Theoretical Computer Science 945 (2023) 113686

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# Finding optimal non-datapath caching strategies via network flow ☆

Steven Lyons, Raju Rangaswami, Ning Xie *

*Knight Foundation School of Computing and Information, Florida International University, Miami, FL 33199, USA*

ARTICLE INFO

*Article history:*
Received 30 November 2021
Received in revised form 18 July 2022
Accepted 24 December 2022
Available online 2 January 2023
Communicated by G.F. Italiano

*Keywords:*
Offline cache management
Non-datapath caching
Network flow based algorithm
Cache hit and cache write tradeoff

ABSTRACT

Flash and non-volatile memory (NVM) devices have only a limited number of write-erase cycles. Consequently, when employed as caches, cache management policies may choose not to cache certain requested items in order to extend device lifespan. In this work, we propose a simple single-parameter utility function to model the trade-off between maximizing hit-rate and minimizing write-erase cycles for such caches, and study the problem of developing an off-line strategy for deciding whether to write a new item to cache, and if so which item already in the cache to replace. Our main result is, $_{m}$OPT, an efficient, network flow based algorithm which finds optimal cache management policy under this new setting.

Published by Elsevier B.V.

## 1. Introduction

Flash and non-volatile memory (NVM) devices (such as 3D-XPoint) have replaced magnetic disk drives in many systems to accelerate boot times and enhance overall system performance [12]. However, the cost and size of these devices are comparatively unfavorable when compared to magnetic disk drives, a much slower medium. As a result, enterprise and cloud systems that host and serve large amounts of data instead utilize flash and NVM devices as caches to speed up access to frequently requested items from the much larger but slower magnetic disk drive based storage [12]. These host-side caches, upon a request, have a choice unavailable for DRAM, or other datapath caches: to *not cache* the requested item [17]. We refer to such caches as *non-datapath caches*.

Non-datapath caches, being faster than their backing storage devices but slower than DRAM, demonstrate immediate impact on system performance when employed with appropriate cache management [13]. However, these caches are primarily composed of flash or other NVM devices, which support limited number of write-erase cycles compared to their volatile counterparts (DRAM and SRAM) [6]. Additional writes performed beyond such limits render these devices unusable. As such, cache management policies designed in the non-datapath caching context should focus not only on the hit-rate performance but also on limiting the number of writes made to the cache in order to extend their useful lifetime [17].

In this work, we propose a simple, single-parameter utility function to model the trade-off between maximizing hit-rate and minimizing write-rate for non-datapath caches. As a first step of investigation, we would like to know the best achievable cache performance in this new setting, free of constraints — i.e., the off-line optimal. Specifically, we study the

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
*E-mail addresses:* slyon001@fiu.edu (S. Lyons), raju@cs.fiu.edu (R. Rangaswami), nxie@cis.fiu.edu (N. Xie).

https://doi.org/10.1016/j.tcs.2022.12.036
0304-3975/Published by Elsevier B.V.

following *off-line* cache management problem. Suppose we are given an empty cache consisting of $w$ cache spaces, together with a sequence of requests $\mathcal{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ which are known in advance. On each request $\sigma_i$, if it is already in the cache (i.e. a *cache hit* occurs) our gain is 1; otherwise the item must be fetched from the backing store. When this happens, we may either choose to cache the item and pay a write cost[1] of $\alpha$ or choose not to cache it. Note that the parameter $\alpha$ captures the trade-off between cache hit rate and cache write rate, whose value may be set depending on flash or NVM device quality and performance requirements of the cache.

Now, a cache replacement algorithm $A$ that decides what to do upon each request is associated with a utility function

$$\mathscr{U}(A, \alpha, w, \mathcal{R}) = (\text{number of hits}) - \alpha \cdot (\text{number of writes})$$

The Off-line Non-Datapath Cache Management Problem is to find an optimal replacement policy which maximizes the utility function

$$_{\text{m}}\text{OPT}(\alpha, w, \mathcal{R}) := \max_{A} \mathscr{U}(A, \alpha, w, \mathcal{R}).$$

In the traditional setting, when an item is fetched from a slower back-end storage device, it is necessarily written into the cache; this causes some item that is already in the cache has to be evicted if the cache is currently full — hence the name cache "replacement" problem. It is well-known that Bélády's optimal algorithm [4], the so-called **MIN** algorithm,[2] which always evicts an item in the cache whose next request is the furthest in the future, is optimal in this setting.

Denote by $\text{MIN}(w, \mathcal{R})$ the number of cache hits **MIN** algorithm achieves for a request $\mathcal{R}$ with a cache of size $w$. By setting the write cost $\alpha$ to zero, we may compare the performance of the same size cache under these two different settings. In fact, it is not hard to show (for completeness, we include a proof in Appendix A) that $\text{MIN}(w, \mathcal{R}) \leq {}_{\text{m}}\text{OPT}(0, w, \mathcal{R}) \leq \text{MIN}(w+1, \mathcal{R})$.

Our main result of this work is, ${}_{\text{m}}$OPT, an efficient, network flow based algorithm that finds an optimal solution to the Off-line Non-Datapath Cache Management Problem. Specially, we prove

**Theorem 1.1** *(Main Theorem). Given a request sequence $\mathcal{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ from a set $\Sigma$, with $q = |\Sigma|$, there is an algorithm that computes an optimal caching policy for a non-datapath cache of size $w$, and runs in time $O(w \cdot (qT + T \log T))$.*

**Our approach.** Our solution to the problem of finding an optimal cache management policy is a very natural and simple one. Consider the simplest case when there is only one space in the cache. The set of requested items that have been written into the cache form a sub-sequence of the original request sequence. Therefore, it is natural to model this sub-sequence as a flow among requested items; that is, the history of the cache can be mapped to a flow in a network in which requested items are nodes and all arcs are from earlier requested items to later ones. How do we model the utility function (maximizing hit rates and minimizing write rates)? If the current node is $(\sigma, t)$ (i.e., item $\sigma$ is requested at time $t$ and is written into the cache), and the next item written into the cache is $(\sigma', t')$ with $t' > t$, then we need to pay a write cost $\alpha$ plus the $(t' - t)$ cache misses[3] between these two consecutive cache writes, assuming all the items in the request sequence between $(\sigma, t)$ and $(\sigma', t')$ are distinct from $\sigma$ (we will discuss cache hits shortly). We therefore connect nodes $(\sigma, t)$ and $(\sigma', t')$ by a directed arc with cost $\alpha + (t' - t)$. How to model cache hits? Suppose $(\sigma, t'')$, where $t'' > t$, is the *immediate* next appearance of item $\sigma$ in the request sequence, then we draw an arc from $(\sigma, t)$ to $(\sigma, t'')$ with cost $(t'' - t - 1)$ to model the scenario that item $\sigma$ stays in the cache between time $t$ and $t''$ so that it incurs only $(t'' - t - 1)$ cost of cache misses (the extra $-1$ in the cost accounts for a cache hit), but incurs no write cost. In general, we reduce Off-line Non-Datapath Cache Management Problem to the Minimum Cost Flow Problem by constructing a flow network with vertices being the set of items in the requests sequence (with an additional source node and a sink node), and connect two requested items with an edge if and only if they *may* be cached into the same cache space *consecutively*. Proof of the correctness of this reduction is less trivial. We first show that the minimum cost flow of the network must be integral-valued. Since all the edge weights in our constructed network are integers, this implies that the optimal flow can be decomposed into a set of edge-disjoint paths between the source and sink. Then we prove that, due to our choice of the edge weights and flow costs, one can read off the value of the maximum utility function from the value of the minimum cost flow. This is because there is a one-to-one correspondence between the edge-disjoint paths from the source to the sink in the network, and the cache state time-evolution of the non-datapath cache spaces.

**Related work.** The use of network flows to solve caching problems has produced several solutions to other problems. In order to find optimal off-line caching with variable object sizes, a network flow based algorithm *FOO* [5] is designed, in which flow is used to represent the "interval decision variables" — variables that track the state of each requested item

[1] Of course, if the cache is already full, we have to evict one of the existing entries, hence $\alpha$ is actually the total cost of eviction and write operations. For simplicity of the model, we will not distinguish between the cost of writing with and the cost of writing without evictions in this work.

[2] Proof of the optimality of the **MIN** algorithm was first shown in [15]; see also [18] for a more recent simplified proof.

[3] It turns out to be more convenient to work with minimizing cache misses instead of maximizing cache hits.

at every time step. Another solution that uses network flows is *CHOPT* [20], which determines the level of a multi-tiered (2 or more) cache hierarchy an object should be placed in to minimize latency. Finally, *BELATEDLY* [3] uses a network flow to minimize latency while accounting for delayed hits, which accounts for the latency of cache insertions. All of work mentioned above addresses various objects sizes or latency from different perspectives; $_{\mathrm{m}}$OPT, on the other hand, focuses on constant sized objects and optimizing hit-rate and write-rate.

Related solutions that have focused on non-datapath caches with a desire to minimize writes include the following heuristics. $M^+$ [7] runs Bélády's **MIN** with the caveat of not inserting objects into the cache that do not result in a cache hit. $M_N$ [7] focuses on achieving a targeted limit on writes to the cache, in particular flash erasures. And lastly, $M_T$ [7], which chooses not to insert an object into the cache if it is not read for a certain number of times. While these heuristics have goals similar to that of $_{\mathrm{m}}$OPT, their solutions are approximations while $_{\mathrm{m}}$OPT solves for *exact* optimal values of utility functions.

### 1.1. Further remarks on our model and the write cost parameter $\alpha$

Instead of viewing our new model as a realistic accounting of caching processes in practical systems, we prefer to regard the model bundled with the efficient $_{\mathrm{m}}$OPT algorithm as a useful toolkit for studying the general caching problems of systems with various write constraints. While in this work we focus on cache misses and writes as optimality objective, the cost function defined in terms of misses and writes can be adapted to deal with latency, power consumption, cost, etc to solve similar problems.

Suppose that we are given a cache with lifetime[4] $W$. First of all, we should try to avoid wearing out the device, as after that caching is no longer valid. Given a (very long) request sequence $\mathcal{R}$, how to set the write cost parameter $\alpha$ to avoid wearing out? Note that choice of $\alpha$ depends not only on device lifetime but also on the input request sequence as well.

As write rate is clearly a monotone decreasing function of $\alpha$, one way to explore the relation between write cost $\alpha$ and write rates is to first set $\alpha$ to be a very small number, say $1/(T+1)$, so that write cost will not affect the hit rate. Then gradually increase the value of $\alpha$ (or perform a more efficient binary search), and record the number of writes to the cache for each value of $\alpha$. In this way, we can estimate, for a *fixed* request sequence, which value of $\alpha$ will cause say, $W/2$ writes to the cache. Such estimates should in principle provide a good guideline for request sequences with similar patterns and behaviors. Indeed, we believe the main power of our simple single-parameter model comes from the flexibility in choosing the write cost parameter $\alpha$. Consequently, we expect that in a typical practical application, $_{\mathrm{m}}$OPT is run multiple times with different settings $\alpha$ to study the optimal caching behaviors.

While the celebrated off-line Belady algorithm provides a very nice characterization for the online optimal solutions in the simple caching scenario, we believe that such a clean characterization is very unlikely to exist for the more complicated situations in which write constrains are imposed. Instead, a more likely scenario is that for different request sequence patterns and different settings of trade-off between hit rate and write rate, there are correspondingly different heuristics which achieve the best possible performance. It is thus our hope that the proposed simple model and the efficient $_{\mathrm{m}}$OPT will provide the optimal solution as a yardstick, shed light on searching for such heuristics, and furnish intuitions along the path toward developing a more unified theory of caching strategies. Such a hope is buttressed by the fact that online algorithms for related problems have been designed based on understanding gleaned from the network flow decisions obtained from offline optimal solutions, e.g., *BELATEDLY* and it's online algorithm MAD [3].

## 2. Background on flow networks and algorithms

In the following, we use $\mathbb{R}^+$ to denote the set of non-negative real numbers.

### 2.1. Flow networks

A *network* is a quadruple $(G, c, \mathbf{s}, \mathbf{t})$, where $G = (V, E)$ is a directed graph, $c : E(G) \to \mathbb{R}^+$ is an edge capacity function, and $\mathbf{s}$ (the source) and $\mathbf{t}$ (the sink) are two specified vertices in $V(G)$. A *flow* in a network $(G, c, \mathbf{s}, \mathbf{t})$ is a real-valued function $f : E(V) \times E(V) \to \mathbb{R}^+$ satisfying the following two properties. (i) Capacity constraint: $f(i, j) \le c(i, j)$ for all $(i, j) \in E(G)$ and (ii) Flow conservation: for all $i \in V(G) \setminus \{s, t\}$, $\sum_{j \in V(G)} f(i, j) = \sum_{j \in V(G)} f(j, i)$. If $(i, j)$ is an edge in $G$, then the non-negative quantity $f(i, j)$ is called the flow from vertex $i$ to vertex $j$. The value of a flow $f$ is defined as $|f| = \sum_{i \in V(G)} f(\mathbf{s}, i) - \sum_{i \in V(G)} f(i, \mathbf{s})$.

The following useful fact allows us, after combining with the special edge capacities of the constructed network, to decompose any feasible flow from $s$ to $t$ into a collection of *edge-disjoint* paths from $s$ to $t$.

**Theorem 2.1** *(Flow Decomposition Theorem [9]). Let $(G, c, \mathbf{s}, \mathbf{t})$ be a network and let $f$ be an $\mathbf{s}$-$\mathbf{t}$-flow in $G$. Then there exists a family $\mathcal{P}$ of $\mathbf{s}$-$\mathbf{t}$-paths and a family $\mathcal{C}$ of cycles in $G$ along with a weight function $\mathscr{W} : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}^+$ such that for every edge $(i, j) \in E(G)$, $f(i, j) = \sum_{P \in \mathcal{P} \cup \mathcal{C} : (i,j) \in E(P)} \mathscr{W}(P)$ and $\sum_{P \in \mathcal{P}} \mathscr{W}(P) = |f|$. Moreover, if $f$ is integral then all weights $\mathscr{W}$ can be chosen to be integral.*

[4] That is, the device will wear out after $W$ writes. We are grateful to an anonymous reviewer for bringing up this question to our attention.

### 2.2. The minimum cost flow problem

Our approach for the Off-line Non-Datapath Cache Management Problem is to reduce it to the Minimum Cost Flow Problem,[5] and then apply a well-known Edmond-Karp algorithm to solve the problem.

**Definition 2.2** *(Minimum Cost Flow Problem).* An instance of the Minimum Cost Flow Problem is a quadruple $(G, c, a, b)$, where $G = (V, E)$ is a directed graph, $c : E(G) \to \mathbb{R}^+$ is an edge capacity function, $a : E(G) \to \mathbb{R}$ is a cost function (we will often refer these values as the "weights" of edges), and $b : V(G) \to \mathbb{R}$ is a flow function specifying the total flow "injected" into each vertex in $G$ with the constraint $\sum_{v \in V(G)} b(v) = 0$. A *b-flow* in $(G, c, a, b)$ is any real-valued function $f : E(V) \times E(V) \to \mathbb{R}^+$ satisfying the following two properties. (i) Capacity constraint: $f(i, j) \le c(i, j)$ for all $(i, j) \in E(G)$ and (ii) Flow constraint: for all $i \in V(G)$, $\sum_{j \in V(G)} f(i, j) - \sum_{j \in V(G)} f(j, i) = b(i)$ (that is, for every vertex $i$ in the network, the net flow out of the vertex is equal to the "injected" flow $b(i)$ into that vertex). The Minimum Cost Flow Problem is to find a $b$-flow $f$ which minimizes the cost function $\mathcal{C}(f) := \sum_{(i,j) \in E(G)} a(i, j) f(i, j)$.

Minimum Cost Flow Problem is one of the central problems in network flow research and has been extensively studied during the past half century; see e.g. [2,14,19] and references therein. To date, the fastest algorithms for the minimum cost flow problem are [1,11,16], and the fastest strongly polynomial[6] algorithm for the Minimum Cost Flow Problem is due to Orlin [16], with a running time $O(m \log n(m + n \log n))$ for networks with $n$ vertices and $m$ edges.

### 2.3. Edmond-Karp algorithm

Early network flow algorithms rely crucially on the concept of *residual network*, introduced by Ford and Fulkerson [9]. Given a flow network $G$ and a flow $f$, the residual network $G_f$ consists of the same set of vertices as $G$, but with additional edges and modified edge capacity. Specifically,[7] $c_f(i, j) = c(i, j) - f(i, j)$ if $(i, j) \in E(G)$, and $c_f(i, j) = f(j, i)$ if $(i, j) \notin E(G)$. Moreover, the edge weights of the original edges in $G$ are unchanged, and the newly added edges $(j, i)$ has weights $a_f(j, i) = -a(i, j)$.

The Edmond-Karp algorithm [8] for the Minimum Cost Flow Problem is built on the following basic result (see e.g. [9], p. 121).

**Theorem 2.3.** *Let $(G, c, a, b)$ be an instance of the* Minimum Cost Flow Problem *and let $f$ be a minimum cost b-flow. Let $P$ be a shortest* **s**-**t** *path in $G_f$ with respect to edge weight $a_f$ (i.e. use the edge cost function as the edge weights). Let $f'$ be a flow obtained by augmenting $f$ along path $P$ and denote the resulting flow function as $b'$. Then $f'$ is a minimum cost $b'$-flow.*

The Edmond-Karp algorithm starts from an empty flow $f = 0$. It then repeats the following process until no such vertices **s** and **t** can be found: Choose a vertex **s** with $b(\mathbf{s}) > 0$, choose a vertex **t** with $b(\mathbf{t}) < 0$, such that **t** is reachable from **s** in $G_f$; Augment the flow $f$ along a shortest path between **s** and **t** with flow equaling the minimum edge capacity along the path; update $b$.

In this way, computing a minimum cost flow is reduced to finding a shortest **s**-**t** path in the residual network defined by the previous flow, and augmenting the flow along this shortest path. As computing a shortest **s**-**t** path in a general graph with negative edge weights is costly (the best known Bellman-Ford algorithm takes $O(nm)$ time), the key insight of Edmond-Karp is the following. We can introduce a "potential function" $\pi_f(i)$ for each vertex $i$ in the residual graph, which is defined as the shortest path distance $\delta(\mathbf{s}, i)$ between **s** and $i$ in $G_f$. Then use $\bar{a}_f(i, j) = a_f(i, j) + \pi_f(i) - \pi_f(j)$ − which are guaranteed to be non-negative − as the edge weights to perform shortest path calculation. By employing Fibonacci heap [10], Edmond-Karp algorithm runs in $O(B(m + n \log n))$ time, where $B = \frac{1}{2} \sum_{i \in V(G)} |b(i)|$ is the size of the flow, $n$ is the number of vertices and $m$ is the number of edges in the network. See e.g. [14] for a detailed description and analysis.[8]

## 3. Network construction

Let $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$ denote the set of possible requested items with $q = |\Sigma|$. Without loss of generality, assume that every symbol $\sigma \in \Sigma$ appears in the request (otherwise, we can remove $\sigma$ from $\Sigma$). Let $\mathcal{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ be a sequence of $T$

[5] Another related and slightly more popular problem is the Minimum Cost Circulation Problem. It is well-known, see e.g. [19], that the Minimum Cost Flow Problem is reducible to the Minimum Cost Circulation Problem. We choose to use the former because its formulation is more intuitive to model the write/evict history of a cache space, as explained in Section 1.

[6] Roughly speaking, an algorithm is said to be *strongly polynomial* if its running time is a polynomial only of the problem instance size (in our case, the number of vertices and edges of the network) but independent of the parameters of the problem instance (in our case, the quantities such as capacity $c$, cost $a$ and flows $b$), as long as we can assume that all basic arithmetic operations involving these parameters can be computed in $O(1)$ time.

[7] Without loss of generality, for convenience, we assume that in the network $(i, j) \in E(G)$ implies $(j, i) \notin E(G)$. If this condition is not met, we may modify $G$ by adding auxiliary vertices. Such a condition is certainly satisfied for the network constructed in this paper.

[8] The running time stated here applies only to the network constructed in this paper; since our original network does not have any negative weighted edges, so the initial run of the Bellman-Ford algorithm to find a shortest path from **s** to every other vertex in $G$ is avoided in our case.

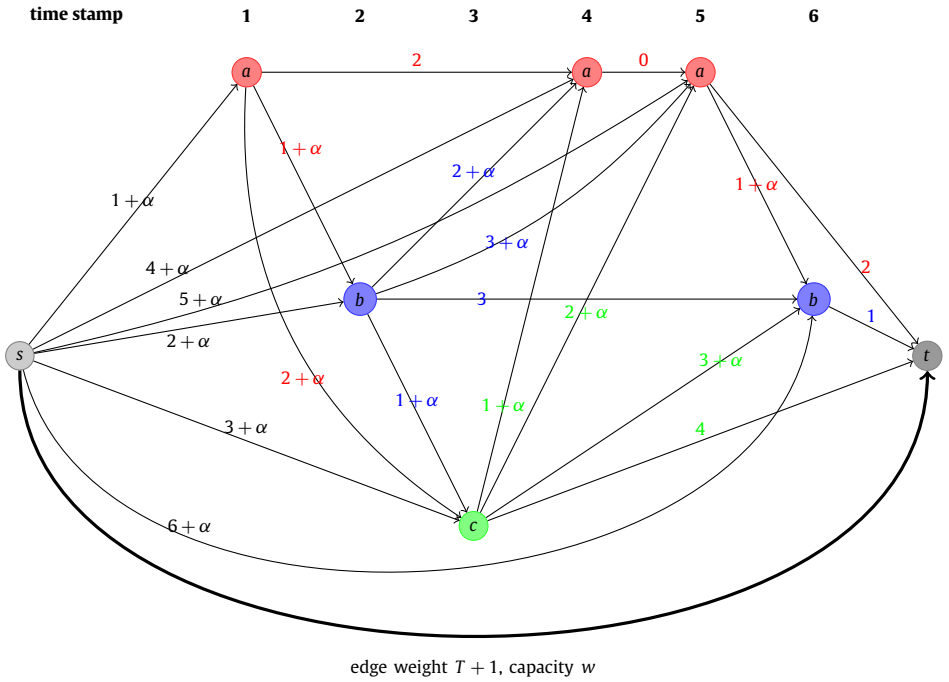


**Fig. 1.** The constructed network when the request sequence is $\{a, b, c, a, a, b\}$, with $T = 6$. Nodes with the same symbols share a unique color. The weight of each edge shares the same color of the node from which it emanates. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

requests. Therefore, each request can be represented by a tuple $(\sigma, t)$, where $\sigma \in \Sigma$ and $t \in \{1, \ldots, T\}$. Abusing notation, we will use $\mathcal{R}$ to also denote the set of such request tuples. For every $\sigma \in \Sigma$, let $\text{first}(\sigma)$ be the time stamp at which $\sigma$ first appears in the request, and $\text{last}(\sigma)$ be the time stamp of its last appearance in the request. Finally, for any $(\sigma, t) \in \mathcal{R}$, let $\text{next}(\sigma, t) = \min\{t' > t \mid (\sigma, t') \in \mathcal{R}\}$ denote the next time stamp at which symbol $\sigma$ appears in the request sequence after time $t$, and set it to $T + 1$ if $(\sigma, t)$ is the last occurrence of item $\sigma$.

Given an instance of the Off-line Non-Datapath Cache Management Problem with cache size $w$, write cost $\alpha$, and request sequence $\mathcal{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$, an instance $(G, c, a, b)$ of the Minimum Cost Flow Problem is constructed as follows.

- **Vertices** $V(G)$
  Let $V' \subset \Sigma \times \{1, 2, \ldots T\}$ be the set of all request tuples; that is $V' = \{(\sigma, t) \mid \text{the requested item at time } t \text{ is } \sigma\ \}$. Then $V(G) = \{\mathbf{s}, \mathbf{t}\} \cup V'$, where **s** is a designated source node and **t** is a designated sink node, neither in $V'$.
- **Edges** $E(G)$ and **edge weights** $a(u, v)$
  - (source edges) For every $(\sigma, t) \in V'$, add an edge from **s** to $(\sigma, t)$ of weight $t + \alpha$. We will refer to these edges as *source edges*.
  - (sink edges) For every $\sigma \in \Sigma$, add an edge from $(\sigma, \text{last}(\sigma))$ to **t** of weight $T + 1 - \text{last}(\sigma)$, and call such edges *sink edges*.
  - (straight edges) For every vertex $(\sigma, t)$ such that $t < \text{last}(\sigma)$ (i.e. $t$ is not the last time stamp at which $\sigma$ appears in the request), add an edge from $(\sigma, t)$ to $(\sigma, \text{next}(\sigma, t))$ of weight $\text{next}(\sigma, t) - t - 1$. Note that the weight is a non-negative integer. We refer to such edges as *straight edges*.
  - (slanted edges) For every vertex $(\sigma, t)$, if $\text{next}(\sigma, t) - t > 1$ (i.e., there are some other symbols between the current appearance of $\sigma$ and the next appearance of $\sigma$), then for every $t < t' < \text{next}(\sigma, t)$, add an edge from $(\sigma, t)$ to $(\sigma', t')$ of weight $t' - t + \alpha$. Since $\sigma'$ must be different from $\sigma$, such edges will be called *slanted edges*.
  - (**s**-**t** edge) Add an edge from **s** directly to **t** of weight $T + 1$.

  All edges in our network have unit capacity, except for the last edge from **s** directly to **t**, which has capacity $w$.
- $b$**-flow**
  We set the $b$-flow as follows: $b(\mathbf{s}) = w$, $b(\mathbf{t}) = -w$, and $b(i) = 0$ for every $i \in V'$.

In Fig. 1 we provide an example to illustrate the construction of the flow network.

**Lemma 3.1.** *The constructed network $(G, c, a, b)$ satisfies that $|V(G)| = T + 2$ and $|E(G)| \leq qT + 2T + 1$. Consequently, the Edmond-Karp algorithm described in Section 2.3 can be used to find a minimum cost flow for $(G, c, a, b)$ in time $O(w(qT + T \log T))$.*

**Proof.** The size of $V(G)$ is straightforward. To calculate $|E(G)|$, first note that there are exactly $T$ source edges, $q$ sink edges, and another edge from source directly to sink. For any vertex $(\sigma, t) \in V'$ its total number of outgoing edges (counting both straight edge and slanted edges) is exactly $\text{next}(t) - t$. For any fixed item $\sigma$, if we sum all the out-degrees of vertices $(\sigma, t)$, then

$$\sum_{t:(\sigma,t)\in V'} \text{out-deg}((\sigma, t)) = \sum_{t:(\sigma,t)\in V'} (\text{next}(t) - t) = T - \text{first}(\sigma) < T.$$

Since there are $q$ such symbols, the total number of such edges is at most $qT$. It follows that $|E(G)| \leq qT + q + T + 1 \leq qT + 2T + 1$, since $q \leq T$ by our assumption. □

## 4. Proof of the main theorem

The main point of the proof is to establish a one-to-one correspondence between the paths from **s** to **t** in the network and which items have been successively cached into a fixed cache space. Specifically, a straight edge corresponds to a cache hit (the requested item is already stored in cache) and a slanted edge corresponds to evicting the old item and write a requested item into that cache space.

We begin by noting that the existence of a capacity $w$ edge from **s** to **t** makes the instance of Minimum Cost Flow Problem *feasible*. It is easy to see that a flow $f_0 \in \mathbb{N}$ along the $(\mathbf{s}, \mathbf{t})$ edge corresponds to that $f_0$ cache spaces of the whole cache have never been used at all, making the effective cache size $w - f_0$. For this reason, in the following, we assume that the flow along this edge is zero.[9] Let $f$ be *any* feasible $b$-flow of the Minimum Cost Flow Problem instance we constructed, and let $\mathcal{C}(f)$ be its corresponding cost. By applying Theorem 2.1 to $f$ and noting that all edges have unit capacity, we conclude that flow $f$ can be decomposed into $w$ edge-disjoint **s**-**t** paths in $G$.

**Lemma 4.1.** *For the purpose of finding optimal cache management policies, we may assume without loss of generality that every caching policy satisfies i) when a cache hit occurs, no cache replacement happens; ii) no item is stored in two distinct cache space at any time.[10] Then there is a one-to-one correspondence between the edge-disjoint* **s**-**t** *paths in $G$ and the lists which store the sequences of items that cached into the same space in the non-datapath cache.*

**Proof.** First of all, it is easy to see that every edge-disjoint **s**-**t** path corresponds to a sequence of items that have been cached into the same cache space. Let $\mathbf{s} \to (\sigma_1, t_1) \to (\sigma_2, t_2) \to \cdots \to (\sigma_k, t_k) \to \mathbf{t}$ be such a path. By our network construction, $t_1 < t_2 < \cdots < t_k$, and each directed edge in $G$ (except for the sink edges) corresponds to a legal cache replacement operation. Therefore, this path can be mapped to the following sequence of cached items: store $\sigma_1$ in cache; for $1 < i \leq k$, if $\sigma_i = \sigma_{i-1}$ then a cache hit occurs ($\sigma_i$ remains stored in the cache); otherwise, evict $\sigma_{i-1}$ and write $\sigma_i$ into the cache.

Next we show that, for every $w \geq 1$, $w$ lists of sequences of symbols which store items that have been cached into the same space by any caching policy, can be mapped to $w$ edge-disjoint **s**-**t** paths in $G$. We prove this by induction on $T$. Additionally, we prove an invariant that, at any time, the items stored in the cache are distinct and they are the latest occurrences of those symbols.

For $T = 1$, it is easy to check that two cache configurations, i.e. either to cache the item (list consists of item $\sigma_1$ only) or not (empty list), correspond to the path $\mathbf{s} \to (\sigma_1, 1) \to \mathbf{t}$ and the path $\mathbf{s} \to \mathbf{t}$, respectively.

Inductively, suppose that there is such a mapping for any request sequence $\mathcal{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ of length $T$ and for any caching policy, and additionally the invariant holds. That is, for any $w$ lists $B_1, \ldots, B_w$ which store the cache sequences of $w$ cache spaces of any caching policy,[11] we can map them into $w$ edge-disjoint **s**-**t** paths in $G$

$$\mathbf{s} \to v_{1,1} \to \cdots \to v_{1,k_1} \to \mathbf{t},$$
$$\cdots\cdots,$$
$$\mathbf{s} \to v_{w,1} \to \cdots \to v_{w,k_w} \to \mathbf{t}.$$

Let $\text{CACHE} := \{\sigma(v_{1,k_1}), \ldots, \sigma(v_{w,k_w})\}$ denote the set of items currently stored in the cache. By our invariant assumption, all items in CACHE are distinct and are the latest occurrences in the request sequence.

[9] Alternatively, we may construct a network with this edge removed from $G$ and try flow value $b = 1, 2, \ldots$ to a maximum value $w'$ at which a $w'$-flow between **s** and **t** is still feasible in the modified network. Then $w'$ is the maximum cache size can be fully utilized by the request sequence.

[10] It is easy to check that both of the assumptions hold as long as the writing cost $\alpha$ is positive.

[11] Note that some of the lists in $B_1, \ldots, B_w$ may be empty, which correspond to unused cache spaces; correspondingly, they all map to the direct **s**-**t** edge and we view the **s**-**t** edge of flow value $b$ as $b$ disjoint paths. However, by our assumption that all cache spaces will be used eventually, such a mapping is temporary and is employed only to facilitate the proof.

Let $\sigma'$ be the $(T+1)$st item in the request sequence. Consider all the possible actions taken by a caching policy on arrival of this new request.

**Case 1:** $\sigma' \in$ CACHE**.** Suppose $\sigma' = \sigma(v_{i,k_i})$ for some $1 \le i \le w$. Since a cache hit occurs, by our assumption, no cache replacement takes place. Then we append $\sigma'$ to the end of list for $B_i$. Note that since $v_{i,k_i}$ is the last occurrence of item $\sigma'$, by our network construction, there is an edge from $v_{i,k_i}$ to $(\sigma', T+1)$, therefore we can modify the $i$-th path to

$$\mathbf{s} \to v_{i,1} \to \cdots \to v_{i,k_i} \to (\sigma', T+1) \to \mathbf{t},$$

and leave other paths unchanged. Now all items in CACHE are still distinct and are the latest occurrences in the request sequence.

**Case 2:** $\sigma' \notin$ CACHE**.** If the caching policy decides to not cache $\sigma'$, then there is nothing to prove: both the lists and the paths remain unchanged. If the caching policy decides to cache $\sigma'$, there are two possibilities. First, if there is an empty cache space (i.e. some list $B_i$ is empty) and the policy decides to store $\sigma'$ there, then we change the original direct **s**-**t** edge corresponding to this cache space to a new path

$$\mathbf{s} \to (\sigma', T+1) \to \mathbf{t}.$$

Such a path is possible because $(\sigma', T+1)$ corresponds to the last occurrence of item $\sigma'$ in the request sequence. Moreover, the invariant is preserved after adding the new path and updating the empty list to a new single-item list. Second, if there is no empty cache space, or there is some empty cache space but the caching policy decides to save the space, then one of the symbols in CACHE will be evicted and $\sigma'$ will be stored into that space. Suppose the symbol currently stored in the $i$-th space is to be evicted, then we append $\sigma'$ to the end of list $B_i$. The rest of the proof is identical to that of Case 1 and is thus omitted. □

We remark that, the proof of Lemma 4.1 implies that the $w$ edge-disjoint **s**-**t** paths decomposed from the optimal flow $f$ are *vertex*-disjoint as well.[12] This is because, if two distinct **s**-**t** paths $P'$ and $P''$ intersect at a vertex $(\sigma, t)$, then there are two vertices $(\sigma', t')$ and $(\sigma'', t'')$, $t > t' > t''$, such that edges $((\sigma', t'), (\sigma, t))$ is in $P'$ and $((\sigma'', t''), (\sigma, t))$ is in $P''$. But this implies that the caching policy stores item $\sigma$ in two distinct cache spaces at time $t$, contradicting to our (natural) assumption made in Lemma 4.1.

To give an example of the one-to-one correspondence shown in Lemma 4.1, we illustrate in Fig. 2 the edge-disjoint path decomposition of the flow network in Fig. 1 for cache size $w = 2$. If $\alpha < 1$ (in real systems, $\alpha$ is usually a very small quantity), then the 2-flow from **s** to **t** will be routed along the blue path (with cost $5+\alpha$) and the red path (with cost $6+\alpha$). Correspondingly, $a$ and $b$ will be stored into the two cache spaces on their first appearances in the request sequence. If $1 < \alpha < 2$, then the 2-flow from **s** to **t** will be routed along the blue path and the black direct path (with cost 7), which corresponds to the caching policy of using only one cache space to cache item $a$ on its first appearance.

**Lemma 4.2.** *In the one-to-one correspondence established in Lemma 4.1, let $P$ be any path from* **s** *to* **t** *with corresponding cache space $B$. Let $n_{write}$ be the number of cache writes occurs in $B$ (i.e. writing a new item into $B$) and let $n_{hit}$ be the number of cache hits occurs in $B$ (i.e. the new request item is already stored in $B$). The cost of path $P$ in the network is equal to*

$$\mathcal{C}(P) := \sum_{(u,v)\in P} a(u,v) = (T+1) + \alpha n_{write} - n_{hit}. \tag{1}$$

**Proof.** Let $\theta$ be a symbol not in the alphabet $\Sigma$ and let $\tau$ denote a wildcard character for $\Sigma$. For the convenience of argument, let us assign time stamp 0 to the source and time stamp $T+1$ to the sink (so $\mathbf{s} = (\theta, 0)$ and $\mathbf{t} = (\tau, T+1)$). Then we have the following invariant for the network

1. $G$ is a *layered* graph in the sense that every vertex in $G$ has a unique representation $(\sigma, t)$, where $\sigma$ denotes the symbol of the node and $t$ denotes layer at which the vertex lie, $0 \le t \le T+1$;
2. Every (directed) edge is from a node $(\sigma, t)$ to $(\sigma', t')$ with $t' > t$;
3. Every edge (except the direct edge from **s** to **t**) has capacity 1 and edge weight

$$a((\sigma,t),(\sigma',t')) = \begin{cases} t'-t+\alpha, & \text{if } \sigma \neq \sigma' \text{ and } \sigma' \neq \tau; \\ t'-t, & \text{if } \sigma \neq \sigma' \text{ and } \sigma' = \tau; \\ t'-t-1, & \text{if } \sigma = \sigma'. \end{cases}$$

[12] We thank an anonymous referee for pointing this out to us.

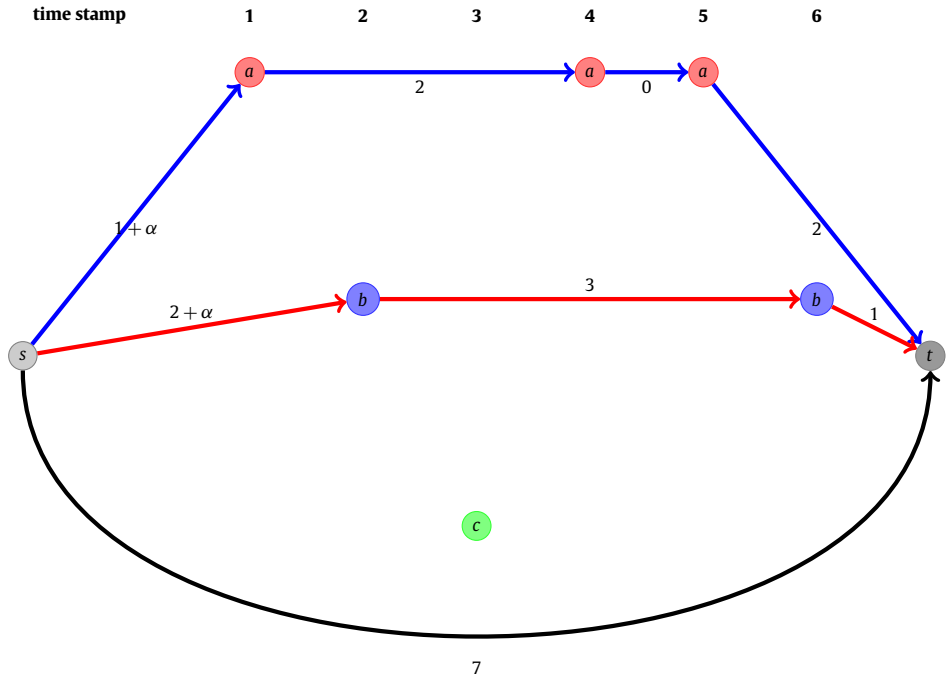


**Fig. 2.** An edge-disjoint path decomposition of the network shown in Fig. 1, when the cache size is 2.

That is, the edge weight of $((\sigma, t), (\sigma', t'))$ is, on the basis of level difference $t' - t$, add $\alpha$ if a cache write occurs, or subtract 1 if a cache hit occurs. Moreover, due to our construction of the network, if $((\sigma, t), (\sigma', t'))$ with $\sigma \neq \sigma'$, then there is no vertex $(\sigma, t'')$ with $t < t'' < t'$. In other words, no **s**-**t** path will miss to count a cache hit.

Therefore, any **s**-**t** path $P$ in $G$ must be a simple path, and the total weight of $P$ is

$$\mathcal{C}(P) = \sum_{(u,v)\in P} a(u, v) = (T + 1) + \alpha n_{\text{write}} - n_{\text{hit}}. \quad \square$$

Since the total flow is $w$, summing the total cost of all $w$ unit flows together gives that, for any feasible $b$ flow $f$ of the network $G$, there is a corresponding cache replacement policy $A$ using $w$ cache spaces of non-datapath cache, and vice versa, and the total cost of flow $f$ is related to the utility function of the corresponding cache replacement policy $A$ as

$$\mathcal{C}(f) = \sum_{\text{disjoint } \mathbf{s}\text{-}\mathbf{t} \text{ path } P} \mathcal{C}(P) = w(T + 1) - \mathscr{U}(A, \alpha, w, \mathcal{R}).$$

Let $f_{\text{opt}}$ be the minimum flow computed by the Edmond-Karp algorithm for the constructed network, it follows that

$$_{\text{m}}\text{OPT}(\alpha, w, \mathcal{R}) = w(T + 1) - \mathcal{C}(f_{\text{opt}}). \tag{2}$$

Furthermore, by outputting all the edges of the shortest path from **s** to **t** in the augmented network computed in the Edmond-Karp minimum cost flow algorithm in each iteration, we can, not only calculate the optimal value of the utility function, but also find the corresponding sequence of cached items in each cache space. This completes the proof of the Main Theorem.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Steven Lyons and Raju Rangaswami report financial support was provided by Florida International University. Steven Lyons and Raju Rangaswami report a relationship with NetApp and National Science Foundation (NSF) that includes: funding grants.

Ning Xie reports financial support was provided by Florida International University. Ning Xie reports a relationship with US Army Research Office that includes: funding grants.

## Acknowledgements

We are grateful to the anonymous referees for their detailed and invaluable comments and suggestions which greatly improve the presentation of the paper. Steven Lyons and Raju Rangaswami's research was supported in part by a NetApp Faculty Fellowship, and NSF grants CCF-1718335, CNS-1563883, and CNS-1956229. Ning Xie's research was partially supported by U.S. Army Research Office (ARO) under award number W911NF1910362.

## Appendix A. A comparison between MIN and $_{\mathrm{m}}$OPT

**Theorem A.1.** *For any cache size $w$ and request sequence $\mathcal{R}$, we have*

$$\mathrm{MIN}(w, \mathcal{R}) \leq {}_{\mathrm{m}}\mathrm{OPT}(0, w, \mathcal{R}) \leq \mathrm{MIN}(w+1, \mathcal{R}).$$

**Proof.** The first inequality is trivial since choosing not to write to the cache is a privilege: the non-datapath cache can definitely simulate the behavior of **MIN** algorithm and achieve the same number of hits. For the second inequality, one can imagine that one cache space of the $(w+1)$-sized traditional cache is a special "temporary storage", and the rest $w$ cache spaces are normal storage. It can simulate the cache behavior of a non-datapath $w$-sized cache as follows. If the fetched item is stored in the non-datapath cache, so does the traditional one, and the item is stored in the same place in the normal storage space. Otherwise, fetched item will be stored in the temporary storage. It follows that **MIN** algorithm with $(w+1)$-sized cache has at least the same number of hits as any cache management algorithm of the non-datapath cache for a $w$-sized cache. □

## References

[1] Ravindra K. Ahuja, Andrew V. Goldberg, James B. Orlin, Robert E. Tarjan, Finding minimum-cost flows by double scaling, Math. Program. 53 (1) (1992) 243–266.
[2] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice Hall, Upper Saddle River, New Jersey, 1993.
[3] Nirav Atre, Justine Sherry, Weina Wang, Daniel S. Berger, Caching with delayed hits, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, pp. 495–513.
[4] Laszlo A. Bélády, A study of replacement algorithms for a virtual-storage computer, IBM Syst. J. 5 (2) (1966) 78–101.
[5] Daniel S. Berger, Nathan Beckmann, Mor Harchol-Balter, Practical bounds on optimal caching with variable object sizes, Proc. ACM Meas. Anal. Comput. Syst. 2 (2) (2018) 1–38.
[6] Simona Boboila, Peter Desnoyers, Write endurance in flash drives: measurements and analysis, in: Proceedings of the 8th USENIX Conference on File and Storage Technologies, FAST'10, Berkeley, CA, USA, USENIX Association, 2010, p. 9.
[7] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, Kai Li, Erasing Belady's limitations: in search of flash cache offline optimality, in: 2016 USENIX Annual Technical Conference, ATC 16, USENIX, 2016, pp. 379–392.
[8] Jack Edmonds, Richard M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J. ACM 19 (2) (1972) 248–264.
[9] Lester Randolph Ford Jr., Delbert Ray Fulkerson, Flows in Networks, Princeton University Press, 1962.
[10] Michael L. Fredman, Robert Endre Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (3) (1987) 596–615.
[11] Andrew V. Goldberg, Robert E. Tarjan, Solving minimum-cost flow problems by successive approximation, Math. Oper. Res. 15 (3) (1990) 430–466.
[12] David A. Holland, Elaine Angelino, Gideon Wald, Margo I. Seltzer, Flash caching on the storage client, Presented as part of the 2013 USENIX Annual Technical Conference, USENIX ATC 13, San Jose, CA, USENIX, 2013, pp. 127–138.
[13] Dongwoo Kang, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh Choi, Onur Mutlu, Amnesic cache management for non-volatile memory, in: 2015 31st Symposium on Mass Storage Systems and Technologies, MSST, May 2015, pp. 1–13.
[14] Bernhard Korte, Jens Vygen, Combinatorial Optimization: Theory and Algorithms, 6th edition, Springer, 2018.
[15] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, Irving L. Traiger, Evaluation techniques for storage hierarchies, IBM Syst. J. 9 (2) (1970) 78–117.
[16] James B. Orlin, A faster strongly polynomial minimum cost flow algorithm, Oper. Res. 41 (2) (1993) 338–350.
[17] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, Jason Liu, To ARC or not to ARC, in: Proc. of USENIX HotStorage, 2015.
[18] Benjamin Van Roy, A short proof of optimality for the MIN cache replacement algorithm, Inf. Process. Lett. 102 (2–3) (2007) 72–73.
[19] David P. Williamson, Network Flow Algorithms, Cambridge University Press, 2019.
[20] Lei Zhang, Reza Karimi, Irfan Ahmad, Ymir Vigfusson, Optimal data placement for heterogeneous cache, memory, and storage systems, Proc. ACM Meas. Anal. Comput. Syst. 4 (1) (2020) 1–27.