An Efficient and Scalable Algorithm for Estimating Kemeny's Constant of a Markov Chain on Large Graphs

Shiju Li*
Florida Institute of Technology
sli2015@my.fit.edu

Xin Huang*
Florida Institute of Technology
xhuang2016@my.fit.edu

Chul-Ho Lee
Florida Institute of Technology
clee@fit.edu

ABSTRACT

The mean hitting time of a Markov chain on a graph from an arbitrary node to a target node randomly chosen according to its stationary distribution is called Kemeny's constant, which is an important metric for network analysis and has a wide range of applications. It is, however, still computationally expensive to evaluate the Kemeny's constant, especially when it comes to a large graph, since it requires the computation of the spectrum of the corresponding transition matrix or its normalized Laplacian matrix. In this paper, we propose a simple yet computationally efficient Monte Carlo algorithm to approximate the Kemeny's constant, which is equipped with an (ϵ, δ)-approximation estimator. Thanks to its inherent algorithmic parallelism, we are able to develop its parallel implementation on a GPU to speed up the computation. We provide extensive experiment results on 13 real-world graphs to demonstrate the computational efficiency and scalability of our algorithm, which achieves up to 500× speed-up over the state-of-the-art algorithm. We further present its practical enhancements to make our algorithm ready for practical use in real-world settings.

CCS CONCEPTS

• **Mathematics of computing** → **Graph algorithms**; • **Computing methodologies** → **Parallel algorithms**;

KEYWORDS

Kemeny's constant; Markov chain on a graph; Monte Carlo algorithm; Parallel computing

ACM Reference Format:
Shiju Li, Xin Huang, and Chul-Ho Lee. 2021. An Efficient and Scalable Algorithm for Estimating Kemeny's Constant of a Markov Chain on Large Graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467431

*Both authors contributed equally to this work.

KDD '21, August 14–18, 2021, Virtual Event, Singapore

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467431

1 INTRODUCTION

Kemeny's constant is a graph/network metric first introduced by Kemeny and Snell [16], where it was originally defined based on the fundamental matrix of a Markov chain on a graph. It has a natural interpretation as the mean hitting time from an arbitrary initial node to a target node randomly chosen according to the stationary distribution of the Markov chain [20]. It has found a wide range of applications in many different areas. For example, it is used to evaluate and improve user navigation efficiency through web graphs [20], and also for search and discovery in peer-to-peer networks by interpreting the Kemeny's constant as the search time of a random search method [7, 26, 30]. It is recently used as an indicator of network robustness to design surveillance policies for the quickest detection of intruders and anomalies in the network [25].

It has also been extensively studied in the literature. It is well known that the Kemeny's constant of a Markov chain on a graph can be expressed by the spectrum (or eigenvalues) of its transition matrix or its normalized Laplacian matrix [9, 13, 20]. It is also shown in [24] that there is a linear relationship between the Kemeny's constant and effective resistance on *regular* graphs. Wang *et al.* [27] recently show a new closed-form expression of the Kemeny's constant based on the pseudoinverse (or Moore-Penrose inverse) of the Laplacian matrix and establish its upper and lower bounds. Very recently, Kooij *et al.* [17] provide two approximations for the Kemeny's constant on non-regular graphs and use their approximations along with the closed-form expression obtained in [27] to evaluate the Kemeny's constant on several families of structured networks, which are limited to *small* graphs.

However, it still remains a challenge to compute the Kemeny's constant in practice, especially when it comes to *large* graphs. It is often infeasible to directly compute the Kemeny's constant based on the spectrum of the transition matrix or its normalized Laplacian matrix, since computing the spectrum of a square matrix is computationally expensive and it requires $O(n^3)$ in practice. Thus, there is a need for a computationally efficient algorithm to compute the Kemeny's constant for large graphs. To fill this gap, very recently, Xu *et al.* [28] propose a randomized algorithm named ApproxKemeny for approximating the Kemeny's constant. They show that estimating the Kemeny's constant boils down to approximating a quadratic form that involves the pseudoinverse of the Laplacian matrix, which is then obtained as a solution of its corresponding Laplacian system of linear equations by employing a nearly linear-time Laplacian solver in [18]. This framework with the pseudoinverse of the Laplacian matrix is also similarly adopted for computing the so-called absorbing random-walk centrality [29].

In this paper, we propose a simple yet computationally efficient Monte Carlo algorithm, which is scalable and amenable to parallel implementation, for estimating the Kemeny's constant, especially

for large graphs. While Monte Carlo-type methods or algorithms have been proposed in the literature, their focus has been limited to computing PageRank [2] and solving a large system of linear equations [10, 15]. To the best of our knowledge, our algorithm is the first algorithm of its kind to approximate the Kemeny's constant. We summarize the contributions of this paper as follows.

- First, we establish an (ϵ, δ)-approximation Monte Carlo estimator to approximate the Kemeny's constant.
- Second, we present its practical algorithm and by leveraging its algorithmic parallelism, we develop its parallel implementation on a GPU. We also characterize their time and space complexity.
- Third, we empirically demonstrate the superiority of our algorithm over ApproxKemeny (the state-of-the-art algorithm) in terms of the computational efficiency based on 13 real-world graphs. In particular, we show that the speed-up by our algorithm is up to 500×, while its estimation accuracy is comparable to that of ApproxKemeny.
- Fourth, we present a 'dynamic' refinement of our algorithm to make it work without requiring a possibly impractical parameter, which is an estimate on the second largest eigenvalue modulus of the transition matrix $\mathbf{P}$. We demonstrate that the dynamic refinement not only resolves this issue but also further speeds up the computation without losing its estimation accuracy.
- Finally, we discuss how the dynamic algorithm can be adopted for estimating the Kemeny's constant, even when the transition matrix $\mathbf{P}$ is not known explicitly.

2 PRELIMINARIES

In this section, we collect definitions and relevant results to set the stage for the development of our Monte Carlo algorithm to estimate the Kemeny's constant of a Markov chain on a graph.

Consider a connected, undirected, non-bipartite graph $G=(V,E)$, where $V := \{1, 2, \ldots, n\}$ is the set of nodes with $|V| = n$ and E is the set of edges with $|E| = m$. The graph G is characterized by an $n \times n$ adjacency matrix $\mathbf{A} = [A_{ij}]$ with elements $A_{ij} = 1$ if there is an edge between nodes i and j, i.e., $(i, j) \in E$, and $A_{ij} = 0$ if otherwise. Let d_i be the degree of node $i \in V$, i.e., $d_i = \sum_j A_{ij}$.

Define a (time-homogeneous) Markov chain $\{X_t\}_{t\geq 0}$ on the graph G, where X_t denotes the location of the Markov chain at time t. Its transition matrix is given by $\mathbf{P} = [P_{ij}]$, where the ij-th entry represents the transition probability $P_{ij} := \mathbb{P}\{X_{t+1} = j | X_t = i\}$. For example, the simple random walk on G is a random walk on G where the next node is chosen uniformly at random from the set of neighbors of the current node, and it is characterized by $P_{ij} = A_{ij}/d_i$. It is well known that the resulting Markov chain $\{X_t\}$ is an ergodic Markov chain, which is irreducible and aperiodic, and it has a unique stationary distribution $\boldsymbol{\pi} := [\pi_1, \pi_2, \ldots, \pi_n]$, where $\pi_i = d_i/(2m)$ for all i. It is also time-reversible, i.e., $\pi_i P_{ij} = \pi_j P_{ji}$ for $i, j \in V$.

Consider a Markov chain $\{X_t\}$ on G that is ergodic and reversible. We define the hitting time of the Markov chain $\{X_t\}$ on node i, which is given by $T_i := \min\{t \geq 0 : X_t = i\}$. The mean hitting time from node i to node j can also be defined by $\mathbb{E}_i[T_j] := \mathbb{E}[T_j | X_0 = i]$, i.e., the expected number of steps to 'hit' node j for the first time, when the Markov chain starts from i. Then, we can define the mean hitting time of $\{X_t\}$ from a given node i to a 'random' destination that is a randomly chosen node according to $\boldsymbol{\pi}$ as follows:

$$\mathbb{E}_i\{T_\pi\} := \sum_{j\in V} \mathbb{E}\{T_j | X_0 = i\}\pi_j, \quad i \in V. \tag{1}$$

This quantity is called Kemeny's constant, and it can be viewed as a 'weighted' mean hitting time [16, 21]. It is also known that this quantity does *not* depend on the initial node i. In other words, $\mathbb{E}_i\{T_\pi\}$ remains the same, no matter where the Markov chain $\{X_t\}$ initially starts from. This result is often called 'random target lemma'. For notational simplicity, we use $\mathcal{K}$ to denote the Kemeny's constant in (1) throughout the paper.

We note that the Kemeny's constant $\mathcal{K}$ can be interpreted from a viewpoint of a 'random surfer' [20]. From the random target lemma and $\sum_i \pi_i = 1$, we can write

$$\mathcal{K} = \sum_{i\in V} \pi_i \mathbb{E}_i\{T_\pi\} = \sum_{i\in V} \pi_i \sum_{j\in V} \mathbb{E}\{T_j | X_0 = i\}\pi_j. \tag{2}$$

Thus, the Kemeny's constant can be viewed as the mean hitting time from an 'unknown' (or randomly chosen) initial node to an unknown destination. Suppose that there is a random surfer moving over G according to $\mathbf{P}$ for a while. At some stage the surfer is in the stationary regime, which can be thought of as if it gets 'lost' without knowing its position and whereabouts. The random surfer then randomly goes through $\mathcal{K}$ steps on average before it reaches its destination. In other words, it is the expected number of steps required for the surfer to reach the destination after getting lost. In a similar vein, $\mathcal{K}$ can be considered as the search time of a random search method and thus it has been used for search and discovery in peer-to-peer networks [7, 26, 30]. It is also often used to measure how long it takes for a random walk having a desired stationary distribution to reach a random destination in various applications [5, 12, 19, 21].

Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of the transition matrix $\mathbf{P}$. Since the Markov chain $\{X_t\}$ is ergodic and reversible, they are real and can be rearranged as

$$1 = \lambda_1 > \lambda_2 \geq \lambda_3 \geq \cdots \geq \lambda_n > -1. \tag{3}$$

We also define the second largest eigenvalue modulus (SLEM) of the transition matrix $\mathbf{P}$ as

$$\lambda_* := \max\{|\lambda_i| : \lambda_i \text{ is eigenvalue of } \mathbf{P}, \lambda_i \neq 1\},$$

which characterizes the *mixing* rate of the Markov chain, i.e., the asymptotic rate of convergence of the Markov chain to the stationary distribution $\boldsymbol{\pi}$ [4]. Then, it is known that the Kemeny's constant can be written in terms of the eigenvalues of $\mathbf{P}$ and is given by

$$\mathcal{K} = \sum_{i=2}^{n} \frac{1}{1-\lambda_i}. \tag{4}$$

It is also known that $\mathcal{K}$ is bounded below and above as follows [13, 20, 24].

$$\frac{(n-1)^2}{n} \leq \mathcal{K} \leq \frac{n-1}{1-\lambda_2}, \tag{5}$$

which implies that $\mathcal{K}$ increases linearly with the size of the graph.

It is worth noting that there is another version of Kemeny's constant in the literature. For example, as recently used in [28], it is the one given by $\mathcal{K}=1+\sum_{i=2}^{n} \frac{1}{1-\lambda_i}$. It is, in fact, the case when the hitting time of $\{X_t\}$ on node i is defined as $T_i^+ := \min\{t \geq 1 : X_t =$

$i\}$. Note that $T_i^+ = T_i$, unless $X_0 = i$ in which case T_i^+ is the first *return* time to i with $\mathbb{E}[T_i^+|X_0 = i] = 1/\pi_i$. Thus, if $\mathbb{E}_i\{T_\pi^+\}$ and $\mathcal{K}$ are defined in (1) and (2), respectively, with T_i replaced by T_i^+, the Kemeny's constant becomes $\mathcal{K} = 1 + \sum_{i=2}^{n} \frac{1}{1-\lambda_i}$, and the difference with (4) is only 1.

In addition, the Kemeny's constant can be represented in terms of the eigenvalues of the normalized Laplacian matrix. Consider a simple random walk on G for now, which corresponds to the normalized Laplacian matrix of the graph G. It will be generalized for *any* ergodic, reversible Markov chain on G below. Define the Laplacian matrix $\mathcal{L}$ and its normalized Laplacian matrix $\mathbf{L}$ by $\mathcal{L} \coloneqq \mathbf{D} - \mathbf{A}$ and $\mathbf{L} \coloneqq \mathbf{D}^{-1/2}\mathcal{L}\mathbf{D}^{-1/2}$, respectively, where $\mathbf{D}$ is the diagonal matrix of degrees, i.e., the i-th diagonal entry of $\mathbf{D}$ is d_i [8]. Letting $\boldsymbol{\sigma} \coloneqq [\sigma_1, \sigma_2, \ldots, \sigma_n]$ be the vector of eigenvalues of $\mathbf{L}$, which satisfy $0 = \sigma_1 < \sigma_2 \leq \sigma_3 \leq \cdots \leq \sigma_n$, we have

$$\mathcal{K} = \sum_{i=2}^{n} \frac{1}{\sigma_i}, \tag{6}$$

which is the Kemeny's constant of a simple random walk on G.

The Kemeny's constant of any given ergodic, reversible Markov chain on G can also be obtained using the identity in (6), as long as the eigenvalues σ_i are the ones of the normalized Laplacian matrix of a properly defined 'weighted' graph. Observe that any ergodic, reversible Markov chain on G can be regarded as a random walk on a weighted graph [1]. For a given Markov chain with $\mathbf{P}$ and $\boldsymbol{\pi}$, we can define a weighted graph, say G', characterized by a weight matrix $\mathbf{W} \coloneqq [W_{ij}]$ with elements $W_{ij} \coloneqq \pi_i P_{ij}$ for $(i, j) \in E$, $W_{ii} \coloneqq \pi_i P_{ii}$ for node i with $P_{ii} > 0$, and $W_{ij} = 0$ if otherwise. The degree d_i of G' is now defined to be $d_i \coloneqq \sum_j W_{ij} = \pi_i$, and thus $\mathbf{D}$ is the diagonal matrix of elements π_i. We can then generalize the normalized Laplacian matrix $\mathbf{L}$ for the weighted graph G' as $\mathbf{L} = \mathbf{D}^{-1/2}(\mathbf{D} - \mathbf{W})\mathbf{D}^{-1/2}$ [8]. Thus, we see that (6) holds with the eigenvalues of this normalized Laplacian matrix $\mathbf{L}$, by observing that $\mathbf{L} = \mathbf{I} - \mathbf{D}^{1/2}\mathbf{P}\mathbf{D}^{-1/2}$ and thus $\sigma_i = 1 - \lambda_i$ for all i, where $\mathbf{I}$ is the identity matrix.

We note that it is generally faster to compute the spectrum (or eigenvalues) of $\mathbf{L}$ than that of $\mathbf{P}$ when they are feasible to compute (or when the graphs are small), since $\mathbf{L}$ is a symmetric matrix but $\mathbf{P}$ is not. Similar to the matrix multiplication, however, the computation of the spectrum of a square matrix is computationally expensive and it requires $O(n^3)$ in practice [28]. Thus, it is often difficult or even infeasible to directly compute the Kemeny's constant based on (4) or (6) for *large* graphs. Therefore, there is a need for a computationally efficient and scalable algorithm to estimate the Kemeny's constant for large graphs.

3 THE STATE-OF-THE-ART ALGORITHM

In this section, we provide an overview of ApproxKemeny proposed by Xu *et al.* in [28], which is the state-of-the-art randomized algorithm for approximating the Kemeny's constant $\mathcal{K}$ of a Markov chain on a graph G.[1] We here focus on a simple random walk on G for the Markov chain on G, as assumed in [28].

[1]Note that ApproxHK is proposed by the same group of authors in [29] for estimating the so-called absorbing random-walk centrality, from which the Kemeny's constant can also be computed. Note that ApproxHK and ApproxKemeny share the common framework that involves the pseudoinverse of the Laplacian matrix and its Laplacian solver. However, in [29], they do not provide any theoretical results nor experiment results under real-world network datasets for estimating the Kemeny's constant. Thus, we do not include ApproxHK for performance comparison in this paper.

ApproxKemeny is based on Hutchinson's estimator [3, 14] to approximate the trace of the pseudoinverse (or Moore-Penrose inverse) of the normalized Laplacian matrix $\mathbf{L}$, since computing $\mathcal{K}$ is equivalent to computing the trace of the pseudoinverse of $\mathbf{L}$. By leveraging the relationship between the pseudoinverse of $\mathbf{L}$ and that of the (original) Laplacian matrix $\mathcal{L}$, estimating the trace of the pseudoinverse of $\mathbf{L}$ boils down to approximating a quadratic form that involves the pseudoinverse of $\mathcal{L}$. Here, instead of directly computing this pseudoinverse, they employ a nearly linear-time Laplacian solver in [18] to solve its corresponding Laplacian system of linear equations. They empirically demonstrate that ApproxKemeny exhibits state-of-the-art performance in terms of the runtime.

We below explain the mathematical framework of ApproxKemeny for the sake of completeness. Since the normalized Laplacian matrix $\mathbf{L}$ is a symmetric matrix, by spectral decomposition, we can write $\mathbf{L} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{U}^T$, where $\mathbf{U}$ is an orthogonal matrix and $\boldsymbol{\Sigma}$ is a diagonal matrix with the eigenvalues of $\mathbf{L}$ on its diagonal. Although $\mathbf{L}$ is not invertible due $\sigma_1 = 0$, it has a pseudoinverse $\mathbf{L}^\dagger$, which is also symmetric. Thus, we have $\mathbf{L}^\dagger = \mathbf{U}\boldsymbol{\Sigma}^{-1}\mathbf{U}^T$, where $\boldsymbol{\Sigma}^{-1}$ is a diagonal matrix whose elements are the eigenvalues of $\mathbf{L}^\dagger$, i.e., $\sigma_1^* = 0$ and $\sigma_i^* = 1/\sigma_i$ for $i = 2, 3, \ldots, n$. Therefore, from (6), we have

$$\mathcal{K} = \sum_{i=2}^{n} \sigma_i^* = \mathrm{tr}(\mathbf{L}^\dagger). \tag{7}$$

Let $\boldsymbol{x}$ be a n-dimensional random vector whose elements are *i.i.d.* Rademacher random variables, i.e., $\mathbb{P}\{x_i = \pm 1\} = 1/2$ for $i = 1, 2, \ldots, n$. It is known from [3, 14] that $\mathbb{E}\{\boldsymbol{x}^T\mathbf{L}^\dagger\boldsymbol{x}\} = \mathrm{tr}(\mathbf{L}^\dagger)$ and thus the following estimator called Hutchinson's estimator can be constructed to approximate the trace of the pseudoinverse $\mathbf{L}^\dagger$:

$$\mathrm{tr}(\mathbf{L}^\dagger) \approx \frac{1}{M}\sum_{i=1}^{M} \boldsymbol{x}_i^T\mathbf{L}^\dagger\boldsymbol{x}_i \tag{8}$$

for some sufficiently large M, where $\boldsymbol{x}_i$ are *i.i.d* copies of $\boldsymbol{x}$. In addition, by noting that $\mathcal{L} = \mathbf{D}^{1/2}\mathbf{L}\mathbf{D}^{1/2}$, we can write

$$\mathbf{L}^\dagger = (\mathbf{I} - \frac{1}{2m}\mathbf{D}^{\frac{1}{2}}\mathbb{1}\mathbb{1}^T\mathbf{D}^{\frac{1}{2}})\mathbf{D}^{\frac{1}{2}}\mathcal{L}^\dagger\mathbf{D}^{\frac{1}{2}}(\mathbf{I} - \frac{1}{2m}\mathbf{D}^{\frac{1}{2}}\mathbb{1}\mathbb{1}^T\mathbf{D}^{\frac{1}{2}}), \tag{9}$$

where m is the number of edges and $\mathbb{1}$ is the n-dimensional column vector whose elements are all ones.

Let $\boldsymbol{y}_i \coloneqq \mathbf{D}^{\frac{1}{2}}(\mathbf{I} - \frac{1}{2m}\mathbf{D}^{\frac{1}{2}}\mathbb{1}\mathbb{1}^T\mathbf{D}^{\frac{1}{2}})\boldsymbol{x}_i$. Observe that $\mathcal{L}$ can be written as $\mathcal{L} = \mathbf{B}^T\mathbf{B}$, where $\mathbf{B}$ is the $m \times n$ edge-node oriented incidence matrix of the graph G, i.e., its (i, j) entry is -1 if the i-th node is the source node of the j-th edge, 1 if it is the target node and zero otherwise (the edge orientation can be arbitrary but fixed). Thus, from (7)–(9), we have

$$\begin{aligned}\mathcal{K} &\approx \frac{1}{M}\sum_{i=1}^{M} \boldsymbol{y}_i^T\mathcal{L}^\dagger\boldsymbol{y}_i = \frac{1}{M}\sum_{i=1}^{M} \boldsymbol{y}_i^T\mathcal{L}^\dagger\mathcal{L}\mathcal{L}^\dagger\boldsymbol{y}_i \\ &= \frac{1}{M}\sum_{i=1}^{M} \boldsymbol{y}_i^T(\mathcal{L}^\dagger)^T\mathbf{B}^T\mathbf{B}\mathcal{L}^\dagger\boldsymbol{y}_i = \frac{1}{M}\sum_{i=1}^{M} \|\mathbf{B}\mathcal{L}^\dagger\boldsymbol{y}_i\|^2, \end{aligned} \tag{10}$$

where the third equality is from that $\mathcal{L}^\dagger$ is a symmetric matrix. Finally, approximating $\mathcal{K}$ boils down to evaluating the quadratic form in (10) that involves $\mathcal{L}^\dagger$, the pseudoinverse of the Laplacian

matrix $\mathcal{L}$. Here, instead of directly computing this pseudoinverse, letting $\boldsymbol{z}_i := \mathcal{L}^\dagger \boldsymbol{y}_i$, ApproxKemeny employs a nearly linear-time Laplacian solver in [18] to solve a Laplacian system $\mathcal{L}\boldsymbol{z}_i = \boldsymbol{y}_i$ for $\boldsymbol{z}_i$, $i = 1, 2, \ldots, M$. We refer to [28] for more details.

4 MONTE CARLO ALGORITHM

In this section, we present a novel Monte Carlo (MC) algorithm that is computationally efficient and amenable to parallel implementation, to approximate the Kemeny's constant $\mathcal{K}$ of a Markov chain on a graph G. We first provide its mathematical framework that leads to an (ϵ, δ)-approximation MC estimator, and then present the MC algorithm and its parallel implementation on a GPU.

4.1 Mathematical Framework

Observe that (4) can be written as

$$\mathcal{K} = \sum_{i=2}^{n}(1 + \lambda_i^1 + \lambda_i^2 + \cdots) = n - 1 + \sum_{i=2}^{n}\sum_{k=1}^{\infty}\lambda_i^k, \tag{11}$$

from $|\lambda_i| < 1$ for all $i \neq 1$. Since the trace of a matrix is the sum of its diagonal elements and is also the sum of its eigenvalues, we see that

$$\sum_{i=2}^{n}\lambda_i^k = \mathrm{tr}(\mathbf{P}^k) - \lambda_1^k = \mathrm{tr}(\mathbf{P}^k) - 1. \tag{12}$$

From (11) and (12), we have

$$\mathcal{K} = n - 1 + \sum_{k=1}^{\infty}\left[\mathrm{tr}(\mathbf{P}^k) - 1\right]. \tag{13}$$

We also observe that

$$\mathrm{tr}(\mathbf{P}^k) = \sum_{i=1}^{n}(\mathbf{P}^k)_{i,i} = n\sum_{i=1}^{n}\mathbb{P}\{X_k = i | X_0 = i\}\mathbb{P}\{U = i\} \tag{14}$$

$$= n \cdot \mathbb{P}\{X_k = X_0 | X_0 = U\}, \tag{15}$$

where U is a uniform random variable on V and $(\mathbf{P}^k)_{i,i}$ is the i-th diagonal entry of $\mathbf{P}^k$, which is the k-step transition probability from node i to itself, i.e., the probability of returning to i after k steps.

We then have two important observations that become the basis for our proposed MC algorithm. First, from (15), we see that the trace $\mathrm{tr}(\mathbf{P}^k)$ can be readily estimated by r *independent* realizations of a Markov chain, which moves on G according to $\mathbf{P}$ for k steps, given that their initial positions are chosen uniformly at random from V. Second, by noting that $\sum_{k=1}^{\infty}\lambda_i^k$ in (11) can be well approximated by its partial sum (recall that $|\lambda_i| < 1$ for all $i \neq 1$), from (11) and (13), we can approximate the Kemeny's constant $\mathcal{K}$ based on a partial sum of $[\mathrm{tr}(\mathbf{P}^k) - 1]$. Note that each trace $\mathrm{tr}(\mathbf{P}^k)$ is estimated by the r independent realizations.

We present our MC estimator as follows. For a given Markov chain with $\mathbf{P}$ and $\boldsymbol{\pi}$, we generate its r independent sample paths of length l, each of which starts from a node that is chosen uniformly at random from V. Let $\{X_0^j, X_1^j, \ldots, X_l^j\}$ be the trajectory of the j-th realization for $j = 1, 2, \ldots, r$. Note that $X_k^1, X_k^2, \ldots, X_k^r$ are independent for each $k = 1, 2, \ldots, l$, so we can use them to estimate $\mathrm{tr}(\mathbf{P}^k)$ for different k values. Specifically, we define $Z_k^j := \mathbb{1}\{X_k^j = X_0^j\}$ for the j-th realization and for each k. For brevity, we also define $h_k := \mathrm{tr}(\mathbf{P}^k)$. Then, from (15), we can build the following MC estimator for h_k:

$$\hat{h}_k(r) := n \cdot \frac{1}{r}\sum_{j=1}^{r} Z_k^j = n \cdot \frac{1}{r}\sum_{j=1}^{r}\mathbb{1}\{X_k^j = X_0^j\}. \tag{16}$$

By the strong law of large numbers, we have

$$\hat{h}_k(r) \xrightarrow{\text{a.s.}} h_k, \text{ as } r \to \infty. \tag{17}$$

Furthermore, from (13), we define

$$\mathcal{K}_l := n - 1 + \sum_{k=1}^{l}\left[h_k - 1\right], \tag{18}$$

and construct its corresponding MC estimator as

$$\hat{\mathcal{K}}_l(r) := n - 1 + \sum_{k=1}^{l}\left[\hat{h}_k(r) - 1\right]. \tag{19}$$

By (17) and the linearity of almost sure convergence, we have

$$\hat{\mathcal{K}}_l(r) \xrightarrow{\text{a.s.}} \mathcal{K}_l, \text{ as } r \to \infty.$$

In other words, $\hat{\mathcal{K}}_l(r)$ is an asymptotically consistent estimator of $\mathcal{K}_l$ in (18).

We below demonstrate that the estimator in (19), with a proper choice of l, can also be used to approximate the Kemeny's constant $\mathcal{K}$ in (13). Specifically, we establish an (ϵ, δ)-approximation of this estimator, which implies, for any small $\epsilon, \delta > 0$, how many realizations r are necessary with a choice of l so that the approximation error can be bounded by ϵ with probability at least $1 - \delta$. To proceed, we need the following .

Theorem 1 (Hoeffding's inequality). *Let $Y_1, \ldots, Y_{n_r}$ be i.i.d. random variables such that $\mathbb{E}[Y_i] = \mu$ and $a \le Y_i \le b$. Then, for any $\epsilon > 0$,*

$$\mathbb{P}\left\{\left|\frac{1}{n_r}\sum_{i=1}^{n_r} Y_i - \mu\right| > \epsilon\right\} \le 2e^{-2n_r\epsilon^2/(b-a)^2}.$$

We below show that $\hat{\mathcal{K}}_l(r)$ achieves an (ϵ, δ)-approximation to $\mathcal{K}$ in (13), when l and r are properly chosen.

Theorem 2. *For any $\epsilon > 0$ and $\delta \in (0, 1)$, choose l such that $l \ge \frac{\log\frac{\epsilon}{2n}(1-\lambda_*)}{\log\lambda_*} - 1$. If $r \ge \frac{2n^2l^2}{\epsilon^2}\log(2l/\delta)$, then we have*

$$\mathbb{P}\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}\right| > \epsilon\right\} \le \delta. \qquad \square$$

Proof. Fix $\epsilon > 0$ and $\delta \in (0, 1)$. We observe that

$$\begin{aligned}
\mathbb{P}\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}\right| > \epsilon\right\} &= \mathbb{P}\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l + \mathcal{K}_l - \mathcal{K}\right| > \epsilon\right\} \\
&\overset{(a)}{\le} \mathbb{P}\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l\right| + \left|\mathcal{K}_l - \mathcal{K}\right| > \epsilon\right\} \\
&\overset{(b)}{\le} \mathbb{P}\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l\right| > \frac{\epsilon}{2}\right\} + \mathbb{P}\left\{\left|\mathcal{K}_l - \mathcal{K}\right| > \frac{\epsilon}{2}\right\} \\
&\overset{(c)}{=} \mathbb{P}\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l\right| > \frac{\epsilon}{2}\right\},
\end{aligned} \tag{20}$$

where (a) uses the triangle inequality, and (b) follows from

$$\begin{aligned}
&\left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l\right| + \left|\mathcal{K}_l - \mathcal{K}\right| > \epsilon\right\} \\
&\subseteq \left\{\left|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l\right| > \frac{\epsilon}{2}\right\} \cup \left\{\left|\mathcal{K}_l - \mathcal{K}\right| > \frac{\epsilon}{2}\right\}
\end{aligned}$$

and the union bound. Also, (c) holds since

$$\mathcal{K} - \mathcal{K}_l = \sum_{k=l+1}^{\infty} [h_k - 1] < \sum_{k=l+1}^{\infty} n\lambda_*^k \leq \frac{\epsilon}{2},$$

where the first inequality is from

$$h_k = \sum_{i=1}^{n} \lambda_i^k \leq 1 + (n-1)\lambda_*^k < 1 + n\lambda_*^k, \tag{21}$$

and the last inequality is due to the choice of l. Since $0 \leq \hat{h}_k(r) \leq n$ and $\mathbb{E}\{\hat{h}_k(r)\} = h_k$ for $k = 1, 2, \ldots, l$, (20) is further bounded by

$$\begin{aligned}
\mathbb{P}\left\{|\hat{\mathcal{K}}_l(r) - \mathcal{K}_l| > \frac{\epsilon}{2}\right\} &= \mathbb{P}\left\{\left|\sum_{k=1}^{l} \hat{h}_k(r) - \sum_{k=1}^{l} h_k\right| > \frac{\epsilon}{2}\right\} \\
&\overset{(d)}{\leq} \mathbb{P}\left\{\sum_{k=1}^{l} \left|\hat{h}_k(r) - h_k\right| > \frac{\epsilon}{2}\right\} \overset{(e)}{\leq} \sum_{k=1}^{l} \mathbb{P}\left\{\left|\hat{h}_k(r) - h_k\right| > \frac{\epsilon}{2l}\right\} \\
&\overset{(f)}{\leq} 2l \exp\left(-\frac{r}{2}\frac{\epsilon^2}{n^2 l^2}\right),
\end{aligned} \tag{22}$$

where (d) uses the triangle inequality, (e) can be shown as was done to show (b), and (f) holds by Hoeffding's inequality. Therefore, with the given choice of r, the result follows since the RHS of (22) is less than or equal to δ. □

4.2 Algorithm and Parallel Implementation

The MC estimator $\hat{\mathcal{K}}_l(r)$ in (19) is simply based on r independent realizations (or sample paths) of a given Markov chain with $\mathbf{P}$ and $\boldsymbol{\pi}$, each of which has a *uniform* start thanks to (15). When it comes to its practical implementation, however, to better control 'randomness' we generate $r = \alpha n$ independent realizations of the Markov chain, among which each group of α realizations start from *each* node i, where α is a positive integer value. To correctly indicate the initial node i, we write $\{X_{i,0}^j, X_{i,1}^j, \ldots, X_{i,l}^j\}$ to denote the j-th realization of length l that starts from node i, for $j = 1, 2, \ldots, \alpha$. Then we see that each group of α realizations, i.e., $X_{i,k}^1, X_{i,k}^2, \ldots, X_{i,k}^\alpha$, individually estimates each diagonal entry $(\mathbf{P}^k)_{i,i}$ in (14), and thus the estimator $\hat{h}_k(r)$ in (16) is now redefined to be

$$\hat{h}_k(r) = \sum_{i=1}^{n} \frac{1}{\alpha} \sum_{j=1}^{\alpha} \mathbb{1}\{X_{i,k}^j = X_{i,0}^j\}.$$

This is then used for the MC estimator $\hat{\mathcal{K}}_l(r)$ in (19). Our MC algorithm is summarized in Algorithm 1.

Theorem 3. *The time complexity of Algorithm 1 is $O(r \cdot l)$. The space complexity of Algorithm 1 is $O(m + n)$.*

Proof. The nested loop structure of Algorithm 1 makes the overall time complexity $O(r \cdot l)$. For the space complexity, it requires $O(m+n)$ to store up to $2m$ non-zero transition probabilities and up to n possible self-transition probabilities (along with their associated 'directional' edges), and $O(1)$ to keep track of the current position of each realization of the chain to check the 'if-condition' in line 7, together with a frequency counter. Thus, we have $O(m + n)$. □

Thanks to the inherent algorithmic parallelism of our MC algorithm, we are able to leverage the power of parallel computing to greatly reduce the runtime of Algorithm 1. Specifically, we develop

Algorithm 1: Monte Carlo Algorithm

Input: transition matrix $\mathbf{P}$, chain length l, the number of realizations $r = \alpha n$

1 $C \leftarrow 0$
2 **for** $j = 1, 2, \ldots, \alpha$ **do**
3 **for** $i = 1, 2, \ldots, n$ **do**
4 $X_0 \leftarrow i$; $X \leftarrow X_0$
5 **for** $k = 1, 2, \ldots, l$ **do**
6 $X \overset{\mathbf{P}}{\leftarrow} RandomSelect(Neighbor(X))$
7 **if** $X == X_0$ **then**
8 $C \leftarrow C + 1$
9 $\mathcal{K} \leftarrow C/\alpha + n - 1 - l$
10 **return** $\mathcal{K}$

a parallel implementation of our MC algorithm on a GPU based on the NVIDIA CUDA programming framework[2], as described in Algorithm 2. Note that accessing a GPU is no longer an expensive option, since a modern computer is typically equipped with a GPU and it can also be used via a free cloud service like Google Colab[3], which is the one used in this paper. Given a graph G, chain length l, and the number of realizations r, our GPU implementation efficiently computes (approximates) the Kemeny's constant $\mathcal{K}$ of the Markov chain on G. We first create a list of initial nodes $\boldsymbol{V}$, in which each element is the initial starting node of each realization of the Markov chain. Note that α realizations start from the same node. The GPU kernel parallels the operation of each realization by assigning it to each thread, and simply counts how many times each realization hits its initial starting node. The hitting frequencies are all added together to compute the Kemeny's constant $\mathcal{K}$. This GPU implementation is illustrated in Figure 1. We refer to Appendix A for more details on the GPU implementation.

Suppose that the number of threads that can be created and the number of available GPU cores are both greater than the number of realizations. Then we have the following.

Theorem 4. *The time complexity of Algorithm 2 is $O(l + r)$. The space complexity of Algorithm 2 is $O(m + n + r)$.*

Proof. Since each realization of the chain can be assigned to a different thread running on a different GPU core, its time complexity is bounded by the chain length l, i.e., $O(l)$. The 'for-loop' in lines 9 and 10 takes $O(r)$, which in turn makes the time complexity $O(l+r)$. For the space complexity, it requires storing r frequency counters, the initial and current positions of r realizations, and up to $2m$ non-zero transition probabilities and up to n possible self-transition probabilities, which leads to a total of $O(m + n + r)$. □

It is worth noting that the number of available GPU cores can be smaller than the number of realizations in practice, although the number of threads that can be created is generally large enough. In such a case, the time complexity of Algorithm 2 becomes $O(\lceil r/T \rceil \cdot l + r)$, where T is the number of cores. We also emphasize that the parallel implementation of our MC algorithm here is based on a single GPU, but it can be readily extended to multiple GPUs to realize 'perfect' parallelization in case the number of cores on a single GPU is not enough to do so.

[2] https://developer.nvidia.com/cuda-toolkit
[3] https://colab.research.google.com/

Algorithm 2: Parallel Implementation on a GPU

Input: transition matrix $\mathbf{P}$, chain length l, the number of realizations $r = \alpha n$

Create a list of initial nodes $\boldsymbol{V} = [V_1, V_2, \ldots, V_r] =$ [repeat each element in $[1, 2, \ldots, n]$ α times]
$c_i \leftarrow 0$ for $i = 1, 2, \ldots, r$; $C \leftarrow 0$
GPU kernel: Parallel **for** $i = (1, 2, \ldots, r)$ **do**
  $X_0 \leftarrow V_i$; $X \leftarrow X_0$
  for $k = 1, 2, \ldots, l$ **do**
    $X \overset{\mathrm{P}}{\leftarrow} RandomSelect(Neighbor(X))$
    if $X == X_0$ **then**
      $c_i \leftarrow c_i + 1$
for $i = 1, 2, \ldots, r$ **do**
  $C \leftarrow C + c_i$
$\mathcal{K} \leftarrow C/\alpha + n - l - 1$
return $\mathcal{K}$

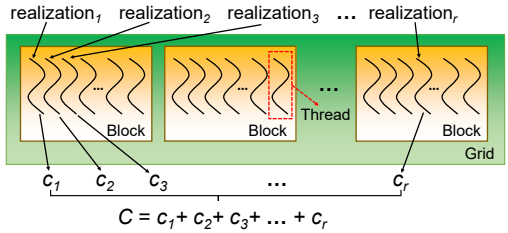


Figure 1: An illustration of the GPU implementation.

5 EXPERIMENT RESULTS

In this section, we present extensive experiment results to demonstrate the effectiveness of our MC algorithm on estimating the Kemeny's constant $\mathcal{K}$, especially for large graphs.

Datasets. We consider 13 real-world undirected network datasets from SNAP[4] and Network Repository[5], which are listed in an ascending order of graph size n in Table 1.[6] We classify four graphs whose sizes are smaller than 50K as small graphs and the rest of them as large graphs. In this experimental evaluation, we focus on computing the Kemeny's constant $\mathcal{K}$ of a simple random walk on the largest strongly connected components (LSCC) of a graph, as used in [28]. The statistics in Table 1 are for the LSCCs. The datasets include infrastructure networks, online social networks, collaboration networks, and communication networks, where the largest one has about 1.7 million nodes.

Experiment setup. We implement two baseline methods, which are to directly compute the eigenvalues of the transition matrix $\mathbf{P}$ and those of its normalized Laplacian matrix $\mathbf{L}$ in order to compute $\mathcal{K}$. We refer to them as 'Eigen-P' and 'Eigen-N', respectively. Note that they are only feasible for *small* graphs. We also implement ApproxKemeny from scratch, as described in [28], since its implementation code is not publicly available. See Appendix A for more details on the implementation. For ApproxKemeny, there is a parameter ϵ_A that controls the precision of its estimation of $\mathcal{K}$ and

[4]http://snap.stanford.edu/data/
[5]http://networkrepository.com/
[6]We are not able to find all the datasets in [28] due to the policy change of KONECT, where the datasets are no longer freely available. Nonetheless, we still have the six datasets in common compared to the ones in [28].

Table 1: Graph statistics

	# Nodes (n)	# Edges (m)	Chain length l
HEP-TH	8638	24806	3566
Astro-ph	17903	196972	1640
CAIDA	26475	53381	869
EmailEnron	33696	180811	3124
Brightkite	56739	212945	25791
wiki-Talk	92117	360767	25791
Gowalla	196591	950327	25791
com-DBLP	317080	1049866	25791
Amazon	334863	925872	25791
soc-flickr	513969	3190452	25791
soc-digg	770799	5907132	25791
Youtube	1134890	2987624	25791
Skitter	1694616	11094209	25791

ϵ_A is set to be from 0.05 to 0.3 in [28]. In this paper, we report the results of ApproxKemeny when $\epsilon_A = 0.1, 0.2, 0.3$. We observed that the case with $\epsilon_A = 0.05$ makes the runtime of ApproxKemeny way too long without any benefit in the estimation accuracy.

For our MC algorithm, we use the parallel implementation in Algorithm 2, whose details are provided in Appendix A. We first need to determine the length l of the Markov chain, which requires a choice of the value of ϵ and the value of λ_*, as in Theorem 2. For the former, we consider $\epsilon = 0.01n$, which is a function of the graph size n, since $\mathcal{K}$ increases linearly with n, as seen from (5). For the latter, we use the exact value of λ_* for small graphs. When it comes to large graphs, it can be expensive to compute λ_*. Thus, we use a very conservative estimate of λ_*, i.e., $\lambda_* = 0.9995$, which could be much larger than its actual value for some graphs. Their corresponding lengths l are reported in Table 1. Note that the exact value of λ_* may not be readily available in practice and the conservative estimate of λ_* could turn out to be an underestimated one. Thus, to cope with this problem, we will present an updated version of our MC algorithm that does *not* require the knowledge on λ_* in Section 6. In addition to the length l, we also need to determine the number of realizations of the Markov chain, r. In this experimental evaluation, we consider $r = n, 10n, 100n$ to demonstrate how effective our MC algorithm can be in practice. We refer to Appendix B for more details on the experiment setup.

Experiment results. We first demonstrate the efficiency and scalability of our MC algorithm when compared with two baseline methods and ApproxKemeny in terms of the runtime. We report the detailed results in Table 3 in Appendix C. As shown in Figure 2, the runtime of our MC algorithm remains almost the same for small graphs, while it increases when the graph becomes larger and r increases. The latter happens because all the realizations cannot be executed in parallel concurrently, when the number of realizations is greater than the number of available GPU cores, as explained in Section 4. Nonetheless, our algorithm is *far faster* than Eigen-P, Eigen-N, and ApproxKemeny. It only takes *about one second* to compute the Kemeny's constant for small graphs and *a few minutes* for large graphs in the worst case, such as Youtube and Skitter, each of which has more than a million nodes. Note that Eigen-P and Eigen-N are only feasible for small graphs. Furthermore, the improvement of our algorithm over ApproxKemeny becomes more significant for large graphs.

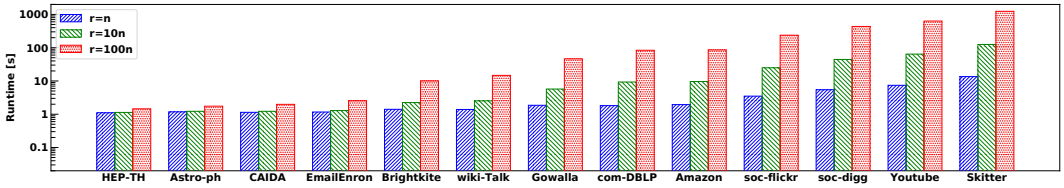


Figure 2: Runtimes of our MC algorithm on a log scale.

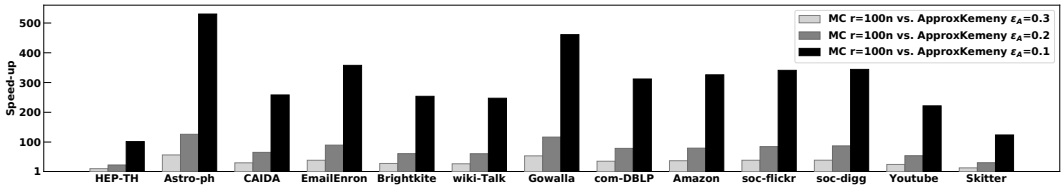


Figure 3: Speed-up by our MC algorithm over ApproxKemeny.

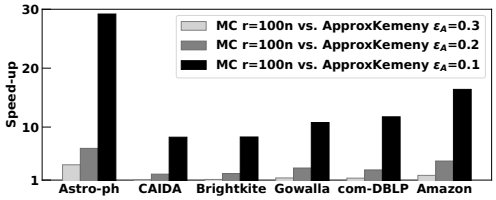


Figure 4: Speed-up by our MC algorithm over ApproxKemeny (based on the runtime results reported in [28]).

We also show the speed-up by our algorithm over ApproxKemeny (with three different choices of ϵ_A) in Figure 3. Here, we use the case with $r = 100n$, which is the slowest (but most accurate) case. It is far faster than ApproxKemeny for all test cases and the speed-up is up to 500×. It is worth noting that the runtimes of ApproxKemeny (of our implementation) on our workstation is over ten times longer than the ones reported in [28]. It may be because of different computing environments and different levels of programming optimization. Thus, we further compare the runtimes of our MC algorithm under the six common datasets with the runtimes of ApproxKemeny reported in [28], as depicted in Figure 4. We observe that the MC algorithm is still faster than ApproxKemeny and the speed-up is up to 30×.

We next demonstrate the accuracy of the MC algorithm for computing the Kemeny's constant. To this end, we measure the relative error $|\mathcal{K} - \widetilde{\mathcal{K}}|/\mathcal{K}$ for each graph, where $\widetilde{\mathcal{K}}$ is our estimated value and $\mathcal{K}$ is the reference value. For the reference value, we use the value of $\mathcal{K}$ obtained by Eigen-P or Eigen-N (both return the same value). Note that they are only feasible for small graphs. Thus, for large graphs, we use the estimated value by ApproxKemeny (with $\epsilon_A = 0.1$) as the reference value. Table 2 presents the mean relative errors along with their standard deviations of 100 runs. We observe that the relative errors remain at a negligible level for all test cases and the accuracy improves with the increase of r. We also find that the accuracy of our MC algorithm is comparable to that of ApproxKemeny. See Table 5 in Appendix C for all the computed values of Kemeny's constant by all four methods.

Table 2: Relative error of the MC algorithm ($\times 10^{-4}$)

r	n	10n	100n
HEP-TH	93.53 ± 71.10	31.69 ± 22.27	9.84 ± 7.81
Astro-ph	42.46 ± 32.25	13.31 ± 9.97	4.08 ± 3.25
CAIDA	28.22 ± 19.83	9.26 ± 7.40	2.68 ± 2.05
EmailEnron	46.59 ± 36.87	13.13 ± 10.05	4.24 ± 3.14
Brightkite	29.55 ± 22.44	10.35 ± 7.03	3.19 ± 2.26
wiki-Talk	18.17 ± 13.57	5.93 ± 4.58	1.89 ± 1.40
Gowalla	13.39 ± 10.38	4.32 ± 3.52	1.45 ± 1.13
com-DBLP	14.66 ± 11.08	5.05 ± 3.38	1.42 ± 1.14
Amazon	22.80 ± 17.03	11.28 ± 6.99	10.78 ± 2.74
soc-flickr	11.79 ± 10.35	3.73 ± 3.00	1.36 ± 0.97
soc-digg	4.63 ± 3.68	1.49 ± 1.04	0.68 ± 0.45
Youtube	7.71 ± 5.62	2.81 ± 2.06	0.86 ± 0.62
Skitter	4.63 ± 4.22	1.80 ± 1.29	0.56 ± 0.38

6 DYNAMIC MONTE CARLO ALGORITHM

We turn our attention to a 'dynamic' version of our MC algorithm to overcome the limitation of its plain 'vanilla' version that requires the value of λ_*. In particular, we empirically demonstrate that this dynamic version not only eliminates the need for the value of λ_* but also further improves the runtime of our algorithm without losing its estimation accuracy.

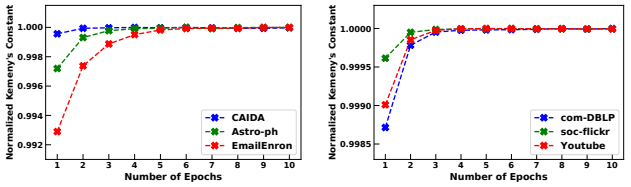
As mentioned in Section 5, despite its remarkable performance, the plain vanilla version of our MC algorithm requires the value of λ_* to determine the length l of the Markov chain. While it can be just a rough and conservative estimate of λ_*, it could possibly be an underestimated one in fact and thus lead to an underestimate of the Kemeny's constant $\mathcal{K}$. To resolve this problem, we first observe the following. From (16) and (19), when increasing the length l without specifying its value, one can naturally expect that the estimated value of $\mathcal{K}$ by our MC algorithm tends to increase and gets saturated after a certain value of l. This is indeed the case, as can be seen from Figure 5, which is obtained by simply running our algorithm (its parallel implementation in Algorithm 2) with varying chain length l. In Figure 5, an 'epoch' refers to the time instance when $\mathcal{K}$ is measured/estimated and its epoch length is Δl. In other words, we report the estimated value of $\mathcal{K}$ by our algorithm every Δl. We also normalize the estimated value of $\mathcal{K}$ at each epoch by the largest value for each graph. The number of realizations is $r = 100n$.

Thus motivated, we develop a dynamic version of our MC algorithm, as described in Algorithm 3, which does not require the knowledge of λ_* nor its rough estimated value. It is still nothing but Algorithm 1, except that each realization of the Markov chain runs

Algorithm 3: Dynamic Monte Carlo Algorithm

Input: transition matrix $\mathbf{P}$, epoch length Δl, the number of realizations $r = \alpha n$

$C \leftarrow 0$; $\mathcal{K} \leftarrow 0$; $\mathcal{K}' \leftarrow \infty$; $l \leftarrow 0$
for $j = 1, 2, \ldots, \alpha$ **do**
  for $i = 1, 2, \ldots, n$ **do**
    $X_i^j \leftarrow i$
while $|\mathcal{K} - \mathcal{K}'| \geq \epsilon_d$ **do**
  $\mathcal{K}' \leftarrow \mathcal{K}$; $l \leftarrow l + \Delta l$
  for $j = 1, 2, \ldots, \alpha$ **do**
    for $i = 1, 2, \ldots, n$ **do**
      $X_0 \leftarrow i$; $X \leftarrow X_i^j$
      for $k = 1, 2, \ldots, \Delta l$ **do**
        $X \overset{\mathbf{P}}{\leftarrow} RandomSelect(Neighbor(X))$
        if $X == X_0$ **then**
          $C \leftarrow C + 1$
      $X_i^j \leftarrow X$
  $\mathcal{K} \leftarrow C/\alpha + n - 1 - l$
return $\mathcal{K}$

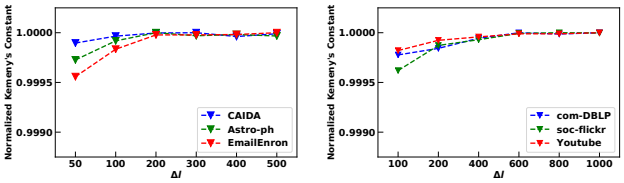


(a) Small graphs; $\Delta l = 200$ **(b) Large graphs;** $\Delta l = 600$

Figure 5: (Normalized) Kemeny's constant estimated by our MC algorithm with varying chain length l.

until a stopping criterion is met instead of running for a predefined length l. Specifically, the dynamic version of our MC algorithm returns the estimated value of $\mathcal{K}$ every Δl, as it runs. This operation continues until the difference between two consecutive estimates of $\mathcal{K}$ is less than a certain threshold, i.e., $|\mathcal{K} - \mathcal{K}'| < \epsilon_d$, where $\mathcal{K}'$ is the previous estimate. In addition, since there is no change in the core algorithm operation, the parallel implementation of the dynamic version can be done in a similar way as in Algorithm 2. See Appendix A for the implementation details.

Before showing the experiment results, we explain how we select the threshold ϵ_d for the stopping criterion and the epoch length Δl in Algorithm 3. First, we consider $\epsilon_d = 0.0005n$ and $\epsilon_d = 0.0001n$, which are again a function of n, since $\mathcal{K}$ grows linearly with n, as seen from (5). As shall be shown below, they are small enough for our dynamic algorithm to obtain an accurate estimate of $\mathcal{K}$ for each graph. Next, to choose the value of Δl, we evaluate its impact on the performance of our dynamic algorithm. We consider different values of Δl and present the corresponding estimation results by our dynamic algorithm on three small graphs and three large graphs as representative results in Figure 6, where we normalize the estimated value of $\mathcal{K}$ with each value of Δl by the largest value for each graph. The number of realizations is $r = 100n$. We observe that the final estimate of $\mathcal{K}$ by our dynamic algorithm becomes saturated around when $\Delta l = 200$ for small graphs and when $\Delta l = 600$ for large graphs, respectively, while the runtime of our dynamic algorithm increases



(a) Small graphs **(b) Large graphs**

Figure 6: Impact of Δl when $\epsilon_d = 0.0001n$.

with the value of Δl. Thus, we choose $\Delta l = 200$ for small graphs and $\Delta l = 600$ for large graphs. It is still worth noting that even with a small value of Δl, e.g., $\Delta l = 50$ for small graphs and $\Delta l = 100$ for large graphs, our dynamic algorithm still provides a reasonably accurate estimate of $\mathcal{K}$.

We evaluate the performance of our dynamic algorithm (with the aforementioned values of ϵ_d and Δl) under the datasets used in Section 5 and observe that it exhibits excellent performance in terms of both computational efficiency and accuracy. Here we again focus on the case with $r = 100n$. We report the detailed results in Table 4 in Appendix C. As shown in Figure 7, we notice that the runtime of our dynamic algorithm is more or less the same as that of its plain vanilla version for small graphs and Amazon graph, while it achieves up to 11× speed-up over the vanilla version for the other eight graphs. Considering the fact that the plain vanilla version of our algorithm is already faster than ApproxKemeny for all the datasets, its dynamic version is *far more* computationally efficient than ApproxKemeny. In addition, to show the estimation accuracy of our dynamic algorithm, we compute the mean relative errors, as was done for its vanilla version in Table 2. While we provide the results in Table 6 in Appendix C, we observe that the estimation errors are again at a negligible level. To summarize, our dynamic algorithm not only resolves the issue with requiring the value of λ_* but also further improves the runtime without losing its estimation accuracy.

7 DISCUSSION

So far, we have assumed that the transition matrix $\mathbf{P}$ (or its Laplacian matrix $\mathcal{L}$) on a graph G is known explicitly for computing its corresponding Kemeny's constant $\mathcal{K}$ by ApproxKemeny and both vanilla and dynamic versions of our MC algorithm. One may wonder what if the transition matrix $\mathbf{P}$ is not known explicitly, or the graph G needs to be crawled or explored to access nodes and edges and to find their associated transition probabilities P_{ij}. While it is infeasible with ApproxKemeny that needs the entire Laplacian matrix $\mathcal{L}$ and solves its Laplacian system of linear equations, the dynamic version of our algorithm can still be used even in such a case. We below elaborate on this extension.

Recall that our MC algorithm in both versions (Algorithms 1 and 3) requires a different group of α realizations to start from each node i. While it was designed to better regulate the randomness in practice, as explained in Section 4.1, each realization just needs to start from an initial node that is chosen *uniformly at random* from the node set V. Thus, the dynamic version of our algorithm can be viewed as just launching r parallel random walks to move over the graph G according to transition probabilities P_{ij} until the stopping criterion is met, as long as they independently start from initial nodes that are chosen uniformly at random. Then, the question

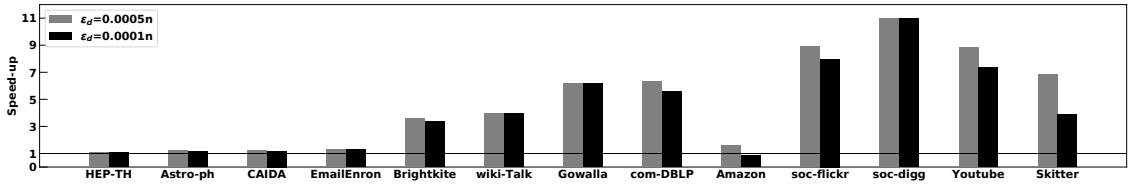


Figure 7: Speed-up by the dynamic version of our MC algorithm over its vanilla version when $r = 100n$.

boils down to how to achieve the latter, i.e., how to realize such a *uniform* start from the node set V of G that is unknown but needs to be discovered.

The famous Metropolis-Hastings (MH) algorithm [22] can come to the rescue. Specifically, as used in [6, 11, 23], it allows us to construct the following transition matrix $\mathbf{P}^{mh} := [P_{ij}^{mh}]$ such that its stationary distribution $\boldsymbol{\pi}^{mh}$ is a uniform distribution on V.

$$P_{ij}^{mh} = \begin{cases} \min\left\{\frac{1}{d_i}, \frac{1}{d_j}\right\} & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E, i \neq j, \end{cases} \tag{23}$$

with $P_{ii}^{mh} = 1 - \sum_{j \neq i} P_{ij}^{mh}$. The operation here can be made *locally*. Assuming that the current node is i, it can be interpreted as proposing one of its neighbors as a next node with probability $1/d_i$ and accepting the proposed move with probability $\min\{1, d_i/d_j\}$. If this proposed move is rejected, the next node remains the same as i. Therefore, we launch r parallel random walks to initially move over the graph G according to $\mathbf{P}^{mh}$ until they get 'mixed' (or they are in the stationary regime), and then execute our dynamic algorithm based on the parallel random walks that now move over G according to $\mathbf{P}$. Note that transition probabilities P_{ij} are still locally available when vising node i, although they are not available in their entirety. We refer to [6, 23] for other methods, in addition to the above MH algorithm, on generating a uniform node in a graph.

8 CONCLUSION

We have studied the problem of computing the Kemeny's constant of a Markov chain on large graphs, where its direct computation is generally infeasible. We proposed a computationally efficient and scalable Monte Carlo algorithm for estimating the Kemeny's constant and provided a mathematical analysis on its approximation accuracy. We presented its parallel implementation on a GPU and demonstrated its superiority over the state-of-the-art algorithm called ApproxKemeny based on a dozen real-world graphs. The benefit of our algorithm over ApproxKemeny in the computational efficiency remains significant even for large graphs and its speed-up is up to 500×, while its estimation accuracy is comparable to that of ApproxKemeny. In addition, we presented a 'dynamic' refinement of our algorithm to make it work without the knowledge of λ_* and demonstrated that it even further speeds up the computation without losing its estimation accuracy. We finally discussed how this dynamic algorithm can be adopted in the case where the transition matrix $\mathbf{P}$ is not known explicitly.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation under grants IIS-1908375 and CNS-2007828, and a gift from NVIDIA Corporation. C. Lee is the corresponding author.

REFERENCES

[1] D. Aldous and J. Fill. 2002. Reversible Markov chains and random walks on graphs.
[2] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. 2007. Monte Carlo methods in PageRank computation: When one iteration is sufficient. *SIAM J. Numer. Anal.* 45, 2 (2007), 890–904.
[3] H. Avron and S. Toledo. 2011. Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix. *J. ACM* 58, 2 (2011), 1–34.
[4] S. Boyd, P. Diaconis, and L. Xiao. 2004. Fastest mixing Markov chain on a graph. *SIAM Rev.* 46, 4 (2004), 667–689.
[5] C.-K. Chau and P. Basu. 2009. Exact analysis of latency of stateless opportunistic forwarding. In *IEEE INFOCOM*. 828–836.
[6] F. Chiericetti, A. Dasgupta, R. Kumar, S. Lattanzi, and T. Sarlós. 2016. On sampling nodes in a network. In *WWW*. 471–481.
[7] V. Cholvi, P. Felber, and E. Biersack. 2004. Efficient search in unstructured peer-to-peer networks. *Eur. Trans. Telecommun.* 15, 6 (2004), 535–548.
[8] F. R. Chung and F. C. Graham. 1997. *Spectral graph theory*. Number 92. American Mathematical Soc.
[9] P. G. Doyle. 2009. The Kemeny constant of a Markov chain. *arXiv preprint arXiv:0909.2636* (2009).
[10] P. Drineas, R. Kannan, and M. W. Mahoney. 2006. Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *SIAM J. Comput.* 36, 1 (2006), 158–183.
[11] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. 2011. Practical recommendations on crawling online social networks. *IEEE J. Sel. Areas Commun.* 29, 9 (2011), 1872–1892.
[12] C. Gkantsidis, M. Mihail, and A. Saberi. 2004. Random walks in peer-to-peer networks. In *IEEE INFOCOM*. 130.
[13] J. J. Hunter. 2014. The role of Kemeny's constant in properties of Markov chains. *Communications in Statistics-Theory and Methods* 43, 7 (2014), 1309–1321.
[14] M. F. Hutchinson. 1989. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics-Simulation and Computation* 18, 3 (1989), 1059–1076.
[15] H. Ji, M. Mascagni, and Y. Li. 2013. Convergence Analysis of Markov Chain Monte Carlo Linear Solvers Using Ulam–von Neumann Algorithm. *SIAM J. Numer. Anal.* 51, 4 (2013), 2107–2122.
[16] J. G. Kemeny and J. L. Snell. 1976. *Markov chains*. Springer-Verlag.
[17] R. E. Kooij and J. L. Dubbeldam. 2020. Kemeny's constant for several families of graphs and real-world networks. *Discrete Appl. Math.* 285 (2020), 96–107.
[18] R. Kyng and S. Sachdeva. 2016. Approximate gaussian elimination for laplacians-fast, sparse, and simple. In *IEEE FOCS*. 573–582.
[19] C.-H. Lee and D. Y. Eun. 2015. On the efficiency-optimal Markov chains for distributed networking applications. In *IEEE INFOCOM*. 1840–1848.
[20] M. Levene and G. Loizou. 2002. Kemeny's constant and the random surfer. *The American Mathematical Monthly* 109, 8 (2002), 741–745.
[21] L. Lovász. 1993. Random walks on graphs: A survey. *Combinatorics, Paul Erdos is Eighty* 2, 1 (1993), 1–46.
[22] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. *J. Chem. Phys.* 21, 6 (1953), 1087–1092.
[23] A. Nazi, Z. Zhou, S. Thirumuruganathan, N. Zhang, and G. Das. 2015. Walk, not wait: Faster sampling over online social networks. *VLDB Endowment* 8, 6 (2015), 678–689.
[24] J. L. Palacios and J. M. Renom. 2010. Bounds for the Kirchhoff index of regular graphs via the spectra of their random walks. *IJQC* 110, 9 (2010), 1637–1641.
[25] R. Patel, P. Agharkar, and F. Bullo. 2015. Robotic surveillance and Markov chains with minimal weighted Kemeny constant. *IEEE TAC* 60, 12 (2015), 3156–3167.
[26] J. Risson and T. Moors. 2006. Survey of research towards robust peer-to-peer networks: Search methods. *Computer Networks* 50, 17 (2006), 3485–3521.
[27] X. Wang, J. L. Dubbeldam, and P. Van Mieghem. 2017. Kemeny's constant and the effective graph resistance. *Linear Algebra Appl.* 535 (2017), 231–244.
[28] W. Xu, Y. Sheng, Z. Zhang, H. Kan, and Z. Zhang. 2020. Power-Law Graphs Have Minimal Scaling of Kemeny Constant for Random Walks. In *WWW*. 46–56.
[29] Z. Zhang, W. Xu, and Z. Zhang. 2020. Nearly Linear Time Algorithm for Mean Hitting Times of Random Walks on a Graph. In *WSDM*. 726–734.
[30] M. Zhong and K. Shen. 2006. Popularity-Biased Random Walks for Peer-to-Peer Search under the Square-Root Principle. In *IPTPS*.

A IMPLEMENTATION DETAILS

We implement Eigen-P and Eigen-N in MATLAB. We implement ApproxKemeny in Julia v1.5.2 and provide a code snippet in Figure 8. We refer to Section 3 and [28] for more details on its algorithm operation. It is worth noting that in our implementation of ApproxKemeny, we utilize the Cholesky factorization algorithm to solve the Laplacian system of linear equations. This algorithm is named 'chol-lap' solver in the Julia library[7], as shown in line 11 of Figure 8. However, as in [28], ApproxKemeny originally adopts an approximation algorithm in [18], which is based on the sparse Cholesky factorization. It is named 'approxchol-lap' solver in the Julia library and is known to be the fastest one. While this solver takes a 'tolerance' value as an (optional) input to control the level of its approximation accuracy, we observe that it is fast at the sacrifice of accuracy (with high tolerance values). We further observe that the tolerance value used in [28] is generally smaller than 1.0×10^{-11} for all the datasets. The way of determining the tolerance value in [28] is provided in line 12 of Figure 8. It turns out that such a small tolerance value makes the 'approxchol-lap' solver even (about three to five times) *slower* than the 'chol-lap' solver. Thus, instead of using the 'approxchol-lap' solver, we use the 'chol-lap' solver in our implementation of ApproxKemeny. We here omit the detailed experiment results regarding the solvers due to space constraint.

We implement two versions of our MC algorithm (both vanilla and dynamic versions) in Python and present a code snippet of the GPU kernel in Figure 9, which is the common core routine for both versions.[8] We here focus on the implementation of a simple random walk, while it can be readily extended for any Markov chains. We use the compressed sparse row (CSR) data structure to store the transition matrix $\mathbf{P}$ of the Markov chain on G. Note that the random walk is implemented based on a random number generator (RNG). We use Numba's GPU RNG[9] to ensure independent parallel generations, which adopts the xoroshiro128+ algorithm, as shown in line 13 of Figure 9. Note also that the operation of adding the hitting frequencies of all the realizations can be implemented by an atomic addition on the GPU, as shown in line 20 of Figure 9.

B ADDITIONAL EXPERIMENT SETUP

We conduct the experiments of Eigen-P, Eigen-N and ApproxKemeny on a Linux workstation with two Intel Xeon 2.2-GHz CPUs and 64-GB RAM. We run the vanilla and dynamic versions of our MC algorithms on an NVIDIA Tesla V100 SXM2 GPU with CUDA toolkit 10.1, which is available on Google Colab. We run both versions of our algorithm 100 times for all test cases and report their average values along with the standard deviations. As was done in [28], however, we run ApproxKemeny only once and report its experiment results, although it is a randomized algorithm. Note that we ran ApproxKemeny multiple times and observed that the results remain quite consistent over different runs. In addition, we exclude the time of loading and processing raw data, when we measure the runtime of each method in this paper.

[7] https://github.com/danspielman/Laplacians.jl
[8] Our code is available at https://github.com/xhuang2016/Kemeny-computation.
[9] https://numba.pydata.org/numba-doc/latest/cuda/random.html

```
1  using Laplacians, LinearAlgebra, Arpack
2  function lps(adj)
3     la = lap(adj)
4     n = size(adj)[1]
5     for i in 1:n
6        deg[i] = la[i,i]
7        degsq[i] = sqrt(deg[i])
8     end
9     m = 0.5*sum(deg)
10    epsilonA = 0.3
11    sol = chol_lap(adj)
12    # tol = epsilonA*(n^(-2.5))/(3*sqrt(2))
13    # sol = approxchol_lap(adj, tol=tol)
14    U = wtedEdgeVertexMat(adj)
15    M = ceil(Int,48*log(2*n)*epsilonA^(-2))
16    s = 0.0
17    for i in 1:M
18       dx = degsq.*(2.0*rand(0:1, n)-ones(n))
19       y = dx-deg*transpose(ones(n))*dx/(2.0*m)
20       z = sol(y)
21       s = s+(norm(U*z))^2
22    end
23    return s/M
```

Figure 8: Code snippet of ApproxKemeny.

```
1  from numba import cuda
2  from numba.cuda.random import
       create_xoroshiro128p_states,
       xoroshiro128p_uniform_float32
3  @cuda.jit
4  def GPU_MC(initial_start_nodes, start_nodes, indptr,
       indices, hitting, sum_hitting, rng_states, r, l):
5     thread_id = cuda.grid(1)
6     if thread_id < r:
7        curr_node = start_nodes[thread_id]
8        hitting[thread_id] = 0
9        for i in range(0, l):
10          start_idx = indptr[curr_node]
11          end_idx = indptr[curr_node + 1]
12          neighbors = indices[start_idx: end_idx]
13          rand_float =
                xoroshiro128p_uniform_float32(rng_states,
                thread_id)
14          choice = int(rand_float * len(neighbors))
15          next_node = neighbors[choice]
16          if next_node == initial_start_nodes[thread_id]:
17             hitting[thread_id] += 1
18          curr_node = next_node
19       start_nodes[thread_id] = curr_node
20       cuda.atomic.add(sum_hitting,0,hitting[thread_id])
```

Figure 9: Code snippet of the GPU kernel for the parallel implementation.

C DETAILED RESULTS

We provide the detailed results of Eigen-P, Eigen-N, ApproxKemeny and our MC algorithm (its vanilla version) in Table 3 and Table 5. We also provide the performance results of the dynamic version of our algorithm in Table 4 and its relative error results in Table 6.

Table 3: Runtime [s]

Dataset	Eigen-P / Eigen-N	ApproxKemeny		MC algorithm		
		ϵ_A		r	Overall	Kernel
HEP-TH	259.017 / 225.644	0.3	15.274	n	1.107	0.314
		0.2	33.257	10n	1.134	0.326
		0.1	147.626	100n	1.454	0.447
Astro-ph	1944.000 / 187.821	0.3	99.031	n	1.179	0.324
		0.2	220.322	10n	1.226	0.340
		0.1	926.875	100n	1.748	0.455
CAIDA	4247.000 / 1022.100	0.3	59.849	n	1.142	0.312
		0.2	129.505	10n	1.225	0.328
		0.1	512.460	100n	1.984	0.443
EmailEnron	15981.000 / 1095.700	0.3	98.200	n	1.167	0.319
		0.2	226.992	10n	1.290	0.374
		0.1	906.230	100n	2.533	0.815
Brightkite	-	0.3	279.593	n	1.406	0.534
		0.2	609.015	10n	2.242	1.241
		0.1	2535.510	100n	9.991	7.672
wiki-Talk	-	0.3	396.922	n	1.391	0.628
		0.2	894.091	10n	2.523	1.636
		0.1	3643.216	100n	14.732	11.857
Gowalla	-	0.3	2465.796	n	1.859	0.843
		0.2	5367.000	10n	5.722	4.290
		0.1	21219.222	100n	46.030	40.515
com-DBLP	-	0.3	2974.014	n	1.814	1.031
		0.2	6596.251	10n	9.304	7.851
		0.1	26133.130	100n	83.778	74.998
Amazon	-	0.3	3175.491	n	1.946	1.187
		0.2	6783.415	10n	9.626	7.966
		0.1	27826.857	100n	85.368	75.907
soc-flickr	-	0.3	9223.951	n	3.501	2.643
		0.2	20092.785	10n	24.889	22.809
		0.1	81039.405	100n	237.622	224.859
soc-digg	-	0.3	16966.402	n	5.451	4.488
		0.2	37898.909	10n	44.349	41.735
		0.1	149844.868	100n	435.375	414.885
Youtube	-	0.3	15625.003	n	7.412	6.348
		0.2	34183.074	10n	64.188	60.648
		0.1	140241.327	100n	632.674	605.491
Skitter	-	0.3	16372.912	n	13.536	12.322
		0.2	37786.870	10n	125.057	120.136
		0.1	153644.728	100n	1240.976	1199.908

Table 4: Performance of our dynamic algorithm

		Runtime [s]		Kemeny's constant
	ϵ_d	Overall	Kernel	
HEP-TH	0.0005n	1.325	0.412	16183.44 ± 16.89
	0.0001n	1.354	0.441	16186.59 ± 18.50
Astro-ph	0.0005n	1.468	0.353	22935.96 ± 12.52
	0.0001n	1.487	0.372	22937.73 ± 12.90
CAIDA	0.0005n	1.648	0.346	31928.86 ± 11.51
	0.0001n	1.678	0.376	31931.71 ± 11.05
EmailEnron	0.0005n	1.923	0.448	45234.00 ± 18.52
	0.0001n	1.991	0.517	45236.50 ± 21.50
Brightkite	0.0005n	2.797	0.764	80900.25 ± 24.07
	0.0001n	2.993	0.953	80910.83 ± 27.29
wiki-Talk	0.0005n	3.718	0.983	102480.77 ± 14.63
	0.0001n	3.709	0.982	102478.26 ± 11.73
Gowalla	0.0005n	7.452	2.150	271688.15 ± 45.23
	0.0001n	7.463	2.150	271694.83 ± 44.15
com-DBLP	0.0005n	13.292	5.497	581181.99 ± 95.09
	0.0001n	15.039	7.235	581167.76 ± 103.76
Amazon	0.0005n	53.715	45.577	882944.21 ± 267.78
	0.0001n	98.816	90.697	884973.02 ± 249.79
soc-flickr	0.0005n	26.652	12.785	801000.18 ± 113.64
	0.0001n	29.778	15.912	800990.47 ± 102.18
soc-digg	0.0005n	39.638	19.549	858209.23 ± 44.56
	0.0001n	39.603	19.546	858200.12 ± 49.05
Youtube	0.0005n	71.798	42.232	1754463.35 ± 179.51
	0.0001n	85.817	56.251	1754526.35 ± 172.83
Skitter	0.0005n	181.943	138.512	2085191.05 ± 121.76
	0.0001n	321.462	278.046	2086621.88 ± 126.97

Table 5: Kemeny's constant (± shows standard deviation)

	Eigen-P/ Eigen-N	ApproxKemeny		MC algorithm	
		ϵ_A		r	
HEP-TH	16185.03	0.3	16182.32	n	16186.02 ± 190.15
		0.2	16185.48	10n	16185.14 ± 62.69
		0.1	16185.39	100n	16185.13 ± 20.33
Astro-ph	22937.00	0.3	22937.71	n	22937.55 ± 122.30
		0.2	22937.42	10n	22937.31 ± 38.15
		0.1	22936.88	100n	22937.09 ± 11.96
CAIDA	31931.00	0.3	31931.53	n	31930.41 ± 110.13
		0.2	31930.84	10n	31930.33 ± 37.85
		0.1	31931.34	100n	31930.18 ± 10.73
EmailEnron	45236.00	0.3	45234.74	n	45236.51 ± 268.76
		0.2	45236.17	10n	45236.31 ± 74.81
		0.1	45236.46	100n	45236.24 ± 23.87
Brightkite	-	0.3	80915.60	n	80885.67 ± 299.62
		0.2	80911.99	10n	80909.19 ± 101.09
		0.1	80904.26	100n	80906.74 ± 31.53
wiki-Talk	-	0.3	102480.96	n	102498.29 ± 231.75
		0.2	102479.73	10n	102486.18 ± 76.62
		0.1	102481.04	100n	102480.41 ± 24.15
Gowalla	-	0.3	271688.97	n	271702.26 ± 460.20
		0.2	271700.54	10n	271700.13 ± 151.02
		0.1	271690.07	100n	271695.75 ± 49.55
com-DBLP	-	0.3	581140.76	n	581209.07 ± 1067.52
		0.2	581099.40	10n	581175.35 ± 353.09
		0.1	581185.13	100n	581195.61 ± 104.95
Amazon	-	0.3	885417.75	n	884561.00 ± 2318.19
		0.2	885629.18	10n	884606.34 ± 701.29
		0.1	885549.04	100n	884594.18 ± 242.75
soc-flickr	-	0.3	801046.77	n	801146.45 ± 1249.07
		0.2	801043.69	10n	801057.63 ± 379.94
		0.1	801005.04	100n	801010.20 ± 133.46
soc-digg	-	0.3	858137.83	n	858233.50 ± 499.82
		0.2	858147.77	10n	858210.82 ± 141.17
		0.1	858144.32	100n	858196.93 ± 46.60
Youtube	-	0.3	1754534.05	n	1754516.49 ± 1671.88
		0.2	1754523.47	10n	1754522.61 ± 604.74
		0.1	1754433.25	100n	1754506.14 ± 170.62
Skitter	-	0.3	2087138.18	n	2087146.76 ± 1306.91
		0.2	2087110.36	10n	2087031.49 ± 454.86
		0.1	2087093.20	100n	2087091.08 ± 139.75

Table 6: Relative error of our dynamic algorithm ($\times 10^{-4}$)

ϵ_d	0.0005n	0.0001n
HEP-TH	9.97 ± 7.00	9.43 ± 6.53
Astro-ph	4.42 ± 3.23	4.37 ± 3.56
CAIDA	2.96 ± 2.17	2.73 ± 2.14
EmailEnron	3.44 ± 2.26	3.79 ± 2.88
Brightkite	2.52 ± 1.66	2.75 ± 2.11
wiki-Talk	1.16 ± 0.83	0.94 ± 0.70
Gowalla	1.41 ± 0.89	1.33 ± 0.95
com-DBLP	1.31 ± 0.98	1.47 ± 1.06
Amazon	29.41 ± 3.02	6.50 ± 2.82
soc-flickr	1.15 ± 0.83	1.05 ± 0.74
soc-digg	0.79 ± 0.47	0.72 ± 0.48
Youtube	0.82 ± 0.63	0.86 ± 0.72
Skitter	9.20 ± 0.58	2.34 ± 0.61