# A Simplified Description of Child Tables for Sequence Similarity Search

Martin C. Frith and Anish M. S. Shrestha

**Abstract**—Finding related nucleotide or protein sequences is a fundamental, diverse, and incompletely-solved problem in bioinformatics. It is often tackled by seed-and-extend methods, which first find "seed" matches of diverse types, such as spaced seeds, subset seeds, or minimizers. Seeds are usually found using an index of the reference sequence(s), which stores seed positions in a suffix array or related data structure. A child table is a fundamental way to achieve fast lookup in an index, but previous descriptions have been overly complex. This paper aims to provide a more accessible description of child tables, and demonstrate their generality: they apply equally to all the above-mentioned seed types and more. We also show that child tables can be used without LCP (longest common prefix) tables, reducing the memory requirement.

**Index Terms**—Biology and genetics, indexing methods, data structures, arrays, trees

## 1 Introduction

Sequence similarity search remains a fundamental and incompletely-solved task in bioinformatics. It is also a diverse task: we may wish to compare two whole genomes (which may be closely or distantly related), align long error-prone DNA reads to a genome (or to each other), compare metagenomic DNA to a protein database (allowing for frameshifts), compare highly biased sequences such as bisulfite-converted or AT-rich malaria DNA, etc. Different sequence types have different characteristics, e.g., transition mutations (a↔g and c↔t) are often over-represented, while sequencing technologies such as PacBio and nanopore have characteristic error patterns.

A general and powerful approach to these tasks is to define a statistical model, with specific probabilities for each type of substitution (e.g., g→t), and for opening and extending deletions and insertions [1]. It is possible to incorporate per-base quality data (e.g., from fastq files) into such models, for improved accuracy [2]. In any case, we then seek sequence segment-pairs with high model-likelihood of being related. There are dynamic-programming algorithms to find such segment-pairs in an optimal manner [1], but they are too slow for large datasets, so heuristic algorithms are used.

## 2 The Seed-and-Extend Approach

The typical heuristic is seed-and-extend, whereby we first find "seeds" (simple alignments that can be found quickly), and then check whether each seed can be extended into a high-likelihood alignment. Many kinds of seed have been proposed. The simplest is exact matches of a fixed length, e.g., 7 bases (Fig. 1).

- *M. C. Frith is with the Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology, Tokyo 305-8567, Japan and with the Graduate School of Frontier Sciences, University of Tokyo, Kashiwa 113-8654, Japan and also with the AIST-Waseda University CBBD-OIL, Tokyo 169-8050, Japan. E-mail: mcfrith@edu.k.u-tokyo.ac.jp.*
- *A. M. S. Shrestha is with the Graduate School of Frontier Sciences, University of Tokyo, Kashiwa 113-8654, Japan. E-mail: anish@edu.k.u-tokyo.ac.jp.*

*Manuscript received 22 June 2017; revised 13 Jan. 2018; accepted 17 Jan. 2018. Date of publication 9 Feb. 2018; date of current version 6 Dec. 2018.*
*(Corresponding author: Martin C. Frith.)*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TCBB.2018.2796064*

### 2.1 Spaced Seeds

More sophisticated are spaced seeds, which also have fixed length, but tolerate mismatches at predefined positions, e.g., positions 3 and 5 out of 8 (Fig. 1). The choice of predefined positions is termed the *pattern*. A pattern is commonly described by a sequence of symbols, e.g., 11010111, where 0 indicates positions that tolerate mismatches and 1 indicates positions that do not. Spaced seeds are advantageous for certain types of sequence, e.g., protein-coding DNA tends to mutate at every 3rd position. Less obviously, they are often advantageous even for sequences with completely random and independent substitutions [3].

### 2.2 Subset Seeds

Subset seeds are a further generalization and improvement over spaced seeds. Subset seeds also have fixed length, but tolerate *some* mismatches at predefined positions, using reduced alphabets. For example, one position might use the reduced alphabet ag ct, meaning that a:g and c:t mismatches are allowed, but other mismatches are not (Fig. 1). This is advantageous for both nucleotides and proteins, where some substitutions are more frequent than others [4], [5].

Even better performance (sensitivity per run time) can be achieved by using multiple co-designed seed patterns, where each pattern tends to find similarities that tend to be missed by the others [6], [7].

### 2.3 Variable-Length Seeds

Fixed-length seeds deal poorly with non-uniform composition, a ubiquitous feature of biopolymers. For example, many genomes are strikingly depleted in cg dinucleotides, whereas "simple" sequences such as atatatat are over-represented, and there are many types of repeated sequence. For instance, if we compare the human and chimp genomes, each of which has $\sim 10^6$ Alu sequences, we risk an overwhelming $\sim 10^{12}$ matches. In this situation, our practical aim cannot be to find all significant similarities. Often, what is really wanted is to find a few top hits to each part of each "query" sequence. This is accomplished by "adaptive seeds", defined as follows: starting at each position in the query, use the shortest seed with $\leq m$ occurrences in the reference [8]. Here, $m$ is a tunable parameter (e.g., $m = 10$) meaning "maximum frequency".

It is possible to combine adaptive and subset seeds. To do this, we must define a variable-length pattern. One way is to tandemly repeat a fixed-length pattern (e.g., 1101 →110111011101...), and use variable-length prefixes of this repeated pattern [8], [9].

### 2.4 Sparse Seeds

Sparse seeding reduces run time and/or memory use at a cost in sensitivity. Instead of finding seed hits at all positions in the sequences, we may only consider hits starting at (say) every 2nd position in the query, or in the reference. A promising variant of this is "minimizers", where we only consider hits starting at positions that are "minima" in sliding windows of $w$ consecutive positions [10], [11]. Minima can be defined in various ways, e.g., by alphabetic order of the sequence starting at each position. The point is to use minima defined by the same criterion in both query and reference, so we achieve sparsity in both, while tending to choose matching positions.

### 2.5 Seed Summary

All these seeding approaches are orthogonal, and can be combined. For example, it is possible to use adaptive-subset-minimizer seeds, and get the combined benefit. To summarize so far, there is a wide diversity of alignment tasks and seeding schemes. The purpose of reviewing them here is to emphasize the generality of child tables (described below), which apply equally to *all* these seeding schemes.

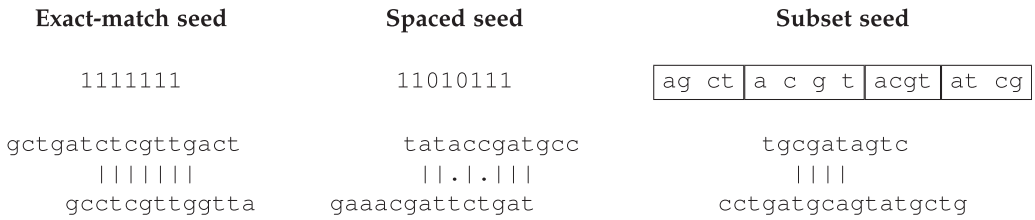


Fig. 1. Three types of "seed" for sequence similarity search. For each case, an example pattern is shown above, and an example match using this pattern is shown below. The subset seed has four (boxed) positions, where each position allows mismatches between grouped letters. For example, the first position allows `a:g` and `c:t` mismatches, but not other mismatches.

## 3 Array and Range Conventions

We shall be concerned with linear arrays of numbers (such as the position table in Fig. 2), and especially with ranges in such arrays. Let us use *in-between coordinates*, shown as –0–, –1–, etc. in Fig. 2, as if a ruler were placed along the array. This makes it very clear which array elements are encompassed by a range, such as –5– to –7–. To address individual array elements, let us use zero-based indexing, so if we have an array `a` of length `n`, its first element is `a[0]` and its last element is `a[n-1]`. We shall also denote ranges such as –5– to –7– by `a[5,7)`. The `)` indicates that `a[7]` is excluded.

## 4 Indexing

The typical way of finding seeds is to first construct an *index* of the "reference" sequences, and then scan through the queries, looking up seed matches in the index. An index fundamentally represents *positions* in the reference, to allow fast lookup of reference positions matching a seed.

An example index, for exact-match seeds of length 2, is shown in Fig. 2. Unsurprisingly, it includes a "position table", which groups all positions for each 2-mer. For example: one group contains 2 and 7, which are all the positions where `ct` occurs; another group contains 1 and 6, which are all the positions where `cc` occurs; etc. There is also a 2-mer table, which enables us to look up the part of the position table corresponding to each 2-mer. For example, given the query 2-mer `gc`, we can look up the corresponding 2-mer table entries (shown above and below `gc` in Fig. 2), 5 and 7, which indicate the start and end of a range in the position table. This range contains 0 and 5, which are indeed the positions of `gc`.

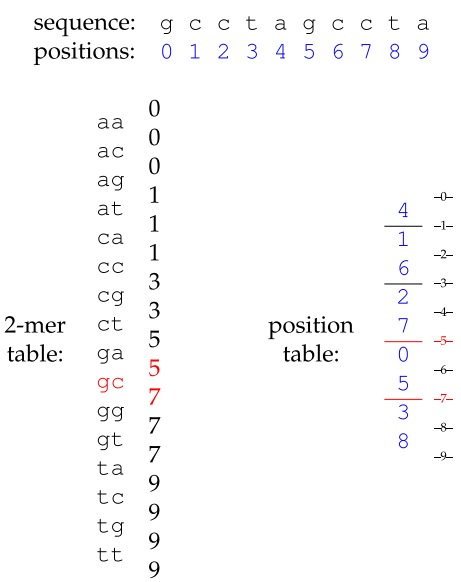


Fig. 2. Example of an index, for exact-match seeds of length 2, for the DNA sequence shown at the top. The position table groups all positions for each 2-mer: the horizontal lines indicate boundaries between the groups. Lookup of `gc` is shown in red.

This index structure can be modified straightforwardly for sparse seeds (simply omit some positions from the position table), and for spaced or subset seeds. Its main limitation is that it works only for short fixed-length seeds. The $k$-mer table has $a^k + 1$ entries, where $a$ is the alphabet size, so it consumes too much memory for larger $k$. We are now in a position to understand what a child table does: it is an alternative to the $k$-mer table that can be used for all values of $k$.

## 5 Suffix Arrays

The suffix array [12] (which has been used at least since the 1970s [13]) is a generalization of the position table from Fig. 2, for arbitrary-length seeds. It is a table of positions, sorted in alphabetical order of the sequence (i.e., suffix) starting at each position. An example is shown at the left of Fig. 3. Note that it has the same length as the reference sequence, and is similar to the position table in Fig. 2.

From here on, we will assume that the reference ends in a special delimiter character, `$`, which never occurs in any query. This delimiter, defined to be alphabetically greater than other letters, simplifies some algorithms.

### 5.1 Binary Search in Suffix Arrays

We can look up arbitrary-length seeds in a suffix array, without any $k$-mer table or child table, by binary search. Given a query $k$-mer such as `ccta`, we first compare it to the position in the middle of the suffix array, in this case position 7, which points to `cta$`. Because `ccta` is alphabetically less than `cta$`, its matching range must be in the top half of the suffix array. So we continue binary search in the top half.

Classic binary search finds a single position in a sorted array, but here we wish to find a range. This can be done by a variant of binary search: the equal range algorithm (e.g., in the C++ standard library).

Binary search is slower than lookup with a $k$-mer table. It requires $O(\log n)$ steps, where $n$ is the suffix array length. For a fixed-length $k$-mer, each step compares up to $k$ letters, for a worst case $O(k \log n)$ letter comparisons. This can be improved to $O(k + \log n)$, at a cost in memory, by supplementing the suffix array with an additional data structure [12]. In practice, however, a carefully-implemented binary search is often faster without this extra structure [14].

### 5.2 Suffix Array Generalizations

Suffix arrays can be generalized for sparse seeding and/or subset seeds.

For sparse seeding, we simply omit some positions, and sort the remaining ones alphabetically as usual (Fig. 3 middle column). Sparse suffix arrays are promising for huge datasets (hundreds of gigabytes), where a full suffix array might be unsuitable.

For subset seeding, we must define a variable-length pattern, which can again be done by tandemly repeating a fixed-length pattern. Here, the suffix array is a table of positions sorted in a *modified* alphabetical order (Fig. 3 right column) [8], [9]. In this example, the first seed position uses a purine (`a+g`) / pyrimidine (`c+t`) reduced

sequence: g c c t a g c c t a $
positions: 0 1 2 3 4 5 6 7 8 9

| suffix array | suffixes |
|---|---|
| 4 | agccta$ |
| 9 | a$ |
| 1 | cctagccta$ |
| 6 | ccta$ |
| 2 | ctagccta$ |
| 7 | cta$ |
| 0 | gcctagccta$ |
| 5 | gccta$ |
| 3 | tagccta$ |
| 8 | ta$ |

| sparse suffix array | suffixes |
|---|---|
| 4 | agccta$ |
| 6 | ccta$ |
| 2 | ctagccta$ |
| 0 | gcctagccta$ |
| 8 | ta$ |

| subset suffix array | suffixes |
|---|---|
| 0 | Rc.tRg.cYa$ |
| 5 | Rc.tR$ |
| 4 | Rg.cYa$ |
| 9 | R$ |
| 3 | Ya.cYt.$ |
| 8 | Ya$ |
| 1 | Yc.aRc.tR$ |
| 6 | Yc.a$ |
| 2 | Yt.gYc.a$ |
| 7 | Yt.$ |

Fig. 3. A standard suffix array, a sparse suffix array, and a subset suffix array, for the DNA sequence shown at the top. In each case, the suffix array is a table of positions, sorted in alphabetical order of the sequence (i.e., suffix) starting at each position. The sparse suffix array omits some positions (in this example, every 2nd position). The subset suffix array uses a *modified* alphabetical order, defined by reduced alphabets, according to a subset seed pattern. This example uses the pattern `T101`, which is considered to tandemly repeat (so it becomes `T101T101T101`...). The `T` indicates that "transition" mutations are allowed, i.e., mismatches between `a` and `g` (purines, `R`) or between `c` and `t` (pyrimidines, `Y`). The suffixes are shown in the reduced alphabets defined by this pattern: `ag`→`R` and `ct`→`Y` in `T` positions, and `acgt`→`.` in `0` positions.

alphabet, so we want all the purines to sort next to each other, and likewise all the pyrimidines. (A detail is that the reduced alphabets always leave $ as a distinct character.)

## 6 Child Tables

Child tables were introduced in [15], but their original description is overly complex. An improved variant of child tables (with a description more similar to the present one) was published in [16], and there is a textbook description for readers comfortable with rigorous computing theory [17]. The present description is aimed at informaticians without a formal computing theory background.

A child table allows us to replace binary search with "guided binary search". An example, using the same suffix array as the left column of Fig. 3, is shown in Fig. 4. Let us search for the same query $k$-mer as before, `ccta`. We start by getting the topmost element of the child table, in this case 6, which points to the location indicated by –6– in the suffix array. This is the boundary between positions starting with `c` and those starting with `g`. Because `ccta` is alphabetically less than `g`, its matching range must be before this boundary, so we continue searching in the suffix array range [0,6). Next, we get the child table element immediately above –6–, which is 2: this points to the boundary between positions starting with `a` and those starting with `c`.

In summary, searching with a child table is similar to binary search, except that the child table guides us to the key boundaries in the suffix array. Although Fig. 4 uses a standard suffix array, child tables are not specific to this case. They apply equally well to any arbitrary set of sorted strings. For example, child tables have been used with sparse suffix arrays [18].

### 6.1 LCP Arrays

Before defining child tables precisely, it helps to first describe LCP arrays. An LCP array holds the length of the longest common prefix between sequences pointed to by adjacent suffix array elements. For example, in Fig. 4, the first two suffixes are `agccta$` and `a$`, and their longest common prefix has length 1. More precisely, `LCPar-ray[i]` is the length of the longest common prefix between the sequences starting at `suffixArray[i-1]` and `suffixArray[i]`.

### 6.2 Child Table Definition

A child table points to *minima* of the LCP array, which are the "key boundaries" of the suffix array. Initially, a child table stores one number (6 in Fig. 4), which points to an LCP minimum, and defines a split (at –6–) into upper and lower intervals. For each of these two intervals, it again stores one number that points to an LCP minimum within that interval and splits it into upper and lower subintervals. This continues recursively, stopping at un-splittable length=1 intervals.

As can be seen in Fig. 4, a child table stores its entries at the lower ends of upper intervals, and at the upper ends of lower intervals. Thus, each entry is stored adjacent to the split in the middle of the parent interval. Since we only store entries for intervals of length $> 1$, there is no danger of entries over-writing each other.

We have not yet discussed tie-breaking, when an interval has more than one LCP minimum. Actually, we can break ties however we wish, to produce different child table variants. The original child table breaks ties by choosing the first minimum [15], which leads to worst-case search time proportional to the alphabet size. The newer child table of Kim et al. chooses a "middle" minimum, with search time proportional to log(alphabet size) [16].

The recursive definition of a child table just described is performed by the algorithm in Fig. 5, which should be invoked for the outermost interval like this:

```
makeChildTable(0, suffixArray.length, 0)
```

sequence: g c c t a g c c t a $
positions: 0 1 2 3 4 5 6 7 8 9

| child table | indented child table | LCP array | suffix array | suffixes |
|---|---|---|---|---|
| –0– | | | –0– | |
| 6 | 6 | | 4 | agccta$ |
| –1– | | 1 | –1– | |
| 1 | 1 | 0 | 9 | a$ |
| –2– | | | –2– | |
| 4 | 4 | 4 | 1 | cctagccta$ |
| –3– | | | –3– | |
| 3 | 3 | 1 | 6 | ccta$ |
| –4– | | | –4– | |
| 5 | 5 | 3 | 2 | ctagccta$ |
| –5– | | | –5– | |
| 2 | 2 | 0 | 7 | cta$ |
| –6– | | | –6– | |
| 8 | 8 | 5 | 0 | gcctagccta$ |
| –7– | | | –7– | |
| 7 | 7 | 0 | 5 | gccta$ |
| –8– | | | –8– | |
| 9 | 9 | 2 | 3 | tagccta$ |
| –9– | | | –9– | |
| | | | 8 | ta$ |
| | | | –10– | |

Fig. 4. Example of a child table, with a standard suffix array, for the DNA sequence shown at the top. The "indented child table" is displayed in a way that indicates its hierarchical structure: the 6 defines an initial split (at –6–) into upper and lower intervals, then the 2 and 8 split these intervals into subintervals, etc.

```
makeChildTable(beg, end, storePos):
    if end - beg < 2: return
    mid ← any i in [beg+1, end) such that
                          LCParray[i] is minimal
    childTable[storePos] ← mid
    makeChildTable(beg, mid, mid-1)
    makeChildTable(mid, end, mid)
```

Fig. 5. A simple algorithm for making a child table. It is not necessarily the most efficient way: its purpose is to define precisely what a child table is.

(We arbitrarily define the outermost interval to be a "lower" interval, so its entry is stored at the start, not the end. The opposite definition would work equally well.)

### 6.3 Relationship to Tree Data Structures

A child table is an array representation of a bifurcating tree, which recursively cuts the suffix array in two, at LCP minima. A tree that bifurcates at minima has arisen in other contexts, and is named a "Cartesian tree" [19]. It is also related to the suffix tree, an older index data structure. The difference is that a suffix tree is multifurcating: when an interval has tied LCP minima, the suffix tree splits at all of them simultaneously. A child table can be regarded as a space-efficient implementation of a suffix tree (which implements a multifurcation by several bifurcations), thus a child table-based index has been termed a "linearized suffix tree" [16].

### 6.4 Child Table Search

An algorithm to search for a query $k$-mer using a child table is presented in Fig. 6. It finds the suffix array range matching the first query letter (`depth` = 0), then the sub-range matching the next letter (`depth` = 1), and so on until it has matched the whole query.

The algorithm establishes two invariants: that `query[0..depth]` is not alphabetically earlier than the start of the current suffix array range, and not alphabetically later than the end of the current range. If ever these invariants do not hold, there are no matches and the algorithm returns `NULL`.

Given these invariants, if the positions at the start and end of the current suffix array range point to equal (`depth+1`)-mers, then we have found the range matching `query[0..depth]`, so we increment `depth` and proceed to the next query letter. Otherwise, we update the current range to the upper or lower sub-interval from the child table.

```
childSearch(query, text, suffixArray, childTable):
    depth ← 0
    beg ← 0
    end ← suffixArray.length
    storePos ← 0

  label0:
    if depth == query.length: return [beg, end)
    q ← query[depth]

    b ← text[suffixArray[beg] + depth]
    if q < b: return NULL
  label1:
    e ← text[suffixArray[end-1] + depth]
    if q > e: return NULL
  label2:
    if b == e:
        depth ← depth+1
        goto label0
    mid ← childTable[storePos]
    m ← text[suffixArray[mid] + depth]
    if q < m:
        end ← mid
        storePos ← mid-1
        goto label1
    else:
        beg ← mid
        b ← m
        storePos ← mid
        goto label2
```

Fig. 6. Algorithm for finding positions in a text sequence that match a query string, using a child table.

```
childSearch1st(query, text, suffixArray, childTable):
    depth ← 0
    beg ← 0
    end ← suffixArray.length
    storePos ← 0

  label0:
    if depth == query.length: return [beg, end)
    q ← query[depth]

    e ← text[suffixArray[end-1] + depth]
    if q > e: return NULL
  label1:
    b ← text[suffixArray[beg] + depth]
    if q < b: return NULL

    if b < e:
        mid ← childTable[storePos]
        if q > b:
            beg ← mid
            storePos ← mid
            goto label1
        end ← mid
        storePos ← mid-1
    depth ← depth+1
    goto label0
```

Fig. 7. Algorithm for finding positions in a text sequence that match a query string, using a child table that always selects the first of tied LCP minima.

To the best of our knowledge, all previously-published child table search algorithms use an LCP array. This one does not, which saves memory. (It finds the depth of each dividing point in the child table by comparing the suffixes pointed to by the start and end of the suffix array interval.)

This algorithm works for all child table variants, no matter how they break ties. If we restrict ourselves to a particular tie-breaking method, a simpler algorithm may be possible. Specifically, Fig. 7 shows an algorithm for a child table that always selects the first minimum.

### 6.5 Search Algorithm Variants

The algorithms in Figs. 6, 7 find fixed-length exact matches, but are easily generalized to other cases.

To find adaptive seeds, replace `if depth == query.length` with `if end - beg <= maxHits`. To avoid running off the end of the query, we assume here that the query ends with a unique sentinel character.

To find subset seeds, replace `query[...]` with `subset(depth, query[...])` and `text[...]` with `subset(depth, text[...])`. Here, `subset` is a lookup table which maps each letter in the original alphabet to a letter in a reduced alphabet. This mapping may vary by position, i.e., `depth`.

## 7 Remarks on Construction

Making a suffix array by naïve sorting may be slow: sorting typically uses $O(n \log n)$ comparisons, and for strings each comparison is $O(n)$. There exist $O(n)$ suffix array construction algorithms [20], [21], which can be adapted for subset seeding [9], [21]. For sparse seeding, one route is to first construct a non-sparse suffix array and then sparsify.

In practice, we have found that a general sorting method, MSD (most significant digit) radix sort, is usually fast enough [8] (specifically, American flag sort [22] accelerated by variants of Dutch flag sort [23] for small alphabets). This method's advantages are: it is simple, it generalizes straightforwardly to subset and sparse seeds, and most importantly, in all cases it uses little more memory than

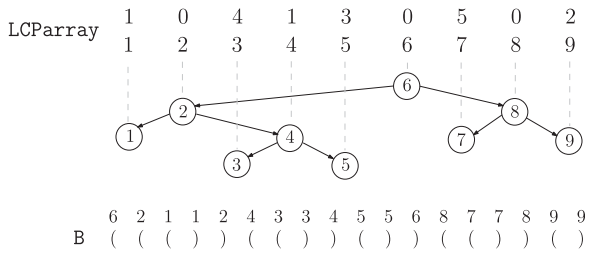


Fig. 8. A Cartesian tree (middle) of the LCP array (top), and its BP representation (bottom). Above each parenthesis is the label of the corresponding node. The labels are shown here for illustrative purposes only – they are not stored.

TABLE 1
Array, Tree, and BP Operations for Child Table Search

| Array | Tree | BP |
|---|---|---|
| `storePos ← 0` | `v ← root` | `v ← 0` |
| `mid ← childTable[storePos]` | `mid ← in-order(v)` | `w ← close(v)`<br>`mid ← rank(w)` |
| `storePos ← mid-1` | `v ← leftChild(v)` | `v ← v+1` |
| `storePos ← mid` | `v ← rightChild(v)` | `v ← w+1` |

*The child table search algorithms (Figs. 6, 7) can be modified to use the tree or BP representation, by replacing the four lines shown here.*

that needed to store the result [22]. MSD radix sort initially sorts by the first letter of each string, so that all strings starting with a are placed above those starting with c, and so on. It then sorts the strings starting with a based on their second letter, then proceeds recursively to third letters, etc.

A child table can be constructed straightforwardly during MSD radix sort. This is because each sorting phase yields LCP minima within the current interval (the boundaries between as, cs, etc).

## 8 Alternatives

### 8.1 $k$-mer / Bucket Tables

A fast suffix array lookup method should probably use some variant of the $k$-mer table from Fig. 2, which is still the fastest way to look up small $k$-mers. It cannot be used exactly as shown in Fig. 2, because suffix arrays are slightly different from position tables: suffix arrays include positions $< k$ bases before delimiters. (In practice we might have delimiters in multiple places, e.g., between concatenated sequences, or standing in for unknown or repeat-masked letters). This means that the end of (say) the `at` range is not necessarily the start of the `ca` range, as it is in Fig. 2. One possible solution is to add range endpoints to the $k$-mer table: it is possible to store the start and end of all $k$-mers for all $k \leq d$, in $(a^{d+1}-1)/(a-1)$ entries, where $a$ is the alphabet size excluding delimiters: $a = 4$ for DNA (see the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TCBB.2018.2796064).

With some such $k$-mer table (termed a "bucket table" in [15]), we can look up $k$-mers $\leq d$. For a longer $k$-mer, we can first look up its length-$d$ prefix, then perform binary search or child-table search *within the range of that prefix*.

This leads to a problem. In order to do child table search, we need to know whether the child table entry for the outermost interval is stored at the upper or lower end of that interval. But if the outermost interval comes from $k$-mer table lookup, we do not know this. Fortunately, it is easy to determine. If the entry is stored at the lower end, then the upper end must hold an ancestral interval's entry, which points to $\geq$ the end of the current interval:

```
getStorePos(childTable, beg, end):
if childTable[beg] < end: return beg
else:                     return end-1
```

### 8.2 Shrunk Child Tables

To save memory, child tables can be shrunk with a simple heuristic [15]. Instead of storing the absolute positions of LCP minima in the child table, we can store offsets to those positions. In other words:

```
childTable[i] ← |childTable[i] - i|
```

These offsets are usually small, so can be stored in fewer bytes (e.g., 1 or 2 bytes). For larger offsets, we store a dummy value (such as 0 or $-1$): when the search algorithm encounters a dummy value, it knows that the information is missing, and falls back to binary search. More complex ways of shrinking child tables were explored in [24].

### 8.3 Compact / Succinct / Compressed Indexes

There has been extensive research into compact indexes [25], some of which use compressed child tables [26], [27]. These save memory by not storing a full suffix array, at a cost in speed [28], [29], [30]. Specifically, they are fast at counting reference positions that match a query $k$-mer, but slow at retrieving those positions: it is fastest to have the positions available contiguously in a suffix array [28], [30]. "Succinct data structures should therefore only be used where memory constraints prohibit the use of traditional data structures" [29]. Even if a full suffix array would be too large, we can consider a sparse index, and/or distributing the reference sequences into separately-indexed volumes. Furthermore, these indexes usually compress standard suffix arrays, and it is unclear how effectively they can be extended to subset seeding, minimizers, etc [8], [31].

### 8.4 A Compact Child Table

A child table describes a bifurcating tree: the tree for Fig. 4 is shown more explicitly in Fig. 8. The search algorithms (Figs. 6, 7) can be expressed in terms of tree operations (Table 1). Note this requires an `in-order` operation, which returns a node's rank in left-to-right node order (i.e., its rank in the in-order traversal of the tree).

There are several ways to represent a tree with $t$ nodes using only $2t + o(t)$-bits [25], not all of which support a fast `in-order` operation. One way that does is an array `B` of balanced parentheses (BP) [25], [32]. Given a binary tree, `B` is generated recursively, starting from the root, in the following manner: write `(`; write BP representation of the subtree rooted on left child; write the matching `)`; write the BP representation of the subtree rooted on right child. Fig. 8 shows an example.

Our search algorithms can use `B` as follows. We identify a node by the position of its `(` in `B`, e.g., in Fig. 8, 0 is the root, 5 is the node labeled 4, etc. We define two basic operations: `close` takes a position containing a `(` and returns the position of its matching `)`, and `rank(p)` returns the number of occurrences of `)` in `B[0,p]`. For example in Fig. 8, `close(0)=11` and `rank(8)=4`. The in-order number of a node is the rank of its `)`. Our search algorithms can be implemented as shown in the last column in Table 1. With additional $o(n)$-bit data, both `rank` and `close` can be performed in constant time [32]. Unfortunately, `close` is complex and slow in practice [33].

A slightly different BP representation appears in [27], but it relies on an LCP array for both construction and traversal. Reconnecting to our earlier observation that an LCP array is not required for searching, an interesting direction is to explore the direct construction of BP and its usage alongside a (compact) suffix array.

### 8.5 Multiple Seed Patterns

If we wish to use multiple co-designed seed patterns, the simplest way is to use a separate index (with separate child table) for each pattern. It is possible to use just one index, however, for "neighbor

TABLE 2
Memory and Alignment Time for Various Indexing Strategies

| child table | bucket depth $d$ | memory (GB) | alignment time (s) | | CPU instructions | cache misses |
|---|---|---|---|---|---|---|
| | | | gapped | gapless | | |
| none | 23 | 9.97 | 493 | 273 | 596G | 5.64G |
| 1-byte | 23 | 11.4 | 471 | 239 | 576G | 4.81G |
| 2-byte | 23 | 12.9 | 438 | 204 | 563G | 3.15G |
| full | 23 | 15.8 | 449 | 213 | 561G | 3.14G |
| none | 24 | 11.5 | 484 | 250 | 586G | 4.69G |
| none | 26 | 14.6 | 482 | 255 | 581G | 4.73G |
| full | 0 | 14.8 | 705 | 474 | 890G | 5.92G |

*Gapless alignment was done by adding* `-j1` *to the* `lastal` *options. Each time (in seconds) is the sum of* `user` *and* `sys` *from the* `time` *command, and is the median of 5 replicates. Instructions and cache misses for gapless alignment were counted by* `perf stat`*. "G" means billion. CPU: Intel(R) Xeon(R) E5-2695 v3 @ 2.30 GHz. LAST version: 744.*

TABLE 3
Memory and Alignment Time for Sparse Indexing

| sparsity option | child table | memory (GB) | alignment time (s) | CPU instructions | cache misses |
|---|---|---|---|---|---|
| `-w16` | none | 3.99 | 493 | 1327G | 10.6G |
| | 1-byte | 4.17 | 478 | 1296G | 8.79G |
| | 2-byte | 4.36 | 414 | 1280G | 7.61G |
| | full | 4.72 | 456 | 1272G | 7.71G |
| `-W31` | none | 4.15 | 81.2 | 216G | 1.73G |
| | 1-byte | 4.37 | 72.4 | 211G | 1.33G |
| | 2-byte | 4.60 | 61.3 | 207G | 0.87G |
| | full | 5.04 | 65.4 | 206G | 0.88G |

*In these tests, the bucket depth was left at its default setting (which was 14 in all cases). LAST version: 912.*

seeds" [34]. It is also possible to compress a spaced-seed suffix array relative to a normal suffix array [31].

## 9 Tests

### 9.1 Subset Seeding for Bisulfite-Converted DNA

As an example, the child table implementation in LAST (http://last.cbrc.jp/) was used to align bisulfite-converted human DNA reads (with many `c`→`t` substitutions; the first million length=85 reads from SRR094461 [35]) to a human genome:

```
lastdb [opt] -uBISF myDB hg38_no_alt_analysis_set.fa
lastal -Q1 -s1 -e120 myDB queries.fastq > outfile
```

`lastdb` makes an index (named `myDB`) of the genome, and `lastal` aligns the queries. Option `-uBISF` specifies sparse seeding in the reference (every 2nd position) with subset seed pattern `bbbbbb0b0bb00`, where `b` positions allow `c:t` mismatches and `0` positions allow all mismatches [36]. The `[opt]` parameters were varied to try full or shrunk child tables, and different values of the bucket depth $d$ (Table 2). In each case, index construction took $<$ 17 minutes.

Here, alignment is fastest using a bucket table with $d = 23$ (the default for this index size) plus a shrunk 2-byte child table, which is slightly faster than a full child table. A 1-byte child table results in speed and memory intermediate between a 2-byte child table and no child table. Another way to trade memory for speed is to increase the bucket depth, but here this is less effective than using a child table. On the other hand, a child table without any bucket table is slow.

Of course, this is just one example, for a particular implementation, dataset, and seeding strategy (with adaptive seeds). Child tables may provide greater or lesser speed-up in other cases. In a previous test with fixed-length seeds, child tables boosted speed for short seeds [14]. This is because the run time of child table search depends on the seed length but not on the index size, whereas binary search does depend on the index size.

### 9.2 Sparse Seeding with Human DNA

Child tables combine well with sparse seeding. To show this, we used LAST to align human DNA reads (the first million length=101 reads from SRR1514950_1 [37]) to a human genome:

```
lastdb [opt] -uNEAR myDB hg38_no_alt_analysis_set.fa
lastal -f0 -Q1 -j1 myDB queries.fastq > outfile
```

The `[opt]` parameters were varied to try full or shrunk child tables, and two kinds of sparse index. First, we used `-w16`, to index every 16th position. Greater sparsity reduces the memory use, but the interesting point is that it reduces the percentage of that memory used by child tables (Table 3). This is because the memory use becomes dominated by the genome sequence rather than its index.

Next, we used `-W31`, to specify "minimizer" sparsity with sliding windows of length 31. This selects each position (in reference *and* query sequences) that is the minimum within any window, according to alphabetic order of the sequence starting at each position. Thus, it often selects positions with `a` and rarely selects positions with `t`. The interesting result here is that child tables produced a greater percentage speed-up (Table 3). E.g., the run-time of 81.2 sec was cut by 25 percent with a 2-byte child table, whereas the `-w16` run-time of 493 sec was cut by 16 percent. The likely reason is that the search tree is unbalanced: we have to search more deeply to find adaptive seeds ($\leq m$ occurrences in the index) starting with `a`. Note the bisulfite subset seeds are also unbalanced, and in that case a 2-byte child table also cut the gapless alignment time by 25 percent (Table 2).

### 9.3 CPU Cache Misses

The run time of index-lookup algorithms is often dominated by cache misses. Memory access is orders-of-magnitude slower than CPU arithmetic in modern CPUs, and CPUs try to accelerate memory access by prefetching and caching memory locations near recently-accessed locations. Unfortunately, binary search and child table search access widely-scattered elements of large arrays, which does not benefit from this caching. As the search narrows down, the accesses into the suffix array and child table become more and more localized, so we can hope they are in cache, especially if the early search steps are skipped using a bucket table. However, the accesses into the text do not become more localized.

Tables 2, 3 suggest that cache misses are indeed key. The faster run times with child tables correspond to only small reductions in CPU instruction counts, but large reductions in cache misses.

### 9.4 Cache-Friendly Layouts

If cache misses are key, we should consider cache-friendly data layouts. We can reduce scattered access into the text, at a cost in memory, by storing copies of text characters alongside the suffix array [38]. Such copied characters have been termed a "fringe" [39] or "discriminating characters" [24].

Binary search is classically performed on sorted arrays, but it can instead be performed on arrays whose elements are in a different order, e.g., the ahnentafel order used in binary heaps [40]. Such layouts can result in faster search [40], so would be intriguing to try with suffix arrays. However, we wish to not only find a suffix array range but also retrieve the elements in the range: with non-sorted layouts these elements are not contiguous, so more costly to retrieve.

## 10 Conclusion

We hope to have shown that child tables are straightforward, and very general: they apply equally to diverse sequence matching strategies, using inexact subset seeds and/or sparse seeds such as minimizers. They are a practical option to trade greater memory usage for faster sequence similarity search.

## Acknowledgments

The authors thank Paul Horton for suggesting improvements to the text. This work was supported by KAKENHI grant number 26700030.

## References

[1] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge, U.K.: Cambridge University Press, 1998.
[2] M. C. Frith, R. Wan, and P. Horton, "Incorporating sequence quality data into alignment improves DNA read mapping," *Nucleic Acids Res.*, vol. 38, no. 7, 2010, Art. no. e100.
[3] B. Ma, J. Tromp, and M. Li, "PatternHunter: Faster and more sensitive homology search," *Bioinform.*, vol. 18, no. 3, pp. 440–445, 2002.
[4] G. Kucherov, L. Noé, and M. Roytberg, "A unifying framework for seed sensitivity and its application to subset seeds," *J. Bioinform. Comput. Biol.*, vol. 4, no. 2, pp. 553–569, 2006.
[5] M. Roytberg, et al., "On subset seeds for protein alignment," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 6, no. 3, pp. 483–494, Jul. 2009.
[6] M. Li, B. Ma, D. Kisman, and J. Tromp, "PatternHunter II: Highly sensitive and fast homology search," *J. Bioinform. Comput. Biol.*, vol. 2, no. 3, pp. 417–439, 2004.
[7] M. C. Frith and L. Noé, "Improved search heuristics find 20,000 new alignments between human and mouse genomes," *Nucleic Acids Res.*, vol. 42, no. 7, 2014, Art. no. e59.
[8] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith, "Adaptive seeds tame genomic sequence comparison," *Genome Res.*, vol. 21, no. 3, pp. 487–493, 2011.
[9] P. Horton, S. M. Kiełbasa, and M. C. Frith, "DisLex: A transformation for discontiguous suffix array construction," in *Proc. Workshop Knowl. Language Learn. Bioinform.*, 2008, pp. 1–11.
[10] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2003, pp. 76–85.
[11] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinform.*, vol. 20, no. 18, pp. 3363–3369, 2004.
[12] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.
[13] J. Bentley, *Programming Pearls*. Boston, MA, USA: Addison-Wesley Professional, 1999.
[14] D. Weese, "Indices and applications in high-throughput sequencing," Ph.D. dissertation, Freie Universität Berlin, Berlin, Germany, 2013.
[15] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.
[16] D. K. Kim, M. Kim, and H. Park, "Linearized suffix tree: An efficient index data structure with the capabilities of suffix trees and suffix arrays," *Algorithmica*, vol. 52, no. 3, pp. 350–377, 2008.
[17] E. Ohlebusch, *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Berlin, Germany: Oldenbusch Verlag, 2013.
[18] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt, "essaMEM: Finding maximal exact matches using enhanced sparse suffix arrays," *Bioinform.*, vol. 29, no. 6, pp. 802–804, 2013.
[19] J. Vuillemin, "A unifying look at data structures," *Commun. ACM*, vol. 23, no. 4, pp. 229–239, 1980.
[20] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Comput. Surv.*, vol. 39, no. 2, 2007, Art. no. 4.
[21] A. M. Shrestha, M. C. Frith, and P. Horton, "A bioinformatician's guide to the forefront of suffix array construction algorithms," *Brief. Bioinform.*, vol. 15, no. 2, pp. 138–154, 2014.
[22] P. M. McIlroy, K. Bostic, and M. D. McIlroy, "Engineering radix sort," *Comput. Syst.*, vol. 6, no. 1, pp. 5–27, 1993.
[23] E. W. Dijkstra, *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall, 1976.
[24] T. D. Wu, "Bitpacking techniques for indexing genomes: II. Enhanced suffix arrays," *Algorithms Mol. Biol.*, vol. 11, 2016, Art. no. 9.
[25] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge, U.K.: Cambridge University Press, 2016.
[26] J. Fischer and V. Heun, "A new succinct representation of RMQ-information and improvements in the enhanced suffix array," in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Berlin, Germany: Springer, 2007, pp. 459–470.
[27] E. Ohlebusch and S. Gog, "A compressed enhanced suffix array supporting fast string matching," in *String Processing and Information Retrieval*. Berlin, Germany: Springer, 2009, pp. 51–62.
[28] P. Ferragina, R. González, G. Navarro, and R. Venturini, "Compressed text indexes: From theory to practice," *J. Exp. Algorithmics*, vol. 13, 2009, Art. no. 12.
[29] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Softw.: Practice Experience*, vol. 44, no. 11, pp. 1287–1314, 2014.
[30] S. Gog, G. Navarro, and M. Petri, "Improved and extended locating functionality on compressed suffix arrays," *J. Discrete Algorithms*, vol. 32, pp. 53–63, 2015.
[31] T. Gagie, G. Manzini, and D. Valenzuela, "Compressed spaced suffix arrays," *Math. Comput. Sci.*, vol. 11, no. 2, pp. 151–157, 2017.
[32] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," *SIAM J. Comput.*, vol. 31, no. 3, pp. 762–776, 2001.
[33] J. Cordova and G. Navarro, "Simple and efficient fully-functional succinct trees," *Theoretical Comput. Sci.*, vol. 656, pp. 135–145, 2016.
[34] M. Csűrös and B. Ma, "Rapid homology search with neighbor seeds," *Algorithmica*, vol. 48, no. 2, pp. 187–202, 2007.
[35] R. Lister, et al., "Hotspots of aberrant epigenomic reprogramming in human induced pluripotent stem cells," *Nature*, vol. 471, no. 7336, pp. 68–73, 2011.
[36] M. C. Frith, R. Mori, and K. Asai, "A mostly traditional approach improves alignment of bisulfite-converted DNA," *Nucleic Acids Res.*, vol. 40, no. 13, 2012, Art. no. e100.
[37] M. J. Chaisson, et al., "Resolving the complexity of the human genome using single-molecule sequencing," *Nature*, vol. 517, no. 7536, pp. 608–611, Jan. 2015.
[38] L. Colussi and A. De Col, "A time and space efficient data structure for string searching on large texts," *Inform. Process. Lett.*, vol. 58, no. 5, pp. 217–222, 1996.
[39] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin, "Improving suffix array locality for fast pattern matching on disk," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2008, pp. 661–672.
[40] P.-V. Khuong and P. Morin, "Array layouts for comparison-based searching," *J. Exp. Algorithmics*, vol. 22, no. 1, pp. 1–3, 2017.