Appendix A Efficient Construction of k-ary Search Tree

1. Structure of k-ary Search Tree

For a perfect tree derived from sorted array with the size of n, we assume that $n = k^H - 1$:

$$\underbrace{k-1}_{\text{level 1}} + \underbrace{(k-1)\times k}_{\text{level 2}} + \underbrace{(k-1)\times k^2}_{\text{level 3}} + \cdots + \underbrace{(k-1)\times k^{H-1}}_{\text{level } H} = k^H - 1$$

So that the tree has a height of H, at depth d (counting from 0) there are k^d nodes and $k^d(k-1)$ entries. To convert a sorted array into a perfect k-ary tree, only one node is generated from the current range. The entries inside a node are extracted from the $k-1$ *separators* that partition the current range into k equally-sized sets. The partitioning procedure initiates from the whole array to build the root node and rolls on to its subranges. Meanwhile, the tree is also materialized as an array using a level-order traversal manner without explicit storing any tree pointers, for saving space overhead [3]. These tree structures are shown in Fig. 1.

Schlegel et al. [1] introduced an algorithm to sequentially convert sorted array into search tree. The sorted-array index i is mapped to linearized-tree index $f_H(i)$ by calculating its related depth $d_H(i) = \sum_{x=1}^{H-1} \text{sgn}(i \mod k^{H-x})$ and offset $o_H(i) = \lfloor \frac{k-1}{k} \cdot \frac{i}{k^{H-d_H(i)-1}} \rfloor$. Then, we have

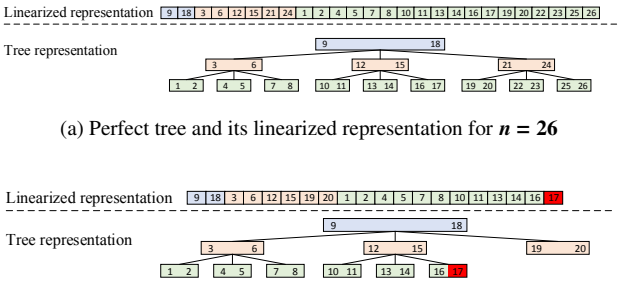
$$f_H(i) = k^{d_H(i)} + o_H(i) \tag{1}$$

However, the length of the input array may not suffice to form a perfect tree, extra operations are introduced to construct a complete tree. They referred to the largest entry at the leaf level as *fringe entry*, also note that it resides at position $n-1$ in the linearized representation of the search tree. Then, all entries which are smaller than the fringe entry have the same position in both the complete tree and the corresponding perfect tree. For those entries larger than it (reside at depth smaller than $H-1$), their position is identical to those in the perfect tree of height $H-1$. By calculating the sorted-array index of the fringe entry $f_H^*(n)$, the rest positions are given by

$$g_n(i) = \begin{cases} f_H(i) & i \leqslant f_H^*(n) \\ f_{H-1}(i - o_H^*(n) - 1) & \text{otherwise} \end{cases} \tag{2}$$

2. Pseudo Code of the Proposed Method

In Algorithm 1, we present pseudo code for the conversion into a perfect tree. And the array $PERFECT\_SIZES$ used in line 2 is the precomputed lookup-table.



(a) Perfect tree and its linearized representation for $\boldsymbol{n = 26}$

(b) Complete tree and its linearized representation for $\boldsymbol{n = 20}$, the entry 17 shaded in red is the fringe entry

Fig. 1: Structure of perfect tree and complete tree for $\boldsymbol{k = 3}$

In the previous work, the conversion for a complete tree requires to locate the largest leaf entry first, then computes other entries before or after it in the array differently, as shown in Equation 2. Rather than assigning a fringe entry to calculate its anterior and posterior nodes respectively, we notice a complete tree can always be split into three parts: the left and right parts are perfect trees, and the middle m-th subtree is a complete tree of smaller height. To reuse the conversion method in Algorithm 1, we only need to determine m and then generate perfect subtrees from both sides. Similarly, the m-th complete subtree will be disassembled recursively till we reach the leaf level, where the subtrees of height $h = 1$ have only one node, are perfect trees by definition. The last node of leaf level may not be fully loaded (number of entries less than $k-1$), but this situation can be handled separately.

Algorithm 1: REORDER

Input: pointer to the start of input array p_1,
rank r, depth d, height h,
pointer to the start of the output array p_2.
Output: array of linearized k-ary tree A.

```
1 for i = 0 to k − 2 do
2     p2[PERFECT_SIZES[d] + (k − 1) ∗ r + i] =
        p1[PERFECT_SIZES[h − 1] ∗ (i + 1) + i];
3 if − − h then
4     for i = 0 to k − 1 do
5         REORDER(p1 + i∗(PERFECT_SIZES[h] +
            1), r ∗ k + i, d + 1, h, p2);
```

An illustration is shown in Figure 1b. The complete tree is composed of three subtrees, and all of them are perfect trees. The first two subtrees are of height $h = 2$ and the last one is actually a single node.

It remains how to compute m for each level $d \in [0, H-1]$, denoted as m_d. The height H of a complete tree is defined

as $H = \lceil \log_k(n+1) \rceil$, so the leaf level has $e = n - k^{H-1}$ remaining elements. Then m_d is the number of subtrees of the root node which has full nodes, for these subtrees there are $k^{H-d-1}(k-1)$ elements, so we have $m_d = \lfloor e/[k^{H-d-1}(k-1)] \rfloor$. Similarly, we climb down the m_d-th complete tree by setting $e = e\%(k^{H-d-1} \cdot (k-1))$ and $d = d+1$.

Since a complete tree has insufficient size, the size of its subtrees also varies and the separators for a node may not scatter equidistantly in the sorted array. In order to gather these separators, the partitioning procedure always starts from the subranges on both sides, whose lengths are consistent with the size of perfect trees, then the remaining part as a complete tree.

In Algorithm 2 we present pseudo code for the conversion into a complete tree. The code is generally partitioned into three parts as described before. The second part is shown in line 5-12, which is used to generate the m-th subtree, might be skipped when the subtree becomes a perfect tree. Because when we climb down the complete tree, the fringe entry will be shifted left as the tree gets smaller and finally be eliminated from the leaf level. Thus, we would have one more subtree in the third part and that is the reason we use line 14 to update m. Still, we can preprocess the size of perfect trees and the number of nodes in each level into lookup tables to reduce real-time calculations for m and e.

Before the conversion starts, we need first to calculate the height h and the number of remaining elements e to decide which algorithm to invoke. And a prerequisite for Algorithm 2 is $h \geqslant 2$. If we get $h = 1$, there would be only one root node in the tree and the conversion becomes unnecessary, because the elements are identically arranged in the node as they are in the array.

Hitherto we have present the algorithms to convert lists of any length to a k-ary search tree. In search engines or databases, most inverted lists contain thousands of elements. No matter mapping from sorted array index to unsorted tree index, or vice versa, long-distance prefetch is inevitable during the conversion, due to the large node fanout. And a slew of cache misses will take up the majority of construction time. In view of efficiency and simplicity, a solution without explicit calculation is more preferable. More importantly, the leaf nodes are actually contiguous slices clipped from the array, because the last separators have been extracted and there only leaves slices of $k-1$ consecutive elements for the leaf level. And they accounts for more than half of the tree, for a perfect tree with $n = k^H - 1$ entries, its leaf nodes are exactly $(n+1) \cdot \frac{k-1}{k}/(k-1) = k^{H-1}$.

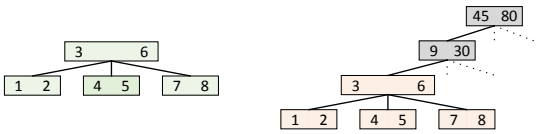
Thus, we can simply use a sequential copy operation to

Algorithm 2: REORDER_WITH_REMAINDER

Input: pointer to the start of input array p_1,
size of the array n,
number of elements in the leaf level e,
rank r, depth d, height h,
pointer to the start of the output array p_2.
Output: array of linearized k-ary tree A.

```
1  m = e/(k^{h-2} * (k-1));
   /* 1. the left sibling nodes are
      perfect trees with height of h-1.
      */
2  for i = 0 to k-2 do
3      REORDER(p1 + i*(PERFECT_SIZES[h-1]+1), r*k+i, d+1, h-1, p2);
4      p2[PERFECT_SIZES[d] + (k-1)*r + i] =
         p1[PERFECT_SIZES[h-1]*(i+1) + i];
   /* 2. the m-th subtree is one
      complete tree with height of h-1
      and it only exists when following
      condition is satisfied.          */
5  if e%(k^{h-1}*(k-1)/k) then
6      next_e=e%(k^{h-1}*(k-1)/k);
7      if h == 2 then
8          for i = 0 to next_e - 1 do
9              p2[n - next_e + i] =
                 p1[PERFECT_SIZES[1]*m + m + i];
10     else
11         REORDER_WITH_REMAINDER(p1 +
             m*(PERFECT_SIZES[h-1]+1), n, next_e, r*k+m, d+1, h-1, p2);
12     m++;
13 p1 = p1 + PERFECT_SIZES[h-2]*m + m - 1 + e;
   /* 3. the right sibling subtrees
      are perfect trees with height of
      h-2.                             */
14 m--;
15 for i = m to k-1 do
16     p2[PERFECT_SIZES[d] + r*(k-1) + i] =
         p1[(i-m)*(PERFECT_SIZES[h-2]+1)];
17     if h! = 2 then
18         REORDER(p1 + (i-m)*
             (PERFECT_SIZES[h-2]+1)+1, r*k+
             i+1, d+1, h-2, p2);
```

(a) The way *skip* works. After searching entry 3 and 13 of the left tree, we would know there are no entries smaller than 5 or larger than 12. The entries shaded can be droped without validation.



(b) The way *narrow* works. Visually, all the potential matches reside in the bottom-left subtree of the right tree, nodes in grey are trivial when intersecting with the left tree.

Fig. 2: Illustration of early termination techniques used in the intersection of ***k***-ary tree

implement such procedure, or to be more efficient, use the SIMD instructions to load and move these consecutive integers in parallel. Since the register size varies on different hardware (from 128 to 512 bits), given an integer size of 32-bit, the node capacity is recommended to be coherent with register size. Thus, we can expect to fully utilize the data loaded and avoid cache misses (say, $k = 5$ for 128 bits and $k = 17$ for 512 bits). Otherwise, we will have to mask out invalid ones for small k and concatenate operands for large k.

Appendix B Early termination techniques

1. Pseudo Code of Skip

We illustrate the two early termination techniques ***skip*** and ***narrow*** in Fig. 2. Each time the tree on the left offers an entry to be searched on another one, in Figure 2a, given an entry 3, the pointer would reach the leftmost leaf node with no match encountered yet, then the left subtree of 3, which contains 1 and 2, could be dropped immediately. Similarly, when searching for 13 the pointer would reach the rightmost leaf node, then its right sibling entry and node (14 and {16,17} respectively), even its parent entry 15 could be dropped safely. In Figure 2b, we can observe that all the matches lie in the bottom-left subtree, and restrict the search procedure within it. Then we can avoid unnecessary comparisons starting from the root node.

Algorithm 3 shows how ***skip*** works, we postulate entries are fetched from p_1 and searched on p_2. As shown in line 4, the search range for the current left subtree is marked as empty when the newly found entry resides at the first entry of the left sentry. Similarly, the search ranges for the right sibling entry and subtree will become empty when it resides at the last entry of the right sentry. By discreetly looking into each entry of the node, we can ensure all the entries are strictly considered before exclusion. Next the algorithm searches the whole tree p_2 to find a match, and adjust the related search ranges according to the returned position. As shown in line 10, we fetch entries via **`sequential`**, and this would pose a tough challenge to store these search ranges. For the right sibling, its range can be squeezed instantly as the last search update its left sentry.However, the subtrees have to wait until the intersection procedure moves into them, so their search range can get updated. Since the subtrees spread out with depths, a large lookup-table is inevitable to store these sentry nodes. In the experiment part, we will show that **`sequential`** is a not a good choice to fetch entries, **`hierarchical`** offers a compacter boundary for each entry to be searched, thus reinforcing the functionality of ***skip***.

Algorithm 3: Intersection Algorithm for k-ary Search Tree

Input: Pointers p_1 and p_2 to the two arrays of intersecting trees , pointer p to the output array which stores all the intersection.
Output: An ordered list of answer A.

```
1  initialize the search range (d_l, r_l) and (d_r, r_r);
2  set the target to be searched ϵ = *p_1;
3  while ϵ is not EOF do
4      if the search range is empty then
5          return;
6      SIMD search ϵ starting from the root node of p_2,
         and returns an entry ϵ° which is not less than ϵ
         and its corresponding position (d', r');
7      if ϵ° == ϵ then
8          *p++ = ϵ;
9      adjust the search range for the left subtree, right
         subtree and right sibling of ϵ;
10     ϵ = *(++p_1);
```

Supporting complete tree. ***Skip*** is independent of the integrality of the leaf level, whether it is a perfect tree or complete tree, our method will be able to eliminate invalid entries in the intersecting tree. When the searcher tree becomes a complete tree, the right sentry node is initialized as the rightmost node of the last but one level rather the leaf level, and the left sentry node remains the leftmost leaf node. Thus, we can guarantee no nodes are ignored when ***skip*** determines the search range for each entry. Since the tree is materialized

as an array, we have deduced how to calculate the offset of the subnode given the offset of its parent node in Appendix A. The search algorithm can correctly stop when it exceeds the boundary of the leaf level no matter how many nodes remain. Also, ***skip*** determines whether the search range becomes empty by calculating the distance between the sentry nodes in the array. It has nothing to do with the type of tree, the sentry nodes are allowed to stay at different depths. Note that the deeper a node resides at the tree, the larger its offset in the linearized array.

2. Pseudo Code of Narrow

A trivial way to find the LCA for any two nodes is to trace back along the tree until a common node is reached. Simple as it is, however, its runtime performance always hinders the overall efficiency due to superfluous loops and branches. An efficient solution resorts to reducing LCA problem to Range Minimum Query (RMQ). RMQ is used on arrays to find the position of an element with the minimum value between two specified indices. It first builds a depth-first Euler Tour on the search tree, and generates an array E recording the index of each node in its linearized order. In E, consecutive array elements differ by exactly 1 and the length of E is exactly $\lceil 2n/(k-1) \rceil$, as each node will be visited twice. Along with it we need another array of the same length, L, denoting depths of the corresponding nodes visited in E, and an array H of length n, where $H[i]$ is the index of the first occurrence of node i in E.

Thus, given any pair of nodes with their positions v and w, we first lookup their first occurrence in H, then find the minimum in L between the range $H[v]$ and $H[w]$, lastly we return the corresponding value in E which has the same index with that in L. Namely, we get $LCA(v, w) = E[RMQ_L(H[v], H[w])]$, and RMQ_L returns the index of the minimal depth in L given two indexes. A trivial lookup using these three arrays will lead to an $O(n)$ time complexity. To improve the efficiency, a sparse table is generated, which is used to preprocess RMQ_L for subarrays of length 2^k using dynamic programming. The table boils down to an array $M[0, n-1][0, \log n]$ where $M[i][j]$ is the index of the minimum depth in the sub array of L starting at i having length 2^j. And given a range of any distance, say $[i, j]$, we can split it into two contiguous subranges which might overlap in the middle, and the minimum is elected from the smaller one offered by the two ranges. Let $k = \lfloor \log(j - i - 1) \rfloor$, then we have

$$RMQ_L(i, j) = \begin{cases} M[i][k] & M[i][k] \leqslant M[j - 2^k + 1][k] \\ M[j - 2^k + 1][k] & otherwise \end{cases}$$

Thus, we are able to calculate RMQ_L in O(1) time.

With *narrow* the average comparisons drop from H to $H/2$ for each entry in the searchee tree. Note all the arrays can be built simultaneously when converting a tree. Although its preprocessing adds burden to the construction time and space occupancy at the same time, query processing is more imperative than storage, and the efficiency achieved is more evident than the space occupied. Besides, the solution of LCA we use here is a preliminary one, there are more efficient and complicated algorithms for this problem.

Algorithm 3 only needs a little modification: in line 6, we first get the LCA according to the two entry nodes, then the SIMD search begins from the LCA rather than the root node, and the rest stays the same. Figure 1 shows how *narrow* and *skip* collaborate to calculate the intersection. Given an entry and its two sentry nodes, the algorithm locates the LCA the rest works within it. After the search ends, the range is split apart and two new LCAs are spotted.

Supporting complete tree. Even for a perfect tree, the search range cannot always split evenly given two symmetric sentry nodes, and the splitter node would fall into any slot of the tree. As a classic problem, LCA is widely studied in various kinds of tree structures. Given two nodes regardless of their depths, there is always an Euler tour to connect both of them. And the LCA remains in the shortest path which spans the fewest nodes, the node in the shallowest level is right the LCA for the given nodes. In view of the above, *narrow* is also compatible with this tree structure.

3. Complexity Analysis

In terms of the construction complexity, the original method involves much more arithmetic than the proposed method in this manuscript. Given the formulae by Schlegel et al., the mapping procedure comprises of depth and offset calculation, where depth takes $O(H) = O(\log_k n)$ time and offset takes $O(1)$ time. Since the conversion sequentially traverses the whole array only once, the overall time complexity is

$$n(O(\log_k n) + O(1)) = O(n \log_k n)$$

If the size of the array is inadequate to build a perfect tree, the time complexity will double due to the branch judgement, so the time complexity is $O(n \log_k n)$.

In our method, the calculation is replaced by recursion. According to Algorithm 1, all the calculations are table

lookup and constant manipulation, which can be done in $O(1)$ time. And the conversion also traverses the array once, so the time complexity is $O(n)$.

As for the intersection complexity, **`sequential`** without optimization is quite clear, suppose the sizes of two intersecting tree are n_1 and n_2, respectively. Then the time complexity is $O(n_1 \log_k n_2)$.

However, time complexity for our two early termination techniques is more complicated to compute. For ***skip***, it is hard to predict how many nodes are pruned, because it is highly dependent on the distribution of the possible matches and the selectivity of two intersecting trees. In our experiment, the search pruned by single ***skip*** is quite limited, as we have discussed in Fig 8, the efficiency introduced is insignificant.

For ***narrow***, it reduce the search times for each entry on the searchee tree. And the average search time drops from $\log_k n_2$ to $\frac{\log_k n_2}{2}$. To be more specific, we assume each level on the searchee tree has its corresponding LCA one level deeper than its parent level, then we have

$$T(0) = (k-1)\log_k n_2 \qquad \text{(level 0)}$$
$$T(1) = k(k-1)(\log_k n_2 - 1) \qquad \text{(level 1)}$$
$$\cdots$$
$$T(i) = k^i(k-1)(\log_k n_2 - i) \qquad \text{(level i)}$$

Here, $T(i)$ indicates the total number of searches performed for each level of the searchee tree, namely, the number of nodes multiplies the average searches of its level. Note $i = 0, 1, 2, \ldots, \lceil \log_k(n_1+1) - 1 \rceil$. So the total number of searches will be sum of $T(i)$:

$$\begin{aligned} S(n) &= \sum_{i=0}^{\lceil \log_k(n_1+1)-1 \rceil} T(i) \\ &= \sum_{i=0}^{\lceil \log_k(n_1+1)-1 \rceil} k^i(k-1)(\log_k n_2 - i) \end{aligned}$$

For readability, we set $h = \lceil \log_k(n_1+1) - 1 \rceil$, then

$$\begin{aligned} S(n) &= \sum_{i=0}^{h} k^i(k-1)(\log_k n_2 - i) \\ &= \log_k n_2(k^{h+1} - 1) - (hk^{h+1} - \frac{k^{h+1}-2}{k-1}) \end{aligned}$$

So the total time complexity would be

$$\begin{aligned} &O(\log_k n_2(k^{h+1} - 1) - (hk^{h+1} - \frac{k^{h+1}-2}{k-1})) \\ \approx &O(n_1 \log_k n_2 - \log_k n_2 - n_1 \log_k n_1 + \frac{n_1}{k-1}) \\ = &O(n_1 \log_k \frac{n_2}{n_1}) \end{aligned}$$

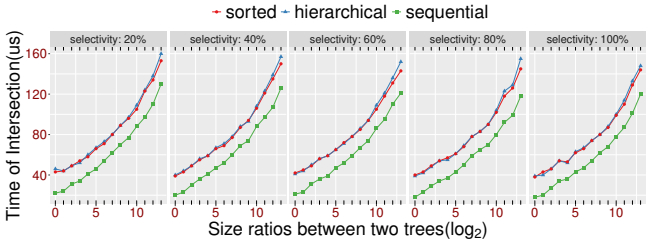


Fig. 3: Intersection time on k-ary search tree via three traversal algorithms for different size ratios and selectivities, fixing the smaller size to 1024.

So far, we have analyzed the time complexity of our construction and intersection algorithm, which are theoretically superior to the original ones. However, on realistic data, the result may not strictly follow the above computation. Since the data distribution varies considerably from time to time, the results are more like an empirical estimation. More importantly, provided that modern hardware uses superscalar pipeline and multilevel cache technologies, the cache misses and branch mispredictions are more expensive than the above arithmetic. So it is essential to design architecture-sensitive algorithms than pursuing lower complexity bound.

Appendix C supplementary experiment

1. Traversal

Before evaluating the early termination techniques, we first examine the intersection via three traversal methods without optimizations. As our goal is to find out the best combination of traversal and early termination techniques, we fix $k = 3$ for the trees to be examined, namely, we use 64-bit registers for SIMD instruction. The performance gaps caused by varying node sizes will be discussed in the overall comparison part.

As shown in Figure 3, **`sequential`** outperforms the other two methods by a considerable margin in all cases, resulting in a 25% ~75% less time. Note the abscissa represents the size ratios between the two intersecting trees, we fix the smaller size to 1024 and keeps doubling the size of another to observe the differences brought. Undoubtedly, the time grows as we prolong the size, however, the plot presents an approximately linear relationship between the time and size ratio. When the size gap grows, all three methods slow down at the same pace, the time discrepancy between **`sequential`** and the other two barely changes. Since these three traversal methods eliminate no entries in the searchee tree, the intersection efficiency of these three methods mainly depends on

the size of the searchee tree. Given an entry fetched from the searchee tree, the search procedure always starts from the root of larger one, and its average time complexity is $O(\log_k n)$, given the number of entries to be searched, which is fixed to m (size of the searchee tree), then the time complexity for intersection would be $O(m \cdot \log_k n)$. Thus, the time grows in accordance with n. The gaps between these methods are mainly caused by the efficiencies they fetch entries. It is conspicuous **`sequential`** triggers the fewest cache misses while the other two fetch entries discreetly after a series of calculations. Also observe **`hierarchical`** and **`sorted`** hardly differ from each other, which indicates the calculations are equivalent between them. Though not displayed, the time costs grow relatively when the smaller size is enlarged. We omit these results as they are visually similar. The facets show how performances vary with the selectivity, as expected, these plots almost stay the same. This can be explained by the fact that the intersection works without any early termination optimizations. The algorithm exhaustively traverses the whole trees and the only thing that affects the efficiency is the sizes involved.

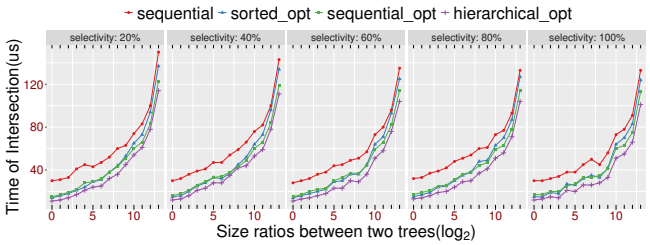


Fig. 4: Intersection time of the three traversal algorithms augmented with early termination optimizations when varying size ratios and selectivities, fixing the smaller size to 1024.

2. Early Termination

Next, we introduce the two early termination optimizations to these three traversal methods and validate their improvements for the intersection efficiency. As shown in Figure 4, we retain **`sequential`**, which is the previous best one, as a benchmark, and the optimized methods are marked as "_opt". These optimizations display a constant superiority against the original ones, they further reduce the intersection time by around 20%. It turns out that the gaps between the original methods and the optimized ones become larger when the size ratio grows. These three optimized methods hardly differ from each other, and among them, **`hierarchical_opt`** becomes the most efficient one, it even marginally outperforms **`sequential_opt`** in the experiments. Surprisingly, the intersection time is not sensitive to selectivity. As the selectivity grows from 20% to 100%, the time slightly drops a little. We were expecting a small selectivity can lead to an earlier termination with the help of **`skip`**, however, necessary comparisons are still inevitable to confirm all the possible match.

Table 1 shows the the heft of the proposed optimizations to these intersection methods on GOV2, we separate the performances of ***skip*** and ***narrow***, and "none" represents the original methods without optimizations. The blank in **`sorted`** is due to it is a one-sided traversal, ***skip*** is not compatible.

We can observe that both ***skip*** and ***narrow*** contributes to the efficiency improvement, when anyone is removed, the intersection time begins to rise. As can be seen, ***narrow*** is more appreciable than ***skip***, which is also coherent with the results in Figure 4. The optimizations remain insensitive to selectivity while showing better efficiency for larger size ratios. These differences can be attributed to the reason that ***skip*** cuts off subtrees in the searchee tree, while ***narrow*** works on the searcher tree to maintain the LCA for each search operation. Namely, reducing the length of the searcher tree is more important than its counterpart. Also observe that the average time remains less than 100us, which corresponds to the fact that half of the list lengths in GOV2 is less than 2^{10}. Optimizations can always gain a fairly good result by focusing on shortlists.

We also show the results of optimizations on synthetic data in Figure 5. The methods we compare are **`sequential`** and **`hierarchical`** with either or both optimizations. We also add the size of the searchee tree as one parameter, varying from 2^{10} to 2^{13}, to observe the differences caused. For better visualization we omit **`sorted`** due to its inferiority. The searches that occurred on the two intersecting trees are also displayed via different colors.

As stated before, the intersection procedure always lookup entries from the searchee tree on the searcher tree, so the searches performed on the searchee tree are n_{small} in one pass, and grow n_{small} times larger on another tree. A large proportion of the searches is composed by those on the searcher tree. For intersection without early terminations, searches on either tree become fixed numbers, namely n_{small} and $n_{small} \cdot d_{large}$. From Figure 5 we can easily observe the efficiency gaps caused by these optimizations. First of all, the search numbers grow linearly with the size of searchee tree, the shape among different rows barely changes. **`hierarchical`** outperforms **`sequential`** by a slight margin. For both intersection algorithms, ***skip*** performances

Table 1: Average intersection time in microseconds of three traversals with or without optimizations on GOV2.

Method	none	only ***skip***	only ***narrow***	both
hierarchical	73.34	57.54(-21.5%)	41.13(-43.9%)	39.39(-46.3%)
sequential	84.14	77.20(-8.2%)	63.44(-24.6%)	56.57(-32.8%)
sorted	86.47	-	63.53(-26.5%)	-

Fig. 5: Comparisons used in these methods.

twice searches than ***narrow***, and ***opt*** further reduces the number. We can see searches on the searchee tree nearly stay the same in all the results, the blue lines in each facets denote the size of the searchee tree, which are nearly overlapped with the boundaries between different colors. The searches reduced by ***skip*** are two orders of magnitudes lower than that of ***narrow***, the main reason is the clustered distribution of integers spread the matches all over the intersecting trees. By looking into the nodes, we find ***skip*** incapable to prune any branches in most cases, which results in many more searches occurred on the searcher tree. Also, without ***narrow***, the search procedure on the searcher tree always starts from the root, and the search number for each eliminator is exactly d_{large}. So the total search numbers on the searcher tree are linear with that on searchee tree. This might be inconsistent with the result shown in Table 1, where ***skip*** is able to cut down 21.5% time in average. However, in the realistic dataset, the intersecting trees might have few matches and they are localized rather than scattered, where ***skip*** fits in well.

Narrow is more effective compared with ***skip***, it sharply reduces the search numbers needed in the searcher tree. In terms with the result of ***opt***, we can see ***narrow*** contributes most of the improvements. Lastly, a clear performance degradation appears when the size ratio grows. When the size ratio is greater than 50, the numbers of searches tend to stay the same for all the methods. We attribute it to the distribution used in synthetic data, the scattered matches make our optimizations unable to omit invalid branches when the size grows large. In contrast, the real dataset shows better effectiveness in Table 1.