Graph Representation With Succint Data Structure

Thomas Messi Nguelé

January 30, 2021

Abstract

In this document, we show graph representation with succinct data structure. The end goal is to make the graph more accessible during application execution and thus accentuates the cache misses reduction and finally the execution time reduction of the entire application.

1 Generalities

Let n be the information-theoretic optimal number of bits needed to store data. A representation of this data is called:

- **Implicit** if it takes $n + O(1)$ bits of space.
- **Succinct** if it takes $n + o(n)$ bits of space ("o" is "small o").
- **Compact** if it takes $O(n)$ bits of space.

This document focus on succinct data structures. The goal of succinct data structure is to use an amount of space "close" to the information-theoretic lower bound and, in comparison to classical compressed representation, still allow efficient query operations. Succinct data structure have two principal operations called **rank** and **select**. Near to these two operations we also have successor and predecessor operations.

S	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	0	1	0	0	1	0

Table 1: Binary vector

1.1 Rank operation

$rank_b(S,k)$ function counts the number of elements b (most often bits) in the prefix of an array S up to some index k. In other words, this function tells us the number of "b" we have from the first position to k^{th} position. Mathematically, this function is expressed as follow:

$rank_b(S,k) = |\{i, i \in [1,k] \; and \; S[i] = b\}|$.

Example: Consider vector S in table 1.

- $rank_1(S,4) = 2$, $rank_0(S,4) = 2$
- $rank_1(S,5) = 2$, $rank_0(S,5) = 3$
- $rank_1(S,6) = 3$, $rank_0(S,6) = 3$
- $rank_1(S,14) = 7$, $rank_0(S,14) = 7$
- $rank_1(S,24) = 11$, $rank_0(S,24) = 13$

1.2 Select operation

$select_b(S,k)$ function gives the position of the k^{th} b. In other words, $select_b(S,k)$ gives the minimum i such that $rank_b(S,i) = k$. Mathematically, this function is expressed as follow:

$select_b(S,k) = min_i\{i \in [1,|S|], (rank_b(S,i) = k)\}$.

Example: Consider vector S in table 1.

- $select_1(S,4) = 9$, $select_0(S,4) = 7$
- $select_1(S,5) = 10$, $select_0(S,5) = 8$
- $select_1(S,6) = 11$, $select_0(S,6) = 12$
- $select_1(S,14) = -1$, $select_0(S,14) = -1$, -1 means that there no such position.
- $select_1(S,24) = -1$, $select_0(S,24) = -1$

1.3 Succ operation

$succ_p(S,b)$ gives the position of the next b after the position p. It can be computed with *select* and *rank* operations as follow:

$succ_p(S,b) = select_b(rank_b(S,p) + 1)$

Example: With the same vector S in table 1, we have:

- $succ_4(S,1) = select_1(rank_1(S,4)+1) = select_1(S,2+1) = select_1(S,3) = 6$
- $succ_7(S,1) = select_1(S, rank_1(S,7)+1) = select_1(S,3+1) = select_1(S,4) = 9$
- $succ_4(S,0) = select_0(S, rank_0(S,4)+1) = select_0(S,2+1) = select_0(S,3) = 5$
- $succ_7(S,0) = select_0(S, rank_0(S,7)+1) = select_0(S,4+1) = select_0(S,5) = 8$

1.4 Pred operation

$pred_p(S,b)$ gives the position of the previous b before the position p. Like succ operation, it can also be computed with *select* and *rank* operations as follow:
$pred_p(S,b) = select_b(S, rank_b(S,p) - 1)$

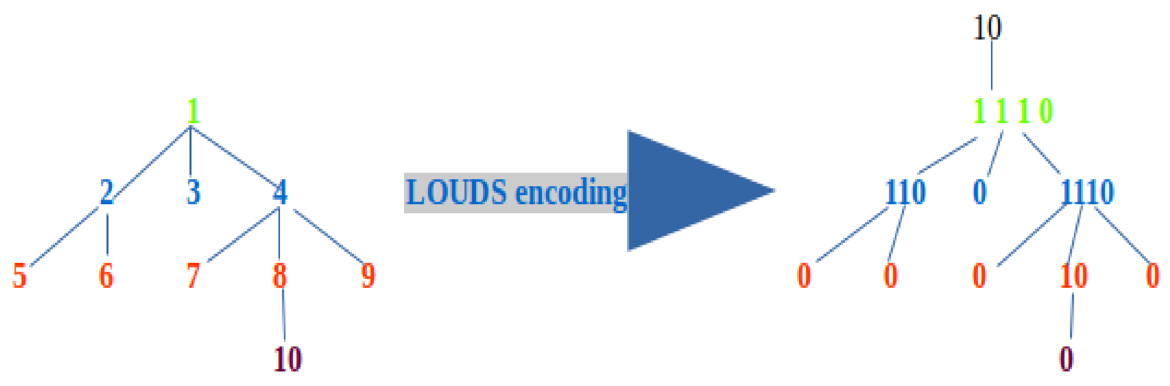
Example: With the same vector S in table 1, we have:

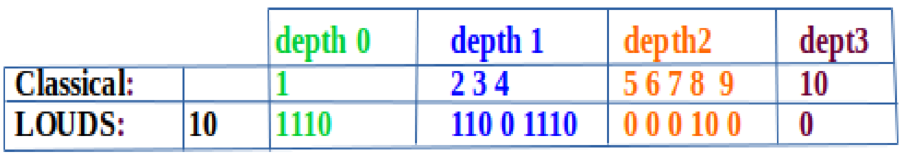
- $pred_4(S,1) = select_1(S, rank_1(S,4)-1) = select_1(S,2-1) = select_1(S,1) = 1$
- $pred_7(S,1) = select_1(S, rank_1(S,7)-1) = select_1(S,3-1) = select_1(S,2) = 4$
- $pred_4(S,0) = select_0(S, rank_0(S,4)-1) = select_0(S,2-1) = select_0(S,1) = 2$
- $pred_7(S,0) = select_0(S, rank_0(S,7)-1) = select_0(S,4-1) = select_0(S,3) = 5$

2 LOUDS Formalization

In this section, we described LOUDS encoding as it is described in [1]. LOUDS stands for Level-Order Unary Degree Sequence. LOUDS encoding consists in turning a tree into an array of bits via a breadth first search. The resulting array is the ordered concatenation of the bit representation of each node.

Each node is represented by a list of bits that contains as many as 1-bits as there are children and that is terminated by a 0-bit. Figure 1 shows an example of tree with its LOUDS encoding. For example node 1 (the root) is represented in LOUDS encoding by 1110. This is because it has three children. There is a node before the root represented by 10.





		depth 0	depth 1	depth2	dept3
Classical:		1	2 3 4	5 6 7 8 9	10
LOUDS:	10	1110	110 0 1110	0 0 0 10 0	0

Figure 1: Example of tree with its LOUDS encoding

2.1 Breadth first search in a tree

Algorithm 1 realizes the breadth first search of a tree. At line 2, the root is put in file. While the file is not empty, one node is taken from the queue and then it is treated. His children are put in the file.

Suppose that the treatment is printing. Then, BFS algorithm applied to the tree of figure 1 is: 1 234 56 789 10.

2.2 Example of LOUDS encoding with a 5-ary tree

In this section, we consider a tree with at most 5 children by node. We then write an algorithm 2 that generates the LOUDS encoding of any tree in parameter.

2.3 Parent of a node

Let $i (1 \leq i \leq n)$ be a node. Parent of i, $parent(i)$, is found by using the following steps:

- Jump to the position y of the i^{th} 1-bit in S: $y = select_1(S, i)$.
- Count the number j of $0's$ that appear before y in S: $j = rank_0(S, y)$.

j is the level-order number of the parent of i.

Algorithm 1 : Tree BFS (**B**readth **F**irst **S**earch)

Input: *Tree* **T**

Output:

```
File f;
f.enqueue(T.root);
while f.empty()<>true do
    $t \leftarrow f.dequeue()$
    treat(t);
    treated.add(t);
    $children \leftarrow t.children()$
    for all $child \in children$ do
        if $child \notin treated$ then
            f.enqueue(child);
        end if
    end for
end while
```

Example: Consider LOUDS encoding in Figure 1. Let B be this LOUDS encoding. We have:

1. $parent(4) =?$

   (a) $y = select_1(B, 4) = 5$

   (b) $j = rank_0(B, y) = rank_0(B, 5) = 1$

   So, $parent(4) = 1$.

2. $parent(9) =?$

   (a) $y = select_1(B, 9) = 13$

   (b) $j = rank_0(B, y) = rank_0(B, 13) = 4$

   So, $parent(9) = 4$.

2.4 Children of a node

Let $i (1 \leq i \leq n)$ be a node. The children of a node i, $children(i)$, are found by using the following steps:

Algorithm 2 : LOUDS encoding

Input: $5 - ary - Tree$ **T**

Output: LOUDS encoding CODE[N], N is the number of nodes of T.

```
integer i,k; File f;  init(f);
f.enqueue(T.root);
while f.empty()<>true do
   $k \leftarrow 0$
   $t \leftarrow f.dequeue()$
   for $i \leftarrow 1$ to 5 do
      if $(t.child[i] <> Nil)$ then
         f.enqueue($t.child[i]$);
         $k \leftarrow k + 1$;
      end if
   end for
   CODE[$t.info$] $\leftarrow (1...1)_k 0$;        // $(1...1)_k$ is sequence of k 1
end while
return  CODE;
```

- Jump to the position x of the i^{th} 0-bit in S: $x = select_0(S, i)$.
- Iterate over the position $x + 1, x + 2, ...,$ as long as the corresponding bit is '1'.

For each such position $x + k$ with $B[x + k] = 1$, the level-order numbers of i's children are $rank_1(B, x + k)$, which can be simplified to $x - i + k$.

Example: Consider LOUDS encoding in Figure 1. Let B be this LOUDS encoding. We have:

1. $children(4) =?$

   (a) $x = select_0(B, 4) = 10$

   (b) $chil = \{rank_1(B, x+k), 1 \leq k \leq t\, where\, t\, is\, the\, first\, integer\, such\, that\, B[x+t] = 0\} = \{rank_1(B, 10 + 1), rank_1(B, 10 + 2), rank_1(B, 10 + 3)\} = \{10 - 4 + 1, 10 - 4 + 2, 10 - 4 + 3\} = \{7, 8, 9\}$.

   So, $children(4) = \{7, 8, 9\}$.

2. $children(8) =?$

(a) $x = select_0(B, 8) = 17$

(b) $chil = \{rank_1(B, x+k), 1 \leq k \leq t \, where \, t \, is \, the \, first \, integer \, such \, that \, B[x+t] = 0\} = \{rank_1(B, 17+1)\} = \{17-8+1\} = \{10\}$

So, $children(8) = 10$.

3 GLOUDS Formalization

In this section we present GLOUDS formalization as it was described at [3]. Graph 2 will be used as example.

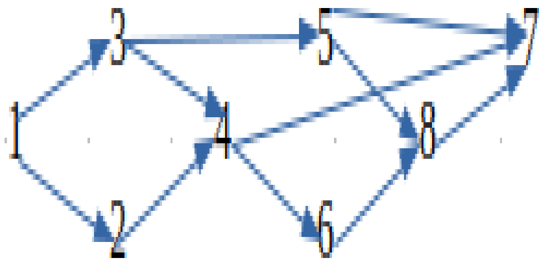


Figure 2: Example of graph (G)

As in [3], we assume that the graph is directed. The following characteristics are considered:

- n is the number of nodes in G
- m is the number of edges in G
- $c \leq n$ is the number of roots in G (the number of nodes that form the set from which we can reach every node of G)
- $k = m - n + 1$ is the number of non-tree edges in G (non-tree edges are nodes to be added to a spanning tree of G to obtain G). k is so because the number of tree edges is $n - 1$.
- $h \leq k$ is the number of nodes with more than 1 incoming edge (non-tree nodes).

3.1 Breadth first search in a graph

Algorithm 3 realizes the breadth first search of a graph G from a root. This algorithms assumes that there is a single root from which a path to every node exists. If no such root does not exists, and there are many roots in the graph, one can still run algorithm 3 for each root and finally build a super-root that will be the parent of all these roots.

Algorithm 3 starts by putting the root in a file data structure at line 2. At line 3, we put the root in the marked nodes list. While the file is not empty, one node is taken from the file (line 6), then their non-marked neighbors are put in the file (line 9), and finally we treat it (line 13). Suppose that the treatment is printing. Then, BFS algorithm applied to the graph of figure 2 is: 1 2 3 4 5 6 7 8.

Algorithm 3 : **Graph BFS** (**B**readth **F**irst **S**earch)

Input: *Graph* **G**

Output: Graph nodes in BFS fashion

```
1:  File f;
2:  f.enqueue(G.root);
3:  $marked\_nodes.add(G.root)$;
4:  while f.empty()<>true do
5:     $g \leftarrow f.dequeue()$
6:     $neighbors \leftarrow g.neighbors()$
7:     for all $node \in neighbors$ do
8:        if $node \notin marked\_nodes$ then
9:           f.enqueue(node);
10:          $marked\_nodes.add(node)$;
11:       end if
12:    end for
13:    $treat(g)$;
14:    $treated\_nodes.add(g)$;
15: end while
```

Let T_G^{BFS} be the resulting BFS-tree after running algorithm 2. T_G^{BFS} is augmented as follows:

- For each node w that is inspected but not visited during the BFS at node v (w has already been visited at an earlier point), we make a copy of w and append it as a child of v in the BFS-tree T_G^{BFS}. These nodes are called shadow nodes.

- We add a super-root to the initial root and call the resulting tree T_G. This tree has exactly $m + 2$ nodes. See the resulted tree of graph G at figure 3.

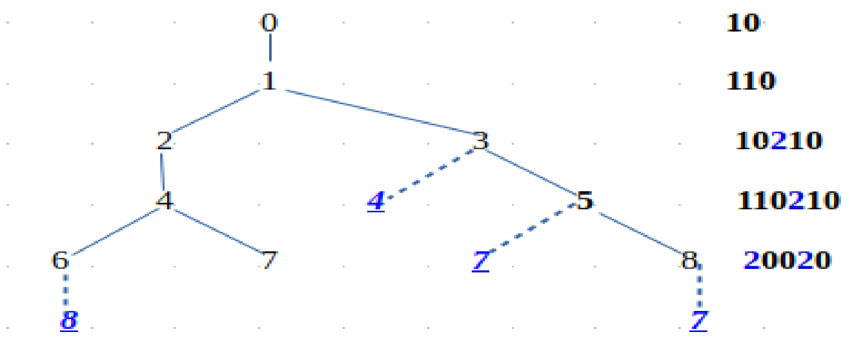


Figure 3: Resulting tree and shadow nodes of Graph G

3.2 GLOUDS encoding

In this section we show how one can represent the resulting tree T_G efficiently (considering space). The encoding used is close to LOUDS encoding but in this case, we build B a sequence of trits (values are taken in {0,1,2}) instead of a bit-vector. This is because one should distinguish between real nodes and shadow nodes. Algorithm 4 details Glouds encoding.

B is initially empty. Nodes of T_G are visited in level-order. Each visited node is a sequence of {0,1,2} as follows:

- For the super root (line 2), we put "10" in the sequence (line 3)
- For every child of a node:
  - If this child is a real node, then put "1" in the sequence (line 11)
  - If this child is shadow node, then put "2" in the sequence (line 14) and add this node to the list of non-tree nodes H (line 15).
- Add "0" when there is no more child (line 18).

Figure 4 presents the resulting B and H vectors.

3.3 Listing children of a node with GLOUDS code

Algorithm 5 gives the children of a node i, which are the nodes directly reachable from i. The algorithm start by reaching the first child. This first child is at the next position in B of the i^{th}0, which is $select_0(B, i) + 1$ (line 1). The children are listing as follows (while a zero is not met):

Algorithm 4 : GLOUDS encoding

Input: *Graph* **G**

Output: GLOUDS encoding $B \in \{0, 1, 2\}^*$ and H, the list of shadow nodes.

```
1:  File f;
2:  f.enqueue(G.root);
3:  B.add("10");                      //For the super root
4:  $marked\_nodes.add(G.root)$;
5:  while f.empty()<>true do
6:     $g \leftarrow f.dequeue()$
7:     $neighbors \leftarrow g.neighbors()$
8:     for all $node \in neighbors$ do
9:        if $node \notin marked\_nodes$ then
10:          f.enqueue(node);
11:          B.add("1");              //One child found
12:          $marked\_nodes.add(node)$;
13:       else
14:          B.add("2");              //Another child found, but it is a child of another node.
15:          H.add(node);             //We add node in H, the set of shadow nodes.
16:       end if
17:    end for
18:    B.add("0");                    //Mark the end of children
19:    $treat(g)$;
20:    $treated\_nodes.add(g)$;
21: end while
```

- If $B[x] = 1$, then the node is a real node, reaching it require the operation $rank_1(B, x)$. It is the x^{th} 1 of B vector.

- If $B[x] = 2$, then the node is a shadow node, reaching it require the operation $H[rank_2(B, x) - 1]$ (line 6). We first find the x^{th} 2 of B vector. Since it is a shadow node, this position is used in H vector to identify the child.

Example: Consider B and H vectors in Figure 4. What are $children(3)$ and $children(4)$?

1. $children(4) =?$

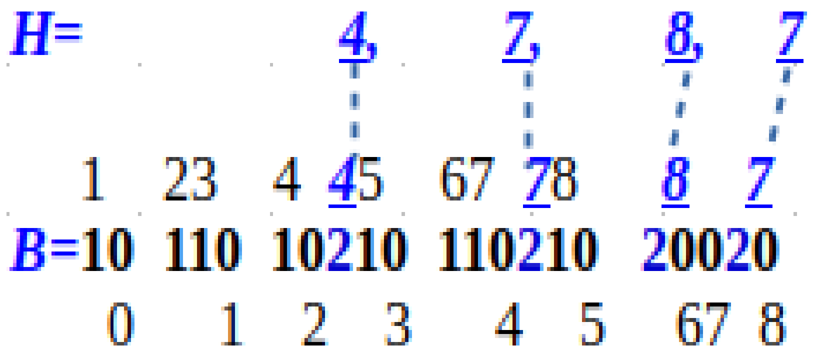


Figure 4: Resulting B and H vectors

Algorithm 5 : Children(i)

Input: *integer* **i,** *GLOUDS_encode* **B**

Output: List of children of node **i**.

```
1: $x \leftarrow select_0(B, i) + 1$;
2: while $(B[x] \neq 0)$ do
3:    if $(B[x] = 1)$ then
4:       Output$(rank_1(B, x))$;          //This is a real node.
5:    else
6:       Output$(H[rank_2(B, x) - 1])$;//This is a shadow node.
7:    end if
8:    $x \leftarrow x + 1$;
9: end while
```

(a) $x = select_0(B, 4) + 1 = 10 + 1 = 11$

(b) $output(rank_1(B, x)) = output(rank_1(B, 11)) = output(6)$.

(c) $x = x + 1 = 12$

(d) $output(rank_1(B, x)) = output(rank_1(B, 12)) = output(7)$.

So, $children(4) = \{6, 7\}$.

2. $children(3) = ?$

(a) $x = select_0(B, 3) + 1 = 7 + 1 = 8$

(b) $output(H[rank_2(B, x) - 1]) = output(H[1 - 1]) = output(H[0]) = output(4)$.

(c) $x = x + 1 = 9$

(d)

(e) $output(rank_1(B, x)) = output(rank_1(B, 9)) = output(5)$.

So, $children(3) = \{4, 5\}$.

3.4 Listing parent of a node with GLOUDS code

Algorithm 6 gives the parents of a node i; these are the nodes from which i is directly reachable. The algorithm start by reaching the first parent which is the only tree node. This is done with two operations rank and select as follows:

- Give the position p of the i^{th} 1 (its gives node i at line 1).
- Count the number of 0 before p (its gives the first parent at line 2).

After printing the first parent, Algorithm 6 gives the other parents (now shadow nodes) as follows:

- Look for the position x of the j^{th} occurrence of i in H (Line 1 and line 9)
- Set p the position of $(x + 1)^{th}$ 2 in B vector (Line 6).
- Look for the parent of p (counting the number of 0 from 1 to p). And then print it (Line 7).

Example: Consider B and H vectors in Figure 4. What are $parent(7)$?:

1. Output the first parent (tree node):

   (a) $p = select_1(B, 7) = 12$

   (b) $output(rank_0(B, p)) = output(rank_0(B, 12)) = output(4)$

2. Output the other parents (shadow nodes):

   (a) $x = select_7(H, j) = select_7(H, 1) = 2$

   (b) $p = select_2(B, x) = select_2(B, 2) = 14$

   (c) $output(rank_0(B, p)) = rank_0(B, 14) = 5$

   (d) $j = 2$

   (e) $x = select_7(H, j) = select_7(H, 2) = 4$

Algorithm 6 : Parents(i)

Input: *integer* **i,** *GLOUDS_encode* **B**

Output: List of parents of node **i**.

```
1: $p \leftarrow select_1(B, i)$;                  // Look for node i
2: output $rank_0(B, p)$;                          // Print the first parent which is the only tree node
3: $x \leftarrow j$
4: $x \leftarrow select_i(H, j)$;                  // Look for the position of the first occurrence of i in H
5: while $(x < k)$ do
6:     $p \leftarrow select_2(B, x)$;              // p is the position of the $x^{th}$ occurrence of 2 in B
7:     output $rank_0(B, p)$;                      // Print a parent which is shadow node (the number of 0 from 1 to p
8:     $j \leftarrow j + 1$
9:     $x \leftarrow select_i(H, j)$;              // x is the position of the $j^{th}$ occurrence of i in H
10: end while
```

(f) $p = select_2(B, x) = select_2(B, 4) = 20$

(g) $output(rank_0(B, p)) = rank_0(B, 20) = 8$

So, $parent(7) = \{4, 5, 8\}$.

4 Log(graph) Formalization

In this section we present Log(graph) formalization as it was described at [2].

5 Adopted Formalization

References

[1] Reynald Affeldt, Jacques Garrigue, Xuanrui Qi, and Kazunari Tanaka. Proving tree algorithms for succinct data structures. *arXiv preprint arXiv:1904.02809*, 2019.

[2] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. Log (graph) a near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[3] Johannes Fischer and Daniel Peters. Glouds: Representing tree-like graphs. *Journal of Discrete Algorithms*, 36:39–49, 2016.