LAD Report: gLAD

Leonardo Gazzarri
November 21, 2016

1 Introduction

gLAD is a top-k autocompletion system written in C++11 and based on a succinct representation of a ternary search tree (TST). Succinct representation is achieved using the Succinct Data Structure Library (SDSL) [1]. In this work are proposed four versions of the index that represent the implemented TST, although all are based on a balanced parenthesis (BP) succinct representation of an ordered tree there are differences on how the information about the node pointers is represented, notably the information needed to discern for a generic node X and its parent Y if X is the *lonode*, *eqnode* or *hinode* of Y.

2 Design

In this section are presented in a very informal way the main building blocks of the work, *aka* the algorithms and the data structures used, and how they are used to build the application

2.1 Ternary Search Tree

A Ternary Search Tree (abbrev. TST) is a prefix-search tree in which every node is a *struct* containing a character *ch* and three pointers to its children, called *lonode*, *eqnode* and *hinode*. The lookup/search procedure for a prefix string P is straightforward: starting with node X equal to the root node of the TST and index i equal to 0, we compare *P[i]* with $X.ch$:

- case $P[i] < X.ch$: recurse using X as $X.lonode$
- case $P[i] = X.ch$: increment i and recurse using X as $X.eqnode$
- case $P[i] > X.ch$: recurse using X as $X.hinode$

When i is equal to *P.size()* or X is equal to *null* we stop, in the first case the response is positive (P is found), in the second one is negative (P is not found).

2.1.1 Compacted Ternary Search Tree

A Ternary Search Tree is built over a dictionary D of strings, therefore it is likely to have a string $s \in D$ for which the longest common prefix possible between s and another string of the dictionary has length much smaller than *s.length()*. This turns out to a TST representing D with *unary paths*. In order to avoid unary paths and, therefore useless wasting of space due to the pointers, a compacted form for a TST is used: here a node is a struct containing a string *label* and the usual three pointers to its children. Figure 1 shows how an unary path is compacted starting from a non-compacted TST. The lookup/search procedure is essentially the same of the uncompacted case, the difference resides in the fact that the comparison is made between *P[i]* and *X.label.back()* (the last character of *X.label*). At the end of the search a further comparison

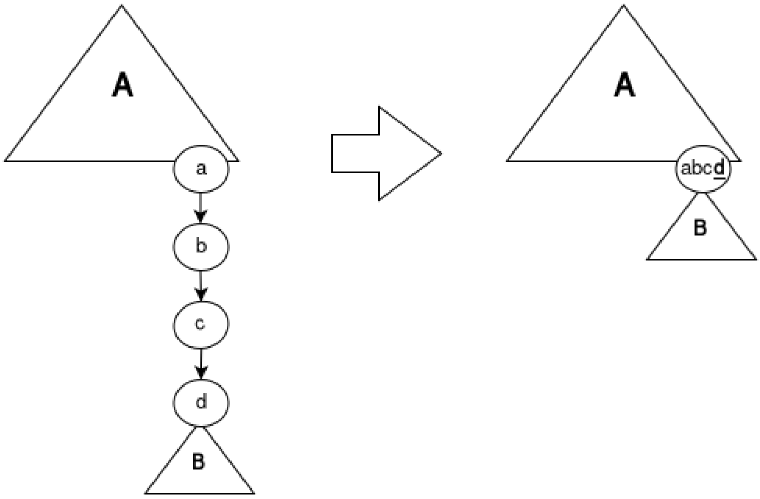


Figure 1: From a TST with one unary path (left) to its compacted version (right). A and B are subtrees without unary paths.

between P and the string built percolating the TST has to be made in order to ensure that P is represented by the TST.

2.2 Succinct Representation of a Ternary Search Tree

The succinct representation of trees that has been used is the one based on *Balanced Parenthesis* sequence to represent an ordered tree. The BP sequence of a given tree is obtained by performing a depth-first traversal, and writing an open parenthesis (1) each time a node is visited, and a closing parenthesis (0) immediately after all its descendants are visited [2]. Figure 2 shows an example. Together with the BP sequence we need to represent TST specific *satellite data*:

(A) the label for each node and its length,

(B) some *extra information* that tell us if a node is the *lonode*, *eqnode* or *hinode* for its parent.

Assuming a TST of n nodes, the labels are stored consecutively in an array of n entries. Together with this array we keep also an array *start_bv* (with *start_bv[0]=1*) that encodes the label lengths in unary (from *start_bv[1]*). From *start_bv* we can build a *select* data structure that enables the retrieval of the i-*th* label and its length. The i-*th* label is associated to the node having ID equal to i, *i.e.* the node represented as the i-*th* '1' (or '(') in the BP sequence. In order to represent (B), knowing that for *each* internal node its *eqnode* is always present, two approaches are possible:

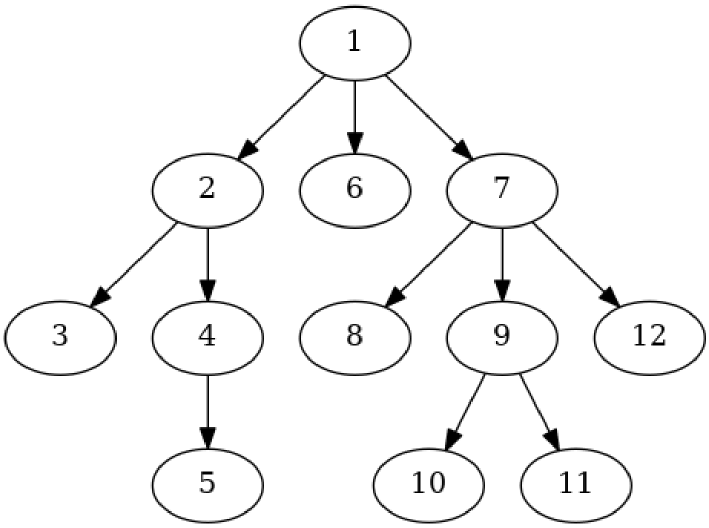
1. perform a depth-first traversal over the TST, and write '1' if the visited node is the *eqnode* of the parent or '0' otherwise
2. perform a depth-first traversal over the TST, and write '1' when the visited node has the *lonode* and *eqnode* or '0' if it has the *eqnode* and *hinode*. In the other cases (the node has three children or only one) is unspecified. One way to accomplish this is to write '1' iff the visited node has the *lonode*.

Figure 3 shows an example of the two approaches (H1 is the bit array corresponding to the first one, H2 the one corresponding to the second).

2.3 Top-k Search

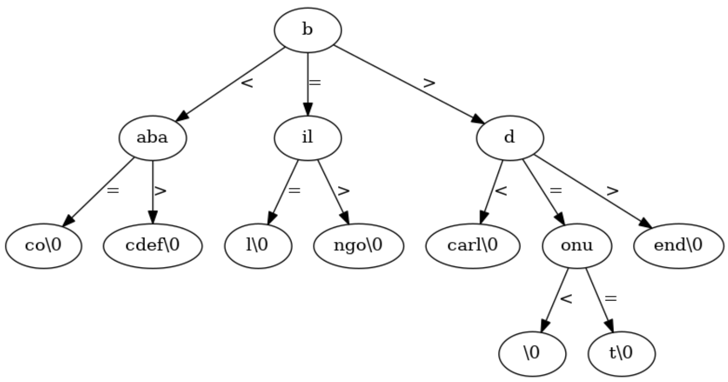
2.3.1 Prefix Search

It is possible to traverse the tree by means of `findclose(i)` operation over the BP sequence. This operation find the position of the close parenthesis (0) matching the open parenthesis (1) in position i and can be performed in constant time [3]. Given a position `i` in BP relative to a




```
BP: 1 1 1 0 1 1 0 0 0 1 0 1 1 0 1 1 0 1 0 0 1 0 0 0
    1 2 3 3 4 5 5 4 2 6 6 7 8 8 9 10 10 11 11 9 12 12 7 1
```

Figure 2: Example of a BP sequence for a given ordered tree.




```
S : 1010001000100000010010010000101000000100010100100001
L :  b aba co0 cdef0 il l0 ngo0 d carl0 onu 0 t0 end0
BP:  1 1   10  100   1  10 100  1 10    1   101001000
H1:  1 0   1   0     1  1  0    0 0     1   0 1  0
H2:  1 0             0          1       1
W :        3   1        5  4      7         1 8  5
```

Figure 3: Relevant sequences. Sequence L is the label sequence, S encodes the label lengths in unary. H1 and H2 represent the two different approaches in order to keep the information about lonode, eqnode and hinode.

node X, the first child of X is at position `c1 = i+1` (if it doesn't exist *BP[c1]* is equal to 0), the second one is at position `c2 = findclose(c1)+1` (if it doesn't exist *BP[c2]* is equal to 0), the third one at position `c3 = findclose(c2)+1` (if it doesn't exist *BP[c3]* is equal to 0). Given a position `i` in BP relative to a node X it is possible also to get its *node id* through $\texttt{rank}_1\texttt{(i)}$ and therefore retrieve the *i-th* label and its length using the *select* data structure over the unary length encoding of the label lengths already mentioned in the previous subsection 2.2. With this *node id* is possible to access also to the other support sequences (H1 or H2 of the figure 3) that together represent the compact TST. With these informations the prefix search procedure is essentially the lookup procedure described in 2.1.1.

2.3.2 Return the k Heaviest Indexes

The weights for each string are stored contiguously in memory as an array of N integers (where N is the number of the strings of the dictionary). A RMQ data structure ($O(N)$ additional space) is built over this array in order to determine the position of the maximum value in an arbitrary subrange $[i, j]$ in $O(1)$ time using the `rmq(i,j)` operation. After the prefix search for a prefix P we get the position v of the node X where the path from the root to X represents P itself. From this position we can get the range of strings prefixed by P in $O(1)$ by ranking on the bit pattern '10', thus the range $[rnk_{10}(v), rnk_{10}(findclose(v + 1)))$. Over this range we can apply recursively for k times the `rmq` operation in order to retrieve the k heaviest indexes for the strings. Then, given an index i we can reconstruct the *i-th* string getting its position in BP ($\texttt{select}_{10}\texttt{(i)}$ and percolating backward the tree until we reach position v (keeping attention on the extra-information in the two different cases).

3 Implementation Details

The goal of this section is to explain how the indexes **tst1**, **tst2**, **tst3** and **tst4** are built and show the differences among them eventually showing the most important parts of the code, trying to explain the critical parts.

3.1 Building the Index

Each one of the indexes implemented have been built roughly in the same way, exception given for the *extra-information* for the *lonode*, *eqnode* and *hinode* in the succinct representation. Listing 1 shows how a non-compacted TST is built recursively.

```
tnode * rec_build_tst (tVS& strings, int_t first, int_t last, int_t index) {
    if ( last < first ) {
        return nullptr;
    }
    uint8_t ch;
    int_t sx, dx;
    std::tie(sx, dx, ch) = partitionate(strings, first, last, index);
    tnode *node = new tnode(ch);
    node->lonode = rec_build_tst (strings, first, sx-1, index);
    if ( sx < dx || (sx == dx && ch != EOS)) {
        node->eqnode = rec_build_tst (strings, sx, dx, index+1);
    }
    if ( sx == dx && ch == EOS ) {
        strings[sx].clear();
    }
    node->hinode = rec_build_tst(strings, dx+1, last, index);

    return node;
}
```

Listing 1: Recursive building of the TST.

The procedure `partitionate` takes the string at position `first + (last-first)/2` and partitionates the strings between `first` and `last` position according to its `index`-th character `ch`. In order to keep the memory utilization reasonably low during this construction phase, `rec_build_tst` procedure is not directly called with `first == 0` and `last == strings.size()-1` but a two-step approach is adopted:

1. Build incrementally the higher part of the TST, iteratively.
2. When the range of strings have prefix length at least equal to one (`index == 1`, call the lambda `fun` described in the Listing 2 on this range of strings.

The `compress(node)` procedure compacts the unary paths of the TST rooted on `node` turning it to a compacted TST. The `mark` procedure builds the bit vectors that represent the tree performing a depth-first traversal as explained in 2.2. Then, the TST subtree is deleted.

```
auto fun = [&](tnode*& node, int_t start, int_t end, int_t index, bool markval){
    node = rec_build_tst (strings, start, end, index);
    compress(node);
    mark(node, markval);
    delete node;
    node = nullptr;
};
```

Listing 2: Lambda for the construction of the subtree when `index == 1`

After the construction of the bit vectors that represents the TST for a given dictionary, a `sdsl::bp_support_sada` structure that provides operations like `find_close` is built over the bit vector `m_bp` that represents the balanced parenthesis sequence. Then, *select* and *rank* data structures described in the previous section are built and also `sdsl::rmq_succinct_sct m_rmq` data structure is built over the `sdsl::vlc_vector<> m_weight` where are stored the weights (scores) of the dictionary strings sorted in a lexicographic order.

3.2 Top-K search

The implemented `top_k` returns the K heaviest strings and performs a case insensitive search lowering all the characters of the prefix string.

```
tVPSU top_k (std::string prefix, size_t k) const {
    std::transform(prefix.begin(), prefix.end(), prefix.begin(), ::tolower);
    std::string new_prefix(prefix.size(), 0);
    auto v       = search(prefix, new_prefix);
    auto range   = prefix_range(v);
    auto top_idx = heaviest_indexes(range, k);
    tVPSU result_list;
    for (auto idx : top_idx){
        std::string s;
        s.reserve(avgstrsize);
        s += new_prefix;
        s += std::move( build_string(m_bp_sel10(idx+1)-1, parent(v)) );
        result_list.push_back(tPSU(std::move(s), m_weight[idx]));
    }
    return result_list;
}
```

Listing 3: Top-K procedure

The `build_string` procedure reconstructs the string percolating the tree backward from the position of the leaf relative to the index found, until `parent(v)` where `v` is the result of `search`.

3.3 Search

The general *search* procedure is the following one:

```
int64_t search(const string& prefix, string& str) const {
    int64_t v = 0, i = 0;
    const char * data = (const char *) m_label.data();
```

```
    const size_t pref_len = prefix.size();
    size_t plen  = pref_len;
    size_t start = get_start_label(v);
    size_t end   = get_end_label(v);
    size_t llen  = end-start;
    std::strncpy(&str[i], data+start, (llen <= plen )*llen + ( plen < llen )*plen↩
        );
    while ( plen >= llen && i != pref_len && v >= 0 ) {
        i     += llen-1;
        v     = map_to_edge(v, prefix.at(i), data[end-1]);
        if ( v < 0 ) return v;
        i     += (prefix.at(i) == data[end-1]);
        plen  = pref_len-i;
        start = get_start_label(v);
        end   = get_end_label(v);
        llen  = end-start;
        std::strncpy(&str[i], data+start, (llen <= plen )*llen + ( plen < llen )*↩
            plen);
    }
    // here I have to match if the string is correct
    if ( v > 0 && prefix.compare(str) == 0 ) {
        for ( ; plen != 0 && llen != 0 ; plen--, llen-- )
            str.pop_back();
        return v;
    }
    return -1;
}
```

Listing 4: General search procedure. Returns position `v` of a node in `m_bp` bit vector where the path from the root to that node spells `prefix` string

The procedure is essentially the same for each tst (some differences for `tst4` version). Instead the `map_to_edge` procedure is specific for each tst index: given the current position `v` for the current node, the current character of the prefix and the last character of the current label, returns the next position in `m_bp` relative to a child node exploiting also the information in the `m_helper`s data structures that are specific for each index. Since for each label only the last character is compared, the labels are appended (entirely or minus the last character respectively if the last label character match with the i-th character of the prefix or not) to `str` and an additional comparison with the `prefix` has to be done at the end.

3.4 Difference of Indexes

3.4.1 Tst1

The *satellite data* giving the information that tell us if a node is the *lonode*, *eqnode* or the *hinode* for its parent is stored as a bit sequence in `sdsl::bit_vector m_helper` according to the first approach explained in 2.2. Therefore, given a position `v` relative to a node, `m_helper[node_id(v)-1]` is 1 if this node is the *eqnode* for `parent(v)`, 0 otherwise. Listing 5 shows the `map_to_edge` procedure for the **tst1** version.

```
int64_t map_to_edge(size_t v, uint8_t ch1, uint8_t ch2) const {
    size_t cv = v+1;
    for ( bool b = false, h = false; m_bp[cv] ; b = h ) {
        // "eqnode" is always present for internal nodes:
        // (h == true) => (b = true)
        h = m_helper[node_id(cv)-1];
        if ( (!b && !h && ch1 < ch2) || (h && ch1 == ch2) || (b && !h && ch1 > ↩
            ch2)) {
            return cv;
        }
        cv = m_bp_support.find_close(cv)+1;
    }
    if ( cv == v+1 && ch1 == ch2 ) {
        return v;
    }
    return -1;
}
```

Listing 5: `map_to_edge` procedure for tst1 version

If `h == 0` and `ch1 < ch2`, `cv` is the position relative to `lonode`, if `h == 0` and `ch1 > ch2`, `cv` is the position relative to `hinode`, if `h == 1` and `ch1 == ch2`, `cv` is the position relative to `eqnode`. Using this approach we have to store information on `m_helper` also for the leaves. This is avoided in the second version `tst2`.

3.4.2 Tst2

In this second version the *satellite data* giving the information that tell us if a node is the *lonode*, *eqnode* or the *hinode* for its parent is stored as a bit sequence in `sdsl::bit_vector m_helper` according to the second approach explained in 2.2. This approach permits to don't store bits relative to leaves in `m_helper` saving a little bit of space. Therefore, given a position `v` relative to a node X and knowing that `children(v).size == 2`, `m_helper[node_id(v)-1-m_bp_rnk10(v+1)]` equal to 1 tells us that the children of X are $X.lonode$ and $x.eqnode$, otherwise if it is equal to 0 it tells us that the children are $X.eqnode$ and $X.hinode$. Listing 6 shows the `map_to_edge` procedure for the **tst2** version.

```
int64_t map_to_edge(size_t v, uint8_t ch1, uint8_t ch2) const {
    auto nodes = children(v);
    auto out_size = nodes.size();
    auto d = m_bp_rnk10(v+1);
    auto h = m_helper[node_id(v)-1-d];
    if (out_size == 3)
        return nodes[ (ch1==ch2) + (ch1>ch2)*2 ];
    else if (out_size == 2 && h == 1 && ch1 <= ch2)
        return nodes[ch1==ch2];
    else if (out_size == 2 && h == 0 && ch1 >= ch2)
        return nodes[ch1>ch2];
    else if (out_size == 1 && ch1 == ch2)
        return nodes[0];
    else if (out_size == 0 && ch1 == ch2 )
        return v;
    return -1;
}
```

Listing 6: `map_to_edge` procedure for tst2 version

The `children(v)` procedure returns the children positions of a node in position `v` and in order to accomplish this it repeatedly executes `cv = m_bp_support.find_close(cv)+1` until `m_bp[cv]` is false and then stops. For this reason, `search` in `tst2` is much slower respect to the previous version. Results in section 5 show how bad is this version with respect to the others. Profiling with `perf` finds out that the problem of this inefficiency lies in `children` itself. This procedure is called inside `check_if_eqnode` (see 3.6) and inside `map_to_edge` and in order to execute it, 19% + 23% of the *total* CPU cycles are used only for a top-5 search. This is ridicously too much.

3.4.3 Tst3

This third version faces the problem -already exposed in the previous subsection- of repeatedly call the `find_close` procedure (in `children`). Here there two helper `sdsl::bit_vector`, `m_helper0` and `m_helper1`, that have been built during the construction phase. Therefore, during the depth-first traversal, for each `node` that is non-leaf evaluate:

1. `h0 = node->eqnode && ((node->lonode!=0) != (node->hinode!=0 ))`
2. `h1 = node->eqnode && ((node->lonode!=0) != (node->hinode!=0 ))`

Then append '1' if `h0` evaluates true (or '0' if false) to the `m_helper0`, and append '1' if `h1` evaluates true (or '0' if false) to the `m_helper1`. In this way is possible to look just two bits for a node X in order to determine how many children X has and what kind of children they are. Listing 7 shows the `map_to_edge` procedure for the **tst3** version.

```
int64_t map_to_edge(const size_t v, const uint8_t ch1, const uint8_t ch2) const {
    const size_t idx = node_id(v)-m_bp_rnk10(v+1)-1;
    const bool h0 = m_helper0[idx];
    const bool h1 = m_helper1[idx];
    const bool nl = m_bp[v+1];      // true if v is not a leaf
    const size_t cv = v+1;
    int64_t retval = 0;

    auto l3  = !h0 && h1 && nl;                      //(1) case 3 children
    auto l2a = h0 && h1 && nl && ch1 <= ch2;         //(2) case 2 children with lonode
    auto l2b = h0 && !h1 && nl && ch1 >= ch2;        //(3) case 2 children with hinode
    auto l1  = !h0 && !h1 && nl && ch1 == ch2;       //(4) case 1 children
    auto l0  = !h0 && !h1 && !nl && ch1 == ch2;      //(5) case leaf
    auto l23 = l3 || l2a || l2b;
    auto len = ( l3 && ch1 == ch2 ) + ( l3 && ch1>ch2 )*2 + \
               ( l2a && ch1 == ch2 ) + \
               ( l2b && ch1 > ch2 );

    retval += (!l23 && !l0 && !l1)*(-1);     //case no mapping
    retval += (l23 || l1)*cv;                //cv for (1) (2) (3) (4)
    retval += (l0)*v;                        //v for l0

    /* here we go: for-loop is not executed if !l23 */
    for ( int i = 0; i < len; i++ )
        retval = m_bp_support.find_close(retval)+1;
    return retval;
}
```

Listing 7: `map_to_edge` procedure for tst3 version

The procedure is written in this way trying to avoid as much as possible branch mispredictions during the execution of the code. The search procedure for version **tst3** is comparable to the one of version **tst1** but results in 5 show that in general **tst3** is slower than **tst1** and this is due principally to the different implementation of `check_if_eqnode` for the reconstruction of the found strings (see 3.6).

3.4.4 Tst4

This fourth version is essentially the third one, the difference lies in the fact that here a label relative to a leaf of the TST can contain a sequence of suffixes of strings in the dictionary with the same prefix. Therefore, `rec_build_tst` procedure of subsection 3.1 is modified as shown in Listing 8.

```
tnode * rec_build_tst (tVS& strings, int_t first, int_t last, int_t index) {
    tnode * node;
    if ( last - first < 0 ) {
        return nullptr;
    }
    int_t sx, dx; uint8_t ch;
    if ( last - first < thresh && ( count_chars(strings, first, last, index) <= ↩
        L1_line || first == last ) ) {
        *(first_it++) = first;
        node = new tnode("");
        node->label.reserve( L1_line );
        for ( size_t i = first; i <= last; i++ ) {
            node->label.append(strings[i], index, strings[i].size()-index);
        }
    } else {
        std::tie(sx, dx, ch) = partitionate(strings, first, last, index);
        node = new tnode(ch);
        node->lonode = rec_build_tst (strings, first, sx-1, index);
        node->eqnode = rec_build_tst (strings, sx, dx, index+1);
        node->hinode = rec_build_tst (strings, dx+1, last, index);
    }
    return node;
}
```

Listing 8: Recursive building of the TST for *Tst4*.

When `last - first` is less than `thresh` (e.g. 32) the suffixes of the strings in the range [first, last] are written contiguously as a single label in a leaf node if the total sum of the sizes of these suffixes is less equal than `L1_line` (e.g. 128). Searching here is a little bit different, the game is to percolate the tree for the prefix search P and properly handle the case if this search jumps into a leaf or not. The chosen approach for the prefix search inside the label associated to a leaf is a simple scan approach, since the label size is chosen to be ≤ 128 that is a possible value of a cache line and a scan should be the simplest and fastest approach. If the `search` does not end in a position for a leaf, the (at most) K leaves in which the results lie have to be found and then each suffix is found by a simple scan inside the label associated to the found leaf. Then `build_string` as usually. Useful for these purposes is the `m_first` array that has been built such that the `idx`-th leaf (from left to right) contains the suffixes of the strings starting from position `m_first[idx]`.

Results in 5 show that **tst1** and **tst4** are comparable in terms of time performances. This is true because the tree generated by this version has less nodes and therefore a smaller height. For this reason the `build_string` procedure (see 3.6) has less impact on performances since the number of executions for `parent` and `check_if_eqnode` is reduced. Moreover also cache misses due to the navigation of the tree have been reduced.

3.5 Heaviest Indexes

The approach explained in 2.3.2 has been implemented in the following way:

```
std::vector<size_t> heaviest_indexes( t_range& range, size_t k ) const {
    typedef std::tuple<t_range, size_t, size_t> t_q;
    auto cmp = [](const t_q& a, const t_q& b) {
        return get<2>(a) < get<2>(b);
    };
    std::vector<size_t> indexes;
    std::priority_queue<t_q, std::vector<t_q>, decltype(cmp)> q(cmp);
    if ( range[0] <= range[1] ) {
        size_t index = m_rmq(range[0], range[1]);
        q.push(make_tuple(range, index, m_weight[index]));
    }
    while ( indexes.size() < k && !q.empty() ) {
        auto t = q.top();
        auto r = get<0>(t);
        auto i = get<1>(t);
        auto w = get<2>(t);
        if ( r[0] < i ) {
            auto idx = m_rmq(r[0],i-1);
            q.emplace((t_range){{r[0],i-1}}, idx, m_weight[idx]);
        }
        if ( r[1] > i ) {
            auto idx = m_rmq(i+1,r[1]);
            q.emplace((t_range){{i+1,r[1]}}, idx, m_weight[idx]);
        }
        indexes.push_back(i);
        q.pop();
    }
    return indexes;
}
```

Listing 9: Get the K heaviest indexes in a range.

3.6 Reconstruction of a String

The following procedure 10 is used in order to reconstruct from a given position in `m_bp` bit vector.

```
std::string build_string(const size_t v_from, const size_t v_to) const {
            const char * data = (const char *) m_label.data();
            std::vector<bool> b;
            b.reserve(avgstrsize);
            std::vector<size_t> v;
```

```
            v.reserve(avgstrsize);
            v.push_back(0);
            b.push_back(true);
            for ( size_t k = v_from ; k != v_to ; ) {
                auto p = parent(k);
                v.push_back(k);
                b.push_back(check_if_eqnode(k, p));
                k = p;
            }
            std::string str;
            str.reserve(avgstrsize);
            size_t start, end;
            const size_t last = v.size()-1;
            for ( size_t i = last; i > 0; i-- ) {
                start   = get_start_label(v[i]);
                end     = get_end_label(v[i]);
                str.append(data + start, end - start - !b[i-1]);
            }
            if (str.back() == EOS)
                str.pop_back();
            return str;
        }
```

Listing 10: Reconstruct the a string from the position `v_from` in BP up to the position `v_to` in BP

The procedure is the same for each tst. The `check_if_eqnode` procedure is specific for each tst index: given two positions `k` and `p` it says if the node in position `k` is the *eqnode* of its parent node in position `p`. The **tst1** solution for `check_if_eqnode` is the simplest and most efficient since it's sufficient to check a single bit value in `m_helper[node(k)-1]`. The solutions in the other versions are more complicated (**tst3-4**) and definitely less efficient (**tst2**). Indeed one of the motivations for which **tst2** and **tst3** are slower than **tst1** lie in how the procedure `check_if_eqnode` is implemented in the two cases.

4 Tests

4.1 Correctness

Correctness has been proved comparing the results of the implemented solutions with the ones of *sigir16-topkcomplete* example project presented at SIGIR 2016 [4]. In order to accomplish this, I've forked that project [5] in order to modify it (case insensitiveness) and develop some useful tests in order to prove the correctness of gLAD. For example, calling on the project's root directory `./tests.sh 5 ./test/test_cases/big_range_queries1.txt ./data/italian_cities.txt` , where *big_range_queries1.txt* is a file in which line by line are written prefixes to search in the different versions **tst*** built over the *italian_cities.txt* dictionary file , returns the following output:

```
[tst1] 5        7.75499  206      702      13189890084728113741
[tst2] 5        10.9601  206      702      13189890084728113741
[tst3] 5        7.82336  206      702      13189890084728113741
[tst4] 5        7.53989  206      702      13189890084728113741
```

Where the format of a line is the following:
`[NameIndex]  K  AvgTime  ResFound  NumOfPrefixes  HashOfResults`
Where *HashOfResults* is the hash of a file created appending all the results for each query.

In the forked version of *sigir16-topcomplete* I have modified the indexes in order to perform case insensitive queries and modified the *index-main* in order to perform generic *top-K queries*. In this case, calling on the forked project's root directory `./tests.sh 5 ./test/test_cases/big_range_queries1.txt ./data/italian_cities.txt` we get the following output:

```
[index1] 5      1.04843  206      702      13189890084728113741
[index2] 5      1.58262  206      702      13189890084728113741
```

```
[index2a] 5      1.78632 206      702      13189890084728113741
[index3] 5       6.77208 206      702      13189890084728113741
[index3a] 5      7.01994 206      702      13189890084728113741
[index3b] 5      7.11823 206      702      13189890084728113741
[index3c] 5      8.14245 206      702      13189890084728113741
[index4a] 5      7.17949 206      702      13189890084728113741
[index4b] 5      7.47578 206      702      13189890084728113741
[index4c] 5      8.14245 206      702      13189890084728113741
[index4ci] 5     7.86182 206      702      13189890084728113741
```

The hashes are clearly equal and therefore correctness for this test file and dictionary is proven. Several tests have been made with all case files in `./test/test_cases/` and the same dictionary `italian_cities.txt` and all passed. Using bigger files as dictionaries, like the one of titles and click counts of Wikipedia (900 MB), can lead to different hashes. This is probably due (and I've checked with `diff` unix command) to the fact that many possible results with the same prefix can have the same score and the two version can choose different subsets of strings with the same score, or choose the same but order them in a different way. The dictionary `italian_cities.txt` is a small but significative "dictionary case" in which all the cities have different scores (number of citizens)!

5 Results and interpretation

This section shows some results of the implemented solutions and compares them to the ones got from *sigir16-topkcomplete*. All the performance tests have been performed on a workstation providing two Intel XEON CPU E5-260 (each one 8 cores, clocked at 2GHz, 2 private + 1 shared levels of cache). These results have been computed taking in account the different "nature" of the prefix queries, indeed different lengths for a prefix query P can vary the average top-k query time. For this reason there have been developed different files as test cases:

1. File `./test/test_cases/big_range_queries1.txt` contains all the possible combinations of strings with length at most 2 characters. The alphabet used to build these strings is formed by all the characters from 'a' to 'z'.
2. File `./test/test_cases/big_range_queries2.txt` contains all the possible combinations of strings with length at most 3 characters. The alphabet used here is the same of the previous file.
3. File `./test/test_cases/big_range_queries1.txt` contains all the possible combinations of strings with length at most 2 character. The alphabet used here is the one formed by all the characters from '!' (33 in ASCII) to 'z' (122 in ASCII).
4. File `./test/test_cases/range_queries.txt` contains different prefixes (of different sizes) of strings taken at random from a file literally fished somewhere in the Web.

What is following now are some graphics made to see "visually" how the different solutions behave one respect to the other and with respect to some of the solutions in [5]. The **index1** solution represents the baseline solution in which the prefix range search is implemented with a basic binary search over the dictionary and the top-K entries are determined through a scan approach and the use of an auxiliary priority queue. The **index4*** solutions are based on a succinct representation of a trie.

Figure 4 shows the graph relative to the *top-K search* average times using the queries in the first file. The prefix query here is at most of length two and its characters are included between 'a' and 'z', thus it's very likely that the `prefix_search` procedure returns a big range. Indeed for the queries in this file each top-K search returns exactly K strings. Therefore, when K grows, the procedure `build_string` becomes the big bottleneck since it builds K-times from K leaves the strings that have to be returned. Taking as example the **tst1** solution, profiling with *perf* shows that its heaviest bottlenecks with K=25 are `build_string` and `heaviest_indexes` (51% + 26% of the total CPU cycles, and `top_k` is 85%) while `search` has a very low impact (less than 1%). For K=5 the situation is different and the two functions `search`, `build_string` have approximately

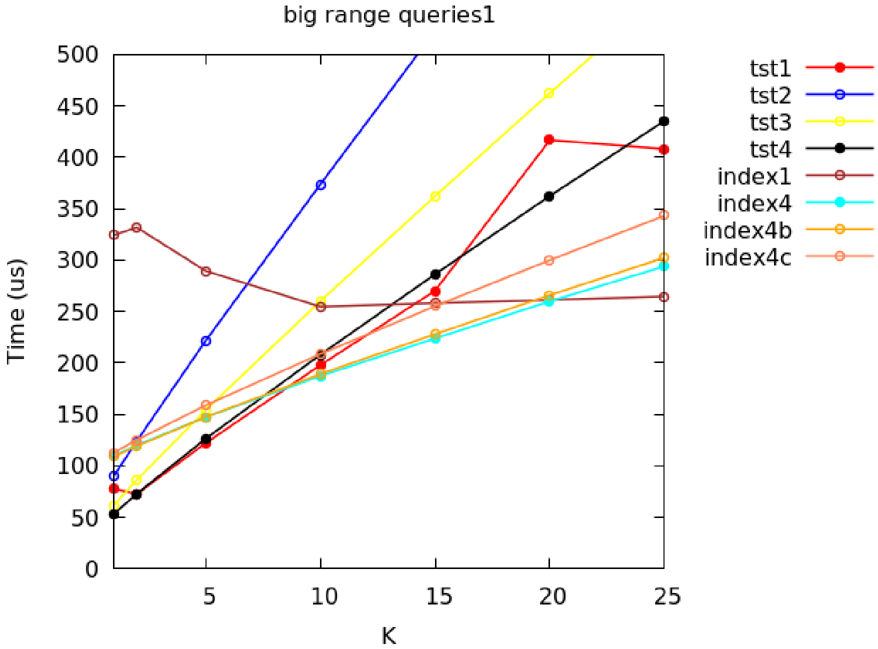


Figure 4: Graph relative to prefix queries of max length equal to 2. It is build using the file `big_range_queries1.txt`. Here the results returned for any top-K query is exactly K.

the same impact on performances (7%) while `heaviest_indexes` is the true bottleneck (34%). The versions **tst2** and **tst3** perform particularly bad on this test case since, as said in 3.6, the `check_if_eqnode` procedure is not efficient. The version **tst4** behave better than **tst3** because the tree built for this version has less nodes and thus a smaller height implying less `parent` and `check_if_eqnode` calls. Moreover for the tests over this file it has been noticed a bigger number of cache misses (circa 35%-40% of all cache refs for each implemented version) respect to the other files.

Figure 5 shows the graph relative to the *top-K search* average times using the queries in the second file. Here we have totally a different behavior, the average number of results returned by a top-K search is less than K (with K=25 we have on average 21 results per search). This impacts on the number of times for which `build_string` is called. Moreover the majority of queries have length three and the average size of the prefix range is less than the previous case. For these reasons the **tst*** curves go better and the average time for a top-K search is sensibly lower with respect to the previous case. Indeed, taking again as example **tst1**, profiling with *perf* shows that the top bottlenecks for the case K=25 are again `build_string` and `heaviest_indexes` but they are not so heavier with respect to `search` procedure.

Figure 6 shows the graph relative to the *top-K search* average times using the queries in the third file. Here, as for the first file, the prefix query is at most two characters long but the alphabet is different. Here we have many punctuation characters that can generate strings that don't constitute a prefix in the dictionary. Here is possible to see that the average number of results returned by a top-K search is far away from K.

Figure 7 shows the graph relative to the *top-K search* average times using the queries in the fourth file. Here the average number of results returned by a top-K search is less than K (with K=25 we have on average 19 results per search) and average prefix query length is equal to 4. Here profiling with *perf* (again on **tst1**) for K=5 the bottlenecks are `heaviest_indexes`, `build_string` and `search` (respectively 33%, 28% and 23% of the total CPU cycles). With K=25 the bottlenecks are `build_string`, `heaviest_indexes` and `search` (respectively 41%, 33% and 9% of the total CPU cycles).

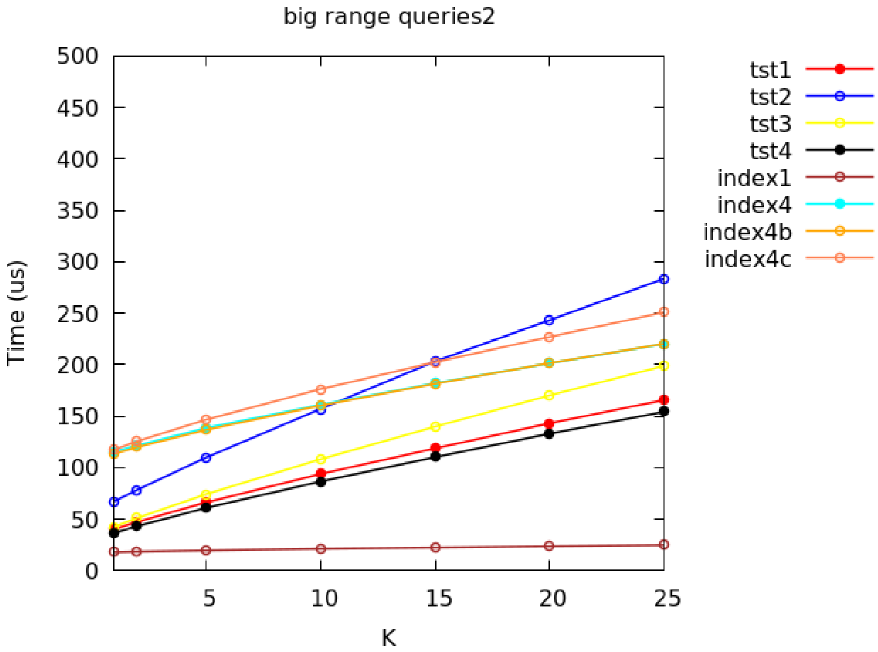


Figure 5: Graph relative to prefix queries of max length equal to 3. It is build using the file `big_range_queries2.txt`.

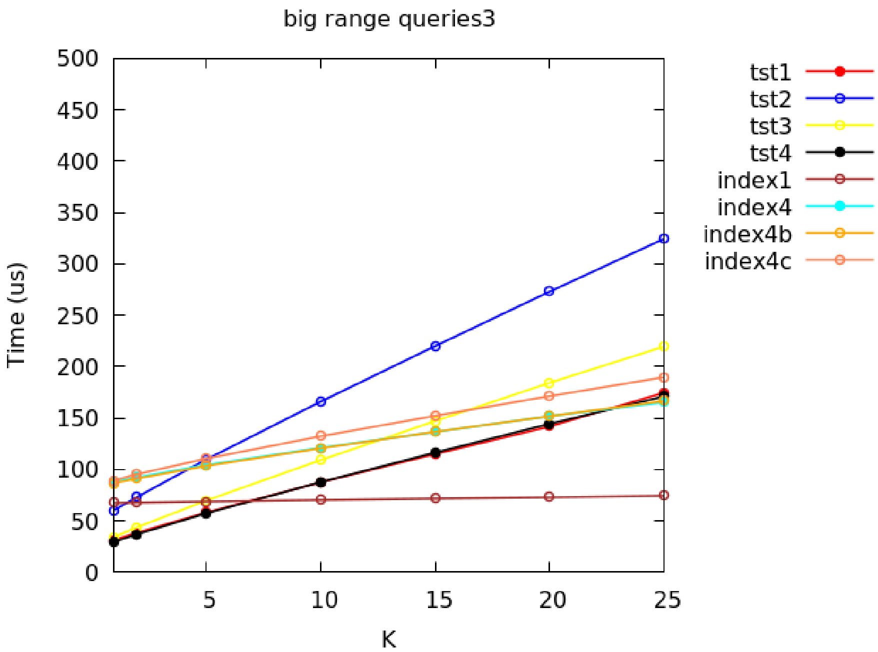


Figure 6: Graph relative to prefix queries of max length equal to 2. It is build using the file `big_range_queries3.txt`.

.

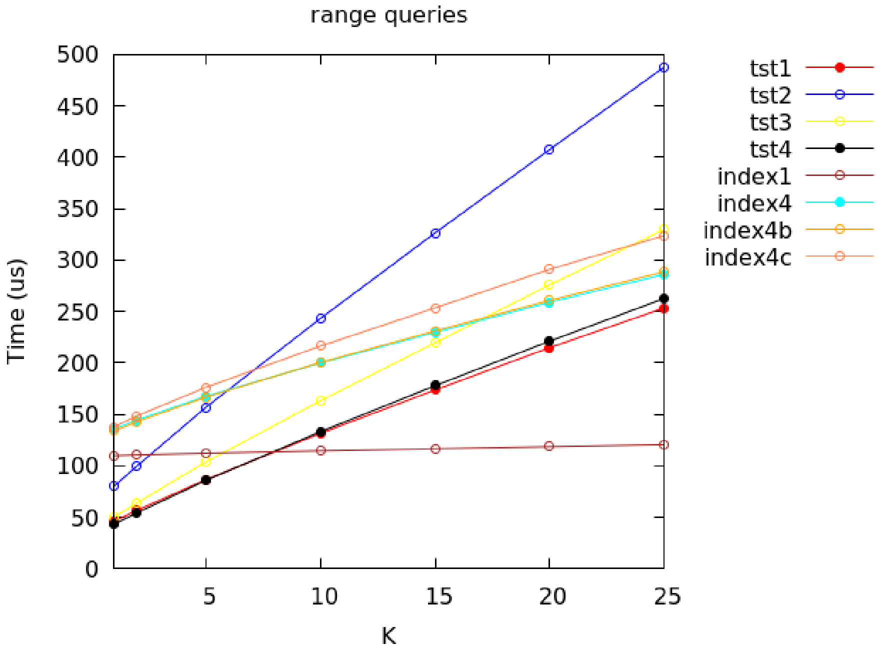


Figure 7: Graph relative to prefix queries of general size. It is build using the file `range_queries.txt`

.

Table 1 shows some recap values for which there are listed the size and the average time for a top-K search (with K=1, K=5 and K=25) for all the versions **tst*** developed for the project and the versions **index*** presented in *sigir16-topkcomplete*.

		Prefixes with size ≤ 2			Prefixes of generic size		
Version	Size	$T(1)$	$T(5)$	$T(25)$	$T(1)$	$T(5)$	$T(25)$
tst1	322	78.031	122.073	408.094	45.902	86.481	253.632
tst2	318	90.017	221.494	810.439	79.962	156.657	487.928
tst3	321	61.765	154.076	560.014	49.922	103.577	330.030
tst4	344	53.260	126.331	435.422	42.902	85.753	262.912
index1	770	324.288	289.514	264.697	110.003	112.289	120.717
index2	730	242.494	246.474	261.965	113.58	116.255	125.988
index2a	688	303.697	306.746	321.968	138.809	140.472	150.469
index3	361	388.664	416.139	529.816	253.192	278.840	378.974
index3a	357	392.677	419.074	531.889	255.606	280.772	379.752
index3b	271	1223.600	1231.180	1351.02	608.006	633.467	736.988
index3c	264	∼34000	∼34000	∼34000	∼14000	∼14000	∼14000
index4	366	109.705	147.393	294.095	135.566	167.552	286.299
index4a	366	110.007	147.691	496.194	135.843	167.435	286.518
index4b	281	108.942	147.372	302.487	133.736	166.663	288.86
index4c	273	112.814	158.854	343.287	137.76	176.278	323.909
index4ci	463	111.879	150.367	300.356	136.757	170.136	290.841

Table 1: Some values for the different versions of `tst*` developed in *gLAD* and the other ones present in *sigir16-topkcomplete*. The *Size* of the indexes is expressed in MB while $T(K)$ is the average time to retrieve the top-K results for a prefix.

References

[1] https://github.com/simongog/sdsl-lite

[2] Andrej Brodnik, Alejandro Lopez-Ortiz, Venkatesh Raman, Alfredo Viola, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro, on the Occasion of His 66th Birthday.* Springer, 13 ago 2013

[3] Benoit, D., Demaine, E.D., Munro, J.I.J., Raman, V.: *Representing trees of higher degree.* In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 169–180. Springer, Heidelberg (1999)

[4] https://github.com/simongog/sigir16-topkcomplete

[5] https://github.com/Murray1991/sigir16-topkcomplete