Greedy Caching: An Optimized Content Placement Strategy for Information-centric Networks

[1]Bitan Banerjee*, [2]Adita Kulkarni*, [2]Anand Seetharam
[1]Department of Electrical and Computer Engineering, University of Alberta, Canada
[2]Computer Science Department, SUNY Binghamton, USA
bitan@ualberta.ca, akulka17@binghamton.edu, aseethar@binghamton.edu

Abstract

Most content placement strategies in information-centric networks (ICN) primarily focus on pushing popular content to the network edge, fail to effectively utilize the caches in the network core and provide limited performance improvement. In this paper, we propose *Greedy Caching*, a content placement strategy that determines the set of content to be cached at each network node so as to maximize the network hit rate. *Greedy Caching* caches the most popular content at the network edge, recalculates the relative popularity of each piece of content based on the request miss stream from downstream caches and then determines the content to be cached in the network core. We perform exhaustive simulation in the Icarus simulator [1] using realistic Internet topologies (e.g., GARR, GEANT, WIDE, scale-free networks) as well as real-world request stream traces, and demonstrate that *Greedy Caching* provides significant improvement in content download delay (referred to as latency) over state-of-the-art dynamic caching and routing strategies for ICN for a wide range of simulation parameters. Our simulation results suggest an improvement of 5-28% in latency and 15-50% improvement in hit rate over state-of-the-art policies for synthetic traces.

1. Introduction

Content caching at storage-enabled network nodes is one of the most attractive features of information-centric networking (ICN), a new networking paradigm that aims to evolve today's Internet from a host-centric model to a content-centric one. In-network content caching enables requests for content to be served from intermediate caches, in addition to the content custodians (origin servers). Serving a request from an intermediate cache has several benefits such as reduced content download delay, increased throughput and decreased network congestion.

Video delivery companies (e.g., YouTube, Netflix) already use simple forms of popularity based in-network caching in today's Internet to improve user performance. These video delivery applications primarily determine the popularity of multimedia content based on parameters such as release date, viewership of past series of a show and push popular content to the network edge [2]. In recent years, proposed caching policies have also identified that caching popular content within the network is essential to improve performance [3, 4, 5]. However, existing policies focus mainly on the network edge and fail to effectively leverage caches in the network core. Deploying network-wide caches in ICN is likely to be expensive. Therefore, it is important to design efficient caching and routing policies that maximize cache utilization, both at the network edge and in the network core and minimize unnecessary content duplication. While it is tempting to think that determining what content to cache at a node only requires local information, content cached at downstream nodes drastically impacts the request stream seen by upstream caches and may ultimately reduce network-wide cache utilization. We use the words cache and node interchangeably in this paper.

To this end, in this paper, we propose *Greedy Caching*, a content placement strategy that determines the optimized set of content to be cached at each network node based on relative content popularity so as to maximize overall network hit rate. *Greedy Caching* estimates the relative content popularity at each node based on the request stream from directly connected users as well as the request miss stream from downstream nodes and then greedily caches the topmost relatively popular content at that node. The difficulty of the problem stems from the fact that different pairs of network nodes can forward requests to one another, resulting in interdependencies, and cycles in the underlying graph, thereby making it difficult to estimate the relative content popularity. We demonstrate via extensive experiments that *Greedy Caching* not only maximizes network level metrics such as hit rate, but also results in reduced content download delay (referred to as latency).

The main contributions of this paper are given below.

- We assume that the network has an underlying routing policy for forwarding requests for content toward the custodian. We propose *Greedy Caching*, a content placement policy that greedily caches the most popular content at each cache based on their relative content popularity. To estimate relative content popularity, *Greedy Caching* first leverages routes provided by the routing algorithm to cre-

*The first two authors have made equal contribution ate a directed acyclic graph (DAG). For the single custodian case, DAG construction is relatively straightforward. However, for the multiple custodian case, simply combining the routes provided by the routing algorithm results in a cyclical graph, due to node pairs sending traffic to one another. *Greedy Caching* therefore uses the feedback arc set algorithm to prune this cyclical graph and construct a DAG. *Greedy Caching* then combines the request stream from users with the constructed DAG to determine the set of content to be cached at each network node, starting from the network edge and ending at the custodians.

- We perform simulations in Icarus [1], a simulator built exclusively for implementing and testing new ICN routing and caching policies to demonstrate the efficacy of *Greedy Caching*. We compare the performance of *Greedy Caching* with state-of-the-art dynamic caching and routing policies, Least Copy Everywhere (LCE), Leave Copy Down (LCD), Cache Less for More (CL4M), ProbCache, Random Caching (Random), and Hash Routing (HR) on real-world internet topologies (e.g., GARR, GEANT, and WIDE) and scale-free networks. We also evaluate the performance of *Greedy Caching* on real-world request stream traces. We study the impact of various simulation parameters (e.g., cache size, content-universe, content popularity skewness) on the performance of *Greedy Caching* and demonstrate that it provides approximately 5-28% improvement in latency and 15-50% improvement in hit rate over state-of-the-art strategies.

The rest of the paper is organized as follows. We discuss related work in Section 2. We describe the problem and the proposed *Greedy Caching* algorithm in Sections 3 and 4 respectively. Experimental results are then presented in Section 5. We provide an outlook for future work in Section 6 and conclude the paper in Section 7.

2. Related Work

Caching strategies have been proposed in literature in the context of Content Delivery Networks (CDNs), heterogeneous cellular networks as well as ICN. We note that most existing static caching strategies have mainly focused on pushing popular content to the edge and have been designed for specialized settings (e.g., heterogeneous networks consisting of few small cells and base stations) [6, 7, 8]. In this section, we demonstrate how *Greedy Caching* differs from prior work.

As the proposed caching strategy in this paper is greedy in nature, we first compare it with other greedy approaches proposed in literature. The fundamental difference between our work and prior research is that we consider a general network setting and not just a specific scenario and determine the optimized set of content to be cached at all network nodes including those in the network core. In [6, 7], the authors consider the data caching and routing problem in heterogeneous cellular networks. The authors show that the problem is NP-hard and design greedy and approximation algorithms to minimize delay. The benefit of proactively caching content based on file popularity, correlation among users and the network structure can been explored in [4, 9, 10]. Greedy algorithms for replica management in CDNs have also been proposed in literature [11, 12, 13]. These papers develop a greedy algorithm for content replication in a CDN scenario subject to several parameters such as cache size, distance from users, and access cost. However, these papers do not consider the request miss streams from downstream nodes.

In additional to the above-mentioned strategies, a number of other caching policies have also been proposed for ICN. Some of the most widely accepted caching policies are LCD [14], CL4M [15], and ProbCache [16]. All these caching policies aim to reduce cache redundancy by caching content based on parameters such as content popularity, connectivity of nodes. A modified version of LCD with chunk caching and searching (CLS) is proposed in [17], where a piece of content is cached one level downstream or upstream depending on whether a request is a cache hit or a cache miss. Similarly, a modified version of ProbCache, namely ProbCache+ [18] incorporates a new variable called cache weight to enforce fairness between content. PopCache [19] primarily uses content popularity to determine whether to cache a particular content or not. Authors in [20] propose a caching strategy ProbPD, where the dynamic popularity of a content determines its caching probability. This dynamic popularity is calculated by incorporating the distance of a cache from a user, and incoming content request for a certain time interval. In MPC [21], authors dynamically calculate content popularity locally at each cache by maintaining a popularity table. Topology dependent caching strategies have also been proposed in literature. Authors in [22] develop a caching strategy called Progressive Caching Policy (PCP), where content is cached at the node one hop downstream of the serving node and another intermediate node that has number of incoming links greater than a threshold. Wang *et al.* propose CRCache [23], a caching strategy based on correlation between content popularity and network topology information. Hop-based Probabilistic Caching (HPC) [24] probabilistically caches content depending on the distance between the user and the cache.

Badov *et al.* propose a caching strategy to avoid congested links by caching content at the edge of congested link. A cooperative caching strategy, where off-path caching is explored by controlling the routing algorithm is proposed in [25]. Authors in [26] develop a hash function based joint routing and caching strategy, that helps i) caches decide whether or not to cache a particular content and, ii) routers route requests to relevant caches. Similarly, authors in [27] propose CPHR, a collaborative caching strategy which also uses hash functions. Each content is partitioned according to the hash function and these partitions are then assigned to network caches. Hash function based strategies generally require centralized control which results in high overhead. To overcome the shortfalls of centralized control, distributed cache management (DCM) [28] was proposed to improve cache utilization by sharing holistic information about request patterns and cache configuration. Ioannidis *et. al* study the problem of jointly optimizing routing and caching for a general network [29]. They show that the problem is NP-complete and propose approximation algorithms that run in polynomial time. In [30], Ascigil *et al.* develop a content search algorithm that modifies the forwarding information base (FIB) and integrates opportunistic search and coordinated search approaches.

In contrast to existing literature, we propose a content placement strategy in a general network that adopts a locally optimal approach at each node to determine the set of content to be cached at each network node. *Greedy Caching* caches content at each node based on relative content popularity, which is calculated based on the request miss stream from downstream nodes. This approach aims to maximize hit rate at each network node and increases cache utilization by reducing content duplication.

3. Problem Statement

We begin this section by highlighting some of the limitations of edge caching that push popular content to the network edge and do not effectively utilize caches in the network core via a simple example. Let us consider a network of 3 users (U_1, U_2, U_3), 2 caches (R_1, R_2), and one custodian (C_1) as shown in Figure 1(a). We consider that there are only two unique pieces of content A and B, with probability of requesting content A and B being 0.6 and 0.4 respectively. We assume that R_1 and R_2 can cache only one piece of content and all users generate requests at same rate λ.

In this illustrative example, we consider an edge caching scheme that caches content on absolute popularity to demonstrate the deficiencies of primarily focusing on only pushing popular content to the edge. If content is cached based on absolute popularity (which is the case in most static edge caching schemes), then this will result in content A being cached at both R_1 and R_2. The overall network hit rate in this case is 0.6 (we define network hit rate as getting hits for requests anywhere in the network).

At first glance, this appears to be a good idea, but if content A and content B are cached at R_1 and R_2 respectively, then the overall hit rate is 0.8. This content placement can be achieved if one considers the miss request stream from R_1 to R_2. The total incoming rate at R_2 for content A and B will then be 0.6λ and 1.2λ respectively. We refer to this as *relative content popularity*, which is calculated at each node based on the miss request stream from downstream nodes. Additionally, if we assume that each network link has a delay of 1 second, then caching content based on relative content popularity decreases the overall content download delay to 1.47 seconds from 1.67 seconds.

This example thus highlights the importance of not just pushing popular content to the network edge, but efficiently leveraging the network core caches. Having underlined the shortcomings of edge caching, we next describe the network model and then formally present the content placement problem that maximizes network hit rate.

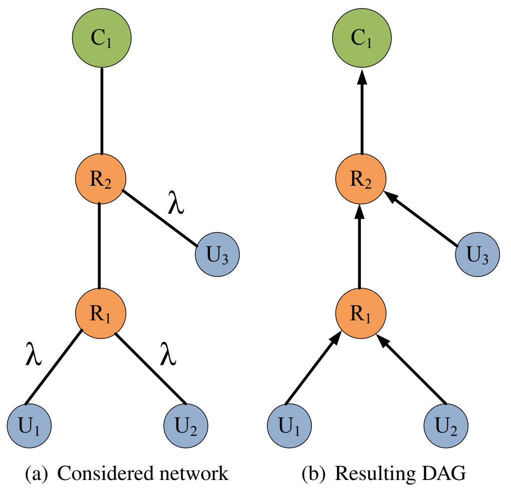


Figure 1: Greedy Caching illustration for single custodian

3.1. Network Model

Let us consider an ICN which is represented by an undirected graph $\mathcal{G}(V, E)$, where V consists of all the nodes in the network including the users, caches and custodians and E consists of the set of interconnected links. We note that custodians are similar to origin servers and permanently house all pieces of content. Let $\mathbb{U} = \{U_1, U_2, ...U_N\}$, $\mathbb{R} = \{R_1, R_2, ...R_M\}$ and $\mathbb{C} = \{C_1, C_2, ...C_L\}$ denote the set of users, caches and custodians respectively. Therefore, the network comprises of N users, M caches and L custodians. We assume that each cache has the same amount of finite storage C. We assume that content universe $F = \{f_1, f_2, ..., f_K\}$ is uniformly distributed among the custodians. Each piece of content is available at only one custodian and is permanently stored there. Let F_j be the set of content stored at the j^{th} custodian. We assume that content popularity varies according to user and follows a certain probability distribution (e.g., Zipfian distribution). User U_i generates requests for content at rate $\Lambda_i = \{\lambda_{i1}, \lambda_{i2}, ...\lambda_{iK}\}$. We abuse notation and let Λ_i also denote the outgoing request rate at any intermediate network node i (apart from the users). The incoming request rate at node i is denoted by Λ'_i. Note that Λ_i and Λ'_i can differ due to caching at node i. We focus primarily on an intra-domain environment, but abstract away details and assume that requests are forwarded toward the custodian based on the underlying routing strategy (e.g., Dijkstra's shortest path routing).

Let $\mathcal{P}_{ij}(V_{ij}, E_{ij})$ denote the shortest path from U_i to the C_j with V_{ij} denoting the set of nodes on that path and E_{ij} denoting the set of directed edges tracing the path from U_i to the C_j. Additionally, for all edges $e_{ij} \in E_{ij}$ connecting nodes i and j, an indicator variable, I^k_{ij} is set to 1 if e_{ij} lies on the shortest path for content k, otherwise it is set to 0. We assume that the shortest path algorithm returns $\mathcal{P}_{ij}(V_{ij}, E_{ij})$ and also sets I^k_{ij}. Note that each request can traverses multiple caches en route to the custodian. If a cache en route to the custodian has the requested content, the cache serves it, otherwise the content is served by the custodian.

3.2. Content Placement Problem

Let x_{km} be a binary variable that denotes if the k^{th} piece of content is cached at the m^{th} node (including users, caches and the custodian). It takes a value 1 if the content is cached and 0 otherwise. For users x_{km} will always take values 0, while for a custodian x_{km} will take a value 1 for content that is housed at the custodian and 0 otherwise. Let $\mathcal{H}$ denote the hit rate for all user requests. A request is said to be a hit if it is served by any in-network cache apart from the custodian. For ease of representation, let us assume that for each path $\mathcal{P}_{ij}(V_{ij}, E_{ij})$ between user U_i and custodian C_j, $(U_i, n_1, \cdots, C_j)$ comprises of the topological ordering of all vertices in V_{ij}. A content request for f_k will be served by node n_l in path $\mathcal{P}_{ij}$ only if the request is not served by any of the preceding nodes n_i. Formally, we denote "$n_l \succ n_i$" to represent that n_l comes after n_i in the ordered list $\mathcal{P}_{ij}$. Using the above notation, the hit rate can be expressed by equation 1. The equation takes into account the traffic for different content from the N users and the fact that if the requested content is cached on any node between the user and custodian, then it results in a hit.

$$\mathcal{H} = \frac{1}{\sum_{i=1}^{N}\sum_{k=1}^{|F|}\lambda_{ik}} \sum_{i=1}^{N}\sum_{j=1}^{L}\sum_{k=1}^{|F_j|} \sum_{\substack{m\in\mathcal{P}_{ij}-C_j:\\ C_j\succ m\succ U_i}} \prod_{\substack{l\in\mathcal{P}_{ij}:\\ m\succ l\succeq U_i}} \lambda_{ik}x_{km}(1-x_{kl}) \tag{1}$$

We can now express the problem of maximizing the hit rate as an optimization problem:

$$\begin{aligned} \max \quad & \mathcal{H} \\ \text{s.t} \quad & \sum_{k=1}^{|F|} x_{km} \leq C \quad \forall m \\ & x_{km} \in \{0,1\} \end{aligned} \tag{2}$$

Our goal is to present a solution to the optimization problem presented above. As the objective function is non-linear, a solution to the optimization problem cannot be easily obtained using a solver. Therefore, in the next section, we present *Greedy Caching*, an optimized content placement policy for ICN that aims to maximize the hit rate. *Greedy Caching* adopts a greedy approach that maximizes the hit rate at each network node and eliminates unnecessary content duplication. The algorithm estimates the relative content popularity at each node based on the miss request stream from downstream nodes. *Greedy Caching* then employs a simple greedy algorithm that caches the most popular content at each node based on the relative content popularity at that node. We note the optimization problem in (2) could also be formulated to minimize latency. But, as link delays are hard to estimate and vary over time in the real-world, we have chosen to maximize hit rate. We demonstrate in Section 5 that *Greedy Caching* outperforms state-of-the-art caching policies both in terms of network metrics such as hit rate and link-load as well as user-facing metrics such as latency.

4. Proposed Solution: *Greedy Caching*

In this section, we leverage the concept of relative content popularity describe in Section 3 to design the *Greedy Caching* algorithm. Estimating the relative content popularity at network nodes lies at the heart of *Greedy Caching*. At the highest level, the *Greedy Caching* algorithm starts by caching the most popular content at the network edge and then iteratively determines the content to be cached at the nodes in the network core by estimating the relative popularity. This iterative process stops when all network nodes have been visited. We first discuss *Greedy Caching* for the relatively simple scenario of an ICN with a single custodian and then move on to the more challenging multiple custodian case. Estimating the relative content popularity, especially for the multiple custodian case is non-trivial because of the interdependencies arising from pairs of network nodes forwarding requests to one another.

4.1. Single Custodian

To determine the relative content popularity with respect to a cache, *Greedy Caching* first combines the routes provided by the underlying shortest path routing algorithm for all users to generate a directed acyclic graph (DAG) $\Psi(V', E')$, where V' and E' are the number of vertices and edges in the DAG respectively. As there is only a single custodian in the network, it is easy to observe that combining these paths will result in a DAG. This is because if a node (say R_1) forwards requests through a node (say R_2) toward the custodian, R_2 lies on the shortest path from R_1 to the custodian. Therefore, R_2 cannot route the requests it receives through R_1. For each node i in Ψ, let $\mathbb{N}'_i$ denote the set of neighbors from which there is an incoming edge to i.

Greedy Caching then performs a topological sort on $\Psi(V', E')$ to determine Θ, an ordering of the vertices in Ψ. Let Θ_i denote the i^{th} vertex in Θ. For a DAG, topological sort provides a linear ordering of the vertices such that for every directed edge from vertex u to vertex v, u comes before v in the ordering. It is evident that the the users and the custodian will be first and last nodes in this topological ordering. *Greedy Caching* then visits the nodes in order.

Greedy Caching then caches the set of content with the highest incoming request rate (i.e., the content with the highest relative popularity). Note that nodes at the network edge will only have incoming edges from the users and thus will be the first group of nodes visited by the algorithm. Therefore, the *Greedy Caching* algorithm will cache the C most popular content at each edge node. Now, these nodes at the network edge will only forward requests for uncached content along their outgoing edges as determined by the routing algorithm. As a result, any node v, which appears in the topological ordering after the edge nodes will take into account the request stream from directly connected users and the request miss stream from nodes that appear earlier than it in the ordering to calculate the relative popularity. Node v will thus cache the C most popular content based on the relative content popularity. Details of the *Greedy Caching* algorithm for a single custodian are provided in Algorithm 1.

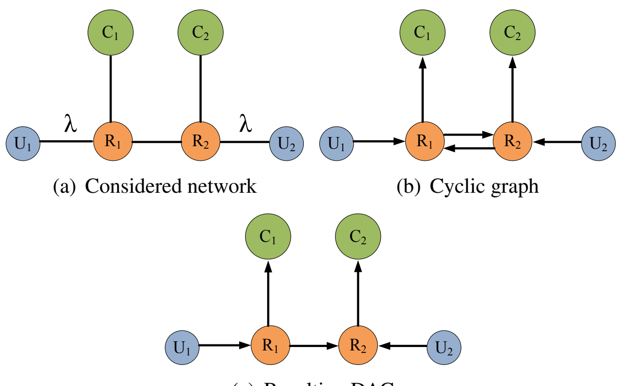
Let us now revisit Figure 1(a) and see how *Greedy Caching* ends up caching content A at R_1 and content B at R_2. For this network, the DAG obtained by combining the shortest paths will be similar to the network itself and is given in Figure 1(b). The topological sort is given by $U_1, U_2, U_3, R_1, R_2, C_1$. Therefore, the algorithm visits R_1 first and caches A. Accounting for the miss stream from R_1 to R_2 and the request stream from U_3, it is easy to see that *Greedy Caching* will cache content B at R_2.

Algorithm 1 Greedy caching for single custodian

Input: Graph $\mathcal{G}(V, E)$, request rate Λ_i for all users
Output: Content placement at network nodes

Step 1: Determine shortest path from users to custodian.
Step 2: Combine shortest paths to generate graph $\Psi(V', E')$.
Step 3: Topologically sort $\Psi(V', E')$.
Step 4: Traverse nodes in topological order. At each node calculate relative popularity for each piece of content and then determine the top C pieces of content to cache.

```
1: for $U_i \in \mathbb{U}$ do
2:     $\mathcal{P}_{i1}(V_{i1}, E_{i1})$ = ShortestPath($U_i, \mathbb{C}_1$)
3: $\Psi(V', E') = \bigcup_{i=1}^{N} \mathcal{P}_{i1}(V_{i1}, E_{i1})$
4: $\Theta$ = TopologicalSort $\Psi(V', E')$
5: procedure Greedy Caching($\Theta$, $\mathbb{R}$)
6:     for $i = 1, i \le |V'|, i++$ do
7:         $r = \Theta_i$
8:         if $r \in \mathbb{R}$ then
9:             for each content $k$ do
10:                $\lambda'_{rk} = \sum_{j \in \mathbb{N}'_r} I^k_{jr} \lambda_{jk}$
11:            $\Lambda'_{r_{sort}}$: Sort $\Lambda'_r$ in descending order
12:            Cache top $C$ content in $\Lambda'_{r_{sort}}$
13:            for each content $k$ cached at $r$ do
14:                $\lambda_{rk} = 0$
```



(a) Considered network (b) Cyclic graph

(c) Resulting DAG

Figure 2: Greedy Caching illustration for multiple custodian

4.2. Multiple Custodian

The multiple custodian scenario is more challenging, primarily due to the fact that simply combining the shortest paths from the users in the network may result in a cyclic graph ($\mathcal{G}'$), as shown in Algorithm 2. Let us consider Figure 2(a) to understand this. In this network, there are two users U_1 and U_2, two caches R_1 and R_2 and two custodians C_1 and C_2. Let us assume that one half of the content universe F_1 is available at C_1 and the other half F_2 is available at C_2. As is evident from the figure, R_1 will forward requests for content in F_2 from U_1 to R_2 and R_2 will forward requests for content in F_1 from U_2 to R_1. Combining the shortest paths will result in a cyclic graph as shown in Figure 2(b).

Greedy Caching attempts to eliminate this problem by leveraging the feedback arc set algorithm [31, 32] that provides the set of edges to be removed from the cyclic graph $\mathcal{G}'$ to create a DAG $\Psi(V', E')$. Therefore, applying the feedback arc set algorithm to Figure 2(b) will result in Figure 2(c). Figure 2(c) demonstrates DAG construction for the multiple custodian network given by Figure 2(a). Once the DAG is constructed, the relative content popularity at each node is determined in a manner similar to Algorithm 1.

Algorithm 2 Greedy caching for multiple custodian

Input: Graph $\mathcal{G}(V, E)$, request rate Λ_i for all users
Output: Content placement at network nodes

Step 1: Determine shortest path from users to custodians.
Step 2: Combine shortest paths to generate graph $\Psi(V', E')$.
Step 3: Apply feedback arc set algorithm to prune the graph and generate a DAG.
Step 4: Topologically sort $\Psi(V', E')$.
Step 5: Traverse nodes in topological order. At each node calculate relative popularity for each piece of content and then determine the top C pieces of content to cache.

```
1: Input network $\mathcal{G}(V, E)$
2: for $U_i \in \mathbb{U}$ do
3:     for $C_j \in \mathbb{C}$ do
4:         $\mathcal{P}_{ij}(V_{ij}, E_{ij})$ = ShortestPath($U_i, \mathbb{C}_j$)
5: $\mathcal{G}' = \bigcup_{i=1}^{N} \bigcup_{j=1}^{L} \mathcal{P}_{ij}(V_{ij}, E_{ij})$
6: Apply feedback arc set over $\mathcal{G}'$ to generate $\Psi(V', E')$
7: $\Theta$ = TopologicalSort $\Psi(V', E')$
8: procedure Greedy Caching($\Theta$, $\mathbb{R}$)
```

The *Greedy Caching* algorithm presented here is a centralized one. Therefore, we assume that the computation is performed in a central location and the results of the computation are dispersed to the different caches. Distributing ‘what to cache’ information to network nodes will incur overhead similar to any routing algorithm used in the network. We note that the proposed algorithm can be easily implemented in a distributed manner for the single custodian case. For the single custodian case, as node pairs do not forward packets to one another, each node can observe the stream of requests coming from downstream nodes, calculate the relative content popularity and cache the most popular based on it. It is evident that this distributed algorithm will converge to the same solution as provided by the centralized algorithm over time. However, in the multiple custodian case, where cyclical dependencies exist in the underlying graph, a more formal approach is needed to design a distributed algorithm. We plan to explore techniques for designing a distributed solution for this problem as part of our future work.

4.3. Discussion on Complexity

In this subsection, we provide a description of the complexity of *Greedy Caching*. The shortest path construction is similar to any routing algorithm and is determined before *Greedy Caching* is executed (in both Algorithms 1 and 2). The time complexity of the shortest path construction is determined and bounded by the approach adopted for computing the shortest path—running Dijkstra's algorithm multiple times, Floyd-Warshall's algorithm or any other routing algorithm that can effectively determine the shortest path between a given set of users and custodians. Union of the paths to make a graph is a simple and deterministic process. A naive (not optimized) approach is to represent each path using an adjacency matrix and adding (specifically, performing an OR operation) all the matrices.

For the multiple custodian case, the feedback arc set algorithm is used to prune the graph. Though determining the optimal feedback arc set is a computationally hard problem, efficient approximation algorithms are available in literature [32] to determine the DAG.

Greedy Caching then parses nodes in the determined DAG in topological order and at each node looks at its neighbors and then determines the topmost relatively popular content to cache at that node. Therefore, at each node the algorithm needs to compute the relative popularity of each piece of content which at first glance might appear to be computationally expensive. However, from an implementation perspective, the algorithm needs to consider the top fraction of the content universe and ignore the extremely unpopular content, as there is limited benefit of caching these pieces of content anywhere in the network.

We note that *Greedy Caching* may not always provide the optimal solution for both single and multiple custodian scenarios. However, as we will see in the following section that *Greedy Caching* performs well in practice and outperforms state-of-the-art policies. The primary reason for its superior performance, especially for user facing metrics such as latency is that it tries to maximize hit rate by pushing content closer to the users and thus lowers overall latency.

5. Performance Evaluation

In this section, we first describe the experimental setup and then present simulation results. We compare the performance of *Greedy Caching* with state-of-the-art dynamic caching and routing policies, namely Leave Copy Everywhere (LCE), Leave Copy Down (LCD), Cache Less for More (CL4M), ProbCache, Random Caching (Random), and basic Hash Routing (HR). These strategies are explained below.

- **LCE:** In this approach, content is cached at every node along the path as it is being downloaded [33].

- **LCD:** In this policy whenever there is a cache hit, the content is replicated at the cache which is one hop downstream toward the requester [14].

- **CL4M:** This policy leverages the concept of betweenness centrality (i.e., the number of shortest paths traversing a cache) to make caching decisions [15]. This policy caches content at nodes with the greatest betweenness centrality, so as to maximize the probability of a cache hit.

- **ProbCache:** This policy reduces cache content redundancy [16] by probabilistically caching content at en route caches.

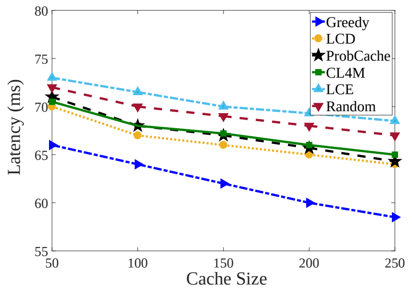
- **Random:** In this caching strategy [34], content is cached at any one of the downstream node. So probability of caching content at a downstream node is inversely proportion to the path length.

- **Hash Routing:** In this approach [26], nodes at the network edge compute a hash function to map the content identifier to a specific cache upon receiving a content request. These edge nodes then forward the request to that particular cache.

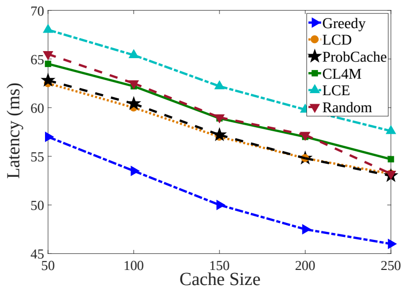
We assume that the network uses Dijkstra's weighted shortest path routing to route content requests to custodians, where weights correspond to link delays. If content is found at a cache en route to a custodian, then it is served from that cache. We perform simulations on a discrete event based simulator Icarus [1], a simulator designed exclusively for ICN research. The simulator consists of four building blocks, scenario generation, experiment orchestration, experiment execution, and result collection. Each request is considered as an event, and whenever an event occurs, a corresponding timestamp is stored. The result collection block gathers the results of the simulation. In Icarus, latency is calculated as the sum of delays on each link traversed during content download.

We perform extensive experiments on various real world networks namely GARR (Italian computer network), GEANT (European academic network), and WIDE (Japanese academic network). GARR is the Italian national computer network for universities with 61 nodes and 89 edges. GEANT is an academic network spread around the world consisting of 40 nodes and 61 edges. The WIDE topology is the first network established in Japan and consists of 30 nodes and 33 edges. To test the scalability of *Greedy Caching*, we also perform experiments on scale-free networks. To avoid cluttering the paper with figures of similar nature, unless mentioned otherwise, all results shown in this paper are for the GARR topology.

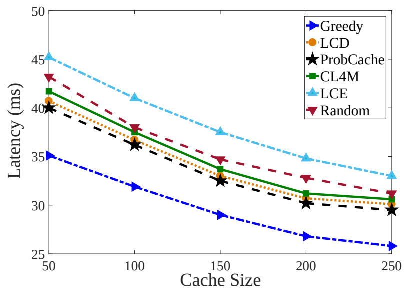
In our simulations, prior to evaluation, all caches are warmed up. Except for large content universe, caches are always warmed up with 100000 requests and the subsequent 100000 requests are used for performance evaluation. Error bars in the figures are obtained over 5 runs of the experiment. We assume



(a) α = 0.6

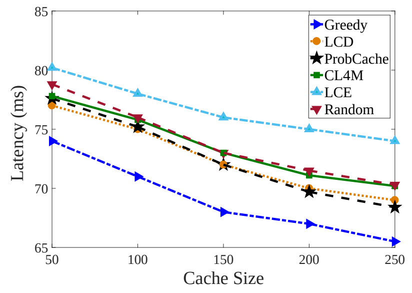


(b) α = 0.8

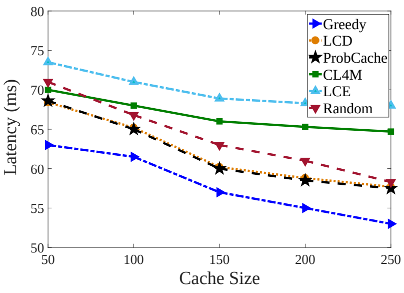


(c) α = 1.1

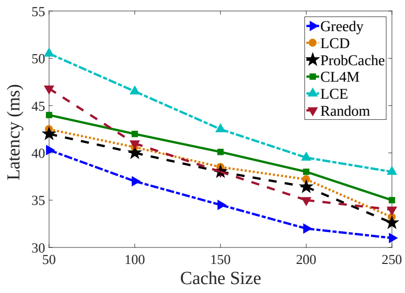
Figure 3: Latency for GARR with single custodian



(a) α = 0.6

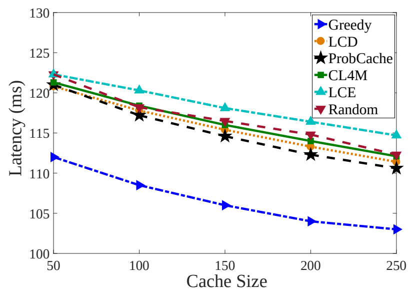


(b) α = 0.8

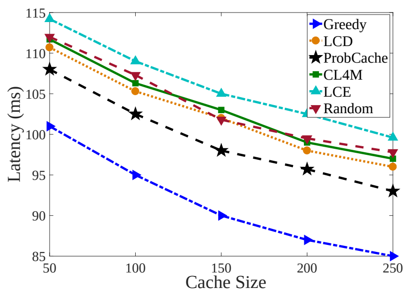


(c) α = 1.1

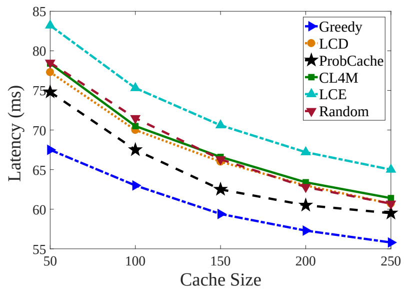
Figure 4: Latency for GARR with 2 custodians



(a) α = 0.6

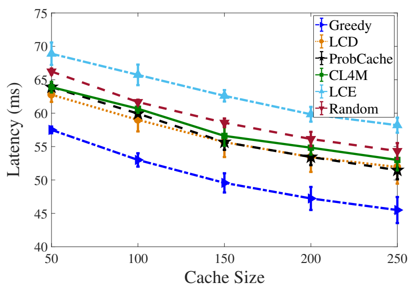


(b) α = 0.8

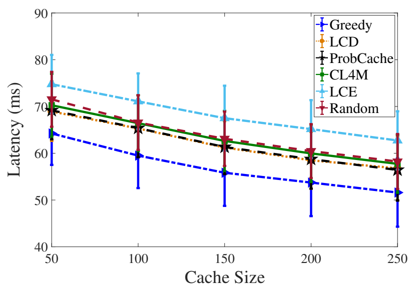


(c) α = 1.1

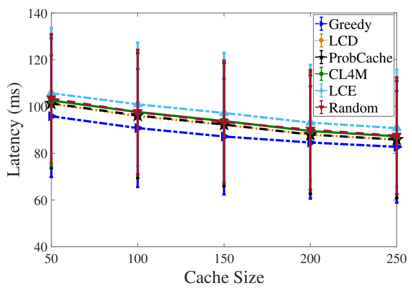
Figure 5: Latency for GARR with 5 custodians



(a) Single custodian



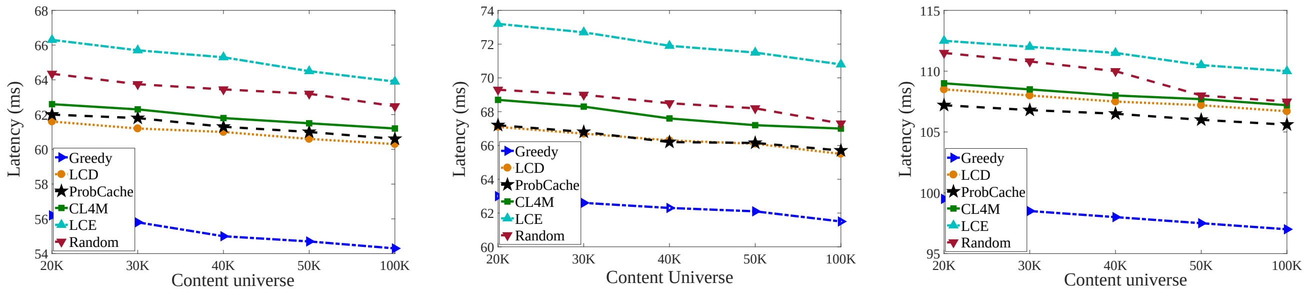
(b) Custodian = 2



(c) Custodian = 5

Figure 6: Latency for GARR topology for random custodian selection

that content popularity follows a Zipfian distribution with skewness parameter α. Nodes with the highest degree (i.e., number of connected links) are considered as custodians, and in case multiple nodes have the same degree, custodians are selected



(a) Single custodian (b) Custodian = 2 (c) Custodian = 5

Figure 7: Latency for GARR topology for varying content universe

randomly among them. For the multiple custodian case, content is uniformly distributed between the custodians and each content is permanently stored at only one of the custodians. Nodes with degree of 1 are selected as users. Unless otherwise mentioned, all results are generated for the following simulation configuration: content universe $F = 10000$, cache size of each node is varied between $50 - 250$, and content popularity skewness (α) is varied between $0.6 - 1.1$. By default, we assume that all dynamic caching policies use Least Recently Used (LRU) as the cache eviction policy.

We consider various performance metrics such as latency, hit rate, and average hop count to demonstrate the superiority of *Greedy Caching*. In our results, performance improvement of using strategy A over strategy B is calculated by taking the percentage of the difference between the two strategies divided by the performance of strategy B. Our experiments demonstrate that *Greedy Caching* outperforms state-of-the-art strategies for a wide range of simulation parameters. We note that we omit results for Hash Routing (HR) from the figures due to aesthetic reasons, as in most scenarios its performance is poorer in comparison to the other strategies.

5.1. Latency Performance for GARR

In this section, we discuss the latency performance of *Greedy Caching* for single custodian and multiple custodian cases for GARR topology with variety of different settings.

5.1.1. Impact of varying custodians

Figure 3 demonstrates the latency results for the different policies for GARR for the single custodian case while figures 4 and 5 show the latency results for the 2-custodian and 5-custodian scenarios for different α values $(0.6, 0.8, 1.1)$. We observe from the figures that *Greedy Caching* outperforms state-of-the-art strategies by 7 - 22% for a wide range of cache sizes. The primary reason behind the superior performance of *Greedy Caching* is better cache utilization, especially in the network core. An interesting point to note is that the overall latency increases for all strategies for the multiple custodian scenario. We observe from our simulations that for the multiple custodian case, the custodians are further away from the users on average when compared to the single custodian case, with some shortest paths from users to a custodian traversing through another custodian. Internally in the simulator, links connected directly to a custodian have higher delay in comparison to the other links, thereby increasing the overall latency. We also demonstrate in Section 5.4, that the overall hit rate decreases for the multiple custodian case.

5.1.2. Impact of custodian placement

An important aspect dictating performance is custodian placement. Custodian placement determines the path traversed by user requests and thus directly influences the content cached within the network and the total cache utilization. Therefore, in this subsection, we study the impact of custodian placement on the performance of *Greedy Caching* to demonstrate the widespread applicability of our results.

We evaluate the performance of *Greedy Caching* for a random custodian placement policy. We implement a random custodian placement policy in our simulator by randomly selecting the custodians from the set of nodes with degree greater than and equal to two. We present performance results for a random custodian placement policy in Figure 6 for α value of 0.8. Different sets of custodians are selected using different seed values and the error bars in Figures 6(a) - 6(c) are obtained for 5 different seed values. From the figures we observe that *Greedy Caching* outperforms other strategies by 7 - 25%. Another interesting observation from the figures is the increased variation in latency performance for the multiple custodian case. The primary reason behind this variation is the change in the average distance of the users from the custodians for different custodian placements.

5.1.3. Impact of content universe

We study the impact of varying content universe on the performance of *Greedy Caching* for GARR (Figure 7). Results are generated for fixed content-to-cache ratio of 0.005, i.e., size of each cache is 0.5% of the content universe. Content universe is varied from 20000 to 100000. Our results suggest that *Greedy Caching* outperforms other strategies for large content universe as well. We observe from Figure 7 that latency decreases as content universe increases. As the absolute cache size increases with content universe (the content-to-cache ratio is fixed), popular content is readily available in caches leading to increased cache utilization. As the primary purpose of Figure 7 is to demonstrate the scalability of our algorithm with respect to content universe, the horizontal axis in Figure 7 is not plotted in linear scale to preserve the aesthetics of the figure. These results demonstrate that the *Greedy Caching* algorithm is efficient and can work with large catalogue sizes.

5.2. Impact of cache eviction policies

In this subsection, we study the performance of *Greedy Caching* with other dynamic caching policies that use FIFO and random cache eviction policies in Figures 8 and 9 respectively. Similar to the LRU cache eviction policy, we observe that *Greedy Caching* outperforms other caching strategies by 8 - 23%. An interesting observation is the superior performance of the Random cache insertion policy over ProbCache and CL4M. The possible reason behind the superior performance of the Random cache insertion policy is that both FIFO and random cache eviction policies do not take content popularity into account while evicting content, and thus ProbCache and CL4M that make caching decisions aimed at reducing content duplication exhibit poor performance.

5.3. Performance for GEANT and WIDE topologies

Results for other topologies, e.g., GEANT and WIDE are shown in Figures 10 and 11 respectively for $\alpha = 0.8$. From the results we conclude that *Greedy Caching* performs best among the existing strategies. For both GEANT and WIDE, we make observations similar to the GARR topology. In general, we observe a) an overall increase in latency with increasing number of custodians and b) decrease in latency for greater cache size and higher values of content popularity skewness.

From our simulations, we observe that the performance improvement for *Greedy Caching* varies between 7-22%, 10-28% and 5-20% for GARR, GEANT and WIDE respectively. We notice that *Greedy Caching* provides better performance for GARR and GEANT compared to WIDE. Primary reason behind this improved performance is that for WIDE, the edge to node ratio (1.1) is significantly lower in comparison to GEANT (1.46) and GARR (1.52). It means that multiple paths are not available to reach a custodian from a randomly selected user in the multiple custodian scenario. We observe from the simulations that in case of WIDE with multiple custodians, most paths to a custodian are via another custodian, as alternate paths are not available. This scenario reduces cache hit significantly and decreases the performance gains.

5.4. Discussion on cache hit rate

In addition to latency, hit rate is an important performance measure in ICN. If a request is served by a network cache, it is referred to as a hit and if it served by the custodian it is referred to as a miss. Therefore, higher hit rate suggests that more content requests are served from network caches, thereby reducing load on the custodians. Hit rate can therefore be considered as a measure for both traffic offloading and cache utilization. Figures 12 and 13 show the hit rate for two different cache sizes 50 and 150 respectively for fixed $\alpha = 0.8$, for single and multiple custodian scenarios. Each sub-figure in Figures 12 and 13 comprises of three topologies (GARR, GEANT, and WIDE). From the figures we observe that *Greedy Caching* improves hit rate by $15 - 50\%$. This can primarily be attributed to the intelligent content placement strategy adopted by *Greedy Caching* where every node caches content based on the total number of requests it receives from other downstream nodes.

We also observe that hit rate performance of *Greedy Caching* degrades with increasing number of custodians. This performance degradation can be attributed to the removal of edges by the feedback arc set algorithm to break cycles for DAG construction. Moreover, we observe that hit rate increases with cache size and content popularity skewness. Greater cache size provides an increased opportunity to cache popular content, while higher value of the skewness parameter results in popular content being requested most of the time.

5.5. Discussion on average hop count

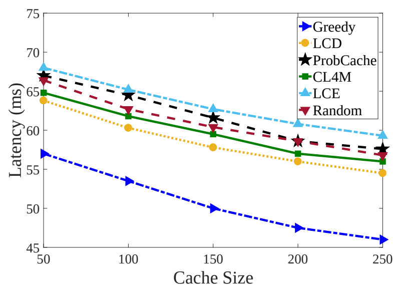
Although hit rate provides an indication of the percentage of requests served from within the network, average hop count provides a measure of how far a request needs to travel to be satisfied. Figures 14 and 15 show the average hop count results for single and multiple custodian cases. As expected, we observe that *Greedy Caching* has the least average hop count. As *Greedy Caching* focuses on the miss stream from downstream nodes to calculate the relative popularity of content at each cache, it makes superior caching decisions at both the network edge and the network core, thereby leading to better overall performance.

5.6. Discussion on average link-load

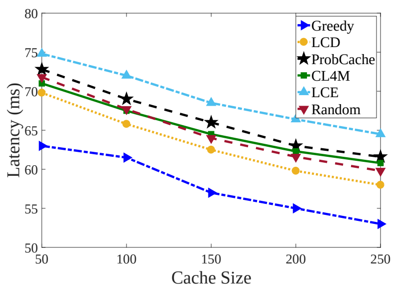
To investigate the impact of the different policies on network congestion, we study a metric called link-load. We calculate link-load of a link as a ratio of the total number of requests traversing the link to the total number of requests. Average link-load for the GARR topology with cache size of 50 and 250, and $\alpha = 0.8$ is tabulated in Table 1. Each entry in the table consists of mean and standard deviation of link load.

We note that the average link-load depends on two parameters - the total number of links carrying traffic in the network and the average hop count. For example, as the number of custodians increases, the total number of links carrying traffic between the users and the custodians increases as well. As the total traffic is distributed over a larger number of links in this case, average link-load is less for a larger number of custodians. We observe this as a general trend for all caching strategies.

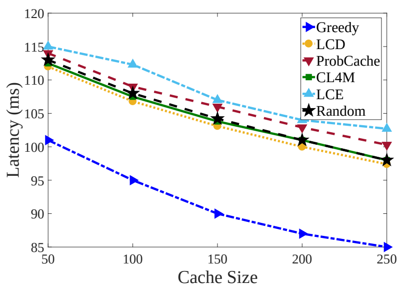
We also observe a direct correlation between the average hop count and the average link-load. A smaller average hop count for a caching policy implies that a greater number of requests are served by caches located closer to the users and hence the number of requests traversing the network decreases. Note that as all caching strategies studied in the paper follow shortest path routing, the total number of network links carrying traffic remains the same for all the caching strategies. Therefore, as the average hop count for *Greedy Caching* is smaller in comparison to the other strategies, its average link-load is smaller as well.



(a) Single custodian

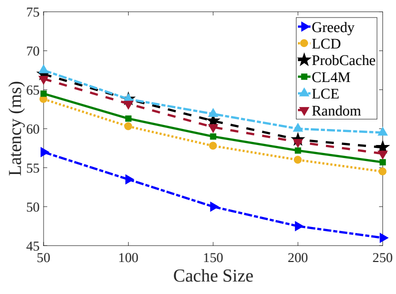


(b) Custodian = 2

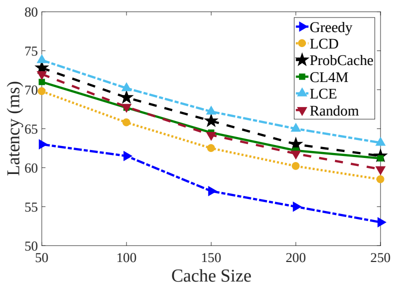


(c) Custodian = 5

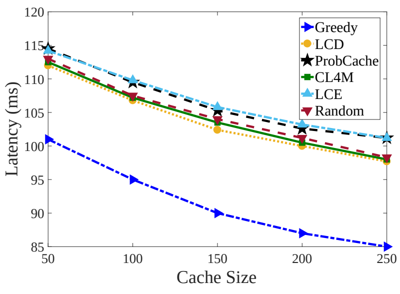
Figure 8: Latency for GARR with FIFO cache eviction for dynamic caching policies



(a) Single custodian

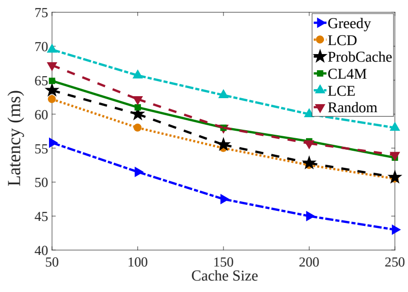


(b) Custodian = 2

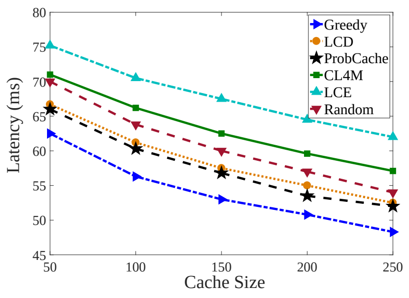


(c) Custodian = 5

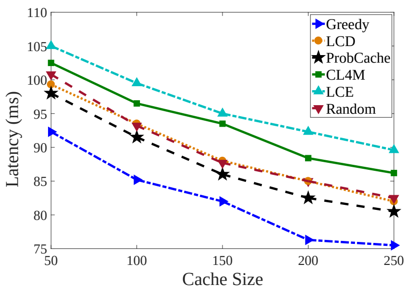
Figure 9: Latency for GARR with Random cache eviction for dynamic caching policies



(a) Single custodian

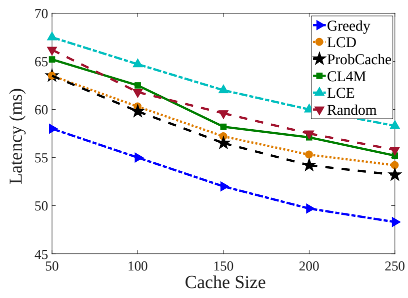


(b) Custodian = 2

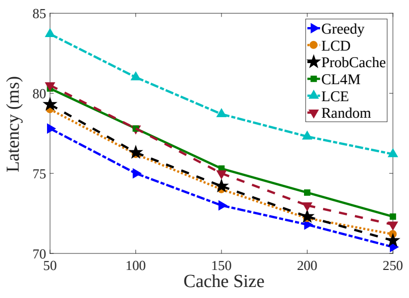


(c) Custodian = 5

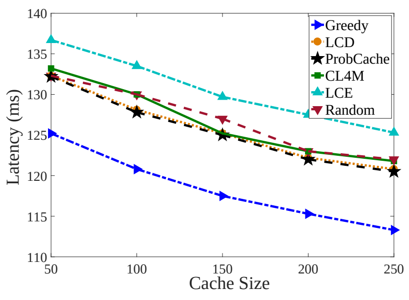
Figure 10: Latency for GEANT



(a) Single custodian



(b) Custodian = 2

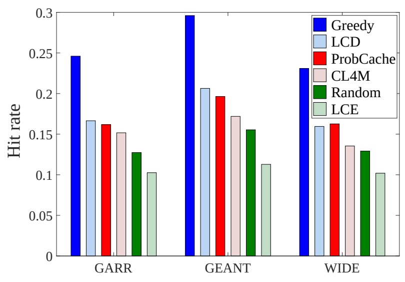


(c) Custodian = 5

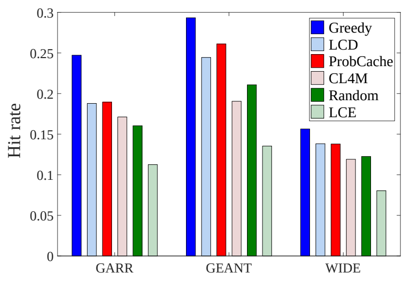
Figure 11: Latency for WIDE

5.7. Performance on Scale-free Network

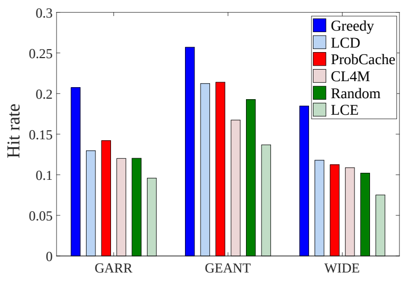
To demonstrate the scalability of *Greedy Caching*, we investigate its performance on scale-free networks. We generate scale-free networks using the Barabási-Albert preferential attachment model [35]. In the preferential attachment model,



(a) Single custodian

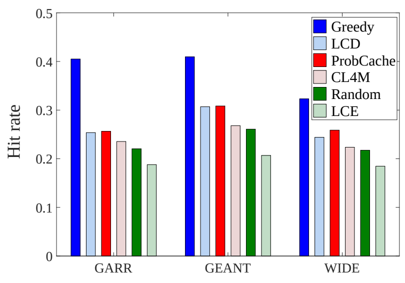


(b) Custodian = 2

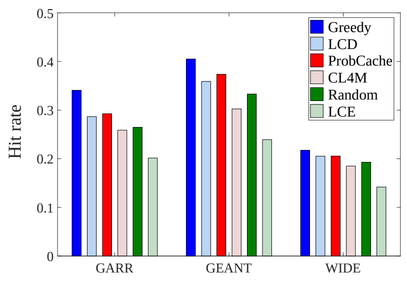


(c) Custodian = 5

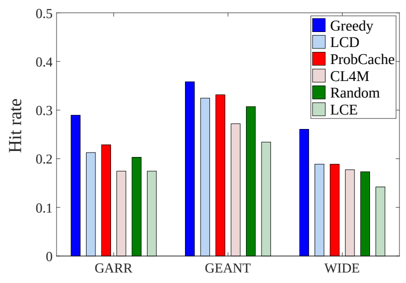
Figure 12: Hit rate for $\alpha = 0.8$, $C = 50$



(a) Single custodian

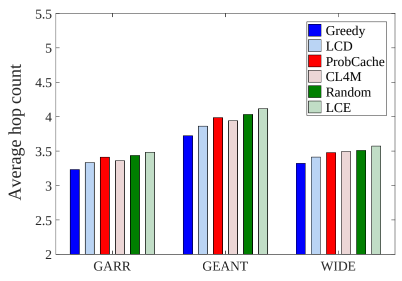


(b) Custodian = 2

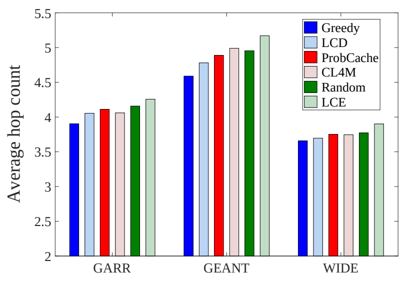


(c) Custodian = 5

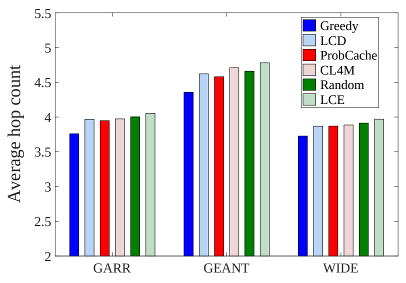
Figure 13: Hit rate for $\alpha = 0.8$, $C = 150$



(a) Single custodian

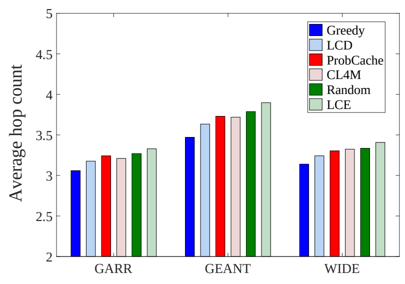


(b) Custodian = 2

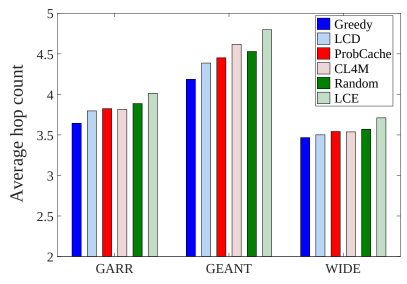


(c) Custodian = 5

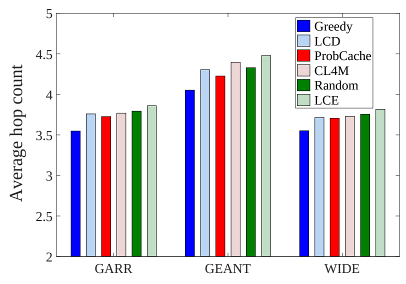
Figure 14: Average hop count for $\alpha = 0.8$, $C = 50$



(a) Single custodian



(b) Custodian = 2



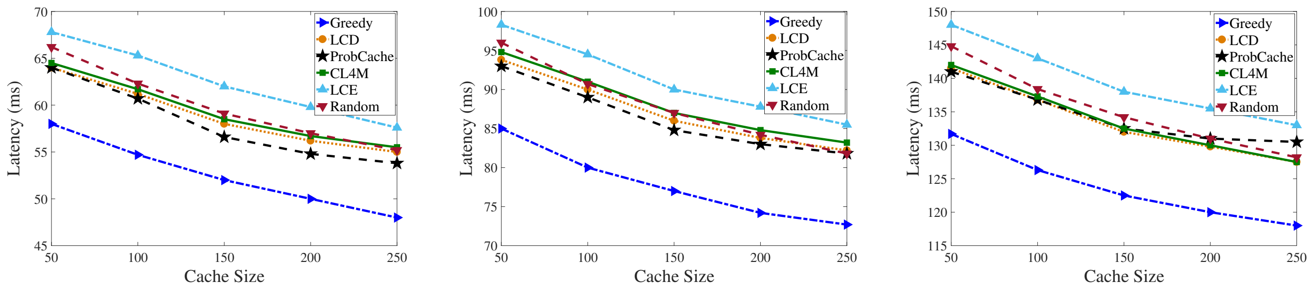
(c) Custodian = 5

Figure 15: Average hop count for $\alpha = 0.8$, $C = 150$

newly added nodes have a greater affinity to connect with nodes with a higher degree. In this subsection, we report results for scale-free networks comprising of 200 nodes. We note that as network size increases, the length of the shortest path to reach

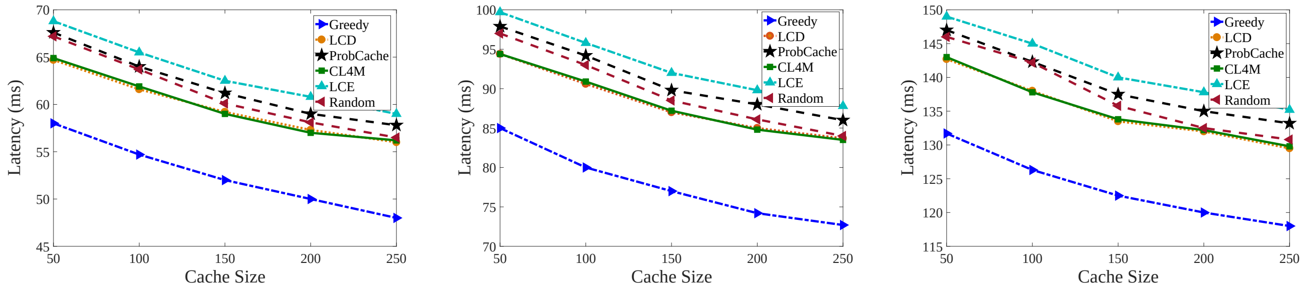
Table 1: Link load for GARR for $\alpha = 0.8$

Strategy	Custodian 1		Custodian 2		Custodian 5	
	$C = 50$	$C = 250$	$C = 50$	$C = 250$	$C = 50$	$C = 250$
Greedy	0.06±0.014	0.053±0.008	0.058±0.027	0.05±0.016	0.048±0.013	0.043±0.009
LCD	0.063±0.018	0.056±0.01	0.061±0.03	0.053±0.019	0.052±0.017	0.046±0.011
ProbCache	0.065±0.02	0.058±0.012	0.062±0.032	0.053±0.018	0.051±0.016	0.046±0.01
CL4M	0.063±0.019	0.057±0.012	0.061±0.03	0.054±0.019	0.052±0.017	0.047±0.011
LCE	0.067±0.023	0.06±0.015	0.065±0.038	0.057±0.024	0.054±0.019	0.048±0.013
Random	0.066±0.021	0.059±0.013	0.063±0.034	0.055±0.02	0.053±0.017	0.047±0.011



(a) Single custodian (b) Custodian = 2 (c) Custodian = 5

Figure 16: Latency for a 200 node scale-free network



(a) Single custodian (b) Custodian = 2 (c) Custodian = 5

Figure 17: Latency for a 200 node scale-free network with FIFO cache eviction for dynamic caching policies

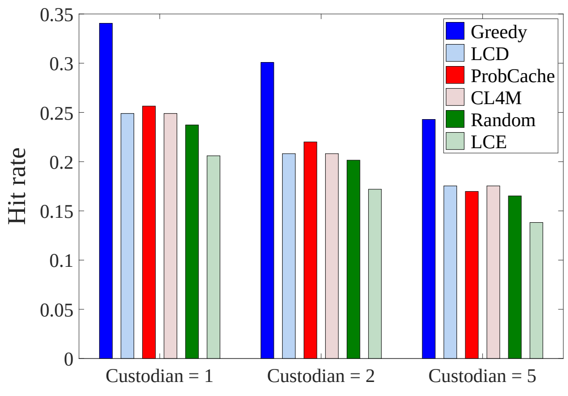
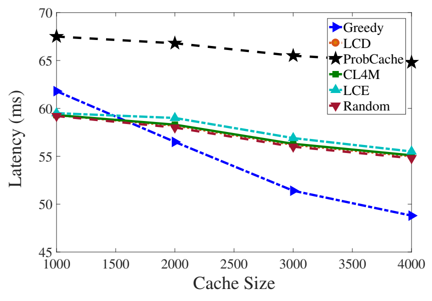


Figure 18: Hit rate for scale-free network

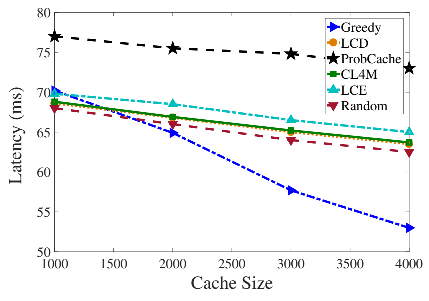
the custodians also increases.

Figures 16 and 17 show the latency for $\alpha = 0.8$ and varying cache size for the LRU and FIFO caching policies respectively. We observe that *Greedy Caching* continues to outperform state-of-the strategies (10 - 23%) for large networks as well. We next present hit rate results for this scale-free network in Figure 18 for $\alpha = 0.8$ and cache size of 50. We observe that the hit rate is greater in comparison to smaller networks for the same set of network parameters (i.e., Figure 12). The main reason is that as the length of the shortest path to the custodian increases, the probability of obtaining the content at an en route cache also increases.

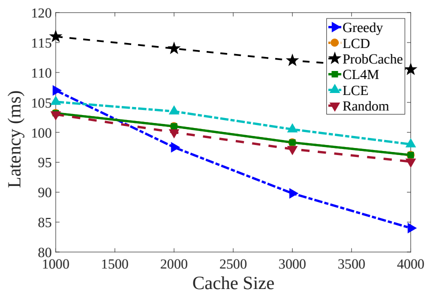
We also study the link-load for the single and multiple custodian scenarios for a scale-free network. We observe that for a scale-free network, the link-load is similar for all policies. We also observe that the average link-load is lower in comparison to earlier results (i.e., Table 1). As the total traffic (which is the same in both cases) is distributed among a greater number of



(a) Single custodian

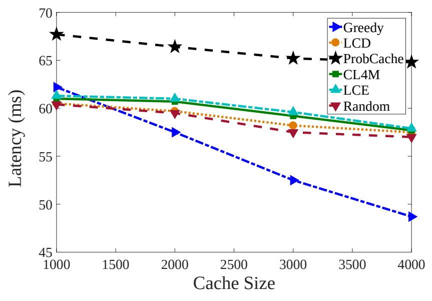


(b) Custodian = 2

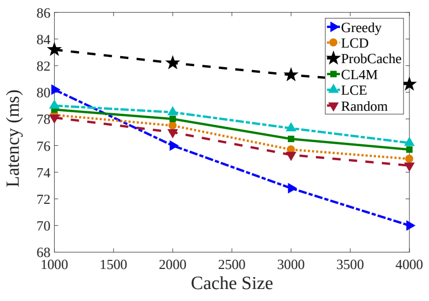


(c) Custodian = 5

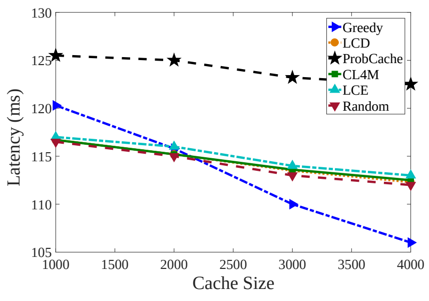
Figure 19: Latency for real request stream traces for GARR



(a) Single custodian



(b) Custodian = 2



(c) Custodian = 5

Figure 20: Latency for real-life trace for WIDE

links in scale-free networks, the average link-load is smaller.

5.8. Performance on Real World Traces

In this section, we conduct trace-based evaluation to investigate the performance of *Greedy Caching* on real-world request streams. For conducting these experiments, we use publicly available request stream traces from the University of Massachusetts Amherst [36]. The traces were collected by monitoring YouTube traffic at the interconnection between the University campus access network and the Internet between June 2007 and March 2008. The trace contains 1467700 total requests, 581527 unique content and 39852 users. We conduct experiments on GARR, WIDE and GEANT, but present results for the GARR and WIDE topology. Once again, we conduct experiments with 1, 2 and 5 custodians. Nodes with degree 1 operate as users in our simulations and the actual 39852 users are mapped to them. In case of multiple custodians, the content is uniformly distributed among the custodians, but permanently stored in only one of them.

We make several observations from the traces and our experiments. The total number of unique content requested is high (around 0.6 million). Approximately 54% of videos are requested only once, while 18% and 13% videos are requested twice and thrice respectively. Therefore, if we consider the entire trace, the popularity is not Zipfian. More importantly, in comparison to synthetic traces that have no temporal correlation, we observe that the real-world trace exhibits high temporal correlation among requests, particularly for unpopular content (i.e., content requested twice or thrice).

Interestingly, we observe that if we consider the entire trace, *Greedy Caching* performs worse than dynamic caching policies. We analyze the main reason behind this behavior. First, *Greedy Caching* caches content based on relative popularity and hence achieves high hit rate for the cached popular content and no hits for the remaining content. We observe that content popularity skewness is not particularly high in this real-world trace. Therefore, if we consider the entire trace, the hit rate obtained for *Greedy Caching* is low.

Second, temporal correlation in request streams aids dynamic caching policies because network nodes dynamically cache new content as it passes through them and can thus obtain hits for recently requested content. By exploiting the correlation in the request stream, dynamic caching policies are able to obtain a significant number of hits for less popular content, thus impacting the overall hit rate. In comparison, *Greedy Caching* that caches content based on relative popularity and filters unpopular content performs poorly, as it fails to serve temporally correlated requests for unpopular content.

Hence, in a real-world setting where content popularity changes over time, we observe that for *Greedy Caching* to be effective it has to be executed periodically. Therefore, we execute *Greedy Caching* periodically after every 50000 requests, assuming that the content popularity for the upcoming 50000 requests is known. We observe that each 50000 requests, consists of approximately 30000 unique content. Figures 19 and 20 show the performance of the different algorithms. We observe that *Greedy Caching* outperforms state-of-the-art strategies for relatively larger cache sizes (i.e., above 2000) at each network node and performs poorly for small cache sizes.

6. Discussion and Future Work

Our initial investigation of *Greedy Caching* raises a number of interesting questions and opens multiple avenues for future research. We discuss them here.

- Our experiments on real-world request stream traces highlight the importance of taking time-varying content popularity and correlations in request streams into account while designing new algorithms and evaluating their performance on multiple real-world traces.

  To determine content popularity in an online fashion, we plan to leverage the rich literature on streaming algorithms and use sketching techniques to efficiently determine the popularity of content based on the incoming request stream [37, 38]. A simple approach to make *Greedy Caching* perform effectively to changing content popularity is to periodically execute the algorithm. We hypothesize that *Greedy Caching* only needs to execute at a coarse time scale (mostly in the order of hours) to capture long-term trends in popularity changes. Of course, determining the optimal time scale at which the algorithm should execute will require further investigation.

  In order to take advantage of the short-term correlations in real-world request streams, we plan to design a hybrid caching strategy where each cache is split into two parts—a static cache that statically caches content as determined by *Greedy Caching* and a dynamic cache that updates itself as new pieces of content pass through it. An interesting question that arises in this context is, what is the optimal split for the hybrid caches? Moreover, we note that performance of the caching algorithms can be improved if future requests can be predicted accurately, content prefetched and cached at nodes closer to the users.

- In this paper, we assume that each network node has a fixed amount of storage. From the content provider's perspective, an interesting question worth investigating is—how to split the overall cache capacity among the different network nodes, so as to maximize network performance? To address the above question, the optimization problem outlined in Section 3 has to be modified such that it outputs the caching capacity at each node along with the content to be cached there.

- Another interesting question worth exploring is the overhead associated with *Greedy Caching* as well as the dynamic caching algorithms considered in this paper. Though a direct comparison is difficult as the nature of the overhead incurred is different, we present an argument as to why *Greedy Caching* and other content placement approaches will incur a lower overhead in comparison to dynamic caching strategies. Assuming that *Greedy Caching* is executed in a centralized manner, the main overhead incurred lies in estimating the content popularity, executing the algorithm periodically and then disseminating the caching information to the different network nodes. As mentioned earlier, content popularity can be efficiently determined via streaming algorithms and distributing 'what content to cache' information to network nodes requires negligible overhead in comparison to the actual content being requested.

  In comparison, though dynamic caching policies do not have to explicitly compute popularity or exchange 'what content to cache' information, the overhead incurred by them lies in writing new content into and erasing old content from the cache memory. As writing content to the cache memory at any network node is expensive and could happen as frequently as on per request basis, we contend that the overall overhead incurred by *Greedy Caching* is significantly lower.

7. Conclusion

In this paper, we proposed *Greedy Caching*, an optimized content placement policy for ICN that works with any underlying routing algorithm and determines the content to be cached at each network node. *Greedy Caching* adopts a greedy approach that considers the request miss stream from downstream caches to make caching decisions at upstream caches. The algorithm attempts to maximize the hit rate at each individual cache. Via extensive simulation, we showed that *Greedy Caching* significantly outperforms state-of-the-art dynamic caching and routing strategies. In future, we plan to extend this work to determine the optimal content placement policy for ICN.

References

[1] L. Saino, I. Psaras, and G. Pavlou, "Icarus: a caching simulator for information centric networking (ICN)," in *SimuTools*, 2014, pp. 66–75.

[2] "Netflix open connect." [Online]. Available: https://openconnect.netflix.com/en/

[3] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: incrementally deployable icn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 147–158.

[4] E. Bastug, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5G wireless networks," *IEEE Commun. Mag.*, vol. 52, no. 8, pp. 82–89, 2014.

[5] A. Dabirmoghaddam, M. M. Barijough, and J. Garcia-Luna-Aceves, "Understanding optimal caching and opportunistic caching at the edge of information-centric networks," in *ACM ICN*. ACM, 2014, pp. 47–56.

[6] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Trans. Inf. Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[7] M. Dehghan, B. Jiang, A. Seetharam, T. He, T. Salonidis, J. Kurose, D. Towsley, and R. Sitaraman, "On the complexity of optimal request routing and content caching in heterogeneous cache networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1635–1648, 2017.

[8] M. Dehghan, A. Seetharam, T. He, T. Salonidis, J. Kurose, and D. Towsley, "Optimal caching and routing in hybrid networks," in *IEEE MILCOM*. IEEE, 2014, pp. 1072–1078.

[9] E. Bastug, M. Bennis, and M. Debbah, "Social and spatial proactive caching for mobile data offloading," in *IEEE ICC Communications Workshops*. IEEE, 2014, pp. 581–586.

[10] E. Zeydan, E. Bastug, M. Bennis, M. A. Kader, I. A. Karatepe, A. S. Er, and M. Debbah, “Big data caching for networking: Moving from cloud to edge,” *IEEE Commun. Mag.*, vol. 54, no. 9, pp. 36–42, 2016.

[11] S. Zaman and D. Grosu, “A distributed algorithm for the replica placement problem,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 9, pp. 1455–1468, Sep. 2011.

[12] L. Qiu, V. Padmanabhan, and G. Voelker, “On the placement of web server replicas,” in *IEEE INFOCOM*, vol. 3, 2001, pp. 1587–1596.

[13] J. Kangasharju, J. Roberts, and K. Ross, “Object replication strategies in content distribution networks,” *Comput. Commun.*, vol. 25, no. 4, pp. 376–383, 2002.

[14] N. Laoutaris, H. Che, and I. Stavrakakis, “The LCD interconnection of LRU caches and its analysis,” *Perform. Eval.*, vol. 63, no. 7, pp. 609–634, July 2006.

[15] W. K. Chai, D. He, I. Psaras, and G. Pavlou, “Cache less for more in information-centric networks,” in *IFIP Networking*, 2012, pp. 27–40.

[16] I. Psaras, W. K. Chai, and G. Pavlou, “Probabilistic in-network caching for information-centric networks,” in *ACM ICN*, 2012, pp. 55–60.

[17] Y. Li, T. Lin, H. Tang, and P. Sun, “A chunk caching location and searching scheme in content centric networking,” in *IEEE ICC*, June 2012, pp. 1550–3607.

[18] I. Psaras, W. K. Chai, and G. Pavlou, “In-network cache management and resource allocation for information-centric networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 2920–2931, Nov. 2014.

[19] K. Suksomboon, S. Tarnoi, Y. Ji, M. Koibuchi, K. Fukuda, S. Abe, N. Motonori, M. Aoki, S. Urushidani, and S. Yamada, “PopCache: cache more or less based on content popularity for information-centric networking,” in *IEEE LCN*, Oct. 2014, pp. 236–243.

[20] A. Ioannou and S. Weber, “Towards on-path caching alternatives in information-centric networks,” in *IEEE LCN*, 2014, pp. 362–365.

[21] C. Bernardini, T. Silverston, and O. Festor, “MPC: popularity based caching strategy for content centric networks,” in *IEEE ICC*, Jun. 2013, pp. 3619–3623.

[22] J. M. Wang and B. Bensaou, “Progressive caching in CCN,” in *IEEE GLOBECOM*, Dec. 2012, pp. 2727–2732.

[23] W. Wang *et al.*, “CRCache: exploiting the correlation between content popularity and network topology information for ICN caching,” in *IEEE ICC*, Jun. 2014, pp. 3191–3196.

[24] Y. Wang, M. Xu, and Z. Feng, “Hop-based probabilistic caching for information-centric networks,” in *IEEE GLOBECOM*, Dec. 2013, pp. 2102–2107.

[25] S. Saha, A. Lukyanenko, and A. Ylä-Jääski, “Cooperative caching through routing control in information-centric networks,” in *IEEE INFOCOM*, Apr. 2013, pp. 100–104.

[26] L. Saino, I. Psaras, and G. Pavlou, “Hash-routing schemes for information centric networking,” in *ACM ICN*, Aug. 2013, pp. 27–32.

[27] S. Wang, J. Bi, J. Wu, and A. V. Vasilakos, “CPHR: In-network caching for information-centric networking with partitioning and hash-routing,” *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2742–2755, Oct. 2016.

[28] V. Sourlas, L. Gkatzikis, P.Flegkas, and L. Tassiulas, “Distributed cache management in information-centric networks,” *IEEE Trans. Netw. Serv. Manage.*, vol. 10, no. 3, pp. 286–299, Sept. 2013.

[29] S. Ioannidis and E. Yeh, “Jointly optimal routing and caching for arbitrary network topologies,” in *ACM ICN*, 2017, pp. 77–87.

[30] O. Ascigil, V. Sourlas, I. Psaras, and G. Pavlou, “A native content discovery mechanism for the information-centric networks,” in *ACM ICN*, 2017, pp. 145–155.

[31] V. Ramachandran, “Finding a minimum feedback arc set in reducible flow graphs,” *Elsevier J. Algoriths*, vol. 9, no. 3, pp. 299–313, Sep. 1988.

[32] P. Eades, X. Lin, and W. F. Smyth, “A fast and effective heuristic for the feedback arc set problem,” *Elsevier Inf. Process. Lett.*, vol. 47, no. 6, pp. 319–323, Oct. 1993.

[33] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, “Networking named content,” in *ACM CoNEXT*, 2009, pp. 1–12.

[34] K. Cho, M. Lee, K. Park, T. T. Kwon, Y. Choi, and S. Pack, “Wave: Popularity-based and collaborative in-network caching for content-oriented networks,” in *IEEE INFOCOM Workshops*. IEEE, 2012, pp. 316–321.

[35] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999.

[36] M. Zink, K. Suh, Y. Gu, and J. Kurose, “Characteristics of YouTube network traffic at a campus network–measurements, models, and implications,” *Comput. Netw.*, vol. 53, no. 4, pp. 501–514, 2009.

[37] G. Cormode, *Sketch techniques for approximate query processing*. NOW Publishers, 2011.

[38] A. Gilbert and P. Indyk, “Sparse recovery using sparse matrices,” *Proc. IEEE*, vol. 98, no. 6, pp. 937–947, 2010.