \log n$.
($\implies$): trivial direction, since $\mathcal{A}$ itself already decides Π using $\log n$-bit registers (which can of course be simulated by larger registers). ◀

Note that Lemma 5 entails that a word size of at least $\max(f(k), \log|x|)$ can be assumed.

TLFPT is Strictly Contained in LFPT Our proof mirrors that given in [6] by Cook and Reckhow to prove the time hierarchy theorem for RAMs. Our task is to make the complexity two-dimensional (i.e. in both n and k) and tailor the discussion to word-RAMs.

Suppose we have an encoding α which maps every string over $\{1, 2\}$ to some word-RAM. We say that the word w *encodes* the word-RAM P_w which it maps to under α. We also assume that for each word-RAM M, there are infinitely many strings encoding M. In particular, it is possible to pad w to increase its length without affecting the word-RAM P_w that it encodes. Let $f \in \omega(1)$ be a computable function[6]. We define the following language $A = A(f)$.

$$\mathbf{A} = \{(w, k) : P_w \text{ with input } w \text{ halts within time } f(k) \cdot n \text{ without accepting } w\}$$

▶ **Lemma 6.** *For any*[7] $\mathbb{N} \xrightarrow{g} \mathbb{N}$ *and* $c \in \mathbb{N}$*, no word-RAM recognized* $\mathbf{A}$ *in time* $g(k) + c \cdot n$*.*

Proof. We follow the proof of Theorem 3 in [6]. Suppose a word-RAM P' accepts $\mathbf{A}$ in time $g(k) + c \cdot n$. Then there exists some (w, k) (possibly with $|w|$ extremely large) so that $P_w = P'$ and $g(k) + c \cdot |w| < f(k) \cdot |w|$. Now:

$$\begin{aligned}(w,k) \in \mathbf{A} &\to P' \text{ accepts } w \text{ within time } g(k) + c \cdot |w| \\ &\to P_w \text{ accepts } w \text{ within time } g(k) + c \cdot |w| \\ &\to P_w \text{ accepts } w \text{ within time } f(k) \cdot |w| \\ &\to w \notin \mathbf{A}\end{aligned}$$

And:

$$\begin{aligned}(w,k) \notin \mathbf{A} &\to P' \text{ does not accept } w \text{ within time } g(k) + c \cdot |w| \\ &\to P_w \text{ does not accept } w \text{ within time } g(k) + c \cdot |w| \\ &\to P_w \text{ does not accept } w \text{ within time } f(k) \cdot |w| \\ &\to w \in \mathbf{A}\end{aligned}$$

◀

We note the existence of a form of universal RAM in [6] called a *Random Access Stored Program* (RASP) machine. A full discussion of its programming language RAM-ALGOL is beyond the scope of this paper. However, In Theorem 1 of [6] it is explained how the RASP can simulate another RAM with time overhead of only a constant factor. We may additionally note that the contents of the registers in the RASP need not exceed in length the contents of the registers of the RAM being simulated. It follows that the RASP functions as a universal word-RAM. We note that the RASP can count the $f(k) \cdot n$ steps of the computation in two registers since $f(k) \leq n$ (applying a fortiori the assumption that $f(k) < \log n$ as discussed above). We are now in a position to obtain the following (and this uses that f is computable, with time complexity h).

[6] In [6] the function needed to be time-constructible. Here we may mildly relax this assumption.
[7] We do not need to impose any restriction on g.

▶ **Lemma 7.** *There is a word-RAM that accepts* **A** *in time* $h(k) \cdot n$ *for some computable function* h.

▶ **Theorem 8.** $\mathbf{A} \subset \mathit{LFPT} \setminus \mathit{TLFPT}$.

Linear Time Kernelization Kernelization has seen extensive use within parameterized algorithmics, and is central to many of our results. Here we define linear time kernelization and then argue that it is fundamentally coupled with the class of TLFPT problems.

We say that a parameterized problem Π admits a *linear time kernelization* if there exists a function g and a self-reduction $\pi\colon (x,k) \mapsto (x',k')$, computable in time linear in $|x|+k$, such that: **1.** (x,k) is a yes-instance if and only if (x',k') is a yes-instance; and **2.** $|x'| \leq g(k)$ and $k' \leq k$. If there is a self-reduction respecting properties (1) and (2) but with runtime $O(|x| + f(k))$ for some superlinear f then we refer to "TLFPT time kernelization" or say that a kernel for Π is computable in time $O(|x| + f(k))$. There is some variability in the definition of kernelization. By default (e.g., in [10]), a kernelization's runtime is polynomial in $|x|+k$. In this sense, TLFPT time kernelizations are not strictly kernelizations themselves, though as we shall see in Corollary 10, any problem admitting a TLFPT time kernelization also admits a linear-time kernelization.

▶ **Theorem 9.** *A nontrivial parameterized problem* Π *admits a linear time kernelization if and only if it is in TLFPT.*

Proof. A kernelization of an instance (x,k) of parameterized problem Π, linear in $|x|+k$, runs in time $c|x|+ck$ for some constant c and yields a kernel of size at most some function $g(k)$. Overall, the time complexity to solve (x,k) is $c|x|+ck+g'(k)$ for some g': applying the kernelization takes time $c|x|+ck$ and solving the resulting kernel takes time $f(|x'|,k')$ for some f.

Conversely, we choose fixed yes- and no-instances to reduce to (they exist, since Π is nontrivial). There must exist an algorithm $\mathcal{A}$ with complexity $c|x| + f(k)$ which witnesses TLFPT membership. We distinguish between two cases.

- If $|x| > f(k)$ then the runtime of $\mathcal{A}((x,k))$ is $c|x| + f(k) \leq (c+1)|x|$ which is linear in $|x|$ (and in $|x|+k$), so we can simply solve the instance and then map it to a constant-size yes- or no-instance, as appropriate.
- Otherwise, $|x| \leq f(k)$ and we map (x,k) to itself, which satisfies the definition of a linear time kernelization (by letting $g = f$).

We note that this does not require that f be computable: we can check whether $|x| > f(k)$ by simply running $\mathcal{A}((x,k))$ for $(c+1)|x|$ steps. If it has terminated then we are in the first case (and return a constant-size yes- or no-instance), and otherwise we are in the second case (and return (x,k) directly). ◀

▶ **Corollary 10.** *For any parameterized problem* Π*, the following properties are equivalent:*

1. *a kernel for* Π *is computable in time* $O(|x|+k)$*, i.e.* Π *admits a linear-time kernelization,*
2. *a kernel for* Π *is computable in time* $O(|x| + f(k))$ *for some function* f*, i.e.* Π *admits a TLFPT time kernelization,*
3. Π *is decidable in time* $O(|x| + f(k))$*, i.e.* Π *is in TLFPT.*

Proof. The implications (1) $\implies$ (2) $\implies$ (3) are trivial, and the implication (3) $\implies$ (1) follows directly from Theorem 9. ◀

3 TLFPT via Degree-Based Kernelization: a Warmup

In this section we describe a general approach to proving membership in TLFPT. In particular, we show that if a problem is *degree reducible* and has *bounded matching number* then we can construct a kernel for the problem in TLFPT time.

We illustrate this method using Vertex Cover, which is both one of the classic NP-complete combinatorial problems [27], and also (with parameter k) one of the prototypical FPT problems [11]. A linear kernelization algorithm for this problem has been previously described [3] and is known to be linear [10]. Here we use the problem to demonstrate our more general method.

Vertex Cover
Input: graph $G = (V, E)$, integer k.
Question: does there exist a set $S \subseteq V$ with $|S| \leq k$ such that for every $(u, v) \in E$, $\{u, v\} \cap S \neq \emptyset$?

A common observation

is that any vertex with degree at least δ must be contained in any vertex cover of size at most $\delta - 1$. That is, if a graph G contains some vertex cover C of size at most k, then C necessarily contains every vertex of G with degree at least $k+1$. Thus, for any vertex v with degree at least $k+1$, (G, k) is a yes-instance of Vertex Cover if and only if the same holds for $(G - v, k - 1)$. Further, if G contains some isolated vertex v, then (G, k) is a yes-instance of Vertex Cover if and only if the same holds for $(G - v, k)$. We call a problem with these properties *degree reducible* as we formally define below.

▶ **Definition 11.** *Let Π be some parameterized graph problem. We say that Π is d-degree reducible for some function $d : \mathbb{N} \to \mathbb{N}$, if for any instance (G, k) of Π and any vertex v of G the following holds:*

- *if $|N(v)| > d(k)$, then $(G, k) \in YES(\Pi)$ if and only if $(G - v, k - 1) \in YES(\Pi)$; and*
- *if $|N(v)| = 0$, $(G, k) \in YES(\Pi)$ if and only if $(G - v, k) \in YES(\Pi)$.*

As suggested above, Vertex Cover is d-degree reducible where d is the identity function.

▶ **Lemma 12.** Vertex Cover *is d-degree reducible when $d(k) = k$ for each $k \in \mathbb{N}$.*

Proof. We show d-degree reducibility of Vertex Cover by showing each condition in Definition 11 holds.

Let (G, k) be an instance of Vertex Cover. Suppose G contains some vertex v with degree at least $k + 1$. We claim that G contains a vertex cover of size at most k, if and only if $G - v$ contains a vertex cover of size at most $k - 1$.

Suppose that there is some vertex cover $C \subseteq V$ of size at most k. We claim that $v \in C$. Otherwise, given every edge in G has at least one endpoint in C, $N(v) \subset C$. This is a contradiction as $|N(v)| > |C|$. It follows that $C \setminus \{v\}$ is a vertex cover of $G - v$ and so $(G - v, k - 1)$ must also be a yes-instance of Vertex Cover.

Now let $(G - v, k - 1)$ be a yes-instance of Vertex Cover, that is, there is some vertex cover $C \subseteq V \setminus \{v\}$ of size at most $k - 1$. It follows that $C \cup \{v\}$ is a vertex cover of G with size at most k.

We now consider isolated vertices. Let v be some vertex with degree 0 in G. It follows that v is not contained in any minimal vertex cover of G and so (G, k) is a yes-instance of Vertex Cover if and only if $(G - v, k)$ is. ◀

The following now holds by definition:

▶ **Observation 13.** *Let Π be some d-degree reducible problem and let G be a graph. For any set R of vertices with degree greater than $d(k)$ and any set I of vertices which are isolated in $G - R$, $(G, k) \in YES(\Pi)$ if and only if $(G - (R \cup I), k - |R|) \in YES(\Pi)$.*

The first phase of our algorithm will, in time $O(|E| + |V| + d(k))$, compute the set R of vertices of degree greater than $d(k)$ and the set I which are isolated in $G - R$. We then build the graph $G - (R \cup I)$. By definition, this graph has degree at most $d(k)$ and contains no isolated vertices.

We then make use of a second property, both of Vertex Cover, as well as many other problems regarding matching. If some graph G contains a matching of size m, then necessarily every vertex cover of G has size at least m. It follows that if G contains a matching of size $k + 1$ then (G, k) is a no-instance of Vertex Cover. More generally, we consider problems which become trivial if our graph contains a sufficiently large matching. We say that such problems have *bounded matching number*, as defined below.

▶ **Definition 14.** *Let Π be some parameterized graph problem. We say that Π has positive (respectively negative) m-bounded matching number for some function $m : \mathbb{N} \to \mathbb{N}$, if, given any instance (G, k) of Π, whenever G contains a matching of size greater than $m(k)$, then (G, k) is a positive (resp. negative) instance of Π. Note that this trivially holds if no instance of Π contains a matching of size greater than $m(k)$.*

Thus, Vertex Cover has negative m-bounded matching number:

▶ **Lemma 15.** Vertex Cover *has m-bounded matching number for $m(k) = k$.*

Proof. Let (G, k) be an instance of Vertex Cover. If G contains a matching of size at least $k + 1$, then any vertex cover of G has size at least $k + 1$. It follows that (G, k) is a no-instance and Vertex Cover has $m(k)$-bounded matching number, where m is the identity function. ◀

Our later proofs shall make use of the functions m and d being monotone increasing without loss of generality:

▶ **Lemma 16.** *If a problem Π has m-bounded matching number (respectively is d-degree reducible) then there exists a function m' (resp. d') which is monotone increasing such that Π has m'-bounded matching number (resp. is d'-degree reducible).*

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be any function. Define $f'(x) = \max(\{f(i) + x : 1 \leq i \leq x\})$. It is clear that f' is monotone increasing, and additionally that if a problem has f-bounded matching number (respectively is f-degree reducible) then it also has f'-bounded matching number (resp. is f'-degree reducible).

This follows from the fact that $f'(x) \geq f(x)$ for all x, and that the usage of the function f in the respective definitions is to give a bound beyond which distinction is irrelevant (i.e. graphs containing matchings larger than $m(k)$ are either all YES-instances or all NO-instances, vertices of degree greater than $d(k)$ can all be reduced away). ◀

We now leverage the properties of d-degree reducibility and m-bounded matching number to kernelize our problems.

▶ **Observation 17.** *Let G be a graph with maximum degree Δ and no isolated vertices and let M be some maximum matching, then G has size at most $2|V(M)|\Delta$.*

We highlight that for any graph G we can find some arbitrary maximal matching M, in time $O(|V|)$ using depth first search [7]. This and reduction of high degree vertices allows us to obtain a kernel of size $2k^2$ for Vertex cover in time $O(n) + f(k)$. We first summarize our algorithm for Vertex cover, before giving the more general algorithm in full detail alongside a formal analysis of runtime.

Algorithm VertexCover(G, k)

1. Find the set R of vertices each with degree greater than k in G.
2. Find the set I of vertices with degree 0 in $G - R$.
3. Construct the graph $\hat{G} = G - (R \cup I)$. (note that $\hat{G}$ may be disconnected)
4. Let $\hat{k} = k - |I|$.
5. Find some arbitrary maximal matching M of $\hat{G}$.
6. If $|M| > k$, then return $(M_{k+1}, \hat{k})$ where M_{k+1} is that graph consisting exactly of a matching of size $k + 1$.
7. Else, return $(\hat{G}, \hat{k})$.

Algorithm 1 Solving d-degree reducible and m-matching bounded graph problems.

Require: $G = ([n], E)$ is a graph on n vertices; integer k; computable functions m and d.

1: Compute `deg` the array of length n with `deg`$[u]$ the degree of vertex u.
2: Compute $d(k)$.
3: Let $G' = (V', E')$ be the graph induced by $V \setminus \{u : \texttt{deg}[u] > d(k)\}$ in G and let $k' = k - |\{u : \texttt{deg}[u] > d(k)\}|$.
4: Recompute `deg` for the graph G'.
5: Let $G'' = (V'', E'')$ be the graph induced by $V' \setminus \{u : \texttt{deg}[u] = 0\}$ in G and let $k'' = k'$.
6: By depth-first search on G'', and greedily build a maximal matching $M \subseteq E(G'')$.
7: **if** $|M| > m(k'')$ **then return** $(m(k'') + 1)P_2$ (i.e. the matching on $m(k'' + 1)$ edges)
8: **else return** G'', k''

▶ **Theorem 18.** *Given an instance (G, k) of a d-degree reducible problem Π which has m-bounded matching number, there is some algorithm that runs in $O(|V| + |E| + d(k))$ time and returns a kernel of Π on at most $2m(k) \cdot (d(k) + 1)$ vertices.*

Proof. In particular, Algorithm 1 has these properties. The algorithm begins (line 1) by initializing `deg` as a zero-array of length $|V|$ and then iterating of E and incrementing entries of `deg` accordingly for each edge. This requires time $O(|E| + |V|)$ (likewise for line 4 later on). The algorithm then computes $d(k)$.

In line 3 (resp. 5), the algorithm simply iterates over vertices and edges and uses the lookup table `deg` (resp. `deg`) to decide whether to include them in the newly constructed instance (G', k') (resp. (G'', k''). This runs in time $O(|V| + |E|)$ and can be implemented in-place, so that the memory used by G is overwritten to hold G' and then G''.

Finally, line 6 runs in $O(|V| + |E|)$ time as detailed in [7], (including if G' is disconnected). The algorithm then returns one of two possible instances. Therefore, the algorithm requires a total of $O(|V| + |E| + d(k))$ time.

Note that applying Definition 11 directly entails that (G'', k'') is a yes-instance iff (G, k) was a yes-instance.

Line 7 is correct by definition of m-matching bounded (Definition 14). This follows from the fact that if $|M| > m(k'')$, the graph returned has a matching of size greater than $m(k'')$. The returned graph is on $2 + 2m(k'')$ vertices and $1 + m(k'')$ edges.

Otherwise, $|M| \leq m(k')$ and line 8 returns (G'', k'') with the property that G'' is a graph on at most $2m(k) \cdot (d(k) + 1)$ vertices. G'' admits M as as a maximal matching. We denote M_V the set of vertices incident to an edge of M (so $|M_V| \leq 2m(k'')$). Note that since there are no vertices of degree 0 in G'' and G'' has maximum degree at most $d(k)$, we obtain that M_V is a dominating set of maximum degree $d(k)$ and thus G'' is a graph on at most

$2m(k'') \cdot (d(k)+1)$ vertices. Since $k'' = k' \leq k$ and m and d are monotone increasing, we obtain that $|V(G'')| \leq 2m(k) \cdot (d(k)+1)$. ◀

By Lemmas 12 and 15,

Vertex Cover is d-degree bounded and has m-bounded matching number, this allows us to apply Theorem 18.

▶ **Corollary 19.** Vertex Cover *is in TLFPT parameterized by* k.

4 TLFPT kernelization via Matchings and Twins: the Census Method

We now describe a second method for linear time kernelization.

To apply this, the problem must satisfy two properties. The first is *bounded matching number* (c.f. Definition 14) which implies that for any non-trivial instance we can compute a vertex cover of bounded size in linear time. We then construct a *census*, with respect to this vertex cover, which characterizes the remaining *cloud* of vertices according to their neighborhood, thus describing an equivalence relation over the vertices.

Two vertices are *open twins* if they are non-adjacent and have the same neighborhood. That is, a set of open twins must be an independent set. As each equivalence class corresponds to a set of open twins, the second property that we use, *twin reducibility*, allows us to bound the size of each equivalence class and yields a problem kernel.

To illustrate this method, we show that Dominating Set is in TLFPT when paramterized by vertex cover number. Note that Dominating Set is shown to be W[2]-hard with respect to the size the dominating set [9]; we instead parameterize by the vertex cover number. Dominating Set is defined as follows.

Dominating Set
Input: graph $G = (V, E)$, integer c.
Question: is there a set $S \subseteq V$ with $|S| \leq c$ such that for each $u \in V$, $N_G[u] \cap S \neq \emptyset$?

We use DomSetVC to refer to the problem Dominating Set parameterized by vertex cover number k of the input graph G. That is, given an integer c and a graph G with vertex cover number k, we ask if there is a dominating set S of G of size at most c.

Our kernelization algorithm for DomSetVC begins by, in linear time, removing any isolated vertices in the graph and decrementing c for each of these. Observe that, following this, any vertex cover of cardinality c in G is also a dominating set. This allows us to assume that $c < k$, otherwise we have a trivial yes-instance. This yields the following lemma.

▶ **Lemma 20.** DomSetVC *has negative* m*-bounded matching number where* $m(k) = k$.

Proof. Note that the set of yes-instances with maximal matching greater than k is empty. ◀

Following from above, we can now assume that we have some maximal matching M of size at most $m(k)$, that is $|V(M)| \leq 2m(k)$. Observe that, since this matching is maximal, every edge of G has at least one endpoint in this matching, hence those vertices not contained in M must form an independent set. We call these vertices the *independence cloud*, I. We now describe how those vertices of I can be represented using a *census vector*. In defining this, we assign every vertex in I a label according to its neighbors in M. This is a bit-string of length $2m(k)$ (a bit for each vertex in the maximal matching), such that the ith bit of this label is 1 if the vertex in question is adjacent to vertex i in M.

▶ **Definition 21.** *A subset S of vertices of some graph G is a set of open twins if for all x and y in S, $N(x) = N(y)$ in G.*

▶ **Observation 22.** *Any set S of open twins is an independent set.*

Notice that any two vertices with the same label in the census vector relative to some maximal matching are open twins.

We note that each label forms an equivalence class of open twins in the independence cloud, and there are at most $2^{m(k)}$ such labels. We create a *census vector* which, for each of the equivalence classes, counts how many of the vertices in the independence cloud has the corresponding label.

To ensure that the algorithm runs in TLFPT time overall, we truncate the count by a function of k. That is, we take the minimum of some $\tau(k)$ and the number of vertices with that label. We refer to problems where we can truncate the count in the census vector by some function τ, as τ-twin reducible.

The second property that we require for the census method ensures that we only have to consider a bounded number of vertices with the same neighborhood. For dominating set, we need only consider at most c vertices with the same neighborhood. This follows from the fact that no dominating set of size c can be a strict subset of a set of at least c open twins; either the remaining vertices are undominated, or all are dominated by a shared neighbor. With our preprocessing of isolated vertices, this implies there is always at least one twin in the set which is not in a dominating set of size c. Ensuring that each such vertex is dominated will also dominate all vertices not included in our bounded census.

▶ **Definition 23.** *Let Π be a some parameterized graph problem. We say that Π is τ-twin reducible for some function $\tau : \mathbb{N} \to \mathbb{N}$ if the sets $YES(\Pi_k)$ and $NO(\Pi_k)$ are closed under the deletion of any vertex with at least $\tau(k)$ open twins.*

▶ **Lemma 24.** *If a problem Π is τ-twin reducible then there exists a function τ' which is monotone increasing such that Π is τ'-twin reducible.*

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be any function. Define $f'(x) = \max\{f(i) + x : 1 \leq i \leq x\}$. It is clear that f' is monotone increasing, and additionally that if a problem is f-twin reducible then it is also f'-twin reducible (since f is pointwise at most f').

◀

▶ **Lemma 25.** DomSetVC *is τ-twin reducible for $\tau(k) = k + 1$.*

Proof. Suppose S is a set of open twins. We will show that, if $|S| > k$, then for any $v \in S$ we have that (G, c, k) is a yes-instance if and only if $(G - v, c, k)$ is.

We begin by assuming that D is a dominating set of G. If $S \cap D = \emptyset$, then D dominates $G - v$. Otherwise, consider some $v \in D \cap S$. Since $|S| > k \geq |D|$, there must be some vertex in $S \setminus D$; call it u. Then, since u and v are open twins, $(D \setminus \{v\}) \cup \{u\}$ dominates $G - v$.

Now consider a dominating set D' of $G - v$. Note that, since $|S| > k \geq |D'|$, there must be a vertex u in $S \setminus D$ which is dominated by D'. Since v shares a neighborhood with u, it must be dominated by the same vertex in D' in G. Therefore, D' is a dominating set of size k in G, and we have a yes-instance. ◀

As our problem has matching number at most $m(k)$, it contains a vertex cover with size at most $2m(k)$. It follows that the remaining vertices of the graph can be partitioned into $2^{2m(k)}$ equivalence classes. From twin reducibility, we need only consider $\tau(k)$ vertices from each class, thus yielding a kernel with size only a function of k. This gives us the following theorem – the main algorithmic result of this section.

▶ **Theorem 26** (The Census Method). *For any parameterised graph problem* Π*, if there exist functions* $\tau : \mathbb{N} \to \mathbb{N}$ *and* $m : \mathbb{N} \to \mathbb{N}$ *such that* Π *is* τ*-twin reducible and has* m*-bounded matching number, then a kernel for* Π *is computable in time* $O(|E(G)| + f(k))$.

Proof. Let Π_k be some parameterized graph problem which is both τ-twin reducible and has m-bounded matching number. For any instance of Π_k with corresponding graph $G = (V, E)$ we describe the following kernelization algorithm running in time $O(|E|) + f(k)$ for some fixed function f. Let $M \subseteq E$ be some arbitrary maximal matching of G (note that such a matching can be found in time $O(|V|)$ via depth first search [7]). As Π_k has m-bounded matching number, if $|M| > m(k)$, then we return the graph $K_{2m(k)+2}$.

Assume now that $|M| \leq m(k)$. We now describe the *matching census* representation of a graph with respect to M. Note that this will allow us to find and remove twin vertices in the necessary time bounds. Let $M_V = V(M)$. As M is a maximal matching, M_V is a vertex cover and G can be partitioned into the *matching* graph $G[M_V]$ (which admits M as a perfect matching) and the graph $G[V \setminus M_V]$ consisting of only isolated vertices. From this, we describe the following encoding of G. We first fix some arbitrary bijection $f : 2^{|M_V|} \to 2^{M_V}$. Let $\vec{c}$ be that vector of length $2^{|M_V|}$ such that, for every $i \in [2^{|M_V|}]$, there are $\vec{c}[i]$ vertices in $V \setminus M_V$ with neighborhood $f(i)$. We call the tuple $(G[M_V], \vec{c})$ the matching census representation of G with respect to M.

For any $t \geq 1$, we call the array $\vec{c'}$ such that $\vec{c'}[i] = min(t, \vec{c}[i])$ for every $i \in [2^{|M_V|}]$, the t-pruning of $\vec{c}$. Further, we call the graph corresponding to $(G[M_V], \vec{c'})$ the t-pruning of G, with respect to M. As Π_k is τ-twin reducible, the $\tau(k)$-pruning of G is a kernel for G.

We now claim that given such a graph G and maximal matching $M \subseteq G$ with size at most $m(k)$, we can return the matching census representation of the $\tau(k)$-pruning of G with respect to M, in time $O(|E|) + f(k)$ for some fixed function f. As $|M| \leq 2m(k)$, the $\tau(k)$-pruning of G with respect to M, has size at most $2m + \tau(k) \cdot 2^{2m(k)}$. Hence, the problem Π_k admits a linear time kernelization.

Algorithm 2 computes the census vector of the $\tau(k)$-pruning of G with respect to a some maximal matching M, where $s = 2|M|$. By Lemma 5, we can assume that $2^k + 1 \leq \log n$, it follows that the word operations in Algorithm 2 take constant time. We fix an arbitrary bijection $V \to [n]$ such that the vertices in $[s]$ correspond to those contained in M. Since $s \leq 2m(k)$, the arrays $\vec{c}$ and $\vec{f}$ can be constructed in time $f(k)$, for some fixed function f. Moreover, in lines 3 though 7, each edge of G is processed at most once. It follows that Algorithm 2 runs in time $O(|E|) + f(k)$.

Algorithm 2 Solving τ-twin reducible and m-matching bounded graph problems.

Require: $G = ([n], E)$ is a graph on n vertices; $[s]$ is a vertex cover of G; and $\tau(k)$ is some constant.

1: Allocate $\vec{c}$ to be an array of zeroes of length 2^s.
2: Let $\vec{f}$ be an array of length s such that $\vec{f}[i] = 2^i$ for each $0 \leq i < s$.
3: **for all** $v \in \{s, s+1, \ldots, n-1\}$ **do**
4: Let $p = 0$. ▷ *Stores target index.*
5: **for all** $u \in N(v)$ **do**
6: Let $p+ = \vec{f}[u]$.
7: Let $\vec{c}[p] = min(\tau(k), \vec{c}[p] + 1)$
8: **return** $\vec{c}$.

◀

Combining Lemmas 20 and 25 and Theorem 26 gives us the following result.

▶ **Theorem 27.** DomSetVC *admits a TLFPT time kernelization.*

In the following subsections we will apply Lemmas 20 and 25 and Theorem 26 to a variety of parameterized graph problems, thus showing they are also in TLFPT.

4.1 Minimum Maximal Matching

We continue to show that this method gives inclusion in TLFPT for some other problems. We begin with Minimum Maximal Matching, which is NP-hard [37].

Minimum Maximal Matching
Input: graph $G = (V, E)$, integer k.
Question: does there exist a maximal matching $M \subseteq E$ with size at most k?

▶ **Theorem 28.** Minimum Maximal Matching *admits a TLFPT time kernelization.*

Proof. We first claim that Minimum Maximal Matching parameterized by k is τ-twin reducible and has ℓ-bounded matching number for the functions $\tau(k) = k + 1$ and $\ell(k) = 2k$.

Let (G, k) be an instance of Minimum Maximal Matching. Suppose G has a maximal matching $M \subseteq E$ of size at most k. Let M' be some arbitrary matching in G. Since M is maximal, every edge $uv \in M'$ must have at least one endpoint (either u or v) that is incident to an edge in M. As each edge in M can be incident to at most two vertices, it follows that $|M'| \leq 2|M| \leq 2k$. Thus, if G contains a matching of size at least $2k + 1$, then G cannot have any maximal matching of size at most k and so Minimum Maximal Matching has ℓ-bounded matching number.

We now claim that Minimum Maximal Matching is τ-twin reducible. Suppose there exists a set S of open twins of size $k + 2$ and let v be one of its elements. We claim that G has a maximal matching of size at most k if, and only if, the same holds for $G - v$.

Let M be some maximal matching in G with size at most k. As $G[S]$ is an independent set (Observation 22) and S has size $k + 2$, there are at least two vertices in S that do not correspond to the endpoint of some edge in M. Without loss of generality, one of these is v. It follows that M is also a maximal matching in $G - v$.

Conversely, suppose M is a maximal matching in $G - v$ with size at most k. As $G[S]$ is an independent set (Observation 22) and S has size $k + 2$ there exists some vertex $v' \in S$ distinct from v which is not the endpoint of any edge in M. As M is a maximal matching, every vertex in $N(v')$ is the endpoint of some edge in M. Given that $N(v) = N(v')$, M is also a maximal matching in G. Thus, Minimum Maximal Matching is τ-twin reducible.

It follows that applying Theorem 26 we obtain a TLFPT time kernelization algorithm for Minimum Maximal Matching. ◀

4.2 Diverse Pair of Matchings

We turn to a different matching problem; namely, Diverse Pair of Matchings. For two sets X, Y we let $X \triangle Y$ denote the symmetric difference. That is $X \triangle Y = (X \setminus Y) \cup (Y \setminus X)$.

Diverse Pair of Matchings
Input: graph $G = (V, E)$, integer k.
Question: does there exist a pair of matchings $M, M' \subseteq E$ such that $|M \triangle M'| \geq k$?

We note that the ideas necessary to show that Diverse Pair of Matchings admits a TLFPT time kernelization were already presented in [16]. Moreover, the algorithm described in Theorem 26 can be adapted to produce the same $O(k^2)$ kernel in time $O(|V| + k)$. For completeness we repeat those ideas of [16] as well as giving that TLFPT time kernelization algorithm.

▶ **Theorem 29.** Diverse Pair of Matchings *admits a TLFPT time kernelization.*

Proof. We first show that Diverse Pair of Matchings is τ-twin reducible and has ℓ-bounded matching number, for the functions $\tau(k) = 2k$ and $\ell(k) = k - 1$. That is, by Theorem 26 we obtain a TLFPT time kernelization algorithm for Diverse Pair of Matchings.

Let (G, k) be an instance of Diverse Pair of Matchings. If a G contains some matching M of size k, then as $M \triangle \emptyset \geq k$, G is a yes-instance of Diverse Pair of Matchings. That is, Diverse Pair of Matchings has ℓ-bounded matching number.

We now show that Diverse Pair of Matchings is τ-twin reducible for the function $\tau(k) = 2k$. To show this we repeat the stronger claim which appears in [16]. Suppose, for some vertex v, there is a set $X \subseteq G$ such that, $v \notin X$, $G[X \cup \{v\}]$ is an independent set and, for every $u \in N(v)$, $|N(u) \cap X| \geq 2k$. Then (G, k) is a yes-instance of Diverse Pair of Matchings if, and only if, $(G - v, k)$ is a yes-instance of Diverse Pair of Matchings. It follows that, if there is some set S of size $2k$ such that $v \notin S$ and, for every $v' \in S$, the vertices v and v' are open twins, then (G, k) is a yes-instance of Diverse Pair of Matchings if, and only if, $(G - v, k)$ is a yes-instance of Diverse Pair of Matchings. That is, Diverse Pair of Matchings is τ-twin reducible for $\tau(k) = 2k$.

It follows, by Theorem 26, that Diverse Pair of Matchings admits a TLFPT time kernelization. We note that that kernel resulting from Theorem 26 has size $O(2k \cdot 2^{k-1})$, we now claim that we can adapt this algorithm to obtain that kernel of size $O(k^2)$.

▷ Claim 29. Diverse Pair of Matchings admits a kernel of size $O(k^2)$ which can be found in time $O(|E|) + f(k)$, for some fixed function f.

Proof. Our kernelization algorithm for Diverse Pair of Matchings will follows very similarly to that of Theorem 26 bar a few notable differences which we highlight below.

Let (G, k) be an instance of Diverse Pair of Matchings. Let $M \subseteq E$ be some arbitrary maximal matching of G. If $|M| \geq k$, then G is a yes-instance of Diverse Pair of Matchings, hence hence we return the graph K_{2k}.

Assume now that $|M| \leq k - 1$. We will again make use of the matching census representation of a graph and return the matching census representation of our kernel. Note that in the algorithm of Theorem 26 we greedily include a vertex $v \in V \setminus V(M)$ in the kernel if there are at most $2k - 1$ vertices $v' \in V \setminus V(M)$ such that v' is included in the kernel and v, v' are open twins. In this algorithm we will instead include a vertex $v \in V \setminus V(M)$ in the kernel if there exists some vertex $u \in N(v)$ such that there are at most $2k - 1$ vertices in $N(u) \cap (V \setminus V(M))$ which are included in the kernel. Algorithm 3 computes the census vector corresponding to those vertices of $V \setminus V(M)$ contained in this kernel.

Once again, we let $s = 2|M|$ and fix an arbitrary bijection $V \to [n]$ such that the vertices in $[s]$ correspond to those contained in M. Since $s \leq 2m(k)$, the arrays $\vec{c}$, $\vec{n}$ and $\vec{f}$ can be constructed in time $f(k)$, for some fixed function f. Moreover, in lines 4 though 14, each edge of G is processed at most twice, once in lines 7 through 10 and once in lines 12 through 14. It follows that Algorithm 3 runs in time $O(|E|) + f(k)$.

◁

◀

Algorithm 3 Diverse Pair Algorithm

Require: $G = ([n], E)$ is a graph on n vertices and vertex cover $M_v = [s]$.

```
Allocate c⃗ to be an array of zeroes of length 2^s.
Allocate n⃗ to be an array of zeroes of length s.
Let f⃗ be a vector of length s such that f⃗[i] = 2^i for each 0 ≤ i < s.
for all v ∈ {s, s+1, ..., n−1} do
    Let p = 0.                                  ▷ Stores target index.
    Let f = 0.                                  ▷ Flag for adding to kernel.
    for all u ∈ N(v) do
        Let p+ = f⃗[u].
        if n⃗[u] ≤ 2k − 1 then
            f = 1
    if f = 1 then
        for all u ∈ N(v) do
            n⃗[u]+ = 1
        Let c⃗[p]+ = c⃗[p] + 1)
return c⃗.
```

4.3 $(n-k)$-Coloring

We further show that there exist problems which themselves are not τ-twin reducible nor do they have m-bounded matching number but we can reduce them to a problem with these properties. To illustrate this we use the problem $(n-k)$-Coloring. We note that a *proper* coloring of a graph is one where the endpoints of any edge are a colored differently.

$(n-k)$-Coloring
Input: A graph $G = (V, E)$ with n vertices and integer k.
Question: Does there exist an $(n-k)$-proper coloring of G?

Although we are simply properly coloring the vertices in the graph, an intuitive way to think about this problem is as if the input graph is trivially colored with n colors (each vertex assigned a different color) and we have to *save* at least k colors by reusing them for multiple vertices; resulting in an $(n-k)$-proper coloring of the input graph.

For $(n-k)$-Coloring it will be useful to consider the complement of our input graph. Let G be a graph and let $\overline{G}$ be its complement. If M is some matching in $\overline{G}$, then for every edge $uv \in M$, it is possible to assign u and v the same color in a proper coloring of G. That is, if $|M| \geq k$ then (G, k) is a yes-instance for $(n-k)$-Coloring. Since we are working in the complement of the original graph, we consider the following auxiliary problem $(n-k)$-Coloring Complement.

$(n-k)$-Coloring Complement
Input: A graph $G = (V, E)$ with n vertices and integer k.
Question: Does there exist an $(n-k)$-coloring of the complement of G?

We now show that $(n-k)$-Coloring Complement is τ-twin reducible and has positive m-bounded matching number, for the functions $m(k) = k-1$ and $\tau(k) = k$. That is, $(n-k)$-Coloring Complement is in TLFPT.

We now give some intuition regarding the τ-twin reducibility of the problem. Note that if there is some vertex which is unique in its color class, then the removal of this vertex decreases both the number of vertices of the graph and the number of colors needed; preserving yes- and no-instances. As any set of open twins in G form a clique in $\overline{G}$ (Observation 22), each of these vertices must be colored differently. It follows that either at least one of these vertices is unique in its color class, or the graph can be colored with $n-k-1$ colors. That is, it is safe to remove a vertex from this set.

▶ **Lemma 30.** *The problem $(n-k)$-Coloring Complement is τ-twin reducible and has positive m-bounded matching number, where $\tau(k)=k$ and $m(k)=k-1$.*

Proof. We first show that $(n-k)$-Coloring Complement has m-bounded matching number. Let G be some graph, $\overline{G}$ be its complement and let $n=|V|$. Now $(\overline{G},k)$ is an instance of $(n-k)$-Coloring Complement. Recall that $(\overline{G},k)$ is a yes-instance if, and only if, G can be colored with $n-k$ colors.

Suppose there is a matching M of size k in $\overline{G}$. We construct a $(n-k)$-proper coloring of G as follows. Begin by coloring each vertex differently. This is a trivial proper coloring. For each edge $uv \in M$ we now recolor the vertex v giving it the same color as u. As uv is an edge in $\overline{G}$ it is a non-edge in G, that is, this is a proper coloring of G. As for each edge in M this reduces the number of colors used by 1 and $|M|=k$, this describes an $n-k$ coloring. Thus, $(\overline{G},k)$ is a yes-instance of $(n-k)$-Coloring Complement and $(n-k)$-Coloring Complement has positive m-bounded matching number for $m(k)=k-1$.

We continue by showing the problem is τ-twin reducible where $\tau(k)=k$. Suppose S is a set of open twins of size $k+1$ and let v be one of its elements. We claim that $(\overline{G},k)$ is a yes-instance of $(n-k)$-Coloring Complement if, and only if, $(\overline{G}-v,k)$ is a yes-instance of $(n-k)$-Coloring Complement. As $\overline{G}-v$ has size $n-1$, this is equivalent to showing that there is an $(n-k)$-coloring of G if and only if there is an $(n-k-1)$-coloring of $G-v$.

We first show the backwards direction, that is, if there is an $(n-k)$-coloring of G then there is an $(n-k-1)$-coloring of $G-v$. Let $c\colon V(G)\to[n-k]$ be an $(n-k)$-coloring of G. Recall that, by definition, every pair of vertices in $S\cup\{v\}$ are open twins in $\overline{G}$. That is, $S\cup\{v\}$ forms a clique in G and each vertex in $S\cup\{v\}$ is assigned a different color by c. We now consider the following three cases.

1. Suppose v is unique in its color class: it follows that c restricted to the graph, $G-v$ uses at most $n-k-1$ colors.
2. Assume that v is contained in some color class of size at least 2. Suppose now that there is some vertex $u\in S$ such that u is unique in its color class. As u and v are twins in $\overline{G}$, they are also twins in G. It follows that swapping the colors assigned to u and v we obtain an $n-k-1$ coloring of $G-v$.
3. Assume that every vertex in $S\cup\{v\}$ is contained in some color class of size at least 2. By definition $|S\cup\{v\}|=k+1$. It follows that, if every vertex in $S\cup\{v\}$ is contained in some color class of size at least 2, then c uses at most $n-k-1$ colors. That is, c is an $n-k-1$ coloring of $G-v$. This concludes the backward direction of our proof.

We now consider the forward direction. Suppose that there exists an $(n-k-1)$-coloring c' of $G-v$. We extend c' to a coloring of G by assigning v a unique color. By definition, this is a proper coloring of G that uses at most $n-k$ colors. This concludes the proof of our statement. That is, $(n-k)$-Coloring Complement is τ-twin reducible and has positive m-bounded matching number, where $m(k)=k-1$ and $\tau(k)=k$. ◀

By Theorem 26 it follows that $(n-k)$-Coloring Complement is in TLFPT with respect to k. To show now that the same holds for $(n-k)$-Coloring, we note the trivial reduction from $(n-k)$-Coloring to $(n-k)$-Coloring Complement: if $G = (V, E)$ is a graph and $\overline{G}$ its complement, by definition $(\overline{G}, k)$ is a yes-instance of $(n-k)$-Coloring Complement if, and only if, (G, k) is a yes-instance of $(n-k)$-Coloring. It now remains to show that this is a TLFPT-reduction.

We first show that either we have a trivial instance of $(n-k)$-Coloring or the size of our instance of $(n-k)$-Coloring Complement obtained from our reduction is bounded by the size of our instance of $(n-k)$-Coloring. That is, either we have a trivial instance of $(n-k)$-Coloring or $|\overline{E}| < |E|$. In particular, we show that if $|E| < |\overline{E}|$, i.e. $|E| < \frac{n(n+1)}{2}$, then (G, k) is a yes-instance of $(n-k)$-Coloring. Note that we can count the number of edges in G in time $O(|E|)$ and return a trivial yes instance of $(n-k)$-Coloring Complement if $|E| < \frac{n(n+1)}{4}$. We then assume that, since $\overline{G}$ is sufficiently sparse, $|\overline{E}| < |E|$ and we can build $\overline{G}$ in time $O(|V| + |E|)$. This gives us a TLFPT-reduction from $(n-k)$-Coloring to $(n-k)$-Coloring Complement.

We now argue that if G is sufficiently sparse then $\overline{G}$ contains a matching of size k and so (G, k) is a yes-instance of $(n-k)$-Coloring. To make this argument, we make use of Mader's theorem [31, 32]. Here we state an equivalent theorem; the original statement relates the constant c_H with the average degree of the graph, rather than the number of edges in the graph.

▶ **Theorem 31** (Mader [31, 32]). *For every graph H, there exists a constant $c_H \in \mathbb{N}$ such that any n vertex graph G with at least $c_H \cdot n$ edges contains H as a minor.*

We use this to show that, if the input graph is sufficiently sparse, it must have a dense complement that contains a matching of at least k edges. Let c be that constant found by Mader [31], such that every graph with n vertices and at least cn edges contains the complete graph on $2k$ vertices as a minor and so also a matching of size k.

Note that by Lemma 5, we can assume any function of k is bounded above by n. We use this to make the assumption that $k^2 \leq \frac{n+1}{4}$.

▶ **Lemma 32.** *Let $G = (V, E)$ be a graph and let $\overline{G} = (V, \overline{E})$ be the complement of G. For any k, such that $k^2 \leq \frac{n+1}{4}$, either $\overline{G}$ contains a matching of size at least k or $|\overline{E}| \leq |E|$.*

Proof. Mader [31] shows that $c_{K_r} \leq 8(r-2)\lfloor \log_2(r-2) \rfloor$, where K_r is the complete graph on r vertices. For this problem we note that if this threshold is met for $r = 2k$, then the graph contains a matching of size k. For ease of writing, let $c = c_{K_r}$ from this point onward.

Let $G = (V, E)$ be a graph and let $\overline{G} = (V, \overline{E})$ be the complement of G. Let $n = |V|$, $m = |E|$ and $\overline{m} = |\overline{E}|$. We now claim that either $\overline{m} \leq m$ or $\overline{m} \geq n \cdot c$. That is, $\overline{G}$ contains a matching of size at least k, or it is sparse enough to be computed in linear time. For our purposes, the coarse bound

$$n \cdot c \leq 8n(2k-2)\lfloor \log_2(2k-2) \rfloor \leq nk^2$$

is sufficient. We now show that if $\overline{m} > m$ then $\overline{m} \geq n \cdot c$. Recall our assumption that $k^2 \leq \frac{n+1}{4}$. Combining this with the fact that $m + \overline{m} = n(n+1)/2$, if $\overline{m} > m$ then we find that:

$$\overline{m} > \frac{n(n+1)}{4} \geq n \cdot k^2 \geq n \cdot c.$$

Thus concluding the proof of this lemma. ◀

Algorithm 4 Complement builder

Require: A graph $G = ([n], E')$.
1: Initiate $\overline{G} = ([n], E' = \emptyset)$.
2: **for all** $v \in [n]$ **do**
3: Initiate $N_{\overline{G}}(v) = [n]$.
4: **for all** $w \in N_G[v]$ **do**
5: Remove w from $N_{\overline{G}}(v)$.
6: **for all** $v \in [n]$ **do**
7: **for all** $w > v$ in $N_{\overline{G}}(v)$ **do**
8: Add (v, w) to E'.
9: **return** $\overline{G}$.

We now consider the case where $|\overline{E}| \geq |E|$. We can now apply the following lemma to obtain the complement of G, $\overline{G}$ in time $O(|V| + |E|)$.

▶ **Lemma 33.** *Given a graph $G = (V, E)$, we can build the complement of G, $\overline{G} = (\overline{V}, \overline{E})$, in time $O(|V| + |E| + |\overline{E}|)$.*

Proof. In particular we show that Algorithm 4 builds $\overline{G}$ in the required time.

The algorithm begins by initiating the graph $\overline{G}$ with the same vertex set as G and no edges. This takes at most $O(n)$ time. Following this, Algorithm 4 then generates the neighborhood of each vertex in $\overline{G}$. It does this by, for each vertex v, initiating its neighborhood as the entire vertex set, then iterating over all neighbors of v in G and removing them. Initiating the neighborhood requires $O(|V|^2) = O(|E| + |\overline{E}|)$ time. The inner loop requires $O(|E|)$ time since there are two iterations of this loop for each edge. The algorithm then uses the neighborhoods generated to find the edge set of $\overline{G}$. This requires $O(|\overline{E}|)$ time since each edge in $\mathcal{G}$ is added once to $\overline{E}$. It is clear from construction that the graph returned by Algorithm 4 is the complement of the input graph. This requires $O(|V| + |E| + |\overline{E}|)$ time in total. ◀

We now have those components necessary to prove the following theorem.

▶ **Theorem 34.** $(n-k)$-Coloring *is in TLFPT with respect to k.*

Proof. Combining Theorem 26 and Lemma 30, $(n-k)$-Coloring Complement is in TLFPT. We now prove that $(n-k)$-Coloring is in TLFPT via a TLFPT-reduction from $(n-k)$-Coloring to $(n-k)$-Coloring Complement. Let (G, k), be an instance of $(n-k)$-Coloring Complement, such that $G = (V, E)$. Recall that, by Lemma 5, we can assume that $k^2 \leq \frac{n+1}{4}$.

We can count the number of edges in G in time $O(|E|)$. If $|E| < n(n+1)/4$, then combining Lemmas 32 and 30 (G, k) is a yes-instance of $(n-k)$-Coloring and our reduction will return some trivial yes-instance of $(n-k)$-Coloring Complement.

Now suppose that $|E| \geq n(n+1)/4$. This gives us that $|\overline{E}| \leq |E|$. Using Lemma 33 we construct $\overline{G}$, the complement of G, in time $(|V| + |E|)$. By definition, (G, k) is a yes-instance of $(n-k)$-Coloring if and only if $(\overline{G}, k)$ is a yes-instance of $(n-k)$-Coloring Complement. That is, there is a TLFPT-reduction from $(n-k)$-Coloring to $(n-k)$-Coloring Complement and so $(n-k)$-Coloring is in TLFPT with respect to k. ◀

5 TLFPT via Depth and Pruning Kernelization

In Section 4 we placed various problems in TLFPT via the Census Method (Theorem 26). Whereas the Census Method relies on problems having bounded matching number, here

we will consider a larger class (c.f. Corollary 36) of problems: those with *bounded depth*, as defined below.

▶ **Definition 35.** *Let Π be some parameterized graph problem. We say that Π has positive (respectively negative) ℓ-bounded depth for some function $\ell : \mathbb{N} \to \mathbb{N}$, if, given any instance (G, k) of Π, whenever G contains a path of length greater than $\ell(k)$, then (G, k) is a positive (resp. negative) instance of Π. Note that this trivially holds if no instance of Π contains a path of length greater than $\ell(k)$.*

▶ **Corollary 36.** *If some graph problem Π has positive (respectively negative) m-bounded matching number, for some function $m : \mathbb{N} \to \mathbb{N}$. Then Π has positive (resp. negative) ℓ-bounded depth, for the function $\ell(k) = 2m(k) - 1$.*

Any non-trivial instance of a problem with bounded depth must have a short DFS tree. It then follows that such an instance has bounded treedepth [35].

▶ **Definition 37** ([35])**.** *The treedepth of a graph G is the minimum height of a rooted forest F such that $G \subseteq clos(F)$ where the closure of F $clos(F)$ has the vertex set $V(F)$ and the edge set $\{\{x, y\} : x \text{ is an ancestor of } y, x \neq y\}$.*

▶ **Lemma 38** ([35])**.** *For a graph G with a longest path of length ℓ, $\log(\ell) \leq td(G) \leq \ell$.*

This allows us to show that 3-Coloring parameterized by treedepth is both twin reducible and has bounded depth.

▶ **Theorem 39.** 3-Coloring *is in TLFPT parameterized by treedepth.*

Proof. Note that from Lemma 38, the set of instances containing some path of length greater than 2^k is empty. As these instances are trivially negative. 3-Coloring parameterized by treedepth has negative ℓ-bounded depth, where $\ell(k) = 2^k$.

We now claim that τ-twin reducible where $\tau(k) = 1$. Consider a graph G and two vertices $u, v \in V(G)$ such that u and v are open twins in G. We claim that G is 3-colorable if, and only if, $G - v$ is 3-colorable. As any 3-coloring of G also describes a 3-coloring of $G - v$, the backwards direction holds. Suppose now we have some 3-coloring of $G - v$ as u and v are open twins, there is some 3-coloring of G where u and v are assigned the same color. This proves our claim.

As 3-Coloring parameterized by treedepth has ℓ-bounded depth and is τ-twin reducible, where $\ell(k) = 2^k$ and $\tau(k) = 1$. It follows from Theorem 26 that 3-Coloring parameterized by treedepth admits a TLFPT time kernelization. ◀

5.1 Combining Bounded Depth with Twin Subgraphs

We now extend our methods based on bounded depth and twins by generalizing the notion of twin reducibility to *twin subgraphs* (Definition 40). We use this notion to generalize the equivalence relation defined for open twin vertices to larger isomorphic subgraphs, yielding the notion of *twin subgraphs reducibility* (Definition 41).

▶ **Definition 40.** *We say that a pair subgraphs H and H' of a graph G are twins if:*

1. *there is an isomorphism $i \colon H \to H'$ such that for any $v \in H$, the vertices v and $i(v)$ have the same neighborhood outside of $H \cup H'$; that is $N(v) \setminus V(H \cup H') = N(i(v)) \setminus V(H \cup H')$,*
2. *there is no edge $uv \in E(G)$ such that $u \in V(H)$ and $v \in V(H')$.*

We call i a twin isomorphism.

▶ **Definition 41.** *Let Π be a some parameterized graph problem. We say that Π is τ-twin subgraph reducible for some function $\tau : \mathbb{N}^2 \to \mathbb{N}$ if the sets $YES(\Pi_k)$ and $NO(\Pi_k)$ are closed under the deletion of any subgraph H of size n such that H has at least $\tau(k, n)$ twin subgraphs.*

As for Section 4, the following result yields a general "Depth and Prune" method for obtaining TLFPT results by kernelization. Intuitively, we find a bounded height depth-first search tree. Careful use of data-structures allows us to work from the leaves upwards finding and removing subtrees which are twin subgraphs. Due to its bounded height alongside properties of depth first search trees, the resulting graph has bounded size.

▶ **Theorem 42** (Depth & Prune Method)**.** *Let Π_k be some parameterized problem such that for functions $\ell : \mathbb{N} \to \mathbb{N}$ and $\tau : \mathbb{N}^2 \to \mathbb{N}$, Π_k is τ-twin subgraph reducible and has ℓ-bounded depth. Π_k admits a linear time kernelization.*

Proof. Let Π_k be some parameterized graph problem which is both τ-twin subgraph reducible and has ℓ-bounded depth, for functions $\ell : \mathbb{N} \to \mathbb{N}$ and $\tau : \mathbb{N}^2 \to \mathbb{N}$. For any instance $G = (V, E)$ of Π_k

we describe the following $O(|E|) + f(k)$ kernelization algorithm, for some fixed function f.

Let T be a depth-first-search tree of G and denote its root by $root(T)$.

We let $depth(T) = max_{v \in V} dist_T(root(T), v)$. If $depth(T) \geq \ell(k) + 1$ then G contains a path of length $\ell(k) + 1$ and we return the graph $P_{\ell(k)+1}$. As Π_k has ℓ-bounded depth, this is a kernel for G.

We now assume that $depth(T) \leq \ell(k)$. Let $d = depth(T)$. Our kernelization algorithm will then proceed by removing subgraphs

of size r having at least $\tau(k, r)$ twin subgraphs. As Π_k is size τ- subgraph reducible, the resulting instance is a yes-instance of Π_k if and only if G is. To allow this to be done in the necessary time bounds, we will only remove those twin subgraphs admitting an isomorphism that preserves the parent child relations of T. Towards this, we define some notation regarding trees. For every $v \in V$, let $parent_T(v)$ denote the parent of v. Let $dec_T(v)$ be the set of descendants of v and let $anc_T(v)$ be the set of ancestors of v. We write $dec_T[v] := dec_T(v) \cup \{v\}$.

We will now define the *type* of a vertex. This will be an equivalence relation on pairs of vertices at the same depth in the tree T. Our definition will rest upon an auxiliary notion of *ancestor consistency*, which we define below.

Let v be some vertex, the *ancestor vector* of v, $\vec{a}$, is that binary vector of length $depth_T(v) - 1$ such that the ith element of $\vec{a}$ is 1 if, and only if, v is adjacent to that vertex with depth i in $anc_T(v)$. We say that a pair of vertices (v, v') are ancestor consistent if they have the same ancestor vector. Note that any two vertices with the same ancestor vector necessarily have the same depth.

We are now define types recursively: two vertices v and v', both at some given depth r, are of the same type if: they are ancestor consistent and have the same number of children of each type.

We illustrate the concept of types with a small example in Figure 2.

▷ Claim 42. Let u and v be vertices with the same type. If $parent_T(u) = parent_T(v)$, then $G[dec_T[u]]$ and $G[dec_T[v]]$ are twin subgraphs.

Proof. As u and v have the same type, they have the same depth in T. Let $r = depth_T(u) = depth_T(v)$. By definition, every pair of vertices with the same type have the same number of

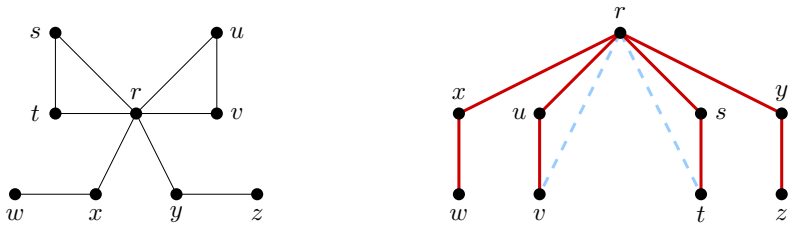

Figure 2 An 9-vertex graph H (left), and a DFS tree T of H (right). Edges of H which are not edges of T are shown in dashed light blue. In the DFS tree shown:

- v and t are of the same type, because they have the same ancestor vector, i.e. $\vec{a}(v) = \vec{a}(t) = (1)$ (both vertices are adjacent to r), and neither has any children. Call their type α.
- Analogously, w and z are of the same type, because $\vec{a}(w) = \vec{a}(z) = (0)$. Call their type β.
- Each of x, u, s, y has the same (empty) ancestor vector: however, seeing as x and y each have a single child of type β whereas u and s each have a single child of type α. Consequently, x and y are of the same type, which is different from the shared type of u and s.
- The root r has its own type. As we shall see, the type of r "encodes" the entire structure of the graph.

children of each type. It follows that, as u and v have the same type, there is a bijection $i : dec_T[u] \to dec_T[v]$ that preserves both vertex types and parent–child relations. That is, for every vertex $c \in dec_T[u]$, c and $i(c)$ have the same type and $parent_T(c) = i^{-1}(parent_T(i(c))$.

We now claim that i is a twin isomorphism. That is, for every edge $(c, c') \in E(G)$ such that $c \in dec_T[u]$, either: **1.** $c' \in dec_T[u]$ and $(i(c), i(c')) \in E(G)$, or **2.** $c' \in V \setminus (dec_T[u] \cup dec_T[v])$ and $(i(c), c') \in E(G)$. The case where $c \in dec_T[u]$ will follow symmetrically.

As T is a depth first search tree, c' is that unique vertex in $anc_T(c)$ with depth $depth_T(c')$. As $anc_T(c) \cap dec_T[c] = \emptyset$, $c' \notin dec_T[v]$. We first consider case (i), that is, $c' \in dec_T[u]$. As $parent_T(w) = parent_T(i(w))$, for every $w \in dec_T[u]$, $i(c')$ is the unique vertex in $anc_T(i(c))$ with depth $depth_T(c')$. By definition of i, c and $i(c)$ have the same type and so are also ancestor consistent. It follows that the edge $i(c)i(c') \in E(G)$.

We now consider consider case (ii), that is, $c' \in V \setminus (dec_T[u] \cup dec_T[v])$. Given that $parent_T(u) = parent_T(v)$, it follows that c' is that unique vertex in $anc_T(i(c))$ with depth $depth_T(c')$. As c and $i(c)$ are ancestor consistent, $i(c)i(c') \in E(G)$. ◁

We can partition the vertices of G into *equivalence classes* such that any pair of vertices belong to the same equivalence class if, and only if, they have the same type. For every equivalence class C, by definition there exist constants r and s such that every vertex v in C has depth r and $|dec_T[v]| = s$. We say that the class C, and its corresponding type, has depth r, subtree size s and capacity $\tau(k, s)$.

We say G is τ-pruned with respect to T if, for every vertex v and equivalence class C, $children_T(v) \cap C \leq \tau(k, s)$ where s is the subtree size of C.

▷ Claim 42. There exist functions $eq : \mathbb{N}^3 \to \mathbb{N}$ and $maxSize : \mathbb{N}^2 \to \mathbb{N}$, such that if G is τ-pruned with respect to T, then: (1) for every $r \in \{1, \ldots, d\}$ there are at most $eq(k, d, r)$ equivalence classes with depth r and (2) G has size at most $maxSize(k, d)$.

Proof. We define the function eq recursively. In the process of this, we also determine the subtree size associated with each type, which will later allow us to define $maxSize$.

Consider first a vertex v of depth d. By definition, v is a leaf, and so its type is determined only by its ancestor vector. Since there are 2^{d-2} possible ancestor vectors for a vertex of

depth d, it follows that $eq(k,d,d) = 2^{d-2}$. Moreover, every type at depth d has subtree size 1.

We now define $eq(k,d,r)$ for $r \in \{1,\dots,d-1\}$. We first fix some total ordering of the $eq(k,d,r+1)$ types of vertices at depth $r+1$, and let m_i and s_i denote the capacity and subtree size, respectively, of the ith type of vertex with depth $r+1$. Since G is τ-pruned, any vertex of depth r has at most m_i children of type i. That is, for each type i, there are m_i+1 possible choices for the number of children of type i, and hence the total number of distinct such choices is $\prod_{i=1}^{eq(k,d,r+1)}(m_i+1)$.

Further, there are $2^{r-1}-1$ possible ancestor vectors for a vertex of depth r. Combining these observations, $eq(k,d,r) = (2^{r-1}-1)\cdot\prod_{i=1}^{eq(k,d,r+1)}(m_i+1)$.

Let v be a vertex of depth r with exactly c_i children of type i for each $i \in \{1,\dots,eq(k,d,r+1)\}$. Recall that the ith type of vertex with depth $r+1$ has subtree size s_i, that is, for every vertex w with this type $dec_T[w] = s_i$. That is, the subtree size of the equivalence class containing v is $1+\sum_{i=1}^{eq(k,d,r+1)} c_i \cdot s_i$.

Finally, we define $maxSize(k,d)$. We again fix a total ordering of the $eq(k,d,2)$ types of vertices at depth 2, and let m_i and s_i denote the capacity and subtree size, respectively, of the ith type. Then G has size at most $maxSize(k,d) = \sum_{i=1}^{eq(k,d,2)} m_i \cdot s_i$. ◁

We define the *τ-pruning* of G as follows. Beginning with the leaves, we process vertices in order of decreasing depth. For a vertex v, if there exist at least $\tau(k,|dec_T[v]|)$ vertices of the same type as v and share the same parent in T, then remove the entire subtree $dec_T[v]$ from the graph.

As Π_k is τ-twin subgraph reducible, the τ-pruning of G is a kernel for G. It remains to show that the τ-pruning of G can be computed in time $O(|E(G)|)+f(k)$, for some function f depending only on k.

Our algorithm will, for each vertex in G, return a ID corresponding to its type in the τ-pruning of G (supposing it has not been removed). That is, there is a bijection between the set of IDs assigned to vertices of G and the equivalence classes of the τ-pruning of G. The ID of v consists of two parts. The first part, which we call the *ancestor ID*, encodes the ancestor vector of v and the second, which we call the *subtree ID*, encodes the number of children v has of each type. To build intuition regarding the subtree ID, we begin by defining the *subtree vector* of v.

For each $r \in \{1,\dots,d\}$, we fix a total order on the $eq(k,d,r)$ equivalence classes with depth r, and refer to the ith class under this order as the ith equivalence class with depth r.

The subtree vector of v, $\vec{s}$, is a vector of length $eq(k,d,depth_T(v)-1)$ with each element of $\vec{s}$ corresponding to the number of children that v has in each equivalence class. For $i \in \{1,\dots,eq(k,d,depth_T(v)-1)\}$, let C_i be the ith equivalence class with depth $depth_T(v)-1$ and let m_i be the capacity of C_i. Let

$$\vec{s}[i] := min(|children_T(v) \cap C_i|, m_i).$$

For each $v \in V$, let $\vec{a}_v$ denote the ancestor vector of v and let $\vec{s}_v$ denote the subtree vector of v. Since $\sum_{v\in V} |\vec{a}_v| + |\vec{s}_v| = f(k)\cdot|V|$, for some function f, it is not possible to naively allocate the memory for these vectors in TLFPT time. Instead, as both the length of these vectors and the value of their elements is bounded by some function of k, Lemma 5 will allow us to assume that k is small enough that each vector can be encoded within a single machine word of size $\log n$. As we assign two words for each vertex in G this requires only time $O(|V|)$.

We now explicitly define the encoding of these vectors and give a bound on k which will be sufficient for our purposes. The ancestor ID, $ancID(v) := \sum_{i\in\{1,\dots,|\vec{a}_v|\}} \vec{a}_v[i]\cdot 2^i$. As

$\vec{a}_v$ is a binary vector whose final element is 1, the ancestor vector is uniquely determined by $ancID(v)$ and $ancID(v) \leq 2^{depth_T(v)-1} + 1$. Likewise, let m_{max} denote the maximum capacity of any equivalence class of depth $depth_T(v) + 1$, we define the subtree ID of v by $subtreeID(v) := \sum_{i \in \{1,\ldots,|\vec{s}|\}} \vec{s}[i] \cdot (m_{max} + 1)^i$. This construction results in a bijection between the possible subtree vectors and the subtree IDs that a vertex with depth r could take. Further, $subtreeID(v) \leq (m_{max} + 1)^{eq(k,d,depth_T(v)+1)}$. We then let the ID of v be $ID(v) = ancVec(v) + subtreeID(v) \cdot 2^{|\vec{a}|}$. If m_{root}, is the maximum capacity of any equivalence class of depth 2, then for every vertex v, $ID(v) \leq (m_{root} + 1)^{eq(k,d,2)}$. That is, applying Lemma 5 we assume that $(m_{root} + 1)^{eq(k,d,2)} \leq \log n$ and so operations on these IDs can be done in constant time.

Recall that the ID of $root(T)$ encodes the τ-pruning of G. We can now begin to compute the ID of each vertex of G. To this end, we introduce the following auxiliary functions, which we precompute and store in arrays so as to allow constant-time access thereafter.

Let $subtreeIDtoVec : \mathbb{N}^4 \to \mathbb{N}^{\mathbb{N}}$ be the function such that a vertex with depth r and subtree ID i has subtree vector $subtreeIDtoVec(k, d, r, i)$. Let $IDtoType : \mathbb{N}^3 \to \mathbb{N}$ be the function such that a vertex with ID i belongs to the $IDtoType(k, d, i)$th equivalence class (with the same depth). Let $typetoCapacity : \mathbb{N}^4 \to \mathbb{N}$ be that function such that the ith equivalence class with depth r has capacity $typetoCapacity(k, d, r, i)$.

Since these functions will always be taken with the same first two arguments, k, d, to simplify notation we suppress these parameters in the following. We compute the value $eq(k, d, r)$, for every $r \in \{1, \ldots, d\}$ and store this in an array eq such that $eq[r] = eq(k, d, r)$. Each of these values can thereafter be accessed in constant time. Likewise, we compute the vector $subtreeIDtoVec(k, d, r, i)$, for each possible subtree ID i which could appear at depth r, and store this in a (two dimensional) array $subtreeIDtoVec$ such that $subtreeIDtoVec[r, i] = subtreeIDtoVec(k, d, r, i)$. We compute the value $IDtoType(k, d, i)$, for each possible ID i and store this in an array $IDtoType$ such that $IDtoType[i] = IDtoType(k, d, i)$. We compute the value $typetoCapacity(k, d, r, i)$, for every $i \in \{1, \ldots, eq(k, d, r)\}$ and store this in a (two dimensional) array $typetoCapacityt$ such that $typetoCapacity[r, i] = typetoCapacity(k, d, r, i)$. Finally, we compute the array $maxCapacity$ such that

$$maxCapacity[r] = max_{i \in \{1,\ldots,eq(k,d,r)\}} typetoCapacity[r, i].$$

By definition $d \leq \ell(k)$, that is these arrays can be constructed in time $f(k)$, for some fixed function f.

We note that for every $v \in V$, Algorithm 5 returns $ID(v)$. Further, Algorithm 5 is called on each vertex of V at most once (assuming memoization). A single iteration of $id(v)$ requires $O(|N(v)|)$ time, that is $id(root(T))$ can be computed in $O(|E|)$ time. It follows that in total our kernelization algorithm takes $O(|E|) + f(k)$, for some function f. ◀

We now turn to three classical problems: k-Path, k-Vertex Ranking, and Satisfiability.

k-Path
Input: graph $G = (V, E)$,
Parameter: k,
Question: does G contain P_k as a subgraph?

▶ **Theorem 43.** *k-Path admits a TLFPT time kernelization.*

Algorithm 5 id(v)

Require: Graph G; DFS tree T of G with depth d; some vertex v with depth r in T; arrays eq, $subtreeIDtoVec$, $IDtoType$, $typetoCapacity$, $maxCapacity$.

Let $\vec{f_a}$ be a vector of length $r-1$ such that $\vec{f_a}[i] = 2^i$ for each $0 \leq i < r-1$.
Let $\vec{f_s}$ be a vector of length $eq(r+1)$ such that $\vec{f_s}[i] = (maxCapacity[r+] + 1)^i$ for each $0 \leq i < eq[r+1]$. ▷ *We construct $\vec{f_a}$ and $\vec{f_s}$ only once for each depth $r \in \{1, \ldots, d\}$.*
Let $ancID = 0$.
Let $subtreeID = 0$.
for all $u \in N(v)$ **do**
    if $depth_T(u) = r+1$ **then** ▷ *u is a child of v*
        $subtreeVec = subtreeIDtoVec[r, subtreeID]$
        $type_u = IDtoType[r+1, id(u)]$
        if $subtreeVec[type_u] \leq typetoCapacity[r+1, type_u] - 1$ **then**
            $subtreeID = subtreeID + \vec{f_s}[type_u]$ ▷ *increment count of children of $type_u$*
    if $depth_T(u) < r-1$ **then** ▷ *u is an ancestor of v*
        $ancID = ancID + \vec{f_a}[depth_T(u)]$.
return $ancID + subtreeID(2^{r-1})$

Proof. We first claim that k-Path parameterized by k is $\lfloor \frac{k}{2} \rfloor$-twin subgraph reducible and has $k-1$-bounded depth.

Let (G, k) be an instance of k-Path. By definition, if G contains a path of length k then (G, k) is a yes-instance of k-Path, that is k-Path has $k-1$-bounded depth.

We now claim that k-Path is $\lfloor \frac{k}{2} \rfloor$-twin subgraph reducible. To show this, let S be a set of pairwise twin subgraphs, let P be some path of length k in G and let $Z \subseteq S$ be those subgraphs containing some vertex from P. By definition, any path between distinct subgraphs $H, H' \in Z$ must contain some vertex in $G \setminus S$. It follows that $|Z| \leq \lfloor \frac{k}{2} \rfloor$ meaning k-Path is $\lfloor \frac{k}{2} \rfloor$-twin subgraph reducible. ◀

A *t-ranking* of a graph G, for some integer t, is a t-coloring of G such that, for every pair of vertices x and y with $c(x) = c(y)$ and for every path between x and y, there is a vertex z on this path with $c(z) > c(x)$. The rank of G, is the smallest t for which G admits a t-ranking. The k-Vertex Ranking problem is defined as follows.

k-Vertex Ranking
Input: graph $G = (V, E)$,
Parameter: k,
Question: does there exist a k-ranking of G?

The k-Vertex Ranking problem is also interesting as it is known that the treedepth of a graph is equal to its vertex ranking number and also to the minimum height of any elimination tree of G (see [35], Chapter 6 for a fuller discussion of treedepth). It follows that, by showing k-Vertex Ranking is in TLFPT, the same also holds for computing treedepth.

We also show that Theorem 42 can be leveraged for problems over objects which are not necessarily graphs. We showcase this with the classical problem of boolean Satisfiability, which asks whether a given Boolean formula ψ in conjunctive normal form (CNF) is satisfiable. In our choice of parameter, we aim to capture the structural properties of an instance, particularly the relationships between its variables and clauses. We do this by encoding the

formula as a graph. The *incidence graph*, $I(\psi)$, of a formula ψ, is that bipartite graph such that one part contains the variables of ψ and the other part contains the clauses of ψ. Two vertices are adjacent to one another in $I(\psi)$ if one of these vertices corresponds to a clause and the other corresponds to some variable that appears in this clause. We will focus in particular on the treedepth of the incidence graph, which we call its incidence treedepth.

▶ **Theorem 44.** *k-Vertex Ranking admits a TLFPT time kernelization.*

Proof. We first claim that k-Vertex Ranking parameterized by k has 2^{k+1}-bounded depth and is τ-twin subgraph reducible where $\tau(k,i) = (k-1)\cdot k^i + 1$ for $i \geq 1$. The class of graphs which admit a k-ranking are closed under taking minors (Lemma 6.2 in [35]) and a path of length l (on $l+1$ vertices) has rank $\lceil \log_2(l+2) \rceil$ (Eqn. 6.2 of [35]).

Let (G,k) be an instance of k-Vertex Ranking. If G contains a path of length 2^{k+1}, then (G,k) has rank at least $k+1$, hence, k-Vertex Ranking has 2^k-bounded depth. We now claim that k-Vertex Ranking is τ-twin subgraph reducible where $\tau(k,i) = (k-1)\cdot k^i + 1$ for $i \geq 1$. Let S be some subgraph and let T be a set such that $T \cup \{S\}$ is a set of pairwise twin subgraphs. By definition of twin subgraphs, each of these subgraphs has the same size, say they each have size n. We now claim that if $|T| = (k-1)\cdot k^n + 1$, then there is a k-ranking of $G - V(S)$ if, and only if there is a k-ranking of G.

If there is some k-ranking of G, by definition, this is also a k-ranking of $G - V(S)$. Let us now consider the backwards direction. Suppose that there is some k-ranking c for $G - V(S)$. We now describe a coloring c' of G and claim that this is a k-ranking. Towards this, we say that a pair of subgraphs $X, Y \in T \cup \{S\}$ have the same coloring under c, if that neighborhood preserving isomorphism between X and Y also preserves the colors of c. As there are k^n possible colorings of the subgraphs of $T \cup \{S\}$ and $|T| = (k-1)\cdot k^{|S|} + 1$, it follows that there is some set $E \subseteq T$ such that $|E| \geq k$ and the subgraphs of E each have the same coloring under c. We obtain c' from c by coloring the vertices of S such that, S has the same coloring as every subgraph in E under c'.

We now claim that c' is a k-ranking of G. Assume for the sake of contraction that c' is not a k-ranking of G, that is, there is some path $P = (p_1, \ldots, p_\ell)$ such that $c'(p_1) = c'(p_\ell)$ and, for every $i \in \{1, \ldots, \ell\}$, $c'(p_i) \leq c'(p_1)$. As c is a k-ranking of $G - V(S)$, P is not a path in $G - V(S)$. It follows that $V(S) \cap V(P) \neq \emptyset$.

We claim that P contains vertices from at most k different subgraphs in $T \cup \{S\}$. This will allow us to reason that there is some subgraph X in E which does not contain any vertices in P. This will allow us to map the path P to a path $P' = (p'_1, \ldots, p'_\ell)$ such that $V(S) \cap V(P') = \emptyset$ and $c'(p'_i) = c'(p_i)$, for every $i \in \{1, \ldots, \ell\}$. As P' is a path in $G - V(S)$ we can conclude that c was not a k-ranking of $G - V(S)$.

We will now show that P contains vertices from at most k different subgraphs in $T \cup \{S\}$. By definition of twin subgraphs, for every $S' \in T$, $N(S) \setminus V(S) = N(S') \setminus V(S')$. Let N denote the set $N(S) \setminus V(S)$. It follows that $G - V(S)$ contains the complete bipartite graph $K_{|N|,|T|-1}$ as a subgraph. Given that $K_{|N|,|T|-1}$ has rank $\min(|N|, |T| - 1) + 1$ (this is a folklore result with an easy proof). As $|T| > k$ and $G - V(S)$ has rank at most k, it follows that $|N| \leq k - 1$. Combining this with the fact that, by definition of twin subgraphs, the subgraphs of T are connected components in $G - N$, we find that P contains vertices from at most k different subgraphs in T. That is, there is some subgraph $X \in E$ which does not contain any vertices from P.

We are now ready to construct that path P'. Let $i : V(S) \to V(X)$ be that neighborhood preserving isomorphism between S and X, by definition this isomorphism also preserves the color of vertices. Let $P' = (p'_1, \ldots, p'_\ell)$ be the path such that for $j \in \{1, \ldots, \ell\}$, $p'_j = i(p_j)$, if

$p_j \in V(S)$, and $p'_j = p_j$ otherwise. As P' contains no vertices from S and $c'(p_j) = c'(p'_j)$ for each $j \in \{1, \ldots, \ell\}$, it follows that c was not not a k-ranking $G \setminus S$. As this is a contradiction, we find that c' is a k-ranking of G.

It follows that there is a k-ranking of $G \setminus S$ if, and only if there is a k-ranking of G and so k-Vertex Ranking is τ-twin subgraph reducible where $\tau(k, i) = (k-1) \cdot k^i + 1$ for $i \geq 1$. ◀

We now give the definition of Satisfiability.

Satisfiability
Input: Boolean formula ψ in conjunctive normal form (CNF).
Question: is ψ satisfiable?

Many parameterizations of SAT aim to capture the structural properties of an instance, particularly the relationships between its variables and clauses. A common way to express this structure is to encode the formula as a graph and analyze structural parameters of that graph. Notable examples include the treewidth of the associated graph, as well as the parameter known as *maximum deficiency*, as introduced in [17], which is based on matchings in this graph.

There are several ways to encode a SAT instance as a graph.

Recall that the *incidence graph*, $I(\psi)$, of ψ is that bipartite graph such that one part contains the variables of ψ and the other part contains the clauses of ψ. Two vertices are adjacent to one another in $I(\psi)$ if one of these vertices corresponds to a clause and the other corresponds to some variable that appears in this clause. We will focus on the incidence graph and in particular the treedepth of the incidence graph, which we call its incidence treedepth.

It will be useful to consider the following variant of the incidence graph which we will call the *literal incidence graph*. For a CNF formula ψ, the literal incidence graph, $I_L(\psi)$, of ψ is that bipartite graph such that one part contains the clauses of ψ and one part contains the literals of ψ. That is, for each variable x in ψ, $I_L(\psi)$ contains a pair of vertices, one corresponding to x and the other to $\neg x$. The following lemma will allow us to work on the literal incidence graph.

▶ **Lemma 45.** *For every CNF formula ψ, $td(I_L(\psi)) \leq 2 \cdot td(I(\psi))$.*

Proof. Let ψ be some CNF formula. To simplify notation, let G_I be the incidence graph of ψ and let G_L be the literal incidence graph of ψ. Let $T = (V(G_I), E_T)$ be some minimal treedepth decomposition of G_I. That is, every edge in G_I is between an ancestor and a descendant in T. Further, T has depth $td(G_I)$. We now give a treedepth decomposition of G_I, T', with height at most $2 \cdot td(G_I)$. For every variable x in ψ, let $parent_{T'}(x) = parent_T(x)$, $parent_{T'}(\neg x) = x$ and $children_{T'}(\neg x) = children_T(x)$. Note that T' has depth at most $2 \cdot depth(T)$. It remains to show that T' is indeed a treedepth decomposition for G_L. This follows from the observation that, for every variable in ψ, $anc_{T'}(\neg x) \setminus \{x\} = anc_T(x)$ and $dec_{T'}(\neg x) = dec_T(x)$. As every edge in G_I is between an ancestor and a descendant in T, every edge in G_L is between an ancestor and a descendant in T'. It follows that $td(G_L) \leq 2 \cdot td(G_I)$. ◀

▶ **Theorem 46.** Satisfiability*, when parameterized by its incidence treedepth, admits a TLFPT time kernelization.*

Proof. To apply those techniques developed previously, we will define a graph problem taking as input the literal incidence graph of our formula. We note that there is a unique formula corresponding to any literal incidence graph[8], and vice versa. Due to this, it will be useful to consider the following auxiliary problem called Literal Incidence Graph Satisifability or LIGSat for short.

LIGSat
Input: A literal incidence graph G_L.
Question: is the formula corresponding to G_L satisfiable?

We will now show that LIGSat is in TLFPT when parameterized by treedepth. This will then allow us to show that Satisfiability parameterized by incidence treedepth is in TLFPT via a TLFPT reduction to LIGSat parameterized by treedepth.

We first show that LIGSat is in TLFPT when parameterized by treedepth. In particular, we show that this problem is τ-twin subgraph reducible and has ℓ-bounded depth, for functions $\tau(k, n) = 2^n$ and $\ell(k) = 2^k$.

From Lemma 38, if a graph treedepth at most k then it contains no path of length greater than 2^k. That is, the set of instances of our problem containing a path of length at least 2^k is empty. That is, trivially, LIGSat parameterized by treedepth has negative ℓ-bounded depth, where $\ell(k) = 2^{2k}$.

▷ Claim 46. LIGSat parameterized by treedepth is τ-twin reducible.

Proof. Let (G_L, k) be some instance of LIGSat parameterized by treedepth. Let H be some subgraph of G_L and let S be a set of subgraphs such that $S \cup \{H\}$ consists of pairwise twin subgraphs. By definition of twin subgraphs (Definition 40), every graph in $S \cup \{H\}$ has the same size, which we shall denote by n.

Our task is to prove that if $|S| \geq 2^n$, then (G_L, k) is a yes-instance of LIGSat, if, and only if, $(G_L - V(H), k)$ is. We let ψ denote the formula corresponding to G_L and let ψ' be that formula corresponding to $G_L - V(H)$.

First suppose that (G_L, k) is a yes-instance, and denote by a some satisfying assignment for ψ. To simplify arguments, we let a be an assignment over the literals of ψ. As $S \cup \{H\}$ consists of pairwise twin subgraphs, for any pair of subgraphs X, Y in $S \cup \{H\}$ there is a twin isomorphism i from X to Y. We note that as G_L is bipartite, i must map clauses to clauses and literals to literals. We say that X and Y have the same assignment under a if there exists a twin isomorphism i from X to Y such that, for every literal l in X, $a(l) = a(i(l))$. Recall that every subgraph in $S \cup \{H\}$ has size n. Thus, the number of literal vertices in any such subgraph is at most n, and the number of possible truth assignments to these literal vertices is at most 2^n. As $|S \cup \{H\}| \geq 2^n + 1$, there must exist some pair of subgraphs in $S \cup \{H\}$ with the same assignment under a. Without loss of generality these are H and some other subgraph $X \in S$.

Let i_H be that twin isomorphism from H to X. We now claim that a is a satisfying assignment for ψ'. Consider some clause C in ϕ which is satisfied by some literal l in H. Then either C is in H, in which case C does not appear in ϕ', or $i_H(l) \in C$, since H and X

[8] To allow literals and clauses in $V(G_L)$ to be distinguished, we shall assume that there is an odd number of clauses (possibly by introducing a dummy clause containing some literal and its negation). Note that there is always an even number of literals.

are twin subgraphs. In the latter case, C is satisfied by the literal $i_H(l)$, since X has the same assignment as H under a. Thus, a is a satisfying assignment for ϕ'.

We now consider the other direction. Suppose that there is some satisfying assignment a' for ψ'. We extend this to an assignment a for ψ as follows. For some arbitrary subgraph $X \in S$, we take a twin isomorphism i from X to H and we assign values to the literals of H such that X and H have the same assignment under a' (i.e. for each literal l in H $a(l) = a(i(l))$).

We now claim that a is a satisfying assignment for ψ. By assumption, every clause in ψ' is satisfied by a' and thus by a, which extends it. Let us now consider some clause C which is in ψ but not in ψ', that is C is in H. As a' is a satisfying assignment ψ', the clause $i(C) \in X$ is satisfied by some literal l' in $G - V(H)$. If $l' \notin X$ then $l' \in C$ by the definition of a twin subgraph isomorphism and C is satisfied. Otherwise, $l' \in X$ and the literal $l = i^{-1}(l')$ is in C and is also true under a, and again C is satisfied. Thus a satisfies every clause of ψ and the claim follows. ◀

That is, we have now shown both directions and (G_L, k) is a yes-instance, if, and only if, $(G_L - V(H), k)$ is a yes-instance. It follows that LIGSat parameterized by treedepth is τ-twin reducible and has ℓ-bounded depth, for functions $\tau(k, n) = 2^n$ and $\ell(k) = 2^k$. Applying Theorem 42, LIGSat parameterized by treedepth is in TLFT.

We now show that there is a TLFPT reduction from Satisfiability parameterized by incidence treedepth to LIGSat parameterized by treedepth. The literal incidence graph can be constructed from a formula in time linear in the number of variables and clauses, and conversely, the formula can be reconstructed from the graph within the same time bound. Further, by Lemma 45, the treedepth of a literal incidence graph is at most 2 times larger than the incidence treedepth of its corresponding formula. Thus, there is a TLFPT reduction from Satisfiability parameterized by incidence treedepth to LIGSat parameterized by treedepth. It follows that Satisfiability parameterized by incidence treedepth is in TLFPT, thus concluding the proof of our theorem. ◀

6 TLFPT via Dynamic Programming: Parameterizing by BFS-Width

In Sections 3–5, we gave TLFPT results via kernelization. We now turn to another workhorse of parameterized complexity: dynamic programming. It may seem that dynamic programming algorithms align more naturally with the multiplicative (rather than additive) definition of FPT. Maintaining a state space bounded by a function of k, a classical algorithm would need to expend $f(k)$ operations for each of the $O(n)$ parts the decomposition processed, yielding a runtime on the order of $f(k) \cdot O(n)$. In fact, we show that by carefully considering our model of computation we can also apply dynamic programming with a TLFPT runtime.

In this section, we parameterize graph problems by *BFS-width*, a parameter which in some sense mirrors treedepth: graphs of bounded treedepth have "nice" DFS trees, a structure we leverage in Section 5, whereas graphs of bounded BFS-width have "nice" BFS trees. The techniques we apply here could plausibly be lifted to other decompositions (e.g. tree decompositions), but the question of *producing* an optimal tree decomposition

in time TLFPT parameterized by treewidth remains open. BFS-width, on the other hand, has the desirable property that producing a decomposition in linear time is straightforward. This natural parameter was only recently formally studied from a parameterized perspective [13].

▶ **Definition 46** (BFS-decomposition, BFS-width, [13]). *For a graph $G = (V, E)$, a BFS-decomposition of G rooted at vertex r is a partition of V into layers $L_0, \ldots, L_d$ satisfying $L_i = \{u \in V : \text{dist}(r, u) = i\}$. The width of a BFS-decomposition is the size of its largest layer. The BFS-width of a graph is the maximum width of any BFS-decomposition it admits.*

In this section, we show that many
classic problems can be solved in TLFPT time when parameterized by BFS-width, using algorithms that we call BFS-nice, defined below. We note that any BFS-decomposition is also a path decomposition (and hence a tree decomposition) and that many algorithms operating on a path or tree decomposition are also BFS-nice.

At a high level, for a given (slice of a parameterized) problem Π_k, our approach consists of:

1. Describing a "BFS-nice" dynamic programming algorithm $\mathcal{A}$ solving the problem Π_k in time $O(n) \cdot f(k)$. (Note: $\mathcal{A}$ is never run on the input.)
2. Describing an algorithm $\mathcal{DC}$ which decomposes and compresses the input graph G in time $O(n) + f(k)$.
3. By simulating $\mathcal{A}$ (in time bounded by some function of k), "compile" it into a Deterministic Finite Automaton-like representation: obtain a state space (including a starting state), accepting states, and transition function which together describe the behavior of $\mathcal{A}$ on an input graph decomposition.
4. Run $\mathcal{DC}$ on the input graph G to obtain its compressed representation. Then, apply the compiled transition function of $\mathcal{A}$ on the compressed representation of G in $O(n)$ time and return YES if $\mathcal{A}$ would.

▶ **Definition 47** (Bicolored induced subgraph). *Let $G = (V, E)$ be a graph and $A, B \subseteq V$ be disjoint sets of vertices. The bicolored induced subgraph of G with respect to A and B is the induced subgraph $G[A \cup B]$ together with a (not necessarily proper) 2-coloring of its vertices assigning color 0 to all vertices in A and color 1 to all vertices in B. When the bicoloring is clear from context, we use the shorthand "bicolored graph $G[A \cup B]$" to mean the bicolored induced subgraph of G with respect to A and B.*

Before formally presenting the main result of the section, we need to detail the steps above.

6.1 Step 1: What is a BFS-nice algorithm?

We take this opportunity to note that many algorithms operating on a path decomposition are also BFS-nice, and that a BFS-decomposition is also a path decomposition.

▶ **Definition 48.** *A dynamic programming algorithm $\mathcal{A}$ which takes as input a graph G and a width-k BFS-decomposition $L_0, \ldots L_d$ of G, and which decides parameterized problem Π is BFS-nice if there exists a function f such that:*

- *$\mathcal{A}$ maintains an internal state σ (initially σ_0, which is independent of the input but possibly dependent on k) which never exceeds $f(k)$ bits in size,*
- *For each $i \in \{1, \ldots, d\}$, σ_i is computed as a function of σ_{i-1} and the bicolored graph $G[L_i \cup L_{i-1}]$ alone, and*
- *$\mathcal{A}$ returns YES or NO as a function of σ_d alone.*

A parameterized problem Π is BFS-nice if there exists a BFS-nice algorithm $\mathcal{A}$ solving Π.

This definition captures many natural dynamic programming algorithms, including the one given in [15] to solve K_3-Coloring with a given a tree decomposition. We generalize this slightly and describe a BFS-nice algorithm for the more general H-Coloring (defined just below), proving that it is BFS-nice.

H-Coloring
Input: graph G.
Question: does there exist a homomorphism from G to H, that is, a map $m : V(G) \to V(H)$ such that $(u, v) \in E(G) \implies (m(u), m(v)) \in E(H)$?

▶ **Lemma 49.** *For any fixed graph H, H-Coloring admits a BFS-nice algorithm.*

Proof. Algorithm 6 is a simple dynamic program which decides H-Coloring for an input graph and its k-width BFS-decomposition. This is nothing revolutionary; cf. [15] Example 11.35, which solves K_3-Coloring given a tree decomposition of width at most k. It is easy to verify that this algorithm is BFS-nice: the internal state which here is denoted by $\mathcal{L}_i$ is of size bounded by a function of k (namely, the number of assignments of $[k] \to V(H)$ – recall that H is part of the problem definition) and is updated exclusively as a function of the previous internal state and the bicolored graph $G[L_i \cup L_{i-1}]$, and the algorithm returns YES or NO as a function of its final state $\mathcal{L}_d$. ◀

Algorithm 6 BFS-nice H-Coloring algorithm

Require: Graph $G = (V, E)$ on n; integers k and d; BFS-decomposition of G into layers $L_0 \dots L_d$ each of size at most k; graph H with which to color G.
1: Compute all $|V(H)|$ labelings of the root layer L_0 with $V(H)$. Denote $\mathcal{L}_0$ the set of these (partial) solutions (i.e. labelings) any of which is locally admissible for the vertices in bag L_0 (since L_0 consists of a single vertex, that vertex can be mapped to any vertex in H).
2: **for all** $i \in \{1, \dots d\}$ **do**
3: Compute all possible labelings of layer L_i. Denote these by $\mathcal{L}_i$.
4: **for all** $\ell_i \in \mathcal{L}_i$ **do**
5: **if** $\nexists \ell_{i-1} \in \mathcal{L}_{i-1}$ such that $\ell_{i-1} \cup \ell_i$ is a an H-coloring of $G[L_i \cup L_{i-1}]$ **then**
6: Remove ℓ_i from $\mathcal{L}_i$
7: **if** $\mathcal{L}_d = \emptyset$ **then return** NO
8: **else return** YES

6.2 Step 2: Compressing small graphs into registers

Leveraging the fact that we may assume that the word size w (the number of bits which can be stored in a register) exceeds any $f(k)$ for any function f we choose (Lemma 5), we can fit encodings of mathematical objects of size bounded by some function of k in a single register. For example, a simple labeled k-vertex graph might be represented by its adjacency matrix using k^2 single-bit registers. The analogous "compressed" representation is a single k^2-bit word. Actually, a $\binom{k}{2}$-bit word suffices, since the diagonal is all-zero and the matrix is symmetric. As shown in the figure, the $\binom{k}{2}$-bit representation of a k-vertex graph H on k vertices can be constructed as the sum of $|E(H)|$ terms in $[2^{\binom{k}{2}}]$. We define `kPunch` to be the lookup table storing these terms, indexed by the adjacencies they encode. Analogously, a $2k$-vertex balanced bipartite graph H can be represented as the sum of $|E(H)|$ terms from

$[2^{k^2}]$, and we define `kkPunch` the lookup table storing the set of integers $[2^{k^2}]$ indexed by the adjacencies they encode.

We provide a little more illustration of this idea now. A graph $G = (V, E)$ on k vertices and m edges with $E = \{e_1, \ldots, e_m\}$ can be obtained as the union of m graphs, namely $(V, E) = \cup_{1 \leq i \leq m}(V, \{e_i\})$. Fixing k, we can denote by $P_{i,j} = ([k], \{(i, j)\})$ the *punch graph* encoding the edge (i, j) for k-vertex graphs. Then any k-vertex graph G with m edges is the union of m different punch graphs. The adjacency matrix of G is correspondingly the sum of the adjacency matrices of m different punch graphs. The table `kPunch` exactly stores the (compressed) adjacency matrices of all punch graphs on k vertices. This is illustrated in Figure 3.

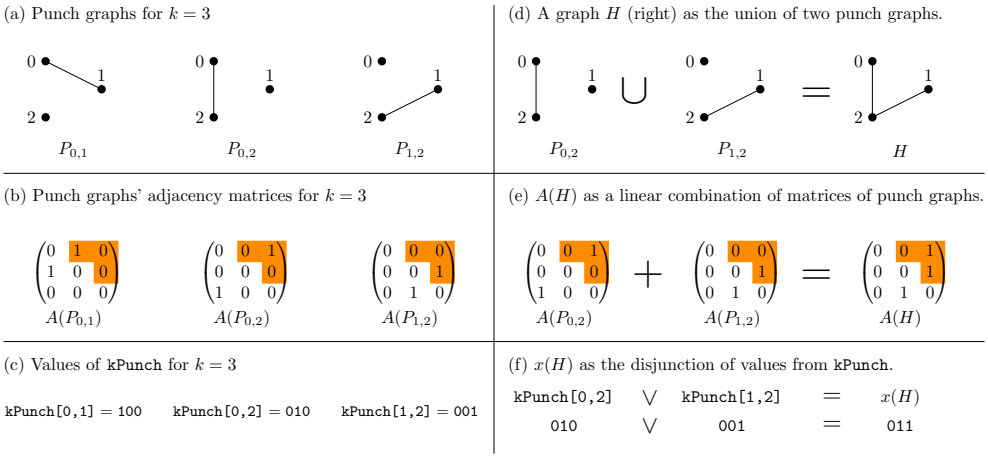

Figure 3 Illustration of punch graphs and the lookup table `kPunch` when $k = 3$. The word $x(H) = \texttt{011}$ represents the 3-vertex graph H with $V(H) = \{0, 1, 2\}$ and $E(H) = \{(0, 2), (1, 0)\}$. More generally, for any graph H, $b(H)$ can be computed as $x(H) = \bigvee_{(u,v) \in E(H)} \texttt{kPunch}[u, v]$.

Likewise, a balanced bigraph $G = ([k] \cup [k + 1, 2k], E)$ on $2k$ vertices and m edges can be obtained as the union of m bigraphs on $2k$ vertices and 1 edge. The graph G and each of these one-edge bigraphs can be represented by a k^2-bit word (the compressed version of its biadjacency matrix). Again, the k^2 bit word representing G can be computed as the sum of the m different k^2-bit "punch" words. Punch words for balanced $2k$ vertex bigraphs are stored in the lookup table `kkPunch`, indexed by the adjacencies which they encode, so that $\texttt{kkPunch}[i, j]$ is the k^2-bit word encoding the bigraph with an edge from i to $k + j$.

Our algorithm $\mathcal{DC}$ (Algorithm 7) takes as input a graph G together with a vertex r and returns a compressed representation of the entire graph. This is done by first creating its BFS decomposition rooted at r and then computing a special symbol b_i for each consecutive pair of layers of the decomposition so that b_i fits in a single register but fully describes both layers and their relationship to one another. In particular, b_i encodes: the size of each layer; the adjacency matrix of each layer; and the biadjacency matrix between the two layers. This is illustrated in Figures 4 and 5. In that example the tuple $(|L_i|, |L_{i-1}|, x_i, x_{i-1}, y_i) = (2, 3, \texttt{011}, \texttt{100}, \texttt{101010000})$ fully describes the two layers shown. Noting that $|L_i| = 3 = \texttt{11}_2$ and $|L_{i-1}| = 2 = \texttt{10}_2$, it follows that the word $b_i = \texttt{111001110010101000}$ exactly encodes the two layers as well.

In our algorithm, too, x_i is used to denote that word encoding the internal adjacency matrix of each layer, and y_i that word encoding the biadjacency matrix between any two

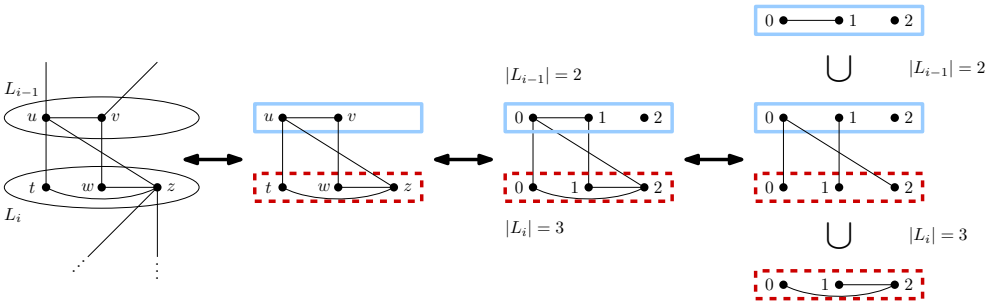

Figure 4 Illustrating the ideas used in Algorithm 7 for $k = 3$. Left: two consecutive layers of a decomposition. Center left: a bicolored graph representing the two layers. Center right: the inclusion of the layers' sizes allows the addition of a dummy vertex, so each graph is on k vertices. Right: the graph is the union of a bigraph on $2k$ vertices and two graphs on k vertices each.

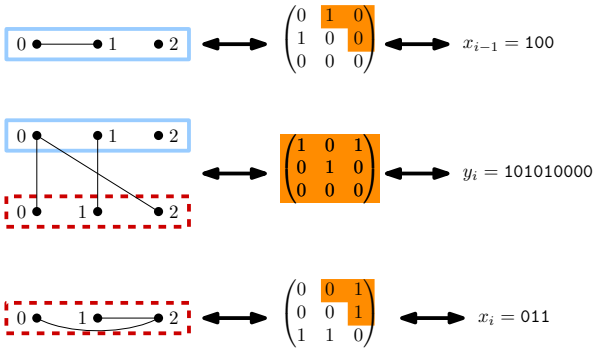

Figure 5 Illustrating how the two graphs and the bicolored bigraph from Figure 4 can be encoded are encoded binary strings.

layers.

▶ **Lemma 50.** *Algorithm 7 runs in time $O(n) + f(k)$ and fully encodes the graph G into d integers $b_1, \ldots, b_d$.*

Proof. Line 1 can be done in linear time. A breadth-first search takes $O(|V| + |E|)$ time [7]; the tables defined each are of size linear in that of the input, and are indexed by the vertex set V (in the case of l and index and d (in the case of $|L_i|$ both of which are bounded by n.
Line 2 takes time $f(k)$.
The loop on line 3 take time d in total.
Line 5 iterates exactly $|E|$ times. Each iteration of the loop takes constant time.
Line 12 iterates d times. Each iteration consists of a single bitwise operation which can be done in constant time. It is clear from the description of the algorithm each word b_i returned encodes the bicolored graph $G[L_i \cup L_{i-1}]$, as illustrated in Figures 4 and 5. ◀

Algorithm 7 Decomposition and Compression algorithm ($\mathcal{DC}$)

Require: Graph $G = (V, E)$ on n vertices; vertex r which roots a BFS tree of width at most k in G.

Compute the BFS tree rooted at r. Denote by d the depth of the tree, and by $L_0, \ldots, L_d$ its layers such that $L_0 = \{r\}$. Define the following functions using lookup tables:

- $l : V \to [0, d]$ which returns the depth of a vertex (the index of the layer it belongs to)
- index : $V \to [0, k-1]$ which returns the index of a vertex in its layer
- $|L_i|$ which returns the number of vertices in a layer i.

Initialize the two-dimensional arrays `kPunch` and `kkPunch` as defined above.
for all $i \in [d]$ **do**
    Set $x_i = 0$ and $y_i = 0$.
for all $(u, v) \in E$ **do**
    $i,\ j = \text{index}(u), \text{index}(v)$
    if $l(u) == l(v)$ **then** ▷ *u and v are in the same layer.*
        $x_{l(u)} = x_{l(u)} \vee$ `kPUNCH[i,j]` ▷ *Bitwise logical OR*
    else ▷ *$|l(u) - l(v)| = 1$ – adjacent layers.*
        if $l(u) < l(v)$ **then** swap u and v and swap i and j. ▷ *Ensure $l(u) > l(v)$*
        $y_{l(u)} = y_{l(u)} \vee$ `kkPUNCH[i,j]`
for all $i \in \{1, \ldots, d\}$ **do**
    Do $b_i = |L_i|$. ▷ *Encoding size of layer L_i*
    Do $b_i = b_i << \lfloor \log_2 k \rfloor + 1$ ▷ *Left-shift by $\lfloor \log_2 k \rfloor + 1$ bits.*
    Do $b_i = b_i \vee |L_{i-1}|$. ▷ *Encoding size of layer L_{i-1}*
    Do $b_i = b_i << \binom{k}{2}$ ▷ *Left-shift by $\binom{k}{2}$ bits.*
    Do $b_i = b_i \vee x_i$ ▷ *Encoding adjacencies within layer i.*
    Do $b_i = b_i << \binom{k}{2}$ ▷ *Left-shift by $\binom{k}{2}$ bits.*
    Do $b_i = b_i \vee x_{i-1}$ ▷ *Encoding adjacencies within layer $i-1$.*
    Do $b_i = b_i << k^2$ ▷ *Left-shift by k^2 bits.*
    Do $b_i = b_i \vee y_i$ ▷ *Encoding adjacencies between layers i and $i-1$.*
return $b_1, \ldots, b_d$. ▷ *Each word b_i fully encodes the bicolored graph $G[L_i \cup L_{i-1}]$.*

6.3 Step 3: state space and transition function of $\mathcal{A}$

We had alluded earlier to the internal state σ of $\mathcal{A}$ and its properties. Recall in particular:

- σ is initially σ_0, which is independent of the input but possibly dependent on k
- σ never exceeds $f(k)$ bits in size for some (possibly non-computable) function f
- At each layer i of the BFS decomposition, σ_i is computed as a function of σ_{i-1} and the bicolored graph $G[L_i \cup L_{i-1}]$ alone.
- $\mathcal{A}$ returns YES or NO as a function of the final state σ_d alone.

We denote by $\mathcal{B}_k$ the set of bicolored (not necessarily bipartite) graphs with at most k vertices of each color. These correspond to the possible values that a bicolored graph $G[L_i \cup L_{i-1}]$ can take. Also, we denote by Σ_k the set of possible values of σ_i over all inputs to $\mathcal{A}$ and integers i. Note that $\max_{\sigma \in \Sigma_k} |\sigma| \leq f(k)$. We shall compute the transition function $\varsigma : \Sigma_k \times \mathcal{B}_k$ which exactly describes how σ_i depends on σ_{i-1} and the bicolored graph $G[L_i \cup L_{i-1}]$ in the execution of $\mathcal{A}$.

This is very straightforward if f is computable; simply enumerate all structures on at most $f(k)$ bits (since Σ_k is necessarily a subset of this) and simulate $\mathcal{A}$ on tuples consisting

of such a structure and a bicolored graph from $\mathcal{B}_k$. However, if f is not computable, we can instead compute ς using (fittingly) breadth-first search, by simply maintaining a fringe of all the states reachable from σ_0 by $\mathcal{A}$; since $f(k)$ is finite, it follows that Σ_k is finite, and our BFS will eventually terminate. Thus we can compute, with respect to any fixed k which bounds the BFS-width of the input of $\mathcal{A}$: the state space Σ_k of $\mathcal{A}$; the set of accepting states (in the finite-state automaton sense) Σ_k^{YES}; and its transition function ς.

The transition function $\varsigma : \Sigma_k \times \mathcal{B}_k \to \Sigma_k$ is then "compressed" (in the same sense as we compress G above): we make use of the bijection described in Algorithm 7 which encodes bicolored graphs as single words (or equivalently integers), which we denote $b : \mathcal{B}_k \to N$, and also fix an arbitrary bijection $s : \Sigma_k \to [|\Sigma_k|]$. Note that both the domain of b $\text{dom}(b)$ and the domain of s $\text{dom}(s)$ are (interpretable as) sets of bitstrings of length bounded by a function of k, namely $k^2 + 2\left(\lfloor \log k \rfloor + 1 + \binom{k}{2}\right)$ in the case of b and $\lfloor \log |\Sigma_k| \rfloor + 1$ in the case of s.

Now any tuple $(b_i, s_i) \in \text{dom}(b) \times \text{dom}(s)$ can be encoded as a single bitstring $b_i \frown s_i$ obtained by left-shifting b_i by $\lfloor \log |\Sigma_k| \rfloor + 1$ bits and summing the result with s_i. Thus, the rightmost $\lfloor \log |\Sigma_k| \rfloor + 1$ bits of $b_i \frown s_i$ are equal to s_i and the right-shift of $b_i \frown s_i$ by $\lfloor \log |\Sigma_k| \rfloor + 1$ bits is exactly b_i. Note that:

- $b_i \frown s_i$ is of size bounded by a function of k (and so at most the word size by Lemma 5).
- each of b_i and s_i can be retrieved in constant time from $b_i \frown s_i$
- $b_i \frown s_i$ can be computed in constant time from b_i and s_i

This enables us to encode the transition function ς as a lookup table `transition` (again, of size bounded by a function of k) indexed by $\text{dom}(b) \times \text{dom}(s)$. Then $\texttt{transition}[b(B) \frown s(\sigma)] = s(\varsigma(B, \sigma))$ for each $B \in \mathcal{B}_k$, $\sigma \in \Sigma_k$. Also, we can compress the starting state σ_0 as `start` $= s(\sigma_0)$. Lastly, we produce a table `accepting` holding all elements of $\{s(\sigma) : \sigma \in \Sigma_k^{\text{YES}}\}$. Note that checking for membership of `accepting` can be done in time $f(k)$.

6.4 Step 4: applying `transition` over $(b_1, b_2, \ldots, b_d)$

We can now simulate $\mathcal{A}$ on input graph G by applying our `transition` table and the compressed encoding $(b_1, b_2, \ldots, b_d)$ of G which is output by Algorithm 7. The algorithm is very simple:

Algorithm 8 Applying the `transition` over the compressed input.

Require: Graph $G = (V, E)$, vertex r of G which roots a BFS tree with layers $L_0, \ldots L_d$ each of size at most k, BFS-nice algorithm $\mathcal{A}$.

1: Obtain $(b_1, b_2, \ldots, b_d)$ by running Algorithm 7 from Section 6.2.
2: Obtain the word `start`, tables `transition` and `accepting`, and functions s, b and compressing its outputs as detailed in Section 6.3 above.
3: Do `state` = `start`
4: **for all** $1 \leq i \leq d$ **do**
5: ⌊ Do `state` = `transition`$[b_i \frown$ `state`$]$
6: **if** `state` $\in$ `accepting` **then return** YES
7: **else**
8: ⌊ **return** NO

▶ **Lemma 51.** *Algorithm 8 terminates in time $O(n) + f(k)$ and returns YES on an input graph G with BFS decomposition $L_0, \ldots, L_d$ if and only if $\mathcal{A}$ returns YES on the same input.*

Proof. The runtime of line 1 is $O(n + f(k))$ as already mentioned in Lemma 50.
As described in the Section 6.3 above, Line 2 terminates in time bounded by some function of k, since it does not depend on the input G, only on k and on the algorithm $\mathcal{A}$ which is fixed.
The loop at line 4 iterates $d \leq n$ times, and the execution of line 5 takes constant time: each of computing $b_i \smile$ `state` a lookup in `transition` take constant time.
To check whether `state` appears in the table `accepting` on line 6 takes time bounded by a function of k, since both objects are bounded in size by such a function.
The correctness of the algorithm is straightforward to verify; on the ith iteration of the loop on line 4 the value of `state` is equal to $s(\sigma_i)$ where σ_i is the actual internal state which algorithm $\mathcal{A}$ would have immediately after processing the ith layer of the BFS-decomposition. The algorithm then returns YES if and only if $\mathcal{A}$ would return YES with G and $L_0, , \ldots, L_d$ as input. ◀

We can now state our main result.

▶ **Theorem 52.** *If $\mathcal{A}$ is a BFS-nice algorithm solving parameterized problem Π, then there exists an algorithm $\mathcal{A}'$ solving Π in time $O(n) + f(k)$.*

Proof. Follows directly from Algorithm 8 and Lemma 51. ◀

▶ **Corollary 53** (Of Theorem 52 and Lemma 49)**.** *For any fixed graph H, H-Coloring is in TLFPT parameterized by the BFS-width of the input graph.*

We take this opportunity to note that, interestingly, 3-Coloring is then TLFPT parameterized by either treedepth (Theorem 39) or BFS-width.

7 Discussion and Future Work

Inspired by the realities of modern computational needs, ours is an early exploration of truly linear FPT. Here distinctions between the additive and multiplicative definitions of FPT make a difference: LFPT – $O(n) \cdot f(k)$ – strictly contains TLFPT – $O(n) + f(k)$.

Towards being of service to practical computation, we have concentrated on the *positive toolkit* of TLFPT. Via appropriate parameterizations and paying attention to the data structures involved in linear-time algorithmics, we placed many problems in TLFPT including: SAT, Vertex Cover, Min-Max Matching, $(n-k)$-Coloring, Diverse Pair of Matchings, k-Path, k-Vertex Ranking and H-Coloring.

In the case of H-Coloring, where we parameterized by BFS-width and used dynamic programming, the similarity to parameterizations by tree-width is evident. So we ask:

▶ **Question 1.** *Is MSO_2 model checking in TLFPT parameterized by BFS-width?*

Turning to k-Path, our result is a first step towards being able to integrate TLFPT and the graph minors machinery; thus we ask:

▶ **Question 2.** *For which families $\{H_i\}_{i\in\mathbb{N}}$ is H_i-minor TLFPT parameterized by $|H_i|$?*

Although not our focus here, the negative toolkit is equally important. By diagonalization, we have shown that there exist parameterized problems in LFPT (and thus in FPT) that are not in TLFPT. However, still lacking natural TLFPT-hard problems and techniques for proving TLFPT-hardness, we identify the following direction for further work.

▶ **Question 3.** *Which FPT problems are likely not to be in TLFPT?*

As we mentioned in Section 6, it seems plausible to generalize our dynamic programming results to classes of "bounded width". Of course, this would be useful only if we have access to the corresponding decompositions. We ask:

▶ **Question 4.** *Is pathwidth TLFPT parameterized by pathwidth?*

▶ **Question 5.** *Is treewidth TLFPT parameterized by treewidth?*

▶ **Question 6.** *Is rankwidth TLFPT parameterized by rankwidth?*

There is no shortage of directions for further research. Consider, for example, programmatic questions such as determining how TLFPT fares in practice, or how to design TLFPT approximation algorithms. In general, the theory of TLFPT is incipient and we expect its development to spark healthy conversation between theory and practice.

References

1 René van Bevern. Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications. PhD thesis, Universitätsverlag der TU, Berlin, 10 2014. doi:10.14279/depositonce-4131.

2 Hans L. Bodlaender. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. SIAM Journal on Computing, 25(6):1305–1317, December 1996. Publisher: Society for Industrial and Applied Mathematics. URL: https://epubs.siam.org/doi/10.1137/S0097539793251219, doi:10.1137/S0097539793251219.

3 Jonathan F Buss and Judy Goldsmith. Nondeterminism within P^*. SIAM Journal on Computing, 22(3):560–572, 1993.

4 CERN. CERN Annual report 2024. Technical report, CERN, Geneva, 2025. URL: https://cds.cern.ch/record/2931994, doi:10.17181/AnnualReport2024.

5 Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. Theoretical Computer Science, 411(40):3736–3756, 2010. URL: https://www.sciencedirect.com/science/article/pii/S0304397510003609, doi:10.1016/j.tcs.2010.06.026.

6 Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. Journal of Computer and System Sciences, 7(4):354–375, 1973. URL: https://www.sciencedirect.com/science/article/pii/S0022000073800297, doi:10.1016/S0022-0000(73)80029-7.

7 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms (3rd ed.). MIT press, 2016.

8 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. Parameterized Algorithms. Springer, 2015. doi:10.1007/978-3-319-21275-3.

9 Rod G Downey and Michael R Fellows. Fixed-parameter tractability and completeness I: Basic results. SIAM Journal on Computing, 24(4):873–921, 1995.

10 Rodney G Downey, Michael R Fellows, et al. Fundamentals of parameterized complexity, volume 4. Springer, 2013.

11 Rodney G Downey and Michael Ralph Fellows. Parameterized complexity. Springer Science & Business Media, 1999. doi:10.1007/978-1-4612-0515-9.

12 Zdeněk Dvořák, Daniel Král, and Robin Thomas. Testing first-order properties for subclasses of sparse graphs. J. ACM, 60(5), October 2013. doi:10.1145/2499483.

13 David Eppstein, Michael T. Goodrich, and Songyu Liu. Bandwidth vs BFS Width in Matrix Reordering, Graph Reconstruction, and Graph Drawing, 2025. URL: https://arxiv.org/abs/2505.10789, arXiv:2505.10789.

14 Jeff Erickson, Ivor van der Hoog, and Tillmann Miltzow. Smoothing the Gap Between NP and ER. SIAM Journal on Computing, 53(6):FOCS20–102–FOCS20–138, 2024. arXiv:https://doi.org/10.1137/20M1385287, doi:10.1137/20M1385287.

15 Jörg Flum and Martin Grohe. Parameterized Complexity Theory. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. doi:10.1007/3-540-29953-X.

16 Fedor V. Fomin, Petr A. Golovach, Lars Jaffke, Geevarghese Philip, and Danil Sagunov. Diverse pairs of matchings. Algorithmica, 86(6):2026–2040, 2024. URL: https://doi.org/10.1007/s00453-024-01214-7, doi:10.1007/S00453-024-01214-7.

17 John Franco and Allen Van Gelder. A perspective on certain polynomial-time solvable classes of satisfiability. Discrete Applied Mathematics, 125(2-3):177–214, 2003.

18 M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing, STOC '90, page 1–7, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/100216.100217.

19 M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science, pages 719–725 vol.2, 1990. doi:10.1109/FSCS.1990.89594.

20 Michael R Garey and David S Johnson. Computers and intractability, volume 29. wh freeman New York, 2002.

21 Archontia C. Giannopoulou, George B. Mertzios, and Rolf Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. Theor. Comput. Sci., 689:67–95, 2017. URL: https://doi.org/10.1016/j.tcs.2017.05.017, doi:10.1016/J.TCS.2017.05.017.

22 Bernhard Haeupler, Richard Hladík, Václav Rozhoň, Robert E. Tarjan, and Jakub Tetěk. Universal Optimality of Dijkstra Via Beyond-Worst-Case Heaps. In 2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS), pages 2099–2130, 2024. doi:10.1109/FOCS61266.2024.00125.

23 Torben Hagerup. Sorting and searching on the word RAM. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, STACS 98, pages 366–398, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

24 Anne-Sophie Himmel, George B. Mertzios, André Nichterlein, and Rolf Niedermeier. Fast parameterized preprocessing for polynomial-time solvable graph problems. Commun. ACM, 67(4):70–79, March 2024. doi:10.1145/3624713.

25 Bart Jansen. Kernelization for Maximum Leaf Spanning Tree with Positive Vertex Weights. In Tiziana Calamoneri and Josep Diaz, editors, Algorithms and Complexity, pages 192–203, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

26 Frank Kammer. A linear-time kernelization for the Rooted k-Leaf Outbranching Problem. Discrete Applied Mathematics, 193:126–138, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0166218X15002218, doi:10.1016/j.dam.2015.04.028.

27 Richard M. Karp. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, Complexity of Computer Computations, pages 85–103. Plenum Press, New York, 1972.

28 George Karypis, Christian Schulz, Darren Strash, Deepak Ajwani, Rob H. Bisseling, Katrin Casel, Ümit V. Çatalyürek, Cédric Chevalier, Florian Chudigiewitsch, Marcelo Fonseca Faraj, Michael Fellows, Lars Gottesbüren, Tobias Heuer, Kamer Kaya, Jakub Lacki, Johannes Langguth, Xiaoye Sherry Li, Ruben Mayer, Johannes Meintrup, Yosuke Mizutani, François Pellegrini, Fabrizio Petrini, Frances Rosamond, Ilya Safro, Sebastian Schlag, Roohani Sharma, Blair D. Sullivan, Bora Uçar, and Albert-Jan Yzelman. Recent Trends in Graph Decomposition (Dagstuhl Seminar 23331). Dagstuhl Reports, 13(8):1–45, 2024. URL: https://drops.dagstuhl.de/entities/document/10.4230/DagRep.13.8.1, doi:10.4230/DagRep.13.8.1.

29 Tomohiro Koana, Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. Journal of Experimental Algorithmics (JEA), 26:1–30, 2021.

30 Daniel Lokshtanov, M. S. Ramanujan, and Saket Saurabh. Linear time parameterized algorithms for subset feedback vertex set. ACM Trans. Algorithms, 14(1), January 2018. doi:10.1145/3155299.

31 Wolfgang Mader. Homomorphism properties and average edge density of graphs. Mathematical Annals, 174(4):265–268, 1967.

32 Wolfgang Mader. Homomorphiesätze für graphen. Mathematische Annalen, 178(2):154–168, 1968.

33 George Mertzios, André Nichterlein, and Rolf Niedermeier. The Power of Linear-Time Data Reduction for Maximum Matching. Algorithmica, 82, 12 2020. doi:10.1007/s00453-020-00736-0.

34 Wojciech Nadara, Michał Pilipczuk, and Marcin Smulewicz. Computing Treedepth in Polynomial Space and Linear FPT Time. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, 30th Annual European Symposium on Algorithms (ESA 2022), volume 244 of Leibniz International Proceedings in Informatics (LIPIcs), pages 79:1–79:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Inform-

atik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2022.79, doi:10.4230/LIPIcs.ESA.2022.79.

35 Jaroslav Nesetril and Patrice Ossona de Mendez. Sparsity - Graphs, Structures, and Algorithms, volume 28 of Algorithms and combinatorics. Springer, 2012. doi:10.1007/978-3-642-27875-4.

36 Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. A faster parameterized algorithm for treedepth. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I, Lecture Notes in Computer Science, pages 931–942. Springer, 2014. doi:10.1007/978-3-662-43948-7_77.

37 Mihalis Yannakakis and Fanica Gavril. Edge dominating sets in graphs. SIAM journal on applied mathematics, 38(3):364–372, 1980.

A Extended discussion of word-RAM architecture

Because we are interested in truly linear computation, the details of accessing and updating information about a graph are very important. We also therefore need to be clear on our model of computation. We use the word-RAM model [18, 19] because it is the canonical model used for the analysis and design of fine-grained algorithms [22, 14], though this is sometimes left implicit.

A RAM machine [6] classically consists of a finite program operating on an infinite sequence of registers, each with the capacity to hold an arbitrary integer. There are two core versions of the RAM model: one in which reading (or writing) the integer n to a register, or accessing the nth register, takes constant time ("unit-cost RAM"), and another in which the same operations take time $\log |n|$. The former is liable to abuse of the constant-time assumption (e.g. by generating operands of size polynomial in the original input size, each of which can be stored in a single register), whereas the latter is limited compared to the actual hardware algorithms are typically executed on, which can indeed access registers (up to some bound) in constant time.

The word-RAM model [18, 19] is a so-called trans-dichotomous model – it aims to bridge the gap between these two paradigms. The operation set (and the cost of any operation) is the same as for the unit-cost RAM model, but with the additional constraint that each register can only hold words consisting of up to b bits, with $b = \log_2 \lceil N \rceil$ where N is the size of the program input. Thus, arithmetic operations with operands of size at most that of the input can be performed in constant time, but (consistent with the reality that computers have fixed word length) the model forbids "computations that achieve hidden parallelism by doing operations on 'long words'." [19]

Importantly, we will see that it is possible for a word to be long with respect to a parameter k while still being sublinear with respect to the overall input size n. This reflects a sound assumption: that if k is sufficiently small with respect to n, the (real-world) architecture which enables constant-time operations with operands of size n can be leveraged to operate on "long words" of size $f(k) \leq n$.

Recall that $w = \log n$ and that we may assume $n >> k$ (if n is upper-bounded by any function of k then we can simply apply an arbitrary algorithm which decides the problem at hand; the runtime of this algorithm on such an input is then also bounded by a function of k).