Online Tree Caching[*]

Marcin Bienkowski
Institute of Computer Science, University of Wrocław, Poland

Jan Marcinkowski
Institute of Computer Science, University of Wrocław, Poland

Maciej Pacut
Institute of Computer Science, University of Wrocław, Poland

Stefan Schmid
Department of Computer Science, Aalborg University, Denmark

Aleksandra Spyra
Institute of Computer Science, University of Wrocław, Poland

ABSTRACT

We initiate the study of a natural and practically relevant new variant of online caching with bypassing where the to-be-cached items can have dependencies. We assume that the universe is a tree T and items are tree nodes; we require that if a node v is cached then the whole subtree $T(v)$ rooted at v is cached as well. This theoretical problem finds an immediate application in the context of forwarding table optimization in IP routing and software-defined networks.

We present an elegant online deterministic algorithm TC for this problem, and rigorously prove that its competitive ratio is $O(\text{HEIGHT}(T) \cdot k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1))$, where k_{ONL} and k_{OPT} denote the cache sizes of an online and the optimal offline algorithm, respectively. The result is optimal up to a factor of $O(\text{HEIGHT}(T))$.

CCS Concepts

•**Theory of computation** → **Online algorithms;** *Caching and paging algorithms;* •**Networks** → *Programmable networks;* Packet-switching networks;

Keywords

online algorithms; competitive analysis; caching; routers; software defined networking; forwarding information base

[*]Research partially supported by Polish National Science Centre grant DEC-2013/09/B/ST6/01538 as well as Aalborg University's talent management program.

© 2017 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

In the classic online paging problem, items of some universe are requested by a processing entity (e.g., blocks of RAM are requested by the processor). To speed up the access, computers use a faster memory, called *cache*, capable of accommodating k such items. Upon a request to a non-cached item, the algorithm has to fetch it into the cache, paying a fixed cost, while a request to a cached item is free. If the cache is full, the algorithm has to free some space by evicting an arbitrary subset of items from the cache.

The paging problem is inherently online: the algorithm has to make decisions what to evict from the cache without the knowledge of future requests; its cost is compared to the cost of the optimal *offline* solution and the ratio of these two amounts is called *competitive ratio*. The first analysis of this basic problem in an online model was given over three decades ago by Sleator and Tarjan [28]. The problem was later considered in a variety of flavors. In particular, some papers considered a *bypassing model* [11, 15], where item fetching is optional: the requested item can be served without being in the cache, for another fixed cost (usually being at most the cost of item fetching).

In this paper, we introduce a natural extension of this fundamental problem, where items have inter-dependencies. More precisely, we assume that the universe is an arbitrary (not necessarily binary) rooted tree T and the requested items are its nodes. For any tree node v, $T(v) \subseteq T$ is a subtree rooted at v containing v and all its descendants. We require the following property: if a node v is in the cache, then all nodes of $T(v)$ are also cached. In other words, we require that **the cache is a subforest of** T, i.e., a union of disjoint subtrees of T. We call this problem *online tree caching.*

Furthermore, we assume a bypassing model and distinguish between two types of requests: a request can be either *positive* or *negative*. The positive requests correspond to "normal" requests known from caching problems: we pay 1 if the node is not cached; for a negative request, we pay 1 if the corresponding request is cached. After serving the request, we may reorganize our cache arbitrarily, but the resulting cache has to still be a subforest of T. We pay α for fetching or evicting any single node, where $\alpha \geq 1$ is an integer and a parameter of the problem. Our goal is to minimize the overall cost of maintaining the cache and serving the requests.

One interesting application for our model arises in the context of modern IP routers which need to store a rapidly increasing number of forwarding rules [1, 9]. In Section 2, we give a glimpse of this application, discussing how tree caching algorithms can be applied in existing systems to effectively reduce the memory requirements on IP routers.

1.1 Our Contributions

We initiate the study of a natural new caching with bypassing problem which allows to account for tree-dependencies among items. The problem finds immediate applications, e.g., in IP routing and software-defined networking.

In particular, we consider the online tree caching problem within the resource augmentation paradigm: we assume that cache sizes of the online algorithm (k_{ONL}) and the optimal offline algorithm (k_{OPT}) may differ. We assume $k_{\text{ONL}} \geq k_{\text{OPT}}$ and let $R = k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1)$.

We present an elegant deterministic online algorithm TC for this problem. While our algorithm is simple, the analysis requires several non-trivial insights into the problem. In particular, we rigorously prove that TC is $O(h(T) \cdot R)$-competitive, where $h(T)$ is the height of tree T. That is, we show that there exists a constant β, such that $\text{TC}(I) \leq O(h(T) \cdot R) \cdot \text{OPT}(I) + \beta$ for any input I. Note that this result is optimal up to the factor $O(h(T))$: in Appendix C, we show that the lower bound R for the paging problem [28] implies an $\Omega(R)$ lower bound for our problem for any $\alpha \geq 1$.

1.2 Related Work on Caching

Our formal model is a novel variant of competitive paging, a classic online problem. In the framework of the competitive analysis, the paging problem was first analyzed by Sleator and Tarjan [28], who showed that algorithms Least-Recently-Used, First-In-First-Out and Flush-When-Full are $k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1)$-competitive and no deterministic algorithm can beat this ratio. In the non-augmented case when $k_{\text{ONL}} = k_{\text{OPT}} = k$, the competitive ratio is simply k.

The simple paging problem was later generalized to allow different fetching costs (weighted paging) [8, 32] and additionally different item sizes (file caching) [33], with the same competitive ratio. Asymptotically same results can be achieved when bypassing is allowed (see [11, 15] and references therein). With randomization, the competitive ratio can be reduced to $O(\log k)$ even for file caching [3]. The lower bound for randomized algorithms is $H_k = \Theta(\log k)$ [12] and is matched by known paging algorithms [2, 24].

To the best of our knowledge, the variant of caching, where fetching items to the cache is not allowed unless some other items are cached (as it is the case for the restrictions induced by the underlying tree in this paper) was not considered previously in the framework of competitive analysis. Note that there is a seemingly related problem called restricted caching [6] (there are also its variants called matroid caching [7] or companion caching [25]). Despite naming similarities, the restricted caching model is completely different from ours: there the restriction is that each item can be placed only in a restricted set of cache locations. Hence, even the algorithmic ideas developed in [6, 7, 25] are not applicable in our scenario.

1.3 Paper Organization

The remainder of this paper is organized as follows. Section 2 sketches a practical motivation and Section 3 introduces the preliminaries. We present our algorithm in Section 4 and we rigorously analyze its competitve ratio in Section 5. We reason about implementation aspects in Section 6 and conclude in Section 7. Due to space constraints and for ease of presentation, additional technical details are postponed to the Appendix. The Appendix also includes a lower bound as well as a polynomial-time algorithm that solves the static problem variant.

2. APPLICATION: MINIMIZING FORWARDING TABLES IN ROUTERS

Dependencies among to-be-cached items arise in numerous settings and are a natural refinement of many caching problems. To give a concrete example, one important motivation for our tree-based dependency model arises in the context of IP routers. In particular, the online tree caching problem we introduce in this paper is motivated by router memory constraints in IP-based networks. The material presented in this section serves for motivation, and is not necessary for understanding the remainder of the paper.

Nowadays, routers have to store an enormous number of forwarding rules: the number of rules has doubled in the last six years [1] and the superlinear growth is likely to be sustained [9]. This entails large costs for Internet Service Providers: fast router memory (usually Ternary Content Addressable Memory (TCAM)) is expensive and power-hungry [29]. Many routers currently either operate at (or beyond) the edge of their memory capacities. A solution, which could delay the need for expensive or impossible memory upgrades in routers, is to store only a subset of rules in the actual router and store all rules on a secondary device (for example a commodity server with a large but slow memory) [17, 18, 19, 20, 27].

This solution is particularly attractive with the advent of Software-Defined Network (SDN) technology,

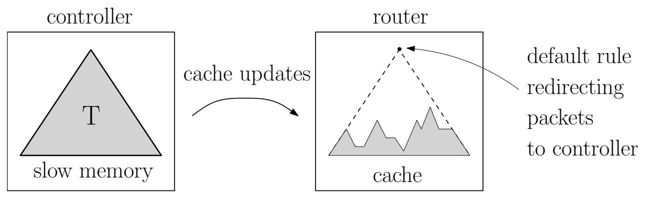

Figure 1: The router (*right*) caches only a subset of all rules, and rules that are not cached are answered by the controller (*left*) that keeps the whole tree of rules. Updates to the rules are passed by the controller to the router.

which allows to manage the expensive memory using a software controller [17, 27]. In particular, our theoretical model can describe real-world architectures like [17, 27], that is, our model formalizes the underlying operational problems of such architectures. Our algorithm, when applied in the context of such architectures, can hence be used to prolong the lifetime of IP routers.

Setup, positive requests, fetches and evictions. The setup (see [27] for a more technical discussion) depicted in Figure 1 consists of two entities: the actual router (e.g., an OpenFlow switch) which caches only a subset of all forwarding rules, and the (SDN) controller, which keeps all rules in its less expensive and slower memory. During runtime, packets arrive at the router, and if an appropriate forwarding rule is found within the rules cached by the router, then the packet is forwarded accordingly, and the associated cost is zero. Otherwise, the packet has to be forwarded to the controller (where an appropriate forwarding rule exists); this indirection costs 1. Hence, the rules correspond to cacheable items and accesses to rules are modelled by positive requests to the corresponding items. At some chosen points in time, the caching algorithm run at the controller may decide to remove or add rules to the cache. Any such change entails a fixed cost α.[1]

Tree dependencies. Note that the technical feasibility of this solution heavily depends on the rule dependencies. In the most ubiquitous scenario, the rules are prefixes of IP addresses (they are bit strings). Whenever a packet arrives, the router follows a longest matching prefix (LMP) scheme: it searches for the rule that is a prefix of the destination IP of the packet and among matching rules it chooses the longest one. In other words, if the prefixes corresponding to rules are stored in the tree[2], then the tree is traversed from the root downwards, and the last found rule is used. This explains why we require the cached nodes to form a subforest: leaving a less specific rule on the router while evicting a more specific one (i.e., keeping a tree node in cache while evicting its descendant) will result in a situation where packets will be forwarded according to the less specific rule, and hence potentially exit through the wrong port. The LMP scheme also ensures that the described approach is implementable: one could simply add an artificial rule at the tree root in the router (matching an empty prefix). This ensures that when no actual matching rule is found in the router (in the cache), the packet will be forwarded according to this artificial rule to the controller that stores all the rules and can handle all packets appropriately.

So far, the papers on IP rule caching avoided dependencies either assuming that rules do not overlap (a tree has a single level) or by changing the tree on the fly, so that the rules become non-overlapping [18, 19, 20]. Unfortunately, this could lead to a large inflation of the routing table. A notable exception is a recent solution called CacheFlow [17]. The CacheFlow model supports dependencies even in the form of directed acyclic graphs. However, CacheFlow was evaluated only experimentally, and no worst-case guarantees were given on the overall cost of caching. Our work provides theoretical foundations for respecting tree dependencies.

Negative requests. Additionally, a rule may need to be updated. For example, due to a change communicated by a dynamic routing protocol (e.g., BGP) the action defined by a rule has to be modified. In either case, we have to update the rules at the controller: we assume that this cost is zero. (This cost is unavoidable for any algorithm, so such an assumption makes our problem only more difficult.) Furthermore, if the rule is also stored at the router, then we have to pay a fixed cost of α for updating the router (see the remark for the cost of fetches and evictions). Such penalties can be easily simulated in our model: we issue a sequence of α negative requests to the updated node. It is straightforward to show that the costs in these two models can differ by a factor of at most 2. For a formal argument, see Appendix B.

Implementability. Note that the whole input (fed to a tree caching algorithm) is created at the controller: positive requests are caused by cache misses (which redirect packet to the controller) and batches of α negative requests are caused by updates sent to the dynamic routing algorithm run at the controller. Therefore, the whole tree caching algorithm can be implemented in software in the controller only. Furthermore, our algorithm is a simple counter-based scheme, which can be implemented efficiently and also fine-tuned for speed, see Section 6.

Other work on forwarding table minimization. Other approaches for minimizing the number of stored

[1]This cost corresponds to the transmission of a message from the controller to the router as well as the update of internal data structures of the router. Such an update of proprietary and vendor-dependent structures can be quite costly [14], but the empirical studies show it to be independent of the rule being updated [13].

[2]We do not have to assume that they are actually stored in a real tree; this tree is implicit in the LMP scheme.

rules were mostly based on *rules compression (aggregation)*, where the set of rules was replaced by another equivalent and smaller set. Optimal aggregation of a fixed routing table can be achieved by dynamic programming [10, 30], but the main challenge lies in balancing the achieved compression and the amount of changes to the routing table in the presence of *updates* to this table. While many practical heuristics have been devised by the networking community for this problem [16, 21, 22, 23, 26, 31, 34], worst-case analyses were presented only for some restricted scenarios [4, 5]. Combining rules compression and rules caching is so far an unexplored area.

3. PRELIMINARIES

We denote the height of T by $h(T)$. For any node v, $T(v)$ denotes the subtree of T rooted at v (containing v and all its descendants). A *tree cap* rooted at v is "an upper part" of $T(v)$, i.e., it contains v and if it contains node u, then it also contains all nodes on the path from u to v. If $A \subseteq B$ are both tree caps rooted at v, then we say that A is a tree cap of B.

We assume discrete time slotted into rounds, with round $t \geq 1$ corresponding to time interval $(t-1, t)$. In round t, the algorithm is given one (positive or negative) request to exactly one tree node and has to process it, i.e., pay associated costs (if any). Right after round t, at time t, the algorithm may arbitrarily reorganize its cache, (i) ensuring that the resulting cache is a subforest of T (i.e., if the cache contains node v, then it contains the entire $T(v)$) and (ii) preserving the cache capacity constraint. An algorithm pays α for a single node fetch or eviction. We denote the contents of the cache at round t by C_t. (As the cache changes contents only between rounds, C_t is well defined.) We assume that α is an even integer (this assumption may change costs at most by a constant factor). We assume that the algorithm starts with an empty cache.

We call a non-empty set X a *valid positive changeset* for cache C if $X \cap C = \emptyset$ and $C \cup X$ is a subforest of T, and a *valid negative changeset* if $X \subseteq C$ and $C \setminus X$ is a subforest of T. We call X a *valid changeset* if it is either valid positive or negative changeset. Note that the union of positive (negative) changesets is also a valid positive (negative) changeset. We say that the algorithm applies changeset X, if it fetches all nodes from X (for a positive changeset) and evicts all nodes from X (for a negative one). Note that not all valid changesets may be applied as the algorithm is also limited by its cache capacity (k_{ONL} for an online algorithm and k_{OPT} for the optimal offline one).

4. ALGORITHM

The algorithm TREE CACHING (TC) presented in the following is simple. It operates in multiple phases. The first phase starts at time 0. TC starts each phase with an empty cache and proceeds as follows. Within a phase, every node keeps a counter, which is initially zero. If at round t it pays 1 for serving the request, it increments its counter. Whenever a node is fetched or evicted from the cache, its counter is reset to zero. Note that this implies that the counter of v is equal to the number of negative (positive) requests to v since its last fetching to the cache (eviction from the cache). For a set $A \subseteq T$, we denote the sum of all counters in A at time t by $\text{cnt}_t(A)$. At time t, TC verifies whether there exists a valid changeset X, such that

- *(saturation property)* $\text{cnt}_t(X) \geq |X| \cdot \alpha$ and
- *(maximality property)* $\text{cnt}_t(Y) < |Y| \cdot \alpha$ for any valid changeset $Y \supsetneq X$.

In this case, the algorithm modifies its cache applying X.

If, at time t, TC is supposed to fetch some set X, but by doing so it would exceed the cache capacity k_{ONL}, it evicts all nodes from the cache instead, and starts a new phase at time t. Such a *final eviction* might not be present in the last phase, in which case we call it *unfinished*.

In Lemma 1 (below), we show that at any time, all valid changesets satisfying both properties of TC are either all positive or all negative. Furthermore, right after the algorithm applies a changeset, no valid changeset satisfies saturation property.

Section 6 shows that TC can be implemented efficiently.

5. ANALYSIS OF TC

Throughout the paper, we fix an input I, its partition into phases, and analyze both TC and OPT on a single fixed phase P. We denote the times at which P starts and ends by $\text{begin}(P)$ and $\text{end}(P)$, respectively, i.e., rounds in P are numbered from $\text{begin}(P) + 1$ to $\text{end}(P)$. A proof of the following technical lemma follows by induction and is presented in Appendix A.

LEMMA 1. *Fix any time $t > \text{begin}(P)$. For any valid changeset X for C_t, it holds that $\text{cnt}_t(X) \leq |X| \cdot \alpha$. If a changeset X is applied at time t, the following properties hold:*

1. *X contains the node requested at round t,*
2. *$\text{cnt}_t(X) = |X| \cdot \alpha$,*
3. *$\text{cnt}_t(Y) < |Y| \cdot \alpha$ for any valid changeset Y for C_{t+1} (note that C_{t+1} is the cache state right after application of X),*
4. *X is a tree cap of a tree from C_{t+1} if X is positive and it is a tree cap of a tree from C_t if X is negative.*

In the following, we assume that no positive requests are given to nodes inside cache and no negative ones to nodes outside of it. (This does not change the behavior of TC and can only decrease the cost of OPT.)

For the sake of analysis, we assume that at time end(P), TC actually performs a cache fetch (exceeding the cache size limit) and then, at the same time instant, empties the cache. This replacement only increases the cost of TC. Let k_P denote the number of nodes in the cache of TC at end(P). In a finished phase, we measure it after the artificial fetch, but right before the final eviction, and thus $k_P \geq k_{\text{ONL}} + 1$; in an unfinished phase $k_P \leq k_{\text{ONL}}$.

The crucial part of our analysis that culminates in Section 5.2 is the technique of shifting requests. Namely, we modify the input sequence by shifting requests up or down the tree, so that the resulting input sequence (i) is not harder for OPT and (ii) is more structured: we may lower bound the cost of OPT on each node separately and relate it to the cost of TC.

5.1 Event Space and Fields

In our analysis, we look at a two-dimensional discrete, spatial-temporal space, called the *event space*. The first dimension is indexed by tree nodes, whose order is an arbitrary extension of the partial order given by the tree. That is, the parent of a node v is always "above" v. The second dimension is indexed by round numbers of phase P. The space elements are called *slots*. Some slots are occupied by requests: a request at node v given at round t occupies slot (v, t). From now on, we will identify P with a set of requests occupying some slots in the event space.

We partition slots of the whole event space into disjoint parts, called *fields*, and we show how this partition is related to the costs of TC and OPT. For any node v and time t, $\text{last}_v(t)$ denotes the last time strictly before t, when node v changed state from cached to non-cached or vice versa; $\text{last}_v(t) = \text{begin}(P)$ if v did not change its state before t in phase P. For a changeset X_t applied by TC at time t, we define the field F^t as

$$F^t = \{\ (v, r) : v \in X_t \ \wedge\ \text{last}_v(t) + 1 \leq r \leq t\ \}\quad .$$

That is, field F^t contains all the requests that eventually trigger the application of X_t at time t. We say that F^t ends at t. We call field F^t *positive* (*negative*) if X_t is a positive (negative) changeset. An example of a partitioning into fields is given in Figure 2. We define $\text{req}(F^t)$ as the number of requests belonging to slots of F^t and let $\text{size}(F^t)$ be the number of involved nodes (note that $\text{size}(F^t) = |X_t|$). The observation below follows immediately by Lemma 1.

OBSERVATION 2. *For any field F, req(F) = size(F) $\cdot$ α. All these requests are positive (negative) if F is positive (negative).*

Finally, we call the rest of the event space defined by phase P *open field* and denote it by F^∞. The set of all fields except F^∞ is denoted by $\mathcal{F}$. Let $\text{size}(\mathcal{F}) = \sum_{F \in \mathcal{F}} \text{size}(F)$.

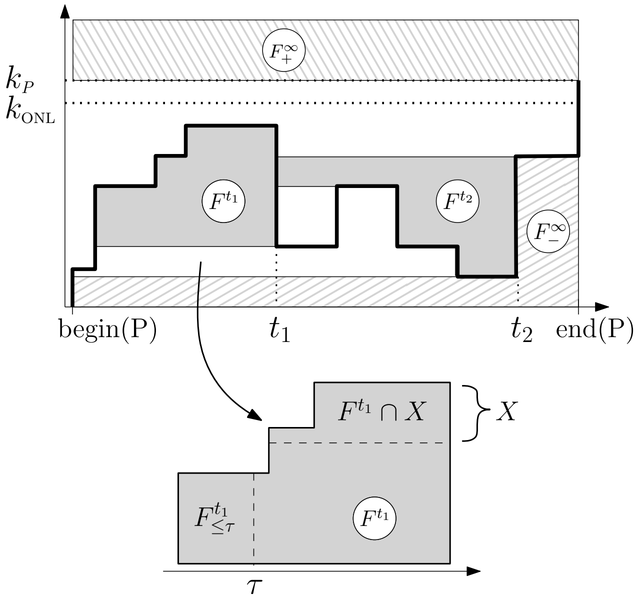

Figure 2: Partitioning of a single phase into fields for a line (a tree with no branches). The thick line represents cache contents. Possible final eviction at end(P) is not depicted. F^{t_1} is a negative field and F^{t_2} is a positive one. In the particular depicted example, nodes are ordered from the leaf (bottom) to the root (top of the picture). We emphasize that for a general, branched tree, some notions (in particular fields) no longer have nice geometric interpretations.

LEMMA 3. *For any phase P partitioned into a set of fields $\mathcal{F} \cup \{F^\infty\}$, it holds that* $\text{TC}(P) \leq 2\alpha \cdot \textit{size}(\mathcal{F}) + \textit{req}(F^\infty) + k_P \cdot \alpha$.

PROOF. By Observation 2, the cost associated with serving the requests from all fields from $\mathcal{F}$ is $\sum_{F \in \mathcal{F}} \alpha \cdot \text{size}(F) = \alpha \cdot \text{size}(\mathcal{F})$. The cost of the cache reorganization at the fields' ends is exactly the same. The term $\text{req}(F^\infty)$ represents the cost of serving the requests from F^∞ and $k_P \cdot \alpha$ upper-bounds the cost of the final eviction (not present in an unfinished phase). □

5.2 Shifting Requests

The actual challenge in the proof is to relate the structure of the fields to the cost of OPT. The rationale behind our construction is based on the following thought experiment. Assume that the phase is unfinished (for example, when the cache is so large that the whole input corresponds to a single phase). Recall that the number of requests in each field $F \in \mathcal{F}$ is equal to $\text{size}(F) \cdot \alpha$. Assume that these requests are evenly distributed among the nodes of F (each node from F receives α requests in the slots of F). Then, the history of any node v is alternating between periods spent in positive fields and periods spent in negative fields. By our even distribution assumption, each such a period contains exactly α requests. Hence, for any two consecutive periods of a single node, Opt has to pay at least α (either α for positive requests or α for negative ones, or α for changing the cached/non-cached state of v). Essentially, this shows that Opt has to pay an amount that can be easily related to $\alpha \cdot \text{size}(\mathcal{F})$.

Unfortunately, the requests may not be evenly distributed among the nodes. To alleviate this problem, we will modify the requests in phase P, so that the newly created phase P' is not harder for Opt and will "almost" have the even distribution property. In this construction, the time frame of P and its fields are fixed.

5.2.1 Legal Shifts

We say that a request placed originally (in phase P) at slot (v, t) is *legally shifted* if its new slot is $(m(v), t)$, where (i) for a positive request, $m(v)$ is either equal to v or is one of its descendants and (ii) for a negative request, $m(v)$ is either equal to v or is one of its ancestors. For any fixed sequence of fetches and evictions within phase P, the associated cost may only decrease when these actions are replayed on the modified requests.

Observation 4. *If P' is created from P by legally shifting the requests, then* $\text{Opt}(P') \le \text{Opt}(P)$.

The main difficulty is however in keeping the legally shifted requests within the field they originally belonged to. For example, a negative request from F shifted at round t from node u to its parent may fall out of F as the parent may still be outside the cache at round t. In effect, a careless shifting of requests may lead to a situation where, for a single node v, requests do not create interleaved periods of positive and negative requests, and hence we cannot argue that $\text{Opt}(P')$ is sufficiently large.

In the following subsections, we show that it is possible to legally shift the requests of any field $F \in \mathcal{F}$, so that they remain within F, and they will be either exactly or approximately evenly distributed among nodes of F. This will create P' with appropriately large cost for Opt.

5.2.2 Notation

We start with some general definitions and remarks. For any field F and set of nodes A, let $F \cap A = \{(v,t) \in F : v \in A\}$. Analogously, if L is a set of rounds, then let $F \cap L = \{(v,t) \in F : t \in L\}$. For any field F^t and time τ, we define

$$F^t_{\le \tau} = F^t \cap \{t' : t' \le \tau\} \ .$$

It is convenient to think that F^t evolves with time and $F^t_{\le \tau}$ is the snapshot of F^t at time τ. Note that F^t may have some nodes not included in $F^t_{\le \tau}$. These notions are depicted in Figure 2.

We may extend the notions of req and size to arbitrary subsets of fields in a natural way. For any subset $S \subseteq F$, we call it *over-requested* if $\text{req}(S) > \text{size}(S) \cdot \alpha$.

Lemma 5. *Fix any field F^t, the corresponding changeset X_t, and any time τ.*

1. *If F^t is negative, then for any tree cap D of X_t, the set $F^t_{\le \tau} \cap D$ is not over-requested.*
2. *If F^t is positive, then for any subtree $T' \subseteq T$, the set $F^t_{\le \tau} \cap T'$ is not over-requested.*

Proof. As the nodes from $F^t_{\le \tau} \cap D$ form a valid changeset at time τ, Lemma 1 implies $\text{req}(F^t_{\le \tau} \cap D) = \text{cnt}_\tau(F^t_{\le \tau} \cap D) \le |F^t_{\le \tau} \cap D| \cdot \alpha$.

The proof of the second property is identical: As $F^t_{\le \tau} \cap T'$ is also a valid changeset at time τ, by Lemma 1, $\text{req}(F^t_{\le \tau} \cap T') = \text{cnt}_\tau(F^t_{\le \tau} \cap T') \le |F^t_{\le \tau} \cap T'| \cdot \alpha$. □

By Lemma 5 applied at $\tau = t$ and Observation 2, we deduct the following corollary.

Corollary 6. *Fix any field F^t, the corresponding changeset X_t and any tree cap D of X_t.*

1. *If F^t is positive, then $\text{req}(F^t \cap D) \ge \alpha \cdot |D|$.*
2. *If F^t is negative, then $\text{req}(F^t \cap (X_t \setminus D)) \ge \alpha \cdot |X_t \setminus D|$.*

Informally speaking, the corollary above states that the average amount of requests in a positive field is *at least as large at the top of the field as at its bottom.* For a negative field this relation is reversed.

5.2.3 Shifting Negative Requests Up

Fix any valid negative changeset X_t applied at time t and the corresponding field F^t. We call a tree cap $Y \subseteq X_t$ *proper* if

1. $\text{req}(F^t \cap Y) = |Y| \cdot \alpha$ and
2. $F^t_{\le \tau} \cap D$ is not over-requested for any tree cap $D \subseteq Y$ and any time $\tau \le t$.

The first property of Lemma 5 states that before we shift the requests of F_t, the set X_t is proper. We start with $Y = X_t$, and proceed in a bottom-up fashion, inductively using the lemma below. We take care of a single node of Y at a time and ensure that after the shift the number of requests at this node is exactly α and the remaining part of Y remains proper.

Lemma 7. *Given a negative field F^t, the corresponding changeset X_t and a proper tree cap $Y \subseteq X_t$, it is possible to choose a leaf v and legally shift some requests inside Y, so that in result $\text{req}(v) = \alpha$ and $Y \setminus \{v\}$ is proper.*

Proof. As $\text{req}(F^t \cap Y) = |Y| \cdot \alpha$, Corollary 6 implies that any leaf of Y was requested at least α times inside F^t. We pick an arbitrary leaf v, and let $r \ge \alpha$ be the number of requests to v in F^t.

We look at all the requests to v in F^t ordered by their round. Let s be the round when $(\alpha+1)$-th of them arrives. We will now show that at round s, TC already has $p(v)$ in its cache. If it had not, $\{v\}$ would be a tree cap of $F^t_{\le s}$, and by the first property of Lemma 5, it would contain at most α requests, which is a contradiction. Hence, if we shift the chronologically last $r - \alpha$ requests from v to $p(v)$, these requests stay within F^t.

It remains to show that $Y \setminus \{v\}$ is proper after such a shift. We choose any tree cap $D \subseteq Y$ and any time $\tau \leq t$. If D does not contain $p(v)$ or $\tau < s$, then the number of requests in $F^t_{\leq \tau} \cap D$ was not changed by the shift, and hence $F^t_{\leq \tau} \cap D$ is not over-requested. Otherwise, $D \cup \{v\}$ was a tree cap in Y and by the lemma assumption, $F^t_{\leq \tau} \cap (D \cup \{v\})$ was not over-requested. As $F^t_{\leq \tau} \cap D$ has now exactly α less requests than $F^t_{\leq \tau} \cap (D \cup \{v\})$ had, it is not over-requested, either. □

Corollary 8. *For any negative field F^t, it is possible to legally shift its requests up, so that they remain within F^t and after the modification each node is requested exactly α times.*

5.2.4 Shifting Positive Requests Down

We will now focus on the problem of shifting the positive requests down in a single positive field F^t, corresponding to a single fetch of TC at the time t. Our goal is to devise a shifting strategy, that will result in at least $\Omega(\text{size}(F^t)/h(T))$ nodes having $\alpha/2$ requests each. While this result may be suboptimal, deriving a shifting strategy for a positive field that would have the same equal distribution guarantee as the one provided by Corollary 8 is not possible (see Appendix C).

First, we prove that from any node v in the field, we can shift down a constant fraction of its requests within the field, distributing them to different nodes.

Lemma 9. *Let F^t be a positive field and let X_t be the corresponding changeset fetched to the cache at time t. Fix any node $v \in X_t$ that has been requested at least $c \cdot (\alpha/2)$ times in F^t, where c is an integer. It is possible to shift down its requests to the nodes of $T(v) \cap X_t$, so that these requests remain inside F^t and $\lceil c/2 \rceil$ nodes of $T(v)$ get $\alpha/2$ requests each.*

Proof. We order the nodes $u_1, u_2, \ldots u_{|T(v) \cap X_t|}$ of $T(v) \cap X_t$, so that $\text{last}_{u_i}(t) \leq \text{last}_{u_{i+1}}(t)$ for all i. In case of a tie, we place nodes that are closer to v first. Note that this linear ordering is an extension of the partial order defined by the tree: the parent of a node cannot be evicted later than the node itself (otherwise the cache would cease to be a subforest of T). In particular, it holds that $u_1 = v$.

We number $c \cdot (\alpha/2)$ requests to v chronologically, starting from 1. For any $j \in \{1, \ldots, \lceil c/2 \rceil\}$ we look at round τ_j with the $((j-1) \cdot \alpha + 1)$-th request to v. When this request arrives, node u_j is already present in the cache. Otherwise, we would have at least $j \cdot \alpha + 1$ requests in $F^t_{\leq \tau_j} \cap \{u_1, \ldots, u_j\}$ (already in $F^t_{\leq \tau_j} \cap \{u_1\}$ alone), which would make it over-requested, and thus contradict the second property of Lemma 5. Hence, we may take requests numbered from $(j-1) \cdot \alpha + 1$ to $(j-1) \cdot \alpha + \alpha/2$, shift them down from v to u_j, and after such a modification these requests are still inside F^t. Note that for $j = 1$ requests are not really shifted, as u_1 is v itself. We perform such a shift for any $j \in \{1, \ldots, \lceil c/2 \rceil\}$, which yields the lemma. □

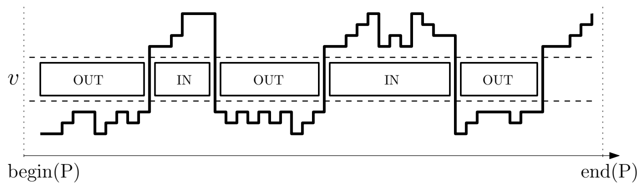

Figure 3: Partitioning of the phase into interleaving IN and OUT periods for node v. The thick line represents cache contents. The *leftover* OUT period is present for node v as it has finished phase P inside TC's cache. The periods can be followed by requests contained in F^∞.

Lemma 10. *For any positive field F^t, it is possible to legally shift its requests down, so that they remain within F^t and after the modification at least size$(F^t)/(2h(T))$ nodes in F^t have at least $\alpha/2$ requests each.*

Proof. Let X_t be the changeset corresponding to field F^t, fetched to the cache at time t. By Observation 2, $\text{req}(F^t) = |X_t| \cdot \alpha$. We gather the requests at every node into groups of $\alpha/2$ consecutive requests. In every node at most $\alpha/2$ requests remain not grouped. Let $\overline{\text{req}}(X)$ denote the number of grouped requests in the set X. Clearly, $\overline{\text{req}}(F^t) \geq |X_t| \cdot \alpha/2$, i.e., there are at least $|X_t|$ groups of requests in set X_t.

Let $X_t = X_t^1 \sqcup X_t^2 \sqcup \cdots \sqcup X_t^{h(T)}$ be a partition of the nodes of the tree X_t into layers according to their distance to the root. By the pigeonhole principle, there is a layer X_t^i containing at least $\lceil |X_t|/h(T) \rceil$ groups of requests (each group has $\alpha/2$ requests).

Nodes of X_t^i are independent, i.e., for $u, v \in X_t^i$ the trees $T(u)$ and $T(v)$ are disjoint. Therefore, we may use the shifting strategy described in Lemma 9 for each node of X_t^i separately. After such modification, at least $\lceil |X_t|/(2h(T)) \rceil \geq \text{size}(F_t)/(2h(T))$ nodes have at least $\alpha/2$ requests each. □

5.2.5 Using Request Shifting for Bounding OPT

Finally, we may use our request shifting to relate $\text{size}(\mathcal{F}) = \sum_{F \in \mathcal{F}} \text{size}(F)$ to the cost of Opt in a single phase P. Recall that k_P denotes the size of TC's cache at the end of P. We assume that Opt may start the phase with an arbitrary state of the cache.

Lemma 11. *For any phase P, it holds that* $\text{Opt}(P) \geq$ *$(size(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2$.*

Proof. We transform P using legal shifts described in Section 5.2.3 and Section 5.2.4. That is, we create a corresponding phase P' that satisfies both Corollary 8 and Lemma 10. By Observation 4, it is sufficient to show that $\text{Opt}(P') \geq (\text{size}(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2$.

We focus on a single node v. We cut its history into interleaved periods: OUT *periods*, when v is outside the cache and receives positive requests, and IN *periods* when TC keeps v in the cache and v receives negative requests. A final (possibly empty) part corresponding to the time when v is in the F^∞ field is not accounted in OUT or IN periods, i.e., each IN or OUT period corresponds to some field $F \in \mathcal{F}$. Let p^{IN} and p^{OUT} denote the total number of IN and OUT periods (respectively) for all nodes during the phase. An example is given in Figure 3.

Recall that TC starts each phase with an empty cache, and hence each node starts with an OUT period. For k_P nodes that are in TC's cache at the end of the phase (and only for them) their history ends with an OUT period not followed by an IN period. We call them *leftover periods*. Thus, $p^{\text{OUT}} = p^{\text{IN}} + k_P$. The total number of periods $(p^{\text{IN}} + p^{\text{OUT}})$ is equal to the total size of all *fields*, $\text{size}(\mathcal{F})$, and thus $p^{\text{OUT}} = (\text{size}(\mathcal{F}) + k_P)/2 \geq \text{size}(\mathcal{F})/2$.

We call a period *full* if it has at least $\alpha/2$ requests. The shifting strategies described in the previous section ensure that all IN periods are full and at least $1/(2h(T))$ of all OUT periods are full. Thus, there are at least $p^{\text{OUT}}/(2h(T)) - k_P$ full OUT non-leftover periods; each of them together with the following IN period constitutes a *full* OUT-IN *pair*.

OPT has to pay at least $\alpha/2$ for the node in the course of the history described by a full OUT-IN pair: it pays α either for changing the cached/non-cached state of a node, or $\alpha/2$ for all positive requests or $\alpha/2$ for all negative ones. Thus, $\text{OPT}(P') \geq (p^{\text{OUT}}/(2h(T)) - k_P) \cdot \alpha/2 \geq (\text{size}(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2$. □

5.3 Competitive Ratio

To relate the cost of OPT to TC in a single phase P, we still need to upper-bound $\text{req}(F^\infty)$ and relate $k_P \cdot \alpha$ to the cost of OPT (cf. bounds on TC and OPT provided by Lemma 3 and Lemma 11, respectively).

For the next two lemmas, we define V_{OPT} as the set of all nodes that were in OPT cache at some time of P and let $V^{\text{c}}_{\text{OPT}} = T \setminus V_{\text{OPT}}$. Note that V_{OPT} is a union of subforests (nodes present in OPT's cache at consecutive times), and hence a subforest itself.

LEMMA 12. *For any phase P, it holds that $\text{req}(F^\infty) \leq 2 \cdot k_{\text{ONL}} \cdot \alpha + 2 \cdot \text{OPT}(P)$.*

PROOF. We assume first that P is a finished phase. Then, P ends with an artificial fetch of $X_{\text{end}(P)}$ at time $\text{end}(P)$ (followed by the final eviction). We split F^∞ into two disjoint parts (see Figure 2):

$$F^\infty_- = \{(v,t) : v \in C_{\text{end}(P)}, t \geq \text{last}_v(\text{end}(P))\} \ ,$$
$$F^\infty_+ = \{(v,t) : v \notin C_{\text{end}(P)} \sqcup X_{\text{end}(P)}, \\ t \geq \text{last}_v(\text{end}(P))\} \ .$$

Note that F^∞_- contains only negative requests and F^∞_+ only positive ones. As $\text{req}(F^\infty) = \text{req}(F^\infty_-) + \text{req}(F^\infty_+ \cap V^{\text{c}}_{\text{OPT}}) + \text{req}(F^\infty_+ \cap V_{\text{OPT}})$, we estimate each of these summands separately.

- Nodes from F^∞_- are in the cache $C_{\text{end}(P)}$ and were not evicted from the cache. Thus, $\text{req}(F^\infty_-) \leq |C_{\text{end}(P)}| \cdot \alpha \leq k_{\text{ONL}} \cdot \alpha$.
- All the requests from $V^{\text{c}}_{\text{OPT}}$ are paid by OPT, and hence $\text{req}(F^\infty_+ \cap V^{\text{c}}_{\text{OPT}}) \leq \text{req}(V^{\text{c}}_{\text{OPT}}) \leq \text{OPT}(P)$.
- F^∞_+ is a valid changeset for cache $C_{\text{end}(P)} \sqcup X_{\text{end}(P)}$. As V_{OPT} is a subforest of T, $F^\infty_+ \cap V_{\text{OPT}}$ is also a valid changeset for the cache $C_{\text{end}(P)} \sqcup X_{\text{end}(P)}$. Therefore, $\text{req}(F^\infty_+ \cap V_{\text{OPT}}) \leq \text{size}(F^\infty_+ \cap V_{\text{OPT}}) \cdot \alpha$, as otherwise the set fetched at time $\text{end}(P)$ would not be maximal. (TC could then fetch $X_{\text{end}(P)} \sqcup (F^\infty_+ \cap V_{\text{OPT}})$ instead of $X_{\text{end}(P)}$.) Thus, $\text{req}(F^\infty_+ \cap V_{\text{OPT}}) \leq |V_{\text{OPT}}| \cdot \alpha = k_{\text{OPT}} \cdot \alpha + (|V_{\text{OPT}}| - k_{\text{OPT}}) \cdot \alpha \leq k_{\text{ONL}} \cdot \alpha + \text{OPT}(P)$. The last inequality follows as — independently of the initial state — OPT needs to fetch at least $|V_{\text{OPT}}| - k_{\text{OPT}}$ nodes to the cache during P.

Hence, in total, $\text{req}(F^\infty) \leq 2 \cdot k_{\text{ONL}} \cdot \alpha + 2 \cdot \text{OPT}(P)$ for a finished phase P.

We note that if there was no cache change at $\text{end}(P)$, the analysis above would hold with $X_{\text{end}(P)} = \emptyset$ with virtually no change. Therefore, for an unfinished phase P ending with a fetch or ending without cache change at $\text{end}(P)$, the bound on $\text{req}(F^\infty)$ still holds. However, if an unfinished phase P ends with an eviction, then we look at the last eviction-free time τ of P. We now observe the evolution of field F^∞ from time τ till $\text{end}(P)$. At time τ, $\text{req}(F^\infty) \leq 2 \cdot k_{\text{ONL}} \cdot \alpha + 2 \cdot \text{OPT}(P)$. Furthermore, in subsequent times it may only decrease: at any round F^∞ gets an additional request, but on eviction $\text{req}(F^\infty)$ decreases by α times the number of evicted nodes (i.e., at least by $\alpha \geq 1$). Hence, the value of $\text{req}(F^\infty)$ at $\text{end}(P)$ is also at most $2 \cdot k_{\text{ONL}} \cdot \alpha + 2 \cdot \text{OPT}(P)$. □

By combining Lemma 3, Lemma 11 and Lemma 12, we immediately obtain the following corollary (holding for both finished and unfinished phases).

COROLLARY 13. *For any phase P, it holds that $\text{TC}(P) \leq O(h(T)) \cdot \text{OPT}(P) + O(h(T) \cdot (k_P + k_{\text{ONL}}) \cdot \alpha)$.*

Using the corollary above, its remains to bound the value of k_P. This is easy for an unfinished phase, as $k_P \leq k_{\text{ONL}}$ there. For a finished phase, we provide another bound.

LEMMA 14. *For any finished phase P, it holds that $k_P \cdot \alpha \leq \text{OPT}(P) \cdot (k_{\text{ONL}} + 1)/(k_{\text{ONL}} + 1 - k_{\text{OPT}})$.*

PROOF. First, we compute the number of positive requests in $V^{\text{c}}_{\text{OPT}}$. Let $X_{t_1}, X_{t_2}, \ldots, X_{t_s}$ be all positive changesets applied by TC in P. For any t, let $X'_t = X_t \setminus V_{\text{OPT}}$. As X_t is some tree cap and V_{OPT} is a subforest of T, X'_t is a tree cap of X_t. By Corollary 6, the number of requests to nodes of X'_t in field F^t is at least $|X'_t| \cdot \alpha$. These requests for different changesets X_t are disjoint and they are all outside of V_{OPT}. Hence the total number of positive requests outside of V_{OPT} is at least $\sum_{i=1}^s |X'_{t_i}| \cdot \alpha$, where $\sum_{i=1}^s |X'_{t_i}| \geq |\bigcup_{i=1}^s X'_{t_i}| = |(\bigcup_{i=1}^s X_{t_i}) \setminus V_{\text{OPT}}| \geq |\bigcup_{i=1}^s X_{t_i}| - |V_{\text{OPT}}| \geq k_P - |V_{\text{OPT}}|$.

Now $\text{OPT}(P)$ can be split into the cost associated with nodes from V_{OPT} and V^c_{OPT}, respectively. For the former part, OPT has to pay at least $(|V_{\text{OPT}}| - k_{\text{OPT}}) \cdot \alpha$ for the fetches alone. For the latter part, it has to pay 1 for each of at least $(k_P - |V_{\text{OPT}}|) \cdot \alpha$ positive requests outside of V_{OPT}. Hence, $\text{OPT}(P) \geq (|V_{\text{OPT}}| - k_{\text{OPT}}) \cdot \alpha + (k_P - |V_{\text{OPT}}|) \cdot \alpha = (k_P - k_{\text{OPT}}) \cdot \alpha$. Then, $k_P \cdot \alpha \leq k_P \cdot \text{OPT}(P)/(k_P - k_{\text{OPT}})$. As the phase is finished, $k_P \geq k_{\text{ONL}} + 1$, and thus $k_P \cdot \alpha \leq (k_{\text{ONL}} + 1) \cdot \text{OPT}(P)/(k_{\text{ONL}} + 1 - k_{\text{OPT}})$. □

THEOREM 15. TC *is* $O(h(T) \cdot k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1))$*-competitive.*

PROOF. Let $R = h(T) \cdot k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1)$. We split the input I into a sequence of finished phases followed by a single unfinished phase (which may not be present). For a finished phase P, we have $k_P > k_{\text{ONL}}$, and hence Corollary 13 and Lemma 14 imply that $\text{TC}(P) \leq O(R) \cdot \text{OPT}(P)$. For an unfinished phase $k_P \leq k_{\text{ONL}}$, and therefore, by Corollary 13, $\text{TC}(P) \leq O(h(T)) \cdot \text{OPT}(P) + O(h(T) \cdot k_{\text{ONL}} \cdot \alpha)$. Summing over all phases of I yields $\text{TC}(I) \leq O(R) \cdot \text{OPT}(I) + O(h(T) \cdot k_{\text{ONL}} \cdot \alpha)$. □

6. IMPLEMENTATION OF TC

Recall that at each time t, TC verifies the existence of a valid changeset that satisfies saturation and maximality properties (see the definition of TC in Section 4). Here, we show that this operation can be performed efficiently. In particular, in the following two subsections, we will prove the following theorem.

THEOREM 16. TC *can be implemented using* $O(|T|)$ *additional memory, so that to make a decision at time* t*, it performs* $O(h(T) + \max\{h(T), deg(T)\} \cdot |X_t|)$ *operations, where* $deg(T)$ *is a maximum node degree in* T *and* X_t *is the changeset applied at time* t *(*$|X_t| = 0$ *if no changeset is applied).*

Let v_t be the node requested at round t. Note that we may restrict our attention to requests that entail a cost for TC, as otherwise its counters remain unchanged and certainly TC does not change cache contents. We use Lemma 1 to restrict possible candidates for changesets that can be applied at time t. First, we note that if a node v_t requested at round t is outside the cache, then, at time t, TC may only fetch some changeset, and otherwise it may only evict some changeset. Therefore, we may construct two separate schemes, one governing fetches and one for evictions.

In Section 6.1, using Lemma 1, we show that after processing a positive request, TC needs to verify at most $h(T)$ possible positive changesets, each in constant time, using an auxiliary data structure. The cost of updating this structure at time t is $O(h(T) + h(T) \cdot |X_t|)$.

The situation for negative changesets is more complex as even after applying Lemma 1 there are still exponentially many valid negative changesets to consider. In Section 6.2, we construct an auxiliary data structure that returns a viable candidate in time $O(h(T) + \deg(T) \cdot |X_t|)$. The update of this structure at time t requires $O(h(T) + \deg(T) \cdot |X_t|)$ operations.

6.1 Positive Requests and Fetches

At any time t and for any non-cached node u, we may define $P_t(u)$ as a tree cap rooted at u containing all non-cached nodes from $T(u)$. During an execution of TC, we maintain two values for each non-cached node u: $\text{cnt}_t(P_t(u))$ and $|P_t(u)|$. When a counter at node v_t is incremented, we update $\text{cnt}_t(P_t(u))$ for each ancestor u of v (at most $h(T)$ updated values). Furthermore, if a node v changes its state from cached to non-cached (or vice versa), we update the value of $|P_t(u)|$ for any ancestor u of v (at most $h(T)$ updates per each node that changes the state). Therefore, the total cost of updating these structures at time t is at most $O(h(T) + h(T) \cdot |X_t|)$.

By Lemma 1, a positive valid changeset fetched at time t has to contain v_t and is a single tree cap. Such a tree cap has to be equal to $P_t(u)$ for u being an ancestor of v_t. Hence, we may iterate over all ancestors u of v_t, starting from the tree root and ending at v_t, and we stop at the first node u, for which $P_t(u)$ is saturated (i.e., $\text{cnt}_t(P_t(u)) \geq |P_t(u)| \cdot \alpha$). If such a u is found, the corresponding set $P_t(u)$ satisfies also the maximality condition (cf. the definition of TC) as all valid changesets that are supersets of $P_t(u)$ were already verified to be non-saturated. Therefore, in such a case, TC fetches $P_t(u)$. Otherwise, if no saturated changeset is found, TC does nothing. Checking all ancestors of v_t can be performed in time $O(h(T))$.

6.2 Negative Requests and Evictions

Handling evictions is more complex. If the request to node v_t at round t was negative, Lemma 1 tells us only that the negative changeset evicted by TC has to be a tree cap rooted at u, where u is the root of the cached tree containing v_t. There are exponentially many such tree caps, and hence their naïve verification is intractable. To alleviate this problem, we introduce the following helper notion. For any set of cached nodes A and any time t, let

$$\text{val}_t(A) = \text{cnt}_t(A) - |A| \cdot \alpha + \frac{|A|}{|T| + 1} \ .$$

Note that for any non-empty set A, $\text{val}_t(A) \neq 0$ as the first two terms are integers and $|A|/(|T| + 1) \in (0, 1)$. Furthermore, val_t is additive: for two disjoint sets A and B, $\text{val}_t(A \sqcup B) = \text{val}_t(A) + \text{val}_t(B)$. For any time t and a cached node u, we define

$$H_t(u) = \arg\max_D \{\text{val}_t(D) : D \text{ is a non-empty tree cap rooted at } u\} \ .$$

Our scheme maintains the value $H_t(u)$ for any cached node u. To this end, we observe that $H_t(u)$ can be defined recursively as follows. Let $H'_t(u) = H_t(u)$ if $\text{val}_t(H_t(u)) > 0$ and $H'_t(u) = \emptyset$ otherwise. Then, for any node v and time t, by additivity of val_t,

$$H_t(u) = \{u\} \sqcup \bigsqcup_{w \text{ is a child of } u} H'_t(w) \ .$$

Each cached node u keeps the value $\text{val}_t(H_t(u))$. Note that set $H_t(u)$ itself can be recovered from this information: we iterate over all children of u (at most $\deg(T)$ of them) and for each child w, if $\text{val}_t(H_t(w)) > 0$, we recursively compute set $H_t(w)$. Thus, the total time for constructing $H_t(u)$ is $O(\deg(T) \cdot |H_t(u)|)$.

During an execution of TC, we update stored values accordingly. That is, whenever a counter at a cached node v_t is incremented, we update $\text{val}_t(H_t(u))$ values for each cached ancestor u of v_t, starting from $u = v_t$ and proceeding towards the cached tree root. Any such update can be performed in constant time, and the total time is thus $O(h(T))$. For a cache change, we process nodes from the changeset iteratively, starting with nodes closest to the root in case of an eviction and furthest from the root in case of a fetch. For any such node u, we appropriately stop or start maintaining the corresponding value of $\text{val}_t(H_t(u))$. The latter requires looking up the stored values at all its children. As u does not have cached ancestors, sets H_t (and hence also the stored values) at other nodes remain unchanged. In total, the cost of updating all H_t values at time t is at most $O(h(T) + \deg(T) \cdot |X_t|)$.

Finally, we show how to use sets H_t to quickly choose a valid changeset for eviction. Recall that for a negative request v_t, the changeset to be evicted has to be a tree cap rooted at u, where u is the root of a cached subtree containing v_t. For succinctness, we use H^u to denote $H_t(u)$. We show that if $\text{val}_t(H^u) < 0$, then there is no valid negative changeset that is saturated, and hence TC does not perform any action, and if $\text{val}_t(H^u) > 0$, then H^u is both saturated and maximal, and hence TC may evict H^u.

1. First, assume that $\text{val}_t(H^u) < 0$. Then, for any tree cap X rooted at u, it holds that $\text{cnt}_t(X) - |X| \cdot \alpha < \text{val}_t(X) \le \text{val}_t(H^u) < 0$, i.e., X is not saturated, and hence cannot be evicted by TC.

2. Second, assume that $\text{val}_t(H^u) > 0$. As $\text{cnt}_t(H^u) - |H^u| \cdot \alpha$ is an integer and $|H^u|/(|T|+1) < 1$, it holds that $\text{cnt}_t(H^u) - |H^u| \cdot \alpha \ge 0$, i.e., H^u is saturated. Furthermore, by Lemma 1, $\text{cnt}_t(H^u) \le |H^u| \cdot \alpha$, and therefore $\text{cnt}_t(H^u) - |H^u| \cdot \alpha = 0$, i.e., $\text{val}_t(H^u) = |H^u|/(|T|+1)$. It remains to show that H^u is maximal, i.e., there is no valid saturated changeset $Y \supsetneq H^u$. By Lemma 1, Y has to be a tree cap rooted at u as well. If Y was saturated, $\text{val}_t(Y) = \text{cnt}_t(Y) - |Y| \cdot \alpha + |Y|/(|T|+1) \ge |Y|/(|T|+1) > |H^u|/(|T|+1) = \text{val}_t(H^u)$, which would contradict the definition of H^u.

Note that node u can be found in time $O(h(T))$, and the actual set H^u (of size $|X_t|$) can be computed in time $O(\deg(T) \cdot |X_t|)$. Therefore the total time for finding set $|X_t|$ is $O(h(T) + \deg(T) \cdot |X_t|)$.

6.3 Practical Considerations

In practical implementations, one may also resort to standard techniques such as sampling. Namely, for each request that costs 1 (that would normally entail counter updates), we toss a biased coin. Then, we ignore this request in counter updates with probability $(1 - 1/\sqrt{\alpha})$ and we increment a corresponding counter by $\sqrt{\alpha}$ with probability $1/\sqrt{\alpha}$. This modification can tremendously speed up the execution of the algorithm, while sacrificing the (expected) competitive ratio only by a constant factor. Details will be given in the full version of the paper.

7. CONCLUSION

This paper attended to a novel variant of online paging which finds applications in the context of IP routing networks where forwarding rules can be cached. We presented a deterministic online algorithm that achieves a provably competitive tradeoff between the benefit of caching and update costs. We believe that our work opens interesting directions for future research. Most importantly, it will be interesting to study the optimality of the derived competitive ratio.

Acknowledgements

The authors would like to thank Fred Baker from Cisco, Moti Medina from the Max-Planck-Institute and Paweł Gawrychowski from University of Wrocław for useful inputs.

8. REFERENCES

[1] BGP statistics from route-views data. http://bgp.potaroo.net/bgprpts/rva-index.html.

[2] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1–2):203–218, 2000.

[3] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *23rd ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1681–1689, 2012.

[4] M. Bienkowski, N. Sarrar, S. Schmid, and S. Uhlig. Competitive FIB aggregation without update churn. In *Proc. 34th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 607–616, 2014.

[5] M. Bienkowski and S. Schmid. Competitive FIB aggregation for independent prefixes: Online ski rental on the trie. In *Proc. 20th Int. Colloq. on Structural Information and Communication Complexity (SIROCCO)*, pages 92–103, 2013.

[6] M. Brehob, R. J. Enbody, E. Torng, and S. Wagner. On-line restricted caching. *Journal of Scheduling*, 6(2):149–166, 2003.

[7] N. Buchbinder, S. Chen, and J. Naor. Competitive algorithms for restricted caching and matroid caching. In *Proc. 22th European Symp. on Algorithms (ESA)*, pages 209–221, 2014.

[8] M. Chrobak, H. J. Karloff, T. H. Payne, and S. Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[9] L. Cittadini, W. Muhlbauer, S. Uhlig, R. Bushy, P. Francois, and O. Maennel. Evolution of internet address space deaggregation: myths and reality. *IEEE Journal on Selected Areas in Communications*, 28(8):1238–1249, 2010.

[10] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing optimal IP routing tables. In *Proc. 18th IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 88–97, 1999.

[11] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.

[12] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[13] P. François, C. Filsfils, J. Evans, and O. Bonaventure. Achieving sub-second IGP convergence in large IP networks. volume 35, pages 35–44, 2005.

[14] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 43–48, 2013.

[15] S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[16] E. Karpilovsky, M. Caesar, J. Rexford, A. Shaikh, and J. E. van der Merwe. Practical network-wide compression of IP routing tables. *IEEE Transactions on Network and Service Management*, 9(4):446–458, 2012.

[17] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. ACM Symposium on SDN Research (SOSR)*, 2016.

[18] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting route caching: The world should be flat. In *Proc. 10th Int. Conf. on Passive and Active Network Measurement (PAM)*, pages 3–12, 2009.

[19] H. Liu. Routing prefix caching in network processor design. In *Proc. 10th Int. Conf. on Computer Communications and Networks (ICCCN)*, pages 18–23, 2001.

[20] Y. Liu, V. Lehman, and L. Wang. Efficient FIB caching using minimal non-overlapping prefixes. *Computer Networks*, 83:85–99, 2015.

[21] Y. Liu, B. Zhang, and L. Wang. FIFA: fast incremental FIB aggregation. In *Proc. 32nd IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 1213–1221, 2013.

[22] Y. Liu, X. Zhao, K. Nam, L. Wang, and B. Zhang. Incremental forwarding table aggregation. In *Proc. Global Communications Conference (GLOBECOM)*, pages 1–6, 2010.

[23] L. Luo, G. Xie, K. Salamatian, S. Uhlig, L. Mathy, and Y. Xie. A trie merging approach with incremental updates for virtual routers. In *Proc. 32nd IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 1222–1230, 2013.

[24] L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.

[25] M. Mendel and S. S. Seiden. Online companion caching. *Theoretical Computer Science*, 324(2–3):183–200, 2004.

[26] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger. Compressing IP forwarding tables: towards entropy bounds and beyond. In *Proc. ACM SIGCOMM Conference*, pages 111–122, 2013.

[27] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 42(1):16–22, 2012.

[28] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[29] E. Spitznagel, D. E. Taylor, and J. S. Turner. Packet classification using extended tcams. In *Proc. 11th IEEE Int. Conf. on Network Protocols (ICNP)*, pages 120–131, 2003.

[30] S. Suri, T. Sandholm, and P. R. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[31] Z. A. Uzmi, M. E. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis. SMALTA: practical and near-optimal FIB aggregation. In *Proc. 7th Int. Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, 2011.

[32] N. E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[33] N. E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

[34] X. Zhao, Y. Liu, L. Wang, and B. Zhang. On the aggregatability of router forwarding tables. In *Proc. 29th IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 848–856, 2010.

APPENDIX

A. PROOF OF LEMMA 1

Before showing Lemma 1 (restated below for convenience), we show the following technical claim.

Claim 17. *For any phase P, the following invariants hold for any time $t > begin(P)$:*

1. *$cnt_{t-1}(X) < |X| \cdot \alpha$ for a valid changeset X for C_t,*
2. *$cnt_t(X) \leq |X| \cdot \alpha$ for a valid changeset X for C_t,*
3. *any changeset X with property $cnt_t(X) = |X| \cdot \alpha$ contains the node requested at round t.*

Proof. First observe that Invariant 1 (for time t) along with the fact that round t contains only one request immediately implies that $\text{cnt}_t(X) \leq \text{cnt}_{t-1}(X) + 1 \leq (|X| \cdot \alpha - 1) + 1 = |X| \cdot \alpha$, i.e., Invariant 2 for time t. Furthermore the equality may hold only for changesets containing the node requested at round t, which implies Invariant 3 for time t.

It remains to show that Invariant 1 holds for any step $t > \text{begin}(P)$. Invariant 1 is trivially true for $t = \text{begin}(P)+1$ as $\text{cnt}_{t-1}(X) = 0$ then. Let $t+1$ be the earliest time in phase P for which Invariant 1 does not hold; we will then show a contradiction with the definition of TC or a contradiction with other Invariants at time t. That is, we assume that there exists a positive changeset X for C_{t+1} such that $\text{cnt}_t(X) \geq |X| \cdot \alpha$ (the proof for a negative changeset is analogous). Note that TC must have performed an action (fetch or eviction) at time t as otherwise X would be also a changeset for $C_t = C_{t+1}$ with $\text{cnt}_t(X) \geq |X| \cdot \alpha$, which means that X should have been applied by TC at time t. We consider two cases.

If TC fetches a positive changeset Y at time t, $C_{t+1} = C_t \sqcup Y$ and $\text{cnt}_t(Y) = |Y| \cdot \alpha$. Then, $Y \sqcup X$ is a changeset for C_t, and $\text{cnt}_t(Y \sqcup X) \geq |Y \sqcup X| \cdot \alpha$. This contradicts the maximality property of set Y chosen at time t by TC.

If TC evicts a negative changeset Y at time t, $C_{t+1} = C_t \setminus Y$. Invariant 2 and the definition of TC implies $\text{cnt}_t(Y) = |Y| \cdot \alpha$, and thus, by Invariant 3, Y contains the node requested at round t. As $X \cap Y \subseteq C_t$, $X \cap Y$ does not have any positive requests at time t, and therefore $\text{cnt}_t(X \setminus Y) = \text{cnt}_t(X) \geq |X| \cdot \alpha \geq |X \setminus Y| \cdot \alpha$. By Invariant 2, $\text{cnt}_t(X \setminus Y) \leq |X \setminus Y| \cdot \alpha$, and hence $\text{cnt}_t(X \setminus Y) = |X \setminus Y| \cdot \alpha$. This contradicts Invariant 3 as $X \setminus Y$ cannot contain the node requested at round t (because Y contains this node). □

Lemma 1. *Fix any time $t > begin(P)$. For any valid changeset X for C_t, it holds that $cnt_t(X) \leq |X| \cdot \alpha$. If a changeset X is applied at time t, the following properties hold:*

1. *X contains the node requested at round t,*
2. *$cnt_t(X) = |X| \cdot \alpha$,*
3. *$cnt_t(Y) < |Y| \cdot \alpha$ for any valid changeset Y for C_{t+1} (note that C_{t+1} is the cache state right after application of X),*
4. *X is a tree cap of a tree from C_{t+1} if X is positive and it is a tree cap of a tree from C_t if X is negative.*

Proof. The inequality $\text{cnt}_t(X) \leq |X| \cdot \alpha$ is equivalent to Invariant 2 of Claim 17. Assume now that X is applied at time t. By the definition of TC, $\text{cnt}_t(X) \geq |X| \cdot \alpha$, and thus $\text{cnt}_t(X) = |X| \cdot \alpha$, i.e., Property 2 follows. Then, Invariant 3 of Claim 17 implies Property 1. Finally, Invariant 1 of Claim 17 for time $t+1$ is equivalent to Property 3.

To show Property 4 of the lemma, observe that the changeset X applied at time t cannot be a disjoint union of two (or more) valid changesets X_1 and X_2. By Property 2, $|X| \cdot \alpha = \text{cnt}_t(X) = \text{cnt}_t(X_1) + \text{cnt}_t(X_2)$. If $\text{cnt}_t(X_1) < |X_1| \cdot \alpha$ or $\text{cnt}_t(X_2) < |X_2| \cdot \alpha$, then $\text{cnt}_t(X_1) + \text{cnt}_t(X_2) < (|X_1| + |X_2|) \cdot \alpha = |X| \cdot \alpha$, a contradiction. Therefore, $\text{cnt}_t(X_1) = |X_1| \cdot \alpha$ and $\text{cnt}_t(X_2) = |X_2| \cdot \alpha$. But then Invariant 3 of Claim 17 would imply that both X_1 and X_2 contain a node requested at time t, which is a contradiction as they are disjoint.

Therefore, if X is a positive changeset applied at t, then X is a single tree cap of a tree from subforest C_{t+1}, and likewise if X is negative, then X is a single tree cap of a tree from subforest C_t. □

B. MINIMIZING FORWARDING TABLES USING TREE CACHING

In this section, we present a formal argument showing why we can use any q-competitive online algorithm A_T for the tree caching problem to obtain a $2q$-competitive online algorithm A that minimizes forwarding tables.

Namely, we take any input I for the latter problem and create, in online fashion, an input I_T for the tree caching problem in a way described in Section 2. For any solution for I_T, we may replay its actions (fetches and evictions) on I and vice versa. However, there is one place, where these solutions may have different costs. Recall that an update of a rule stored at node v in I is mapped to a *chunk* of α negative requests to v in I_T. It is then possible that an algorithm for I_T modifies the cache *during* a chunk. An algorithm that never performs such an action is called *canonic*.

To alleviate this issue, we first note that any algorithm B for I_T can be transformed into a canonic solution B' by postponing all cache modifications that occur during some chunk to the time right after it. Such a transformation may increase the cost of a solution on a chunk at most by α and such an increase occurs only when B modifies a cache within this chunk. Hence, the additional cost of transformation can be mapped to the already existing cost of B, and thus the cost of B' is at most by a factor of 2 larger than that of B.

Furthermore, note that there is a natural cost-preserving bijection between solutions to I and canonic solutions to I_T (solutions perform same cache modifications). Hence, the algorithm A for I runs A_T on I_T, transforms it in an online manner into the canonic solution $A'_T(I_T)$, and replays its cache modification on I. We obtain that

$$\begin{aligned} A(I) = A'_T(I_T) &\le 2 \cdot A_T(I_T) \\ &\le 2q \cdot \textsc{Opt}(I_T) \le 2q \cdot \textsc{Opt}(I) \ . \end{aligned}$$

The second inequality follows immediately by the q-competitiveness of A_T. The third inequality follows by replaying cache modifications as well, but this time we take solution $\textsc{Opt}(I)$ and replay its actions on I_T, creating a canonic (not necessarily optimal) solution of the same cost.

C. LOWER BOUND ON THE COMPETITIVE RATIO

Theorem 18. *For any $\alpha \ge 1$, the competitive ratio of any online algorithm for the online tree caching problem is at least $\Omega(k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1))$*

Proof. We will assume that in the tree caching problem, evictions are free (this changes the cost by at most by a factor of two). We consider a tree whose leaves correspond to the set of all pages in the paging problem. The rest of the tree will be irrelevant.

For any input sequence I for the paging problem, we may create a sequence I_{T} for tree caching, where a request to a page is replaced by α requests to the corresponding leaf. Now, we claim that any solution A for I of cost c can be transformed, in online manner, into a solution A_{T} for I_{T} of cost $\Theta(\alpha \cdot c)$ and vice versa.

If upon a request r, an algorithm A fetches r to the cache and evicts some pages, then A_{T} bypasses α corresponding requests to leaf r, fetches r afterwards and evicts the corresponding leaves, paying $O(\alpha)$ times the cost of A. By doing it iteratively, A_{T} ensures that its cache is equivalent to that of A. In particular, a request free for A is also free for A_{T}.

Now take any algorithm A_{T} for I_{T}. It can be transformed to the algorithm A'_{T} that (i) keeps only leaves of the tree in the cache and (ii) performs actions only at times that are multiplicities of α (losing at most a constant factor in comparison to A_{T}). Then, fix any chunk of α requests to some leaf r' immediately followed by some fetches and evictions of A'_{T} leaves. Upon seeing the corresponding request r' in I, the algorithm A performs fetches and evictions on the corresponding pages. In effect, the cost of A is $O(1/\alpha)$ times the cost of A_{T}.

The bidirectional reduction described above preserves competitive ratios up to a constant factor. Hence, applying the adversarial strategy for the paging problem that enforces the competitive ratio $R = k_{\text{ONL}}/(k_{\text{ONL}} - k_{\text{OPT}} + 1)$ [28] immediately implies the lower bound of $\Omega(R)$ on the competitive ratio for the tree caching problem. □

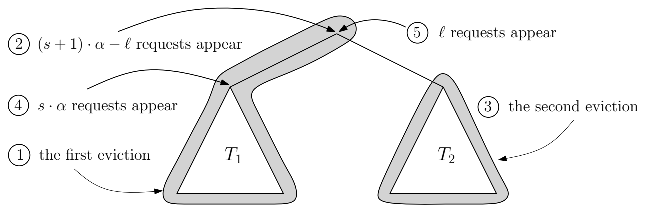

Figure 4: A troublesome example of a positive field. Numbers in circles describe the chronology of the events.

D. IMPOSSIBILITY OF EXACT SHIFTING WITHIN POSITIVE FIELDS

In this section, we present an example showing that, within a positive field, we cannot shift positive requests down, obtaining α requests in every node, like we did in the case of negative requests (cf. Corollary 8). T consists of root r and two distinct subtrees T_1 and T_2, each of size s and containing ℓ leaves.

Suppose that at the beginning TC has the entire tree T in its cache and the following ordered events happen (cf. Figure 4).

1. TC evicts $T_1 \cup \{r\}$ from the cache.
2. $(s+1)\cdot\alpha-\ell$ requests appear one by one at r. The number of requests is too small to trigger a fetch of any subtree of $T_1 \cup \{r\}$.
3. TC evicts T_2 from the cache.
4. $s \cdot \alpha$ requests appear one by one at the root of T_1. This time, the number of requests is too small to trigger a fetch of any subtree of T.
5. ℓ requests appear one by one at r. After the last one appears, TC fetches the entire T to the cache.

The evictions happen because of some feasible sequence of negative requests that is irrelevant from our perspective.

Now, observe that when requests appear at the root in the second stage of our construction, T_2 is still in the cache (i.e., does not belong to the field yet). Thus, all the requests, except for the last ℓ ones can be shifted down only to nodes from T_1. Hence, for large α and s, shifting can deliver $\Omega(\alpha)$ requests only to half of the nodes.

E. STATIC CACHING

In addition to our solution to online tree caching, we show how to choose a static set of cached nodes, in the setting where only positive requests are present and their number is known to the algorithm. More precisely, let $\textsc{freq}_v$ be the total number of positive requests to v. We want to construct an optimal cache, so that the cost of an algorithm is minimized. As only positive requests are present, we may assume that the number of nodes in the cache is exactly k. We note that the generalization of this problem when the underlying dependencies do not form a tree but a directed acyclic graph was shown to be NP-complete [17].

In the context of our IP routing / SDN application, this problem corresponds to choosing a fixed subset of rules that are always in the cache. For example, if the frequency distribution over the rules is relatively stable, it may be sufficient to compute a new solution at certain time intervals only (e.g., every one or two hours).

Notation. From now on, we denote the number of nodes in the tree T by n. Our algorithm first introduces (at most n) additional *artificial* nodes in T, so that the resulting tree T' is binary (each non-leaf has exactly two children). We call the original, non-artificial nodes *real*.

Hence, the problem is to find a tree cap S of T consisting of $b = n - k$ real nodes (and any number of artificial ones) that are outside the cache, so that $\sum_{v \in S} \text{FREQ}_v$, the total number of requests to S is minimized.

The Recurrence. We use dynamic programming on the tree T'. By $m \leq 2n$ we denote the total size of T' (including artificial nodes).

For each node $v \in T'$ and for each $j \in [0, b]$, we define $g(v, j)$ to be the minimum total cost of a tree cap of $T'(v)$ consisting of exactly j real nodes. For a leaf v, we have

$$g(v, j) = \begin{cases} 0 & \text{if } j = 0, \\ \text{FREQ}_v & \text{if } j = 1 \text{ and } v \text{ is real}, \\ \infty & \text{otherwise.} \end{cases}$$

For an internal node v, we exploit the optimal substructure of the solution and use the values computed at its children, denoted $\ell(v)$ and $r(v)$. Namely, if v is a real node, then either the selected tree cap of $T'(v)$ is empty or it contains $j \geq 1$ real nodes. In the latter case, one of these nodes is v itself, i real nodes can be selected from $T'(\ell(v))$ and $j - 1 - i$ real nodes can be selected from $T'(r(v))$. Thus, the recursive formula is for a real node v is $g(v, 0) = 0$ and for $j > 0$ it holds that

$$g(v, j) = \text{FREQ}_v + \min_{0 \leq i \leq j-1} \{g(\ell(v), i) + g(r(v), j-1-i)\} \ .$$

If v is an artificial node, there are no requests to v and we have to choose all j elements from both subtrees. Hence, in this case, for any $j \geq 0$

$$g(v, j) = \min_{0 \leq i \leq j} \{g(\ell(v), i) + g(r(v), j - i)\} \ .$$

The Algorithm. Our algorithm evaluates the values of function g, starting from the leaves and continuing to the root. For any node v and value $j \in [0, b]$, it also records how the minimum was achieved. It then uses these values, traversing the tree from the root downwards to actually recover which nodes should be put in set S. As explained below, a naïve implementation of this scheme yields a runtime of $O(n^3)$, which can be later improved to $O(n^2)$.

THEOREM 19. *An optimal static cache can be computed in time $O(n^2)$.*

PROOF. First, we show a simpler but weaker bound. The algorithm traverses m tree nodes of T'. Computation performed at each node v requires computing values $g(v, j)$ for $j \in [0, b]$. (The computed values are stored in a lookup table.) Computing a single entry $g(v, j)$ can be performed by taking the minimum among the $j \leq b$ entries: each one can be looked up in constant time. Thus, in total the execution time is $O(m \cdot b^2) = O(n \cdot b^2)$.

In most applications, b is of the same order as n which makes this complexity cubic in n. However, the computation of function g can be slightly tweaked, which can tremendously improve the running time.

Namely, for a single node v we compute all the values $g(v, j)$ at a single fell swoop. Precisely, we traverse the tree T' as before. For a node v, we assign infinities to all values $g(v, j)$ at the very beginning. Later, we iterate over all possible values $j_\ell \in [0, b]$ and $j_r \in [0, b]$ and try to improve the current value of $g(v, 1 + j_\ell + j_r)$ (or $g(v, j_\ell + j_r)$ for an artificial node) with the value of $\text{FREQ}_v + g(\ell(v), j_\ell) + g(r(v), j_r)$.

This alone does not improve the running time of $O(m \cdot b^2)$, but we may neglect the values of of $g(\ell(v), j_\ell)$ and $g(r(v), j_r)$ for which we know that they are infinite. Namely, we only have to iterate j_ℓ over the set $[0, \min\{b, \text{size}(\ell(v))\}]$ and similarly iterate j_r over the set $[0, \min\{b, \text{size}(r(v))\}]$, where $\text{size}(u)$ denotes the number of real nodes in $T'(u)$ (including u).

Computing all entries $g(v, j)$ for a fixed node v (including the initialization to infinities) takes then time $O(b + \text{size}(\ell(v)) \cdot \text{size}(r(v)))$. Thus, the total runtime is $O(m \cdot b + \sum_{v \in \text{inner}(T')} \text{size}(\ell(v)) \cdot \text{size}(r(v))$, where $\text{inner}(T')$ is the set of all inner (non-leaf) nodes of T'.

To bound the latter summand, we may write it as a cardinality of a set Z that contains all possible triplets (v, x, y) of active nodes, such that $x \in T(\ell(v))$ and $y \in T(r(v))$. For a given pair of nodes (x, y) there is at most one value of v, such that $(v, x, y) \in Z$ as v has to be least common ancestor of x and y. This implies that $\sum_{v \in \text{inner}(T')} \text{size}(\ell(v)) \cdot \text{size}(r(v)) = |Z| \leq m^2 = O(n^2)$.

This gives the desired runtime of $O(m \cdot b + n^2) = O(n^2)$. □