

# Implementation of LOUDS: Level-Ordered Unary Degree Sequence

Yuze Fu

University of Tokyo

fu-yuze@g.ecc.u-tokyo.ac.jp

## I. LOUDS

LOUDS (Level-Ordered Unary Degree Sequence) is a data structure for efficient (constant time complexity) tree operations. It uses a succinct index for binary vectors which has a constant time complexity for *rank* and *select* operations.

### A. Bit Vectors

Bit vectors could be represented efficiently in memory with one bit for each element in the vector, like in Figure 1. It takes 12 bits for representing a bit vector with 12 elements.

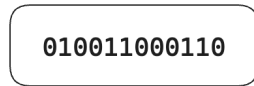


Figure 1: bit vector in memory

When operating on bit vectors, *rank* and *select* are the two most frequent operations. *rank* operation computes the number of 1s or 0s from the beginning of the vector to the position specified. For example, assuming the beginning position is 0, in the bit vector in Figure 2, the *rank*(5) returns 3, which is the count of 1 from  $V_0$  to  $V_5$ .

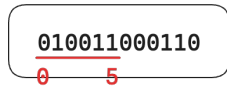


Figure 2: rank and select operations

In contrast, *select* operation returns the  $i$ -th 1 or 0 from the beginning of the vector. In Figure 2, *select*(3) returns 5, which is the position of the third 1. Therefore, we could see rank and select operations as dual, which means it always has

$$\text{rank}(\text{select}(i)) = \text{rank}(\text{select}(i)) = i \quad (1)$$

### B. FID Index for Bit Vectors

A kind of succinct index for bit vectors is called FID (Fully Indexable Dictionary). This kind of index provides constant time complexity for rank and select operations. In [1], they proposed a possible data structure for FID.

### C. LOUDS

With the power of FID indices, LOUDS [2] is proposed. The tree structure is first represented by a series of bits, where each 1 means a tree node, and each 0 means the separation between tree nodes. For example, for the tree in Figure 3, it is represented as the bit sequence on the bottom of the figure.

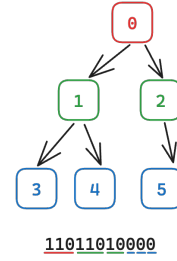


Figure 3: tree

The nodes are encoded in a bread-first order. The direct children (green) of the root node (red) is encoded at the beginning of the bit vector. As there are two children, it is two 1 and an ending 0 as 110. Then, for node 1, its children are encoded is 110, and for node 2, its children are encoded directly follows as 10. Then, for node 3, 4 and 5, they do not have children, so the three nodes are encoded as 000.

This data structure supports efficient queries between the identifier of a node and the index of the location of corresponding 1 in the bit vector. These two operations are called  $x = \text{bfs\_select}(i)$  and  $i = \text{bfs\_rank}(x)$  respectively. Here,  $i$  stands for the index of the bit vector, and  $x$  stands for the identifier of the node. Therefore,  $\text{bfs\_select}(6) = 5$ , and  $\text{bfs\_rank}(4) = 4$ .

We know that each 1 represents for a node, and the nodes are encoded in a bread-first order, so the  $x$ -th 1 is the node  $x$ . Therefore, there is

$$\text{bfs\_rank}(x) = \text{rank}_1(L, x) \quad (2)$$

where  $L$  is the bit vector. Similarly, there is

$$\text{bfs\_select}(i) = \text{select}_1(L, i) \quad (3)$$

as rank and select are dual just as *bfs\_select* and *bfs\_rank*. Since rank and select are  $O(1)$  with FID, *bfs\_select* and *bfs\_rank* are also  $O(1)$ .

The next pair of operations are getting the position  $i$  of a position  $i_0$  (`first_child_select( $i_0$ )`), and getting the identifier  $x$  of the parent of a node  $x_0$  (`parent_rank`). We know that each 0 stands for the end of a children sequence, so the  $i$ -th 0 is the end of children of the node  $i - 1$  (recall node 0 is the phantom root, and physical node identifiers begin from 1). Also, since the nodes are encoded bread-first, so the next position in  $L$  is the first children of node  $x$ . Therefore, there is

$$\text{first\_child\_select}(i_0) = \text{select}_0(L, i_0) + 1 \quad (4)$$

In addition, to get the parent position of a child identifier, we just need to count for 0 to the left of the current position to get the parent identifier, so there is

$$\text{parent\_rank}(x_0) = \text{rank}_0(L, x_0 - 1) \quad (5)$$

Since rank and select are  $O(1)$  with FID, `first_child_select` and `parent_rank` are also  $O(1)$ .

Additionally, one could get the starting and end position of child of a node  $x$ . The two operations are called `firstchild` and `lastchild`. `firstchild` is straightforward as it only need to convert  $x$  to position  $i_0$  and use `first_child_select` to get the position  $i$ .

$$\text{firstchild}(i_0) = \text{first\_child\_select}(\text{bfs\_rank}(i_0)) \quad (6)$$

If the result returns a position of value 0, it means the node does not have any child, so the function should return  $-1$  for *not found*.

similarly, `lastchild` could be implemented as getting the first child position of the next node and seek backward to get the resulting position. This is

$$\text{lastchild}(i_0) = \text{select}_0(\text{bfs\_rank}(i_0) + 1) - 1 \quad (7)$$

Finally, there are some trivial operations with  $O(1)$  to implement:

- `is_leaf( $x$ )`  $\rightarrow$  `bool` can be implemented as checking the first child position is 0 or not.
- `parent( $x$ )`  $\rightarrow$   $i$  can be implemented as getting the `parent_rank` and then apply `bfs_select` to get the position.
- `sibling( $i$ )`  $\rightarrow$   $i$  can be implemented as increasing the position once if the current position is not 0.
- `degree( $i$ )`  $\rightarrow$  `int` can be implemented as count the elements between `firstchild` and `lastchild`.
- `child( $x, k$ )`  $\rightarrow$   $i$  for getting the  $k$ -th child of  $x$  could be implemented as increasing the `firstchild` position by  $k - 1$ .
- `childrank( $x$ )`  $\rightarrow$  `int` for getting the rank for a child of  $x$  could be implemented as subtracting the position by the `firstchild` position of the parent and then add 1.

There are some operations with greater time complexity:

- `depth( $x$ )`  $\rightarrow$  `int` could be implemented as recursively adding 1 until reaching the root node.

- `lca( $x, y$ )`  $\rightarrow$   $x'$  could be implemented as recursively choosing the lca of the node with smaller index in  $\{x, y\}$  and the parent of the node with larger index.

## II. Implementation

I conducted the implementation with Rust. First, I have found a useful bit vector library with FID called `fid`, so I uses it for my base. It supports the `rank0`, `rank1`, `select1` and `select1` operations, and it could be constructed by inserting bit-by-bit.

### A. Construction

I defines a `TreeNode` for a user-friendly specification of tree nodes.

```
pub struct TreeNode {
    pub children: Vec<TreeNode>,
}

The child of each node is stored in the field children. For example, the tree in Figure 3 could be coded as

let root = TreeNode { // 0
    children: vec![
        TreeNode { // 1
            children: vec![
                TreeNode { children: vec![] }, // 3
                TreeNode { children: vec![] }, // 4
            ],
        },
        TreeNode { // 2
            children: vec![
                TreeNode { children: vec![] }, // 5
            ],
        },
    ],
};
```

Then, I wrote code to convert the tree into a bit vector. It is implemented just as the definition of LOUDS. The struct `Louds` represents the LOUDS data structure, and only contains the underlying bit vector with FID.

```
impl Into<Louds> for TreeNode {
    fn into(self) -> Louds {
        Louds::from_bit_vec(&*Into::<Vec<bool>>::into(self))
    }
}

impl Into<Vec<bool>> for TreeNode {
    fn into(self) -> Vec<bool> {
        let mut bits = Vec::new();
        // encode current node
        bits.resize(self.children.len(), true);
        bits.push(false);
        // encode children
        bits.extend(self.children.into_iter().flat_map(|
child| Into::<Vec<bool>>::into(child)));
        bits
    }
}
```

```
#[derive(Debug)]
pub struct Louds {
    lbs: BitVector,
}

impl Louds {
    pub fn from_bit_vec(s: &[bool]) -> Self {
        let mut lbs = BitVector::new();
        for c in s.iter() {
            lbs.push(*c);
        }
        Self { lbs }
    }
}
```

### B. Utilities

As the fid library's index definition is different from the one described before, I wrote wrapper over the fid library to make the index consistent.

```
impl Louds {
    fn select0(&self, i: u64) -> u64 {
        self.lbs.select0(i - 1)
    }
    fn select1(&self, i: u64) -> u64 {
        self.lbs.select1(i - 1)
    }
    fn rank0(&self, i: u64) -> u64 {
        self.lbs.rank0(i + 1)
    }
    fn rank1(&self, i: u64) -> u64 {
        self.lbs.rank1(i + 1)
    }
}
```

Since both the bit vector index and the node identifier is typed as u64, I created type wrappers with *New Type* pattern to avoid confusion in the code. In this way, one cannot accidentally assign node identifier to bit vector index.

```
#[derive(NewType, PartialEq, Eq, PartialOrd, Ord,
Clone, Copy, Debug)]
pub struct NodeId(pub u64);
```

```
#[derive(NewType, PartialEq, Eq, PartialOrd, Ord,
Clone, Copy, Debug)]
pub struct BitIndex(pub u64);
```

### C. Operations

I have implemented the operations just as the LOUDS data structure defines.

```
impl Louds {
    /// Convert the position x into the node i. Dual to
    /// `bfs_select`.
    pub fn bfs_rank(&self, x: BitIndex) -> NodeId {
        self.rank1(*x).into()
    }

    /// Convert the node i to position x. Dual to
```

```
`bfs_rank`.
    /// If the node i is out of range, return `None`.
    pub fn bfs_select(&self, i: NodeId) ->
Option<BitIndex> {
    Some(self.select1(*i))
        .filter(|i| *i < self.lbs.len())
        .map(BitIndex)
    }

    /// Get the parent node i of the position x.
    pub fn parent_rank(&self, x: BitIndex) -> NodeId {
        self.rank0(*x - 1).into()
    }

    /// Get the position of the first child of node i.
    pub fn first_child_select(&self, i: NodeId) ->
Option<BitIndex> {
    Some(self.select0(*i))
        .map(|x| x + 1)
        .filter(|i| *i < self.lbs.len())
        .map(BitIndex)
    }

    /// Check if the position is a leaf node.
    pub fn is_leaf(&self, x: BitIndex) -> bool {
        self.lbs.get(*self.bfs_rank(x)) == false
    }

    /// Get the parent node position of the position x.
    pub fn parent(&self, x: BitIndex) ->
Option<BitIndex> {
    self.bfs_select(self.parent_rank(x))
    }

    /// Get the position of the first child of the
    position x.
    /// If position x is not a valid node, the behavior
    is undefined.
    pub fn first_child(&self, x: BitIndex) ->
Option<BitIndex> {
    let y = self.first_child_select(self.bfs_rank(x));
    y.and_then(|y| {
        if self.lbs.get(*y) == false {
            None
        } else {
            Some(y)
        }
    })
    }

    /// Get the position of the last child of the
    position x.
    /// If position x is not a valid node, the behavior
    is undefined.
    pub fn last_child(&self, x: BitIndex) ->
Option<BitIndex> {
    let y = Some(self.select0(*self.bfs_rank(x) +
1)).map(|y| y - 1);
    y.and_then(|y| {
        if self.lbs.get(y) == false {
```

```

        None
    } else {
        Some(y)
    }
})
.map(BitIndex)
}

/// Get the position of the sibling of the position
x.
/// If position x is not a valid node, the behavior
is undefined.
pub fn sibling(&self, x: BitIndex) ->
Option<BitIndex> {
    Some(*x + 1)
    .filter(|x| self.lbs.get(*x) != false)
    .map(BitIndex)
}

/// Get the degree of the node of the position x.
pub fn degree(&self, x: BitIndex) -> u64 {
    if self.is_leaf(x) {
        0
    } else {
        *self.last_child(x).unwrap() -
        *self.first_child(x).unwrap() + 1
    }
}

/// Get the i-th child of the node at position x.
Dual to `child_rank`.
pub fn child(&self, x: BitIndex, i: usize) ->
Option<BitIndex> {
    if i as u64 >= self.degree(x) {
        return None;
    }
    self.first_child(x).map(|x| *x + i as
u64).map(BitIndex)
}

/// Get the rank of the current node at position x.
Dual to `child`.
/// Returns `None` if x is not a node or does not
have a parent.
pub fn child_rank(&self, x: BitIndex) ->
Option<usize> {
    self.parent(x)
    .and_then(|x| self.first_child(x))
    .map(|y| (*x - *y + 1) as usize)
}

/// Get the depth of a node.
pub fn depth(&self, x: NodeId) -> usize {
    if *x == 1 {
        return 0;
    }
    1 + self.depth(
        self.parent_rank(self.bfs_select(x).unwrap()))
}

```

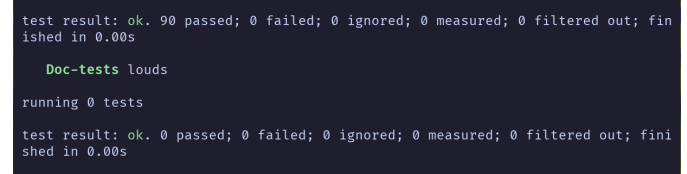
```

/// Get the least common ancestor of two nodes.
pub fn lca(&self, x: NodeId, y: NodeId) -> NodeId
{
    if x == y {
        return x;
    }
    self.lca(
        std::cmp::min(x, y),
        self.parent_rank(self.bfs_select(std::cmp::max(x,
y)).unwrap()),
    )
}
}

```

### III. Evaluation

I have used the test cases from <https://github.com/laysakura/louds-rs> for checking the correctness of the implementation. However, louds-rs has different operations naming and some operations are not implemented, so I borrowed the tree defined and manually created some test cases. The test cases could be run with the Rust cargo's builtin testing as cargo test, and has passed shown in Figure 4.



```

test result: ok. 90 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests louds

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

Figure 4: test cases

The source code for my implementation is available at <https://github.com/xfoxfu/louds>.

### References

- [1] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding  $k$ -ary trees, prefix sums and multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, p. 43, 2007, doi: 10.1145/1290672.1290680. [Online]. Available: <https://doi.org/10.1145/1290672.1290680>
- [2] Y. Kitamura, and M. M. and Yoshiaki Shiraishi, "Storage-efficient tree structure with level-ordered unary degree sequence for packet classification," in *Third Int. Symp. Comput. Networking, CANDAR 2015, Sapporo, Hokkaido, Japan, December 8-11, 2015*, 2015, pp. 487–490, doi: 10.1109/CANDAR.2015.86. [Online]. Available: <https://doi.org/10.1109/CANDAR.2015.86>