

Subcubic Equivalences Between Path, Matrix, and Triangle Problems*

Virginia Vassilevska Williams[†] Ryan Williams[‡]

Abstract

We say an algorithm on $n \times n$ matrices with integer entries in $[-M, M]$ (or n -node graphs with edge weights from $[-M, M]$) is *truly subcubic* if it runs in $O(n^{3-\delta} \cdot \text{poly}(\log M))$ time for some $\delta > 0$. We define a notion of *subcubic reducibility*, and show that many important problems on graphs and matrices solvable in $O(n^3)$ time are *equivalent* under subcubic reductions. Namely, the following weighted problems either *all* have truly subcubic algorithms, or none of them do:

- The all-pairs shortest paths problem on weighted digraphs (APSP).
- Detecting if a weighted graph has a triangle of negative total edge weight.
- Listing up to $n^{2.99}$ negative triangles in an edge-weighted graph.
- Finding a minimum weight cycle in a graph of non-negative edge weights.
- The replacement paths problem on weighted digraphs.
- Finding the second shortest simple path between two nodes in a weighted digraph.
- Checking whether a given matrix defines a metric.
- Verifying the correctness of a matrix product over the $(\min, +)$ -semiring.
- Finding a maximum subarray in a given matrix.

Therefore, if APSP cannot be solved in $n^{3-\varepsilon}$ time for any $\varepsilon > 0$, then many other problems also need essentially cubic time. In fact we show generic equivalences between matrix products over a large class of algebraic structures used in optimization, verifying a matrix product over the same structure, and corresponding triangle detection problems over the structure. These equivalences simplify prior work on subcubic algorithms for all-pairs path problems, since it now suffices to give appropriate subcubic triangle detection algorithms.

Other consequences of our work are new combinatorial approaches to Boolean matrix multiplication over the (OR,AND)-semiring (abbreviated as BMM). We show that practical advances in triangle detection would imply practical BMM algorithms, among other results. Building on our techniques, we give two new BMM algorithms: a derandomization of the recent combinatorial BMM algorithm of Bansal and Williams (FOCS'09), and an improved quantum algorithm for BMM.

*This work originated while the authors were members of the Institute for Advanced Study, Princeton, NJ and visiting the Computer Science Department at Princeton University. While the paper was written, the first author spent time as a postdoctoral fellow at UC Berkeley and Stanford, and as an assistant professor at Stanford supported by NSF Grants CCF-1417238 and CCF-1514339. At Berkeley, she was supported by a CRA Computing Innovations Fellowship and by NSF Grants CCF-0830797 and CCF-1118083; as a postdoc at Stanford she was supported by NSF Grants IIS-0963478 and IIS-0904325, and an AFOSR MURI Grant. While the paper was written, the second author spent time as a postdoctoral fellow at the IBM Almaden Research Center supported by the Josef Raviv Memorial Fellowship, and as an assistant professor at Stanford supported by NSF Grant CCF-1212372.

[†]EECS CSAIL, MIT. Supported by NSF Grants CCF-1417238 and CCF-1514339. Email: virgi@mit.edu

[‡]EECS CSAIL, MIT. Supported by NSF Grant CCF-1212372. Email: rrw@mit.edu

1 Introduction

Many computational problems on graphs and matrices have natural cubic time solutions. For example, $n \times n$ matrix multiplication over any algebraic structure can be done in $O(n^3)$ operations. For algebraic structures that arise in optimization, such as the $(\min, +)$ -semiring, it is of interest to determine when we need only a subcubic number of operations.¹ The all-pairs shortest paths problem (APSP) also has an $O(n^3)$ time algorithm on n -node graphs, known for over 40 years [29, 69]. One of the “Holy Grails” of graph algorithms is to determine whether this cubic complexity is basically inherent, or whether a significant improvement (say, $O(n^{2.9})$ time) is possible. (It is known that this question is equivalent to finding a faster algorithm for $(\min, +)$ matrix multiplication [28, 47]). Many researchers believe that cubic time is essentially necessary: there are n^2 pairs of nodes, and in the worst case we should not expect to improve too much on $O(n)$ time per pair.² (We should note that a long line of work has produced subcubic algorithms with small $\text{poly}(\log n)$ improvements in the running time. This work recently culminated in $n^3/2^{\Theta(\log^{1/2} n)}$ time algorithms [71, 18] that “shave” all polylogarithms, unfortunately without giving a truly subcubic solution.)

Related to APSP is the *replacement paths problem* (RPP): given nodes s and t in a weighted directed graph and a shortest path P from s to t , compute the length of the shortest simple path that avoids edge e , for all edges e on P . This problem is studied extensively (cf. [74, 40, 27, 39, 55, 52, 10]) for its applications to network reliability. A slightly subcubic time algorithm is not hard to obtain from a slightly subcubic APSP algorithm, but nothing faster than this is known. It does seem that cubic time may be inherent, since for all edges in a path (and there may be $\Omega(n)$ of them) we need to recompute a shortest path. A well-studied restriction of RPP is to find the *second* shortest (simple) path between two given nodes s and t . This problem also has an $O(n^3)$ time algorithm, but again nothing much faster is known. Here, the cubic complexity does not seem to be so unavoidable: we simply want to find a certain type of path between two endpoints. Similarly, finding a minimum weight cycle in a graph with non-negative weights is only known to be possible in slightly subcubic time.³

An even simpler example is that of finding a triangle in an edge-weighted graph where the sum of edge weights is negative. Exhaustive search of all triples of nodes takes about $O(n^3)$ time, and applying the best APSP algorithm makes this $n^{3-o(1)}$ time, but we do not know a truly subcubic algorithm. Recent work has suggested that this negative triangle problem might have a faster algorithm, since the *node-weighted* version of the problem can be solved faster [63, 64, 67, 20, 21]. (In fact the node-weighted version of the problem is no harder than the *unweighted* triangle detection problem, which is solvable in $O(n^{2.38})$ time [36].) Since the cubic algorithm for negative triangle is *so* simple, and many restrictions of the problem have faster algorithms, it would appear that cubic complexity is unnecessary for finding a negative triangle.

We give theoretical evidence that these open algorithmic questions may be hard to resolve, by showing that they and other well-studied problems are all surprisingly *equivalent*, in the sense that there is a substantially subcubic algorithm for one of them if and only if all of them have substantially subcubic algorithms. Compare with the phenomenon of NP-completeness: one reason P versus NP looks so hard to resolve is that many researchers working in different fields have all been working on essentially the *same* (NP-complete) problem,

¹Note that in the specific case when the structure is a *ring*, it is well known that one can solve the problem much faster than $O(n^3)$ operations [59, 19]. However it is unknown if this fact can be used to compute the matrix product fast on many other important structures such as commutative semirings.

²Of course, as noted by a referee, a similar heuristic argument would say that $n \times n$ matrix multiplication requires $\Omega(n^3)$ time, which has long known to be false. One should be careful with intuition.

³Note that if we allowed negative weights, this problem is NP-hard.

with no concrete resolution of the problem in sight. Our situation is entirely analogous: either these problems really need essentially cubic time, or we are missing a fundamental insight which would make all of them simultaneously easier.

Informally, we say that an algorithm on $n \times n$ matrices (or n -node graphs) with entries in $\{-M, \dots, M\} \cup \{-\infty, \infty\}$ is *truly subcubic* if it uses $O(n^{3-\delta} \cdot \text{poly}(\log M))$ time for some $\delta > 0$. In general, $\text{poly} \log M$ factors are natural: truly subcubic ring matrix multiplication algorithms (such as Strassen's) have $\text{poly} \log M$ overhead if one counts the bit complexity of operations. We develop *subcubic reductions* between many problems solvable in $\tilde{O}(n^3)$ time, proving Theorem 1.1 below.

Theorem 1.1 *The following problems (with weights in $\{-M, \dots, M\} \cup \{-\infty, \infty\}$) either all have truly subcubic algorithms, or none of them do:*

1. *The all-pairs shortest paths problem on weighted digraphs (APSP).*
2. *The all-pairs shortest paths problem on undirected weighted graphs.*
3. *Detecting if a weighted graph has a triangle of negative total edge weight.*
4. *Listing up to $n^{3-\delta}$ negative triangles in an edge-weighted graph, for a fixed $\delta > 0$.*
5. *Computing the matrix product over the $(\min, +)$ -semiring.*
6. *Verifying the correctness of a matrix product over the $(\min, +)$ -semiring.*
7. *Checking whether a given matrix defines a metric.*
8. *Finding a minimum weight cycle in a graph of non-negative edge weights.*
9. *The replacement paths problem on weighted digraphs.*
10. *Finding the second shortest simple path between two nodes in a weighted digraph.*
11. *Finding a maximum subarray of a given matrix.*

Note the only previously known equivalence in the above was that of (1),(2) and (5).

Out of the above reductions, only the reduction from (1) to (3) actually introduces a $\text{poly}(\log M)$ factor in the running time. We prove that there is in fact a randomized subcubic reduction from (1) to (3) that replaces the $\text{poly}(\log M)$ factor with a $\log n$ factor and works with high probability (cf. Section 4.3). Therefore, any $O(n^{3-\delta})$ -time algorithm for one of the above problems can be converted into an $O(n^{3-\delta'})$ -time (randomized) algorithm for any of the other problems.

An explicit definition of our reducibility concept is given in Section 3. The truly subcubic runtimes may vary depending on the problem: for example, an $\tilde{O}(n^{2.9})$ algorithm for negative triangle implies an $\tilde{O}(n^{2.96})$ algorithm for APSP.

Perhaps the most interesting aspect of Theorem 1.1 is that some of the listed problems are *decision* problems, while others are *functions*. Hence to prove lower bounds for these decision problems, it would suffice to prove them for certain multi-output functions. It is counterintuitive that an $O(n^{2.9})$ algorithm returning one

bit can be used to compute a function returning n^2 bits, in $O(n^{2.96})$ time. Nevertheless, it is possible and in retrospect, our reductions are very natural.

Subsequently to the conference version of this paper, some more subcubic equivalences to APSP have been proven. Notably, Abboud, Grandoni and Vassilevska Williams [1] showed that computing the Radius, Median or (Unique Path) Betweenness Centrality of a graph are all subcubically equivalent to APSP.

A few equivalences in Theorem 1.1 follow from a more general theorem, which can be used to simplify prior work on all-pairs path problems.

In general we consider (\min, \odot) structures defined over a set R of the form $\{-M, \dots, M\} \cup \{-\infty, \infty\}$ for some $M \in \mathbb{Z}^+ \cup \{\infty\}$, together with an operation $\odot : R \times R \rightarrow R$.⁴ We define a type of (\min, \odot) structure that we call *extended*, which allows for an “identity matrix” and an “all-zeroes matrix” over the structure. (For definitions, see the Preliminaries.) Almost all structures we consider in this paper are extended, including the Boolean semiring over OR and AND, the (\min, \max) -semiring, and the $(\min, +)$ -semiring. In Section 4 we prove:

Theorem 1.2 (Informal Statement of Theorems 4.1 and 4.2) *Let \bar{R} be an extended (\min, \odot) structure. The following problems over \bar{R} either all have truly subcubic algorithms, or none of them do:*

- **Negative Triangle Detection.** *Given an n -node graph with weight function $w : V \times V \rightarrow R$, find nodes i, j, k such that $w(i, j) \in R$, $w(i, k) \in R$, $w(k, j) \in R$, and $(w(i, k) \odot w(k, j)) + w(i, j) < 0$.*
- **Matrix Product.** *Given two $n \times n$ matrices A, B with entries from R , compute the product of A and B over \bar{R} .*
- **Matrix Product Verification.** *Given three $n \times n$ matrices A, B, C with entries from R , determine if the product of A and B over \bar{R} is C .*

The relationship between matrix product and its verification is particularly surprising, as $n \times n$ matrix product verification *over rings* can be done in $O(n^2)$ randomized time [11, 31] but it is not known whether ring matrix multiplication can be reduced to this fast verification. Spinrad [57] (Open Problem 8.2) and Alon [2] asked if the verification of various matrix products can be done faster than the products themselves. Our reductions rely crucially on the fact that the addition operation in a (\min, \odot) structure is a *minimum*.

We have as a consequence of Theorem 1.2:

Theorem 1.3 *The following all have truly subcubic “combinatorial” algorithms, or none of them do:*

- *Boolean matrix multiplication (BMM).*
- *Detecting if a graph has a triangle.*
- *Listing up to $n^{3-\delta}$ triangles in a graph for constant $\delta > 0$.*
- *Verifying the correctness of a matrix product over the Boolean semiring.*

⁴An analogous treatment is possible for (\max, \odot) structures. We omit the details, as they merely involve negations of entries.

The notion of a “combinatorial” algorithm does not have a formal definition. Intuitively, such algorithms are not only theoretically, but also practically efficient. In the above theorem, by combinatorial we mean an algorithm with low leading constants. One can verify that the reductions in our paper have low leading constants and low overhead; hence any simple fast triangle algorithm would yield a simple (and only slightly slower) BMM algorithm. The relation between BMM and the triangle problem has been investigated by many researchers, *e.g.* ([73], Open Problem 4.3(c)) and ([57], Open Problem 8.1).

An extra bullet can be added to Theorem 1.3 relating the above problems to context-free grammar (CFG) parsing. The CFG parsing problem is to determine whether a given n -symbol string can be generated by a given CFG of size g . Valiant [61] showed that the problem can be reduced to BMM, showing that any $O(n^{3-\varepsilon})$ time algorithm for BMM implies an $O(gn^{3-\varepsilon})$ time algorithm for CFG parsing. Lee [42] showed a converse: that any $O(gn^{3-\varepsilon})$ time algorithm for CFG parsing would imply an $O(n^{3-\varepsilon/3})$ time algorithm for $n \times n$ BMM. The reductions in [61] and [42] are combinatorial algorithms. In this sense, CFG parsing for constant size grammars can be added to Theorem 1.3.

Using Theorem 1.3, we also show how our techniques can be used to design alternative approaches to Boolean matrix multiplication (BMM). More concretely, Theorem 1.3 can already yield new BMM algorithms, with a little extra work. First, we can derandomize the recent combinatorial BMM algorithm of Bansal and Williams [6]:

Theorem 1.4 *There is a deterministic combinatorial $O(n^3 / \log^{2.25} n)$ -time algorithm for BMM.*

Subsequent to our result, there have been two improved combinatorial BMM algorithms, by Chan [17] and by Yu [75]. The latter constitutes the current fastest algorithm, running in $O(n^3 / \log^4 n)$ time. Yu’s improvement is partially based on Theorem 6.1 presented here.

The BMM algorithm of [6] uses randomness in two different ways: it reduces BMM to a graph theoretic problem, computes a pseudoregular partition of the graph in randomized quadratic time, then it uses random samples of nodes along with the partition to speed up the solution of the graph problem. We can avoid the random sampling by giving a triangle algorithm with $O(n^3 / \log^{2.25} n)$ running time, and applying a stronger version of Theorem 1.3 that shows an equivalence preserving polylogarithmic factors (see Corollary 4.1). To get a deterministic triangle algorithm, we show (using a new reduction) that in fact any *polynomial time* algorithm for computing a pseudoregular partition suffices for obtaining a subcubic triangle algorithm. With this relaxed condition, we can replace the randomized quadratic algorithm for pseudoregularity with a deterministic polynomial time algorithm of Alon and Naor [4]. A similar result holds for APSP: assuming that APSP requires essentially cubic time, we obtain essentially quadratic time lower bounds for a natural weighted graph query problem, for any polynomial amount of processing on the graph.

In more detail, for a graph with edge weight function $c : E \rightarrow \mathbb{Z}$, a *price query* is an assignment of node weights $p : V \rightarrow \mathbb{Z}$. Such a query has a *yes* answer if and only if there is a $(u, v) \in E$ such that $p(u) + p(v) > c(u, v)$. (Intuitively, the $p(v)$ are “prices” on the nodes, the $c(u, v)$ are costs of producing u and v , and a price query asks if there is an edge we are willing to “sell” at the prices given by the query.)

Theorem 1.5 *Suppose there is a constant $k > 0$ and a function $f(n)$ such that every n -node edge-weighted graph can be preprocessed in $O(n^k)$ time so that any subsequent price query can be answered in $O(n^2 / f(n))$ time. Then, the negative triangle detection problem is solvable in $O(n^3 / f(n^{1/(2k)}))$ time.*

In particular, if the hypothesis above is true for $f(n) = 2^{\Omega(\log^\delta n)}$ for some constant $\delta > 0$, then negative triangle is in $n^3 / 2^{\Omega(\log^\delta n)}$ time. Our reduction from APSP to negative triangle is tight in this case and the

hypothesis would imply that APSP also is in $n^3/2^{\Omega(\log^\delta n)}$ time. If $\delta = 1$, then APSP is in truly subcubic time. Subsequent work [71] has in fact shown that the statement is true for $\delta = 1/2$, i.e. that APSP is in $n^3/2^{\Omega(\sqrt{\log n})}$ time.

We also obtain a new quantum algorithm for BMM in the query complexity setting, improving the previous best by Buhrman and Špalek [13]:

Theorem 1.6 *There is an $\tilde{O}(\min\{n^{1.3}L^{17/30}, n^{1.5}L^{1/4}\})$ -query quantum algorithm for computing the product of two $n \times n$ Boolean matrices, where L is the number of ones in the output matrix.*

The first time bound of Theorem 1.6 is obtained by simply applying the best known quantum algorithm for triangle [45] to our generic matrix product to triangle detection reduction, already improving the previous best [13] output-sensitive quantum algorithm for BMM. The second time bound is obtained by applying ideas of Lingas [43, 44]. Our results have recently been improved by Le Gall [41] and Jeffery, Kothari and Magniez [37].

1.1 A Little Intuition

One of our key observations is the counterintuitive result that subcubic algorithms for certain triangle detection problems can be used to obtain subcubic matrix products in many forms, including products that are not known to be subcubic. Let us first review some intuition for why fast triangle detection should *not* imply fast matrix multiplication, then discuss how our approach circumvents it. For simplicity, let us focus on the case of Boolean matrix multiplication (BMM) over OR and AND.

First, note that triangle detection returns one bit, while BMM returns n^2 bits. This seems to indicate that $O(n^{2.99})$ triangle detection would be useless for subcubic BMM, as the algorithm would need to be run $\Omega(n^2)$ times. Furthermore, BMM can determine *for all edges* if there is a triangle using the edge, while triangle detection only determines if *some edge* is in a triangle. Given our intuitions about quantifiers, it looks unlikely that the universally quantified problem could be efficiently reduced to the existentially quantified problem. So there appears to be strong intuition for why such a reduction would not be possible.

However, there is an advantage in calling triangle detection on small graphs corresponding to small submatrices. Let A and B be $n \times n$ matrices over $\{0, 1\}$. Triangle detection can tell us if $A \cdot B$ contains any entry with a 1: Set up a tripartite graph with parts S_1, S_2 and S_3 , each containing n nodes which we identify with the set $[n] := \{1, \dots, n\}$. The edge relation for $S_1 \times S_2$ is defined by A , and the edge relation for $S_2 \times S_3$ is defined by B (in the natural way). A path of length two from $i \in S_1$ to $j \in S_3$ corresponds to a 1 in the entry $(A \cdot B)[i, j]$. Therefore, putting all possible edges between S_1 and S_3 , there is a triangle in this graph if and only if $A \cdot B$ contains a 1-entry. (Note we are already relying on the fact that our addition operation is OR.)

The above reasoning can also be applied to *submatrices* A' and B' , to determine if $A' \cdot B'$ contributes a 1-entry to the matrix product. More generally, triangle detection can tell us if a product of two submatrices contains a 1-entry, *among just those entries of the product that we have not already computed*. That is, we only need to include edges between S_1 and S_3 that correspond to undetermined entries of the product. Hence triangle detection can tell us if submatrices A' and B' have any new 1-entries to contribute to the current matrix product so far.

On the one hand, if all possible pairs of submatrices from A and B do not result in finding a triangle, then we have computed all the 1-entries and the rest must be zeroes. On the other hand, when we detect a triangle, we determine at least one new 1-entry (i, j) in $A \cdot B$, and we can keep latter triangle detection calls from

recomputing this entry by simply removing the edge (i, j) between S_1 and S_3 . By balancing the number of triangle detection subproblems we generate with the number of 1-entries in $A \cdot B$, we get a subcubic runtime for matrix multiplication provided that the triangle algorithm was also subcubic. (In fact we get an *output sensitive* algorithm.) With additional technical effort and a simultaneous binary search method, these ideas can be generalized to any matrix product where “addition” is a minimum operator.

2 Preliminaries

Throughout the paper, for two integers a and b with $a < b$, we denote by $[a, b]$ the interval of integers between a and b , i.e. $\{a, a + 1, \dots, b\}$.

Unless otherwise noted, all graphs in the paper are directed, and have n vertices and m edges. Whenever an algorithm in our paper uses ∞ or $-\infty$, these can be substituted by numbers of suitably large absolute value. We use ω to denote the smallest real number such that $n \times n$ matrix multiplication over an arbitrary ring can be done in $n^{\omega+o(1)}$ additions and multiplications over the ring.

Structures and Extended Structures. We give a general definition encompassing all algebraic structures for which our results apply. Let R be a set of the form $[-M, M] \cup \{-\infty, \infty\}$, where $M \in \mathbb{Z}^+ \cup \{\infty\}$. We call M the *maxint* of R .

A (\min, \odot) structure over R is defined by a binary operation $\odot : R \times R \rightarrow R$. We use the variable \mathcal{R} to refer to a (\min, \odot) structure. We say a (\min, \odot) structure is *extended* if R contains elements ε_0 and ε_1 such that for all $x \in R$, $x \odot \varepsilon_0 = \varepsilon_0 \odot x = \infty$ and $\varepsilon_1 \odot x = x$ for all $x \in R$. That is, R is extended to include two additional elements: ε_0 which is a type of annihilator, and ε_1 which is a left identity. We use the variable $\bar{\mathcal{R}}$ to refer to an extended structure.

The elements ε_0 and ε_1 allow us to define (for every n) an $n \times n$ identity matrix I_n and a $n \times n$ zero matrix Z_n over $\bar{\mathcal{R}}$. More precisely, $I_n[i, j] = \varepsilon_0$ for all $i \neq j$, $I_n[i, i] = \varepsilon_1$, and $Z_n[i, j] = \varepsilon_0$ for all i, j . We shall omit the subscripts of I_n and Z_n when the dimension is clear.

Examples of extended structures $\bar{\mathcal{R}}$ are the (OR, AND) (or Boolean) semiring,⁵ as well as the (\min, \max) and $(\min, +)$ semirings⁶ (also called *subtropical* and *tropical*), and the (\min, \leq) structure used to solve *all pairs earliest arrivals* [62, 68]. An example of a structure that is *not* extended is the “existence dominance” structure defined with $R = \mathbb{Z} \cup \{-\infty, \infty\}$, and $a \odot b = 0$ if $a \leq b$ and $a \odot b = 1$ otherwise.

Matrix Products Over Structures. The matrix product of two $n \times n$ matrices over \mathcal{R} is

$$(A \odot B)[i, j] = \min_{k \in [n]} (A[i, k] \odot B[k, j]).$$

It is easy to verify that for all matrices A over an extended $\bar{\mathcal{R}}$, $I \odot A = A$ and $Z \odot A = A \odot Z = F$ where $F[i, j] = \infty$ for all i, j . The problem of *matrix product verification* over an extended structure $\bar{\mathcal{R}}$ is to determine whether $\min_{k \in [n]} (A[i, k] \odot B[k, j]) = C[i, j]$ for all $i, j \in [n]$, where A, B, C are given $n \times n$ matrices with entries from \bar{R} . Although it looks like a simpler problem, matrix product verification for the $(\min, +)$ semiring (for instance) is not known to have a truly subcubic algorithm.

⁵Observe the Boolean semiring is easily embedded into the R -structure with $R = [0, 0] \cup \{-\infty, \infty\}$, $\varepsilon_0 = \infty$ and $\varepsilon_1 = 0$, where $x \odot y = x + y$.

⁶For the $(\min, +)$ semiring when R is taken to be $[-M, M] \cup \{-\infty, \infty\}$ for a finite integer M , we replace $+$ with the \odot operation where $a \odot b = a + b$ when $a + b \in [-M, M]$, $a \odot b = -\infty$ if $a + b < -M$ and $a \odot b = \infty$ otherwise.

Negative Triangles Over Structures. The *negative triangle problem over \mathcal{R}* is defined on a weighted tripartite graph with parts I, J, K . All edge weights are from R . The problem is to detect if there are $i \in I, j \in J, k \in K$ so that $(w(i, k) \odot w(k, j)) + w(i, j) < 0$. Note that if one negates all weights of edges between I and J , the condition becomes $(w(i, k) \odot w(k, j)) < w(i, j)$, and since R is symmetric, all edge weights are still from R . Thus the negative triangle problem can be equivalently defined with the condition $(w(i, k) \odot w(k, j)) < w(i, j)$.

In the special case when $\odot = +$, the tripartiteness requirement is unnecessary, and the negative triangle problem is defined on an *arbitrary* graph with edge weights from R . This holds for the negative triangle problem over both the $(\min, +)$ and Boolean semirings.

2.1 Prior and Related Work

Matrix Products and Path Problems. Matrix multiplication is fundamental to computer science. The case of multiplying over a field is well known to admit surprisingly fast algorithms using the magic of subtraction, beginning with the famous $O(n^{\log_2 7})$ time algorithm of Strassen [59]⁷. After many improvements on Strassen’s original result, the current best upper bound on matrix multiplication over an arbitrary field is by Le Gall [34], very slightly improving upon the looser bound $O(n^{2.373})$ [72] which together with an independent result by Stothers [58, 22] improved on the longstanding bound of $O(n^{2.376})$ of Coppersmith and Winograd [19].

Over algebraic structures without subtraction, there has been little progress in the search for truly subcubic algorithms. These “exotic” matrix products are extremely useful in graph algorithms and optimization. For example, matrix multiplication over the (\max, \min) -semiring, with \max and \min operators in place of plus and times (respectively), can be used to solve the *all pairs bottleneck paths problem* (APBP) on arbitrary weighted graphs, where we wish to find a maximum capacity path from s to t for all pairs of nodes s and t . Related work [66, 24] has shown that fast matrix multiplication over rings can be applied to obtain a truly subcubic algorithm over the (\max, \min) -semiring, yielding truly subcubic APBP. Matrix multiplication over the $(\min, +)$ -semiring (also known as the *distance product*) can be used to solve *all pairs shortest paths* (APSP) in arbitrary weighted graphs [28]. That is, truly subcubic distance product would imply truly subcubic APSP, one of the “Holy Grails” of graph algorithms. The fastest known algorithms for distance product are a recent randomized $n^3/2^{\Omega(\log^{1/2} n)}$ -time solution by Williams [71], a deterministic version with analogous runtime by Chan and Williams [18], and an $\tilde{O}(Mn^\omega)$ algorithm where M is the largest weight in the matrices, due to Alon, Galil and Margalit [3] (following Yuval [76]). Unfortunately, the latter algorithm is *pseudopolynomial* (exponential in the bit complexity), and can only be used to efficiently solve APSP in special cases such as when the weights in the graph are small [56, 77], or when the weights are on the nodes instead of the edges [15, 16].

Many over the years have asked if APSP can be solved faster than cubic time. For an explicit reference, Shoshan and Zwick [56] asked if the distance product of two $n \times n$ matrices with entries in $\{1, \dots, M\}$ can be computed in $O(n^{3-\delta} \log M)$ for some $\delta > 0$. (Note that an APSP algorithm of similar runtime would follow from such an algorithm.)

Triangles and Matrix Products. Itai and Rodeh [36] were the first to show that triangle detection can be done with Boolean matrix multiplication.

⁷Strassen’s algorithm also works over an arbitrary ring.

The trilinear decomposition of Pan [48, 49] implies that any bilinear circuit for computing the trace of the cube of a matrix A (i.e., $\text{tr}(A^3)$) over any ring can be used to compute matrix products over any ring. So in a sense, algebraic circuits that can *count the number of triangles* in a graph can be turned into matrix multiplication circuits. Note, this correspondence relies heavily on the algebraic circuit model: it is non-black box in an extreme way. (Our reductions are all black box.)

The k Shortest Paths Problem. A natural generalization of the s, t -shortest path problem is that of returning the first k of the shortest paths between s and t . In the early 1970s, Yen [74] and Lawler [40] presented an algorithm which solved this problem for directed graphs with nonnegative edge weights; with Fibonacci heaps [30] their algorithm runs in $O(k(mn + n^2 \log n))$ time. Eppstein [27] showed that if the paths can have cycles, then the problem can be solved in $O(k + m + n \log n)$ time. When the input graph is undirected, even the k shortest *simple* paths problem is solvable in $O(k(m + n \log n))$ time [39]. For directed unweighted graphs, the best known algorithm for the problem is the $\tilde{O}(km\sqrt{n})$ time randomized combinatorial algorithm of Roditty and Zwick [55]. Roditty [51, 52] noticed that the k shortest simple paths can be approximated fast, culminating in Bernstein’s [10] $\tilde{O}(km/\varepsilon)$ running time for a $(1 + \varepsilon)$ -approximation. When the paths are to be computed exactly, however, the best running time is still the $O(k(mn + n^2 \log n))$ time of Yen and Lawler’s algorithm.

Roditty and Zwick [54, 55] showed that the k shortest simple paths can be reduced to k computations of the second shortest simple path, and so any $T(m, n)$ time algorithm for the second shortest simple path implies an $O(kT(m, n))$ algorithm for the k shortest simple paths. The second shortest simple path always has the following form: take a prefix of the shortest path P to some node x , then take a path to some node y on P using only edges that are not on P (this part is called a detour), then take the remaining portion of P to t . The problem then reduces to finding a good detour.

Verifying a Metric. In the *metricity problem*, we are given an $n \times n$ nonnegative matrix A and want to determine whether it defines a metric space on $[n]$, that is, that A is symmetric, has 0s on the diagonal and its entries satisfy the triangle inequality. Matrices with this property are called *distance matrices*, and the metricity problem just asks whether a given matrix is a distance matrix.

Brickell et al. [12] studied a more general problem, called the metric nearness problem (MNP). MNP is defined with respect to a matrix norm $\|\cdot\|$, and is as follows: given a nonnegative symmetric $n \times n$ matrix D with 0s on the diagonal, find a distance matrix D' that is closest to D in the $\|\cdot\|$ norm. Brickell *et. al* [12] gave algorithms for MNP for different choices for the norm. Solving the metricity problem is exactly checking the stopping condition for their algorithms. They showed some connections of MNP to APSP, and asked whether the metricity problem can be solved faster than APSP. Theorem 4.2 partially answers their question in the sense that subcubic metricity implies subcubic APSP.

The Maximum Subarray Problem. The Maximum Subarray problem is as follows: given an $n \times n$ matrix A of integers, find $i, j, k, l \in [n]$ with $i \leq j, k \leq l$ maximizing $\sum_{x=i}^j \sum_{y=k}^l A[x, y]$, that is, find a contiguous subarray of A of maximum sum.

Bentley [9] introduced the problem in his Programming Pearls column in 1984. An $O(n^3)$ time algorithm for the maximum subarray problem is a simple exercise in divide-and-conquer. Tamaki and Tokuyama [60] showed how to use the ideas behind the standard divide-and-conquer solution to reduce the problem to computing the distance product of two matrices. Their reduction implies that if the distance product of two $n \times n$ matrices is in $T(n)$ time, then the Maximum Subarray problem for $n \times n$ matrices can be

solved in time $t(n)$ satisfying the recurrence $t(n) \leq 4t(n/2) + O(T(n))$. The recurrence solves to $t(n) \leq O\left(\sum_{i=0}^{\log n} 4^i T(n/2^i)\right)$. Using this reduction, Tamaki and Tokuyama obtained the first subcubic algorithm for the problem just by applying the slightly subcubic time algorithms for distance product. Moreover, for all T for which $T(n/2) \leq T(n)/(4 + \varepsilon)$ for some $\varepsilon > 0$, we get that $t(n) \leq O(T(n))$ and hence. the Tamaki and Tokuyama reduction shows that if distance product is in $O(n^{3-\varepsilon})$ time for some $\varepsilon > 0$, then the Maximum Subarray problem can also be solved in $O(n^{3-\varepsilon})$ time.

Prior reductions of APSP to other problems. Roditty and Zwick [53] consider the incremental and decremental versions of the single source shortest path problem in weighted and unweighted directed graphs. They show that either APSP has a truly subcubic algorithm, or any data structure for the decremental/incremental single source shortest paths problem must either have been initialized in cubic time, or its updates must take amortized $\Omega(n^2)$ time, or its query time must be $\Omega(n)$. They also give a similar relationship between the problem for unweighted directed graphs and combinatorial algorithms for BMM.

3SUM and its relatives. In this work, we build a complexity class of hard problems based on the difficulty of APSP. There is another well-known class of hard problems built around the 3SUM problem: *given a set of numbers, are there three which sum to zero?* There is a simple quadratic-time algorithm for this problem, and it is a major open problem to find a $n^{1.99}$ time algorithm. Gajentaan and Overmars [33] showed that for many problems Π solvable in quadratic time, one can reduce 3SUM to Π in such a way that a subquadratic algorithm for Π implies one for 3SUM. Hence under the conjecture that the 3SUM problem is hard to solve faster, many other Π are also hard.⁸

Most reductions showing 3SUM-hardness have been reductions between decision problems. One strength of our work is that we reduce function problems like APSP to decision problems like negative weight triangle. A significant advance in the theory of 3SUM-hardness was made by Patrascu [50] who showed how to reduce 3SUM to interesting function problems, such as triangle listing in sparse graphs. At the heart of his results is a reduction from 3SUM to Convolution-3SUM: given an array A of n numbers, determine if there are i, j such that $A[i] + A[j] = A[(i + j) \pmod n]$. Convolution-3SUM has an obvious quadratic-time algorithm: simply try all pairs i, j !

Theorem 2.1 (Patrascu [50]) *If Convolution-3SUM is in $O(n^2/f^2(n \cdot f(n)))$ time, then 3SUM on n numbers is in $O(n^2/f(n))$ time.*

In the notation of this paper, the above theorem says that $3SUM \equiv_2$ Convolution-3SUM, i.e., the two problems are *subquadratic equivalent*.

3 Subcubic Reducibility

Here we formally define the notion of subcubic reducibility used in this paper, and prove a few consequences of it. Recall that an *algorithm with oracle access to B* has special workspace in memory reserved for oracle calls, and at any step in the algorithm, it can call B on the content of the special workspace in one unit of time and receive a solution to B in the workspace.

⁸Sometimes Π is *defined* to be 3SUM-hard if “ Π is in subquadratic time implies 3SUM is in subquadratic time”. This definition leaves something to be desired: if 3SUM is in subquadratic time then all problems are 3SUM-hard, and if 3SUM is not in subquadratic time then no subquadratic problem is 3SUM-hard. Hence the 3SUM-hardness of some problems would depend on the complexity of 3SUM itself. Note this is *not* the definition of [33] – they give a reducibility notion similar to but weaker than ours.

Let Σ be an underlying alphabet. We define a *size measure* to be any function $m : \Sigma^* \rightarrow \mathbb{N} \cup \{\infty, -\infty\}$. In this paper, the size measure on n -node weighted graphs with integer weights from $[-M, M] \cup \{-\infty, \infty\}$ (or $n \times n$ matrices with entries in this range) is typically taken to be $n \cdot (\log M)^c$ for a fixed constant c . That is, we measure the complexity of a square matrix by its dimension times some polynomial in the bit complexity of the weights. This notion of measure is easier to understand in the context of the following definition.

Definition 3.1 *Let A and B be computational problems with a common size measure m on inputs. We say that there is a subcubic reduction from A to B if for every constant $\varepsilon > 0$ there are constants $\delta > 0$ and d , and an algorithm \mathcal{A} with oracle access to B , satisfying three properties:*

- For every instance x of A , $\mathcal{A}(x)$ solves the problem A on x .
- \mathcal{A} runs in $O(m^{3-\delta})$ time on instances of size m .⁹
- For every instance x of A of size m , let m_i be the size of the i th oracle call to B in $\mathcal{A}(x)$. Then, $\sum_i m_i^{3-\varepsilon} \leq d \cdot m^{3-\delta}$.

We use the notation $A \leq_3 B$ to denote the existence of a subcubic reduction from A to B , and define $A \equiv_3 B$ if $A \leq_3 B$ and $B \leq_3 A$. In such a case we say that A and B are subcubic-equivalent.

There is a natural extension of the concept to $O(n^q)$ running times, for any constant $q \geq 1$, by replacing all occurrences of 3 in the above definition with q . For such reductions we denote their existence by $A \leq_q B$, and say there is a *sub- q reduction* from A to B , for values of q such as “quadratic”, “cubic”, “quartic”, etc.

First let us observe that the reducibility relation is transitive.

Proposition 1 *Let A, B, C be problems so that $A \leq_q B$ and $B \leq_q C$. Then $A \leq_q C$.*

Proof. By definition, we have:

1. For every $\varepsilon > 0$ there exist constants $\delta > 0$ and d_A and an algorithm $P_{A,\varepsilon}$ for A that on all instances x of size n runs in $O(n^{q-\delta})$ time and makes oracle calls to B on instances x_1, \dots, x_t of sizes $|x_1|, \dots, |x_t|$ where $\sum_i |x_i|^{q-\varepsilon} \leq d_A \cdot n^{q-\delta}$.
2. For every $\varepsilon' > 0$ there exist constants $\delta' > 0$ and d_B and an algorithm $P_{B,\varepsilon'}$ for B that on all instances y of size m runs in $O(m^{q-\delta'})$ time and makes oracle calls to C on instances y_1, \dots, y_r of sizes $|y_1|, \dots, |y_r|$ where $\sum_i |y_i|^{q-\varepsilon} \leq d_B \cdot m^{q-\delta'}$.

We will show that:

3. For every $\varepsilon' > 0$ there exist constants $\delta'' > 0$ and d and an algorithm $P_{\varepsilon'}$ for A that on all instances z of size n runs in $O(n^{q-\delta''})$ time and makes oracle calls to C on instances z_1, \dots, z_w of sizes $|z_1|, \dots, |z_w|$ where $\sum_i |z_i|^{q-\varepsilon''} \leq d \cdot n^{q-\delta''}$.

Let $\varepsilon' > 0$ be given. Consider $P_{B,\varepsilon'}$ and let $\delta' > 0$ be the value corresponding to ε' , as in 2. Pick $\varepsilon = \delta'$. Consider algorithm $P_{A,\varepsilon}$ and let $\delta > 0$ be the value corresponding to ε , as in 1. Replace each oracle call on

⁹Note that for the size measure $n \cdot (\log M)^c$, this translates to $n^{3-\delta} \cdot (\log M)^{(3-\delta)c}$ time.

an instance x_i from algorithm $P_{A,\varepsilon}$ running on input x with a call to $P_{B,\varepsilon'}$ on x_i . This forms a new algorithm $P_{\varepsilon'}$ that makes oracle calls to C .

The running time of the $P_{\varepsilon'}$ is

$$O\left(n^{q-\delta} + \sum_i |x_i|^{q-\delta'}\right).$$

As we picked $\varepsilon = \delta'$, $\sum_i |x_i|^{q-\delta'} = \sum_i |x_i|^{q-\varepsilon} \leq d_A \cdot n^{q-\delta}$ (from 1), and the runtime of $P_{\varepsilon'}$ is $O(n^{q-\delta})$.

From 2, for each x_i , the algorithm makes oracle calls to C on instances x_{i1}, \dots, x_{ir} where $\sum_j |x_{i,j}|^{q-\varepsilon} \leq d_B \cdot |x_i|^{q-\delta}$. Hence

$$\sum_{ij} |x_{ij}|^{q-\varepsilon'} \leq \sum_i d_B \cdot |x_i|^{q-\delta'} = d_B \sum_i |x_i|^{q-\varepsilon} \leq (d_A d_B) n^{q-\delta},$$

where the last inequality is from 1. We can set $\delta'' = \delta$ and $d = d_A d_B$, and so $A \leq_q C$. \square

Corollary 3.1 *The relation \leq_q is a partial order, and the relation \equiv_q is an equivalence relation.*

Now let us verify that the definition gives us the property we want. In the following, let A and B be computational problems on $n \times n$ matrices with entries in $[-M, M]$ (or equivalently, weighted graphs on n nodes).

Proposition 2 *If $A \leq_3 B$ then a truly subcubic algorithm for B implies a truly subcubic algorithm for A .*

Proof. If there is an $O(m^{3-\varepsilon} \text{poly log } M)$ algorithm for B according to the measure m , then the algorithm for A in the reduction runs in time $O(m^{3-\delta} + \sum_i m_i^{3-\varepsilon} \text{poly log } M) \leq O(m^{3-\delta} \text{poly log } M)$. \square

Strongly Subcubic Reductions. All subcubic equivalences proved in this paper have one additional property in their reductions: the number of oracle calls and the sizes of oracle calls depend *only* on the input, and *not* on the parameter ε . (In some other reductions, such as the example below, this is not the case.) Let us define a reduction with this property to be a *strongly subcubic reduction*.

Definition 3.2 *Let A and B be computational problems with a common size measure m on inputs. We say that there is a strongly subcubic reduction from A to B if there exists an algorithm \mathcal{A} with oracle access to B and a constant d , satisfying the following properties:*

- For every instance x of A , $\mathcal{A}(x)$ solves the problem A on x .
- \mathcal{A} runs in $O(m^{3-\gamma})$ time on instances of size m for some $\gamma > 0$ independent of m .
- For every instance x of A of size m , let m_i be the size of the i th oracle call to B in $\mathcal{A}(x)$. Then, for every $\varepsilon > 0$, there exists a $\delta > 0$ such that $\sum_i m_i^{3-\varepsilon} \leq d \cdot m^{3-\delta}$.

We now show that these stronger reductions have the nice quality that, with respect to polylogarithmic (or even slightly higher) improvements, running times are preserved.

Theorem 3.1 *If there is a strongly subcubic reduction from A to B , then*

- If the size measure is $m = n(\log M)^{r/3}$, then for all $c > 0$, an $O(n^3(\log M)^r/(\log n + \log \log M)^c)$ time algorithm for B implies an $O(n^3(\log M)^{3r}/(\log n + \log \log M)^c)$ time algorithm for A . If the size measure is n , then an $O(n^3/\log^c n)$ algorithm for B implies an $O(n^3/\log^c n)$ algorithm for A .
- If the size measure is n , then for all $\alpha > 0$, an $n^3/2^{\Omega(\log^\alpha n)}$ algorithm for B implies an $n^3/2^{\Omega(\log^\alpha n)}$ algorithm for A .

Proof. First let n be the input size measure. First, we show that

$$\sum_i n_i^3 \leq d \cdot n^3. \quad (1)$$

A strongly subcubic reduction gives us a fixed algorithm such that for all sizes n , the number of oracle calls and the sizes of oracle calls $\{n_i\}$ depend only on the input (and not the parameter ε). Then, for all $\varepsilon > 0$, there is a $\delta > 0$ satisfying

$$\sum_i n_i^{3-\varepsilon} \leq dn^{3-\delta} < dn^3.$$

Since d , $\{n_i\}$ and n are independent of ε , this means that for every fixed set $\{n_i\}$, d and n , we can take the limit on both sides of the above inequality as $\varepsilon \rightarrow 0$. We obtain that for every n and every set of oracle call sizes $\{n_i\}$ on an input of size n , $\sum_i n_i^3 \leq dn^3$.

Now consider an algorithm for B that runs in $O(n^3/\log^c n)$ time. Then an algorithm for A that uses the reduction calling B as an oracle would run in $O(n^{3-\gamma} + \sum_i n_i^3/\log^c n_i)$ time for some $\gamma > 0$. Let $a < \delta/3$. Then

$$\sum_i n_i^3/\log^c n_i = \sum_{i: n_i < n^a} n_i^3/\log^c n_i + \sum_{i: n_i \geq n^a} n_i^3/\log^c n_i,$$

which is at most

$$O\left(n^{3-\delta} \cdot n^{3a} + \sum_{i: n_i \geq n^a} n_i^3/\log^c(n^a)\right),$$

since the number of oracle calls is at most $O(n^{3-\delta})$. The first term is $n^{3-\varepsilon'}$ for some $\varepsilon' > 0$, by our choice of a . By (1), we have

$$O\left(n^{3-\varepsilon'} + \sum_{i: n_i \geq n^a} n_i^3/(a^c \cdot \log^c n)\right) \leq O(n^3/\log^c n).$$

Suppose now that the input measure m is $m = n(\log M)^{r/3}$. Then, an $O(n^3(\log M)^r/(\log n + \log \log M)^c)$ time algorithm for B is an $O(m^3/(\log m)^c)$ time algorithm for B . By the argument above, using m instead of n , we obtain that a strongly subcubic reduction with respect to measure m , implies that there is also an $O(m^3/(\log m)^c)$ time algorithm for A , and hence an $O(n^3(\log M)^r/(\log n + \log \log M)^c)$ time algorithm for A .

For the proof of the second item, suppose the size measure is n and consider an algorithm for B that runs in $n^3/2^{c \log^\alpha n}$ time. Then an algorithm for A that uses the reduction calling B as an oracle would run in $O(n^{3-\gamma} + \sum_i n_i^3/2^{c \log^\alpha n_i})$ time for some $\gamma > 0$. Similar to above, let $a < \delta/3$ and use that

$$\sum_i n_i^3/2^{c \log^\alpha n} = \sum_{i: n_i < n^a} n_i^3/2^{c \log^\alpha n_i} + \sum_{i: n_i \geq n^a} n_i^3/2^{c \log^\alpha n_i},$$

which is at most

$$O\left(n^{3-\delta} \cdot n^{3a} + \sum_{i: n_i \geq n^a} n_i^3 / 2^{c \log^\alpha n^a}\right).$$

The first term is $n^{3-\varepsilon'}$ for some $\varepsilon' > 0$, and by (1) the second term is $O(n^3 / 2^{ca^\alpha \log^\alpha n}) = n^3 / 2^{\Omega(\log^\alpha n)}$. \square

It can be shown that strongly subcubic reductions are *necessary* for Theorem 3.1 to hold. If the sizes of oracle calls or their number depend on ε , one can find cases where polylog factors are diminished in the algorithm for A . (In fact, the reduction below of Matoušek is one example.)

Subcubic reductions were certainly implicit in prior work (even in the generic setting we give here), but have not been studied systematically. For one example, Matoušek [46] showed that computing dominances in \mathbb{R}^n between pairs of n vectors can be done in $O(n^{(3+t)/2})$ time, where $t > \omega$ is any upper bound on the exponent ω of $n \times n$ integer matrix multiplication. For the current best upper bound on the matrix multiplication exponent $\omega < 2.373$, Matoušek's algorithm runs in $O(n^{2.687})$ time. The algorithm works by making $O(n^{3/2}/n^{t/2})$ calls to $n \times n$ integer matrix multiplication. (Note this is *not* a strongly subcubic reduction, since the number of calls depends on t .) Notice that for any $t < 3$, the running time $O(n^{(3+t)/2})$ is truly subcubic. Hence we can say:

Dominances in $\mathbb{R}^n \leq_3$ Integer Matrix Multiplication.

Another example is that of *3SUM-hardness* in computational geometry, as mentioned in Section 2.1. A proof that a problem Π is 3SUM-hard implies $3\text{SUM} \leq_2 \Pi$, but the notion of reduction used in [33] is weaker than ours. (They only allow $O(1)$ calls to the oracle for Π .)

4 Equivalences Between Problems on Generic Structures

A generic approach to computing fast (\min, \odot) matrix products (for an arbitrary binary operation \odot) would be of major interest. Here we prove truly subcubic equivalences between matrix products, negative triangles, and matrix product verification for (\min, \odot) structures. (For definitions, see the Preliminaries.) For simplicity, most of our theorems will assume $T(n)$ -time algorithms; they can easily be extended to handle dependencies on the weights. However, all of the proofs have the property that if the original maximum weight absolute value was W , then the maximum weight absolute value after the reduction is $\text{poly}(nW)$, and hence the subcubic reduction properties are preserved.

Reminder of Theorem 1.2 *Let $\bar{\mathcal{R}}$ be an extended (\min, \odot) structure. The following problems over $\bar{\mathcal{R}}$ either all have truly subcubic algorithms, or none of them do:*

- **Negative Triangle Detection.** *Given a tripartite n -node graph with node partitions I, J, K and weight function $w : E \rightarrow R$, find nodes $i \in I, j \in J, k \in K$ such that $(w(i, k) \odot w(k, j)) + w(i, j) < 0$.*
- **Matrix Product.** *Given two $n \times n$ matrices A, B with entries from R , compute the product of A and B over $\bar{\mathcal{R}}$.*
- **Matrix Product Verification.** *Given three $n \times n$ matrices A, B, C with entries from R , determine if the product of A and B over $\bar{\mathcal{R}}$ is C .*

4.1 Matrix Product Verification Implies Negative Triangle Detection

We start by showing that matrix product verification can solve the negative triangle problem over any extended structure $\bar{\mathcal{R}}$ in the same asymptotic runtime. For two problems A and B , we write $A \leq_3 B$ to express that there is a subcubic reduction from A to B . (For formal definitions, see Section 3.)

Theorem 4.1 (Negative Triangle Over $\bar{\mathcal{R}} \leq_3$ Matrix Product Verification Over $\bar{\mathcal{R}}$) *Suppose matrix product verification over $\bar{\mathcal{R}}$ can be done in time $T(n)$. Then the negative triangle problem for graphs over $\bar{\mathcal{R}}$ can be solved in $O(T(2n))$ time.*

Proof. From the tripartite graph $G = (I \cup J \cup K, E)$ given by the negative triangle problem over $\bar{\mathcal{R}}$, construct matrices A, B, C as follows. For each edge $(i, j) \in (I \times J) \cap E$ set $C[i, j] = -w(i, j)$. Similarly, for each edge $(i, k) \in (I \times K) \cap E$ set $A[i, k] = w(i, k)$ and for each edge $(k, j) \in (K \times J) \cap E$ set $B[k, j] = w(k, j)$. When there is no edge in the graph, the corresponding matrix entry in A or B is set to ε_0 and in C it is set to ∞ . The problem becomes to determine whether there are $i, j, k \in [n]$ so that $A[i, k] \odot B[k, j] < C[i, j]$. Let A' be the $n \times 2n$ matrix obtained by concatenating A to the left of the $n \times n$ identity matrix I . Let B' be the $2n \times n$ matrix obtained by concatenating B on top of C . Then $A' \odot B'$ is equal to the componentwise minimum of $A \odot B$ and C . One can easily complete A', B' and C to be square $2n \times 2n$ matrices, by concatenating:

- an $n \times 2n$ matrix of all ε_0 's to the bottom of A' ,
- a $2n \times n$ matrix of all ε_0 's to the right of B' , and
- n columns of all ε_0 's and n rows of all ε_0 's to the right and bottom of C , respectively.

Notice that all entries of A', B', C are from R since R is symmetric and contains $\varepsilon_0, \varepsilon_1$, and ∞ . Run matrix product verification on A', B', C . Suppose there are i, j so that $\min_k (A'[i, k] \odot B'[k, j]) \neq C[i, j]$. Then since

$$\min_k (A'[i, k] \odot B'[k, j]) = \min\{C[i, j], \min_k (A[i, k] \odot B[k, j])\} \leq C[i, j],$$

there must exist a $k \in [n]$ so that $A[i, k] \odot B[k, j] < C[i, j]$. In other words, i, k, j is a negative triangle over $\bar{\mathcal{R}}$. If on the other hand for all i, j we have $\min_k (A'[i, k] \odot B'[k, j]) = C[i, j]$, then for all i, j we have $\min_k (A[i, k] \odot B[k, j]) \geq C[i, j]$ and there is no negative triangle. \square

4.2 Negative Triangle Detection Implies Matrix Multiplication

Next we show that from negative triangle detection over a (\min, \odot) structure \mathcal{R} , we can obtain the full matrix product over \mathcal{R} . Specifically, we prove the following.

Theorem 4.2 (Matrix Product Over $\mathcal{R} \leq_3$ Negative Triangle Over \mathcal{R}) *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. Suppose the negative triangle problem over \mathcal{R} in an n -node graph can be solved in $T(n)$ time. Then the product of two $n \times n$ matrices over \mathcal{R} can be performed in $O(n^2 \cdot T(n^{1/3}) \log W)$ time, where W is the maxint of R .*

Before we proceed, let us state some simple but useful relationships between triangle detecting, finding, and listing.

Lemma 4.1 (Folklore) *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. If there is a $T(n)$ time algorithm for negative triangle detection over \mathcal{R} on a graph $G = (I \cup J \cup K, E)$, then there is an $O(T(n))$ algorithm which returns a negative triangle over \mathcal{R} in G if one exists.*

Proof of Lemma 4.1. The algorithm is recursive: it proceeds by first splitting I , J and K each into two roughly equal parts I_1 and I_2 , J_1 and J_2 , and K_1 and K_2 . Then it runs the detection algorithm on all 8 induced subinstances (I_i, J_j, K_k) , $i, j, k \in \{1, 2\}$. If none of these return 'yes', then there is no negative triangle in G . Otherwise, the algorithm recurses on exactly one subinstance on which the detection algorithm returns 'yes'. The base case is when $|I| = |J| = |K| = 1$ and then one just checks whether the three nodes form a negative triangle in $O(1)$ time. The running time becomes

$$T'(n) = 8T(n/2) + T'(n/2), T'(1) = O(1).$$

If $T(n) = nf(n)$ for some nondecreasing function $f(n)$, then $T(n) = 2\frac{n}{2}f(n) \geq 2\frac{n}{2}f(n/2) = 2T(n/2)$. Hence, $T(n/2^i) \leq T(n)/2^i$ for all positive integers i , and the recurrence above solves to $T'(n) = O(T(n))$. \square

It will be useful in our final algorithm to have a method for finding many triangles, given an algorithm that can detect one. We can extend Lemma 4.1 in a new way, to show that subcubic negative triangle detection implies *subcubic negative triangle listing*, provided that the number of negative triangles to be listed is subcubic.

Theorem 4.3 (Negative Triangle Listing Over $\mathcal{R} \leq_3$ Negative Triangle Over \mathcal{R}) *Let $\delta > 0$ be fixed, and let $\Delta = O(n^{3-\delta})$. Suppose there is a truly subcubic algorithm for negative triangle detection over \mathcal{R} . Then there is a truly subcubic algorithm which lists Δ negative triangles over \mathcal{R} in any graph with at least Δ negative triangles.*

Proof of Theorem 4.3. Let P be an $O(n^{3-\varepsilon} \log^c M)$ time algorithm for negative triangle over \mathcal{R} for $\varepsilon > 0$ and where M is the maxint of \mathcal{R} . Let $\Delta = O(n^{3-\delta})$ for $\delta > 0$. Given a $3n$ -node tripartite graph $G = (I \cup J \cup K, E)$ with at least Δ negative triangles over \mathcal{R} we provide a procedure to list Δ negative triangles over \mathcal{R} .

We partition the nodes in I, J, K into $\Delta^{1/3}$ parts, each of size $O(n/\Delta^{1/3})$. For all Δ triples $I' \subset I, J' \subset J, K' \subset K$ of parts, run P in $O(\Delta(n/\Delta^{1/3})^{3-\varepsilon} \log^c M)$ time overall to determine all triples which contain negative triangles over \mathcal{R} .

On the triples which contain negative triangles, we run the following procedure. We begin by creating a list L_0 consisting of all the triples $\{I', J', K'\}$ above with $|I'| = |J'| = |K'| = O(n/\Delta^{1/3})$ for which P determined that they contain a negative triangle over \mathcal{R} .

In general, we will iterate through i and build a list L_i , for each integer i ranging from 0 to $\log \frac{n}{\Delta^{1/3}}$, so that L_i contains a list of disjoint triples $\{I', J', K'\}$ with $|I'| = |J'| = |K'| = O(n/(2^i \Delta^{1/3}))$ so that each triple contains a negative triangle over \mathcal{R} . In iteration i we will process list L_i , building list L_{i+1} as follows. L_{i+1} is initialized to be the empty list. Now we process each triple in L_i in turn.

Let $\{I', J', K'\}$ be the next triple in L_i to be processed. Split I', J' and K' each into two roughly equal halves. On each of the 8 possible triples of halves, run P and determine the triples of halves which contain negative triangles. Place each such triple into the list L_{i+1} . If $|L_{i+1}|$ becomes $\geq \Delta$, then run the algorithm from Lemma 4.1 using P on the triples in L_{i+1} (there are at most $\Delta + 7$ of them), and return the triangles found. In this case, one is guaranteed to return at least Δ (and at most $\Delta + 7$) different negative triangles

over \mathcal{R} . If on the other hand $|L_{i+1}| < \Delta$, then move on to the next triple in L_i looking for more triples of halves to add to L_{i+1} . If L_i becomes empty, then increment i to $i + 1$ and process L_{i+1} . If i becomes equal to $\log \frac{n}{\Delta^{1/3}}$, then each triple in L_i contains exactly 3 nodes, and since the algorithm maintains the invariant that $|L_i| < \Delta$ (otherwise it returns $\geq \Delta$ triangles), we can return that the instance does not contain at least Δ triangles. (Alternatively, one can return a maximal subset of the negative triangles in L_i .)

For each i , the list L_i contains triples that contain $O\left(\frac{n}{2^i \Delta^{1/3}}\right)$ nodes, and each L_i has $O(\Delta)$ triples. Therefore, for each iteration i , the running time is $O\left(\Delta \cdot \left(\frac{n}{2^i \Delta^{1/3}}\right)^{3-\varepsilon} \log^c M\right)$. Since $\varepsilon < 3$, the overall runtime becomes asymptotically

$$\Delta \left(\frac{n}{\Delta^{1/3}}\right)^{3-\varepsilon} \log^c M \cdot \sum_i \left(\frac{1}{2^{3-\varepsilon}}\right)^i = O\left(\Delta^{\varepsilon/3} n^{3-\varepsilon} \log^c M\right).$$

When $\Delta \leq O(n^{3-\delta})$, the runtime is

$$O(n^{3-\varepsilon+3\varepsilon/3-\delta\varepsilon/3} \log^c M) = O(n^{3-\delta\varepsilon/3} \log^c M),$$

which is truly subcubic for any $\varepsilon, \delta > 0$. □

Next we show that fast negative triangle detection over \mathcal{R} implies a fast algorithm for finding many edge-disjoint negative triangles over \mathcal{R} . Consider a tripartite graph with parts I, J, K . We say a set of triangles $T \subseteq I \times J \times K$ in the graph is IJ -disjoint if for all $(i, j, k) \in T, (i', j', k') \in T, (i, j) \neq (i', j')$.

Lemma 4.2 *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. Given a $T(n)$ algorithm for negative triangle detection over \mathcal{R} , there is an algorithm A which outputs a maximal set L of IJ -disjoint negative triangles over \mathcal{R} in a tripartite graph with distinguished parts (I, J, K) , in $O(T(n^{1/3})n^2)$ time. Furthermore, if there is a constant $\varepsilon : 0 < \varepsilon < 1$ such that for all large enough n , $T(n) \geq T(2^{1/3}n)/(2(1-\varepsilon))$, then there is an output-sensitive $O(T(n/|L|^{1/3})|L|)$ -time algorithm.¹⁰*

In particular, Lemma 4.2 implies that given any graph on n nodes, we can determine those pairs of nodes that lie on a negative triangle in $O(T(n^{1/3})n^2)$ time. The condition required for the output sensitive algorithm holds for all subcubic polynomials, but it does not necessarily hold for runtimes of the form $n^3/f(n)$ with $f(n) = n^{o(1)}$. In the special case when $T(n)$ is $\Theta(n^3/\log^c n)$ for a constant c , the output sensitive algorithm only multiplies a $\log |L|$ factor to the runtime. Notice also that although we state Lemma 4.2 only for runtimes in terms of n , one can modify its proof to obtain a similar result for runtimes that also depend on the maxint of R .

Proof. Algorithm A maintains a global list L of negative triangles over \mathcal{R} which is originally empty and will be the eventual output of the algorithm. Let a be a parameter to be set later. At each point the algorithm works with a subgraph \tilde{G} of the original graph, containing all of the nodes, all of the edges between I and K and between J and K but only a subset of the edges between I and J . In the beginning $\tilde{G} = G$ and at each step A removes an edge from \tilde{G} .

Algorithm A starts by partitioning each set I, J, K into n^a parts where each part has at most $\lceil n^{(1-a)} \rceil$ nodes each. It iterates through all n^{3a} possible ways to choose a triple of parts (I', J', K') so that $I' \subset I, J' \subset J$ and $K' \subset K$. For each triple (I', J', K') in turn, it considers the subgraph G' of \tilde{G} induced by

¹⁰The condition is satisfied for instance when $T(n)/n^{3-\delta}$ is nonincreasing for some $\delta > 0$.

$I' \cup J' \cup K'$ and repeatedly uses Lemma 4.1 to return a negative triangle over \mathcal{R} . Each time a negative triangle (i, j, k) is found in G' , the algorithm adds (i, j, k) to L , removes edge (i, j) from \tilde{G} and attempts to find a new negative triangle in G' . This process repeats until G' contains no negative triangles, in which case algorithm A moves on to the next triple of parts.

Now, let us analyze the running time of A . For a triple of parts (I', J', K') let $e_{I'J'K'}$ be the number of edges (i, j) in $I' \times J'$ that are found in the set of $I'J'$ -disjoint negative triangles when (I', J', K') is processed by A . Let $T(n)$ be the complexity of negative triangle detection over \mathcal{R} . Then the runtime can be bounded from above as:

$$O\left(\sum_{\text{all } n^{3a} \text{ triples } I', J', K'} (e_{I'J'K'} \cdot T(n^{1-a}) + T(n^{1-a}))\right). \quad (2)$$

Note that the sum of all $e_{I'J'K'}$ is at most n^2 , since if edge $(i, j) \in I' \times J'$ is reported to be in a negative triangle, then it is removed from the graph. Hence there is a constant $c > 0$ such that (2) is upper bounded by:

$$\begin{aligned} c \cdot T(n^{1-a}) \cdot \sum_{\text{all } n^{3a} \text{ triples } I', J', K'} (e_{I'J'K'} + 1) &\leq c \cdot T(n^{1-a}) \cdot \left(n^{3a} + \sum_{\text{all } n^{3a} \text{ triples } I', J', K'} e_{I'J'K'} \right) \\ &\leq c \cdot T(n^{1-a}) \cdot (n^{3a} + n^2). \end{aligned}$$

Setting $a = 2/3$, the runtime becomes $O(n^2 T(n^{1/3}))$.

To get an output-sensitive algorithm A' , we make the following modification. For all $i = 1, \dots, 2 \log n$, run algorithm A with $a := i/(3 \log n)$. In stage i , stop the algorithm when the list L contains at least 2^i IJ -disjoint negative triangles. Let L_{i-1} be the list of IJ -disjoint negative triangles found in stage $i - 1$. If $|L| = |L_{i-1}|$ then return L ; otherwise set $L_i := L$ and continue with stage $i + 1$.

The runtime of A' is

$$\begin{aligned} \sum_{i=1}^{\log |L|} T(n^{1-i/(3 \log n)}) \cdot \left(n^{3i/(3 \log n)} + \sum_{\text{all } n^{3i/(3 \log n)} \text{ triples } I', J', K'} (e_{I'J'K'}) \right) &\leq \\ \sum_{i=1}^{\log |L|} \left(n^{i/\log n} + 2^i \right) \cdot T(n^{1-i/(3 \log n)}) &= 2 \sum_{i=1}^{\log |L|} 2^i T(2^{\log n - i/3}) = 2 \sum_{i=1}^{\log |L|} 2^i T(n/2^{i/3}). \end{aligned}$$

Since there is a constant $\varepsilon < 1$ so that for all n , $T(n) \geq T(2^{1/3}n)/(2(1-\varepsilon))$, then for all i , $2^i T(n/2^{i/3}) \leq 2^{i+1}(1-\varepsilon)T(n/2^{(i+1)/3})$ and hence the runtime is bounded by

$$O\left(T(n/|L|^{1/3})|L| \sum_{i=0}^{\log |L|} (1-\varepsilon)^i\right) = O(T(n/|L|^{1/3})|L|).$$

□

We are now ready to prove Theorem 4.2, via a simultaneous binary search on entries of the matrix product. The ‘‘oracle’’ used for binary search is our algorithm for IJ -disjoint triangles.

Proof of Theorem 4.2. Let A and B be the given $n \times n$ matrices, and let $C = A \odot B$ be the matrix to be computed. Let W be the maxint of R . We will binary search on $[-W, W]$ for the finite entries of C .

We maintain two $n \times n$ matrices S and H , initializing them as $S[i, j] := -W$ and $H[i, j] := W + 1$ for all $i, j \in [n]$. The algorithm proceeds in iterations. It maintains the invariant that for every $i, j \in [n]$, either $C[i, j] \in \{-\infty, \infty\}$, or $S[i, j] \leq C[i, j] < H[i, j]$, that is, $S[i, j]$ and $H[i, j]$ are a lower and upper bounds on the finite product entries. The invariant is true at the onset of the algorithm by our initialization.

In each iteration, we create a complete tripartite graph G on partitions I, J and K . The edges of G have weights $w(\cdot)$ so that for $i \in I, j \in J$ and $k \in K$, $w(i, k) = A[i, k]$, $w(k, j) = B[k, j]$ and $w(i, j) = -\lceil (S[i, j] + H[i, j])/2 \rceil$ if $\lceil (S[i, j] + H[i, j])/2 \rceil \leq W$ and $w(i, j) = -\infty$ otherwise.

Notice that the weights of the edges of G are always in R since we guaranteed that $-w(i, j) \leq W$ or $w(i, j) = -\infty$, and in addition, if $w(i, j) \notin \{-\infty, \infty\}$, we have by the invariant $-w(i, j) = \lceil (S[i, j] + H[i, j])/2 \rceil \geq \lceil (-W + C[i, j])/2 \rceil \geq -W$.

Now, using the algorithm from Lemma 4.2 on G , we generate a list L of IJ -disjoint negative triangles over \mathcal{R} for G in $O(T(n))$ time.

From this we modify S and H as follows. If (i, j) appears in a triangle in L for $i \in I, j \in J$, then we set $H[i, j] := -w(i, j)$ if $w(i, j)$ is finite, and make no changes otherwise. If (i, j) does not appear in a triangle in L , we set $S[i, j] := -w(i, j)$ if $w(i, j)$ is finite, and make no changes otherwise. Notice that the invariant is still satisfied. Indeed, if (i, j) appears in L , then there is some $k \in K$ for which $A[i, k] \odot B[k, j] < -w[i, j]$, and hence in particular, $C[i, j] = \min_k A[i, k] \odot B[k, j] < -w[i, j]$. Otherwise, if (i, j) is not in L , then for all $k \in K$, $A[i, k] \odot B[k, j] \geq -w(i, j)$, and hence in particular $C[i, j] \geq -w(i, j)$.

We continue iterating until for all i, j , $H[i, j] \leq S[i, j] + 1$. Each iteration shrinks the interval $[S[i, j], H[i, j]]$ for every i, j roughly in half, except for the special case $w(i, j) = -\infty$ when the interval is not changed. However, $w(i, j) = -\infty$ if and only if $S[i, j] = W$ and $H[i, j] = W + 1$, so the loop ending condition $H[i, j] \leq S[i, j] + 1$ is satisfied. Hence the number of iterations is $O(\log W)$.

First we show that we can easily tell whether some $C[i, j]$ is not finite. Notice that for any finite choice of $w(i, j)$, if $C[i, j] = \infty$, then (i, j) will not appear in L , and if $C[i, j] = -\infty$, then (i, j) will appear in L . After the last iteration, if $C[i, j] = -\infty$, then $H[i, j] = -W$, and if $C[i, j] = \infty$, then $S[i, j] = W$ and $H[i, j] = W + 1$. Thus we can set $C[i, j] = -\infty$ for all i, j with $H[i, j] = -W$. To determine the (i, j) with $C[i, j] = \infty$ we can run one more call to negative triangle detection using $w(i, j) = -\infty$ for all (i, j) . If some (i, j) with $S[i, j] = W$ and $H[i, j] = W + 1$ is in L , then there exists some $k \in K$ such that $w(i, k) \odot w(k, j) < \infty$, and hence $C[i, j] = \min_k w(i, k) \odot w(k, j) \leq W$. However, by the invariant, $C[i, j] \geq W$ and we can conclude that $C[i, j] = W$. Otherwise if $S[i, j] = W$ and $H[i, j] = W + 1$ and (i, j) is not in L , then $C[i, j] = \infty$.

After determining all infinite entries, for all i, j for which $-W \leq S[i, j] \leq W$, we set $C[i, j] := S[i, j]$. To see that this correctly computes C , notice that after the last iteration the algorithm has determined (by the invariant) that for every pair of nodes i, j with finite $C[i, j]$, $S[i, j] \leq C[i, j] < H[i, j] \leq S[i, j] + 1$. Since the finite values $C[i, j]$ are integers, we can safely set $C[i, j] := S[i, j]$ for all (i, j) . \square

Corollary 4.1 *Suppose the negative triangle problem over \mathcal{R} is in $O(n^3 / \log^c n)$ time for some constant c . Then the product of $n \times n$ matrices over \mathcal{R} can be done in $O((\log W)n^3 / \log^c n)$ time where W is the maxint of R .*

An important special case of matrix multiplication is that of multiplying rectangular matrices. Negative

triangle detection can also give a speedup in this case as well.

Theorem 4.4 *Suppose the negative triangle problem over \mathcal{R} is in $T(n)$ time. Let m, n, p be such that $mp \leq n^3$ and $\sqrt{p} \leq m \leq p^2$. Then two matrices of dimensions $m \times n$ and $n \times p$ can be multiplied over \mathcal{R} in $O(mp \cdot T(n^{1/3}) \log W)$ time, where W is the maxint of R .*

If $T(n) = n^c$ the runtime is $O(mp(n)^{c/3})$. Notice that if $c < 3$ and if $p = n^{(3-c)/3}$, then the runtime would be $O(mn)$, for all $m \in [n^{(3-c)/6}, n^{2(3-c)/3}]$. That is, for any $c < 3$, there is some $p \geq n^\varepsilon$ such that multiplication of $m \times n$ and $n \times p$ matrices over \mathcal{R} can be done *optimally*, for any $m \in [\sqrt{p}, p^2]$. Theorem 4.4 follows from a more general lemma:

Lemma 4.3 *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. Suppose there is a $T(n)$ time algorithm for negative triangle detection over \mathcal{R} in an n node graph. Let W be the maxint of R . Let A be an $m \times n$ matrix over R and let B be an $n \times p$ matrix over R , so that their product over \mathcal{R} does not contain $-\infty$ entries. Then:*

- *There is an algorithm that computes ℓ finite entries of the product over \mathcal{R} of A and B in $O(\ell \cdot T((mnp/\ell)^{1/3}) \log W)$ time, whenever $\ell \leq m^3, n^3, p^3$.*
- *If there is a constant $\varepsilon : 0 < \varepsilon < 1$ such that for all large enough n , $T(n) \geq T(2^{1/3}n)/(2(1-\varepsilon))$, then there is an $O(\ell \cdot T((mnp/\ell)^{1/3}) \log W)$ -time algorithm for computing the product of A and B over \mathcal{R} , where ℓ is the number of finite entries in the product matrix, whenever $mp \leq n^3$ and $\sqrt{p} \leq m \leq p^2$.*

Proof. Following the ideas from Theorem 4.2, ℓ distinct IJ -disjoint negative triangles over \mathcal{R} can be found in

$$O((\ell + a^3) \cdot T((mnp)^{1/3}/a))$$

time, where a is a bucketting parameter such that $a \leq m, n, p$. Since $\ell^{1/3} \leq m, n, p$ in the first bullet of the theorem, we set $a = \ell^{1/3}$ and we get a runtime of $O(\ell \cdot T((mnp/\ell)^{1/3}))$ for finding ℓ distinct IJ -disjoint negative triangles. Armed with this algorithm and given an $m \times n$ matrix A and an $n \times p$ matrix B over \mathcal{R} with no $-\infty$ s, we create the corresponding instance of negative triangle listing as in Theorem 4.2 where the $I \times K$ edge weights are taken from A , the $K \times J$ edge weights are taken from B and the $I \times J$ weights are all ∞ . On this graph we run the IJ -disjoint negative triangles algorithm from above, and determine the indices of ℓ finite entries of the product. (Since the product does not contain $-\infty$ entries, all entries that are $< \infty$ will be finite.)

Now, we start a binary search procedure as follows: for each of the ℓ finite entries (i, j) that we have found above, we set $S[i, j] = -W$, $H[i, j] = W + 1$, and for all other entries we set $S[i, j] = W[i, j] = W + 1$. Then we let the binary search procedure run exactly as in Theorem 4.2, except that in each search we use the procedure that lists ℓ IJ -negative triangles. These ℓ triangles are guaranteed to be the original ℓ ones, as any other (i, j) will have $w(i, j)$ set to $-\infty$, and thus no triangles going through (i, j) can be negative. The number of binary search steps is $O(\log W)$ and each one takes $O(\ell \cdot T((mnp/\ell)^{1/3}))$ time and we have proven the first bullet.

To get an output-sensitive algorithm as in the second bullet of the theorem statement, for each $i = 1, \dots, \log(mp)$, we set $a = 2^{i/3}$ and attempt to list 2^i negative triangles over \mathcal{R} as in the first bullet. One of these attempts will compute all ℓ finite entries as there will be some i with $2^i \geq \ell > 2^{i-1}$. We can run the algorithm from the first bullet since for each i , $2^i \leq mp \leq m^3, n^3, p^3$. The runtime together with the binary

search is now

$$\sum_{i=1}^{\log(mnp)} 2^i \cdot T((mnp/2^i)^{1/3}) \log W.$$

Since there is a constant $\varepsilon < 1$ so that for all n , $T(n) \geq T(2^{1/3}n)/(2(1-\varepsilon))$, then for all i , $2^i T((mnp)^{1/3}/2^{i/3}) \leq 2^{i+1}(1-\varepsilon)T((mnp)^{1/3}/2^{(i+1)/3})$ and hence the runtime is $O(\ell \cdot T((mnp/\ell)^{1/3}) \log W)$, where ℓ is the number of finite entries in the output. \square

4.3 Strongly Polynomial Subcubic Reductions

Applying a very useful randomization trick by Chan [14] in a new way, the logarithmic dependence on M in our generic reductions can be replaced with a polylogarithmic dependence on n . That is, the running time of our reductions can be made *strongly polynomial*, independent of the weights. The only reduction we need to improve is the one from matrix product to negative triangle:

Theorem 4.5 (Matrix Product Over $\mathcal{R} \leq_3$ Negative Triangle Over \mathcal{R} , Strongly Polynomial) *Let $T(n)$ be a function so that $T(n)/n^2$ is nondecreasing. Suppose the negative triangle problem over \mathcal{R} in an n -node graph can be solved in $T(n)$ time. Then the product of two $n \times n$ matrices over \mathcal{R} can be computed in $O(n^2 \cdot T(n^{1/3}) \log n)$ time, with high probability.*

In this section we provide a Monte Carlo algorithm as in the statement of the theorem, however it is not hard to obtain a Las Vegas algorithm as well.

Fix an instance of negative triangle with node sets I, J, K and weight function w . Let $i \in I, j \in J, k \in K$. Recall that the triple (i, j, k) is a negative triangle iff $(w(i, k) \odot w(k, j)) + w(i, j) < 0$. Fix a total ordering $<$ on the nodes in K in the negative triangle instance. For any $i \in I, j \in J$, a node $k \in K$ is called a *minimum witness* for (i, j) if (i, j, k) is a negative triangle but (i, j, k') is not a negative triangle for all $k' < k$ according to the ordering.

First we show that any superlinear time negative triangle detection algorithm can be converted to a minimum witness negative triangle finding algorithm, running in roughly the same time.

Lemma 4.4 (Minimum Witness Finding From Detection) *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. Let A be an algorithm which detects a negative triangle over \mathcal{R} in an n -node graph in $T(n)$ time. Then in $O(T(n))$ time, given an n -node weighted graph G one can find a single negative triangle (i, j, k) over \mathcal{R} such that k is the minimum witness for pair (i, j) , or determine that there is no negative triangle in G over \mathcal{R} .*

Proof. If there are only 3 remaining nodes, $i \in I, j \in J, k \in K$, return $\{i, j, k\}$ if it forms a negative triangle over \mathcal{R} . Otherwise, $|I|, |J|, |K| \geq 2$. Split each part I, J, K into two pieces of roughly $n/2$ nodes each, so that K is split into K_1, K_2 where K_1 contains the first half of the nodes of K according to the ordering, and K_2 contains the rest of the nodes of K . Then, for all 8 triples $a, b, c \in [2]^3$, use the negative triangle detection algorithm on the graphs induced by the unions of parts $I_a \cup J_b \cup K_c$, in $8T(n/2)$ time. Let $c \in \{1, 2\}$ be the smallest index such that at least one of the 4 subgraphs containing K_c contains a negative triangle over \mathcal{R} . If no subgraph contains a negative triangle, then return that there are no negative triangles. Otherwise, recurse on one of the subgraphs containing K_c which contains a negative triangle over \mathcal{R} .

Suppose that the recursive call returns a triangle $i \in I_a, j \in J_b, k \in K_c$. By induction suppose that k is the smallest witness in K_c for (i, j) . If $c = 1$, then k must also be the smallest witness for (i, j) in the entire

Algorithm $\mathcal{A}(A, B)$: Choose a random permutation π of $[n]$.
 Permute the columns of A and rows of B according to π .
 Set $H[i, j] := A[i, 1] \odot B[1, j]$ for all i, j .
 Run *FindMinWitness*(A, B) for $2C(1 + \ln n)$ times, each time updating the values of $H[i, j]$.

FindMinWitness(A, B): Create an instance of all-pairs minimum witness triangles with node partitions I, J, K over $[n]$ such that $w(i, k) = A[i, k]$ for $i \in I, k \in K$, $w(k, j) = B[k, j]$ for $k \in K, j \in J$ and $w(i, j) = -H[i, j]$ for $i \in I, j \in J$.

Compute all-pairs minimum witness negative triangles over \mathcal{R} in this graph.

For every i, j , we either find the minimum $k(i, j)$ according to π such that $(A[i, k(i, j)] \odot B[k(i, j), j]) < H[i, j]$ (the minimum associated with (i, j) changed), or determine that $H[i, j] = \min_k A[i, k] \odot B[k, j]$.

For every witness $k(i, j)$, set $H[i, j] = (A[i, k(i, j)] \odot B[k(i, j), j])$.

Figure 1: Algorithm for reducing matrix product over \mathcal{R} to all-pairs minimum witness triangles over \mathcal{R} .

K . If $c = 2$, since there are no negative triangles with witnesses in K_1 , k must also be the smallest witness for (i, j) in the entire K .

The running time recurrence is $T'(n) \leq 8T(n/2) + T'(n/2)$. This solves to $T'(n) = 8 \sum_{i=1}^{\log n} T(n/2^i) \leq O(T(n))$ for any superlinear nondecreasing function $T(n)$. \square

Now we show that minimum witnesses for all pairs of nodes can be found efficiently, given a negative triangle detection algorithm.

Lemma 4.5 (All Pairs Minimum Witness Triangles) *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. Given a $T(n)$ algorithm for negative triangle detection over \mathcal{R} , there is an $O(T(n^{1/3})n^2)$ -time algorithm A which outputs a maximal set of IJ -disjoint negative triangles over \mathcal{R} in a tripartite graph with distinguished parts (I, J, K) , so that each of the triangles is a minimum witness triangle.*

Proof. Proceed as in Lemma 4.2, with a few changes. Partition I, J, K into n^a parts of roughly n^{1-a} nodes each, where the nodes of K are partitioned consecutively in their sorted order. Then, go through the triples of pieces I_i, J_j, K_k , but here make sure that all triples containing $K_{k'}$ with $k' < k$ are processed before those containing K_k . For each triple, find a minimum witness triangle with $x \in I_i, y \in J_j, z \in K_k$, add xyz to the running list L and remove edge (x, y) from the graph (by setting its weight to ∞). Since we are processing the triples in nondecreasing order of K and z is the minimum witness for (x, y) in K_k , it must also be the minimum witness in K . As in Lemma 4.2, the number of triangles returned is at most n^2 , and the running time is minimized at $O(n^2T(n^{1/3}))$ when $a = 2/3$. \square

Proof of Theorem 4.5. Let $C \geq 1$ be a parameter. We will analyze the algorithm \mathcal{A} shown in Figure 1.

It follows from Lemma 4.5 that the running time of \mathcal{A} is $O(n^2T(n^{1/3}) \log n)$. We will now prove that with probability at least $1 - 1/n^C$, algorithm \mathcal{A} outputs the matrix product of A and B over \mathcal{R} .

In the call of \mathcal{A} on $n \times n$ matrices A and B , $K = \{1, \dots, n\}$ is sorted according to π . Fix any $i \in I, j \in J$. Consider iterating through $k \in \{1, \dots, n\}$ in reverse order of π , and let $T_{i,j}$ be the number of times that the minimum value of $A[i, k] \odot B[k, j]$ changes before finally settling on the minimum value. We first show that $E[T_{i,j}] \leq 1 + \ln n$, using a standard backwards analysis argument.

Let Y_k be an indicator variable which is 1 if and only if $(A[i, k] \odot B[k, j]) < (A[i, k'] \odot B[k', j])$ for all $k' > k$. $E[Y_k]$ is exactly the fraction of permutations over $n - k + 1$ elements $\{k, \dots, n\}$ for which k is the first element, i.e., $E[Y_k] = ((n - k + 1) - 1)! / (n - k + 1)! = 1 / (n - k + 1)$. Hence

$$E[T_{i,j}] = \sum_{k=1}^n E[Y_k] = \sum_{k=1}^n \frac{1}{n - k + 1} \leq 1 + \ln n.$$

Analogous to the standard analysis of treaps (see [25], Problems 1.8 and 2.7), we observe that for a fixed (i, j) and for all $S \subseteq [n]$,

$$\Pr \left[\bigwedge_{k \in S} Y_k \right] \leq \prod_{k \in S} \Pr[Y_k = 1].$$

This follows from the fact that the Y_k satisfy a limited independence condition: conditioned on any setting of values for $Y_{k'}$ where $k' < k$, the probability that $Y_k = 1$ is still $1 / (n - k + 1)$. Hence a Chernoff bound applies to $T_{i,j}$, meaning that for sufficiently large $C \geq 1$ we have $\Pr[T_{i,j} > 2C(1 + \ln n)] \leq e^{-3 \ln n} = 1/n^3$.¹¹

Algorithm \mathcal{A} runs the all pairs minimum witness algorithm (*FindMinWitness*) for $2C(1 + \ln n)$ times. By the calculation above, for any fixed (i, j) , the probability that the minimum value of $A[i, k] \odot B[k, j]$ is not found is at most $1/n^3$. By the union bound, the probability that some pair (i, j) does not have its minimum computed is at most $n^2/n^3 = 1/n$. Hence with $1 - 1/n$ probability all minima are computed by \mathcal{A} . \square

5 Problems Equivalent to All-Pairs Shortest Paths

The goal of this section is to prove Theorem 1.1 from the Introduction.

Reminder of Theorem 1.1 *The following problems (with weights in $\{-M, \dots, M\} \cup \{-\infty, \infty\}$) either all have truly subcubic algorithms, or none of them do:*

1. *The all-pairs shortest paths problem on weighted digraphs (APSP).*
2. *The all-pairs shortest paths problem on undirected weighted graphs.*
3. *Detecting if a weighted graph has a triangle of negative total edge weight.*
4. *Listing up to $n^{3-\delta}$ negative triangles in an edge-weighted graph, for a fixed $\delta > 0$.*
5. *Computing the matrix product over the $(\min, +)$ -semiring.*
6. *Verifying the correctness of a matrix product over the $(\min, +)$ -semiring.*
7. *Checking whether a given matrix defines a metric.*
8. *Finding a minimum weight cycle in a graph of non-negative edge weights.*
9. *The replacement paths problem on weighted digraphs.*
10. *Finding the second shortest simple path between two nodes in a weighted digraph.*

¹¹We thank an anonymous referee for suggesting this analysis; our earlier analysis incurred an extra $\log n$ multiplicative factor.

11. Finding a maximum subarray in a given matrix.

The subcubic equivalence of problems (1), (3), (4), (5) and (6) directly follow from Theorems 4.1, 4.2, and 4.3. The rest of the equivalences are proved in the following paragraphs. Most of these equivalences use the negative triangle problem, since it is so easy to reason about.

The equivalence between problems (1) and (2) is probably folklore, but we have not seen it in the literature so we include it for completeness.

Theorem 5.1 (Undirected APSP \equiv_3 Directed APSP) *Let $\delta, c > 0$ be any constants. APSP in undirected graphs with weights in $[0, M]$ is in $T(n, M)$ time if and only if APSP in directed graphs with weights in $[-M, M]$ is in $T(n, \Theta(M))$ time.*

Proof of Theorem 5.1. Clearly, undirected APSP in graphs with nonnegative weights is a special case of directed APSP since we can replace each undirected edge by two directed edges in opposite directions. We show that a truly subcubic algorithm for undirected APSP can be used to compute the $(\min, +)$ product of two matrices in truly subcubic time, and hence directed APSP is in truly subcubic time.

Suppose that there is a truly subcubic algorithm P for undirected APSP. Let A and B be the $n \times n$ matrices whose $(\min, +)$ product we want to compute. Suppose the entries of A and B are in $[-M, M]$ ¹². Consider the edge-weighted undirected tripartite graph G with n -node partitions I, J, K such that there are no edges between I and K , and for all $i \in I, j \in J, k \in K$, (i, j) and (j, k) are edges with $w(i, j) = A[i, j] + 6M$ and $w(j, k) = B[j, k] + 6M$. Using P , compute APSP in G . Notice that all weights are nonnegative.

Any path on at least 3 edges in G has weight at least $15M$, and any path on at most 2 edges has weight at most $2 \times 7M < 15M$. Hence P will find for every two nodes $i \in I, k \in K$, the shortest path between i and k using *exactly* 2 edges, thus computing the $(\min, +)$ product of A and B . \square

Theorem 5.2 (Metricity \equiv_3 Negative Triangle) *Let $T(n, M)$ be nondecreasing. Then there is an $O(n^2) + T(O(n), O(M))$ algorithm for negative triangle in n node graphs with weights in $[-M, M]$ if and only if there is an $O(n^2) + T(O(n), O(M))$ algorithm for the metricity problem on $[n]$ such that all distances are in $[-M, M]$.*

Proof of Theorem 5.2. Given an instance D of the metricity problem, consider a complete tripartite graph G on $3n$ nodes n nodes in each of the partitions I, J, K . For any $i \in I, j \in J, k \in K$, define the edge weights to be $w(i, j) = D[i, j], w(j, k) = D[j, k]$ and $w(i, k) = -D[i, k]$. A negative triangle in G gives $i \in I, j \in J, k \in K$ so that $D[i, j] + D[j, k] - D[i, k] < 0$, i.e. $D[i, j] + D[j, k] < D[i, k]$. Hence D satisfies the triangle inequality iff there are no negative triangles in G . Checking the other properties for a metric takes $O(n^2)$ time. This shows that Metricity \leq_3 Negative Triangle.

For the opposite direction, let G be a given graph with edge weights $w : E \rightarrow \mathbb{Z}$ such that for all $e \in E, w(e) \in [-M, M]$ for some $M > 0$. Build a tripartite graph with n node partitions I, J, K and edge weights $w'(\cdot)$ so that for any $i \in I, j \in J, k \in K, w'(i, j) = 2M + w(i, j), w'(j, k) = 2M + w(j, k)$ and $w'(i, k) = 4M - w(i, k)$. For all pairs of distinct nodes a, b so that a, b are in the same partition, let $w'(a, b) = 2M$. Finally, let $w'(x, x) = 0$ for all x . Clearly, w' satisfies all requirements for a metric, except possibly the triangle inequality. For any three vertices x, y, z in the same partition $w'(x, y) + w'(y, z) = 4M > 2M = w'(x, z)$.

¹²Infinite edge weights can be replaced with suitably large finite values, WLOG.

Consider triples x, y, z of vertices so that x and y are in the same partition and z is in a different partition. We have: $w'(x, z) + w'(z, y) \geq M + M = 2M = w'(x, y)$ and $w'(x, z) - w'(y, z) \leq 2M = w'(x, y)$. Furthermore, if $i \in I, j \in J, k \in K$, $w'(i, k) + w'(k, j) \geq M + 3M \geq w(i, j)$ and $w'(i, j) + w'(j, k) \geq M + 3M \geq w(i, k)$.

Hence the only possible triples which could violate the triangle inequality are triples with $i \in I, j \in J, k \in K$, and w' is not a metric iff there exist $i \in I, j \in J, k \in K$ such that $w'(i, j) + w'(j, k) < w'(i, k)$. That is, w' is a metric if and only if $w(i, j) + w(j, k) + w(i, k) < 0$ and i, j, k is a negative triangle in G . \square

Theorem 5.3 (Minimum Cycle \equiv_3 Negative Triangle) *If there is a $T(n, M)$ algorithm for finding a minimum weight cycle in graphs on n nodes and weights in $[1, M]$ then there is a $T(n, O(M))$ algorithm for finding a minimum weight triangle in n -node graphs with weights in $[-M, M]$.*

It is known that the Minimum-Cycle problem in directed or undirected graphs can be reduced to APSP via a subcubic reduction. Hence, we get that $\text{APSP} \equiv_3 \text{Minimum-Cycle}$.

Proof. Let $G = (V, E)$ be given with $w : E \rightarrow [-M, M]$. Consider graph G' which is just G with weights $w' : E \rightarrow [7M, 9M]$ defined as $w'(e) = w(e) + 8M$. For any k and any cycle C in G with k edges, $w'(C) = 8Mk + w(C)$, and hence $7Mk \leq w'(C) \leq 9Mk$. Hence, all cycles C with ≥ 4 edges have $w'(C) \geq 28M$ and all triangles have w' weight $\leq 27M < 28M$. That is, the minimum weight cycle in G' is exactly the minimum weight triangle in G . \square

We now prove that the Maximum Subarray problem is equivalent to Negative Triangle. One direction was already proven by Tamaki and Tokuyama [60]; here we prove the other.

Theorem 5.4 (Maximum Subarray \equiv_3 Negative Triangle) *The negative triangle problem in an n node (tri-partite) graph with weights in $[-M, M]$ can be reduced in $O(n^2)$ time to the Maximum Subarray problem for an $n \times n$ matrix with entries in $[-O(M), O(M)]$.*

Proof. We are given a graph G with three parts A, B, C and edges $E \subseteq (A \times B) \cup (B \times C) \cup (A \times C)$ with weight function $w : E \rightarrow \{-W, \dots, W\}$ and we are searching for a triangle of negative sum. We will first turn G into a complete graph by adding each original non-edge (x, y) with weight $w(x, y) = 2W$. Note that no original non-edge can be completed to a negative triangle as all other edges have weight $\geq -W$. For the non-edges within A, B and C , make their edge weights slightly higher, $8W$. Now we can assume that G is a complete graph with edge weights in $\{-M, \dots, M\}$ between different partitions, and of weight $4M$ within the partitions, where $M = 2W$.

Now we will reduce the negative triangle problem to the following problem: given a complete undirected graph H with a self-loop at each node and with weights $w' : V \times V \rightarrow \{0, \dots, 101M\}$, find four nodes i, j, k, l for which $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l)$ is maximized.

H will contain the node sets A, B, C and a copy A' of A . Denote the copy in A' of node $a \in A$ with a' . H is a complete graph on these nodes. The weights w' are as follows:

- In $A \times A'$ we have weight 0 edges between two copies of the same node and an edge of weight $10M$ otherwise, i.e. $w'(a, a') = 0$ and $w'(a, x) = 10M$ for all $x \in A', x \neq a'$.
- For all edges $(a, b) \in A \times B$ in H , let their weight be $w'(a, b) = 100M + w(a, b)$ (recall, w is the weight function of G). Similarly, for all $(c, a') \in C \times A'$, let $w'(c, a') = 100M + w(c, a)$. Notice that these edges have weight between $99M$ and $101M$. We call these *heavy* edges.

- For all $(b, c) \in B \times C$, set $w'(b, c) = w(b, c)$.
- For all pairs of nodes in $A \times C$ and $A' \times B$, make their edge w' weights $50M$.
- The edges within A , A' , B , and C come from the original graph G , i.e. for each s, t in A (resp., A' , B , C), we get an edge (s, t) in A (resp., A' , B , C) of weight $4M$.
- Finally, we will add a self-loop at each node, of weight $40M$.

Notice that all edge weights w' above are nonnegative except for $w(b, c)$ for $b \in B, c \in C$, in which case they are $\geq -M$, so that all weights are $\geq -M$. Also notice that the non-heavy edges have weight $\leq 50M$.

Consider now any 4-tuple i, j, k, l of not necessarily distinct nodes in this new graph H and consider $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l)$.

If (i, j) and (k, l) aren't both heavy edges, then $w'(i, j) + w'(k, l) \leq 101M + 50M = 151M$ and $-w'(i, k) - w'(j, l) \leq 2M$ (since all edge weights are $\geq -M$). Thus in this case $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l) \leq 153M$.

On the other hand, suppose that both (i, j) and (k, l) are heavy edges. First let's consider the case when the nodes are not distinct. Then without loss of generality $i = k$, and $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l) = w'(i, j) + w'(i, l) - w'(i, i) - w'(j, l) = 160M + w(i, j) + w(i, l) - w'(j, l) \leq 163M$.

The final case is when all four nodes are distinct and both (i, j) and (k, l) are heavy edges. This can happen in four ways (up to symmetry):

- If $i \in A, j \in B, l \in C$ and $k \in A'$, then $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l) = 200M - w(i, j) - w(k, l) - w'(i, k) - w(j, l)$. This equals $200M - w(i, j) - w(j, l) - w(l, i) \geq 197M$ when k is the copy of $i \in A$ in A' , and otherwise equals $190M - w(i, j) - w(k, l) - w(j, l) \leq 193M$.
- If $i \in A, j \in B, k \in C$ and $l \in A'$, then $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l) = 100M + w(i, j) + w(k, l) \leq 102M$.
- If $i \in A, j \in B, k \in A, l \in B$, then $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l) = (200M - 8M) + w(i, j) + w(k, l) \leq 196M$. (The case $i \in C, j \in A', k \in C, l \in A'$ is symmetric.)
- If $i \in A, j \in B, l \in A, k \in B$, then $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l) = (200M - 200M) + w(i, j) + w(k, l) - w(i, k) - w(j, l) \leq 4M$. (The case $i \in C, j \in A', l \in C, k \in A'$ is symmetric.)

Thus to maximize the quantity $w'(i, j) + w'(k, l) - w'(i, k) - w'(j, l)$, we need to be in the first case above. More precisely, we get that $i \in A, j \in B, l \in C$ and $k \in A'$, and that k is the copy of i in A' , so that the maximized quantity is $200M - w(i, j) - w(j, l) - w(l, i) = 200M -$ (the minimum weight of a triangle). If there is a negative triangle in particular, we would get a value of $> 200M$. In other words, the maximum value is $> 200M$ if and only if the original graph had a negative triangle.

We'll reduce the above problem to the Maximum Subarray problem.

Let A^* be the $n \times n$ matrix with $A^*[i, j] = w'[i, j]$ for every (i, j) in the above graph (including the self-loops: $A^*[i, i] = 40M$).

Create the $n \times n$ matrix D where $D(i, j) = A^*[i, j] - A^*[i - 1, j]$ for all i, j . Then create the matrix $E[i, j] = D[i, j] - D[i, j - 1]$.

Consider a submatrix of E from the a th row to the u th row and from the b th column to the v th column. For any i ,

$$\sum_{j=b}^v E[i, j] = D[i, b] - D[i, b-1] + D[i, b+1] - D[i, b] + \dots + D[i, v] - D[i, v-1] = D[i, v] - D[i, b-1].$$

Then, $\sum_{i=a}^u \sum_{j=b}^v E[i, j] = (\sum_{i=a}^u D[i, v]) - (\sum_{i=a}^u D[i, b-1])$. Now, for any x ,

$$\sum_{i=a}^u D[i, x] = A^*[a, x] - A^*[a-1, x] + A^*[a+1, x] - A^*[a, x] + \dots + A^*[u, x] - A^*[u-1, x] = A^*[u, x] - A^*[a-1, x],$$

and so

$$\sum_{i=a}^u \sum_{j=b}^v E[i, j] = A^*[u, v] - A^*[a-1, v] - A^*[u, b-1] + A^*[a-1, b-1].$$

The Maximum Subarray problem for matrix E seeks to maximize the above quantity over all choices of a, b, u, v . The above quantity is a sum as in the four node problem and what we are maximizing is exactly what we needed in that problem. Hence, if the maximum subarray of E has sum $> 200M$ then G has a negative triangle, and otherwise G doesn't. Thus, we have a subcubic reduction from negative triangle and hence also APSP to the Maximum Subarray problem. \square

To complete the proof of Theorem 1.1, it remains to show the equivalences of Replacement Paths and Second Shortest Paths with the other problems.

5.1 Replacement Paths and Second Shortest Paths

The replacement paths and second shortest simple path problems have been known to be closely related to APSP in an informal sense. For instance, any algorithm for APSP can solve the two problems in asymptotically the same time: remove all edges from the shortest path P between s and t and compute APSP in the remaining graph. This computes the minimum weight detour for all pairs of nodes on P , and so in additional $O(n^2)$ time one can solve both the replacement paths problem, and the second shortest simple path problem. It was not clear however that the two problems cannot be solved faster than APSP. For instance, Roditty [52] took his fast approximation algorithms as evidence that the two problems might be easier than APSP. In an attempt to explain why it has been so hard to find fast algorithms, Hershberger et al. [35] showed that in the path comparison model of Karger et al. [38] the replacement paths problem needs $\Omega(m\sqrt{n})$ time. This bound does not apply to second shortest path, and is the only known lower bound for these problems.

Here we present a reduction which shows that if the second shortest simple path in a directed graph with n nodes can be found in time which is truly subcubic in n , then APSP is in truly subcubic time. Thus, the two problems are equivalent with respect to subcubic algorithms, for dense graphs. Since the second shortest simple path problem is a special case of the replacement paths problem, our result implies that for dense graphs the replacement paths problem is equivalent to APSP, with respect to subcubic algorithms.

In the next section we modify the reduction to show that if for some $m(n)$ and nondecreasing $f(n)$ there is a combinatorial algorithm which runs in $O(m(n)\sqrt{n}/f(n))$ time and computes the second shortest simple path in unweighted directed graphs, then there is an $O(n^3/f(n))$ combinatorial algorithm for triangle detection, and hence a corresponding subcubic combinatorial algorithm for BMM. This implies that if there

Figure 2: The reduction from minimum weight triangle to second shortest simple path for $n = 3$.

is no truly subcubic combinatorial algorithm for BMM, then in order to improve on the algorithm of Roditty and Zwick [55], one would need to use algebraic techniques.

Theorem 5.5 (Minimum Triangle \equiv_3 Second Shortest Simple Path) *Suppose there is a $T(n, W)$ time algorithm for computing the second shortest simple path in a weighted directed graph with n nodes and integer weights in $[0, W]$. Then there is a $T(O(n), O(nW))$ time algorithm for finding a minimum weight triangle in an n node graph and integer weights in $[-W, W]$, an $O(n^2 T(O(n^{1/3}), O(nW)) \log W)$ time algorithm for the distance product of two $n \times n$ matrices with weights in $[-W, W]$, and an $O(n^2 T(O(n^{1/3}), O(n^2 W)) \log W n)$ time algorithm for APSP in graphs with integer weights in $[-W, W]$.*

Proof. Let G be an instance of Minimum Triangle with integer weights in $[-M, M]$. Without loss of generality, G has 3 parts U, V, T with no edges within them, and the edges going from U to V , from V to T and from T to U . Furthermore, again without loss of generality, all edge weights are positive (otherwise add $M + 1$ to all edges, so that now the minimum weight triangle has weight $3(M + 1) +$ its original weight). Also without loss of generality, G contains edges between every two nodes $u_i \in U, v_j \in V$, between any two nodes $v_j \in V$ and $t_k \in T$ and between any two nodes $t_k \in T$ and $u_i \in U$ (if some edge does not appear, add it with weight $3M' + 1$ where $M' = 4M + 1$ is the current maximum edge weight in G). Note that all of these transformations increase the maximum weight by at most a constant factor. Let M'' be the new maximum edge weight of G .

Now we will reduce any instance of minimum weight triangle to one of finding the second shortest simple path. First, create a path on $n + 1$ nodes, $P = p_0 \rightarrow p_1 \rightarrow \dots \rightarrow p_n$. For every edge (p_i, p_{i+1}) in P , let it have weight 0. All other edges in the graph we will construct will be positive and hence P will be the shortest path between p_0 and p_n .

Create three parts with n nodes each, $A = \{a_1, \dots, a_n\}, B = \{b_1, \dots, b_n\}, C = \{c_1, \dots, c_n\}$ so that for each $i, j \in [n]$ there is an edge (a_i, b_j) with weight $w(u_i, v_j)$ (the weight in G), and an edge (b_i, c_j) with weight $w(v_i, t_j)$; that is, we have created a copy of G except that the edges between T and U are removed (no edges between C and A).

Let $W = 3M'' + 1$, where recall that $M'' = O(M)$ is the maximum edge weight of G . Now, for every

$j > 0$, add an edge from c_j to p_j with weight jW . All these weights are positive and at most $nW = O(Mn)$.

For every $i < n$ and any $r \in [n]$, add an edge from p_i to a_r with weight $(n - i - 1)W + w(c_{i+1}, a_r)$. Since $i < n$ and $0 < w(c_{i+1}, a_r) \leq O(M)$, these weights are also positive and at most $O(Mn)$.

An example of the full construction appears in Figure 2.

The second shortest path must have the form $p_0 \rightarrow \dots \rightarrow p_s$ followed by a path of length two from some a_i through a node in B to a node c_t in C with $t > s$, followed by an edge (c_t, p_t) and then $p_t \rightarrow \dots \rightarrow p_n$: we are looking for the shortest detour between a node p_s and a node p_t on P with $t > s$.

The weight of a detour between p_s and p_t going through nodes a_i, b_j, c_t is

$$(n - s - 1)W + w(c_{s+1}, a_i) + w(a_i, b_j) + w(b_j, c_t) + tW.$$

Claim 1 *In the graph we have constructed, any optimal detour must have $t = s + 1$.*

Proof of Claim: Clearly $t > s$. If $t \geq s + 2$, then the weight of the detour is at least

$$(n - s - 1 + s + 2)W + w(c_{s+1}, a_i) + w(a_i, b_j) + w(b_j, c_t) > (n + 1)W.$$

Consider any detour between p_s and p_{s+1} , say going through a_i, b_j, c_{s+1} . Its weight is

$$(n - s - 1 + s + 1)W + w(c_{s+1}, a_i) + w(a_i, b_j) + w(b_j, c_{s+1}) \leq nW + W = (n + 1)W,$$

since W is greater than three times the largest weight in the graph. □

Now, the detours between p_s and p_{s+1} have weight $nW + w(a_i, b_j) + w(b_j, c_{s+1}) + w(c_{s+1}, a_i)$. In particular, the shortest detour between p_s and p_{s+1} has weight nW plus the minimum weight of a triangle containing c_{s+1} . The second shortest path hence has weight exactly nW plus the minimum weight of a triangle in G . □

Since the second shortest path problem is a special case of the replacement paths problem, we have:

Corollary 5.1 *If the replacement paths problem in graphs with integer weights in $[0, M]$ is in $T(n)$ poly log M time then APSP in graphs with integer weights in $[-M, M]$ is in $O(n^2T(O(n^{1/3}))$ poly log M) time.*

Corollary 5.2 *Replacement Paths \equiv_3 APSP.*

6 Boolean Matrix Multiplication and Related Problems

In this section, we describe several applications of our techniques to the problem of finding fast practical Boolean matrix multiplication algorithms, a longstanding challenge in graph algorithms. (For more background on this problem, see the Preliminaries.)

As a direct consequence of Theorems 4.2, 4.1 and 4.3 we obtain:

Theorem 6.1 *The following either all have truly subcubic combinatorial algorithms, or none of them do:*

1. *Boolean matrix multiplication (BMM).*
2. *Detecting if a graph has a triangle.*

3. Listing up to $n^{2.99}$ triangles in a graph.
4. Verifying the correctness of a matrix product over the Boolean semiring.

Theorem 6.2 *For every constant c , the problems listed in Theorem 6.1 either all have combinatorial algorithms running in $O(n^3 / \log^c n)$ time, or none of them do.*

Another immediate corollary of Theorem 4.3 is an efficient triangle listing algorithm:

Corollary 6.1 *There is an algorithm that, given Δ and a graph G on n nodes, lists up to Δ triangles from G in time $O(\Delta^{1-\omega/3} n^\omega) \leq O(\Delta^{0.21} n^{2.38})$.*

Note when $\Delta = n^3$, one recovers the obvious $O(n^3)$ algorithm for listing all triangles, and when $\Delta = O(1)$, the runtime is the same as that of triangle detection.

6.1 Output-Sensitive BMM

Lemma 4.3 can be applied to show that in the special case of BMM, there is an improved randomized output-sensitive algorithm:

Theorem 6.3 *Let $T(n)$ be a function so that $T(n)/n$ is nondecreasing. Let $L \geq n \log n$. Suppose there is a $T(n)$ time algorithm for triangle detection in an n node graph. Then there is a randomized algorithm R running in time*

$$\tilde{O}(n^2 + L \cdot T(n^{2/3}/L^{1/6})),$$

so that R computes the Boolean product C of two given $n \times n$ matrices with high probability, provided that C contains at most L nonzero entries. When $T(n) = O(n^\Delta)$ for some $2 \leq \Delta \leq 3$, the runtime becomes $\tilde{O}(n^{2\Delta/3} L^{1-\Delta/6})$.

Proof. The algorithm uses ideas from a paper by Lingas [43, 44]. Lingas showed how to reduce, in $O(n^2 \log n)$ time, computing the Boolean matrix product of two $n \times n$ matrices to computing $O(\log n)$ Boolean matrix products of an $O(\sqrt{L}) \times n$ by an $n \times O(\sqrt{L})$ matrix and 2 output-sensitive Boolean matrix products of an $O(\sqrt{L}) \times n$ by an $n \times n$ matrix.

The conditions of Lemma 4.3 require that $L \leq n^3, L^{3/2}$ and that $n\sqrt{L} \leq L^{3/2}, n^3$. The first condition is clearly met since $L \leq n^2$ and the second condition is met since $L \geq n \log n$.

Using Lemma 4.3 we get an asymptotic runtime of

$$n^2 \log n + \log n \cdot L \cdot T(n^{1/3}) + L \cdot T(n^{2/3}/L^{1/6}).$$

Since $T(n)$ is nondecreasing and since $L \leq n^2$, we get that $T(n^{2/3}/L^{1/6}) \geq T(n^{1/3})$ and hence we can bound the runtime by $O((n^2 + L \cdot T(n^{2/3}/L^{1/6})) \log n)$.

If $T(n) = O(n^\Delta)$ for some $2 \leq \Delta \leq 3$ and $L \geq n$ we have $L \cdot (n^{2/3}/L^{1/6})^\Delta \geq n^2$. Hence the runtime is just $\tilde{O}(n^{2\Delta/3} L^{1-\Delta/6})$. \square

6.2 Second Shortest Paths and BMM

Similar to the case of APSP, we can prove a close relationship between BMM and finding the second simple shortest path between two given nodes in an *unweighted* directed graph. The relationship naturally

extends to a relationship between BMM and RPP in unweighted directed graphs. The theorem below shows that in the realm of combinatorial algorithms, Roditty and Zwick's [54, 55] algorithm for the second shortest simple path problem in unweighted directed graphs would be optimal, unless there is a truly subcubic combinatorial algorithm for BMM. Furthermore, any practical improvement of their algorithm would be interesting as it would imply a new practical BMM algorithm.

Theorem 6.4 *Suppose there exist nondecreasing functions $f(n)$ and $m(n)$ with $m(n) \geq n$, and a combinatorial algorithm which runs in $O(m(n)\sqrt{n}/f(n))$ time and computes the second shortest simple path in any given unweighted directed graph with n nodes and $m(n)$ edges. Then there is a combinatorial algorithm for triangle detection running in $O(n^3/f(n))$ time. If $f(n) = n^\varepsilon$ for some $\varepsilon > 0$, then there is a truly subcubic combinatorial algorithm for BMM.*

Proof. Suppose we are given an instance of triangle detection $G = (V, E)$ where V is identified with $[n]$. Let L be a parameter. Partition V into n/L buckets $V_b = \{bL + 1, \dots, bL + L\}$ of size L .

We will create n/L instances of the second shortest simple path problem. In instance b (for $b \in \{0, \dots, n/L - 1\}$), we will be able to check whether there is a triangle going through a node in bucket V_b .

Fix some b . First, create a path on $L + 1$ nodes, $P = p_0 \rightarrow p_1 \rightarrow \dots \rightarrow p_L$.

In our construction we will make sure that P is the shortest path from p_0 to p_L . The second shortest path would have to go from p_0 to some p_s using P , then take a detour (say of length d) to p_t with $t > s$, and then take P from p_t to p_L . The length of the second shortest path would then be

$$L - t + s + d = L + d + (s - t).$$

Create three parts, $A = \{a_1, \dots, a_n\}$, $B = \{b_1, \dots, b_n\}$, $C = \{c_1, \dots, c_L\}$ so that for each $i, j \in [n]$ there is an edge (a_i, b_j) iff $(i, j) \in E$ and for every $i \in [n]$, $j \in [L]$, there is an edge (b_i, c_j) iff $(i, bL + j) \in E$.

Now, for every $j > 0$, add a path R_j of length $2j$ from c_j to p_j , adding $2j$ new nodes.

For every $i < L$ add a path Q_i of length $2(2L - i)$, ending at some node q_i (thus adding $4L - 2i$ new nodes). The overall number of new nodes is at most $4L(L + 1)$.

For every $r \in [n]$ and $i < L$, add an edge from q_i to a_r iff $(bL + i + 1, r) \in E$.

Now, any simple path from p_0 to p_L which uses nodes from A, B or C must go through one of the paths Q_i , and hence has length at least $2(2L - L + 1) = 2(L + 1) > L + 1$. Hence P is the shortest path between p_0 and p_L .

The second shortest path must have the form $p_0 \rightarrow \dots \rightarrow p_s$ followed by a detour to p_t for $t > s$, followed by $p_t \rightarrow \dots \rightarrow p_L$. The detours between p_s and p_t look like this: take path Q_s from p_s to q_s , then a path of length 3 through some a_i through a node in B to a node c_t in C with $t > s$, and then taking path R_t to p_t . The length of the detour is

$$d_{st} = 2(2L - s) + 3 + 2t = 4L + 3 + 2(t - s).$$

The length of the full path is

$$L + d_{st} + (s - t) = 5L + 3 + (t - s).$$

Hence the closer s and t are, the shorter the path.

Now, G has a triangle $(i, j, bL + s)$ going through V_b iff there is a path with detour between p_{s-1} and p_s going through $Q_{s-1}, a_i, b_j, c_s, R_s$. Its length is $5L + 4$. For any s, t with $t \geq s + 2$, the length of the path

with detour between p_s and p_t is at least $5L + 3 + 2 > 5L + 4$. Hence the shortest that a second shortest path can be is $5L + 4$. It is exactly of this length (and goes between some p_s and p_{s+1}) iff there is a triangle going through V_b . Computing the length of the second shortest simple path then will tell us whether the original graph has a triangle going through V_b .

Each of the n/L graphs (for each setting of b) has $O(n + L^2)$ nodes and $O(n^2)$ edges. For $L = \Omega(\sqrt{n})$ the graph has $O(L^2)$ nodes and $O(n^2)$ edges.

Suppose that for some nondecreasing $m(N)$ and $f(N)$ there is an $O(m(N)\sqrt{N}/f(N))$ combinatorial algorithm for the second shortest simple path in directed unweighted graphs. Then, let L be such that $m(n + 4L(L + 1)) = O(n^2)$. One can find a triangle using a combinatorial algorithm in time

$$O(n/L \cdot (n^2L)/f(L^2)) = O(n^3/f(L^2)) \leq O(n^3/f(n)).$$

If $f(n)$ is a polynomial, then there is a truly subcubic combinatorial algorithm for BMM. \square

6.3 Two New BMM Algorithms

Our results allow us to provide two new algorithms for BMM, relying on the relationship between BMM and triangle detection.

6.3.1 Output-Sensitive Quantum BMM

In the theory of quantum computing there are many fantastic results. One of these is that a triangle in a graph on n nodes can be found using only $\tilde{O}(n^{1.3})$ queries to the graph [45]. Recently, Buhrman and Špalek [13] studied the problem of verifying and computing matrix products using a quantum algorithm, also in the query complexity setting, where the operations counted are the queries to the matrix entries. Among other nice results, their paper showed an $\tilde{O}(n^{1.5}\sqrt{L})$ -query output-sensitive algorithm for computing the Boolean matrix product of two $n \times n$ matrices, where L is the number of ones in the output matrix. Lemma 4.2 is a black box reduction which implies an improved algorithm by plugging in Magniez, Santha and Szegedy's [45] query-efficient triangle algorithm. Our results have been recently improved by Le Gall [41] and Jeffery, Kothari and Magniez [37]. The latter authors showed that the quantum query complexity of BMM is $\tilde{\Theta}(n\sqrt{L})$.

Lemma 6.1 *There is an $\tilde{O}(n^{1.3}L^{17/30})$ -query quantum algorithm for computing the Boolean matrix product of two $n \times n$ matrices, where L is the number of ones in the output matrix.*

Notice that since $L \leq n^2$, we always have $n^{1.3}L^{17/30} \ll \tilde{O}(n^{1.5}\sqrt{L})$.

Proof of Lemma 6.1. Let A and B be the given Boolean matrices. Consider a tripartite graph with partitions I, J, K so that for $i \in I$ and $j \in J$, (i, j) is an edge iff $A[i, j] = 1$, for $j \in J, k \in K$, (j, k) is an edge iff $B[j, k] = 1$, and (i, k) is an edge for all $i \in I, k \in K$. Note this graph does not need to be created explicitly; whenever the algorithm needs to query if (a, b) is an edge in the graph, it can just query A and B , and any output it has already produced. Then, in the output-sensitive part of the proof of Lemma 4.2, we can just use $T(n) = \tilde{O}(n^{1.3})$ given by the algorithm of [45]. Notice that the condition of the lemma is satisfied for $T(n) = \tilde{O}(n^{1.3})$. Hence we obtain an algorithm with quantum query complexity $\tilde{O}(n^{1.3}L^{1-1.3/3}) = \tilde{O}(n^{1.3}L^{17/30})$. \square

Using an algorithm by Lingas [44] that we also utilize in Theorem 6.3, the query complexity of the problem can be shown to be $\tilde{O}(n^{1.5}L^{1/4})$ which is an improvement for all $L \geq n^{12/19}$. We prove Theorem 1.6.

Reminder of Theorem 1.6 *There is an $\tilde{O}(\min\{n^{1.3}L^{17/30}, n^{1.5}L^{1/4}\})$ -query quantum algorithm for computing the Boolean matrix product of two $n \times n$ matrices, where L is the number of ones in the output matrix.*

Proof. Lingas [44] shows that repeating the following approach $O(\log n)$ times computes BMM with high probability, provided the number of ones in the output is $\leq L$:

1. Pick a random permutation of the rows of A and columns of B , and contract each consecutive block of $L^{1/2}$ rows of A or columns of B into a “superrow” (or “supercolumn”) which is just the componentwise OR of the corresponding rows or columns.

This part can be computed using $nL^{1/2}$ Grover searches that find a 1 among $n/L^{1/2}$ entries. This takes $nL^{1/2}\sqrt{n/L^{1/2}}$ queries altogether, resulting in a query complexity of $O(n^{1.5}L^{1/4})$.

2. Compute BMM on the resulting $L^{1/2} \times n$ by $n \times L^{1/2}$ matrices, also extracting witnesses, and then for each 1 in the output, find the lexicographically first nonzero entry in the original instance that lies in the corresponding superrow and supercolumn.

All of these operations can be performed classically. The matrix product part does not require any extra queries to the input. The rest of the computation needs at most $L(n/L^{1/2})$ extra queries to the input, since the number of nonzero entries in the collapsed instance is still at most L , and recovering a real nonzero entry for a collapsed nonzero (i, j) with witness k can be done by just going through the $n/L^{1/2}$ rows corresponding to superrow i and the $n/L^{1/2}$ columns corresponding to supercolumn j until the first entry x and first entry y are found such that $A[x, k] = B[k, y] = 1$.

□

6.4 Polynomial Preprocessing and Faster Combinatorial BMM

The divide-and-conquer ideas in our theorems are admittedly quite simple, but they are also powerful. It is evident that these ideas are useful for solving function problems via algorithms for related decision problems. These ideas can also be applied to greatly relax the conditions needed to achieve faster algorithms for the decision problems themselves. Williams [70] showed that it is possible to preprocess a graph in $O(n^{2+\varepsilon})$ time (for all $\varepsilon > 0$) such that queries of the form “*is S an independent set?*” can be answered in $O(n^2/\log^2 n)$ time. This data structure can be easily used to solve triangle detection in $O(n^3/\log^2 n)$, by simply querying the neighborhoods of each vertex. Bansal and Williams [6, 7] show that every graph can be (randomly) preprocessed in $O(n^{2+\varepsilon})$ time so that any batch of $O(\log n)$ independent set queries can be answered in $O(n^2/\log^{1.25} n)$ (deterministic) time. This implies an $O(n^3/\log^{2.25} n)$ randomized triangle detection algorithm. A major limitation in this approach to fast triangle detection is that the preprocessing time apparently must be subcubic. In fact, this subcubic requirement is the only reason why Bansal and Williams’ preprocessing algorithm needs randomization. It turns out that in fact any *polynomial* amount of preprocessing suffices:

Theorem 6.5 *Suppose there are $k, c > 0$ such that every n -node graph can be preprocessed in $O(n^k)$ time so that all subsequent batches of $\log n$ independent set queries $S_1, \dots, S_{\log n}$ can be answered together in $O(n^2/\log^c n)$ time. Then triangle detection (and hence Boolean matrix multiplication) is solvable in $O(n^3/\log^{c+1} n)$ time.*

That is, in order to attain better combinatorial algorithms for BMM, it suffices to answer independent set queries quickly with any *polynomial* amount of preprocessing. Theorem 6.5 holds for both randomized and deterministic algorithms: a deterministic preprocessing and query algorithm results in a deterministic BMM algorithm. The idea behind the proof of Theorem 6.5 resembles a method by Andersson and Thorup [5] for converting polynomial space static search data structures into linear space dynamic ones.

Proof of Theorem 6.5. Let $a = 1/(2k)$. Divide the n nodes of the graph into n^{1-a} parts, each part having at most $n^a + 1$ nodes each. For each pair i, j of parts, let $G_{i,j} = (V_{i,j}, E_{i,j})$ be the subgraph of G restricted to the nodes in parts i and j . Preprocess $G_{i,j}$ for independent set queries in $O(n^{ak})$ time. This stage takes $O(n^{2(1-a)+ak}) \leq n^{2-1/k+1/2} \leq O(n^{2.5})$ time.

To determine if G has a triangle, partition the set of nodes of G into at most $1 + n/\log n$ groups of $\log n$ nodes each. For each group $v_1, \dots, v_{\log n}$ and all pairs of indices $i, j = 1, \dots, n^{1-a}$, query $N(v_1) \cap V_{i,j}, \dots, N(v_{\log n}) \cap V_{i,j}$ for independence. If some query answers “no” then report that there is a triangle; if all queries answer “yes” over all nodes then report that there is no triangle. This stage takes $O(n/\log n \cdot n^{2(1-a)} \cdot n^{2a}/(a \log^c n)) \leq O(n^3/\log^{c+1} n)$ time. \square

Theorem 6.5 makes it easy to give derandomized versions of Bansal and Williams’ algorithms, since there are deterministic *polynomial time* algorithms for the problems they need to solve, just not subcubic ones.

Reminder of Theorem 1.4 *There is a deterministic combinatorial algorithm for BMM running in $O(n^3/\log^{2.25} n)$ time.*

Proof of Theorem 1.4. We will show that there is a deterministic combinatorial $O(n^3/\log^{2.25} n)$ time algorithm for triangle finding. By Corollary 4.1, this yields a deterministic combinatorial $O(n^3/\log^{2.25} n)$ time algorithm for BMM.

The preprocessing algorithm of Bansal and Williams (Theorem 5.1 in [6]) proceeds by finding an ε -pseudoregular partition (in the sense of Frieze and Kannan [32]) in $O(n^2)$ randomized time. The resulting independent set query algorithm answers $O(\log n)$ independent set queries in $O(n^2/\log^{1.25} n)$ time and is completely deterministic. Alon and Naor [4] give a deterministic polynomial time algorithm for computing an ε -pseudoregular partition, which works for all $\varepsilon \leq c/\sqrt{\log n}$ for a fixed constant $c > 0$. By replacing the randomized preprocessing with the algorithm of Alon and Naor, and applying the reduction of Theorem 6.5, the algorithm is obtained. \square

Using the connection between negative triangle and APSP, we can identify a natural query problem on weighted graphs for which nontrivial solutions would give faster APSP algorithms. On a graph with an edge weight function $c : E \rightarrow \mathbb{Z}$, define a *price query* to be an assignment of node weights $p : V \rightarrow \mathbb{Z}$, where the answer to a query is *yes* if and only if there is an edge $(u, v) \in E$ such that $p(u) + p(v) > c(u, v)$. Intuitively, think of $p(v)$ as a price on node v , the edge weight $c(u, v)$ as the cost of producing both u and v , and we wish to find for a given list of prices if there is any edge we are willing to “sell” at those prices.

Reminder of Theorem 1.5 *Suppose there is a constant $k > 0$ and a function $f(n)$ such that every n -node edge-weighted graph can be preprocessed in $O(n^k)$ time so that any subsequent price query can be answered in $O(n^2/f(n))$ time. Then, the negative triangle detection problem is solvable in $O(n^3/f(n^{1/(2k)}))$ time.*

If the hypothesis of the theorem is true for $f(n) = 2^{\Omega(\log^\delta(n))}$ for some $\delta > 0$, then our results also imply that APSP is solvable in $n^3/2^{\Omega(\log^\delta(n))}$ time (by Theorem 4.5). To some, the contrapositive of Theorem 1.5 may be more interesting: assuming that APSP needs $\Omega(n^3/f(n))$ time, there is a *super-polynomial* time lower bound on the preprocessing time needed for efficiently answering price queries.

Proof. (Sketch) In Theorem 6.5 (and [6]), faster solutions for the independent set query problem (with polynomial preprocessing) are shown to imply faster triangle detection. The key observation here is that the price query problem is the analogue of the independent set query problem, for finding negative weight triangles. More precisely, given any weighted graph on nodes $\{v_1, \dots, v_n\}$ and weight function w , the existence of a negative weight triangle can be determined from n price queries q_1, \dots, q_n , one for each node, where the price on node v_j in q_i is $-w(v_i, v_j)$. Obtaining a *yes* answer to the price query q_i is equivalent to the existence of an edge (v_j, v_k) such that $(-w(v_i, v_j)) + (-w(v_i, v_k)) > w(v_j, v_k)$, which is equivalent to the existence of a negative weight triangle through v_i . This already shows that a subcubic time preprocessing algorithm for answering price queries in subquadratic time implies subcubic-time negative triangle detection. By applying the same divide-and-conquer preprocessing idea as Theorem 6.5, we find that any polynomial-time algorithm for preprocessing graphs for price queries implies a subcubic-time algorithm for preprocessing graphs for price queries (with asymptotically the same query time). \square

7 A Simplified View of All-Pairs Path Problems and Their Matrix Products

In this section we consider various algebraic structures other than the $(\min, +)$ and Boolean semirings. We relate their matrix products and respective triangle problems, showing how several prior results in the area can be simplified in a uniform way.

Existence-Dominance. The dominance product of two integer matrices A and B is the integer matrix C such that $C[i, j]$ is the number of indices k such that $A[i, k] \leq B[k, j]$. The dominance product was first studied by Matoušek [46] who showed that for $n \times n$ matrices it is computable in $O(n^{(3+\omega)/2})$. The existence-dominance product of two integer matrices A and B is the Boolean matrix C such that $C[i, j] = 0$ iff there exists a k such that $A[i, k] \leq B[k, j]$. This product was used in the design of the first truly subcubic algorithm for the minimum node-weighted triangle problem [63, 67]. Although the existence-dominance product seems easier than the dominance product, the best known algorithm for it actually computes the dominance product.

The existence-dominance product is defined over the (\min, \odot) structure for which $R = \mathbb{Z} \cup \{-\infty, \infty\}$ and $a \odot b = 0$ if $a \leq b$ and $a \odot b = 1$ otherwise. The corresponding negative triangle problem, the *dominance triangle* problem, is defined on a tripartite graph with parts I, J, K . The edges between I and J are unweighted, and the rest of the edges in the graph have real weights. The goal is to find a triangle $i, j, k \in I \times J \times K$ such that $w(i, k) \leq w(k, j)$.

Minimum Edge Witness. The minimum edge witness product is defined over a restriction of the (\min, \odot) structure over $R = \mathbb{Z} \cup \{\infty, -\infty\}$, where $\odot = \times$ is integer multiplication. For an integer matrix A and a $\{0, 1\}$ matrix B , the (i, j) entry of the minimum edge witness product C of A and B is equal to $\min_k (A[i, k] \times B[k, j])$. This product is important as it is in truly subcubic time iff APSP on node-weighted graphs is in truly subcubic time. Chan [15, 16] used this relation to obtain the first truly subcubic runtime for node-weighted APSP.

The negative triangle problem corresponding to the minimum edge witness product is again the dominance triangle problem. Hence, by Theorem 4.2 we can conclude that a truly subcubic algorithm for the dominance triangle problem (such as Matoušek’s algorithm for the dominance product) implies truly subcubic node-weighted APSP. That is, we get an alternative subcubic algorithm for node-weighted APSP as a byproduct, although it is a bit slower than the best known. To obtain his algorithm for node-weighted APSP, Chan [16] gave a completely new algorithm for minimum edge witness product with exactly the same runtime

as Matoušek’s dominance product algorithm.

(Min- \leq). The (\min, \leq) structure is defined over $R = \mathbb{Z} \cup \{\infty, -\infty\}$, where the binary operation \leq on input a, b returns b if $a \leq b$ and ∞ otherwise. The first author showed [62, 68] that the (\min, \leq) matrix product is in truly subcubic time iff the all pairs minimum nondecreasing paths problem (also called *earliest arrivals*) is in truly subcubic time. The first truly subcubic runtime for the product, $O(n^{2+\omega/3})$, was obtained by the present authors and R. Yuster [65, 66]. The techniques of Duan and Pettie [24] also imply an $O(n^{(3+\omega)/2})$ algorithm.

The negative triangle problem over (\min, \leq) is the following *nondecreasing triangle* problem: given a tripartite graph with partitions I, J, K and real edge weights, find a triangle $i \in I, j \in J, k \in K$ such that $w(i, k) \leq w(k, j) \leq w(i, j)$.

Both known algorithms for this problem follow from the algorithms for (\min, \leq) -product [65, 66, 24] and are somewhat involved. Below we give a simpler $O(n^{3/2} \sqrt{T(n)})$ algorithm, where $T(n)$ is the best runtime for finding a triangle in an *unweighted* graph. If matrix multiplication is used, the runtime is the same as in Duan-Pettie’s algorithm, $O(n^{(3+\omega)/2})$. Furthermore, the algorithm can actually be applied $O(\log n)$ times to obtain another $\tilde{O}(n^{(3+\omega)/2})$ algorithm for the (\min, \leq) -product.

Theorem 7.1 (Nondecreasing Triangle \leq_3 Triangle) *If a triangle in an unweighted graph can be found in $T(n)$ time, then a nondecreasing triangle can be found in $O(n^{3/2} \sqrt{T(O(n))})$ time, and (\min, \leq) product is in $O(n^{3/2} \sqrt{T(O(n))} \log n)$ time.*

Proof. We are given a weighted tripartite graph with partitions I, J, K and are looking for a triangle $i \in I, j \in J, k \in K$ such that $w(i, k) \leq w(k, j) \leq w(i, j)$.

Begin by sorting all the edges in the graph, breaking ties in the following way: edges from $I \times J$ are considered bigger than edges from $K \times J$ of the same weight which are considered bigger than edges from $I \times K$ of the same weight; within $I \times J$ or $J \times K$ or $I \times K$ equal edges are arranged arbitrarily.

Let t be a parameter. For every vertex v in J or K , consider the sorted order of edges incident to v and partition it into at most n/t buckets of t consecutive edges each and at most one bucket with $\leq t$; let B_{vb} denote the b -th bucket for node v . For each edge (x, v) such that v is in J or K and (x, v) is in B_{vb} , go through all edges (v, y) in B_{vb} and check whether x, v, y forms a nondecreasing triangle. This takes $O(n^2 t)$ time.

Partition the edges of the graph by taking $O(n/t)$ consecutive groups of $\leq nt$ edges in the sorted order of all edges. Let G_g denote the g -th such group. For each g , consider all buckets B_{vb} of vertices v in J or K such that there is some edge $(v, x) \in B_{vb} \cap G_g$. There can be at most $4n$ such buckets: there are at most $n + nt/t = 2n$ buckets completely contained in G_g and at most $2n$ straddling G_g —at most one per vertex per group boundary.

Create a tripartite graph H_g for each g as follows. H_g has partitions H_g^I, H_g^J and H_g^K . H_g^I has a node for each $i \in I$. For $S \in \{J, K\}$, H_g^S has a node for each node bucket B_{vb} such that $B_{vb} \cap G_g \neq \emptyset$ and $v \in S$. Therefore H_g has $\leq 9n$ nodes.

The edges of H_g are as follows. For all $B_{jb} \in H_g^J$ and $B_{kb'} \in H_g^K$, $(B_{jb}, B_{kb'})$ is an edge if (j, k) is an edge and it is in $B_{jb} \cap B_{kb'}$. For $i \in H_g^I$ and $B_{jb} \in H_g^J$, (i, B_{jb}) is an edge in H_g iff $(i, j) \in E$ and there is a bucket $b' < b$ such that $(i, j) \in B_{jb'}$. For $i \in H_g^I$ and $B_{kb} \in H_g^K$, (i, B_{kb}) is an edge in H_g iff $(i, k) \in E$ and there is a bucket $b' > b$ such that $(i, k) \in B_{kb'}$.

Any triangle $i, B_{jb}, B_{k,b'}$ in H_g corresponds to a nondecreasing triangle i, j, k in G . If a nondecreasing triangle i, j, k of G is not contained in any H_g , then for some b either both (i, j) and (j, k) are in B_{jb} or both (i, k) and (j, k) are in B_{kb} , both cases of which are already handled.

The runtime is $O(n^2t + T(9n) \cdot n/t)$. Setting $t = \sqrt{T(9n)/n}$, the time becomes $O(n^{3/2}\sqrt{T(9n)})$. \square

Min-Max. The subtropical semiring (\min, \max) is defined over $R = \mathbb{Z} \cup \{\infty, -\infty\}$. The (\min, \max) matrix product was used by the present authors and R. Yuster [66] to show that the all pairs bottleneck paths problem is in truly subcubic time. The current best algorithm for the problem runs in $O(n^{(3+\omega)/2})$ time by Duan and Pettie [24]. The (\min, \max) product is an important operation in fuzzy logic, where it is known as the *composition of relations* ([26], pp.73).

The negative triangle problem over (\min, \max) is the following *IJ-bounded triangle* problem. Given a tripartite graph with partitions I, J, K and real weights on the edges, find a triangle $i \in I, j \in J, k \in K$ such that both $w(i, k) \leq w(i, j)$ and $w(j, k) \leq w(i, j)$, i.e. the largest triangle edge is in $I \times J$. We note that any algorithm for the nondecreasing triangle problem also solves the *IJ-bounded triangle* problem: any *IJ-bounded triangle* appears as a nondecreasing triangle either in the given graph, or in the graph with partitions J and K swapped. Hence a corollary to Theorem 7.1 is that an *IJ-bounded triangle* can be found in $O(n^{3/2}\sqrt{T(n)})$ time, where $T(n)$ is the runtime of a triangle detection algorithm for unweighted graphs.

8 Extension to 3SUM

Finally, we describe an application of the ideas in this paper to the 3SUM problem: given three lists A, B, C of n integers each, are there $a \in A, b \in B$, and $c \in C$ such that $a + b + c = 0$? The All-Integers-3SUM problem looks like a strict generalization of 3SUM: given three lists A, B, C of n integers each, output the list of *all* integers $a \in A$ such that there exist $b \in B, c \in C$ with $a + b + c = 0$. We can prove a subquadratic equivalence between 3SUM and All-Integers-3SUM.

Theorem 8.1 (All-Integers 3SUM \equiv_2 3SUM) *All-Ints 3SUM is in truly subquadratic time iff 3SUM is in truly subquadratic time.*

Proof. One direction is obvious. For the other, we use a randomized hashing scheme proposed by Dietzfelbinger [23] and used by Baran, Demaine and Patrascu [8], which maps each distinct integer to one of \sqrt{n} buckets.¹³ For each $i \in [\sqrt{n}]$, let A_i, B_i , and C_i be the sets containing the elements hashed to bucket i . The hashing scheme has two nice properties:

1. for every pair of buckets A_i and B_j there are two buckets $C_{k_{ij0}}$ and $C_{k_{ij1}}$ (which can be located in $O(1)$ time given i, j) such that if $a \in A_i$ and $b \in B_j$, then if $a + b \in C$ then $a + b$ is in either $C_{k_{ij0}}$ or $C_{k_{ij1}}$,
2. the total number of elements which are mapped to buckets containing at least $3\sqrt{n}$ elements is $O(\sqrt{n})$ in expectation.

After applying the hash function to all elements, we process all elements that get mapped to large buckets (those with size $> 3\sqrt{n}$). Without loss of generality, suppose such an element is $a \in A$. Then for all elements $b \in B$, we check whether $a + b \in C$. This takes $O(n^{1.5})$ time overall in expectation.

¹³The scheme performs multiplications with a random number and some bit shifts, hence we require that these operations are not too costly. We can ensure this by first mapping the numbers down to $O(\log n)$ bits, e.g., by computing modulo some sufficiently large $\Theta(\log n)$ bit prime.

The remaining buckets A_i, B_i, C_i for all $i \in [\sqrt{n}]$ contain $O(\sqrt{n})$ elements each. In particular, we have reduced the original problem to $2n$ subinstances of 3SUM $((A_i, B_j, C_{k_{ijb}})$ for $b = 0, 1$).

Now for each of these $2n$ subinstances in turn, call the detection algorithm. We can assume WLOG that the detection algorithm actually returns a triple $a \in A_i, b \in B_j, c \in C_{k_{ijb}}$ such that $a + b + c = 0$ (if it does not, we can recover a, b, c using a simple self-reduction). For each triple (a, b, c) detected, remove a from A_i and record that a is in a 3SUM. Try to find a new 3SUM in the subinstance. When the current subinstance contains no more solutions, move to the next subinstance.

Assuming that there is an $O(n^{2-\varepsilon})$ time algorithm for 3SUM, the running time from this portion of the reduction becomes asymptotically

$$(n + 2n) \cdot (\sqrt{n})^{2-\varepsilon} = O(n^{2-\varepsilon/2}).$$

Therefore All-Ints-3SUM can be solved in $O(n^{1.5} + n^{2-\varepsilon/2})$ time. □

Theorem 8.1 shows how subquadratic algorithms for 3SUM actually imply subquadratic algorithms for an apparently harder function problem (detecting 3-SUM solutions for all integers in the set). This should be contrasted with Patrascu’s surprising subquadratic equivalence between 3SUM and the apparently *easier* function problem Convolution-3SUM discussed in Section 2.1. Both reductions use the same hash family, but they are applied in different ways.

9 Conclusion

We have explored a new notion of reducibility which preserves truly subcubic runtimes. Our main contributions are *subcubic reductions* from important function problems (such as all-pairs paths and matrix products) to important decision problems (such as triangle detection and product verification), showing that subcubic algorithms for the latter entail subcubic algorithms for the former. We have shown that these reductions and the ideas behind them have many interesting consequences.

We conclude with three open questions arising from this work:

1. *Does $O(n^{3-\delta})$ negative triangle detection imply $O(n^{3-\delta})$ matrix product (over any \mathcal{R})?* Note we can currently show that $O(n^{3-\delta})$ negative triangle implies $O(n^{3-\delta/3})$ matrix product.
2. *Does a truly subquadratic algorithm for 3SUM imply truly subcubic APSP?* It is quite possible that truly subquadratic 3SUM implies truly subcubic negative triangle, which would answer the question.
3. *Is there a truly subcubic algorithm for minimum edge-weight triangle?* Although this question has been asked in prior work, clearly it takes on much greater importance now that we know it is equivalent to asking for a variety of subcubic algorithms for several fundamental path problems. We feel that if there are truly subcubic algorithms for APSP, they will likely be found by studying the “simpler” minimum edge-weight triangle problem.

Acknowledgments. The authors thank the anonymous referees for their insightful and detailed comments, and Laszlo Kozma for suggesting to study the Maximum Subarray problem in this context.

References

- [1] A. Abboud, F. Grandoni, and V. V. Williams. Subcubic equivalences between graph centrality problems, APSP and diameter. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1681–1697, 2015.
- [2] N. Alon. Personal communication. 2009.
- [3] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.
- [4] N. Alon and A. Naor. Approximating the cut-norm via Grothendieck’s inequality. *SIAM J. Computing*, 35:787–803, 2006.
- [5] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.
- [6] N. Bansal and R. Williams. Regularity lemmas and combinatorial algorithms. In *Proc. FOCS*, pages 745–754, 2009.
- [7] N. Bansal and R. Williams. Regularity lemmas and combinatorial algorithms. *Theory of Computing*, 8(1):69–94, 2012.
- [8] I. Baran, E. Demaine, and M. Patrascu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2008.
- [9] J. Bentley. Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873, Sept. 1984.
- [10] A. Bernstein. A nearly optimal algorithm for approximating replacement paths and k shortest simple paths in general graphs. In *Proc. SODA*, pages 742–755, 2010.
- [11] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.
- [12] J. Brickell, I. Dhillon, S. Sra, and J. Tropp. The metric nearness problem. *SIAM J. Matrix Anal. Appl.*, 30(1):375–396, 2008.
- [13] H. Buhrman and R. Špalek. Quantum verification of matrix products. In *Proc. SODA*, pages 880–889, 2006.
- [14] T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete & Computational Geometry*, 22(4):547–567, 1999.
- [15] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. STOC*, pages 590–598, 2007.
- [16] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39(5):2075–2089, 2010.
- [17] T. M. Chan. Speeding up the Four Russians algorithm by about one more logarithmic factor. In *Proc. SODA*, pages 212–217, 2015.

- [18] T. M. Chan and R. Williams. Deterministic apsp, orthogonal vectors, and more: Quickly derandomizing razborov-smolensky. In *Proc. SODA*, pages 1246–1255, 2016.
- [19] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251–280, 1990.
- [20] A. Czumaj and A. Lingas. Finding a heaviest triangle is not harder than matrix multiplication. In *Proc. SODA*, pages 986–994, 2007.
- [21] A. Czumaj and A. Lingas. Finding a heaviest vertex-weighted triangle is not harder than matrix multiplication. *SIAM J. Comput.*, 39(2):431–444, 2009.
- [22] A. Davie and A. J. Stothers. Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh, Section: A Mathematics*, 143:351–369, 4 2013.
- [23] M. Dietzfelbinger. Universal hashing and k -wise independent random variables via integer arithmetic without primes. In *Proc. STACS*, pages 569–580, 1996.
- [24] R. Duan and S. Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *Proc. SODA*, pages 384–391, 2009.
- [25] D. P. Dubhashi and A. Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.
- [26] D. Dubois and H. Prade. Fuzzy sets and systems: Theory and applications. *Academic Press*, 1980.
- [27] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.
- [28] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *Proc. FOCS*, pages 129–131, 1971.
- [29] R. W. Floyd. Algorithm 97: shortest path. *Comm. ACM*, 5:345, 1962.
- [30] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.
- [31] R. Freivalds. Probabilistic machines can use less running time. *IFIP Congress*, pages 839–842, 1977.
- [32] A. M. Frieze and R. Kannan. Quick approximation to matrices and applications. *Combinatorica*, 19(2):175–220, 1999.
- [33] A. Gajentaan and M. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry*, 5(3):165–185, 1995.
- [34] F. L. Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC*, pages 296–303, 2014.
- [35] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. *ACM TALG*, 3(1):5, 2007.
- [36] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7(4):413–423, 1978.

- [37] S. Jeffery, R. Kothari, and F. Magniez. Improving quantum query complexity of boolean matrix multiplication using graph collision. In *Proc. ICALP*, pages 522–532, 2012.
- [38] D. Karger, D. Koller, and S. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Computing*, 22(6):1199–1217, 1993.
- [39] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.
- [40] E. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401–405, 1971/72.
- [41] F. Le Gall. Improved output-sensitive quantum algorithms for Boolean matrix multiplication. In *Proc. SODA*, pages 1464–1476, 2012.
- [42] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.
- [43] A. Lingas. A fast output-sensitive algorithm for Boolean matrix multiplication. In *Proc. ESA*, pages 408–419, 2009.
- [44] A. Lingas. A fast output-sensitive algorithm for boolean matrix multiplication. *Algorithmica*, 61(1):36–50, 2011.
- [45] F. Magniez, M. Santha, and M. Szegedy. Quantum algorithms for the triangle problem. In *Proc. SODA*, pages 1109–1117, 2005.
- [46] J. Matousek. Computing dominances in E^n . *Information Processing Letters*, 38(5):277–278, 1991.
- [47] J. I. Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971.
- [48] V. Y. Pan. Strassen’s algorithm is not optimal; trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Proc. FOCS*, pages 166–176, 1978.
- [49] V. Y. Pan. New fast algorithms for matrix operations. *SIAM J. Comput.*, 9(2):321–342, 1980.
- [50] M. Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proc. STOC*, pages 603–610, 2010.
- [51] L. Roditty. On the K -simple shortest paths problem in weighted directed graphs. In *Proc. SODA*, pages 920–928, 2007.
- [52] L. Roditty. On the k shortest simple paths problem in weighted directed graphs. *SIAM J. Comput.*, 39(6):2363–2376, 2010.
- [53] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.
- [54] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *Proc. ICALP*, pages 249–260, 2005.

- [55] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Trans. Algorithms*, 8(4):33, 2012.
- [56] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. FOCS*, pages 605–614, 1999.
- [57] J. P. Spinrad. Efficient graph representations. *Fields Institute Monographs*, 19, 2003.
- [58] A. Stothers. On the complexity of matrix multiplication. *Ph.D. Thesis, U. Edinburgh*, 2010.
- [59] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.
- [60] H. Tamaki and T. Tokuyama. Algorithms for the maximum subarray problem based on matrix multiplication. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '98*, pages 446–452, 1998.
- [61] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315, 1975.
- [62] V. Vassilevska. Nondecreasing paths in weighted graphs, or: how to optimally read a train schedule. In *Proc. SODA*, pages 465–472, 2008.
- [63] V. Vassilevska and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *Proc. STOC*, pages 225–231, 2006.
- [64] V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H -subgraph in real weighted graphs and related problems. In *Proc. ICALP*, pages 262–273, 2006.
- [65] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proc. STOC*, pages 585–589, 2007.
- [66] V. Vassilevska, R. Williams, and R. Yuster. All pairs bottleneck paths and max-min matrix products in truly subcubic time. *Theory of Computing*, 5(1):173–189, 2009.
- [67] V. Vassilevska, R. Williams, and R. Yuster. Finding heaviest H -subgraphs in real weighted graphs, with applications. *ACM Trans. Algorithms*, 6(3), 2010.
- [68] V. Vassilevska Williams. Nondecreasing paths in a weighted graph or: How to optimally read a train schedule. *ACM Trans. Algorithms*, 6(4), 2010.
- [69] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.
- [70] R. Williams. Matrix-vector multiplication in sub-quadratic time (some preprocessing required). In *Proc. SODA*, pages 995–1001, 2007.
- [71] R. Williams. Faster all-pairs shortest paths via circuit complexity. In *Proc. STOC*, pages 664–673, 2014.
- [72] V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898, 2012.

- [73] G. J. Woeginger. Open problems around exact algorithms. *Discrete Applied Mathematics*, 156(3):397 – 405, 2008.
- [74] J. Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1970/71.
- [75] H. Yu. An improved combinatorial algorithm for boolean matrix multiplication. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 1094–1105, 2015.
- [76] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. *Inf. Proc. Letters*, 4:155–156, 1976.
- [77] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *JACM*, 49(3):289–317, 2002.