Dynamic Programming and Fast Matrix Multiplication*

Frederic Dorn**

Department of Informatics, University of Bergen, PO Box 7800, 5020 Bergen, Norway

Abstract. We give a novel general approach for solving NP-hard optimization problems that combines dynamic programming and fast matrix multiplication. The technique is based on reducing much of the computation involved to matrix multiplication. We exemplify our approach on problems like Vertex Cover, Dominating Set and Longest Path. Our approach works faster than the usual dynamic programming solution for any vertex subset problem on graphs of bounded branchwidth. In particular, we obtain the currently fastest algorithms for Planar Vertex Cover of runtime $O(2^{2.52\sqrt{n}})$, for Planar Dominating Set of runtime exact $O(2^{3.99\sqrt{n}})$ and parameterized $O(2^{11.98\sqrt{k}}) \cdot n^{O(1)}$, and for Planar Longest Path of runtime $O(2^{5.58\sqrt{n}})$. The exponent of the running time is depending heavily on the running time of the fastest matrix multiplication algorithm that is currently $o(n^{2.376})$.

1 Introduction

Dynamic programming is a useful tool for the fastest algorithms solving NP-hard problems. We give a new technique for combining dynamic programming and matrix multiplication and apply this approach to problems like Dominating Set and Vertex Cover for improving the best algorithms on graphs of bounded treewidth.

Fast matrix multiplication gives the currently fastest algorithms for some of the most fundamental graph problems. The main algorithmic tool for solving the All Pair Shortest Paths problem for both directed and undirected graphs with small and large integer weights is to iteratively apply the distance product on the adjacency matrix of a graph [28],[30],[3],[36]. Next to the distance product, another variation of matrix multiplication—the boolean matrix multiplication—is solved via fast matrix multiplication. Boolean matrix multiplication is used to obtain the fastest algorithm for Recognizing Triangle-Free Graphs [22]. Recently,Vassilevska and Williams [34] applied the distance product to present the first truly sub-cubic algorithm for finding a Maximum Node-Weighted Triangle in directed and undirected graphs.

The fastest known matrix multiplication of two $n \times n$-matrices by Coppersmith and Winograd [10] in time $O(n^\omega)$ for $\omega < 2.376$ is also used for the fastest boolean matrix multiplication in same time. Rectangular matrix multiplication of an $(n \times p)$- and $(p \times n)$-matrix with $p < n$ gives the runtime $O(n^{1.85} \cdot p^{0.54})$. If $p > n$, we get time $O(p \cdot n^{\omega-1})$. The time complexity of the current algorithm for distance product is $O(n^3/\log n)$, but for integer entries less than M, where M is some small number, there is an $\widetilde{O}(M \cdot n^\omega)$ algorithm [36]. For the arbitrarily weighted distance product no truly sub-cubic algorithm is known. Though, [34] show that the most significant bit of the distance product can be computed in sub-cubic time, and they conjecture that their method may be extended in order to compute the distance product.

* A preliminary version of this article appeared at the proceedings of the fourteenth Annual European Symposium on Algorithms (ESA 2006), vol. 4168 of LNCS, Springer, 2006, pp. 280–291.

** Email:*frederic.dorn@ii.uib.no*. Supported by the Research Council of Norway.

Numerous problems are solved by matrix multiplication, e.g. see [21] for computing minimal triangulations, [24] for finding different types of subgraphs as for example clique cutsets, and [11] for LUP-decompositions, computing the determinant, matrix inversion and transitive closure, to only name a few. However, for NP-hard problems the common approaches do not involve fast matrix multiplication. Williams [35] established new connections between fast matrix multiplication and hard problems. He reduces the instances of the well-known problems MAX-2-SAT and MAX-CUT to exponential size graphs dependent on some parameter k, arguing that the optimum weight k-clique corresponds to an optimum solution to the original problem instance.

The idea of applying fast matrix multiplication is basically to use the information stored in the adjacency matrix of a graph in order to fast detect special subgraphs such as shortest paths, small cliques—as in the previous example—or fixed sized induced subgraphs. Uncommonly—as in [35]—we do not use the technique on the graph directly. Instead, it facilitates a fast search in the solution space. In the literature, there has been some approaches speeding up linear programming using fast matrix multiplication, e.g. see [32]. For our problems, we consider dynamic programming, which is a method for reducing the runtime of algorithms exhibiting the properties of overlapping subproblems and optimal substructure. A standard approach for getting fast exact algorithms for NP-hard problems is to apply dynamic programming across subsets of the solution space. Famous applications of dynamic programming are, among others, Dijkstra's algorithm SINGLE SOURCE AND DESTINATION SHORTEST PATH algorithm, Bellman-Ford algorithm, the TSP problem, the KNAPSACK problem, CHAIN MATRIX MULTIPLICATION and many string algorithms including the LONGEST-COMMON SUBSEQUENCE problem. See [11] for an introduction to dynamic programming. We present a novel approach to fast computing these subsets by applying the distance product on the structure of dynamic programming.

Many NP-complete graph problems turn out to be solvable in polynomial time or even linear time when restricted to the class of graphs of bounded treewidth. The tree-decomposition detects how "tree-like" a graph is and the graph parameter treewidth is a measure of this "tree-likeness". The corresponding algorithms typically rely on a dynamic programming strategy. Telle and Proskurowski [31] gave an algorithm based on tree-decompositions having width ℓ that computes the DOMINATING SET of a graph in time $O(9^\ell) \cdot n^{O(1)}$. Alber et al. [1] not only improved this bound to $O(4^\ell) \cdot n^{O(1)}$ by using several tricks, but also were the first to give a subexponential fixed parameter algorithm for PLANAR DOMINATING SET.

Recently there have been several papers [18, 8, 14, 17, 19], showing that for planar graphs or graphs of bounded genus the base of the exponent in the running time of these algorithms could be improved by instead doing dynamic programming along a branch-decomposition of optimal branchwidth—both notions are closely related to tree-decomposition and treewidth. Fomin and Thilikos [18] significantly improved the result of [1] for PLANAR DOMINATING SET to $O(2^{15.13\sqrt{k}}k + n^3)$ where k is the size of the solution. The same authors [19] achieve small constants in the running time of a branch-decomposition based exact algorithms for PLANAR VERTEX COVER and PLANAR DOMINATING SET, namely $O(2^{3.182\sqrt{n}})$ and $O(2^{5.043\sqrt{n}})$, respectively. Dorn et al. [14] use the planar structure of sphere cut decompositions to obtain fast algorithms for problems like PLANAR LONGEST PATH in time $O(2^{6.903\sqrt{n}})$. Dynamic programming along either a branch-decomposition or a tree-decomposition of a graph both share the property of traversing a tree bottom-up and combining tables of solutions to problems on certain subgraphs that overlap in a bounded-size separator of the original graph.

Our contribution. We give a new technique for combining dynamic programming and matrix multiplication and apply this approach to problems like DOMINATING SET, VERTEX COVER and LONGEST PATH for improving the best algorithms on graphs of bounded branchwidth. We

introduce the new dynamic programming approach on branch-decompositions. Instead of using tables, it stores the solutions in matrices that are computed via distance product. Since distance product is not known to have a fast matrix multiplication in general, we only consider unweighted and small integer weighted problems with weights of size $M = n^{O(1)}$.

Our approach is fully general. It runs faster than the usual dynamic programming for any vertex subset problem on graphs of bounded branchwidth. It also can be used for tree-decompositions with a structure proposed in [16]. To simplify matters, we first introduce our technique on the Vertex Cover problem on graphs of branchwidth bw and show the improvement from $O(2^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$ to $O(2^{\frac{\omega}{2}\,\mathrm{bw}}) \cdot n^{O(1)}$ where ω is the exponent of fast matrix multiplication (currently $\omega < 2.376$).

Next, we give the general technique and show how to apply it to several optimization problems such as Dominating Set, that we improve from $O(3^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$ to $O(4^{\mathrm{bw}}) \cdot n^{O(1)}$—please note that here ω influences the runtime indirectly. Finally, we show the significant improvement of the low constants of the runtime for the approach on planar graph problems. On Planar Dominating Set we reduce the time to even $O(2^{0.793\omega\,\mathrm{bw}}) \cdot n^{O(1)}$ and hence an improvement of the fixed parameter algorithm in [18] to $O(2^{11.98\sqrt{k}}) \cdot n^{O(1)}$ where k is the size of the dominating set. For exact subexponential algorithms as on Planar Vertex Cover and Planar Dominating Set, this means an improvement to $O(2^{1.06\omega\sqrt{n}})$ and $O(2^{1.679\omega\sqrt{n}})$, respectively. We also achieve an improvement for several variants in [2] and [16].

Since the treewidth tw and branchwidth bw of a graph satisfy the relation $\mathrm{bw} \leq \mathrm{tw}+1 \leq \frac{3}{2}\,\mathrm{bw}$, it is natural to formulate the following question as done in [16]: Given a tree-decomposition and a branch-decomposition, for which graphs is it better to use a tree-decomposition based approach and for which is branch-decomposition the appropriate tool? Table 1 compares our results to [16]. It illustrates that dynamic programming is almost always faster on branch-decompositions when using fast matrix multiplication rather than dynamic programming on tree-decompositions. For Planar Dominating Set it turns out that our approach is always the better one in comparison to [1], i.e., we achieve $O(3.688^{\mathrm{bw}}) < O(4^{\mathrm{tw}})$. For Planar Longest Path, we preprocess the matrices in order to apply our method using boolean matrix multiplication in time $O(2^{2.347\omega\sqrt{n}})$. In Table 1, we also add the runtimes for solving related problems and the runtime improvement compared to [14], [15], and [18], and [19].

Organization of the paper. After giving the basic definitions in Section 2, we exemplify on Vertex Cover the common dynamic programming approaches on tree-decompositions and branch-decompositions in Section 3. Then we introduce in Section 4 the basic idea of applying fast matrix multiplication on the solution space of a problem by using Max Cut as an example and stating an alternative technique than in [35]. This technique will be the main key for our algorithm that employs distance product on the overlapping solutions of dynamic programming. In Section 5, we will see why planarity helps when using dynamic programming with fast matrix multiplication.

2 Definitions

2.1 Width parameters

The graph parameters treewidth and branchwidth have been introduced by Robertson and Seymour in their seminal work on Graph Minors theory [26, 27] and since then played an important role in both, graph theory and algorithm theory.

Table 1. Worst-case runtime in the upper part expressed also by treewidth tw and branchwidth bw of the input graph. The problems marked with '*' are the only one where treewidth may be the better choice for some cutpoint $\mathrm{tw} \leq \alpha \cdot \mathrm{bw}$ with $\alpha = 1.19$ and 1.05 (compare with [16]). The lower part gives a summary of the most important improvements on exact and parameterized algorithms with parameter k. Note that we use the fast matrix multiplication constant $\omega < 2.376$.

	Previous results	New results
Dominating Set (DS)	$O(n2^{\min\{2\,\mathrm{tw}, 2.38\,\mathrm{bw}\}})$	$O(n2^{2\,\mathrm{bw}})$
Vertex Cover* (VC)	$O(n2^{\mathrm{tw}})$	$O(n2^{\min\{\mathrm{tw}, 1.19\,\mathrm{bw}\}})$
Independent DS	$O(n2^{\min\{2\,\mathrm{tw}, 2.38\,\mathrm{bw}\}})$	$O(n2^{2\,\mathrm{bw}})$
Perfect Code*	$O(n2^{\min\{2\,\mathrm{tw}, 2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw}, 2.09\,\mathrm{bw}\}})$
Perfect DS*	$O(n2^{\min\{2\,\mathrm{tw}, 2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw}, 2.09\,\mathrm{bw}\}})$
Maximum 2-Packing*	$O(n2^{\min\{2\,\mathrm{tw}, 2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw}, 2.09\,\mathrm{bw}\}})$
Total DS	$O(n2^{\min\{2.58\,\mathrm{tw}, 3\,\mathrm{bw}\}})$	$O(n2^{2.58\,\mathrm{bw}})$
Perfect Total DS	$O(n2^{\min\{2.58\,\mathrm{tw}, 3.16\,\mathrm{bw}\}})$	$O(n2^{2.58\,\mathrm{bw}})$
Planar DS*	$O(n2^{\min\{1.58\,\mathrm{tw}, 2.38\,\mathrm{bw}\}})$	$O(n2^{\min\{1.58\,\mathrm{tw}, 1.89\,\mathrm{bw}\}})$
Planar Longest Path*	$O(n2^{\min\{2.58\,\mathrm{tw}, 3.29\,\mathrm{bw}\}})$	$O(n2^{\min\{2.58\,\mathrm{tw}, 2.75\,\mathrm{bw}\}})$
Planar DS	$O(2^{5.04\sqrt{n}})$ [19]	$O(2^{3.99\sqrt{n}})$
Planar VC	$O(2^{3.18\sqrt{n}})$ [19]	$O(2^{2.52\sqrt{n}})$
Planar Longest Path	$O(2^{6.9\sqrt{n}})$ [14]	$O(2^{5.58\sqrt{n}})$
Planar Graph TSP	$O(2^{9.86\sqrt{n}})$ [14]	$O(2^{8.15\sqrt{n}})$
Planar connected DS	$O(2^{9.82\sqrt{n}})$ [14]	$O(2^{8.11\sqrt{n}})$
Planar Steiner Tree	$O(2^{8.49\sqrt{n}})$ [14]	$O(2^{7.16\sqrt{n}})$
Planar Feedback Vertex Set	$O(2^{9.26\sqrt{n}})$ [14]	$O(2^{7.56\sqrt{n}})$
Parameterized Planar DS	$O(2^{15.13\sqrt{k}}k + n^3)$ [18]	$O(2^{11.98\sqrt{k}}k + n^3)$
Parameterized Planar VC	$O(2^{5.67\sqrt{k}}k + n^3)$ [19]	$O(2^{3.56\sqrt{k}}k + n^3)$
Param Planar Longest Path	$O(2^{13.6\sqrt{k}}k + n^3)$ [14]	$O(2^{10.5\sqrt{k}}k + n^3)$

Tree-decompositions. Let G be a graph, T a tree, and let $\mathcal{Z} = (Z_t)_{t \in T}$ be a family of vertex sets $Z_t \subseteq V(G)$, called *bags*, indexed by the nodes of T. The pair $\mathcal{T} = (T, \mathcal{Z})$ is called a *tree-decomposition* of G if it satisfies the following three conditions:

- $V(G) = \cup_{t \in T} Z_t$,
- for every edge $e \in E(G)$ there exists a $t \in T$ such that both ends of e are in Z_t,
- $Z_{t_1} \cap Z_{t_3} \subseteq Z_{t_2}$ whenever t_2 is a vertex of the path connecting t_1 and t_3 in T.

The width $\mathrm{tw}(\mathcal{T})$ of the tree-decomposition $\mathcal{T} = (T, \mathcal{Z})$ is the maximum size over all bags minus one.

Branch-decompositions. A *branch-decomposition* (T, μ) of a graph G consists of an ternary tree T (i.e. all internal vertices of degree three) and a bijection $\mu : L \to E(G)$ from the set L of leaves of T to the edge set of G. We define for every edge e of T the *middle set* $\mathrm{mid}(e) \subseteq V(G)$ as follows: Let T_1 and T_2 be the two connected components of $T \setminus \{e\}$. Then let G_i be the graph induced by the edge set $\{\mu(f) : f \in L \cap V(T_i)\}$ for $i \in \{1, 2\}$. The *middle set* is the intersection of the vertex sets of G_1 and G_2, i.e., $\mathrm{mid}(e) := V(G_1) \cap V(G_2)$. The *width* bw of (T, μ) is the maximum order of the middle sets over all edges of T, i.e., $\mathrm{bw}(T, \mu) := \max\{|\,\mathrm{mid}(e)| : e \in T\}$.

Tree- and branchwidth. For a graph G its treewidth $\mathrm{tw}(G)$ and branchwidth $\mathrm{bw}(G)$ is the smallest width of any tree-decomposition and branch-decomposition of G respectively. The two graph

parameters treewidth and branchwidth were introduced by Robertson and Seymour as tools in their seminal proof of the Graph Minors Theorem. The treewidth $\mathrm{tw}(G)$ and branchwidth $\mathrm{bw}(G)$ of a graph G satisfy the relation [27]

$$\mathrm{bw}(G) \leq \mathrm{tw}(G) + 1 \leq \frac{3}{2}\,\mathrm{bw}(G),$$

and thus whenever one of these parameters is bounded by some fixed constant on a class of graphs, then so is the other.

See Figure 1 for an illustration of tree-decompositions and branch-decompositions.

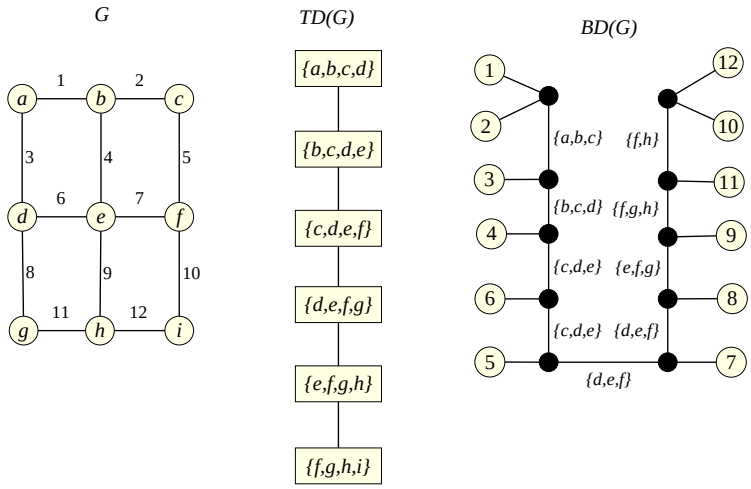

Fig. 1. The left diagram shows a graph with vertex and edge labels. In the middle, we see a tree-decomposition of the graph on the left, and on the right a branch-decomposition.

2.2 Graphs on surfaces

We denote by $\mathbb{S}_0$ the sphere $(x, y, z \mid x^2 + y^2 + z^2 = 1)$. A *line* in $\mathbb{S}_0$ is a subset homeomorphic to $[0, 1]$. An O-arc is a subset of $\mathbb{S}_0$ homeomorphic to a circle. We call a $\mathbb{S}_0$-embedded graph together with its embedding a *plane graph*. To simplify notations we do not distinguish between a vertex of G and the point of $\mathbb{S}_0$ used in the drawing to represent the vertex or between an edge and the line representing it. We also consider G as the union of the points corresponding to its vertices and edges.

Nooses. A subset of $\mathbb{S}_0$ meeting the drawing only in vertices of G is called *G-normal*. If an O-arc is G-normal then we call it *noose* O. The length of a noose O is the number of its vertices and we denote it by $|O|$.

Proposition 1 ([19]). *For any planar graph G,* $\mathrm{bw}(G) \leq \sqrt{4.5n} \leq 2.122\sqrt{n}$.

Sphere-cut Decompositions. For a plane graph G, we define a *sphere-cut decomposition* or *sc-decomposition* (T, μ, π) as a branch-decomposition such that for every edge e of T there exists a noose O_e bounding the two open disks Δ_1 and Δ_2 such that $G_i \subseteq \Delta_i \cup O_e$, $1 \leq i \leq 2$. Thus O_e meets G only in $\mathrm{mid}(e)$ and its length is $|\mathrm{mid}(e)|$. A clockwise traversal of O_e in the drawing of G defines the cyclic ordering π of $\mathrm{mid}(e)$. We always assume that the vertices of every middle set $\mathrm{mid}(e) = V(G_1) \cap V(G_2)$ are enumerated according to π. See Figure 2 for an illustration of a sphere-cut decomposition.

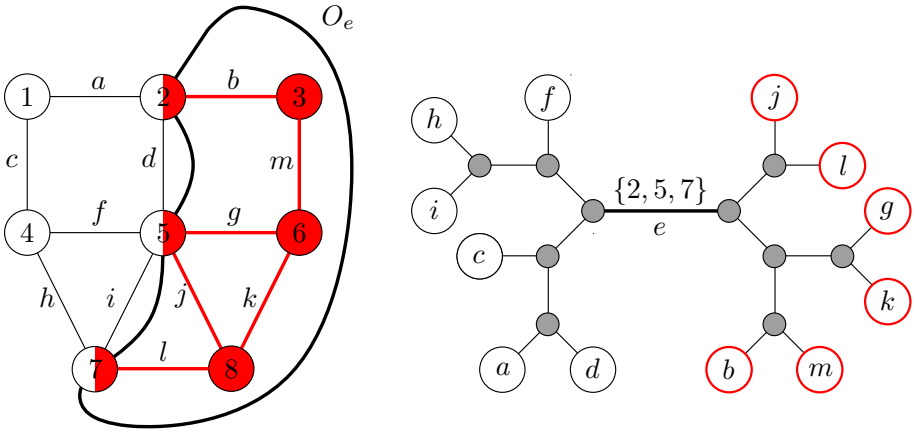

Fig. 2. The left diagram shows a graph which is separated by a noose two parts–emphasized by the red labeling of the one part. On the right, a sc-decomposition, whose labeled edge corresponds to the noose.

2.3 Matrix multiplication.

Two $(n \times n)$-matrices can be multiplied using $O(n^\omega)$ algebraic operations, where the naive matrix multiplication shows $\omega \leq 3$. The best upper bound on ω is currently $\omega < 2.376$ [10].

For rectangular matrix multiplication between two $(n \times p)$- and $(p \times n)$-matrices $B = (b_{ij})$ and $C = (c_{ij})$ we differentiate between $p \leq n$ and $p > n$. For the case $p \leq n$ Coppersmith [9] gives an $O(n^{1.85} \cdot p^{0.54})$ time algorithm (under the assumption that $\omega = 2.376$). If $p > n$, we get $O(\frac{p}{n} \cdot n^{2.376} + \frac{p}{n} \cdot n^2)$ by matrix splitting: Split each matrix into $\frac{p}{n}$ many $n \times n$ matrices $B_1, \ldots, B_{\frac{p}{n}}$ and $C_1, \ldots, C_{\frac{p}{n}}$ and multiply each $A_\ell = B_\ell \cdot C_\ell$ (for all $1 \leq \ell \leq \frac{p}{n}$). Sum up each entry a_{ij}^ℓ overall matrices A_ℓ to obtain the solution.

The *distance product* of two $(n \times n)$-matrices B and C, denoted by $B \star C$, is an $(n \times n)$-matrix A such that

$$a_{ij} = \min_{1 \leq k \leq n} \{b_{ik} + c_{kj}\}, 1 \leq i, j \leq n. \tag{1}$$

The distance product of two $(n \times n)$-matrices can be computed naively in time $O(n^3)$. Zwick [36] describes a way of using fast matrix multiplication, and fast integer multiplication, to compute distance products of matrices whose elements are taken from the set $\{-m, \ldots, 0, \ldots, m\}$ The running time of the algorithm is $\widetilde{O}(m \cdot n^\omega)$. For distance product of two $(n \times p)$- and $(p \times n)$-matrices with $p > n$ we get $\widetilde{O}(p \cdot (m \cdot n^{\omega-1}))$ again by matrix splitting: Here we take the minimum of the entries a_{ij}^ℓ overall matrices A_ℓ with $1 \leq \ell \leq \frac{p}{n}$.

Another variant is the boolean matrix multiplication. The *boolean matrix multiplication* of two boolean $(n \times n)$-matrices B and C, i.e. with only 0,1-entries, is an boolean $(n \times n)$-matrix A such that

$$a_{ij} = \bigvee_{1 \leq k \leq n} \{b_{ik} \wedge c_{kj}\}, 1 \leq i, j \leq n. \tag{2}$$

The fastest algorithm simply uses fast matrix multiplication and sets $a_{ij} = 1$ if $a_{ij} > 0$.

3 Dynamic programming: tree-decompositions vs. branch-decompositions

In Subsection 3.1 and 3.2 we describe how to do dynamic programming on tree-decompositions and branch-decompositions, respectively, using Vertex Cover as an example.

In particular, for a graph G with $|V(G)| = n$ of treewidth tw (branchwidth bw), we will see how the weighted Vertex Cover problem with positive node weights w_v for all $v \in V(G)$ can be solved in time $O(f(\text{tw})) \cdot n^{O(1)}$ ($O(f(\text{bw})) \cdot n^{O(1)}$) where $f(\cdot)$ is an exponential time function only dependent on tw (bw).

3.1 Solving Vertex Cover on tree-decompositions

The algorithm is based on dynamic programming on a rooted tree-decomposition $\mathcal{T} = (T, \mathcal{Z})$ of G. The vertex cover is computed by processing T in post-order from the leaves to the root. For each bag Z_t an optimal vertex cover intersects with some subset U of Z_t. Since Z_t may have size up to tw, this may give 2^{tw} possible subsets to consider. The separation property of each bag Z_t ensures that the problems in the different subtrees can be solved independently.

We root T by arbitrarily choosing a node R. Each internal node t of T now has one adjacent node on the path from t to R, called the *parent node*, and some adjacent nodes toward the leaves, called the *children nodes*.

Let T_t be a subtree of T rooted at node t. G_t is the subgraph of G induced by all bags of T_t. For a subset U of $V(G)$ let $w(U)$ denote the total weight of vertices in U. That is, $w(U) = \sum_{u \in U} w_u$. Define a set of subproblems for each subtree T_t. Each set corresponds to a subset $U \subseteq Z_t$ that may represent the intersection of an optimal solution with $V(G_t)$. Thus, for each vertex cover $U \subseteq Z_t$, we denote by $\mathcal{V}_t(U)$ the minimum weight of a vertex cover S in G_t such that $S \cap Z_t = U$, that is $w(S) = \mathcal{V}_t(U)$. We set $\mathcal{V}_t(U) = +\infty$ if U is not a vertex cover since U cannot be part of an optimal solution. There are $2^{|Z_t|}$ possible subproblems associated with each node t of T. Since T has $O(|V(G)|)$ edges, there are in total at most $2^{\text{tw}} \cdot |V(G)|$ subproblems. The minimum weight vertex cover is determined by taking the maximum over all subproblems associated with the root R.

For each node t the information needed to compute $\mathcal{V}_t(U)$ is already computed in the values for the subtrees. For all children nodes $s_1, \ldots, s_\ell$, we simply need to determine the value of the minimum-weight vertex covers S_{s_i} of G_{s_i} $(1 \leq i \leq \ell)$, subject to the constraints that $S_{s_i} \cap Z_t = U \cap Z_{s_i}$.

With vertex covers $U_{s_i} \subseteq Z_{s_i}$ $(1 \leq i \leq \ell)$ that are not necessarily optimal, the value $\mathcal{V}_t(U)$ is given as follows:

$$\mathcal{V}_t(U) = w(U) + \min \{ \sum_{i=1}^{\ell} \mathcal{V}_{s_i}(U_{s_i}) - w(U_{s_i} \cap U) : U_{s_i} \cap Z_t = U \cap Z_{s_i} \}. \tag{3}$$

The brute force approach computes for all $2^{|Z_t|}$ sets U associated with t the value $\mathcal{V}_t(U)$ in time $O(\ell \cdot 2^z)$ where $z = \max_i\{|Z_{s_i}|\}$.Hence, the total time spent on node t is $O(4^{\mathrm{tw}} \cdot n)$.

Tables. We will see now how this running time can be improved, namely by the use of a more refined data structure. With a table, one has an object that allows to store all sets $U \subseteq Z_t$ in an ordering such that the time used per node is reduced to $O(2^{\mathrm{tw}} \cdot n)$.

Each node t is assigned a table $Table_t$ that is labeled with the sequence of vertices Z_t. When updating $Table_t$ with $Table_{s_i}$, both tables are sorted by the intersecting vertices $Z_t \cap Z_{s_i}$. For computing $\mathcal{V}_t(U)$, this allows to only process the part of the table $Table_{s_i}$ with $U_{s_i} \cap Z_t = U \cap Z_{s_i}$. Thus both tables get processed only once in total.

For achieving an efficient running time, one uses an adequate encoding of the table entries. First define a coloring $c : V(G) \to \{0, 1\}$: For a node t, each set $U \subseteq Z_t$, if $v \in Z_t \setminus U$ then $c(v) = 0$ else $c(v) = 1$. Then sort $Table_{s_i}$ $(1 \leq i \leq \ell)$ to get entries in an increasing order in order to achieve a fast inquiry.

3.2 Solving Vertex Cover on branch-decompositions

On branch-decompositions the algorithm seems to be a bit more circumstantial due to more structure. But as we will see at some point later, this structure is of great help to achieve fast algorithms.

Now we introduce dynamic programming on a rooted branch-decomposition (T, μ) of G. As on tree-decompositions, the vertex cover is computed by processing T in post-order from the leaves to the root. For each middle set $\mathrm{mid}(e)$ an optimal vertex cover intersects with some subset U of $\mathrm{mid}(e)$. Since $\mathrm{mid}(e)$ may have size up to bw, this may give 2^{bw} possible subsets to consider. The separation property of $\mathrm{mid}(e)$ ensures that the problems in the different subtrees can be solved independently.

We root T by arbitrarily choosing an edge e, and subdivide it by inserting a new node s. Let e', e'' be the new edges and set $\mathrm{mid}(e') = \mathrm{mid}(e'') = \mathrm{mid}(e)$. Create a new node *root* r, connect it to s and set $\mathrm{mid}(\{r, s\}) = \emptyset$. Each internal node v of T now has one adjacent edge on the path from v to r, called the *parent edge*, and two adjacent edges toward the leaves, called the *children edges*. To simplify matters, we call them the *left child* and the *right child*.

Let T_e be a subtree of T rooted at edge e. G_e is the subgraph of G induced by all leaves of T_e. For a subset U of $V(G)$ let $w(U)$ denote the total weight of nodes in U. That is, $w(U) = \sum_{u \in U} w_u$. Define a set of subproblems for each subtree T_e. Each set corresponds to a subset $U \subseteq \mathrm{mid}(e)$ that may represent the intersection of an optimal solution with $V(G_e)$. Thus, for each vertex cover $U \subseteq \mathrm{mid}(e)$, we denote by $\mathcal{V}_e(U)$ the minimum weight of a vertex cover S in G_e such that $S \cap \mathrm{mid}(e) = U$, that is $w(S) = \mathcal{V}_e(U)$. We set $\mathcal{V}_e(U) = +\infty$ if U is not a vertex cover since U cannot be part of an optimal solution. There are $2^{|\,\mathrm{mid}(e)|}$ possible subproblems associated with each edge e of T. Since T has $O(|E(G)|)$ edges, there are in total at most $2^{\mathrm{bw}} \cdot |E(G)|$ subproblems. The minimum weight vertex cover is determined by taking the minimum over all subproblems associated with the root r.

For each edge e the information needed to compute $\mathcal{V}_e(U)$ is already computed in the values for the subtrees. Since T is ternary, we have that a parent edge e has two children edges f and g. For f and g, we simply need to determine the value of the minimum-weight vertex covers S_f of G_f and S_g of G_g, subject to the constraints that $S_f \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(f)$, $S_g \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(g)$ and $S_f \cap \mathrm{mid}(g) = S_g \cap \mathrm{mid}(f)$.

With vertex covers $U_f \subseteq \mathrm{mid}(f)$ and $U_g \subseteq \mathrm{mid}(g)$ that are not necessarily optimal, the value $\mathcal{V}_e(U)$ is given as follows:

$$\begin{aligned}\mathcal{V}_e(U) = w(U) + \min\{\ \mathcal{V}_f(U_f) - w(U_f \cap U) + \mathcal{V}_g(U_g) - w(U_g \cap U) - w(U_f \cap U_g \setminus U) :\\ U_f \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(f),\\ U_g \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(g),\\ U_f \cap \mathrm{mid}(g) = U_g \cap \mathrm{mid}(f)\}. \end{aligned} \tag{4}$$

The brute force approach computes for all $2^{|\,\mathrm{mid}(e)|}$ sets U associated with e the value $\mathcal{V}_e(U)$ in time $O(2^{|\,\mathrm{mid}(f)|} \cdot 2^{|\,\mathrm{mid}(g)|})$. Hence, the total time spent on edge e is $O(8^{\mathrm{bw}})$.

Tables. A more sophisticated approach exploits properties of the middle sets and uses tables as data structure. With a table, one has an object that allows to store all sets $U \subseteq \mathrm{mid}(e)$ in an ordering such that the time used per edge is reduced to $O(2^{1.5\,\mathrm{bw}})$.

By the definition of middle sets, a vertex has to be in at least two of three middle sets of adjacent edges e, f, g. You may simply recall that a vertex has to be in all middle sets along the path between two leaves of T.

For the sake of a refined analysis, we partition the middle sets of parent edge e and left child f and right child g into four sets L, R, F, I as follows:

- *Intersection vertices* $I := \mathrm{mid}(e) \cap \mathrm{mid}(f) \cap \mathrm{mid}(g)$,
- *Forget vertices* $F := \mathrm{mid}(f) \cap \mathrm{mid}(g) \setminus I$,
- *Symmetric difference vertices* $L := \mathrm{mid}(e) \cap \mathrm{mid}(f) \setminus I$ and $R := \mathrm{mid}(e) \cap \mathrm{mid}(g) \setminus I$.

We thus can restate the constraints of Equation (4) for the computation of value $\mathcal{V}_e(U)$. Weight $w(U)$ is already contained in $w(U_f \cup U_g)$ since $\mathrm{mid}(e) \subseteq \mathrm{mid}(f) \cup \mathrm{mid}(g)$. Hence, we can change the objective function:

$$\begin{aligned}\mathcal{V}_e(U) = \min\{\ \mathcal{V}_f(U_f) + \mathcal{V}_g(U_g) - w(U_f \cap U_g) :\\ U_f \cap (I \cup L) = U \cap (I \cup L),\\ U_g \cap (I \cup R) = U \cap (I \cup R),\\ U_f \cap (I \cup F) = U_g \cap (I \cup F)\}. \end{aligned} \tag{5}$$

Turning to tables, each edge e is assigned a table $Table_e$ that is labeled with the sequence of vertices $\mathrm{mid}(e)$. More precisely, the table is labeled with the concatenation of three sequences out of $\{L, R, I, F\}$. Define the concatenation '$\|$' of two sequences λ_1 and λ_2 as $\lambda_1 \| \lambda_2$. Then, concerning parent edge e and left child f and right child g we obtain the labels: '$I\|L\|R$' for $Table_e$, '$I\|L\|F$' for $Table_f$, and '$I\|R\|F$' for $Table_g$. $Table_f$ contains all sets U_f with value $\mathcal{V}_f(U_f)$ and analogously, $Table_g$ contains all sets U_g with value $\mathcal{V}_g(U_g)$.

For computing $\mathcal{V}_e(U)$ of each of the $2^{|I|+|L|+|R|}$ entries of $Table_e$, we thus only have to consider $2^{|F|}$ sets U_f and U_g subject to the constraints of Equation (5). Since $\mathrm{mid}(e) \cup \mathrm{mid}(f) \cup \mathrm{mid}(g) = I \cup L \cup R \cup F$, we have that $|I| + |L| + |R| + |F| \leq 1.5 \cdot \mathrm{bw}$. Thus we spend in total time $O(2^{1.5\,\mathrm{bw}})$ on each edge of T.

A technical note: for achieving an efficient running time, one uses an adequate encoding of the table entries. First define a coloring $c : V(G) \to \{0, 1\}$: For an edge e, each set $U \subseteq \mathrm{mid}(e)$, if $v \in \mathrm{mid}(e) \setminus U$ then $c(v) = 0$ else $c(v) = 1$. Then sort $Table_f$ and $Table_g$ to get entries in an increasing order in order to achieve a fast inquiry.

4 Dynamic programming and fast matrix multiplication

The idea of applying fast matrix multiplication and distance product for solving basic graph problems is to multiply the adjacency matrix of the graph with itself in order to fast detect all paths of length two. By iteratively self-multiplying the adjacency matrix one can fast compute all pairs of shortest paths and cycles. For dynamic programming, we exhibit the properties of overlapping subproblems and optimal substructure. In this Section, we show how to compute these subsets by applying the distance product on the structure of dynamic programming instead of the graph itself.

4.1 How fast matrix multiplication improves algorithms

The first application of fast matrix multiplication to NP-hard problems was given by Williams [35], who reduced the instances of the well-known problems Max-2-SAT and Max-Cut to Max Triangle on an exponential size graphs dependent on some parameter k, arguing that the optimum weight k-clique corresponds to an optimum solution to the original problem instance.

Max Cut & fast matrix multiplication In [35], Williams reduces the problems Max-2-SAT and Max Cut to the problem of Max Triangle, that is, finding a maximum weighted 3-clique in a large auxiliary graph. In general, finding 3-cliques in an arbitrary graph G with n vertices can be done in time $O(n^\omega)$: Multiply the adjacency matrix A twice with itself, if A^3 has a non-zero entry on its main-diagonal, then G contains a 3-clique. The key for solving Max Cut is to reduce the problem with the input graph on n vertices to Max Triangle on an exponentially sized graph, i.e., on $O(2^{\frac{n}{3}})$ vertices and $O(2^{\frac{2n}{3}})$ edges. The problem is then solved via distance product on some type of adjacency matrix of the auxiliary graph.

We will now show here a more straight forward way to solve Max Cut that does the trick directly on matrices without making a detour to Max Triangle. Note that both the idea behind it and the runtime are the same than in [35]. We state this alternative technique here for it is the main key for our algorithm.

Max Cut. The Max Cut problem on a graph G is to find a cut of maximum cardinality, that is, a set $X \subset V(G)$ that maximizes the number of edges between X and $V(G) \setminus X$. Max Cut can be solved in runtime $O(2^n)$ by brute force trying all possible subsets of $V(G)$ and storing the maximum number of edges in the cut.

The following lemma is already stated in [35], the new part is the alternative proof:

Lemma 1. Max Cut *on a graph with n vertices can be solved in time $O(2^{\omega\frac{n}{3}})$ with $\omega < 2.376$ the fast matrix multiplication factor.*

Proof: Goal is to find the subset X with the optimal value $\mathcal{V}(X)$ to our problem instance. We partition $V(G)$ into three (roughly) equal sets V_1, V_2, V_3. We consider all subsets $X_{i,j} \subseteq V_i \cup V_j$ $(1 \leq i < j \leq 3)$ and define $\mathcal{V}(X_{i,j}) := |\{\{u,v\} \in E(G) \mid u \in X_{i,j}, w \in V_i \cup V_j \setminus X_{i,j}\}|$.

We compute the value $\mathcal{V}(X)$ for an optimal solution X as follows:

$$\begin{aligned}\mathcal{V}(X) = \max\{\ \mathcal{V}(X_{1,2}) + \mathcal{V}(X_{2,3}) - |\{\{u,v\} \in E(G) \mid u \in X_{2,3} \cap V_2, w \in V_2 \setminus X_{2,3}\}| + \\ \mathcal{V}(X_{1,3}) - |\{\{u,v\} \in E(G) \mid u \in X_{1,3} \cap V_i, w \in V_i \setminus X_{1,3}, (i = 1,3)\}| : \\ X_{1,2} \cap V_1 = X_{1,3} \cap V_1, \\ X_{1,2} \cap V_2 = X_{2,3} \cap V_2, \\ X_{1,3} \cap V_3 = X_{2,3} \cap V_3\}. \end{aligned} \tag{6}$$

That is, we maximize the value $\mathcal{V}$ over all subsets $X_{1,2}, X_{1,3}, X_{2,3}$ such that they form an optimal solution X. When summing up the values $\mathcal{V}(X_{1,2}) + \mathcal{V}(X_{1,3}) + \mathcal{V}(X_{2,3})$, we have to subtract the edges that are counted more than once.

We now use distance product to obtain the optimal solution to Max Cut. In order to apply Equation (6) and not to count the edges several times, we slightly change the values:

- $\mathcal{V}'(X_{1,2}) := \mathcal{V}(X_{1,2})$;
- $\mathcal{V}'(X_{2,3}) := \mathcal{V}(X_{2,3}) - |\{\{u,v\} \in E(G) \mid u \in X_{2,3} \cap V_2, w \in V_2 \setminus X_{2,3}\}|$;
- $\mathcal{V}'(X_{1,3}) := \mathcal{V}(X_{1,3}) - |\{\{u,v\} \in E(G) \mid u \in X_{1,3} \cap V_i, w \in V_i \setminus X_{1,3}, (i = 1,3)\}|$.

We have that

$$\mathcal{V}(X) = \max\{\mathcal{V}'(X_{1,2}) + \mathcal{V}'(X_{2,3}) + \mathcal{V}'(X_{1,3})\}$$

under the constraints of Equation (6).

For $1 \leq i < j \leq 3$, we create the $2^{\frac{n}{3}} \times 2^{\frac{n}{3}}$ matrices $M_{i,j}$ with rows the power set of V_i and columns the power set of V_j. Each row r and column t specify entry $m_{rt}^{i,j} = -\mathcal{V}'(X_{r,t})$ where $X_{r,t} = X_r \cup X_t$ for subsets $X_r \subseteq V_i$ and $X_t \subseteq V_j$. Note that we negate the values because we need to turn the problem into a minimization problem in order to apply the distance product. Then, $\mathcal{V}(X)$ is the minimum entry of $M_{1,3} + (M_{1,2} \star M_{2,3})$,i.e.,

$$\mathcal{V}(X) = \min_{1 \leq r,t \leq 2^{\frac{n}{3}}} \{m_{rt}^{1,3} + m_{rt}^{1,2,3}\},$$

where

$$m_{rt}^{1,2,3} := \min_{1 \leq s \leq 2^{\frac{n}{3}}} \{m_{rs}^{1,2} + m_{st}^{2,3}\}.$$

That is, we first apply distance product on two $2^{\frac{n}{3}} \times 2^{\frac{n}{3}}$ matrices and then add the resulting matrix to a third. Thus, the time we need is $O(n \cdot 2^{\omega \frac{n}{3}} + 2^{\frac{2n}{3}})$. □

4.2 How distance product improves dynamic programming

We introduce a dynamic programming approach on branch-decompositions. Instead of using tables, it stores the solutions in matrices that are computed via distance product. Since distance product is not known to have a fast matrix multiplication in general, we only consider unweighted and small integer weighted problems with weights of size $M = n^{O(1)}$.

Matrices. We start again with the example of Vertex Cover and use the notions of Subsection 3.2. In the remaining section we show how to use matrices instead of tables as data structure for dynamic programming. Then we apply the distance product of two matrices to compute the values $\mathcal{V}(U)$ for a subset $U \subseteq \text{mid}(e)$ of the parent edge e in the branch-decomposition. Recall also from Subsection 3.2 the definitions of intersection-, forget-, and symmetric difference vertices I,F, and L, R, respectively. Reformulating the constraints of Equation (5) in the computation of $\mathcal{V}_e(U)$, we obtain:

$$\begin{aligned} \mathcal{V}_e(U) = \min\{\ & \mathcal{V}_f(U_f) + \mathcal{V}_g(U_g) - w(U_f \cap U_g) : \\ & U \cap I = U_f \cap I = U_g \cap I, \\ & U_f \cap L = U \cap L, \\ & U_g \cap R = U \cap R, \\ & U_f \cap F = U_g \cap F\}. \end{aligned} \tag{7}$$

With $U \cap I = U_f \cap I = U_g \cap I$, one may observe that every vertex cover S_e of G_e is determined by the vertex covers S_f and S_g such that all three sets intersect in some subset $U^I \subseteq I$. From the previous subsection, we got the idea to not compute $\mathcal{V}_e(U)$ for every subset U separately but to simultaneously calculate for each subset $U^I \subseteq I$ the values $\mathcal{V}_e(U)$ for all $U \subseteq \text{mid}(e)$ subject to the constraint that $U \cap I = U^I$. For each of these sets U the values $\mathcal{V}_e(U)$ are stored in a matrix A. A row is labeled with a subset $U^L \subseteq L$ and a column with a subset $U^R \subseteq R$. The entry determined by row U^L and column U^R is filled with $\mathcal{V}_e(U)$ for U subject to the constraints $U \cap L = U^L$, $U \cap R = U^R$, and $U \cap I = U^I$.

We will show how matrix A is computed by the distance product of the two matrices B and C assigned to the children edges f and g: For the left child f, a row of matrix B is labeled with $U^L \subseteq L$ and a column with $U^F \subseteq F$ that appoint the entry $\mathcal{V}_f(U_f)$ for U_f subject to the constraints $U_f \cap L = U^L$, $U_f \cap F = U^F$ and $U_f \cap I = U^I$. Analogously we fill the matrix C for the right child with values for all vertex covers U_g with $U_g \cap I = U^I$. Now we label a row with $U^F \subseteq F$ and a column with $U^R \subseteq R$ storing value $\mathcal{V}_g(U_g)$ for U_g subject to the constraints $U_g \cap F = U^F$ and $U_g \cap R = U^R$. Note that entries have value '$+\infty$' if they are determined by two subsets where at least one set is not a vertex cover.

Lemma 2. *Given a vertex cover $U^I \subseteq I$. For all vertex covers $U \subseteq \text{mid}(e)$, $U_f \subseteq \text{mid}(f)$ and $U_g \subseteq \text{mid}(g)$ subject to the constraint $U \cap I = U_f \cap I = U_g \cap I = U^I$ let the matrices B and C have entries $\mathcal{V}_f(U_f)$ and $\mathcal{V}_g(U_g)$. The entries $\mathcal{V}_e(U)$ of matrix A are computed by the distance product $A = B \star C$.*

Proof: The rows and columns of A, B and C must be ordered that two equal subsets stand at the same position, i.e., U^L must be at the same position in either row of A and B, U^R in either column of A and C, and U^F must be in the same position in the columns of B as in the rows of C. Note that we set all entries with value '∞' to $\sum_{v \in V(G)} w_v + 1$. Now, the only obstacle from applying the distance product (Equation (1)) for our needs, is the additional term $w(U_f \cap U_g)$ in Equation (5). Since U_f and U_g only intersect in U^I and U^F, we substitute entry $\mathcal{V}_g(U_g)$ in C for $\mathcal{V}_g(U_g) - |U^I| - |U^F|$ and we get a new equation:

$$\begin{aligned}\mathcal{V}_e(U) = \min \{\ & \mathcal{V}_f(U_f) + (\mathcal{V}_g(U_g) - |U^I| - |U^F|) : \\ & U \cap I = U_f \cap I = U_g \cap I = U^I, \\ & U_f \cap L = U \cap L = U^L, \\ & U_g \cap R = U \cap R = U^R, \\ & U_f \cap F = U_g \cap F = U^F\}. \end{aligned} \tag{8}$$

Since we have for the worst case analysis that $|L| = |R|$ due to symmetry reason, we may assume that $|U^L| = |U^R|$ and thus A is a square matrix. Every value $\mathcal{V}_e(U)$ in matrix A can be calculated by the distance product of matrix B and C, i.e., by taking the minimum over all sums of entries in row U^L in B and column U^R in C. □

Theorem 1. *Dynamic programming for the* Vertex Cover *problem on weights $M = n^{O(1)}$ on graphs of branchwidth* bw *takes time $\widetilde{O}(M \cdot 2^{\frac{\omega}{2} \cdot \text{bw}})$ with ω the exponent of the fastest matrix multiplication.*

Proof:

For every U^I we compute the distance product of B and C with absolute integer values less than M. We show that, instead of a $O(2^{|L|+|R|+|F|+|I|})$ running time, dynamic programming takes

time $\widetilde{O}(M \cdot 2^{(\omega-1)|L|} \cdot 2^{|F|} \cdot 2^{|I|})$. We need time $O(2^{|I|})$ for considering all subsets $U^I \subseteq I$. Under the assumption that $2^{|F|} \geq 2^{|L|}$ we get the running time for rectangular matrix multiplication: $\widetilde{O}(M \cdot \frac{2^{|F|}}{2^{|L|}} \cdot 2^{\omega|L|})$. If $2^{|F|} < 2^{|L|}$ we simply get $\widetilde{O}(M \cdot 2^{1.85|L|} \cdot 2^{0.54|F|})$ (for $\omega = 2.376$), so basically the same running time behavior. By the definition of the sets L, R, I, F we obtain four constraints:

- $|I| + |L| + |R| \leq \text{bw}$, since $\text{mid}(e) = I \cup L \cup R$,
- $|I| + |L| + |F| \leq \text{bw}$, since $\text{mid}(f) = I \cup L \cup F$,
- $|I| + |R| + |F| \leq \text{bw}$, since $\text{mid}(g) = I \cup R \cup F$, and
- $|I| + |L| + |R| + |F| \leq 1.5 \cdot \text{bw}$, since $\text{mid}(e) \cup \text{mid}(f) \cup \text{mid}(g) = I \cup L \cup R \cup F$.

When we maximize our objective function $\widetilde{O}(M \cdot 2^{(\omega-1)|L|} \cdot 2^{|F|} \cdot 2^{|I|})$ subject to these constraints, we get the claimed running time of $\widetilde{O}(M \cdot 2^{\frac{\omega}{2} \cdot \text{bw}})$. □

4.3 A general technique

Now we formulate the dynamic programming approach using distance product in a more general way than in the previous section in order to apply it to several optimization problems. In the literature these problems are often called *vertex-state* problems. That is, we have given an alphabet λ of vertex-states defined by the corresponding problem. E.g., for the considered Vertex Cover we have that the vertices in the graph have two states relating to an vertex cover U: state '1' means "element of U" and state '0' means "not an element of U". We define a coloring $c : V(G) \to \lambda$ and assign for an edge e of the branch-decomposition (T, μ) a color c to each vertex in $\text{mid}(e)$. Given an ordering of $\text{mid}(e)$, a sequence of vertex-states forms a string $S_e \in \lambda^{|\text{mid}(e)|}$.

Encoding solutions as strings. Recall the definition of concatenating two strings S_1 and S_2 as $S_1 \| S_2$. We then define the strings $S_x(\rho)$ with $\rho \in \{L, R, F, I\}$ of length $|\rho|$ as substrings of S_x with $x \in \{e, f, g\}$ with e parent edge, f left child and g right child. We set $S_e = S_e(I) \| S_e(L) \| S_e(R)$, $S_f = S_f(I) \| S_f(L) \| S_f(F)$ and $S_g = S_g(I) \| S_g(F) \| S_g(R)$. We say that S_e is *formed* by the strings S_f and S_g if $S_e(\rho)$, $S_f(\rho)$ and $S_g(\rho)$ suffice some problem dependent constraints for some $\rho \in \{L, R, F, I\}$. For Vertex Cover we had in the previous section that S_e is formed by the strings S_f and S_g if $S_e(I) = S_f(I) = S_g(I)$, $S_e(L) = S_f(L)$, $S_e(R) = S_g(R)$ and $S_f(F) = S_g(F)$. For problems as Dominating Set it is sufficient to mention that "formed" is differently defined, see for example [16]. With the common dynamic programming approach of using tables, we get to proceed $c_1^{|L|} \cdot c_1^{|R|} \cdot c_2^{|F|} \cdot c_3^{|I|}$ update operations of polynomial time where c_1, c_2 and c_3 are small problem dependent constants. Actually, we consider $|\lambda|^{|L|} \cdot |\lambda|^{|F|} \cdot |\lambda|^{|I|}$ solutions of G_f and $|\lambda|^{|R|} \cdot |\lambda|^{|F|} \cdot |\lambda|^{|I|}$ solutions of G_g to obtain $|\lambda|^{|L|} \cdot |\lambda|^{|R|} \cdot |\lambda|^{|I|}$ solutions of G_e. In every considered problem, we have $c_1 \equiv |\lambda|$, $c_2, c_3 \leq |\lambda|^2$ and $c_1 \leq c_2, c_3$.

Generating the matrices. We construct the matrices as follows: For the edges f and g we fix a string $S_f(I) \in \lambda^I$ and a string $S_g(I) \in \lambda^I$ such that $S_f(I)$ and $S_g(I)$ form a string $S_e(I) \in \lambda^I$. We compute a matrix A with $c_1^{|L|}$ rows and $c_1^{|R|}$ columns and with entries $\mathcal{V}_e(S_e)$ for all strings S_e that contain $S_e(I)$. That is, we label monotonically increasing both the rows with strings $S_e(L)$ and the columns with strings $S_e(R)$ that determine the entry $\mathcal{V}_e(S_e)$ subject to the constraint $S_e = S_e(I) \| S_e(L) \| S_e(R)$.

How to do matrix multiplication. Using the distance product, we compute matrix A from matrices B and C that are assigned to the child edges f and g, respectively. Matrix B is labeled monotonically increasing row-wise with strings $S_f(L)$ and column-wise with strings $S_f(F)$. That is, B has $c_1^{|L|}$ rows and $c_2^{|F|}$ columns. A column labeled with string $S_f(F)$ is duplicated depending

on how often it contributes to forming the strings $S_e \supset S_e(I)$. The entry determined by $S_f(L)$ and $S_f(F)$ consists of the value $\mathcal{V}_f(S_f)$ subject to $S_f = S_f(I)\|S_f(L)\|S_f(F)$.

Analogously, we compute for edge g the matrix C with $c_2^{|F|}$ rows and $c_1^{|R|}$ columns and with entries $\mathcal{V}_g(S_g)$ for all strings S_g that contain $S_g(I)$. We label the columns with strings $S_g(R)$ and rows with strings $S_g(F)$ with duplicates as for matrix B. However, we do not sort the rows by increasing labels. We order the rows such that the strings $S_g(F)$ and $S_f(F)$ match, where $S_g(F)$ is assigned to row k in C and $S_f(F)$ is assigned to column k in B. I.e., for all $S_f(L)$ and $S_g(R)$ we have that $S_f = S_f(I)\|S_f(L)\|S_f(F)$ and $S_g = S_g(I)\|S_g(F)\|S_g(R)$ form $S_e = S_e(I)\|S_e(L)\|S_e(R)$. The entry determined by $S_g(F)$ and $S_g(R)$ consists of the value $\mathcal{V}_g(S_g)$ subject to $S_g = S_g(I)\|S_g(F)\|S_g(R)$ minus an *overlap*. The overlap is the contribution of the vertex-states of the vertices of $S_g(F) \cap F$ and $S_g(I) \cap I$ to $\mathcal{V}_g(S_g)$. That is, the part of the value that is contributed by $S_g(F)\|S_g(R)$ is not counted since it is already counted in $\mathcal{V}_f(S_f)$.

Lemma 3. *Consider fixed strings $S_e(I)$, $S_f(I)$ and $S_g(I)$ such that there exist solutions $S_e \supset S_e(I)$ formed by some $S_f \supset S_f(I)$ and $S_g \supset S_g(I)$. The values $\mathcal{V}_f(S_f)$ and $\mathcal{V}_g(S_g)$ are stored in matrices B and C, respectively. Then the values $\mathcal{V}_e(S_e)$ of all possible solutions $S_e \supset S_e(I)$ are computed by the distance product of B and C, and are stored in matrix $A = B \star C$.*

Proof: For all pairs of strings $S_f(L)$, $S_g(R)$ we compute all possible concatenations $S_e(L)\|S_e(R)$. In row i of B representing one string $S_f(L)$, the values of every string S_f are stored with fixed substrings $S_f(L)$ and $S_f(I)$, namely for all possible substrings $S_f(F)$ labeling the columns. Suppose $S_f(L)$ is updated with string $S_g(R)$ labeling column j of C, i.e., $S_f(L)$ and $S_g(R)$ contribute to forming S_e with substrings $S_e(L)$ and $S_e(R)$. The values of every string $S_g \supset S_g(I)\|S_g(R)$ are stored in that column. For solving a minimization problem we look for the minimum overall possible pairings of $S_f(L)\|S_f(F)$ and $S_g(F)\|S_g(R)$. By construction, a column k of $B = (b_{ij})$ is labeled with the string that that matches the string labeling row k of $C = (c_{ij})$. Thus, the value $\mathcal{V}_e(S_e)$ is stored in matrix A at entry a_{ij} where

$$a_{ij} = \min_k\{b_{ik} + c_{kj}\},\ 1 \le i \le c_1^{|L|},\ 1 \le j \le c_1^{|R|},\ 1 \le k \le c_2^{|F|}.$$

Hence A is the distance product of B and C. □

The following theorem refers especially to all the problems enumerated in Table 1.

Theorem 2. *Dynamic programming for solving vertex-state problems on weights M on graphs of branchwidth* bw *takes time $O(M \cdot \max\{c_1^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} c_2^{\frac{\mathrm{bw}}{2}}, c_2^{\mathrm{bw}}, c_3^{\mathrm{bw}}\})$ with ω the exponent of the fastest matrix multiplication and c_1, c_2 and c_3 the number of algebraic update operations for the sets $\{L, R\}$, F and I, respectively.*

Proof:

For each update step we compute for all possible pairings of $S_f(I)$ and $S_g(I)$ the distance product of B and C with absolute integer values less than M. That is, instead of a $c_1^{2\cdot|L|} \cdot c_2^{|F|} \cdot c_3^{|I|}$ running time, dynamic programming takes time $O(M \cdot c_1^{(\omega-1)|L|} \cdot c_2^{|F|} \cdot c_3^{|I|})$. Note that for the worst case analysis we have due to symmetry reason that $|L| = |R|$. We need time $c_3^{|I|}$ for computing all possible pairings of $S_f(I)$ and $S_g(I)$. Under the assumption that $c_2^{|F|} \ge c_1^{|L|}$ we get the running time for rectangular matrix multiplication: $O(M \cdot \frac{c_2^{|F|}}{c_1^{|L|}} \cdot c_1^{\omega|L|})$. If $c_2^{|F|} < c_1^{|L|}$ we simply get $(M \cdot c_1^{1.85|L|} \cdot c_2^{0.54|F|})$ (for $\omega = 2.376$), so basically the same running time behavior.

For parent edge e and child edges f and g, a vertex $v \in \mathrm{mid}(e) \cup \mathrm{mid}(f) \cup \mathrm{mid}(g)$ appears in at least two out of $\mathrm{mid}(e)$, $\mathrm{mid}(f)$ and $\mathrm{mid}(g)$. From this follows the constraint $|\,\mathrm{mid}(e) \cup \mathrm{mid}(f) \cup \mathrm{mid}(g)| \leq 1.5$ bw which in addition to the constraints $|\,\mathrm{mid}(e)| \leq \mathrm{bw}, |\,\mathrm{mid}(f)| \leq \mathrm{bw}, |\,\mathrm{mid}(g)| \leq \mathrm{bw}$ gives us four constraints altogether: $|L| + |R| + |I| + |F| \leq 1.5$ bw, $|L| + |R| + |I| \leq \mathrm{bw}$, $|L| + |I| + |F| \leq \mathrm{bw}$, and $|R| + |I| + |F| \leq \mathrm{bw}$. When we maximize our objective function $O(M \cdot c_1^{(\omega-1)|L|} \cdot c_2^{|F|} \cdot c_3^{|I|})$, we get above result in dependency of the values of c_1, c_2 and c_3. □

4.4 Application of the technique

The technique can be applied for several optimization problems such as Dominating Set and its variants in order to obtain fast algorithms.

For Dominating Set we have that $c_1 \equiv c_2 = 3$ and $c_3 = 4$. The former running time was $O(3^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$. We have $\widetilde{O}(M \cdot \max\{3^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} 3^{\frac{\mathrm{bw}}{2}}, 3^{\mathrm{bw}}, 4^{\mathrm{bw}}\}) = \widetilde{O}(M \cdot 4^{\mathrm{bw}})$ for node weights M if we use a matrix multiplication algorithm with $\omega < 2.5$ and thus hide the factor ω.

5 Planarity and dynamic programming

For planar embedded graphs, separators have a structure that cuts the surface into two or more pieces onto which the separated subgraphs are embedded on. Miller [25] was the first to describe how to find small simple cyclic separators in planar triangulations. Applying those ideas, one can find small separators whose vertices can be connected by nooses. (e.g. see [4]).

Tree-decompositions and branch-decompositions have been historically the choice when solving NP-hard optimization and FPT problems with a dynamic programming approach (see for example [5] for an overview). Although much is known about the combinatorial structure of tree-decompositions (e.g., [6, 33]) and branch-decompositions (e.g., [18]), there are only few results relating to the topology of tree-decompositions or branch-decompositions of planar graphs (e.g., [7, 12, 13, 23].

Computing sphere-cut decompositions. The main idea to speed-up algorithms obtained by the branch-decomposition approach is to exploit planarity for the three times: First in the upper bound on the branchwidth of a graph and second in the polynomial time algorithm for constructing an optimal branch-decomposition. We present here how planarity is used to improve our dynamic programming approach on graphs of bounded branchwidth.

Our results are based on deep results of Seymour & Thomas [29] on geometric properties of planar branch decompositions. Loosely speaking, their results imply that for a graph G embedded on a sphere $\mathbb{S}_0$, some branch decompositions can be seen as decompositions of $\mathbb{S}_0$ into disks (or sphere-cuts)—so-called sphere-cut decompositions. Sphere-cut decompositions seem to be an appropriate tool for solving a variety of planar graph problems.

The following theorem provides us with the main technical tool. It follows almost directly from the results of Seymour & Thomas [29] (see also [20]).

Theorem 3. *Let G be a connected plane graph of branchwidth at most ℓ without vertices of degree one. There exists an sc-decomposition of G of width at most ℓ and such a branch-decomposition can be constructed in time $O(n^3)$.*

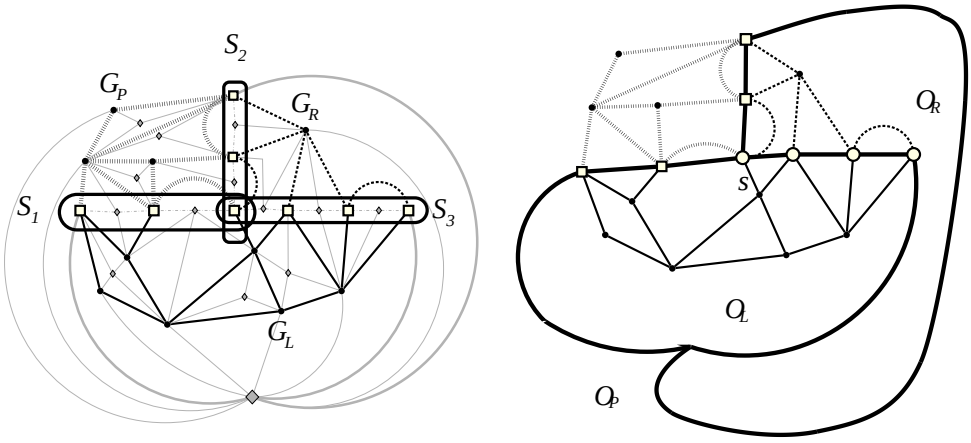

Fig. 3. On the left we see the same graph G as in the last figure. The gray rhombus and gray edges illustrate the radial graph R_G. G is partitioned by the rectangle vertices of S_1, S_2, S_3 into G_L in drawn-through edges, G_R in dashed edges, and G_P in pointed edges. On the right the three nooses O_L, O_R and O_P are marked. Note that the nooses are induced by S_1, S_2, S_3 and the highlighted gray edges on the left hand. All three nooses here intersect in one intersection vertex s.

5.1 Planarity and fast matrix multiplication

Root a given sc-decomposition (T, μ, π). Recall that for three neighboring edges of T, the edge closest to the root is called parent edge e_P and the two other edges, child edges e_L and e_R.

Let O_L, O_R, and O_P be the nooses corresponding to edges e_L, e_R and e_P, and let Δ_L, Δ_R and Δ_P be the disks bounded by these nooses. Due to the definition of branch-decompositions, every vertex must appear in at least two of the three middle sets. We partition the set $(O_L \cup O_R \cup O_P) \cap V(G)$ into three sets accordingly to Subsection 3.2:

- *Intersection vertices* $I := O_L \cap O_R \cap O_P \cap V(G)$,
- *Forget vertices* $F := O_L \cap O_R \cap V(G) \setminus I$,
- *Symmetric difference vertices* $L := O_P \cap O_L \cap V(G) \setminus I$ and $R := O_P \cap O_R \cap V(G) \setminus I$.

See Figure 3 for an illustration of these notions. Observe that $|I| \leq 2$, as the disk Δ_P contains the union of the disks Δ_L and Δ_R. This observation will prove to be crucial for improving dynamic programming for planar graph problems.

With the nice property that $|I| \leq 2$ for all middle sets, we achieve better running times for many discussed problems when we restrict them to planar graphs. The following theorem is the counterpart to Theorem 2 for planarity:

Theorem 4. *Let ω be the exponent of the fastest matrix multiplication and c_1 and c_2 the number of algebraic update operations for the sets $\{L, R\}$ and F, respectively. Then, dynamic programming for solving vertex-subset problems on weights $M = n^{O(1)}$ on planar graphs of branchwidth* bw *takes time* $\widetilde{O}(M \cdot \max\{c_1^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} c_2^{\frac{\mathrm{bw}}{2}}, c_2^{\mathrm{bw}}\})$.

Planar Dominating Set. For Planar Dominating Set with node weights M, the runtime changes from $O(4^{\mathrm{bw}}) \cdot n^{O(1)}$ to $\widetilde{O}(M \cdot 3^{1.188\,\mathrm{bw}}) = \widetilde{O}(M \cdot 3.688^{\mathrm{bw}})$. This runtime is strictly better than the actual runtime of the treewidth based technique of $O(4^{\mathrm{tw}}) \cdot n^{O(1)}$ [1].

Planar Longest Path. For Planar Longest Path, it is not immediately clear how to use matrices since here it seems necessary to compute the entire solution at a dynamic programming step. I.e., in [14] the usual dynamic programming step is applied with the difference that a postprocessing step uncovers forbidden solutions and changes the coloring of the vertices in the L- and R-set. The idea that helps is that we replace the latter step by a preprocessing step, changing the matrix entries of the child edges depending on the change of the coloring. That coloring is only dependent on the coloring of the F-set in both matrices. Hence we do not query the coloring of all three sets L, R and F simultaneously.

Lemma 4. Planar Longest Path *on a planar graph G with branchwidth ℓ can be solved in time $O(2^{2.746\ell}\ell n + n^3)$.*

Proof: According to [14], there are

$$Z := (\frac{z}{2} + 1)^{\frac{\ell}{2}} \cdot 3^{\frac{\ell}{2}}$$

possible ways to update the states of F where $z \leq 3.68133$. That is, we obtain two matrices A, B, A with $4^{\frac{\ell}{2}}$ rows and Z columns and B with Z rows and $4^{\frac{\ell}{2}}$ columns. The rows of A and the columns of B represent the ways $L \cap O_L$ and $R \cap O_R$, respectively, contribute to the solution. The Z columns of A and rows B are ordered such that column i and row i form a valid assignment for all $1 \leq i \leq Z$ and we get

$$C := (-1) \cdot ([(-1) \cdot A] \star [(-1) \cdot B]),$$

the distance product of A and B with inverted signs for obtaining the maximum over the entries. Other than for previous distance product applications, we do not need to post-process matrix C, since the paths in Δ_L and Δ_R do not overlap.

By Theorem 4, we obtain an overall running time of $O(4^{(\omega-1)\frac{\ell}{2}} \cdot 6.68^{\frac{\ell}{2}} \cdot \ell n + n^3) = O(2^{2.746\ell}\ell n + n^3)$ for fast matrix multiplication constant $\omega < 2.376$. □

By Proposition 1 and Lemma 4 we achieve an algorithm for Planar Longest Path of running time $O(2^{5.58\sqrt{n}})$.

6 Conclusions

We established a combination of dynamic programming and fast matrix multiplication as an important tool for finding fast exact algorithms for NP-hard problems. Even though the currently best constant $\omega < 2.376$ of fast matrix multiplication is of rather theoretical interest, there exist indeed some practical sub-cubic runtime algorithms that help improving the runtime for solving all mentioned problems. An interesting side-effect of our technique is that any improvement on the constant ω has a direct effect on the runtime behavior for solving the considered problems. E.g., for Planar Dominating Set; under the assumption that $\omega = 2$, we come to the point where the constant in the computation is 3 what equals the number of vertex states, which is the natural lower bound for dynamic programming. Currently, [34] have made some conjecture on an

improvement for distance product, which would enable us to apply our approach to optimization problems with arbitrary weights.

It is easy to answer the question why our technique does not help for getting faster tree-decomposition based algorithms. The answer lies in the different parameter; even though tree-decompositions can have the same structure as branch-decompositions (see [16]), fast matrix multiplication does not affect the theoretical worst case behavior—though practically it is an improvement. For the planar case, an interesting question arises: can we change the structure of tree-decompositions to attack its disadvantage against sphere cut decompositions? This question has been partially answered in [13] by introducing *geometric tree-decompositions*, where the intersection of two adjacent bags form a noose. Geometric tree-decompositions $\mathcal{T}$ would get applicable to fast matrix multiplication if we had "well-balanced" separators in $\mathcal{T}$. That is, we assume that the three sets L, R, F—defined in the same way for geometric tree-decompositions as for sc-decompositions—are of equal cardinality for every three adjacent bags. Since $|L|+|R|+|F| \leq \mathrm{tw}$, we thus would have that $|L|, |R|, |F| \leq \frac{\mathrm{tw}}{3}$. Applying the fast matrix multiplication method for example to Planar Vertex Cover, this would lead to a $2^{\frac{\omega}{3}\,\mathrm{tw}(\mathcal{T})} \cdot n^{O(1)}$ algorithm, where $\omega < 2.376$. Does every planar graph have a geometric tree-decomposition with well-balanced separators?

Another interesting question, that comes to ones mind, is since the intersection set I is not considered at all for matrix multiplication: Is there anything to win for dynamic programming if we use 3-dimensional matrices as a data structure? That is, if we have the third dimension labeled with $S_e(I)$?

Acknowledgments. Many thanks to Fedor Fomin for his useful comments and his patience, Artem Pyatkin for some fruitful discussions, and an anonymous referee for his comments, and Charis Papadopoulos, and Laura Toma.

References

1. J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier, *Fixed parameter algorithms for dominating set and related problems on planar graphs*, Algorithmica, 33 (2002), pp. 461–493.
2. J. Alber and R. Niedermeier, *Improved tree decomposition based algorithms for domination-like problems*, in Proceedings of the fifth Latin American Theoretical Informatics Symposium (LATIN'02), vol. 2286 of LNCS, Springer, 2002, pp. 613–627.
3. N. Alon, Z. Galil, and O. Margalit, *On the exponent of the all pairs shortest path problem*, Journal of Computer and System Sciences, 54 (1997), pp. 255–262.
4. S. Arora, M. Grigni, D. Karger, P. Klein, and A. Woloszyn, *A polynomial-time approximation scheme for weighted planar graph TSP*, in Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998), ACM, 1998, pp. 33–41.
5. H. L. Bodlaender, *Treewidth: Algorithmic techniques and results*, in Proceedings of the 22nd International Symposium on the Mathematical Foundations of Computer Science (MFCS'97), vol. 1295 of LNCS, Springer, 1997, pp. 19–36.
6. ———, *A tourist guide through treewidth*, Acta Cybernet., 11 (1993), pp. 1–21.
7. V. Bouchitté, F. Mazoit, and I. Todinca, *Chordal embeddings of planar graphs*, Discrete Mathematics, 273 (2003), pp. 85–102.
8. W. Cook and P. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, 15 (2003), pp. 233–248.
9. D. Coppersmith, *Rectangular matrix multiplication revisited*, Journal of Complexity, 13 (1997), pp. 42–49.

10. D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation, 9 (1990), pp. 251–280.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, The MIT Press and McGraw-Hill Book Company, 2001.
12. E. D. Demaine and M. Hajiaghayi, *Bidimensionality: new connections between fpt algorithms and ptass*, in Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005), ACM-SIAM, 2005, pp. 590–601.
13. F. Dorn, *How to use planarity efficiently: new tree-decomposition based algorithms*, in Proceedings of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2007), LNCS, Springer, 2007, p. to appear.
14. F. Dorn, E. Penninkx, H. Bodlaender, and F. V. Fomin, *Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions*, in Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005), vol. 3669 of LNCS, Springer, 2005, pp. 95–106.
15. ———, *Efficient exact algorithms on planar graphs: Exploiting sphere cut decompositions*, 2006. manuscript, http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-2006/2006-006.pdf.
16. F. Dorn and J. A. Telle, *Two birds with one stone: the best of branchwidth and treewidth with one algorithm*, in Proceedings of the seventh Latin American Theoretical Informatics Symposium (LATIN'06), vol. 3887 of LNCS, Springer, 2006, pp. 386–397.
17. F. V. Fomin and D. M. Thilikos, *Fast parameterized algorithms for graphs on surfaces: Linear kernel and exponential speed-up*, in Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), vol. 3142 of LNCS, Berlin, 2004, Springer, pp. 581–592.
18. ———, *Dominating sets in planar graphs: Branch-width and exponential speed-up*, SIAM J. Comput., 36 (2006), pp. 281–309.
19. ———, *New upper bounds on the decomposability of planar graphs*, Journal of Graph Theory, 51 (2006), pp. 53–81.
20. Q.-P. Gu and H. Tamaki, *Optimal branch-decomposition of planar graphs in $O(n^3)$ time*, in Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005), vol. 3580 of LNCS, Springer, Berlin, 2005, pp. 373–384.
21. P. Heggernes, J. A. Telle, and Y. Villanger, *Computing minimal triangulations in time $O(n^\alpha log n) = o(n^{2.376})$*, SIAM Journal on Discrete Mathematics, 19 (2005), pp. 900–913.
22. A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM Journal on Computing, 7 (1978), pp. 413–423.
23. P. N. Klein, *A linear-time approximation scheme for planar weighted TSP*, in 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), IEEE Computer Society, 2005, pp. 647–657.
24. D. Kratsch and J. Spinrad, *Between $O(nm)$ and $O(n^\alpha)$*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03), ACM, 2003, pp. 158–167.
25. G. L. Miller, *Finding small simple cycle separators for 2-connected planar graphs*, Journal of Computer and System Science, 32 (1986), pp. 265–279.
26. N. Robertson and P. D. Seymour, *Graph minors. II. Algorithmic aspects of tree-width*, Journal of Algorithms, 7 (1986), pp. 309–322.
27. ———, *Graph minors X. Obstructions to tree-decomposition*, Journal of Combinatorial Theory Series B, 52 (1991), pp. 153–190.
28. R. Seidel, *On the all-pairs-shortest-path problem in unweighted undirected graphs*, Journal of Computer and System Sciences, 51 (1995), pp. 400–403.
29. P. D. Seymour and R. Thomas, *Call routing and the ratcatcher*, Combinatorica, 14 (1994), pp. 217–241.
30. A. Shoshan and U. Zwick, *All pairs shortest paths in undirected graphs with integer weights*, in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, (FOCS '99), IEEE Computer Society, 1999, pp. 605–615.
31. J. A. Telle and A. Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM J. Discrete Math, 10 (1997), pp. 529–550.

32. P. M. Vaidya, *Speeding-up linear programming using fast matrix multiplication*, in Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS 1989), IEEE Computer Society, 1989, pp. 332–337.
33. J. van Leeuwen, *Graph algorithms*, MIT Press, Cambridge, MA, USA, 1990.
34. V. Vassilevska and R. Williams, *Finding a maximum weight triangle in $n^{(3-\delta)}$ time, with applications*, in Proceedings of the 38th annual ACM Symposium on Theory of computing (STOC 2006), ACM, 2006, pp. 225–231.
35. ———, *A new algorithm for optimal 2-constraint satisfaction and its implications*, Theoretical Computer Science, 348 (2005), pp. 357–365.
36. U. Zwick, *All pairs shortest paths using bridging sets and rectangular matrix multiplication*, Journal of the ACM, 49 (2002), pp. 289–317.