Succinct Data Structure for Balanced Parentheses Sequences Representation

Fabrizio Brioni

June 2, 2022

Abstract

This document will describe a succinct structure to store and perform operations on balanced parentheses sequence. Initially the segment tree data structure will be described, followed by a description of a two-level organization (similar to the Jacobson rank structure) for storing information about balanced parentheses sequences and then an explanation of how to integrate the two structures to handle the operations `find_close`, `find_open`, `find_enclose` efficiently and succinctly.

1 Introduction

A sequence V of parentheses is balanced if in each prefix of V the number of open parentheses is greater than or equal to the number of closed parentheses, and the total number of open parentheses in V is equal to the number of closed parentheses. In a less formal way it can be defined as a sequence of parentheses from which a valid arithmetic operation can be obtained by adding some numbers and operations. Some useful operations on such sequences are the following:

- `find_close(i)`: given an open parenthesis with index i, calculate the index of the corresponding closed parenthesis;
- `find_open(i)`: given a closed parenthesis with index i, calculate the index of the corresponding open parenthesis;
- `find_enclose(i)`: given a parenthesis with index i, calculate the positions (x, y) of matching parentheses enclosing it. If there are multiple pairs with this property, select the innermost one, that is the one that does not contain other valid pairs (for a more formal definition read the **Notations**).

This document will show how, given a balanced parentheses sequence of length $2n$, perform such operations with a time complexity of $\mathcal{O}(\log n)$ using $2n + o(n)$ bits of memory.

2 Notations

Let v be a sequence of parentheses of length $2n$ numbered from 0 to $2n-1$ and v_i the parenthesis with index i. For convenience, assume a value of $v_i = 1$ indicates an open parenthesis at position i and a value of $v_i = 0$ indicates a closed parenthesis at position of i. We define the excess of a position i as:

$$\text{excess}_v(i) = |\{j : \ j < i \wedge v_i = 1\}| - |\{j : \ j \leq i \wedge v_i = 0\}|$$

that is the difference between the number of open parentheses preceding i (excluded) and the number of closed parentheses preceding i (included). It follows that a sequence of parentheses is balanced if and only if:

$$\begin{aligned} \text{excess}_v(i) &\geq 0 \qquad 0 \leq i < 2n-1 \\ \text{excess}_v(2n-1) &= 0 \end{aligned}$$

It also follows the definition of the operations of find_close, find_open and find_enclose:

$$
\begin{aligned}
\text{find\_close}_v(i) &= \min\{j :\ j > i \wedge \text{excess}_v(j) = \text{excess}_v(i)\} \\
\text{find\_open}_v(i) &= \max\{j :\ j < i \wedge \text{excess}_v(j) = \text{excess}_v(i)\} \\
\text{left\_enclose}_v(i) &= \max\{j :\ j < i \wedge \text{excess}_v(j) + 1 = \text{excess}_v(i)\} \\
\text{right\_enclose}_v(i) &= \text{find\_close}_v(\text{left\_enclose}_v(i)) = \min\{j :\ j > i \wedge \text{excess}_v(j) + 1 = \text{excess}_v(i)\} \\
\text{find\_enclose}_v(i) &= (\text{left\_enclose}_v(i), \text{right\_enclose}_v(i))
\end{aligned}
$$

We will use t^v to indicate a sequence of integers of length $2n$ such that:

$$t_i^v = \text{excess}_v(i) \qquad 0 \leq i < 2n - 1$$

Finally we define a function $\texttt{find\_succ}(x, i, v)$ that given a sequence $x_0, x_1, \ldots$ of integers and two integers i and v returns the index of the first element that follows x_i and has value less or equal v:

$$\text{find\_succ}(x, i, v) = \min\{j :\ j > i \wedge x_j \leq v\}$$

similarly we define a function $\texttt{find\_prev}(x, i, v)$ that returns the index of the last element that precedes x_i and has value less than or equal to v

$$\text{find\_prev}(x, i, v) = \max\{j :\ j < i \wedge x_j \leq v\}$$

3 Segment Tree

A Segment Tree for a sequence $x_0, x_1, \ldots, x_{n-1}$ of length n is a binary tree that has a root node containing information about the entire sequence (such as sum, maximum or minimum) and (if $n \neq 1$) having as left subtree a Segment Tree relative to the sequence $x_0, \ldots, x_{\lfloor \frac{n-1}{2} \rfloor}$ and having as right subtree a SegmentTree relative to the sequence $x_{\lfloor \frac{n-1}{2} \rfloor + 1}, \ldots, x_{n-1}$. Given a node k of that tree, with $k.data$ we will indicate the information contained in that node (in our case we will always store the minimum element in the range of competence) and with $k.left$ and $k.right$ we will indicate the left and right child respectively.

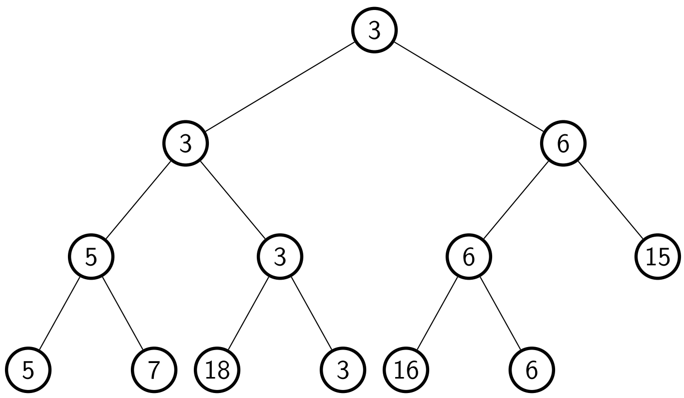

Figure 1: Segment tree for the sequence $5, 7, 18, 3, 16, 6, 15$. Each node contains the minimum value of the corresponding sequence.

3.1 Construction

The following procedure returns the Segment Tree root for the sequence x of length $n = size(x)$:

procedure Build(x)
    $root \leftarrow$Recursive_build$(x, 0, size(x) - 1)$
    return $root$
end procedure

function Recursive_build(x, l, r)
    $tmp \leftarrow$ new Segment Tree node
    if $l = r$ **then**
        $tmp.data \leftarrow x_l$
        $tmp.left \leftarrow null$

```
        tmp.right ← null
    else
        tmp.left ← RECURSIVE_BUILD(x, l, ⌊(l+r)/2⌋)
        tmp.right ← RECURSIVE_BUILD(x, ⌊(l+r)/2⌋ + 1, r)
        tmp.data ← min{tmp.left.data, tmp.right.data}
    end if
    return tmp
end function
```

For a sequence of length n the number of nodes created is:

$$f(n) = \begin{cases} 1 & n = 1 \\ 1 + f(\lfloor \frac{n}{2} \rfloor) + f(\lceil \frac{n}{2} \rceil) & n > 1 \end{cases}$$

it follows that $f(n) = \mathcal{O}(n)$ and since the function `Recursive_build` is called once for each node, the complexity of the construction procedure is $\mathcal{O}(n)$. Also assuming that each element of the sequence has a value between 0 and A, the number of bits needed to contain the information of all nodes is $\mathcal{O}(n \log A)$.

3.2 find_succ and find_prev

With this data structure it is possible to efficiently implement the function find_succ(x, i, v) after building the Segment Tree related to x (whose root is `root`):

```
procedure FIND_SUCC(root, x, i, v)
    return FIND_SUCC_RECURSIVE(root, i, v, 0, size(x) − 1)
end procedure

function FIND_SUCC_RECURSIVE(node, ind, val, l, r)
    if ind ≥ r ∨ node.data > val then
        res ← ∞
    else if l = r then
        res ← l
    else
        res ← FIND_SUCC_RECURSIVE(node.left, ind, val, l, ⌊(l+r)/2⌋)
        if res ≠ ∞ then
            res ← FIND_SUCC_RECURSIVE(node.right, ind, val, ⌊(l+r)/2⌋ + 1, r)
        end if
    end if
    return res
end function
```

The number of nodes visited and the complexity of this procedure is $\mathcal{O}(\log n)$. In a similar way we can implement the procedure find_prev(x, i, v):

```
procedure FIND_PREV(root, x, i, v)
    return FIND_PREV_RECURSIVE(root, i, v, 0, size(x) − 1)
end procedure

function FIND_PREV_RECURSIVE(node, ind, val, l, r)
    if ind ≤ l ∨ node.data > val then
        res ← −∞
    else if l = r then
        res ← l
    else
        res ← FIND_PREV_RECURSIVE(node.right, ind, val, ⌊(l+r)/2⌋ + 1, r)
        if res ≠ −∞ then
            res ← FIND_PREV_RECURSIVE(node.left, ind, val, l, ⌊(l+r)/2⌋)
        end if
    end if
    return res
end function
```

4 Two level organization of the information

Given a sequence v of balanced parentheses of length $2n$, divide it into *super blocks* of size $\log^2 2n$ (first level) and divide each super block into *blocks* of size $\log 2n$ (second level) and let $k = \log 2n$.

4.1 First Level

The number of super blocks in this level is $\frac{2n}{k^2}$ (numbered from 0 to $\frac{2n}{k^2} - 1$), for each of them calculate the minimum excess present obtaining the sequence S:

$$S_i = \min_{ik^2 \leq j < (i+1)k^2} \{t_j^v\} \qquad 0 \leq i < \frac{2n}{k^2}$$

Build a Segment Tree for this sequence: since the elements of t^v are between 0 and $2n$ (because they are the excesses of a sequence of $2n$ parentheses) and the sequence S has length $\frac{2n}{k^2}$, that Segment Tree will occupy $\mathcal{O}(\frac{2n}{k^2} \log 2n) = \mathcal{O}(\frac{2n}{\log 2n}) = o(n)$ bits.

Then calculate the sequence T composed of the initial excess of each super block:

$$T_i = t_{ik^2}^v \qquad 0 \leq i < \frac{2n}{k^2}$$

if we store the sequence T in an array the number of bits needed will also be $\frac{2n}{k^2} \log 2n = o(n)$, so the total number of bits used to store this first layer of information is $o(n)$.

4.2 Second Level

The number of blocks in this level is $\frac{2n}{k}$ (numbered from 0 to $\frac{2n}{k} - 1$), for each of them calculate the minimum excess present as a difference to the initial excess of its super block:

$$B_i = \left(\min_{ik \leq j < (i+1)k} \{t_j^v\} \right) - T_{\lfloor \frac{i}{k} \rfloor} \qquad 0 \leq i < \frac{2n}{k}$$

if we save the sequence B in an array the number of bits needed is $\frac{2n}{k} \log(2(\log 2n)^2) = \frac{2n}{\log 2n} \times$ $\times (2 \log \log 2n + \log 2) = o(n)$ because each element of B will be between $-(\log 2n)^2$ and $(\log 2n)^2$.

Let's also calculate the sequence A of the differences between the initial excess of each block and the initial excess of its super block:

$$A_i = t_{ik}^v - T_{\lfloor \frac{i}{k} \rfloor} \qquad 0 \leq i < \frac{2n}{k}$$

if we save the sequence A in an array the number of bits needed is $\frac{2n}{k} \log(2(\log 2n)^2) = o(n)$, so in total the number of bits used to store this second layer of information is $o(n)$.

5 find_close and find_open

Once built the Segment Tree S (whose root will be indicated with `root`) and calculated the values of T, B and A relative to the sequence of parentheses v you can perform the operation `find_close(i)` (assuming the parenthesis in position i is open) as follows:

1. Calculate the excess $u = t_i^v = T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} + (t_i^v - t_{k(\lfloor \frac{i}{k} \rfloor)}^v) = T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} + 2 \sum_{j=k(\lfloor \frac{i}{k} \rfloor)}^{i-1} v_j - (i - k(\lfloor \frac{i}{k} \rfloor)) + (1 - v_{k(\lfloor \frac{i}{k} \rfloor)})$. It can be calculated in $\mathcal{O}(\log n)$ (the summation contains at most $\log 2n$ addends);

2. Check if the searched position belongs to the same block of the index i by performing a linear scan of all parentheses following i until the end of the block it belongs to $(v_{i+1}, \ldots, v_{k(\lfloor \frac{i}{k} \rfloor + 1) - 1})$: if there is an index x such that the number of open and closed parentheses between i and x is equal then x is the position searched for. During this process, there are at most $k = \log 2n$ accesses to the sequence v;

3. Otherwise check if the searched position belongs to the same super block of the index i by scanning the values of B related to blocks after i until the end of the super block that i belongs to $(B_{\lfloor \frac{i}{k} \rfloor + 1}, \ldots, B_{k(\lfloor \frac{i}{k^2} \rfloor + 1) - 1})$: if there is an index x such that $B_x + T_{\lfloor \frac{i}{k^2} \rfloor} \leq t_i^v$ then the searched position belongs to the block x (and this block is in the same super block

of the index i), in this case to find the exact position in the block is sufficient a scan of all the parentheses of that block $(v_{xk}, v_{xk+1}, \dots, v_{xk+k-1})$ until the first index y such that $t^v_y = t^v_i$ (note how you can calculate the value of t^v_y during the scan using the same formula of step 1). During this process, there are at most $k = \log 2n$ accesses to the sequence B and at most $k = \log 2n$ accesses to the sequence v;

4. If the position was not found after the steps 2 and 3 means that the searched position is in another super block than the one where i is, to find that super block just call the procedure `Find_succ`$(B, \lfloor \frac{i}{k^2} \rfloor, t^v_i)$ that will return the index x of the first super block that follows i containing an excess less than or equal to t^v_i (and since two consecutive values of t^v differ by at most 1, that super block will definitely contain an excess equal to t^v_i). At that point you have to search for the block where the searched position is located and then search for the exact position in a similar way as explained in step 3: scan the values of B relative to the blocks contained in the super block x $(B_{xk}, \dots, B_{xk+k-1})$ and if there is an index y such that $B_y + T_x \le t^v_i$ then the searched position is in the block y, finally to find the exact index you need to scan all the parentheses of that block $(v_{yk}, \dots, v_{yk+k-1})$ until you find the first index z such that $t^v_z = t^v_i$. During this process a call is made to the function `Find_succ` (which has complexity $\mathcal{O}(\log \frac{2n}{k^2}) = \mathcal{O}(\log n)$) and there are at most $k = \log 2n$ accesses to the sequence B and at most $k = \log 2n$ accesses to the sequence v;

In conclusion you can do the operation `find_close` with complexity $\mathcal{O}(\log n)$ using the sequence v of parentheses that occupies $2n$ bits and other auxiliary structures (the Segment Tree related to the sequence S and the sequences T, B, A) that occupy $o(n)$ bits. The following pseudocode shows the implementation of the whole procedure:

Algorithm 1 `Find_close`

```
procedure FIND_CLOSE(root, T, A, B, n, v, i)          ▷ assume v_i is an open parenthesis
    k ← log 2n
    u ← T_{⌊i/k²⌋} + A_{⌊i/k⌋} − (i − k(⌊i/k⌋))                          ▷ step 1
    for j ← k(⌊i/k⌋) to i − 1 do
        u ← u + v_j
    end for
    u ← u + (1 − v_{k(⌊i/k⌋)})

    tmp ← 0                                                               ▷ step 2
    for x ← i to k(⌊i/k⌋ + 1) − 1 do
        if v_x = 1 then
            tmp ← tmp + 1
        else
            tmp ← tmp − 1
        end if
        if tmp = 0 then
            return x
        end if
    end for

    for x ← ⌊i/k⌋ + 1 to k(⌊i/k²⌋ + 1) − 1 do                             ▷ step 3
        if B_x + T_{⌊i/k²⌋} ≤ u then
            currt ← T_{⌊i/k²⌋} + A_x
            for y ← xk to xk + k − 1 do
                if currt = u then
                    return y
                end if
                if v_y = 1 then
                    currt ← currt + 1
                end if
                if v_{y+1} = 0 then
                    currt ← currt − 1
                end if
            end for
```

end if
end for

$x \leftarrow$ FIND_SUCC$(root, B, \lfloor \frac{i}{k^2} \rfloor, u)$ ▷ step 4
for $y \leftarrow xk$ **to** $xk + k - 1$ **do**
if $B_y + T_x \leq u$ **then**
$currt \leftarrow T_x + A_y$
for $z \leftarrow yk$ **to** $yk + k - 1$ **do**
if $currt = u$ **then**
return z
end if
if $v_z = 1$ **then**
$currt \leftarrow currt + 1$
end if
if $v_{z+1} = 0$ **then**
$currt \leftarrow currt - 1$
end if
end for
end if
end for
end procedure

Symmetrically you can implement the operation `find_open(i)`: just note that the operation `find_open(i)` related to a sequence $v = v_0, \ldots, v_{2n-1}$ is equivalent to a `find_(2n-1-i)` related to the sequence $v_{2n-1}, \ldots, v_0$. Initially we check if the searched position is in the same block of i, otherwise we check if it is present in the same super block of i and otherwise we search for the correct super block, the correct block within the super block and finally the exact position within the block:

Algorithm 2 `Find_open`

procedure FIND_OPEN$(root, T, A, B, n, v, i)$ ▷ assume v_i is a closed parenthesis
$k \leftarrow \log 2n$
$u \leftarrow T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} - (i - k(\lfloor \frac{i}{k} \rfloor) + 1)$ ▷ step 1
for $j \leftarrow k(\lfloor \frac{i}{k} \rfloor)$ **to** $i - 1$ **do**
$u \leftarrow u + v_j$
end for
$u \leftarrow u + (1 - v_{k(\lfloor \frac{i}{k} \rfloor)})$

$tmp \leftarrow 0$ ▷ step 2
for $x \leftarrow i$ **down to** $k \lfloor \frac{i}{k} \rfloor$ **do**
if $v_x = 1$ **then**
$tmp \leftarrow tmp + 1$
else
$tmp \leftarrow tmp - 1$
end if
if $tmp = 0$ **then**
return x
end if
end for

for $x \leftarrow \lfloor \frac{i}{k} \rfloor - 1$ **down to** $k(\lfloor \frac{i}{k^2} \rfloor)$ **do** ▷ step 3
if $B_x + T_{\lfloor \frac{i}{k^2} \rfloor} \leq u$ **then**
$currt \leftarrow T_{\lfloor \frac{i}{k^2} \rfloor} + A_{x+1}$
for $y \leftarrow xk + k - 1$ **down to** xk **do**
if $v_{y+1} = 0$ **then**
$currt \leftarrow currt + 1$
end if
if $v_y = 1$ **then**

```
                currt ← currt − 1
            end if
            if currt = u then
                return y
            end if
        end for
    end if
end for

x ← FIND_PREV(root, B, ⌊i/k^2⌋, u)                                    ▷ step 4
for y ← xk + k − 1 down to xk do
    if B_y + T_x ≤ u then
        if y < xk + k − 1 then
            currt ← T_x + A_{y+1}
        else
            currt ← T_{x+1}
        end if
        for z ← yk + k − 1 down to yk do
            if v_{z+1} = 0 then
                currt ← currt + 1
            end if
            if v_z = 1 then
                currt ← currt − 1
            end if
            if currt = u then
                return z
            end if
        end for
    end if
end for
end procedure
```

6 Find_enclose

The operation `left_enclose(i)` is analogous to `find_open(i)` with the only difference that the excess searched is not t_i^v but $t_i^v - 1$ (and there is no solution if $t_i^v = 0$):

Algorithm 3 `Left_enclose`

```
procedure LEFT_ENCLOSE(root, T, A, B, n, v, i)
    k ← log 2n
    u ← T_{⌊i/k^2⌋} + A_{⌊i/k⌋} − (i − k(⌊i/k⌋))                      ▷ step 1
    for j ← k(⌊i/k⌋) to i − 1 do
        u ← u + v_j
    end for
    u ← u + (1 − v_{k(⌊i/k⌋)})
    if v_i = 0 then
        u ← u − 1
    end if
    u ← u − 1                              ▷ the excess searched now is t_i^v − 1
    if u=-1 then
        return −1                                                     ▷ not found
    end if

    tmp ← u + 1                                                       ▷ step 2
    for x ← i down to k⌊i/k⌋ do
        if tmp = u then
            return x
        end if
```

```
        if v_x = 0 then
            tmp ← tmp + 1
        end if
        if v_{x-1} = 1 then
            tmp ← tmp - 1
        end if
    end for
    for x ← ⌊i/k⌋ - 1 down to k(⌊i/k^2⌋) do                      ▷ step 3
        if B_x + T_{⌊i/k^2⌋} ≤ u then
            currt ← T_{⌊i/k^2⌋} + A_{x+1}
            for y ← xk + k - 1 down to xk do
                if v_{y+1} = 0 then
                    currt ← currt + 1
                end if
                if v_y = 1 then
                    currt ← currt - 1
                end if
                if currt = u then
                    return y
                end if
            end for
        end if
    end for

    x ← FIND_PREV(root, B, ⌊i/k^2⌋, u)                            ▷ step 4
    for y ← xk + k - 1 down to xk do
        if B_y + T_x ≤ u then
            if y < xk + k - 1 then
                currt ← T_x + A_{y+1}
            else
                currt ← T_{x+1}
            end if
            for z ← yk + k - 1 down to yk do
                if v_{z+1} = 0 then
                    currt ← currt + 1
                end if
                if v_z = 1 then
                    currt ← currt - 1
                end if
                if currt = u then
                    return z
                end if
            end for
        end if
    end for
end procedure
```

And therefore the operation `find_enclose` is just a call to `left_enclose` and to `find_close`:

Algorithm 4 `Find_enclose`

```
procedure FIND_ENCLOSE(root, T, A, B, n, v, i)
    x ← LEFT_ENCLOSE(root, T, A, B, n, v, i)
    if x = -1 then
        return (-1, -1)                                           ▷ no solution
    else
        return (x, FIND_CLOSE(x))
    end if
end procedure
```

7 Conclusions

It has been shown how to perform the operations `find_close`, `find_open` and `find_enclose` with $\mathcal{O}(\log n)$ complexity using $2n + o(n)$ bits. The initial construction of all the necessary structures has complexity $\mathcal{O}(n)$.