Replacement Paths via Fast Matrix Multiplication

by Raphael Yuster and Oren Weimann

Dominik Häner

April 2018

1 Introduction

This paper[1] shows us a method to solve the replacement paths problem by using fast matrix multiplication. This problem is defined as follows:

Definition 1.1. Replacement Paths: Given a graph G and the shortest path P from a vertex s to a vertex t, a replacement paths algorithm should find the shortest s-to-t path that avoids e, for every edge $e \in P$.

So every edge of the shortest path is individually removed and the new shortest path should be found. Naively, this can be solved by computing the shortest s-to-t path on $G \setminus e$, for every e on the replacement path.

This problem can be expanded to that of *All Pairs Replacement Paths*. Here, the goal is finding the replacement paths for every pair of vertices in the graph. The paper does this by constructing a distance sensitivity oracle, using effectively the same steps as the solution of the replacement paths problem, so we will focus on just that.

1.1 Motivation

The paper's goal is to present an algorithm that solves the replacement paths problem on directed, possibly negatively weighted graphs with no negative cycles. It does so by probabilistically constructing several smaller sub-graphs. Several operations are then performed on these sub-graphs, notably faster than if they had been performed on the initial graph. The information gained from all these operations can then be combined to construct small, representative graphs on which we can find the shortest replacement paths with a very high probability.

The paper is noteworthy as at the time of publication, no solution to the replacement paths problem, other than the naive one, was known for directed, weighted graphs. The naive solution runs in $O(mn+n^2 \log n)$. The paper's solution runs in $\tilde{O}(Mn^{1+\frac{2}{3}\omega})$. Here M is the maximum magnitude of an edge-weight, and ω is the fast matrix multiplication exponent. It is an improvement over the naive algorithm in the case that $m > n^{\frac{2}{3}\omega} = n^{1.584}$ and M is a small constant, so sufficiently dense graphs with small edge-weights. The algorithm is however not really practical. This is because it is based on fast matrix multiplication, which itself does not (yet) have a practical implementation. Since the algorithm is more like a proof of concept, this report will focus on the ideas behind the constructions used to solve the problem, rather than prove the time-bounds achieved by them.

2 High level examination

Before going into the different specific constructions, I introduce the algorithm with an intuitive explanation of what it does. Generally speaking, we are probabilistically shrinking the amount of edges to work over in one step, and the amount of vertices to work over in another, effectively lowering the time bound.

We are given a graph $G = (V, E)$, two vertices s and t, and the shortest path $P = \{e_1, ..., e_k\}$ between s and t. We would like to find the shortest paths P_i, which avoid edge e_i on P for every $i = \{1, ..., k\}$. This is done in three steps.

In the first step we generate several sub-graphs $G_j = (V_j, E_j)$ by removing individual edges with a certain probability, so that $E_j \subset E$. For each edge $e_i \in P$, we then have a set of sub-graphs F_{e_i}, which do not include it. We generate enough sub-graphs so that every edge of P_i should be present in some $G_j \in F_{e_i}$. The number of sub-graphs generated should be lower than the number of edges on P, so we will have captured all replacement paths in a smaller number of graphs than the naive approach would.

In the second step, we select a subset of vertices $B \in V$, which must include s and t, and produce a matrix B_j per G_j. Every B_j has an entry for every vertex in B, and the values are the shortest distances between these vertices over the respective G_j. The idea is that every shortest path P_i (and effectively all paths with a start and end point in B), can be composed of overlapping B-to-B paths.

We then finally combine these matrices to construct the dense distance graph G_B^i. This graph should capture the shortest B-to-B distances over all sub-graphs in F_{e_i}. A s-to-t shortest path is then computed on this consider-

ably smaller graph G_B^i.

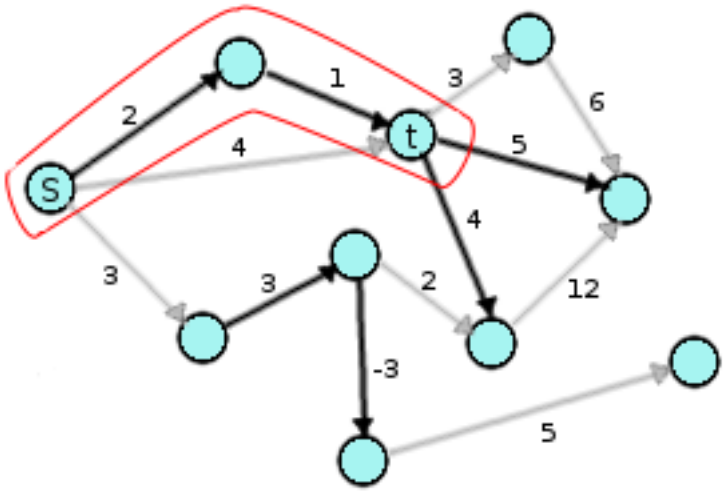
2.1 Step 1

To start, we generate a number of these sub-graphs G_j. Our goal is that every sub-path of length $n^{1-\alpha}$ lies completely within one of these sub-graphs. Here, α is a chosen constant and $0 < \alpha < 1$. The sub-graphs are computed by individually removing every edge of the graph with probability $n^{\alpha-1}$.

To intuitively understand the relevance of these values, consider a set of $\frac{1}{P} = n^{1-\alpha}$ edges. Since they are all individually randomly removed (i.e. by Bernoulli trials), all of them survive with a constant probability $(1-P)^{\frac{1}{P}} \approx \frac{1}{e}$. So all our sub-paths survive in every G_j with constant probability, which is taken into account in all other constructions.

We now have a number of sub-graphs G_j. Let us consider just one replacement shortest path P_e, which avoids edge e. The set of sub-graphs relevant to finding P_e, F_e, is the set of sub-graphs G_j, which do not contain the edge e. To make the earlier goal more specific, every sub-path of P_e of length $n^{1-\alpha}$ (henceforth an *interval*) should survive in some $G_j \in F_e$. This constraint informs the number of sub-graphs we need to generate, and will be explored later.

Figure 1: Example of a shortest path captured in a single interval.



If P_e is no longer than an interval, we will likely already have captured it in one of the sub-graphs, as represented in Figure 1. In this case it we could simply run a shortest path algorithm on the sub-graphs. We however clearly

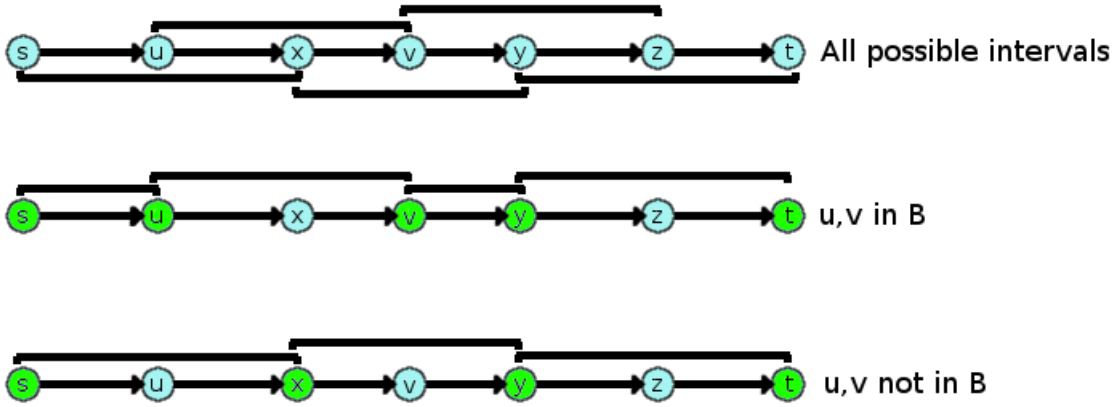
also have to handle the case where P_e is not a single interval, but rather a composition of many.

Since it is unlikely that all intervals of P_e survived in a single sub-graph (in which case the entire path would be captured by it anyway), we need a way to combine intervals from different sub-graphs. Intuitively, we could find the shortest paths over all vertices using only edges in F_e. If this were our approach, we would be better off just using the naive solution. Instead, we want to reduce the number of vertices handled by the shortest-path algorithm we will finally use. To do so we select a random subset of vertices B.

2.2 Step 2

The goal for this subset, is that shortest replacement path can be decomposed into disjoint intervals whose endpoints are both in B. Notably, vertices s and t must be in B. If every interval contains its endpoints in B, then all intervals on the shortest-path P_e will overlap in B, and thus P_e can be composed of B-to-B paths.

Figure 2: The effect of different selections of B on a shortest path



Let us consider two vertices u and v, both on P_e. If u and v are both not in B, then some vertex x, the endpoint of another interval that overlaps the (u, v) interval will be (as at least one vertex per interval should be in B). So the shortest s-to-t path could then, as shown in Figure 2 for intervals of three nodes, for example, be composed of some intervals $(s, x), (x, y), (y, t)$. Conversely, if u and v were in B but no vertex like x is, then it would be $(s, u), (u, v), (v, y), (y, t)$, or something analogous instead. We see that in both selections of B, B-to-B paths can be captured by overlapping intervals.

Given this overlap, and that s and t are in B, every interval on P_e will be captured by some overlapping intervals in B.

We randomly select B with a specific size $|B|$, where $|B|$ is large enough that the above goal is met with high probability. Given B, we then want to compute the matrices B_j, which capture the B-to-B distances over the respective G_j. This is done by finding the *distance product* of a matrix of the distances from B to all vertices in G_j, and a matrix with the distances of all vertices in G_j to all vertices in B. The exact definition of the distance product as well as the process of creating it are explored in another section.

2.3 Step 3

We now have a set of matrices B_j, each representing the shortest paths between two vertices in B over the respective sub-graph G_j. We use these to construct the dense distance graphs G_B^i. Every G_B^i has the vertex set B, and an edge set E' with edges between all vertices B, the weight of an edge (u, v) is the shortest u-to-v path in all $G_j \in F_{e_i}$ (so in all sub-graphs without edge e_i). To find a weight, we simply examine the (u, v) entry in every B_j where the respective $G_j \in F_{e_i}$ and choose the smallest one.

Now we find the shortest s-to-t path on every G_B^i, and these are with high probability the replacement paths P_i. We could do this with some algorithm which handles negative edge-weights, but it is more efficient to apply a cost function making all edge weights of G_B^i positive, and then run Dijkstra's algorithm on it. The details of this cost function are explained in a later section.

2.4 Recap

So to recap: We begin by generating several sub-graphs $G_j \subset G$ by randomly removing some edges per G_j. We then choose a random subset of vertices B which should contain the endpoints of every interval. We compute the matrices B_j, which capture the shortest distances between vertices in B over the respective sub-graph G_j. For every edge e_i on the initial shortest path P, we consider all B_js where $G_j \in F_{e_i}$ to compute the graph G_B^i, which captures the shortest paths between vertices in B over all F_{e_i}. Finally, we adjust the weights to be non-negative, and compute the shortest s-to-t path on every G_B^i.

2.5 All pairs shortest paths

Before I go into more depth on some of the constructions, it is worth mentioning that this algorithm can also be extended to create a *distance sensitivity oracle*, which takes a pair of vertices (u, v), and an edge e as a query, and returns the length of the shortest u-to-v path that avoids e.

Creating the oracle is effectively constructing all B_js. The query phase then involves us adding u and v to all B_js if they aren't already present, and then computing G_B^e and the shortest path on it.

This can further be expanded to avoid multiple edges. To do so, all of the probabilistic bounds and the number of sub-graphs generated have to change to accommodate the shorter average surviving intervals. As my focus lies on the core of the replacement paths algorithm, the details of both these expansions are being left out of this report, and can be read up in the original paper.

3 Selection of details of the algorithm

Following are some of the details I deem the most relevant, that were omitted in the initial overview of the algorithm. The probabilistic bounds and closely related time bounds are mentioned briefly, but the construction of the distance products used to compute the graphs G_B^i, and the construction of these graphs themselves are the focus.

3.1 Computing the distance product

Definition 3.1. Distance Product: A distance product $D = A \star B$ of two matrices A and C is defined by $D[i, j] = min_k\{A[i, k] + C[k, j]\}$

In our case, we want matrix A to represent the minimal distances from every vertex in B, to every vertex in the graph over the sub-graph we are working on, and matrix C to represent the minimal distances from every vertex in the graph to every vertex in B. The distance product of these two matrices gives us, for any vertices $(u, v) \in B$, the shortest u-to-v path, while making use of all possible sub-paths in the graph. We generate a distance product for every sub-graph G_j, these distance products are B_j, which were briefly mentioned earlier.

3.1.1 Constructing the input matrices needed to compute the distance product

First, we need to produce the two matrices A and C, or henceforth D_{j1} and D_{j2}. Here the notation should imply that the D_js were constructed over the respective G_j.

This is done with an algorithm by Yuster-Zwick (notably one of the authors of this paper was a co-author). The following Lemma from the paper defines its in and output. Let $\mathrm{c}(u, v)$ represent the smallest number of edges on a shortest path from u to v and $\delta(u, v)$ represent the distance (sum of edge weights) from u to v.

Lemma 3.1. [2] Given an n-vertex graph, the Yuster-Zwick algorithm constructs in $\tilde{O}(Mn^{\omega})$ time, an $n \times n$ matrix D with the following properties: For any pair of vertices i, j there exists a vertex k on a shortest path realizing $\mathrm{c}(i, j)$ so that $D[i, k] = \delta(i, k), D[k, j] = \delta(k, j)$, and $D[i, k]+D[k, j] = \delta(i, j)$.

This algorithm also lowers the amount of computations necessary by probabalistically reducing the number of entries computed over. It makes use of the distance product algorithm outlined in Lemma 3.2. This makes use of fast matrix multiplication and thus introduces ω, the fast matrix multiplication exponent. It represents the value ω, where fast matrix multiplication can be done in $O(n^{\omega})$. Hence it is also the part where the practicality of the algorithm falls apart.

The input it takes is the matrix W, which represents the integer edge weights of a directed graph $G = (V, E)$. If two vertices have no edge between them the value is set to $+\infty$. It works by iteratively computing the distance product of every node to every other node. It starts by finding the distance product $W \star W$, giving us the shortest paths over at most one intermediary vertex. It then repeats this several times, until a certain numerical threshold that decreases with every iteration is reached. Every iteration, it only updates the values where the newly computed ones are smaller than the ones present, so the matrix now captures the shortest paths over some l intermediary vertices.

It then selects a random subset $C \subset V$, and computes the distance products $V \star C$ and $C \star V$, updating appropriate entries if new minimums are found. The idea is similar to the one we use when constructing the subset of vertices B used in the main algorithm, namely, that some vertex on the shortest path between two vertices will be in B or C respectively. This process is iterated, where C is shrinking by a constant factor every iteration so $C_{i+1} \subset C_i$. It iterates until we have a high probability that all shortest

paths have been found. Further details of reasoning and proof are omitted for brevity.

We get the output D_j, with the properties described by the Lemma. To find the shortest distance from vertex i to vertex j, one would need to take the distance product of the i-th row and j-th column of D_j. Alternatively, to find all shortest distances simply take the distance product $D_j \star D_j$. We do something inbetween, we construct the two matrices D_{j1} and D_{j2} from D_j. D_{j1} simply is all rows of D_j that correspond to vertices in B, so it represents the shortest distance from every vertex in B to every vertex in the graph. D_{j2} similarly is all columns of D_j which correspond to B, representing the shortest distance from every vertex in the graph to every vertex in B. Again, these distances are all over G_j. Thus the distance product $D_{j1} \star D_{j2}$ contains all B-to-B distances over G_j.

3.1.2 Computing the distance product

The distance product is computed according to the following Lemma from the paper.

Lemma 3.2. [3] Let A be an $n^r \times n^s$ matrix and let B be an $n^s \times n^t$ matrix, both with elements taken from $\{-L, ..., L\} \cup \{+\infty\}$. Then, the distance product $A \star B$ can be computed in $\tilde{O}(Ln^{\omega(r,s,t)})$ time, where $\omega(r, s, t)$ is the matrix multiplication exponent of multiplying an $n^r \times n^s$ matrix with an $n^s \times n^t$ matrix.

As time is limited, I wont go into detail about this specific implementation of the distance product computation, but rather point out how the bound L of element magnitude is dealt with.

The issue at hand is that the elements of D_{j1} and D_{j2} can be as large as $M(n-1)$, the maximal edge weight times the longest possible path between two vertices, but we would like to minimize it to reduce the time taken to compute the distance product, without losing any relevant information.

The shortest path between two vertices in B is only of interest to us if the path is no longer than an interval, since otherwise the path can be constructed by combining intervals. Our construction gives us a high probability guarantee that all intervals have a vertex in B, so the overlap necessary to construct the path out of intervals will exist. Worst case, we must combine a couple reads of D_j to represent this long path. Therefore, we only consider elements of D_{j1} and D_{j2} with values less than $Mn^{1-\alpha}$, the maximal length of an interval. All other entries are set to $+\infty$, and Lemma 3.2 gives us a time-bound of $\tilde{O}(Mn^{1-\alpha}n^{\omega(\alpha,1,\alpha)})$.

3.2 Constructing dense distance graphs from B_j & F_{e_i}

Having computed B_j for $j = \{1, ..., r\}$, we can now use these to construct the dense distance graphs $G_B^i = (B, E')$. They have the vertex set B, and the weight of an edge (u, v) is the shortest u-to-v path in all $G_j \in F_{e_i}$. To get this weight for an edge, we examine a single entry in every B_j such that $G_j \in F_{e_i}$, and choose the minimum value.

3.3 Adjusting the weights to be non-negative

To be able to use Dijkstra's shortest-path algorithm, all the weights of G_B^i must be adjusted to be non-negative. This is done with a *feasible price function*.

Definition 3.2. Feasible price function: A price function $\varphi(\cdot)$ maps from the vertices of some graph $G = (V, E)$ to the reals. We use the values $\varphi(u)$ and $\varphi(v)$ to reduce the length of the edge (u, v) with respect to φ. The new weight is $w_\varphi(u, v) = w(u, v) + \varphi(u) - \varphi(v)$. The price function is called feasible if $w_\varphi(u, v) \geq 0$ for all edges $(u, v) \in E$.

The function being used here, $\mathrm{d}(\cdot)$, adds some vertex x to the graph G, where x has an edge of weight 0 to every other vertex on the graph. For some vertex u in V, $\mathrm{d}(u)$ is the length of the shortest path from x to u in $G \cup \{x\}$. For every edge (u, v), $d(v) \leq d(u) + w(u, v)$, so $w_d(u, v) \geq 0$. It follows that $w_d(u, v) = w(u, v) + d(u) - d(v) \geq 0$. If $\mathrm{w}(u, v)$, and/or $\mathrm{d}(u)$ are negative, $\mathrm{d}(v)$ must be as negative as the sum of the two (as it is the shortest distance from x-to-v, which can be decomposed into x-to-u and u-to-v, and thus by subtracting it, the adjusted weight is at least equal to zero).

4 Probabilistic bounds

I now briefly examine the reasoning and proof of some of the probabilistic bounds in this paper.

4.1 Number of sub-graphs G_j

How did we decide on the number of sub-graphs r generated in the first step? Again, the goal of the sub-graphs is that every interval of a replacement path lies in one of the sub-graphs, which does not include the edge being avoided (i.e. in some $G_j \in F_{e_i}$).

The expected number of sub-graphs without edge e_i is $E[|F_{e_i}|] = rn^{\alpha-1}$. Notably, we want both a lower and an upper bound, where the lower bound

gives us a high probability to meet the goal, while the upper bound keeps the time-bound of all following operations low. To achieve this, the paper chooses an r so that $r = O(n^{1-\alpha} \log n)$. So $E[|F_{e_i}|] = O(\log n)$.

The paper proves that every $|F_{e_i}|$ lies within defined bounds with probability $1 - \frac{1}{n}$ on either side by using Chernoff and union bounds, details of which can be found in the paper.

$|F_{e_i}|$ is relevant as we need all intervals of P_i to exist within F_{e_i}. We will also eventually construct a B_j for every $G_j \in F_{e_i}$, so we would like to keep $|F_{e_i}|$ as small as possible to avoid redundant computations.

We can find a lower bound on the probability that all n^2 possible intervals of a replacement path survive in F_{e_i}, using similar reasoning as above.

4.2 Probability that all intervals survive

Every interval survives in a single sub-graph with probability:

$$(1 - n^{\alpha-1})^{n^{(1-\alpha)}-1} > \frac{1}{e}$$

That is, the probability that an edge isn't removed to the power of the length of an interval (in edges). So the probability that some interval doesn't survive in F_{e_i} is less than $(1 - \frac{1}{e})^{|F_{e_i}|}$. Using the precise lower bound for $|F_{e_i}|$ defined in the paper, we get:

$$(1 - \frac{1}{e})^{|F_{e_i}|} < (1 - \frac{1}{e})^{21 \log n} < (1/e)^{4 \log n} < 1/n^4$$

By union bound:

$$P(\bigcup_{j=1}^{n^2} \text{Interval j doesn't survive}) \leq \sum_{j=1}^{n^2} P(\text{Interval j doesn't survive})$$

$n^2 \times \frac{1}{n^4} = \frac{1}{n^2}$, so all n^2 intervals survive with probability $1 - \frac{1}{n^2}$

4.3 Number of elements selected for B

The goal of our construction of B, is to have at least one vertex from every disjoint interval (so given a vertex on P_i, some vertex within the next interval) present in B. This is so that we can combine all the shortest intervals found in the various sub-graphs in F_{e_i} to the actual replacement path P_i. The size $|B|$ is chosen, so that every interval contains some vertex in B with probability of at least $1 - \frac{1}{n}$.

In a general sense, we use the probability that a specific vertex is not in B, which is based on $|B|$ (as we select vertices until we reach the predefined size), and then expand this to the probability of a whole interval not being in B. Finally, using inequality identities and union bounds we prove the lower bound stated above. As these transformations aren't really of interest, for the sake of understanding the time bounds, we should just know that $|B| = \tilde{O}(n^{\alpha})$.

5 Time Bounds

The paper set out to prove that they could solve the replacement paths problem in $\tilde{O}(Mn^{1+\frac{2}{3}\omega})$. Now that we have established the probabilistic bounds, which have a large effect on the time bound, I briefly explain how this total time bound was reached.

1. We generate sub-graphs in
   $O(rn^2) = \tilde{O}(n^{3-\alpha})$, as we handle every one of n^2 possible edges r times.
2. We select B in $\tilde{O}(n^{\alpha})$, as we select that amount of vertices.
3. We then construct the intermediary matrices D_j in $\tilde{O}(rMn^{\omega}) = \tilde{O}(Mn^{\omega+1-\alpha})$, according to the time-bound of the lemma, multiplied by the number of times it must be done.
4. We then compute the distance products in $\tilde{O}(rMn^{1-\alpha}n^{\omega(\alpha,1,\alpha)}) = \tilde{O}(Mn^{1-\alpha+\omega})$, details of this transformation can be found in the paper.
5. We constructed all G_B^i in $O(|B|^2 n \log n) = \tilde{O}(n^{2\alpha+1})$, as all $|B|^2$ (where $|B| = \tilde{O}(n^{\alpha})$) entries of one G_B^i are found by going through every graph in F_{e_i}, and $|F_{e_i}| = O(\log n)$. The extra n represents the maximal number of edges on the initial path (as we must construct a G_B^i for every e_i).
6. Finally, Dijkstra's algorithm runs in $O(|B|^2)$, so running it for all G_B^i takes $O(n|B|^2) = \tilde{O}(n^{1+2\alpha})$

All together, we get:

$$\tilde{O}(n^{3-\alpha}+n^{\alpha}+Mn^{1-\alpha+\omega}+Mn^{1-\alpha+\omega}+n^{1+2\alpha}+n^{1+2\alpha}) = \tilde{O}(n^{3-\alpha}+Mn^{1-\alpha+\omega}+n^{1+2\alpha})$$

The paper sets α to $\frac{\omega}{3}$, giving us $\tilde{O}(Mn^{1+\frac{2}{3}\omega})$, which they set out to prove. (note: $n^{3-\frac{1}{3}\omega} < Mn^{1+\frac{2}{3}\omega}$ as $\omega = 2.376, n^{2.208} < Mn^{2.584}$)

6 Conclusion

The algorithm presented may not be practical due to its reliance on fast matrix multiplication, but the use of probabilistic bounds to reduce the overall workload, as well as the idea to combine sub-paths from different sub-graphs are interesting concepts. The algorithm can also be altered slightly to model the avoidance of vertices, or expanded to handle the all pairs replacement paths problem.

In terms of further expansion, the authors suggest that a possible improvement would be reducing the dependency on M.

References

[1] O. Weimann and R. Yuster, "Replacement Paths via Fast Matrix Multiplication," *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, Las Vegas, NV, 2010, pp. 655-662.

[2] R. Yuster and U. Zwick, Answering distance queries in directed graphs using fast matrix multiplication, in *Proc. of the 46th annual symposium on Foundations Of Computer Science (FOCS)*, 2005, pp. 389396.

[3] N. Alon, Z. Galil, and O. Margalit, On the exponent of the all pairs shortest path problem, *Journal of Computer and System Sciences*, vol. 54, pp. 255262, 1997.