Optimal cache-aware suffix selection

Gianni Franceschini, Roberto Grossi, S. Muthukrishnan

Symposium on Theoretical Aspects of Computer Science 2009 (Freiburg), pp. 457–468
www.stacs-conf.org

OPTIMAL CACHE-AWARE SUFFIX SELECTION

GIANNI FRANCESCHINI [1] AND ROBERTO GROSSI [2] AND S. MUTHUKRISHNAN [3]

[1] Dipartimento di Informatica, Università di Roma "La Sapienza", and PINT, Primorska University
E-mail address: francesc@di.uniroma1.it

[2] Dipartimento di Informatica, Università di Pisa
E-mail address: grossi@di.unipi.it

[3] Google Inc., NY
E-mail address: muthu@google.com

Abstract. Given string $S[1..N]$ and integer k, the *suffix selection* problem is to determine the kth lexicographically smallest amongst the suffixes $S[i \dots N]$, $1 \leq i \leq N$. We study the suffix selection problem in the cache-aware model that captures two-level memory inherent in computing systems, for a *cache* of limited size M and block size B. The complexity of interest is the number of block transfers. We present an optimal suffix selection algorithm in the cache-aware model, requiring $\Theta(N/B)$ block transfers, for any string S over an unbounded alphabet (where characters can only be compared), under the common tall-cache assumption (i.e. $M = \Omega\left(B^{1+\epsilon}\right)$, where $\epsilon < 1$). Our algorithm beats the bottleneck bound for permuting an input array to the desired output array, which holds for nearly any nontrivial problem in hierarchical memory models.

1. Introduction

Background: Selection vs Sorting. A collection of N *numbers* can be sorted using $\Theta(N \log N)$ comparisons. On the other hand, the famous five-author result [2] from early 70's shows that the problem of *selection* — choosing the kth smallest number — can be solved using $O(N)$ comparisons in the worst case. Thus, selection is provably simpler than sorting in the comparison model.

Consider a sorting vs selection question for strings. Say $S = S[1 \cdots N]$ is a string. The *suffix sorting* problem is to sort the suffixes $S[i \cdots N]$, $i = 1, \dots, N$, in the *lexicographic* order. In the comparison model, we count the number of character comparisons. Suffix sorting can be performed with $O(N \log N)$ comparisons using a combination of character sorting and classical data structure of suffix arrays or trees [11, 9, 4]. There is a lower bound of $\Omega(N \log N)$ since sorting suffixes ends up sorting the characters. For the related *suffix selection* problem where the goal is to output the kth lexicographically smallest suffix of S, the result in [6] recently gave an optimal $O(N)$ comparison-based algorithm, thereby showing that suffix selection is provably simpler than suffix sorting.

The first two authors have been partially supported by the MIUR project MAINSTREAM.

The Model. Time-tested architectural approaches to computing systems provide two (or more) levels of memory: the highest one with a limited amount of fast memory; the lowest one with slow but large memory. The CPU can only access input stored on the fastest level. Thus, there is a continuous exchange of data between the levels. For cost and performance reasons, data is exchanged in fixed-size blocks of contiguous locations. These transfers may be triggered automatically like in internal CPU caches, or explicitly, like in the case of disks; in either case, more than the number of computing operations executed, the number of block transfers required is the actual bottleneck.

Formally, we consider the model that has two memory levels. The *cache* level contains M locations divided into blocks (or cache lines) of B contiguous locations, and the *main memory* level can be arbitrarily large and is also divided into blocks. The processing unit can address the locations of the main memory but it can process only the data residing in cache. The algorithms that know and exploit the two parameter M and B, and optimize the number of block transfers are *cache-aware*. This model includes the classical External Memory model [1] as well as the well-known *Ideal-Cache model* [7].

Motivation. Suffix selection as a problem is useful in analyzing the order statistics of suffixes in a string such as the extremes, medians and outliers, with potential applications in bioinformatics and information retrieval. A quick method for finding say the suffixes of rank $i(n/10)$ for each integer i, $0 \leq i \leq 10$, may be used to partition the space of suffixes for understanding the string better, load balancing and parallelization. But in these applications, such as in bioinformatics, the strings are truly massive and unlikely to fit in the fastest levels of memory. Therefore it is natural to analyze them in a hierarchical memory model.

Our primary motivation however is really theoretical. Since the inception of the first block-based hierarchical memory model ([1],[10]), it has been difficult to obtain "golden standard" algorithms i.e., those using just $O(N/B)$ block transfers. Even the simplest permutation problem (PERM henceforth) where the output is a specified permutation of the input array, does not have such an algorithm. In the standard RAM model, PERM can be solved in $O(N)$ time. In both the Ideal-Cache and External Memory models, the complexity of this problem is denoted $\text{PERM}(N) = \Theta\left(\min\left\{N, (N/B\log_{M/B} N/B\right\}\right)$. Nearly any nontrivial problem one can imagine from list ranking to graph problems such as Euler tours, DFS, connected components etc., sorting and geometric problems have the lower bound of $\text{PERM}(N)$, even if they take $O(N)$ time in the RAM model, and therefore do not meet the "golden standard". Thus the lower bound for PERM is a terrible bottleneck for block-based hierarchical memory models.

The outstanding question is, much as in the comparison model, is suffix selection provably simpler than suffix sorting in the block-based hierarchical memory models? Suffix sorting takes $\Theta\left((N/B)\log_{M/B}(N/B)\right)$ block transfers [5]. Proving any problem to be simpler than suffix sorting therefore requires one to essentially overcome the PERM bottleneck.

Our Contribution. We present a suffix selection algorithm with optimal cache complexity. Our algorithm requires $\Theta(N/B)$ block transfers, for any string S over an unbounded alphabet (where characters can only be compared) and under the common tall-cache assumption, that is $M = \Omega\left(B^{1+\epsilon}\right)$ with $\epsilon < 1$. Hence, we meet the "golden standard"; we

beat the PERM bottleneck and consequently, prove that suffix selection is easier than suffix sorting in block-based hierarchical memory models.

Overview. Our high level strategy for achieving an optimal cache-aware suffix selection algorithm consists of two main objectives.

In the first objective, we want to efficiently reduce the number of candidate suffixes from N to $O(N/B)$, where we maintain the invariant that the wanted kth smallest suffix is surely one of the candidate suffixes.

In the second objective, we want to achieve a cache optimal solution for the *sparse suffix selection problem*, where we are given a subset of $O(N/B)$ suffixes including also the wanted kth suffix. To achieve this objective we first find a simpler approach to suffix selection for the standard comparison model. (The only known linear time suffix selection algorithm for the comparison model [6] hinges on well-known algorithmic and data structural primitives whose solutions are inherently cache inefficient.) Then, we modify the simpler comparison-based suffix selection algorithm to exploit, in a cache-efficient way, the hypothesis that $O(N/B)$ (known) suffixes are the only plausible candidates.

Map of the paper. We will start by describing the new simple comparison-based suffix selection algorithm in Section 2. This section is meant to be intuitive. We will use it to derive a cache-aware algorithm for the sparse suffix selection problem in Section 3. We will present our optimal cache-aware algorithm for the general suffix selection problem in Section 4.

2. A Simple(r) Linear-Time Suffix Selection Algorithm

We now describe a simple algorithm for selecting the kth lexicographically smallest suffix of S in main memory. We give some intuitions on the central notion of `work`, and some definitions and notations used in the algorithm. Next, we show how to perform main iteration, called *phase transition*. Finally, we present the invariants that are maintained in each phase transition, and discuss the correctness and the complexity of our algorithm.

Notation and intuition. Consider the regular linear-time selection algorithm [2], hereafter called BFPRT. Our algorithm for a string $S = S[1 \dots N]$ uses BFPRT as a black box.[1] Each run of BFPRT permits to discover a longer and longer prefix of the (unknown) kth lexicographically smallest suffix of S. We need to carefully orchestrate the several runs of BFPRT to obtain a total cost of $O(N)$ time. We use $S =$ `bbbabbbbbaa`\$, where $n = 12$, as an illustrative example, and show how to find the median suffix (hence, $k = n/2 = 6$).

Phases and phase transitions. We organize our computation so that it goes through *phases*, numbered $t = 0, 1, 2, \dots$ and so on. In phase t, we know that a certain string, denoted σ_t, is a prefix of the (unknown) kth lexicographically smallest suffix of S. Phase $t = 0$ is the initial one: we just have the input string S and no knowledge, i.e., σ_0 is the empty string. For $t \geq 1$, a main iteration of our algorithm goes from phase $t-1$ to phase t and is termed *phase transition* ($t-1 \rightarrow t$): it is built around the tth run of BFPRT on a suitable subset of the suffixes of S. Note that $t \leq N$, since we ensure that the condition $|\sigma_{t-1}| < |\sigma_t|$ holds, namely, each phase transition extends the known prefix by at least one symbol.

[1] In the following, we will assume that the last symbol in S is an endmarker $S[N] =$ \$, smaller than any other symbol in S.

Phase transition $(0 \to 1)$. We start out with phase 0, where we run BFPRT on the individual symbols of S, and find the symbol α of rank k in S (seen as a multiset). Hence we know that $\sigma_1 = \alpha$, and this fact has some implications on the set of suffixes of S. Let s_i denote the ith suffix $S[i \dots N]$ of S, for $1 \le i \le N$, and w_i be a special prefix of s_i called `work`. We anticipate that the `work`s play a fundamental role in attaining $O(N)$ time. To complete the phase transition, we set $w_i = S[i]$ for $1 \le i \le N$, and we call *degenerate* the `work`s w_i such that $w_i \neq \alpha$. (Note that degenerate `work`s are only created in this phase transition.) We then partition the suffixes of S into two disjoint sets:

- The set of *active suffixes*, denoted by $\mathcal{A}_1$—they are those suffixes s_i such that $w_i = \sigma_1 = \alpha$.
- The set of *inactive suffixes*, denoted by $\mathcal{I}_1$ and containing the rest of the suffixes—none of them is surely the kth lexicographically smallest suffix in S.

In our example ($k = 6$), we have $\sigma_1 = \alpha =$ `b` and, for $i = 1, 2, 3, 5, 6, 7, 8, 9$, $w_i =$ `b` and $s_i \in \mathcal{A}_1$. Also, we have $s_j \in \mathcal{I}_1$ for $j = 4, 10, 11, 12$, where $w_4 = w_{10} = w_{11} =$ `a` and $w_{12} =$ \$ are degenerate `work`s.

A comment is in order at this point. We can compare any two `work`s in constant time, where the outcome of the comparison is ternary $[<, =, >]$. While this observation is straightforward for this phase transition, we will be able to extend it to longer `work`s in the subsequent transitions. Let us discuss the transition from phase 1 to phase 2 to introduce the reader to the main point of the algorithm.

Phase transition $(1 \to 2)$. If $|\mathcal{A}_1| = 1$, we are done since there is only one active suffix and this should be the kth smallest suffix in S. Otherwise, we exploit the total order on the current `work`s. Letting z_1 be the number of `work`s smaller than the current prefix σ_1, our goal becomes how to find the $(k - z_1)$th smallest suffix in $\mathcal{A}_1$. In particular, we want a longer prefix σ_2 and the new set $\mathcal{A}_2 \subseteq \mathcal{A}_1$.

To this end, we need to extend some of the `work`s of the active suffixes in $\mathcal{A}_1$. Consider a suffix $s_i \in \mathcal{A}_1$. In order to extend its `work` w_i, we introduce its *prospective* `work`. Recall that $w_i = \sigma_1 = \alpha = S[i]$. If $w_{i+1} = S[i+1] \neq \alpha$ (hence, s_{i+1} is inactive in our terminology), the prospective `work` for s_i is the concatenation $w_i w_{i+1}$, where $s_{i+1} \in \mathcal{I}_1$. Otherwise, since $w_i = w_{i+1}$ (and so $s_{i+1} \in \mathcal{A}_1$), we consider $i + 2, i + 3$, and so on, until we find the first $i + r$ such that $w_i \neq w_{i+r}$ (and so $s_{i+r} \in \mathcal{I}_1$). In the latter case, the prospective `work` for s_i is the concatenation $w_i w_{i+1} \cdots w_{i+r}$, where $w_i = w_{i+1} = \cdots = w_{i+r-1} = \sigma_1 = \alpha$ and their corresponding suffixes are active, while $w_{i+r} \neq \sigma_1$ is different and corresponds to an inactive suffix.

In any case, each prospective `work` is a sequence of `work`s of the form $\alpha^r \beta = \sigma_1^r \beta$, where $r \ge 1$ and $\beta \neq \alpha$. The reader should convince herself that any two prospective `work`s can be compared in $O(1)$ time. We exploit this fact by running BFPRT on the set $\mathcal{A}_1$ of active suffixes and, whenever BFPRT requires to compare any two $s_i, s_j \in \mathcal{A}_1$, we compare their prospective `work`s. Running time is therefore $O(|\mathcal{A}_1|)$ if we note that prospective `work`s can be easily identified by a scan of $\mathcal{A}_1$: if $w_i w_{i+1} \cdots w_{i+r}$ is the prospective `work` for s_i, then $w_{i+1} \cdots w_{i+r}$ is the prospective `work` for s_{i+1}, and so on. In other words, a consecutive run of prospective `work`s forms a *collision*, which is informally a maximal concatenated sequence of `work`s equal to σ_1 terminated by a `work` different from σ_1 (this notion will be described formally in Section 2).

After BFPRT completes its execution, we know the prospective `work` that is a prefix of the (unknown) kth suffix in S. That prospective `work` becomes σ_2 and $\mathcal{A}_2$ is made up of the the suffixes in $\mathcal{A}_1$ such that their prospective `work` equals σ_2 (and we also set z_2).

In our example, $z_1 = 3$, and so we look for the third smaller suffix in $\mathcal{A}_1$. We have the following prospective `work`s: one collision is made up of $p_1 =$ `bbba`, $p_2 =$ `bba`, and $p_3 =$ `ba`; another collision is made up of $p_5 =$ `bbbbba`, $p_6 =$ `bbbba`, $p_7 =$ `bbba`, $p_8 =$ `bba`, and $p_9 =$ `ba`. Algorithm BFPRT discovers that `bba` is the third prospective `work` among them, and so $\sigma_2 =$ `bba` and $\mathcal{A}_2 = \{s_2, s_8\}$ (and $z_2 = 5$).

How to maintain the `work`*s.* Now comes the key point in our algorithm. For each suffix $s_i \in \mathcal{A}_2$, we update its `work` to be $w_i = \sigma_2$ (whereas it was $w_i = \sigma_1$ in the previous phase transition, so it is now longer). For each suffix $s_i \in \mathcal{A}_1 - \mathcal{A}_2$, instead, we leave its `work` w_i *unchanged*. Note this is the key point: although s_i can share a longer prefix with σ_2, the algorithm BFPRT has indirectly established that s_i cannot have σ_2 as a prefix, and we just need to record a Boolean value for w_i, indicating if w_i is either lexicographically smaller or larger than σ_2. We can stick to w_i unchanged, and discard its prospective `work`, since s_i becomes inactive and is added to $\mathcal{I}_2$. In our example, $w_2 = w_8 =$ `bba`, while the other `work`s are unchanged (i.e, $w_3 =$ `b` while $p_3 =$ `ba`, $w_5 =$ `b` while $p_5 =$ `bbbbba`, and so on).

In this way, we can maintain a *total order on the* `work`*s*. If two `work`s are of equal length, we declare that they are equal according to the symbol comparisons that we have performed so far, unless they are degenerate—in the latter case they can be easily compared as single symbols. If two `work`s are of different length, say $|w_i| < |w_j|$, then s_i has been discarded by BFPRT in favor of s_j in a certain phase, so we surely know which one is smaller or larger. In other words, when we declare two `work`s to be equal, we have not yet gathered enough symbol comparisons to distinguish among their corresponding suffixes. Otherwise, we have been able to implicitly distinguish among their corresponding suffixes. In our example, $w_3 < w_2$ because they are of different length and BFPRT has established this disequality, while we declare that $w_3 = w_5$ since they have the same length. Recall that the total order on the `work`s is needed for comparing any two prospective `work`s in $O(1)$ time as we proceed in the phase transitions. The `work`s exhibit some other strong properties that we point out in the invariants described in Section 2.

Time complexity. From the above discussion, we spend $O(|\mathcal{A}_1)|)$ time for phase transition $(1 \to 2)$. We present a charging scheme to pay for that. `work`s come again into play for an amortized cost analysis. Suppose that, in phase 0, we initially assign each suffix s_i two kinds of credits to be charged as follows: $O(1)$ credits of the first kind when s_i becomes inactive, and further $O(1)$ credits of the second kind when s_i is already inactive but its `work` w_i becomes the terminator of the prospective `work` of an active suffix. Note that w_i is incapsulated by the prospective `work` of that suffix (which survives and becomes part of $\mathcal{A}_2$).

Now, when executing BFPRT on $\mathcal{A}_1$ as mentioned above, we have that at most one prospective `work` *survives* in each collision and the corresponding suffix becomes part of $\mathcal{A}_2$. We therefore charge the cost $O(|\mathcal{A}_1|)$ as follows. We take $\Theta(|\mathcal{A}_1| - |\mathcal{A}_2|)$ credits of the first kind from the $|\mathcal{A}_1| - |\mathcal{A}_2| \geq 0$ active suffixes that become inactive at the end of the phase transition. We also take $\Theta(|\mathcal{A}_2|)$ credits from the $|\mathcal{A}_2|$ inactive suffixes whose `work` terminates the prospective `work` of the survivors. In our example, the $\Theta(|\mathcal{A}_1| - |\mathcal{A}_2|)$ credits are taken from s_1, s_3, s_5, s_6, s_7, and s_9, while $\Theta(|\mathcal{A}_2|)$ credits are taken from s_4 and s_{10}.

At this point, it should be clear that, in our example, the next phase transition $(2 \to 3)$ looks for the $(k - z_2)$th smaller suffix in $\mathcal{A}_2$ by executing BFPRT in $O(|\mathcal{A}_2|)$ time on the prospective `work`s built with the runs of consecutive occurrences of the `work` $\sigma_2 =$ `bba` into S. We thus identify `bbaa`\$ (with $\sigma_3 =$ `bbaa`) as the median suffix in S.

Phase transition $(t-1 \to t)$ for $t \geq 1$. We are now ready to describe the generic phase transition $(t-1 \to t)$ more formally in terms of the active suffixes in $\mathcal{A}_{t-1}$ and the inactive ones in $\mathcal{I}_{t-1}$, where $t \geq 1$.

The input for the phase transition is the following: (a) the current prefix σ_{t-1} of the (unknown) kth lexicographically smallest suffix in S; (b) the set $\mathcal{A}_{t-1}$ of currently active suffixes; (c) the number z_{t-1} of suffixes in $\mathcal{I}_{t-1}$ whose `work` is smaller than that of the suffixes in $\mathcal{A}_{t-1}$ (hence, we have to find the $(k - z_{t-1})$th smallest suffix in $\mathcal{A}_{t-1}$); and (d) a Boolean vector whose ith element is false (resp., true) iff, for suffix $s_i \in \mathcal{I}_{t-1}$, the algorithm BFPRT has determined that its `work` w_i is smaller (resp., larger) than σ_{t-1}. The output of the phase transition are data (a)–(d) above, updated for phase t.

We now define collisions and prospective `work`s in a formal way. We say that two suffixes $s_i, s_j \in \mathcal{A}_t$ *collide* if their `work`s w_i and w_j are adjacent as substrings in S, namely, $|i - j| = |w_i| = |w_j|$. A *collision* C is the maximal subsequence $w_{l_1} w_{l_2} \cdots w_{l_r}$, such that $w_{l_1} = w_{l_2} = \cdots = w_{l_r} = \sigma_t$, where the active suffixes s_{l_f} and $s_{l_{f+1}}$ collide for any $1 \leq f < r$. For our algorithm, a *collision* can also be a degenerate sequence of just one active suffix s_i (since its `work` does not collide with that of any other active suffix).

The *prospective* `work` of a suffix $s_i \in \mathcal{A}_{t-1}$, denoted by p_i, is defined as follows. Consider the collision C to which s_i belongs. Suppose that s_i is the hth active suffix (from the left) in C, that is, $C = w_{l_1} w_{l_2} \cdots w_{l_{h-1}} w_i w_{l_{h+1}} \cdots w_{l_{r-1}} w_{l_r}$. Consider the suffix $s_u \in \mathcal{I}_{t-1}$ adjacent to w_{l_r} (because of the definition of collision, s_u must be an inactive suffix following w_{l_r}). We define the prospective `work` of s_i, to be the string $p_i = w_i w_{l_{h+1}} \cdots w_{l_{r-1}} w_{l_r} w_u$. Note that $w_i = w_{l_{h+1}} = \cdots = w_{l_{r-1}} = w_{l_r} = \sigma_{t-1}$ since their corresponding suffixes are all active, while w_u is shorter. In other words, $p_i = \sigma_{t-1}^{r-h} w_u$, with $|w_u| < |\sigma_{t-1}|$.

Lemma 2.1. *For any two suffixes $s_i, s_j \in \mathcal{A}_t$, we can compare their prospective* `work`s *p_i and p_j in $O(1)$ time.*

We now give the steps for the phase transition. Note that we can maintain $\mathcal{A}_{t-1}$ in monotone order of suffix position (i.e., $i < j$ implies that s_i comes first than s_j in $\mathcal{A}_{t-1}$).

(1) Scan the active set $\mathcal{A}_{t-1}$ and identify its collisions and the set $\mathscr{T}$ containing all the suffixes $s_u \in \mathcal{I}_{t-1}$ such that w_u immediately follows a collision. For any suffix s_i in $\mathcal{A}_{t-1}$, determine its prospective `work` p_i using the collisions and $\mathscr{T}$.

(2) Apply algorithm BFPRT to the set $\{p_i\}_{s_i \in \mathcal{A}_{t-1}}$ using the constant-time comparison as stated in Lemma 2.1. In this way, find the $(k - z_{t-1})$th lexicographically smallest prospective `work` p, and the corresponding set $\mathcal{A}_t = \{s_i \in \mathcal{A}_{t-1} \mid p_i = p\}$ of active suffixes whose prospective `work`s match p.
   (a) If $|\mathcal{A}_t| = 1$, stop the computation and return the singleton $s_i \in \mathcal{A}_t$ as the kth smallest suffix in S.
   (b) If $|\mathcal{A}_t| > 1$, set $\sigma_t = p$ (and update z_t accordingly).

(3) For each $s_i \in \mathcal{A}_t$: Let $p = w_i w_{l_{h+1}} \cdots w_{l_r} w_u$ be its prospective `work`, where $s_u \in \mathscr{T}$. Set its new `work` to be $w_i = p = \sigma_t$.

(4) For each $s_j \in \mathcal{A}_{t-1} - \mathcal{A}_t$, leave its `work` w_j *unchanged* and, as a byproduct of running BFPRT in step 2, update position j of the Boolean vector (d) given in input, so as to record the fact that w_j is lexicographically smaller or larger than σ_t.

Lemma 2.2. *Executing phase transition* $(t-1 \to t)$ *with* $t \geq 1$*, requires* $O(|\mathcal{A}_{t-1}|)$ *time in the worst case.*

Invariants for phase *t*. Before proving the correctness and the complexity of our algorithm, we need to establish some invariants that are maintained through the phase transitions. We say that w_i is *maximal* if there does not exist another suffix s_j such that w_j contains w_i, namely, such that $j < i$ and $i + |w_i| \leq j + |w_j|$. For any $t \geq 1$, the following invariants holds (where $\mathcal{A}_0$ is trivially the set of all the suffixes):

(i) [prefixes]: σ_{t-1} and σ_t are prefixes of the (unknown) kth smallest suffix of S, and $|\sigma_{t-1}| < |\sigma_t|$.
(ii) [works]: For any suffix s_i, its `work` w_i is either degenerate (a single mismatching symbol) or $w_i = \sigma_{t'}$ for a phase $t' \leq t$. Moreover, $w_i = \sigma_t$ iff $s_i \in \mathcal{A}_t$.
(iii) [comparing]: For any s_i and s_j, $|w_i| \neq |w_j|$ implies that we know whether $w_i < w_j$ or $w_i > w_j$.
(iv) [nesting]: For any two suffixes s_i and s_j, their `works` w_i and w_j do not overlap (either they are disjoint or one is contained within the other). Namely, $i > j$ implies $i + |w_i| \leq j + |w_j|$ or $i \geq j + |w_j|$.
(v) [covering]: The `works` of the active suffixes are all maximal and, together with the maximal `works` generated by the inactive suffixes, form a *non-overlapping covering* of S (i.e. $S = w_{i_1} w_{i_2} \cdots w_{i_r}$, where $i_1 < i_2 < \cdots < i_r$ and either $s_{i_j} \in \mathcal{A}_t$, or $s_{i_j} \in \mathcal{I}_t$ and w_{i_j} is maximal, for $1 \leq j \leq r$).

Lemma 2.3. *After phase transition* $(t-1 \to t)$ *with* $t \geq 1$*, the invariants (i)–(v) are maintained.*

Theorem 2.4. *The algorithm terminates in a phase* $t \leq N$*, and returns the* k*th lexicographically smallest suffix.*

Theorem 2.5. *Our suffix selection algorithm requires* $O(N)$ *time in the worst case.*

This simpler suffix selection algorithm is still cache "unfriendly". For example, it requires $O(N)$ block transfers with a string S with period length $\Theta(B)$ (if S is a prefix of g^i for some integer i, then g is a period of S).

3. Cache-Aware Sparse Suffix Selection

In *the sparse suffix selection problem*, along with the string S and the rank k of the suffix to retrieve, we are also given a set $\mathscr{K}$ of suffixes such that $|\mathscr{K}| = O(N/B)$ and the kth smallest suffix belongs in $\mathscr{K}$. We want to find the wanted suffix in $O(N/B)$ block transfers using the ideas of the algorithm described in Section 2.

Consider first a particular situation in which the suffixes are equally spaced B positions each other. We can split S into blocks of size B, so that S is conceptually a string of N/B metacharacters and each suffix starts with a metacharacters. This is a fortunate situation since we can apply the algorithm described in Section 2 as is, and solve the problem in the claimed bound. The nontrivial case is when the suffixes can be in arbitrary positions.

Hence, we revisit the algorithm described in Section 2 to make it more cache efficient. Instead of trying to extend the `work` of an active suffix s_i by just using the `works` of the following inactive suffixes, we try to batch these `works` in a sufficiently long segment, called *reach*. Intuitively, in a step similar to step 2 of the algorithm in Section 2, we could first apply the BFPRT algorithm to the set of reaches. Then, after we select a subset of equal reaches, and the corresponding subset of active suffixes, we could extend their `works` using

their reaches. This could cause collisions between the suffixes and they could be managed in a way similar to what we did in Section 2. This yields the notion of super-phase transition.

Super-phase transition. The purpose of a super-phase is to group consecutive phases together, so that we maintain the same invariants as those defined in Section 2. However, we need further concepts to describe the transition between super-phases. We number the super-phases according to the numbering of phases. We call a super-phase m if the *first* phase in it is m (in the overall numbering of phases).

Reaches, pseudo-collisions and prospective reaches. Consider a generic super-phase m. Recall that, by the invariant (v) in Section 2, the phase transitions maintain the string S partitioned into maximal `work`s. We need to define a way to access enough (but not too many) consecutive "lookahead" `work`s following each active suffix, before running the super-phase. Since some of these active suffixes will become inactive during the phases that form the super-phase, we cannot prefetch too many such `work`s (and we cannot predict which ones will be effectively needed). This idea of prefetching leads to the following notion.

For any active suffix $s_i \in \mathcal{A}_m$, the *reach* of s_i, denoted by r_i, is the *maximal* sequence of consecutive `work`s $w_{l_1} w_{l_2} \cdots w_{l_f}$ such that

(i) $i < l_1 < l_2 < \cdots < l_f$ and $l_f - l_1 < B$;
(ii) w_i and w_{l_1} are adjacent and, for $1 < x \leq f$, $w_{l_{x-1}}$ and w_{l_x} are adjacent in S;
(iii) if s_j is the leftmost active suffix in $S[i+1 \ldots N]$, then $l_f \leq j$.

We call a reach *full* if $l_f < j$ in condition (iii), namely, we do not meet an active suffix while loading the reach. Since we know how to compare two `work`s, we also know how to compare any two reaches r_i, r_j, seen as sequences of `work`s. We have the following.

Lemma 3.1. *For any two reaches r_i and r_j, such that $|r_i| < |r_j|$, we have that r_i cannot be a prefix of r_j.*

Using reaches, we must possibly handle the collisions that may occur in an arbitrary phase that is internal to the current super-phase. We therefore introduce a notion of collision for reaches that is called pseudo-collision because it does not necessarily implies a collision.

For any two reaches r_i, r_j such that $i < j$, we say that r_i and r_j *pseudo-collide* if $r_i = r_j$ and the last `work` of r_i *is* w_j itself (not just *equal to* w_j). Thus, the last `work` of r_j is active and equal to w_i and w_j. Certainly, the fact that r_i and r_j pseudo-collide during a super-phase does not necessarily imply that the `work`s w_i and w_j collide in one of its phases. A *pseudo-collision PC* (l) is a maximal sequence $r_{l_1} r_{l_2} \cdots r_{l_a}$ such that r_{l_f} and $r_{l_{f+1}}$ pseudo-collide, for any $1 \leq f < a$. For our algorithm, a degenerate pseudo-collision is a sequence of just one reach.

Let us consider an active suffix s_i and the pseudo-collision to which r_i belongs. Let us suppose that the pseudo-collision is $r_{l_1} r_{l_2} \cdots r_{l_{f-1}} r_i r_{l_{f+1}} \cdots r_{l_a}$ (i.e. r_i is the fth reach). Also, let us consider the reach r_u of the last `work` w_u that appears in r_{l_a} (by the definition of pseudo-collision, we know that the last `work` w_u of r_{l_a} is equal to its first `work`, so s_u is active and has a reach). The *prospective reach* of an active `work` w_i, denoted by pr_i, is the sequence $r_i r_{l_{f+1}} \cdots r_{l_a} tail\,(pr_i)$, where $tail\,(pr_i) = lcp\,(r_i, r_u)$ is the *tail* of pr_i and denotes the longest initial sequence of `work`s that is common to both r_i and r_u. Analogously to prospective `work`s, we can define a total order on the prospective reaches. The *multiplicity* of pr_i, denoted by $mult\,(pr_i)$, is $a - f + 1$ (that is the number of reaches following r_i in the pseudo-collision plus r_i).

Lemma 3.2. *If the invariants for the phases hold for the current super-phase then, for any two reaches r_i and r_j such that $r_i = r_j$, we have that their prospective reaches pr_i and pr_j can be compared in $O(1)$ time, provided we know the lengths of tail (pr_i) and tail (pr_j).*

Super-phase transition ($m \to m'$). The transition from a super-phase m to the next super-phase m' *emulates* what happens with phases $m, m+1, \ldots, m'$ in the algorithm of Section 2, but using $O(N/B)$ block transfers.

(1) For each active suffix s_i, we create a pointer to its reach r_i.
(2) We find the $(k - z_m)$th lexicographically smallest reach ρ using BFPRT on the $O(N/B)$ pointers to reaches created in the previous step. The sets $\mathscr{R}_= = \{s_i \mid s_i \text{ is active and } r_i = \rho\}$, $\mathscr{R}_< = \{s_i \mid s_i \text{ is active and } r_i < \rho\}$, and $\mathscr{R}_> = \{s_i \mid s_i \text{ is active and } r_i > \rho\}$ are thus identified, and, for any $s_i \in \mathscr{R}_< \cup \mathscr{R}_>$, the length of $lcp(r_i, \rho)$.[2] If $|\mathscr{R}_=| = 1$, we stop and return s_i, such that $s_i \in \mathscr{R}_=$, as the kth smallest suffix in S.
(3) For any $s_i \in \mathscr{R}_=$, we compute its prospective reach pr_i.
(4) We find the $(k - z_m - |\mathscr{R}_<|)$th lexicographically smallest prospective reach π among the ones in $\{pr_i \mid s_i \in \mathscr{R}_=\}$, thus obtaining $\mathscr{P}_= = \{s_i \mid s_i \text{ is active and } pr_i = \pi\}$, $\mathscr{P}_< = \{s_i \mid s_i \text{ is active and } pr_i < \pi\}$, $\mathscr{P}_> = \{s_i \mid s_i \text{ is active and } pr_i > \pi\}$, and, for any $s_i \in \mathscr{P}_< \cup \mathscr{P}_>$, the length of $lcp(pr_i, \pi)$. If $|\mathscr{P}_=| = 1$, we stop and return s_i, such that $s_i \in \mathscr{P}_=$, as the kth smallest suffix in S.

Theorem 3.3. *The sparse suffix selection problem can be solved using $O(N/B)$ block transfers in the worst case.*

4. Optimal Cache-Aware Suffix Selection

The approach in Sec. 3 does not work if the number of input active suffixes is $\omega(N/B)$. The process would cost $O\left(\frac{N}{B} \log B\right)$ block transfers (since it would take $\Omega(\log B)$ transitions to finally have $O(N/B)$ active suffixes left). However, if we were able to find a set $\mathscr{K}$ of $O(N/B)$ suffixes such that one of them is the kth smallest, we could solve the problem with $O(N/B)$ block transfers using the algorithm in Sec. 3. In this section we show how to compute such a set $\mathscr{K}$.

Basically, we consider all the substrings of length B of S and we select a suitable set of $p > B$ pivot substrings that are roughly evenly spaced. Then, we find the pivot that is lexicographically "closest" to the wanted k-th and one of the following two situations arises:

• We are able to infer that the kth smallest suffix is strictly between two consecutive pivots (that is its corresponding substring of B characters is strictly greater and smaller of the two pivots). In this case, we return all the $O(N/p) = O(N/B)$ suffixes that are contained between the two pivots.

• We can identify the suffixes that have the first B characters equal to those of the kth smallest suffix. We show that, in case they are still $\Omega(N/B)$ in number, they must satisfy some periodicity property, so that we can reduce them to just $O(N/B)$ with additional $O(N/B)$ block transfers.

[2]Given strings S and T, their longest common prefix $lcp(S, T)$ is longest string U such that both S and T start with U.

4.1. Finding pivots and the key suffixes

Let $p = \sqrt{\frac{M^c}{B}}$, for a suitable constant $c > 1$. We proceed with the following steps.

First. We sort the first M^c substrings of length B of S (that is substrings $S[1 \dots B]$, $S[2 \dots B+1]$,..., $S[M^c-1 \dots B+M^c-2]$, $S[M^c \dots B+M^c-1]$). Then we sort the second M^c substrings of length B and so forth until all the N positions in S have been considered. The product of this step is an array V of N pointers to the substrings of length B of S.

Second. We scan V and we collect in an array U of N/p positions the N/p pointers $V[p], V[2p], V[3p], \dots$.

Third. We (multi)-select from U the p pointers to the substrings (of length B) $b_1, \dots, b_p$ such that b_i has rank $i\frac{N}{p^2}$ among the substrings (pointed by the pointers) in U. These are the pivots we were looking for. We store the p (pointers to the) pivots in an array U'.

Fourth. We need to find the rightmost pivot b_x such that the number of substrings (of length B of S) lexicographically smaller than b_x is less than k (the rank of the wanted suffix). We cannot simply distribute all the substrings of length B according to all the p pivots in U', because it would be too costly. Instead, we proceed with the following refining strategy.

1. From the p pivots in U' we extract the group G_1 of δM equidistant pivots, where $\delta < 1$ is a suitable constant, (i.e. the pivots $b_t, b_{2t}, \dots$, where $t = \frac{p}{\delta M}$). Then, for any $b_j \in G_1$, we find out how many substrings of size B are lexicographically smaller than b_j. After that we find the rightmost pivot $b_{x_1} \in G_1$ such that the number of substrings (of length B) smaller than b_{x_1} is less than k.
2. From the $\frac{p}{\delta M}$ pivots in U' following b_{x_1} we extract the group G_2 of δM equidistant pivots. Then, for any $b_j \in G_2$, we find out how many substrings of size B are smaller than b_j. After that we find the rightmost pivot $b_{x_2} \in G_2$ such that the number of substrings smaller than b_{x_2} is less than k.

More generally:

f. Let G_f be the δM pivots in U' following $b_{x_{f-1}}$. Then, for any $b_j \in G_f$, we find out how many substrings of size B are smaller than b_j. After that we find the rightmost pivot $b_{x_f} \in G_f$ such that the number of substrings smaller than b_{x_f} is less than k.

The pivot b_{x_f} found in the last iteration is the pivot b_x we are looking for in this step.

Fifth. We scan S and compute the following two numbers: the number $n_x^<$ of substrings of length B lexicographically smaller than b_x; the number $n_x^=$ of substrings equal to b_x.

Sixth. In this step we treat the following case: $n_x^< < k \leq n_x^< + n_x^=$. More specifically, this implies that the wanted kth smallest suffix has its prefix of B characters equal to b_x. We proceed as follows. We scan S and gather in a contiguous zone R (the indexes of) the suffixes of S having their prefixes of B characters equal to b_x. In this case we have already found the key suffixes (whose indexes reside in R). Therefore the computation in this section ends here and we proceed to discard some of them (sec. 4.2).

Seventh. In this step we treat the following remaining case: $n_x^< + n_x^= < k$. In other words, in this case we know that the prefix of B characters of the wanted kth smallest suffix is (lexicographically) greater than b_x and smaller than b_{x+1}. Therefore, we scan S and gather in a contiguous zone R (the indexes of) the suffixes of S having their prefix of B characters greater than b_x and smaller than b_{x+1}. Since there are less than N/B such suffixes (see below Lemma 4.1), we have already found the set of sparse active suffixes (whose indexes reside in R) that will be processed in Sec. 3.

Lemma 4.1. *For any S and k, either the number of key suffixes found is $O(N/B)$, or their prefixes of B characters are all the same.*

Lemma 4.2. *Under the tall-cache assumption, finding the key suffixes needs $O(N/B)$ block transfers in the worst case.*

4.2. Discarding key suffixes

Finally, let us show how to reduce the number of key suffixes gathered in Sec. 4.1 to $\leq 2N/B$ so that we can pass them to the sparse suffix selection algorithm (Sec. 3). Let us assume that the number of key suffixes is greater than $2N/B$.

The indexes of the key suffixes have been previously stored in an array R. Clearly, the kth smallest suffix is among the ones in R. We also know the number $n^<$ of suffixes of S that are lexicographically smaller than each suffix in R. Finally, we know that there exists a string q of length B such that R *contains all and only the suffixes s_i such that the prefix of length B of s_i is equal to q* (i.e. R contains the indexes of all the occurrences of q in S).

To achieve our goal we exploit the possible periodicity of the string q. A string u is *a period* of a string v ($|u| \leq |v|$) if v is a prefix of u^i for some integer $i \geq 1$. *The period* of v is the smallest of its periods. We exploit the following:

Property 1 ([8])**.** If q occurs in two positions i and j of S and $0 < j - i < |q|$ then q has a period of length $j - i$.

Let u be the period of q. Since the number of suffixes in R is greater than $2N/B$, *there must be some overlapping* between the occurrences of q in S. Therefore, by Property 1, we can conclude that $|u| < |q|$. For the sake of presentation let us assume that $|q|$ *is not a multiple of* $|u|$ (the other case is analogous).

From how R has been built (by left to right scanning of S) we know that the indexes in it are in increasing order, that is $R[i] < R[i+1]$, for any i (i.e. the indexes in R follow the order, from left to right, in which the corresponding suffixes may be found in S). Let us consider a *maximal subsequence* R_i of R such that, for any $1 \leq j < |R_i|$, $R_i[j+1] - R_i[j] \leq B/2$ (i.e. the occurrence of q in S starting in position $R_i[j]$ overlaps the one starting in position $R_i[j+1]$ by at least $B/2$ positions). Clearly, any two of these subsequences of R do not overlap and hence R can be seen as the concatenation $R_1 R_2 \cdots$ of these subsequences. From the definition of the partitioning of R and from the periodicity of q we have:

Lemma 4.3. *The following statements hold:*

(i) *There are less than $2N/B$ such subsequences.*
(ii) *For any R_i, the substring $S[R_i[1] \ldots R_i[|R_i|] + B - 1]$ (the substring of S spanned by the substrings whose indexes are in R_i) has period u.*
(iii) *The substring of length B of S starting in position $R_i[|R_i|] + |u|$ (the substring starting one period-length past the rightmost member of R_i) is not equal to q.*

For any key suffix s_j, let us consider the following prefix: $ps_j = S[j \ldots R_i[|R_i|] + |u| + B - 1]$, where R_i is the subsequence of R where (the index of) s_j belongs to. By Lemma 4.3, we know two things about ps_j: (*a*) the prefix of length $|ps_j| - |u|$ of ps_j has period u; (*b*) the suffix of length B of ps_j *is not equal* to q.

In light of this, we associate with any key suffix s_j a pair of integers $\langle \alpha_j, \beta_j \rangle$ defined as follows: α_j is equal to *the number of complete periods* u in the prefix of length $|ps_j| - |u|$ of ps_j; β_j is equal to $|R_i| + |u|$ (that is the index of the substring of length B starting one period-length past the rightmost member of R_i).

There is natural total order $\lhd$ that can be defined over the key suffixes. It is based on the pairs of integers $\langle \alpha_j, \beta_j \rangle$ and it is defined as follow. For any two key suffixes $s_{j'}, s_{j''}$:

- If $\alpha_{j'} = \alpha_{j''}$ then $s_{j'}$ and $s_{j''}$ are equal (according to $\lhd$).
- If $\alpha_{j'} < \alpha_{j''}$ then $s_{j'} \lhd s_{j''}$ iff $S[\beta_{j'} \dots \beta_{j'} + B - 1]$ is lexicographically smaller than q.

By Lemma 4.3, we know that the suffix of length B of $ps_{j'}$ (which is the substring $S[\beta_{j'} \dots \beta_{j'} + B - 1]$) *is not equal* to q. Therefore the total order $\lhd$ is well defined.

We are now ready to describe the process for reducing the number of key suffixes. We proceed with the following steps.

First. By scanning S and R, we compute the pair $\langle \alpha_j, \beta_j \rangle$ for any key suffix s_j. The pairs are stored in an array (of pairs of integers) $Pairs$.

Second. We scan S and compute the array $Comp$ of N positions defined as follows: for any $1 \leq i \leq N$, $Comp[i]$ is equal to -1, 0 or 1 if $S[i \dots i + B - 1]$ is less than, equal to or greater than q, respectively (the array $Comp$ tells us what is the result of the comparison of q with any substring of size B different from it).

Third. By scanning $Pairs$ and $Comp$ at the same time, we compute the array $PComp$ of size $|Pairs|$, such that, for any l, $PComp[l] = Comp[Pairs[l].\beta]]$ (where $Pairs[l].\beta$ is the second member of the pair of integers in position l of $Pairs$).

Fourth. Using $Pairs$ and $PComp$, we select the $(k - n^<)$-th smallest key suffix s_x and all the key suffixes equal to s_x *according to the total order* $\lhd$ (where $n^<$ is the number of suffixes of S that are lexicographically smaller than each suffix in R, known since Sec. 4.1). The set of the selected key suffixes is the output of the process.

Lemma 4.4. *At the end of the discarding process, the selected key suffixes are less than* $2N/B$ *in number and the* k*th lexicographically smallest suffix is among them.*

Lemma 4.5. *The discarding process requires* $O(N/B)$ *block transfers at the worst case.*

Theorem 4.6. *The suffix selection problem for a string defined over a general alphabet can be solved using* $O(N/B)$ *block transfers in the worst case.*

References

[1] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. In *Communications of ACM*, 1988.
[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. System Sci.*, 7:448–61, 1973.
[3] David Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the ACM*, 28(3):454–461, July 1981.
[4] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE, 1997.
[5] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. 39th Annual Symp. on Foundations of Computer Science (FOCS)*. IEEE, 1998.
[6] G. Franceschini and S. Muthukrishnan. Optimal suffix selection. In *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC)*, 2007.
[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symp. on Foundations of Computer Science (FOCS 1999)*, pages 285–297. IEEE, 1999.
[8] Z. Galil. Optimal parallel algorithms for string matching. *Inf. Control*, 67(1-3):144–157, 1985.
[9] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
[10] J. Vitter. In *Algorithms and Data Structures for External Memory*, 2007.
[11] P. Weiner. Linear pattern matching algorithms. In *Foundations of Computer Science (FOCS)*, 1973.