BloomWalk and CuckooWalk: Fast Random Walks utilizing Probabilistic Data Structure

Ren Inayoshi, Han Nay Aung, Hiroyuki Ohsaki
Graduate School of Science and Technology
Kwansei Gakuin University
Email:{ren, hannayaung, ohsaki}@kwansei.ac.jp

2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC) | 979-8-3503-7696-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/COMPSAC61105.2024.00055

***Abstract*—Random walks are being used for target node search and graph exploration in various fields, including communications and social networking. However, frequent revisits of the random walk agent to the same node degrade the efficiency of node search and graph exploration. To address this issue and improve the efficiency of node search and graph exploration, a variety of random walk-based algorithms with history, such as self-avoiding random walk (SARW) and k-history random walk (k-History), have been developed. Although these approaches accelerate node search and graph exploration with a random walk agent, they require the agent to have a non-negligible amount of memory space to store many nodes on large-scale graphs. To address this issue, it is essential to clarify efficient memory management strategies for random walk agents. In this paper, we propose novel random walk-based algorithms called BloomWalk and CuckooWalk, in which an agent performs history-based random walks on a graph using a probabilistic data structure to record previously visited nodes in memory efficiently. Experimental results demonstrate that BloomWalk and CuckooWalk enable efficient node search on unknown graphs even with the very limited memory capacity.**

***Index Terms*—BloomWalk, CuckooWalk, random walk, probabilistic data structure, target node search, graph exploration, Bloom filter, Cuckoo filter**

I. Introduction

Random walks [1, 2] are widely used for target node search and graph exploration in a large number of applications, such as web page ranking [3, 4], node centrality assessment [5, 6], and community detection [7]. Therefore, improving the efficiency of random walks on graphs is essential, as they are foundational to these advanced applications.

In particular, understanding and enhancing random walks on a graph is crucial for achieving high-quality and reliable protocols, controls, applications, and services in large-scale communication networks. In small-scale and static communication networks, traditional deterministic algorithms may suffice in many cases. However, in large-scale and dynamic communication networks, an entity in the network cannot possess global knowledge of the entire network. Such an entity can only have partial knowledge (i.e., local information) in their vicinity [6, 8].

For instance, in dynamic large-scale networks, algorithms designed for static graphs, such as BFS (Breadth-First Search) and DFS (Depth-First Search), are virtually useless due to the network's dynamics. In such environments, a class of algorithms based on random walks, which rely solely on the local information of an agent (i.e., the random walker), often represents the only viable solution [9, 10].

In the literature, various types of random walks have been proposed, each with distinct characteristics. These include simple random walk (SRW), non-backtracking random walk (NBRW) [11, 12], k-history random walk (k-History), biased random walk (BiasedRW) [13, 14], vicinity-avoiding random walk (VARW) [15], and self-avoiding random walk (SARW) [16].

In the context of a random walk on an unknown graph, frequent revisits to the same node reduce the efficiency of node search and graph exploration. To address this issue and avoid repeatedly revisiting the same node, history-based techniques such as NBRW and SARW are used. In these algorithms, a mobile agent keeps a record of nodes it has previously visited. However, these methods have limited effectiveness when the agent's memory size is insufficient. For large-scale graphs, a substantial amount of memory space is required to record the number of visited nodes. Addressing this issue necessitates investigating efficient memory management strategies for random walk agents.

On the other hand, several databases, caches, routers, and storage systems with limited memory sizes adopt probabilistic data structures to test if a given element is in a set, with some small false positive probability [17-19]. Researchers have developed several probabilistic data structures, such as the Bloom filter (BF) [17], Cuckoo filter[18], and Xor Filter [20], that are popular because of their desirable properties.

However, to the best of our knowledge, it has not been studied in the literature whether probabilistic data structures are effective for improving the performance of random walk algorithms. We believe that the combination of random walk algorithms on a graph and probabilistic data structures should significantly advance the state-of-the-art of many applications based on random walk algorithms.

In this paper, we aim to answer the following research questions:

- Can random walks on a graph be made more efficient using probabilistic data structures? If so, to what extent can they be made more efficient compared to conventional random walk algorithms using deterministic data structures?
- In the context of random walks, how should probabilistic data structures be combined and integrated into a mobile

2836-3795/24/$31.00 ©2024 IEEE
DOI 10.1109/COMPSAC61105.2024.00055

agent to achieve optimal performance?
- To what extent does the uncertainty inherent in probabilistic data structures negatively impact the efficiency of node search and graph exploration with random walks, and what strategies can be employed to minimize these effects?

To address these research questions, we propose two types of random walk algorithms, BloomWalk and CuckooWalk. These algorithms efficiently manage the agent's memory by incorporating probabilistic data structures, commonly employed in implementations of spam filters and analogous applications. Additionally, we analyze the efficiency of BloomWalk and CuckooWalk through simulation experiments on seven different types of graphs.

The main contributions of this paper are summarized as follows:

- We propose novel random walk algorithms called BloomWalk and CuckooWalk, which efficiently manage a small amount of memory capacity using probabilistic data structures.
- Through comparisons with conventional representative random walks on various types of graphs, we reveal that both BloomWalk and CuckooWalk significantly reduce node search time and graph exploration time in rather complex graphs.

The organization of this paper is as follows: In Section II, we provide an overview of random walk algorithms on a graph and probabilistic data structures such as the Bloom filter and Cuckoo filter. In Sections III and IV, we initially outline the problem and then describe our proposed BloomWalk and CuckooWalk in detail. In Section V, we evaluate the performance of BloomWalk and CuckooWalk on five different types of graphs in comparison to other conventional random walk algorithms. Finally, in Section VI, we summarize this paper and discuss future research directions.

II. Random Walk Algorithms and Probabilistic Data Structure

This section provides a comprehensive overview of random walk algorithms utilized in target node search and graph exploration, alongside major probabilistic data structures such as Bloom filter and Cuckoo filter.

A. Random walk algorithms

In the context of graph exploration, random walks are fundamental techniques widely employed in various applications. However, frequent revisits to already visited nodes can hinder efficiency. Traditional random walk algorithms like SRW assign equal probabilities to transitions between neighboring nodes, but they may suffer from inefficiency due to revisitation.

To address this inefficiency, novel algorithms have emerged. NBRW and SARW are notable examples. NBRW [11, 12] prevents revisits to previously traversed nodes, thereby enhancing node search and graph exploration efficiency. SARW [16] takes this further by maintaining a memory of all previously visited nodes and avoiding revisits whenever possible. These methods significantly improve target node search and graph exploration efficiency.

B. Probabilistic data structure

Bloom and colleagues made a seminal contribution to the field of data filters by introducing the Bloom filter, a probabilistic data structure tailored for assessing the membership of an element within a given set [17]. Bloom filter offers a condensed representation of a set of elements and supports two fundamental operations: Insert and Lookup. A Bloom filter consists of k hash functions and an initial bit array where all bits are initialized to zero. During the insertion of an element, the Bloom filter hashes the element to k positions within the bit array using the k hash functions and subsequently sets all k corresponding bits to one. The process of lookup employs a similar methodology, entailing the examination of the corresponding k bits in the array. If all these bits are set, the query returns true; otherwise, it returns false.

The efficacy of a Bloom filter is contingent upon the number of employed hash functions and the size of the bit array. The probability of false positives, wherein a Bloom filter erroneously indicates the presence of an element in the set when it is absent, escalates with the augmentation of the set's cardinality and the diminution of the bit array's size. Mitigation of the false positive is attainable through the enlargement of the bit array's dimensions and the utilization of more hash functions. Nonetheless, this approach entails augmented memory utilization and computational overhead.

The probability of encountering a false positive in a Bloom filter is approximated by [21]:

$$p_{FP} \approx \left(1 - e^{-\frac{k \cdot n'}{M}}\right)^k, \tag{1}$$

where n' denotes the number of elements stored within the Bloom filter and M is the filter size in bits.

Fan *et al.* introduced the Cuckoo filter as an advancement over traditional Bloom filter, boasting superior space efficiency and performance [18]. Cuckoo filter leverages Cuckoo hashing to realize approximate membership checks. It comprises an array of buckets, each composed of (typically 4) cells, wherein each cell can accommodate the fingerprint of an element. During the insertion process, the fingerprint of the element x is computed using a hash function $f(x)$ and stored in lieu of the element x. The Cuckoo filter employs two potential buckets per element. For an element x, it computes the indices of the two candidate buckets as follows [18]:

$$h_1(x) = h(x) \tag{2}$$

$$h_2(x) = h_1(x) \oplus h(f(x)) \tag{3}$$

where $h(x)$ is another hash function.

To ascertain whether an element y is stored in the filter, its fingerprint is first calculated using $f(y)$, and two candidate buckets are determined according to Eqs. (2) and (3). Subsequently, these two buckets are inspected: if any existing

fingerprint in either bucket matches, the filter returns true; otherwise, it returns false.

Bloom and Cuckoo filters find applications across various domains such as networking, database systems, and cloud computing [22-25]. For instance, in networking, Lee *et al.* utilize a Bloom filter to enhance the search performance of Forwarding Information Base (FIB) lookups for packet forwarding in content-centric networks [24]. Additionally, Reviriego *et al.* explore the utilization of Cuckoo filters and Bloom filter for packet classification [25].

III. Problem Definition

A. Problem formulation

Consider a graph $G = (V, E)$, where V represents the finite set of nodes and E denotes the set of links. For a node v, $N(v)$ denotes its neighborhood, defined as the set $\{v|(u, v) \in E\}$. Let the currently visiting node be denoted as v_c, the starting node as v_s, and the target node as v_t. The objective is to minimize the time required to locate the target node v_t (hitting time) and explore all nodes at least once (cover time) on an unknown graph while adhering to the limited memory capacity of an agent. Let M be the memory size of the mobile agent. We assume that every node in G is assigned an unique identifier, and the size of identifier is I.

B. Issues with conventional algorithms

Conventional random walk algorithms such as SARW, NBRW and k-History encounter constraints when the memory size M is small. SARW is highly effective when memory capacity is large, but impractical when M is small. NBRW can be used when M is small (larger than the node identifier size I) but fails to effectively utilize memory resources. Similarly, k-History suffers from reduced performance as it cannot efficiently record a large number of visited nodes within limited memory; i.e., it records at most M/I nodes, which is not sufficient for small M and/or large I.

IV. BloomWalk and CuckooWalk Algorithm

A. Overview

The underlying principle of BloomWalk and CuckooWalk algorithms is to track visited nodes based on the hash of their identifiers rather than storing the identifiers directly although it may lead to false positives. Moreover, BloomWalk and CuckooWalk has the flexibility to incorporate transition probabilities from various random walk algorithms; i.e., they can be seamlessly integrated with other random walk algorithms.

Additionally, BloomWalk and CuckooWalk dynamically control the occupancy ratio of their filters to mitigate false positives. In Bloom filter and Cuckoo filter, the false positive ratio increases with the occupancy ratio of the filter (i.e., the ratio of non-zero bits in the filter). Our BloomWalk and CuckooWalk adopts a mechanism to dynamically control the occupancy ratio by intentionally deleting a fraction of elements from the filter.

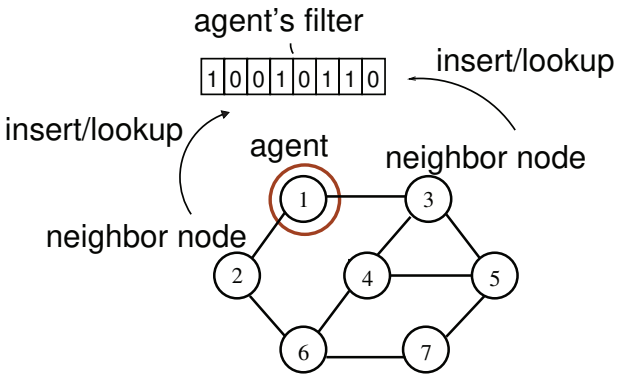


Fig. 1. Operation of BloomWalk mobile agent

B. Example operation

In this section, we detail the operation of the BloomWalk algorithm. As depicted in Fig. 1, BloomWalk begins its exploration from a starting vertex, for instance, node 1. Initially, this node is recorded in a Bloom filter, indicating it as visited. The algorithm then checks the possibility to move to neighbor nodes (e.g., nodes 2 and 3).

For each neighbor node, BloomWalk applies hash functions to generate a specific bit sequence, which is then checked against the Bloom filter. If all bits in the sequence are set to one, the algorithm infers the node has been visited previously. Otherwise, the node is deemed unvisited. The agent then randomly selects its next destination from these unvisited neighbors. For example, if node 2 is determined to be visited while node 3 is not, the mobile agent moves to node 3 and records (inserts) node 3 into the Bloom filter.

Now the mobile agent is on node 3 with nodes 1, 4, and 5 as neighbors. BloomWalk repeats the above process, using hash functions and the Bloom filter to identify which neighbors have been visited. If nodes 1 and 4 are marked as visited and node 5 is not, the agent proceeds to node 5, continuing its exploration by marking the node 5 in the Bloom filter.

This iterative method allows BloomWalk to navigate the graph efficiently, leveraging the Bloom filter to manage memory usage effectively while minimizing the likelihood of revisiting nodes.

C. Algorithms

BloomWalk commences its exploration from a starting node v_s, progressing by choosing an adjacent unvisited vertex for the subsequent move. This algorithm utilizes a Bloom Filter to effectively track which nodes have been visited, offering a memory-efficient solution for indicating their presence. Through looking up the Bloom filter, BloomWalk avoids revisiting vertices that have already been visited, thereby optimizing the efficiency of the node search and the graph exploration processes.

CuckooWalk, akin to BloomWalk, employs a Cuckoo filter to manage visited nodes. In contrast to BloomWalk, which

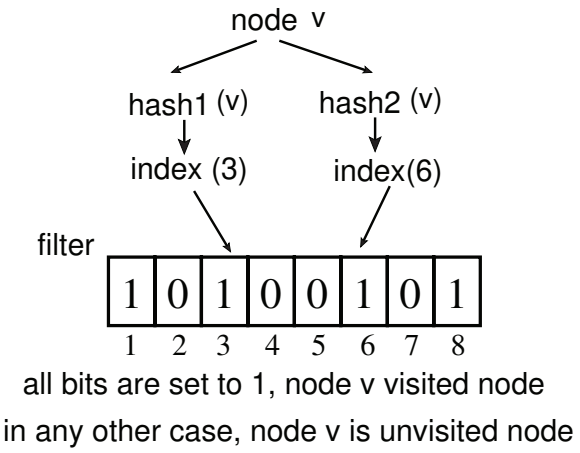


Fig. 2. Bloom filter for detecting visited/unvisited status of node v

Algorithm 1 BloomWalk/CuckooWalk algorithm

input: starting node v_s, filter size M, random walk algorithm A, adjustment threshold θ, reduction factor ψ
initialization: clear filter F, $u \leftarrow v_s$
for each step **do**
  $U \leftarrow \{v|(u,v) \in E, v \notin F\}$ ▷ unvisited node set
  if $U = \phi$ **then** ▷ all neighbor seems visited
    $U \leftarrow \{v|(u,v) \in E\}$
  $v \leftarrow$ randomly chosen node from U with algoirthm A
  $F \leftarrow F \cup \{v\}$ ▷ insert v into F
  if $|F|/M \geq \theta$ **then** ▷ F is too occupied?
    clear $\psi\,|F|/M$ entries in F
  $u \leftarrow v$ ▷ move to neighbor node v

utilizes a single filter with a size of M bits, the Cuckoo filter introduces parameters such as the number of buckets, the number of cells within each filter, and the size of the fingerprint. Initially, agents determine the appropriate configuration of these parameters based on the available memory size M.

The proposed algorithms rely on probabilistic data structures, which exhibit effectiveness but are susceptible to false positives. False positives occur when the filter erroneously indicates the presence of an element that is not actually present. In the context of BloomWalk and CuckooWalk algorithms, false positives can result in the misinterpretation of unvisited vertices as visited, thereby impeding the exploration of certain graph vertices or paths and leading to suboptimal or incomplete exploration.

BloomWalk and CuckooWalk dynamically control the occupancy ratios of their filters to mitigate the impact of false positives. Namely, if the occupancy ratio (i.e., the number of non-zero bits in the filter to the filter size) surpasses a predefined threshold θ (adjustment threshold), a fraction ψ (reduction factor) of elements in the filter is deleted. This proactive measure enhances the filter's resilience against false positives, thereby facilitating more effective node search and graph exploration.

The pseudocode of BloomWalk and CuckooWalk algorithms is shown in Algorithm 1.

The pseudocode describes the BloomWalk/CuckooWalk algorithm, aimed at optimizing graph exploration through intelligent memory management using a probabilistic filter.

- Inputs: The algorithm requires several inputs:
  - A starting node, v_s, from which the exploration begins.
  - The filter size, M, which dictates the capacity of the probabilistic filter.
  - A specific random walk algorithm, A, which guides the selection process of the next node to visit.
  - An adjustment threshold, θ, determining when to clear a portion of the filter to manage its capacity.
  - A reduction factor, ψ, indicating the fraction of the filter to clear once the threshold is exceeded.
- Initialization: Initially, the filter F is cleared to remove any previous data, and the current node u is set to the starting node v_s.
- Exploration Process:

1) At each step of the exploration, the algorithm identifies the set U of unvisited neighboring nodes. These are the neighbors of the current node u not marked in the filter F as visited.
2) If U is empty ϕ, suggesting all neighbors appear to have been visited, the algorithm resets U to include all neighbors of u, regardless of their status in F. This step is necessary to continue exploration, especially in dense graphs where the probabilistic nature of F might inaccurately mark unvisited nodes as visited.
3) Then, a node v is randomly selected from U using the algorithm A. This random selection is crucial for the non-deterministic nature of the walk.
4) The selected node v is added to the filter F, marking it as visited. This operation helps in preventing revisits to the same node.
5) If the occupancy of the filter F reaches or exceeds the threshold θ relative to its maximum capacity M, the algorithm clears a fraction ψ of the filter's entries. This step is essential for maintaining the filter's effectiveness and ensuring the algorithm's continued operation without being hindered by memory limitations.
6) The current node u is updated to v, and the process repeats for the next step. This movement to the neighbor node v progresses the exploration of the graph.

V. Experiments

A. Experimental design

In this section, we conduct simulation experiments aimed at assessing the effectiveness of target node search and graph exploration utilizing BloomWalk and CuckooWalk algorithms. Our analysis focuses on two key metrics: the hitting time and the cover time. We compare the performance of our proposed algorithms with that of traditional random walk algorithms, namely NBRW, SARW, and k-History.

Specifically, we conduct simulation experiments on five types of graphs: Erdos-Renyi (ER), Barabasi-Albert (BA), regular, tree, and ring graphs. These graph models are selected because they are commonly used to simulate real-world networks. We vary both the memory size of the mobile agent and the number of nodes in the graph to explore the impact of these parameters on the performance of BloomWalk and CuckooWalk algorithms.

The initial experiment involved graphs consisting of 100 nodes, with node identifiers size I of 32 bits, and memory sizes M ranging from 32 to 1,000 bits. Note that since NBRW only stores the last visited node, it only utilizes a fixed memory size of 32 bits regardless of the memory size M. Also, an infinite memory size is allocated to SARW.

For each graph type, we conducted 1,000 random walk trials and calculated the averages along with 95% confidence intervals for both the average first arrival time and coverage time. This comprehensive analysis allows us to evaluate the performance of BloomWalk and CuckooWalk under different memory constraints across various graph scenarios.

In BloomWalk, the adjustment threshold θ was set to 0.5, and the reduction factor ψ was set to 1. On the other hand, the occupancy ratio control mechanism is disabled in CuckooWalk (i.e., $\psi = 0$). This is because, to the best of our knowledge, the Bloom filter allows deletion of entries in the filter while the Cuckoo filter does not.

B. Results

Our findings indicate that BloomWalk and CuckooWalk exhibits exceptional performance in scenarios characterized by small to moderate memory capacities. Figure 3 illustrates the average hitting time for mobile agents navigating an unknown graph using BloomWalk, CuckooWalk, and conventional random walk algorithms. The results also encompass the outcomes of graph exploration employing k-History, SARW, and NBRW algorithms.

Especially, BloomWalk significantly reduces the average hitting time in all types of graphs when the memory size M is small. Recall that in our experiments, BloomWalk, CuckooWalk, and k-History are equipped with the same amount of the memory size. On the contrary, NBRW is equipped with just 32 bits and SARW is with the infinite memory. So, we should examine on how the efficiency of node search and graph exploration are improved compared with k-History. Note that results with NBRW and SARW are just for references as baselines.

Looking Fig 3 tells us that the average hitting time of BloomWalk is always significantly smaller than that of k-History in ER, BA, and regular graphs. On the contrary, k-History with small memory size M shows exceptionally better performance in tree and ring graphs. In other words, our BloomWalk works quite effectively in rather complex graphs whereas k-History works well in simple graphs.

We should note that BloomWalk achieves an average hitting time comparable to that of the SARW algorithm (with infinite memory) when the memory size is modestly small (on the order of a few hundred digits) in all graphs.

On the other hand, CuckooWalk, which utilizes a Cuckoo filter, outperforms k-History with a very small memory size. However, in scenarios with moderate memory capacities, CuckooWalk exhibits comparable performance with k-History. This is mostly because, unlike BloomWalk, CuckooWalk does not use the occupancy ratio control mechanism.

Figure 4 illustrates the average cover time for graph exploration conducted by the mobile agent utilizing BloomWalk, CuckooWalk, and conventional random walk algorithms. Consistent with the average hitting time results, a decrease in the average cover time is evident in (not all but) many cases.

Additionally, we explored the impact of graph size on the proposed BloomWalk and CuckooWalk algorithms. This involved changing the node size to 500 and setting the memory size M of mobile agents in the range of 32 bits to 5,000 bits (see Figs. 5 and 6).

Our findings suggest that even when changing the graph size to 500, BloomWalk and CuckooWalk consistently reduce the average hitting time compared to k-History in rather complex graphs. Conversely, in relatively simple graphs like tree and ring graphs, k-History demonstrate better efficiency in exploration than BloomWalk and CuckooWalk.

VI. Conclusion

In this paper, we have proposed BloomWalk and CuckooWalk as innovative random walk algorithms specifically designed for efficient target node search and graph exploration in unknown graphs, employing probabilistic data structures. Through rigorous simulation experiments across a diverse range of graph types, we have showcased the superior efficiency of these algorithms under the constraint of limited memory resources. Our results indicate that BloomWalk and CuckooWalk outperform the conventional algorithm, k-History, in scenarios where the structure of underlying graphs is rather complex.

Looking ahead, our research will delve deeper into the performance of these algorithms in the context of large-scale graphs and explore the integration of additional probabilistic data structures to further enhance their efficiency and applicability.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number 24K02936.

REFERENCES

[1] F. Xia, J. Liu, H. Nie, Y. Fu, *et al.*, "Random Walks: A Review of Algorithms and Applications," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, pp. 95–107, Apr. 2020.

[2] L. László, "Random Walks on Graphs: A Survey," *Combinatorics, Paul Erdos is Eighty*, vol. 2, pp. 1–46, Jan. 1996.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking : Bringing Order to the Web," in *Proceedings of International World Wide Web Conference (WWW 1999)*, pp. 161–172, Nov. 1999.

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," in *Proceedings of the International Conference on World Wide Web (WWW 2017)*, pp. 107–117, Apr. 1998.

[5] M. E. Newman, "A measure of betweenness centrality based on random walks," *Social networks*, vol. 27, pp. 39–54, Sept. 2005.

[6] J. D. Noh and H. Rieger, "Random Walks on Complex Networks," *Physical review letters*, vol. 92, Mar. 2004.

[7] M. Okuda, S. Satoh, Y. Sato, and Y. Kidawara, "Community Detection Using Restrained Random-Walk Similarity," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, pp. 89–103, July 2019.

[8] P. Sarkar, D. Chakrabarti, and M. Jordan, "Nonparametric Link Prediction in Large Scale Dynamic Networks." https://arxiv.org/abs/1109.1077, Nov. 2013.

[9] M. Bonaventura, V. Nicosia, and V. Latora, "Characteristic times of biased random walks on complex networks," *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 89, July 2013.

[10] A. D. Sarma, A. R. Molla, and G. Pandurangan, "Fast Distributed Computation in Dynamic Networks via Random Walks," in *Proceedings of the International Symposium on Distributed Computing (DISC 2012)*, pp. 136–150, Oct. 2012.

[11] C. Bordenave, M. Lelarge, and L. Massoulié, "Non-backtracking Spectrum of Random Graphs: Community Detection and Non-regular Ramanujan Graphs," in *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS 2015)*, pp. 1347–1357, Dec. 2015.

[12] D. Fasino, A. Tonetto, and F. Tudisco, "Hitting times for non-backtracking random walks," *ArXiv*, May 2021.

[13] M. Bonaventura, V. Nicosia, and V. Latora, "Characteristic times of biased random walks on complex networks," *Phys. Rev. E*, vol. 89, Jan. 2014.

[14] A. Fronczak and P. Fronczak, "Biased random walks in complex networks: The role of local navigation rules," *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 80, Aug. 2009.

[15] K. Kitaura, R. Matsuo, and H. Ohsaki, "Random Walk on a Graph with Vicinity Avoidance," in *proceedings of the International Conference on Information Networking (ICOIN 2022)*, pp. 232–237, Jan. 2022.

[16] N. Madras and G. Slade, *The Self-Avoiding Walk*. Springer Science & Business Media, Nov. 2011.

[17] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM (Commun. ACM)*, vol. 13, pp. 422–426, July 1970.

[18] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmache, "Cuckoo Filter: Practically Better Than Bloom," in *Proceedings of the ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT 2014)*, pp. 75–88, Dec. 2014.

[19] S. Sengupta and A. Rana, "Role of Bloom Filter in Analysis of Big Data," pp. 6–9, June 2020.

[20] T. M. Graf and D. Lemire, "Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters," *Journal of Experimental Algorithmics (JEA)*, vol. 25, pp. 1–16, Mar. 2020.

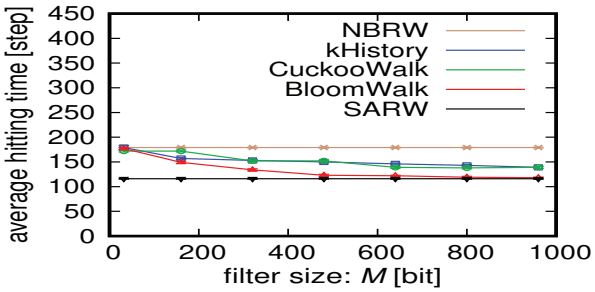
[21] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *IEEE COMMUNICATIONS SURVEYS & TUTORIALS*, vol. 14, pp. 131–155, Mar. 2012.

[22] A. Andrei Broder and M. Mitzenmacher, "Survey: Network Applications of Bloom Filters: A Survey.," *Internet Mathematics*, vol. 1, pp. 485–509, 11 2003.

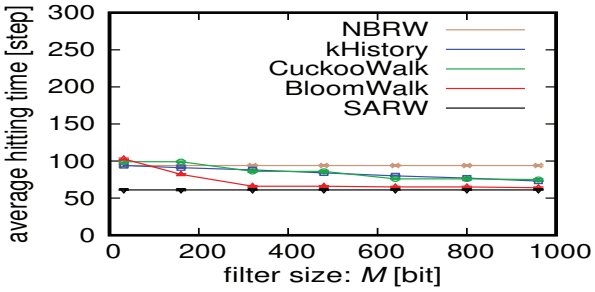
[23] D. Gupta and S. Batra, "A short survey on bloom filter and its variants," in *Proceedings of the International Conference on Computing, Communication and Automation (ICCCA 2017)*, pp. 1086–1092, May 2017.

[24] J. Lee, M. Shim, and H. Lim, "Name Prefix Matching Using Bloom Filter Pre-Searching for Content Centric Network," *Journal of Network and Computer Applications*, vol. 65, pp. 36–47, Apr. 2016.

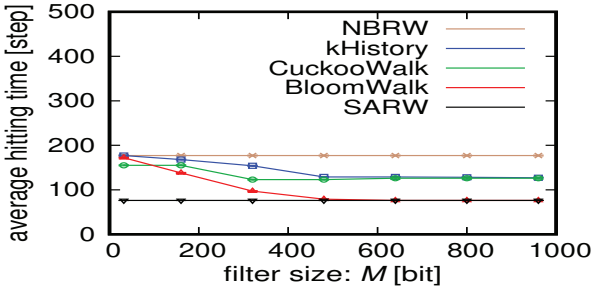
[25] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarell, "Cuckoo Filters and Bloom Filters: Comparison and Application to Packet Classification," *IEEE Transactions on Network and Service Management*, vol. 17, pp. 2690–2701, Dec. 2020.



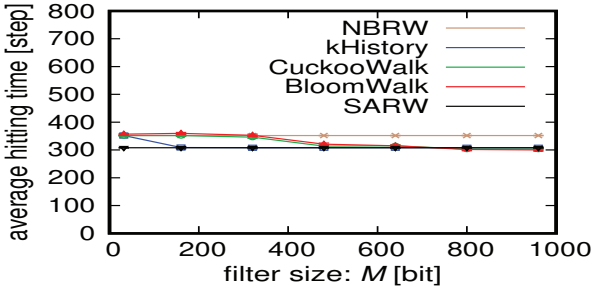
(a) ER graph



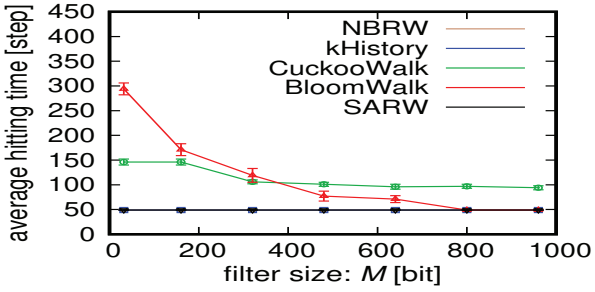
(b) regular graph



(c) BA graph

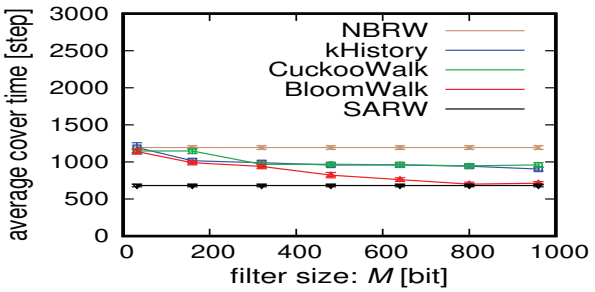


(d) tree graph

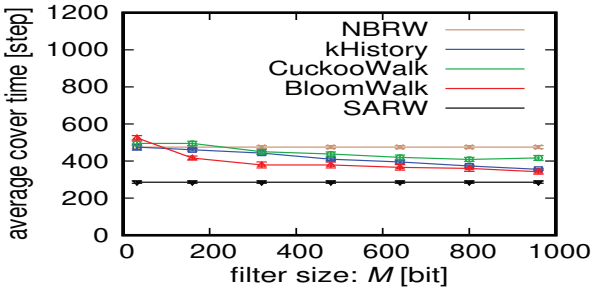


(e) ring graph

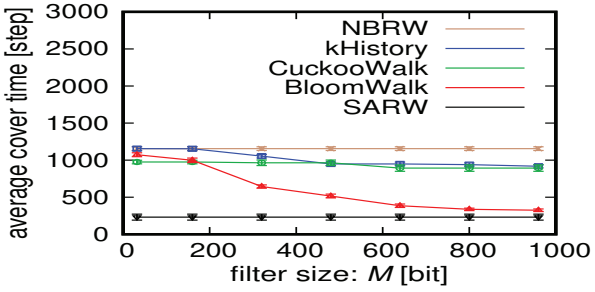
Fig. 3. Hitting time on different graphs with 100 nodes



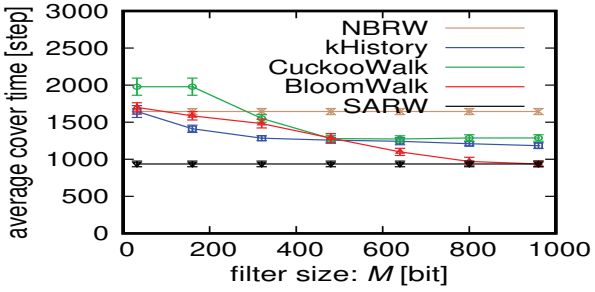
(a) ER graph



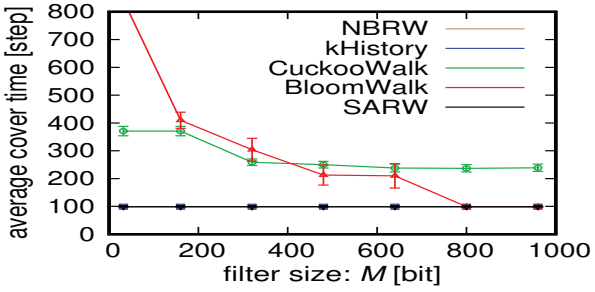
(b) regular graph



(c) BA graph

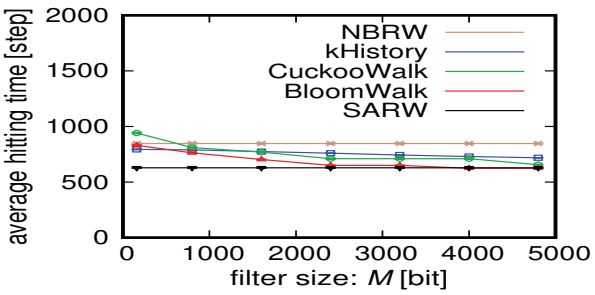


(d) tree graph

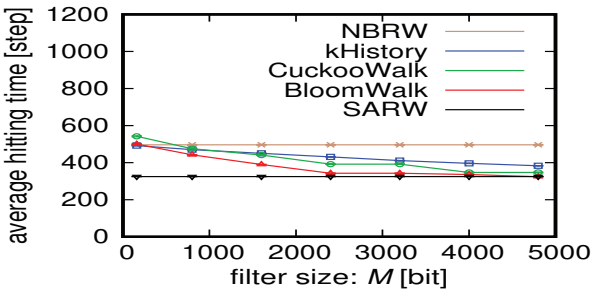


(e) ring graph

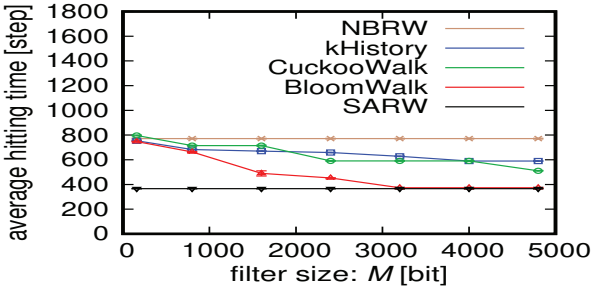
Fig. 4. Cover time on different graphs with 100 nodes



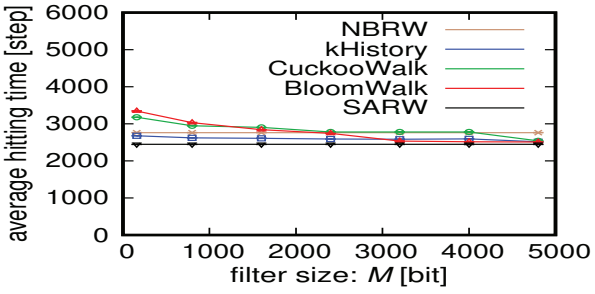
(a) ER graph



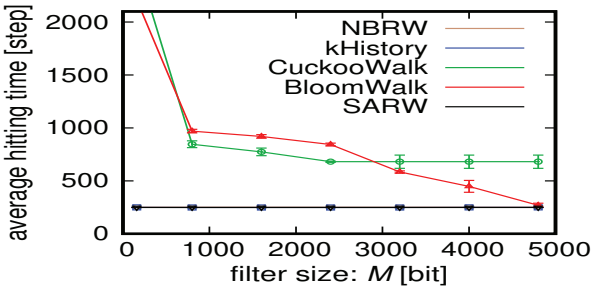
(b) regular graph



(c) BA graph

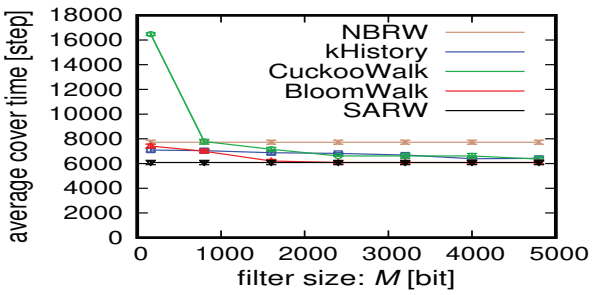


(d) tree graph

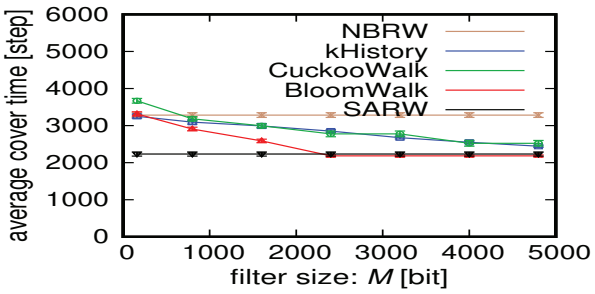


(e) ring graph

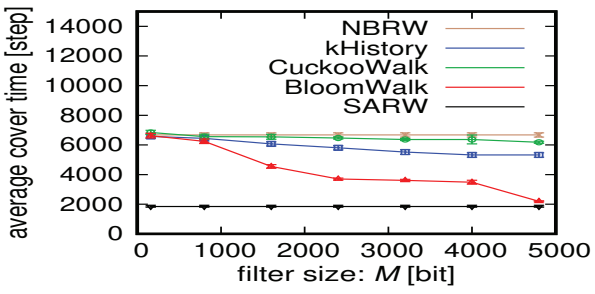
Fig. 5. Hitting time on different graphs with 500 nodes



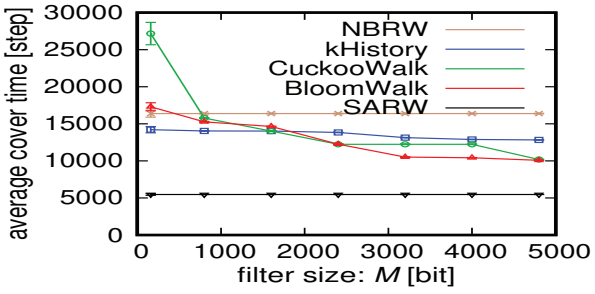
(a) ER graph



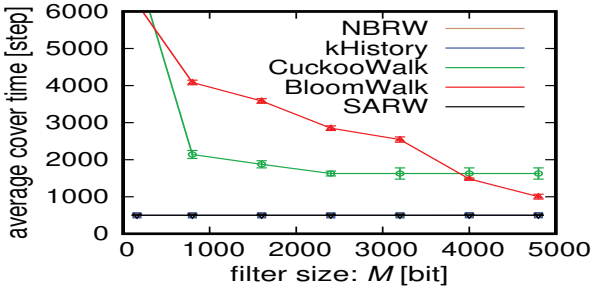
(b) regular graph



(c) BA graph



(d) tree graph



(e) ring graph

Fig. 6. Cover time on different graphs with 500 nodes