Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

An Efficient Algorithm for Assessing the Number of *st*-Paths in Large Graphs

Giulia Punzi* Alessio Conte* Roberto Grossi* Andrea Marino†

Abstract

Counting the number of subgraphs, or patterns, of a certain kind is at the heart of data mining, and *st*-paths are one of the most basic graph patterns to express connectivity. The problem of counting the number of *st*-paths in a graph, both directed and undirected, has been studied since the 70s, and is one of the original #P-complete problems introduced by Valiant [25]. However, counting can be a heavy task and known algorithms already struggle on graphs with hundreds of nodes. For this reason we propose a novel approach: we assess whether the number of *st*-paths of an undirected graph is *at least* a given number z. Instead of finding paths one-by-one (i.e., listing), our algorithm is based on decomposing and collapsing computational tasks arranged in a tree-like structure to enhance the effectiveness of each step in growing the number of paths found. Extensive experimental results on real-world datasets show the algorithm scaling to graphs with millions of nodes and edges, with z in the trillions. Its performance is orders of magnitude better than state-of-the-art listing algorithms adapted to this task.

1 Introduction

Mining paths in graphs is at the heart of network analysis when dealing with connectivity in road networks, communication networks, metabolic networks, protein-to-protein interactions, and social networks, to name a few. Here, one of the computationally intensive tasks is the extraction of suitable paths in the given graph.

In this scenario, an estimation of the number of paths from a source node s and a target node t in a connected graph $G = (V, E)$, also known as *st*-paths, is one of the driving guidelines for making further decisions in the mining task. (We consider each *st*-path not traversing any node twice.) The problem of counting the number of *st*-paths in a graph has been studied since the 70s, and is one of the original #P-complete problems introduced by Valiant [25]. Consequently, counting can be a heavy task and known algorithms already struggle on graphs with hundreds of nodes.

For this reason we consider a novel viewpoint on this counting problem in G: We are given two nodes $s, t \in V$, and an integer parameter $z > 0$. We want to assess whether the number of *st*-paths of G is at least z, spending polynomial in the size of G (i.e. $|V| + |E|$) and in z (which can be larger than the size of G). Assessment on graphs was first considered in [7], with the core advantage that assessment can establish a user-defined lower bound z on the number of solutions, for any given $z > 0$, which is not necessarily guaranteed by approximation algorithms.

In this paper we propose an assessment algorithm pathAssess for *st*-paths that, instead of finding paths one-by-one (i.e., listing), decomposes and collapses the counting tasks arranged in a tree-like structure to enhance the effectiveness of each step in growing the number of paths found.

Algorithm pathAssess provides the same worst-case performance as listing algorithms, i.e., it assesses if the number of *st*-paths of a connected graph $G = (V, E)$ is at least z in $O(|E|\, z)$ time and $O(|E|\,|V|)$ space. The interesting fact, as we shall see, is that the computational difficulty of counting does not prevent pathAssess to perform much better in practice. Extensive experimental results on real-world datasets show pathAssess scaling to graphs with millions of nodes and edges, with z in the trillions. Its performance is orders of magnitude better than state-of-the-art listing algorithms adapted to this task: the latter ones take over 30 minutes to assess a billion paths in our real-world datasets; in contrast, pathAssess takes just tens of seconds to asses trillions of paths, three order of magnitudes more and in much less time, on the same datasets.

1.1 State of the art The problem of listing paths goes back to the 70s, with seminal papers by Read and Tarjan [21] and Johnson [15]. Variations of the problems have been considered with experiments [4, 5, 10, 27], and output-sensitive cost in undirected graphs [3]. Among the variations, listing chordless paths has been investigated in [9, 24], bounded length paths in [17, 19, 20, 22], k disjoint *st*-paths in [12], and paths in temporal graphs in [16].

The problem of counting paths is #P-complete [25]. Formulas for counting open and closed paths using algebraic formulas have been described in [2], but their evaluation does not guarantee $O(|E|\, z)$ time as we

*University of Pisa, giulia.punzi@phd.unipi.it, {alessio.conte, roberto.grossi}@unipi.it

†University of Florence, andrea.marino@unifi.it

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

do. An estimation based on a Monte Carlo stochastic process has been given in [23], but no FPRAS (i.e. efficient approximation algorithm) can exist for *st*-paths unless NP = RP [26]. Counting algorithms can use fast matrix multiplication [11] but the cost of matrix multiplication for large graphs make their running time prohibitive. For planar graphs, upper bounds and asymptotics on the number of paths have been investigated in [8], but many real-world networks are not planar.

As far as we now, there is no literature on the assessment of *st*-paths. Color coding [1] can be adapted for approximated counting of paths with a fixed length k [6], but does not seem suitable for paths of unbounded length, or for assessing. Instead, listing algorithms are suitable, when adapted, to solve our assessment problem, stopping to output when z solutions are found. Some care is required in this case. For instance, the ZDD framework has been successfully used to develop listing algorithms for *st* paths [14, 18], but it requires first to build the entire data structure before counting: the experiments in [27] only manage graphs of small size, with hundreds of nodes. For our experimental study on large graphs, we consider two listing algorithms, which we adapted for assessing: Johnson's algorithm [15], as it is the fastest in practice [4, 11], and Birmelè et al.'s algorithm [3], as it is the fastest in theory.

2 Preliminaries

Given two nodes $s, t \in V$, an *st*-path has nodes s and t as its two endpoints. The set of all *st*-paths of G is denoted by $\mathcal{P}_{s,t}(G)$, where $\mathcal{P}_{t,t}(G)$ only contains the trivial path from t to itself. We say that graph G is *connected* if for every pair of nodes $u, v \in V$ there is a *uv*-path. Graph G is *biconnected* if the removal of any node yields a connected graph. A node u such that $G \backslash u$ is not connected is called an *articulation point*. The *biconnected components* (BCCs) of G are the inclusion-maximal biconnected subgraphs of G. A BCC is called *non-trivial* if it is made up of at least three nodes. We recall that the BCCs of a connected graph form a tree.

Definition 2.1. *Let G be a connected graph, and consider the following graph: add a vertex for each BCC of G, and a vertex for each articulation point of G. Then, add an edge between an articulation point a and a BCC B if and only if $a \in B$. The resulting graph $\mathcal{B}_G$ is a tree, called the* block-cut tree*, or* BC-tree*, of G.*

The vertices of the BC-tree that correspond to BCCs are called *graph-vertices*, while the ones that correspond to articulation points are called *node-vertices*. By construction, the tree alternates levels of node-vertices to levels of graph-vertices. We observe that, given any two nodes x, y of G, there is a unique corresponding shortest path in the BC-tree, which we call the *bead string from x to y*, and we denote with $B_{x,y}$. If the path is trivial, then x and y belong to the same BCC. Otherwise, the two endpoints of $B_{x,y}$ are distinct BCCs: one of them contains x, and the other y.

In the following, we characterize how *st*-paths behave with respect to the BCCs of G. Consider the bead string $B_{s,t}$ in the BC-tree of G: its extremities B_s, B_t (possibly equal) must contain s and t, respectively. Keeping this notation in mind, we can restate the following result by Birmelé et al.:

Lemma 2.1. (Lemma 3.2 from [3]) *All the st-paths in $\mathcal{P}_{s,t}(G)$ are contained in the subgraph of G corresponding to $B_{s,t}$. Moreover, all the articulation points in $B_{s,t}$ are traversed by each of these paths.*

When looking for *st*-paths, we are only concerned with the BCCs along the bead string $B_{s,t}$ from B_s to B_t in the BC-tree, and the paths inside each of these components can be combined independently. We immediately have the following result.

Corollary 2.1. *Let $B_{s,t} = B_s a_0 B_1 a_1 B_2 \cdots B_k a_k B_t$ be the path in $\mathcal{B}_G$ between B_s and B_t, where $a_0, ..., a_k$ are node-vertices, and $B_1, ..., B_k$ are graph-vertices. Then:*

$$\mathcal{P}_{s,t}(G) = \mathcal{P}_{s,a_0}(B_s) \times \left(\prod_{i=1}^{k} \mathcal{P}_{a_{i-1},a_i}(B_i) \right) \times \mathcal{P}_{a_k,t}(B_t). \tag{2.1}$$

3 An Algorithm for Deciding *st*-paths

3.1 Main idea Assume that we are given a lower bounding structural function $L_{s,t}(\cdot)$ that, given as input any biconnected graph and two of its nodes s, t, outputs a lower bound on its number of *st*-paths. Then, we can apply this function individually to each BCC in formula (2.1), and take the product of the resulting values, which we denote with lbP, as it gives a lower bound on the number of *st*-paths. If $lbP \geq z$, we are done. If not so, we need to expand (2.1), where $N(s) = \{u \in V \mid (s,u) \in E\}$ is the set of neighbors of node s, and the union is *disjoint*:

$$\mathcal{P}_{s,t}(G) = \bigcup_{u \in N(s)} s \cdot \mathcal{P}_{u,t}(G \setminus s). \tag{3.2}$$

Specifically, we apply formula (3.2) restricted to just B_s in (2.1): trying to get a better lower bound, we remove s from B_s and we compute the BC-tree of $B_s \backslash \{s\}$. This latter tree replaces the graph-vertex B_s in the original BC-tree. Because of (3.2), the total number of *st*-paths is equal to the sum of the numbers of *ut*-paths, taken over $u \in N_{B_s}(s)$, defined as $N(u) \cap V(B_s)$.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Given $u \in N_{B_s}(s)$, we compute its number of ut-paths in G. We can obtain $B_{u,t}$ in the resulting BC-tree for the whole G. We apply the lower bounding function $L(\cdot)$ to the BCCs of $B_{u,t}$, and take their product according to (2.1). Summing these products over all $u \in N_{B_s}(s)$, according to (3.2), we can update lbP by replacing the lower bound for $B_{s,t}$ with this sum of lower bounds for $B_{u,t}$, taken over $u \in N_{B_s}(s)$.

Looking at the big picture, we begin with a source s and a single path $B_{s,t}$ in the BC-tree of G, initializing lbP as per eq. (2.1). Running the replacement steps above, we get the general case of a BC-tree with several leaves and multiple sources having the same destination t (some sources possibly sharing the same graph-vertex). At each step, we choose a source s' and its leaf $B_{s'}$ in the BC-tree (this s' always exists), and replace $B_{s'}$ with the BC-tree obtained by removing s' from $B_{s'}$. We update the term associated with $B_{s',t}$ in the formula for the current lower bound lbP, replacing it with the sum of the terms for $B_{u',t}$, taken over $u' \in N_{B_{s'}}(s')$. The BC-tree naturally induces a lower bound formula whose syntax follows closely the tree shape, and is made up by nesting instances of (2.1) and (3.2) as mentioned above.

We stop when either the current lower bound lbP is at least z, or we have enumerated all the st-paths and so we know their exact number (and compare it with z). A crucial remark is in order. We need to maintain BCCs, as connectivity from t is not enough. Indeed, the neighbors of s' in $N_G(s') \setminus N_{B_{s'}}(s')$ lead to dead ends, i.e. nodes that cannot reach t in the current graph.

In what follows, we show how to make this scheme work quickly on real-world graphs. First, in Section 3.2, we detail the *multi-source paths tree* data structure. This data structure represents the current BC-tree, and is central in our algorithm as it is used to retain all information concerning BCCs in a compact way, enabling us to guarantee correctness, and to also efficiently update the lower bound. We present in Section 3.3 the complete pseudocode for our algorithm for deciding the number of st-paths of a connected graph G. Then, in Section 3.4, we sketch the proof of the complexity. Finally, our experimental results are presented in Section 4.

3.2 Multi-source tree data structure We introduce a tree data structure, called *Multi-Source Tree Data Structure* (MSTDS), that helps us keep track of the nesting instances of formulas (2.1) and (3.2), as anticipated in Section 3.1. In the following, we denote by $N_{s,t}(G) = |\mathcal{P}_{s,t}(G)|$ the number of st-paths in G.

3.2.1 Exact number of paths We begin by showing how to use the MSTDS and formulas (2.1) and (3.2), to expand the formula for $N_{s,t}(G)$ using the BCCs of G.

First, consider the BC-tree of G, where additional node-vertices for s and t are added, creating an edge from each of them to every BCC in which they appear. Recall that the each node-to-root path in the BC-tree alternates node-vertices (the articulation points in G, along with s, t) to graph-vertices (the BCCs of G). So, take the shortest path from node-vertex s to node-vertex t in the BC-tree, observing that this path has the form $s, B_{s,t}, t = s, B_1, a_1, \ldots, B_{k-1}, a_{k-1}, B_k, t$ (and noting that, if s, t belong to the same BCC B, then this path is simply s, B, t).

The initialization of the MSTDS $\mathcal{T}$ is a homologous path, where s is the only leaf, t is the root, and there is a node for each a_i and B_i mentioned above, following the same order. We associate $N_{a_{i-1},a_i}(B_i)$ with the node of $\mathcal{T}$ representing BCC B_i, for $1 \le i \le k$, where we consider the source s to be a_0 and the destination t to be a_k. By formula (2.1), we obtain the following identity:

$$N_{s,t}(G) = \prod_{i=1}^{k} N_{a_{i-1},a_i}(B_i) \tag{3.3}$$

We now discuss the generic expansion step based on formula (3.2). See Figure 1 for an example of the expansion step. Recall that we may have more than one source, and we maintain the invariant that there is a one-to-one mapping between the sources and the leaves of the MSTDS $\mathcal{T}$. In all that follows, for a vertex x of $\mathcal{T}$, let us denote with $P^{(x)}$ the (unique) node-to-root path in the MSTDS from x to the root t.

Let $\mathcal{S}$ be the current set of sources, where $\mathcal{S} = \{s\}$ initially. Given any source $\tilde{s} \in \mathcal{S}$, let its shortest path to t be $P^{(\tilde{s})} = \tilde{s}, \tilde{B}, a_1, \ldots, B_{k-1}, a_{k-1}, B_k, t$, where we consider the source $\tilde{s}$ to be a_0, the destination t to be a_k, and the first BCC $\tilde{B}$ to be B_1. In the following, we identify the a_i's and the B_i's from the BC-tree with the homologous nodes in $\mathcal{T}$.

Let $u_1, \ldots u_d$ be the neighbors of $\tilde{s}$ in $\tilde{B}$; add them to $\mathcal{S}$, and remove $\tilde{s}$ from $\mathcal{S}$.[1]. Compute the BC-tree $T_{\tilde{s}}$ of $\tilde{B} \setminus \{\tilde{s}\}$ and root it at a_1. Let then $u_1, ..., u_d$ belong to nodes $\ell_1, ..., \ell_d$ of $T_{\tilde{s}}$, respectively (if u_i belongs to more than one node of $T_{\tilde{s}}$, we consider the closest to the root as ℓ_i).[2] Add u_j as a child of ℓ_j in $T_{\tilde{s}}$ for each $j = 1, ..., d$. We now have two possibilities: (1) If $\tilde{s}$ was the only child of $\tilde{B}$, we remove nodes $\tilde{s}$ and $\tilde{B}$ from $\mathcal{T}$. (2) Otherwise, we cannot remove node $\tilde{B}$ from $\mathcal{T}$: there are still other sources using that BCC. We then simply remove node $\tilde{s}$ from $\mathcal{T}$. In the expansion step shown in Figure 1, we are in case (1), and s and B get removed.

Finally, we attach tree $T_{\tilde{s}}$ to node a_1 of $\mathcal{T}$. Note that the final leaves of $\mathcal{T}$ are exactly $\mathcal{S} \setminus \tilde{s} \cup \{u_1, ..., u_d\}$, which

[1] We observe that $d \ge 2$ when $\tilde{B}$ is a non-trivial BCC

[2] We admit the possibility of $\ell_i = \ell_j$ for $i \ne j$.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

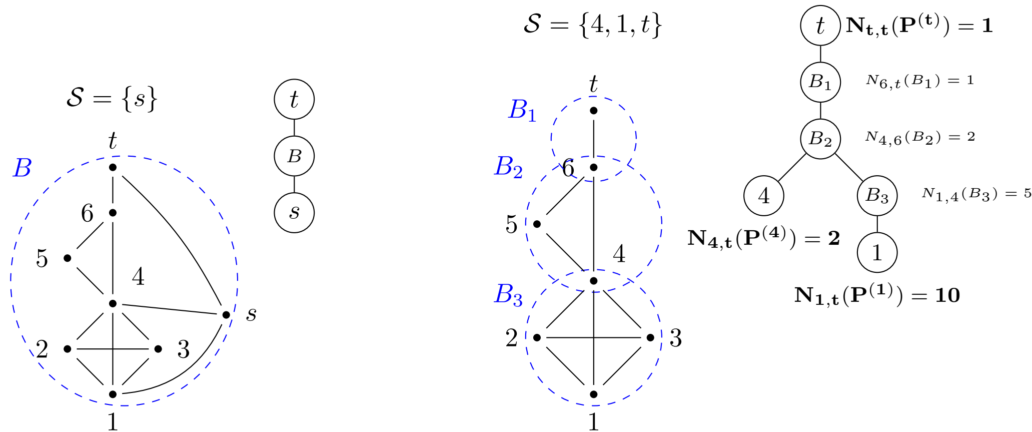

Figure 1: One expansion step of the MSTDS. BCCs of the graph are circled in dashed blue; the MSTDS is shown on the top right of the graph. Note that $13 = N_{s,t}(G) = 10 + 2 + 1 = \sum_{s' \in \mathcal{S}} N_{s',t}(P^{(s')})$, as per equation (3.4).

is the new set of sources. Going back to the figure, we have $a_1 = t$, and tree $T_{\tilde{s}}$ composed of B_1, B_2, B_3. The new sources become the neighbors $\{1, 4, t\}$ of s.

Let P_j be the path from ℓ_j to a_1 in $\mathcal{T}$, for each $j = 1, ..., d$. By formula (3.2), we observe that $N_{\tilde{s},a_1}(\tilde{B}) = \sum_{j=1}^{d} N_{u_j,a_1}(P_j)$, which can be plugged into (3.3):

Lemma 3.1. *The expansion steps in the multi-source tree data structure $\mathcal{T}$ preserve the value of $N_{s,t}(G)$, where $P^{(s')}$ is the leaf-to-root path in $\mathcal{T}$ from s' to t:*

$$N_{s,t}(G) = \sum_{s' \in \mathcal{S}} N_{s',t}(P^{(s')}). \tag{3.4}$$

The pseudocode for the expansion steps, called Explode(), is shown in Algorithm 1. It takes as input a MSTDS $\mathcal{T}$, a source in $\mathcal{S}$ (which is also a leaf of $\mathcal{T}$), and it outputs the new tree resulting from the removal of the given source. We assume to have a function ComputeBCtree that computes the BC-tree of a given graph in linear time, using the classic algorithm by Hopcroft and Tarjan [13].

At this point, we have to discuss two important issues. In Section 3.2.2 we show how to replace $N_{s,t}(G)$ with a lower bound for it. In Section 3.2.3 we show how to implement $\mathcal{T}$ efficiently.

3.2.2 Bounding the number of paths Based on the multi-source tree data structure $\mathcal{T}$, we bound the number of st-paths using a *structural* lower bounding function $L_{s,t}(G)$, with the following properties:

1. Given a biconnected graph G' and two nodes $s', t' \in V(G')$, we have $1 \leq L_{s',t'}(G') \leq N_{s',t'}(G')$.
2. If G' is trivial (size 2), then $L_{s',t'}(G') = 1$.

The function $L_{s,t}(G)$ we adopt in our specific implementation is described in Lemma 3.3. For a node-to-root path $P^{(x)} = xB_1a_1 \dots a_{k-1}B_kt$, let us denote $\mathcal{L}_{x,t} = \prod_{i=1}^{k} L_{a_{i-1},a_i}(B_i)$, where $a_0 = x$ and $a_k = t$. Recall that we keep a running lower bound value lbP, which at any given moment is given by $lbP = \sum_{s' \in \mathcal{S}} \mathcal{L}_{s',t}$. We immediately have the following:

Lemma 3.2. *Consider the multi-source tree data structure $\mathcal{T}$, where $N_{a_{i-1},a_i}(B_i)$ is replaced by $L_{a_{i-1},a_i}(B_i)$, for each path a_{i-1}, B_i, a_i in $\mathcal{T}$, and, consequently, the expansion steps maintain a general formula lbP in place of $N_{s,t}(G)$. Then, $N_{s,t}(G) \geq lbP$*

3.2.3 Efficient implementation We represent the multi-source tree data structure $\mathcal{T}$ by directly linking the BCC nodes, omitting the node-vertices of the BC-tree. We hold this information in a different way; each BCC B is a structure containing the following data:

1. Its set of nodes and edges
2. A boolean expressing whether it is a leaf
3. Its unique target t_B, which is its parent node-vertex in the BC-tree (articulation point leading to t)
4. Its sources: if the BCC is a leaf, these are the actual nodes in $\mathcal{S}$ that it contains; otherwise, we consider as sources its children node-vertices in the BC tree (the articulation points leading to its children).
5. Its *ancestor product* $A(B)$, consisting of the bound $\mathcal{L}_{t_B,t}$. That is, we keep the product of the bounds of all the ancestors of B. This is needed to efficiently update the general bound lbP in $O(1)$.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Algorithm 1 (Expansion Step for the Multi-Source Tree Data Structure (MSTDS))

1: A global lower bounding variable lbP is kept
2: **procedure** EXPLODE($\mathcal{T}$, $\tilde{s} \in \mathcal{S}$)
3: $\tilde{B} \leftarrow$ parent of leaf $\tilde{s}$ in $\mathcal{T}$
4: $\tilde{t} \leftarrow$ parent of $\tilde{B}$ in $\mathcal{T}$
5: $T_{\tilde{s}} =$ COMPUTEBCTREE($\tilde{B} \setminus \tilde{s}, \tilde{t}$) ▷ Rooted at $\tilde{t}$
6: **for all** $u \in N_{\tilde{B}}(\tilde{s})$ **do**
7: $B_u \leftarrow$ BCC of $T_{\tilde{s}}$ containing u closest to $\tilde{t}$
8: Add u as a child leaf of B_u; add u to $\mathcal{S}$.
9: **end for**
10: Remove $\tilde{s}$ from $\mathcal{S}$ and from $\mathcal{T}$
11: $lbP = lbP - \mathcal{L}_{\tilde{s},t}$
12: **if** $\tilde{B}$ has no more children **then**
13: Remove $\tilde{B}$ from $\mathcal{T}$
14: **end if**
15: Add $T_{\tilde{s}}$ as a subtree of $\mathcal{T}$ rooted in $\tilde{t}$
16: **for all** $u \in N_{\tilde{B}}(\tilde{s})$ **do** $lbP = lbP + \mathcal{L}_{u,t}$
17: **end for**
18: **return** $\mathcal{T}$
19: **end procedure**

In our implementation, sources are represented as an array of pairs. In fact, it might happen that the same node s_i becomes a source multiple times. Thus, each source is represented as a pair $\langle s_i, m_i \rangle$, where the number m_i is the *multiplicity* of source s_i, defined as follows. If x is an articulation point and $x \notin \mathcal{S}$, then its multiplicity is -1. The starting source s has multiplicity 1. Every time a source s_i with multiplicity m_i is removed, three things may happen:

- If $x \in N(s_i)$ was already a source with multiplicity $m_x > 0$, then its multiplicity becomes $m_x + m_i$.
- If $x \in N(s_i)$ was already a source with multiplicity $m_x = -1$ (i.e. it was an articulation point), then its multiplicity becomes m_i.
- If $x \in N(s_i)$ was not a source, then it becomes one with multiplicity m_i.

To efficiently update the bound, we can then combine the ancestor product with the information about a source's multiplicity. More formally, if component B contains a source $\langle s', m' \rangle$, its contribution to the general bound can be written as

$$\mathcal{L}_{s',t} = m' \cdot L_{s',t_B}(B) \cdot A(B).$$

Let $O(f(B))$ be the time required to compute $L_{s',t_B}(B)$ for the BCC B; then the value $\mathcal{L}_{s',t}$ is also computable in $O(f(B))$ time, independently of the tree size.

Overall, the running lower bound lbP changes twice during the EXPLODE() procedure. First, at line 11, the contribution $\mathcal{L}_{\tilde{s},t}$ of the chosen source $\tilde{s}$ is removed. With our implementation, this can be performed in $O(f(\tilde{B}))$, as stated above. Then, at line 16, lbP is increased by adding the contribution $\mathcal{L}_{u,t}$ to it, for each neighbor $u \in N_{\tilde{B}}(\tilde{s})$. This requires $O(f(\tilde{B})\,|N_{\tilde{B}}(\tilde{s})|) = O(f(\tilde{B})\,|V|)$ time. In this way, the bound is correctly updated, as per equation (3.4).

Algorithm 2 (Algorithm for Assessing the Number of st-Paths)

1: **procedure** PATHASSESS($G = (V, E)$, s, t, z)
2: $\mathcal{S} = \{s\}$
3: $T_G =$ COMPUTEBCTREE(G, t)
4: Initialize $\mathcal{T}$ as the bead string $B_{s,t}$ in T_G
5: Add t as the root of $\mathcal{T}$, s as only leaf
6: **if** $\mathcal{T}$ is trivial **then** $lbP \leftarrow 1$
7: **else** $lbP \leftarrow \mathcal{L}_{s,t}$
8: **end if**
9: **while** $lbP < z$ **do**
10: **if** $\mathcal{T}$ is trivial **then Output NO**
11: **end if**
12: Choose a source $\tilde{s} \in \mathcal{S}$
13: $\mathcal{T} =$ EXPLODE($\mathcal{T}, \tilde{s}$) ▷ Also updates $\mathcal{S}$, lbP
14: **end while**
15: **Output YES**
16: **end procedure**

3.3 Putting everything together

We summarize what we have discussed so far in the final procedure PATHASSESS, whose pseudocode is given in Algorithm 2. The latter takes as input any connected graph G, two nodes s, t, and a integer parameter $z > 0$.

First of all, we use the following refinement of the Cyclomatic Lemma from [3] as function $L_{s,t}(G)$:

LEMMA 3.3. (REFINEMENT OF LEMMA 4.1 FROM [3]) *Given a biconnected graph $G = (V, E)$, consider the function $\mathcal{C}(G) := |E| - |V| + 2$. Then, for any choice of $s, t \in V$ we have $1 \leq \mathcal{C}(G) \leq N_{s,t}(G)$.*

With the data structures described in Section 3.2.3, function $L_{s,t}(G) \equiv \mathcal{C}(G)$ can be computed in $O(1)$ time.

Secondly, we store the sources $\mathcal{S}$ in a last-in-first-out data structure; this allows us to choose the next source to expand (line 12) in constant time. This specific choice is crucial to obtain the promised space complexity; furthermore, as we will show in section 4, it allows for faster computation in practice.

In the initialization, we compute the BC-tree T_G of G, and we build $\mathcal{T}$ using the only path $a_0 B_1 a_1 B_2 \cdots B_k a_k$ from $a_0 = s$ to $a_k = t$ in T_G. We set $\mathcal{S} = \{s\}$, and we initialize the running lower bound as $lbP := \mathcal{L}_{s,t} = \prod_{i=1}^{k} L_{a_{i-1},a_i}(B_i)$, analogously to what shown in formula (3.3).

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

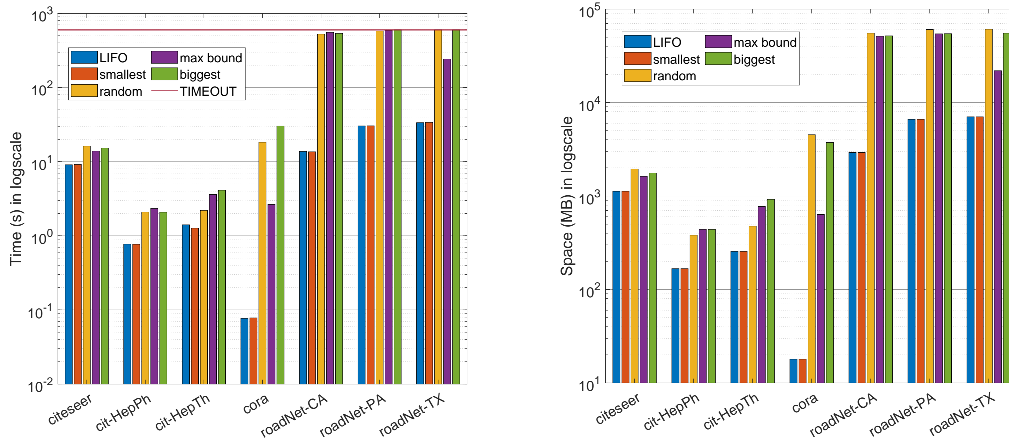

Figure 2: Variant comparison of running time (left) and space (right) for pathAssess on citation and road datasets (the lower the better), with $z = 1$ billion, and a timeout of 10 minutes.

While lbP is smaller than z , we choose a source $\tilde{s}$, and call function Explode($\mathcal{T}, \tilde{s}$). This updates the MSTDS, the sources' set, and the bound lbP accordingly (see Algorithm 1). If now $lbP \geq z$, then we are done, stop and answer YES; otherwise, we repeat the expansion step. If at a certain point of the computation all BCCs of $\mathcal{T}$ become trivial, we stop anyway and say NO: indeed, we enumerated all the st-paths in G and, for this reason, $lbP = N_{s,t}(G)$ is the exact number of these paths, which we know is less than z.

3.4 Complexity

Theorem 3.1. *Algorithm* pathAssess *can decide if the number of st-paths of a connected graph $G = (V, E)$ is at least z in $O(|E|\, z)$ time and $O(|E|\, |V|)$ space.*

We give a sketch of the proof. We start by analysing the time complexity of the Explode() procedure (Algorithm 1). Since our lower bounding function is computable in $O(1)$ time, then any call to Explode() requires $O(|E|)$ time in the worst case. In fact, updating bound lbP requires $O(|V|)$ time, as discussed in section 3.2.3, and every other operation except for the BC-tree computation is constant-time. Thus, the most expensive operation is the computation of the BC-tree of $\tilde{B}$ (line 5), which is still $O(|E|)$ time.

We now prove the $O(|E|z)$ time complexity for pathAssess. Without loss of generality, we can skip trivial BCCs (i.e. composed of two nodes) in linear time: if source $\tilde{s}$ chosen at line 12 belongs to a trivial BCC $\{\tilde{s}, x\}$, we consider x as source instead. This can happen for up to $|V|$ times in a row. It suffices to study the complexity of pathAssess when the lower bounding function is constantly one: $L_{s,t}(\cdot) \equiv 1$. In fact, the steps performed by the algorithm are independent of the function choice; thus choosing a bigger function may only speed up the termination. Since Explode() is only called on non-trivial BCCs, the source always has at least two neighbors, and thus bound lbP increases by at least 1 with every such call. Because of this, the while loop at line 9 iterates for at most z times, each iteration costing the trivial BCC traversal, plus the complexity of Explode(): $O(|V| + |E|) = O(|E|)$. The total time cost of pathAssess is $O(|E|\, z)$, for any choice of $L_{s,t}(\cdot)$.

On the other hand, the space bound given in Theorem 3.1 depends on the leaf choice strategy at line 12: whenever we call the Explode procedure, we might create a copy of the current BCC. By adopting the LIFO strategy, we minimize such memory duplication, and achieve $O(|E|\, |V|)$ overall space. Intuitively, when we explode a source $\tilde{s} \in \tilde{B}$ and duplicate $\tilde{B}$ (adding $O(|E|)$ space), the newly inserted sources are still nodes that belonged to $\tilde{B}$. These sources will be the first to be exploded next. Proceeding in this way, the whole of $\tilde{B}$ will be consumed before duplication occurs for any ancestor or sibling of $\tilde{B}$; since the paths have length $O(|V|)$, this adds up to $O(|E|\, |V|)$ space.

4 Experimental Results

This section is devoted to show our experimental results, evaluating the behaviour of our algorithm pathAssess. As mentioned in Section 1.1, there are no assessment algorithms for st-paths yet, and we single out the listing algorithms in [15, 3] as competitors of pathAssess.

One competitor is Johnson, which is the algorithm of [15], adapted for assessment as follows: each time a new path is found, a counter is incremented by one; if the counter eventually reaches z, the output is YES, otherwise is NO. The other competitor is classicBCC,

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

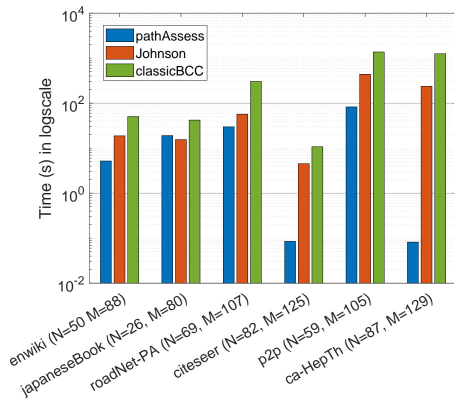

Figure 3: Running times in seconds (the lower the better) of all algorithms with unbounded z. For each graph $G = (V, E)$, we indicate $N = |V|$ and $M = |E|$.

which is conceptually equivalent to the one in [3] but employs simpler data structures to maintain the bead string. This slightly worsens the theoretical complexity, but the recursion tree remains the same. Specifically, classicBCC performs assessment by exploiting equation (3.2) and checking if the path counter is at least z. At every step, the main function takes as input a node u, which is a current source u. The function computes the bead string $B_{u,t}$, and it marks as sources all neighbors $v \in N_{B_{u,t}}(u)$ of u inside the bead string. Then, it recurs over each such v. The base case occurs when $u = t$: in this situation, the path counter is incremented.

Dataset. We tested the algorithms over several datasets from six main categories: citation networks, collaboration networks, language graphs, p2p networks, road networks, and web graphs (see the leftmost columns of Table 1 for the details of these graphs). For each graph, we choose a random pair of nodes s, t, and launch the algorithms with the bead string $B_{s,t}$ as the input graph.

Computing Platform. Our experiments were carried on a dual-processor Intel Xeon Gold 5318Y Icelake machine @2.10GHz, with 48 physical cores each and 1TB of shared RAM, running Ubuntu Server 22.04 LTS. The `C++` code is available at `https://github.com/giuliapunzi/stPaths_assessment`, and compiled with version 11.2.0 of the g++ compiler.

Variants' Comparison and Setup. Before proceeding, in the following we evaluate several possible settings for pathAssess, establishing the setup for the remainder of our experiments. In particular, there are several possible strategies for choosing which source $s' \in \mathcal{S}$ to expand at line 12 of procedure pathAssess. We compared five different strategies: (1) *Random*: pick s' at random; (2) *Smallest*: choose s' belonging to the BCC B' of minimum size; (3) *Biggest*: choose s' belonging to the BCC B' of maximum size; (4) *Maximum ancestor bound*: choose s' belonging to the BCC B' with maximum $A(B')$ value; (5) *LIFO*: choose s' in a last-in-first-out fashion. Figure 2 compares running time (left) and space (right) for the variants over several datasets, with $z = 1$ billion and a timeout of 30 minutes.

We observe how strategies *smallest* and *LIFO* always outperform all others both in terms of time and space, in some cases (road networks) of at least one order of magnitude. More specifically, the performance of *LIFO* and *smallest* (both time and space) was one order of magnitude better than all others on 6 of the 22 considered datasets (27%), while it was at least two times better on 8 of the remaining ones (36%). Lastly, there were 7 graphs where all variants performed equally, and one last language-type graph (*japaneseBook*) where the behaviour differed between time and space: *LIFO* and *smallest* were one order of magnitude worse time-wise, but one order of magnitude better space-wise. Overall, we adopt the constant-time strategy *LIFO* as default in the rest of the experiments.

Comparisons with $z = \infty$. We compare the running times of pathAssess, Johnson, and classicBCC with a virtual value $z = \infty$ to observe the behaviour during all parts of the computation, essentially requiring all the algorithms to find the number of all the paths. To obtain sufficiently small graphs for this purpose, we extracted a small subgraph (25-100 nodes) from each category, and compared the running time of the algorithms. The results are shown in Figure 3, where we report the running times of the competitors for each of the subgraphs considered. We observe that pathAssess almost always outperforms classicBCC of at least one order of magnitude, and Johnson of 1.5 times. The only exception seems again to be graph *japaneseBook*, where the improvement with respect to classicBCC is less evident and Johnson is slightly faster (interestingly, we later show how our algorithm still outperforms Johnson on the whole graph for bounded z).

Comparisons with bounded z. In the following, we report our results concerning true assessment on the full datasets for various values of z. The results for $z = 1$ billion are shown in Table 1, where we report the number of nodes and edges for each graph, and the running time, the space used, and the number of paths found for each algorithm. OOT refers to the fact that the algorithm reached the timeout, which is set to 30 minutes. In this case, in order to give an idea on how far the algorithm is from terminating, we report the number of paths assessed before the timeout. We omit the results of classicBCC, as it almost always went into timeout finding at least one order of magnitude of paths

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Graphs			$z = 1$ billion					
			Johnson			pathAssess		
Name	Nodes	Edges	Time (s)	Space (MB)	Paths assessed	Time (s)	Space (MB)	Paths assessed
citeseer	141,090	422,122	OOT		388,550,223	**5**	**1,125**	1,001,953,579
cit-HepPh	32,997	419,304	358	**30**	1,000,000,007	**0.5**	167	1,047,202,891
cit-HepTh	25,743	350,158	OOT		138,362,303	**0.8**	**256**	1,004,529,657
cora	1,948	4,475	OOT		356,479,058	**0.08**	**18**	1,023,538,676
advogato	4,064	41,593	OOT		69,613,257	**0.7**	**157**	1,012,895,841
ca-AstroPh	15,929	193,922	OOT		47,882,493	**0.95**	**233**	1,023,781,518
ca-CondMat	17,234	84,595	524	**11**	1,000,000,003	**0.24**	70	1,905,902,831
ca-HepPh	9,026	114,047	816	**7**	1,000,000,030	**0.2**	79	1,044,835,780
ca-HepTh	5,903	20,990	OOT		340,243,734	**0.1**	**35**	1,044,651,124
darwinBook	6,403	43,229	OOT		29,654,119	**0.2**	**38**	1,049,829,226
frenchBook	6,395	21,911	OOT		8,359,789	**0.5**	**97**	1,002,665,823
japaneseBook	1,897	7,194	OOT		76,966,934	**204**	**21**	1,000,000,052
spanishBook	8,286	39,774	OOT		3,450,391	**0.07**	**24**	1,098,621,100
p2p	4,705,666	141,363,525	OOT		121,010	**13**	**4,062**	2,869,814,661
p2p-Gnutella31	33,813	119,127	OOT		34,305,802	**6**	**1,569**	1,000,175,372
roadNet-CA	1,563,364	2,357,971	OOT		124,714	**13**	**2,917**	1,257,827,312
roadNet-PA	863,105	1,313,732	491	**346**	1,000,000,000	**30**	6,619	1,060,099,149
roadNet-TX	1,050,435	1,572,563	OOT		843,495,171	**33**	**7,032**	1,039,417,627
cnr_2000	180,581	2,287,332	OOT		88,894,150	**0.1**	**92**	514,859,833,713
enwiki-20071018	2,045,543	42,311,597	OOT		46,962	**8**	**2,024**	8,335,032,150
eu_2005	803,975	16,059,329	859	1,557	1,000,000,049	**2.5**	**987**	1,418,742,364
GoogleNw	13,403	146,225	109	**9**	1,000,000,001	**0.1**	23	2,773,834,099

Table 1: Running time (seconds), space (MB), and paths assessed by Johnson and pathAssess when performing assessment with $z = 1$ billion over all datasets. OOT indicates that the algorithm reached the timeout of 30 minutes and, in this case, the number of paths assessed refers to the ones found within the 30 minutes.

less than Johnson. For both space and time, we report in bold the best performance among the competitors able to complete the task without running out of time.

Looking at Table 1, it is clear how pathAssess largely outperforms the others in terms of time, terminating in a matter of seconds, or less, while Johnson often runs out of time (on 73% of the instances). Furthermore, when Johnson did run out of time it often only found tens of millions of paths, one order of magnitude less than the sought z. Interestingly, the worst behaviour for pathAssess (hundreds of seconds, instead of just a few) is on *japaneseBook*, the same graph whose subgraph performed badly in the exhaustive counting experiments; still, our algorithm correctly assesses the existence of at least a billion paths, while Johnson reaches the timeout having found just 77 millions of them. Concerning space consumption, pathAssess typically uses considerably more space than Johnson, as it has to store the BC-tree; still, the space used appears acceptable even for commodity hardware.

While for z equal to 1 billion Johnson ran very often into timeout, the number z of paths to be assessed can be pushed much further when using pathAssess. Indeed, we also ran pathAssess with $z = 1$ trillion over the same graphs, and the only one which caused a timeout was *japaneseBook*; on almost all other datasets the algorithm still terminated in a matter of tens of seconds. It is worth remarking that, as the bottleneck of our approach seems to be the space consumption (as previously observed in Table 1), to assess for trillions of paths, tens of gigabytes of memory are often required.

Number of paths assessed over time. We performed experiments concerning the number of paths assessed over time. Specifically, all the three competitors while executing, they keep updating a lower bound on the number of paths they were able to asses so far, namely a *running bound*, terminating as soon as this become bigger than z. In the left of Figure 4, we observe how these running bounds grow during the execution of the algorithm in the case of some subgraphs of *RoadNet-PA* and *citeseer* when $z = \infty$ (also previously considered in Figure 3). Our running bound has the fastest growth, while the one for classicBCC unsurprisingly grows the slowest.

Similarly, in the right of Figure 4 we observe the growth of these running bounds for z set to 1 trillion for some road networks (left) and web graphs (right). In this case, as Johnson and classicBCC were not able to perform this task running always out of time, the results are shown only for pathAssess. It is worth remarking that in most of the instances, our running bounds have a more-than-linear growth, hence explaining why pathAssess is so fast to assess z paths, with z much bigger than $N = |V|$.

Acknowledgments Work partially supported by MIUR PRIN Project 20174LF3T8 AHeAD.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

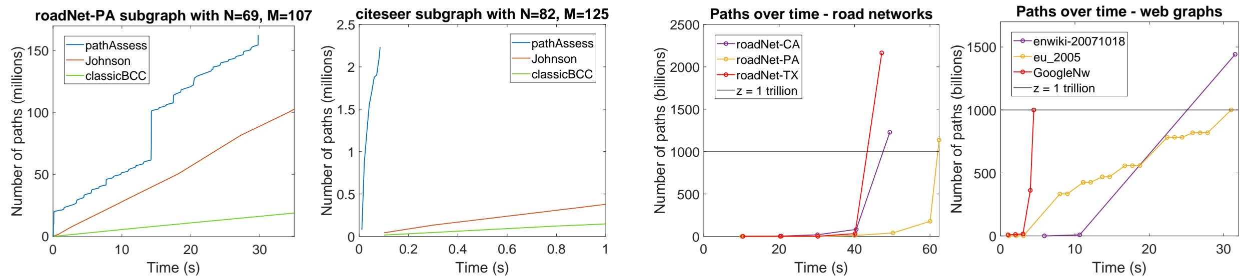

Figure 4: Number of paths assessed over time. Left: trend for all algorithms when assessing with unbounded z on two small graphs. Right: trend for PATHASSESS when assessing for $z = 1$ trillion, with timeout = 10 minutes.

References

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

[2] E. Biondi, L. Divieti, and G. Guardabassi. Counting paths, circuits, chains, and cycles in graphs: a unified approach. *Canadian J. Math.*, 22(1):22–35, 1970.

[3] E. Birmelé, R. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *ACM-SIAM SODA*, pages 1884–1896. SIAM, 2013.

[4] J. Blanusa, P. Ienne, and K. Atasu. Scalable fine-grained parallel cycle enumeration algorithms. In *ACM SPAA*, pages 247–258, 2022.

[5] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck. Shortest-path feasibility algorithms: An experimental evaluation. *Journal of Experimental Algorithmics*, 14:2–7, 2010.

[6] A. Conte, G. Ferraro, R. Grossi, A. Marino, K. Sadakane, and T. Uno. Node similarity with q-grams for real-world labeled networks. In *Proc. 24th ACM SIGKDD*, pages 1282–1291, 2018.

[7] A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, and G. Punzi. Beyond the BEST theorem: Fast assessment of Eulerian trails. In *FCT*, pages 162–175. Springer, 2021.

[8] C. Cox and R. R. Martin. Counting paths, cycles, and blow-ups in planar graphs. *J. of Graph Theory*, 2021.

[9] R. Ferreira, R. Grossi, R. Rizzi, G. Sacomoto, and M.-F. Sagot. Amortized $\tilde{O}(|V|)$-delay algorithm for listing chordless cycles in undirected graphs. In *ESA*, pages 418–429. Springer, 2014.

[10] L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck. An experimental study of minimum mean cycle algorithms. In *WEA*, pages 1–13. SIAM, 2009.

[11] P.-L. Giscard, N. Kriege, and R. C. Wilson. A general purpose algorithm for counting simple cycles and simple paths of any length. *Algorithmica*, 81(7):2716–2737, 2019.

[12] R. Grossi, A. Marino, and L. Versari. Efficient algorithms for listing k disjoint st-paths in graphs. In *LATIN*, pages 544–557. Springer, 2018.

[13] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[14] H. Iwashita, J. Kawahara, and S.-i. Minato. ZDD-based computation of the number of paths in a graph. *Hokkaido University, TCS-TR-A-10-60*, 2012.

[15] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Computing*, 4(1):77–84, 1975.

[16] R. Kumar and T. Calders. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *Proc. VLDB Endow.*, 11(11):1441–1453, 2018.

[17] Z. Lai, Y. Peng, S. Yang, X. Lin, and W. Zhang. Pefp: Efficient k-hop constrained st simple path enumeration on FPGA. In *37th IEEE International Conference on Data Engineering*, pages 1320–1331. IEEE, 2021.

[18] M. Nishino, N. Yasuda, S.-i. Minato, and M. Nagata. Compiling graph substructures into sentential decision diagrams. In *Thirty-First AAAI Conference*, 2017.

[19] Y. Peng, X. Lin, Y. Zhang, W. Zhang, L. Qin, and J. Zhou. Efficient hop-constrained st simple path enumeration. *The VLDB Journal*, 30(5):799–823, 2021.

[20] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou. Hop-constrained st simple path enumeration: Towards bridging theory and practice. *Proc. VLDB Endow.*, 13(4):463–476, 2019.

[21] R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[22] R. Rizzi, G. Sacomoto, and M.-F. Sagot. Efficiently listing bounded length st-paths. In *IWOCA*, pages 318–329. Springer, 2014.

[23] B. Roberts and D. P. Kroese. Estimating the number of st paths in a graph. *JGAA*, 11(1):195–214, 2007.

[24] T. Uno and H. Satoh. An efficient algorithm for enumerating chordless cycles and chordless paths. In *Conf. Discovery Science*, pages 313–324. Springer, 2014.

[25] L. G. Valiant. The complexity of enumeration and reliability problems. *SICOMP*, 8(3):410–421, 1979.

[26] M. Yamamoto. Approximately counting paths and cycles in a graph. *Disc. App. Math.*, 217:381–387, 2017.

[27] N. Yasuda, T. Sugaya, and S.-I. Minato. Fast compilation of st paths on a graph for counting and enumeration. In *Advanced Methodologies for Bayesian Networks*, pages 129–140. PMLR, 2017.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 10/18/23 to 150.217.37.165 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy