REPORTS IN INFORMATICS

ISSN 0333-3590

Dynamic Programming and Fast Matrix Multiplication

Frederic Dorn

REPORT NO 321 **April 2006**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL http://www.ii.uib.no/publikasjoner/texrap/pdf/2006-321.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

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 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. Our approach is applied to obtain the fastest algorithms for various graph problems such as the fastest algorithm for Planar Independent Set 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 Hamiltonian Cycle 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 Independent Set 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 min-plus product on the adjacency matrix of a graph [18],[20],[3],[26]. Next to the min-plus product or 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 [16]. Recently,Vassilevska and Williams [23] applied the distance product to present the first truly subcubic 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 [6] 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 min-plus square matrix multiplication (distance product) is $O(n^3/\log n)$, but for integer entries less than m, where m is some small number, there is an $O(mn^\omega)$ algorithm [25]. For the arbitrarily weighted distance product no truly sub-cubic algorithm is known. Though [23]

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

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.

Numerous problems are solved by matrix multiplication, e.g. see [15] for computing minimal triangulations, [17] for finding different types of subgraphs as for example clique cutsets, and [7] 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 [24] 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 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 [22]. 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. We present a novel approach to fast computing these subsets by applying the distance product on the structure of dynamic programming instead of the graph itself.

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 [7] for an introduction to 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 [21] 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 [11, 4, 8, 12, 13], 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 [11] 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 [13] achieve small constants in the running time of a branch decomposition based exact algorithms for Planar Independent Set and Planar Dominating Set, namely $O(2^{3.182\sqrt{n}})$ and $O(2^{5.043\sqrt{n}})$, respectively. Dorn et al. [8] use the planar structure of sphere cut decompositions to obtain fast algorithms for problems like Planar Hamiltonian Cycle 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 introduce a 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 $O(m) = n^{O(1)}$.

To simplify matters, we first introduce our technique on the Independent Set 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 does influence 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 [11] 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 Independent Set 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 [10]. 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 [10]: 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? See Table 1 for a comparison of our results to [10].

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 cutpoint $\mathrm{tw} \leq 1.05\,\mathrm{bw}$ (compare with [10]). The lower part gives a summary of the most important improvements on exact and parameterized algorithms with parameter k. Note that we use the fastest matrix multiplication constant $\omega < 2.376$.

	Previous results	New results
Dominating Set	$O(n2^{\min\{2\,\mathrm{tw},2.38\,\mathrm{bw}\}})$	$O(n2^{2\,\mathrm{bw}})$
Independent Dominating Set	$O(n2^{\min\{2\,\mathrm{tw},2.38\,\mathrm{bw}\}})$	$O(n2^{2\,\mathrm{bw}})$
Perfect Code*	$O(n2^{\min\{2\,\mathrm{tw},2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw},2.09\,\mathrm{bw}\}})$
Perfect Dominating Set*	$O(n2^{\min\{2\,\mathrm{tw},2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw},2.09\,\mathrm{bw}\}})$
Maximum 2-Packing*	$O(n2^{\min\{2\,\mathrm{tw},2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw},2.09\,\mathrm{bw}\}})$
Total Dominating Set	$O(n2^{\min\{2.58\,\mathrm{tw},3\,\mathrm{bw}\}})$	$O(n2^{2.58\,\mathrm{bw}})$
Perfect Total Dominating Set	$O(n2^{\min\{2.58\,\mathrm{tw},3.16\,\mathrm{bw}\}})$	$O(n2^{2.58\,\mathrm{bw}})$
Planar Dominating Set	$O(2^{5.04\sqrt{n}})$	$O(2^{3.99\sqrt{n}})$
Planar Independent Set	$O(2^{3.18\sqrt{n}})$	$O(2^{2.52\sqrt{n}})$
Planar Hamiltonian Cycle	$O(2^{6.9\sqrt{n}})$	$O(2^{5.58\sqrt{n}})$
Planar Graph TSP	$O(2^{9.86\sqrt{n}})$	$O(2^{8.15\sqrt{n}})$
Planar connected Dominating Set	$O(2^{9.82\sqrt{n}})$	$O(2^{8.11\sqrt{n}})$
Planar Steiner Tree	$O(2^{8.49\sqrt{n}})$	$O(2^{7.16\sqrt{n}})$
Planar Feedback Vertex Set	$O(2^{9.26\sqrt{n}})$	$O(2^{7.56\sqrt{n}})$
Parameterized Planar Dominating Set	$O(2^{15.13\sqrt{k}}k + n^3)$	$O(2^{11.98\sqrt{k}}k + n^3)$
Parameterized Planar Longest Cycle	$O(2^{13.6\sqrt{k}}k + n^3)$	$O(2^{10.5\sqrt{k}}k + n^3)$

Table 1 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 Hamiltonian Cycle, 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 [8], [9], and [11], and [13].

2 Definitions

Branch decompositions. A *branch decomposition* $\langle T, \mu\rangle$ of a graph G is a ternary tree T with a bijection μ from $E(G)$ to the leaf-set $L(T)$. For every $e \in E(T)$ define *middle set* $\mathrm{mid}(e) \subseteq V(G)$ as follows: For every two leaves ℓ_1, ℓ_2 with vertex v adjacent to both $\mu^{-1}(\ell_1)$ and $\mu^{-1}(\ell_2)$, we have that $v \in \mathrm{mid}(e)$ for all edges e along the path from ℓ_1 to ℓ_2. The *width* bw of $\langle T, \mu\rangle$ is the maximum order of the middle sets over all edges of T, i.e., $\mathrm{bw}(\langle T, \mu\rangle) := \max\{|\,\mathrm{mid}(e)|\colon e \in T\}$. An optimal branch decomposition of G is defined by the tree T and the bijection μ which together provide the minimum width, the *branchwidth* $\mathrm{bw}(G)$.

Dynamic programming. For a graph G with $|V(G)| = n$ of bounded branchwidth bw the weighted Independent Set problem with positive node weights w_v for all $v \in V(G)$ can be solved in time $O(f(\mathrm{bw})) \cdot n^{O(1)}$ where $f(\cdot)$ is an exponential time function only dependent on bw. The algorithm is based on dynamic programming on a rooted branch decomposition $\langle T, \mu\rangle$ of G. The independent set is computed by processing T in post-order from the leaves to the root. For each middle set $\mathrm{mid}(e)$ an optimal independent set 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 towards 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 independent set $U \subseteq \mathrm{mid}(e)$, we denote by $\mathcal{V}_e(U)$ the maximum weight of an independent set 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 an independent set 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 maximum weight independent set is determined by taking the maximum 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 maximum-weight independent sets 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 independent sets $U_f \subseteq \mathrm{mid}(f)$ and $U_g \subseteq \mathrm{mid}(g)$ that are not necessarily optimal, the value $\mathcal{V}_e(U)$ is given as follows:

$$\begin{aligned}\mathcal{V}_e(U) = w(U) + \max \quad \{ \quad & \mathcal{V}_f(U_f) - w(U_f \cap U) + \mathcal{V}_g(U_g) - w(U_g \cap U) - w(U_f \cap U_g \setminus U) : \\ & 1) \quad U_f \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(f), \\ & 2) \quad U_g \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(g), \\ & 3) \quad U_f \cap \mathrm{mid}(g) = U_g \cap \mathrm{mid}(f)\}. \end{aligned} \tag{1}$$

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 $O(8^{\mathrm{bw}})$.

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 $\omega < 2.376$ [6].

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 [5] 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* or *min-plus product* of two $(n \times n)$-matrices B and C, denoted by $B \star C$, is an $(n \times n)$-matrix A such that

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

The distance product of two $(n \times n)$-matrices can be computed naively in time $O(n^3)$. Yuval [25] 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 the $O(m \cdot n^\omega)$.

For distance product of two $(n \times p)$- and $(p \times n)$-matrices with $p > n$ we get $O(p \cdot (m \cdot n^{\omega-1}))$ 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

$$a_{ij} = \bigvee_{1 \leq k \leq n} \{b_{ik} \wedge c_{kj}\}, 1 \leq i, j \leq n. \tag{3}$$

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

3 Dynamic programming & distance product

In this section, we will continue our Independent Set example and oppose two techniques on how to obtain faster dynamic programming approaches. The known first technique uses tables in order to decrease the number of times a subset is queried. As a second approach, we introduce a technique using matrices that allows to highly make use of the structure of branch decompositions and of the fast matrix multiplication.

Tables. We will see now a more sophisticated approach that 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 (1) 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:

$$\begin{aligned} \mathcal{V}_e(U) = \max \quad \{ \quad & \mathcal{V}_f(U_f) + \mathcal{V}_g(U_g) - w(U_f \cap U_g) : \\ & 1') \quad U_f \cap (I \cup L) = U \cap (I \cup L), \\ & 2') \quad U_g \cap (I \cup R) = U \cap (I \cup R), \\ & 3') \quad U_f \cap (I \cup F) = U_g \cap (I \cup F)\}. \end{aligned} \tag{4}$$

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 in (4). 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.

Matrices. In the remaining section we show how to use matrices instead of tables as data structure for dynamic programming. Then apply the distance product of two matrices to compute the values $\mathcal{V}(U)$.

Reformulating the constraints 1')-3') in the computation of $\mathcal{V}_e(U)$ in (4), we obtain:

$$\begin{aligned} & 0'') \quad U \cap I = U_f \cap I = U_g \cap I, \\ & 1'') \quad U_f \cap L = U \cap L, \\ & 2'') \quad U_g \cap R = U \cap R, \\ & 3'') \quad U_f \cap F = U_g \cap F. \end{aligned} \tag{5}$$

With the new constraint $0''$) one may observe that every independent set S_e of G_e is determined by the independent sets S_f and S_g such that all three sets intersect in some subset $U^I \subseteq I$. The idea is 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 independent sets 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 independent.

Lemma 1. *Given an independent set $U^I \subseteq I$. For all independent sets $U \subseteq \text{mid}(e)$, $U_f \subseteq \text{mid}(f)$ 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. In order to apply the min-plus product of (2), we change the signs of each entry in B and C since we deal with a maximization rather than a minimization problem. Note that we set all entries wit value '∞' to $\sum_{v \in V(G)} w_v + 1$. Another difference between (2) and (4) is the additional term $w(U_f \cap U_g)$. 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:

$$\begin{aligned} \mathcal{V}_e(U) = \min \quad \{ \quad & -\mathcal{V}_f(U_f) - (\mathcal{V}_g(U_g) - |U^I| - |U^F|) : \\ & 0'') \quad U \cap I = U_f \cap I = U_g \cap I = U^I, \\ & 1'') \quad U_f \cap L = U \cap L = U^L, \\ & 2'') \quad U_g \cap R = U \cap R = U^R, \\ & 3'') \quad U_f \cap F = U_g \cap F = U^F\}. \end{aligned} \tag{6}$$

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 2. *Dynamic programming for the* Independent Set *problem on weights $O(m) = n^{O(1)}$ on graphs of branchwidth* bw *takes time $O(m \cdot 2^{\frac{\omega}{2} \cdot \text{bw}})$ 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 $O(m \cdot 2^{(\omega-1)|L|} \cdot 2^{|F|} \cdot 2^{|I|})$.

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: $O(m \cdot \frac{2^{|F|}}{2^{|L|}} \cdot 2^{\omega|L|})$. If $2^{|F|} < 2^{|L|}$ we simply get $(m \cdot 2^{1.85|L|} \cdot 2^{0.54|F|})$ (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 $O(m \cdot 2^{(\omega-1)|L|} \cdot 2^{|F|} \cdot 2^{|I|})$ subject to these constraints, we get the claimed running time of $O(m \cdot 2^{\frac{\omega}{2} \cdot \text{bw}})$. □

4 A general technique

In this section 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 Independent Set we have that the vertices in the graph have two states relating to an independent set 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 $\langle T, \mu \rangle$ 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 $S_e \in \lambda^{|\text{mid}(e)|}$. For a further details, please consult for example [10].

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 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 Independent Set 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 [10]. 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 $c_1 \leq c_2, c_3$.

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 $S_e(I) \in \lambda^I$. Recall the definition of value $\mathcal{V}_e$.

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 $S_e = S_e(I)\|S_e(L)\|S_e(R)$.

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 $S_f(L)$ and $S_g(R)$ 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 $a_{ij} = \min_{k=1}^{c_2^{|F|}}\{b_{ik}+c_{kj}\}, 1 \leq i \leq c_1^{|L|}, 1 \leq j \leq c_1^{|R|}$. Hence A is the distance product of B and C. □

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

Theorem 4. *Dynamic programming for solving vertex-state problems on weights $O(m)$ 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 $c_2^{|F|} \geq c_1^{|L|}$ 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.

From Section 3, we know that 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. □

5 Application of the new technique

In this section, we show how one can apply the technique for several optimization problems such as Dominating Set and its variants in order to obtain fast algorithms. We also apply our technique to planar graph problems. The branchwidth of a planar graph is bounded by $2.122\sqrt{n}$. There exist optimal branch decompositions whose middle sets are closed Jordan curves in the planar graph embedding [8]. Such a *sphere cut decomposition* has the property that the I-set is of size at most 2, that is, the running time stated in Theorem 4 has no part 'c_3^{bw}'.

For Dominating Set we have that $c_1 \equiv c_2 = 3$ and $c_3 = 4$. The former running time was $O(3^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$. Since we have that $O(m \cdot \max\{3^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} 3^{\frac{\mathrm{bw}}{2}}, 3^{\mathrm{bw}}, 4^{\mathrm{bw}}\}) = O(m \cdot 4^{\mathrm{bw}})$ for node weights $O(m)$, an exponent of the fast matrix multiplication does not come into play at all. Note that the algorithm is better then the fastest known treewidth tw based algorithm of $O(4^{\mathrm{tw}}) \cdot n^{O(1)}$ due to the natural bound $\mathrm{bw} \leq \mathrm{tw} + 1$.

Sphere cut decompositions of planar graphs can be computed in time $O(n^3)$ by an improvement of the famous rat catcher method ([19] and [14]). With the nice property that $|I| \leq 2$ for all middle sets, we achieve a running time in terms of $O(m \cdot \max\{c_1^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} c_2^{\frac{\mathrm{bw}}{2}}, c_2^{\mathrm{bw}}\})$ for planar graph problems. Thus, we improve for Planar Dominating Set with node weights $O(m)$ the $O(4^{\mathrm{bw}}) \cdot n^{O(1)}$ to $O(m \cdot 3^{1.188\,\mathrm{bw}}) = O(m \cdot 3.688^{\mathrm{bw}})$ that is strictly better than the actual runtime of the treewidth based technique of $O(4^{\mathrm{tw}}) \cdot n^{O(1)}$.

As one may read from Table 1 in the introduction, Theorem 4 directly gives the runtime to our technique applied to variants of the Dominating Set problem that are introduced for branch decomposition in [10].

For Planar Hamiltonian Cycle, 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 [8] 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. This means that this step does not increase the runtime of our algorithm that is improved to $O(m \cdot 2^{1.106\omega\,\mathrm{bw}})$ by applying boolean matrix multiplication. With the same trick we improve the runtimes of all problems discussed in [8]. Please refer to Table 1 in the introduction.

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 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 behaviour 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 equals the number of vertex states, which is the natural lower bound for dynamic programming.

Currently, [23] 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 [10]), fast matrix multiplication does not affect the theoretical worst case behaviour—though practically it might be of some use.

For the planar case, an interesting question arises: can we change the structure of tree decompositions to attack its inferiority against sphere cut decompositions?

Finally, is there anything to win for dynamic programming if we use 3-dimensional matrices as data structure? That is, if we have the third dimension labeled with $S_e(I)$?

7 Acknowledgments

Many thanks to Fedor Fomin for his useful comments and his patience, Artem Pyatkin for some fruitful discussions, 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 LATIN'02: Theoretical informatics (Cancun), vol. 2286 of Lecture Notes in Computer Science, Berlin, 2002, Springer, 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] W. Cook and P. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, 15 (2003), pp. 233–248.

[5] D. Coppersmith, *Rectangular matrix multiplication revisited*, Journal of Complexity, 13 (1997), pp. 42–49.

[6] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation, 9 (1990), pp. 251–280.

[7] 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.

[8] 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, Berlin, 2005, pp. 95–106.

[9] ———, *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.

[10] F. Dorn and J. A. Telle, *Two birds with one stone: the best of branchwidth and treewidth with one algorithm*, in LATIN'06: 7th Latin American Theoretical Informatics Symposium (Valdivia), vol. 3887 of Lecture Notes in Computer Science, Berlin, 2006, Springer, pp. 386–397.

[11] F. V. Fomin and D. M. Thilikos, *Dominating sets in planar graphs: branch-width and exponential speed-up*, in SODA'03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003), New York, 2003, ACM, pp. 168–177.

[12] 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.

[13] ———, *A simple and fast approach for solving problems on planar graphs*, in Proceedings of the 21st International Symposium on Theoretical Aspects of Computer Science (STACS 2004), vol. 2996 of LNCS, Springer, Berlin, 2004, pp. 56–67.

[14] 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.

[15] 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.

[16] A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM Journal on Computing, 7 (1978), pp. 413–423.

[17] D. Kratsch and J. Spinrad, *Between $O(nm)$ and $O(n^\alpha)$*, in SODA'03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003), New York, 2003, ACM, pp. 158–167.

[18] R. Seidel, *On the all-pairs-shortest-path problem in unweighted undirected graphs*, Journal of Computer and System Sciences, 51 (1995), pp. 400–403.

[19] P. D. Seymour and R. Thomas, *Call routing and the ratcatcher*, Combinatorica, 14 (1994), pp. 217–241.

[20] A. Shoshan and U. Zwick, *All pairs shortest paths in undirected graphs with integer weights*, in 40th Annual Symposium on Foundations of Computer Science, (FOCS '99), Lecture Notes in Computer Science, Springer, 1999, pp. 605–615.

[21] J. A. Telle and A. Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM J. Discrete Math, 10 (1997), pp. 529–550.

[22] P. M. Vaidya, *Speeding-up linear programming using fast matrix multiplication*, in 30th Annual Symposium on Foundations of Computer Science (FOCS 1989), 1989, pp. 332–337.

[23] V. Vassilevska and R. Williams, *Finding a maximum weight triangle in $n^{(3-\delta)}$ time, with applications*, 2006. To appear in ACM Symposium on Theory of Computing (STOC 2006), http://www.cs.cmu.edu/ ryanw/max-weight-triangle.pdf.

[24] R. Williams, *A new algorithm for optimal constraint satisfaction and its implications*, in Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), vol. 3142 of LNCS, Springer, Berlin, 2004, pp. 1227–1237.

[25] G. Yuval, *An algorithm for finding all shortest paths using $N^{(2.81)}$ infinite-precision multiplications*, Information Processing Letters, 4 (1976), pp. 155–156.

[26] U. Zwick, *All pairs shortest paths using bridging sets and rectangular matrix multiplication*, Journal of the ACM, 49 (2002), pp. 289–317.