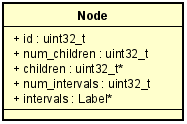
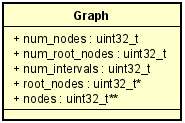
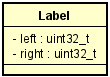
|  |
| --- |
| 0x00000000 |
| 0x00000020 |
| … |
|  |
|  |
|  |
|  |
|  |
|  |
| 0x000001A0 |
| … |
|  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 7 | 9 | 11 | 15 | 19 | 21 |

`

The array “children” contains node ids

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 5 | 8 | 9 | 11 | 20 | … |



|  |  |  |  |
| --- | --- | --- | --- |
| Label1 | Label2 | Label3 | Label4 |

The generation of the graph follows this main ideas:

* the generation is done concurrently by running *MAX\_THREADS\_GRAPH number* of threads
* at each iteration, each thread
  + reads *x* lines
  + increment shared counter of read lines (see p\_curr\_iteration)
  + allocates a single chunk of memory, big enough to store *x* Nodes (see node\_create\_multiple)
  + parse those lines (e.g. find the node\_id of the current node and of its children)
  + save the address of each created Node at graph->nodes[<node\_id>] = p\_node;

repeat until end of file

* find the root nodes. In practice loops through the bitmap b\_incoming\_edge\_nodes in order to find nodes that do not have any incoming edges.

Node\_create\_multiple allocates and initialize x nodes at a time, having in mind to reduce the num of calls to malloc, which it is a slow operation.

Node\_add\_children is an ad-hoc parser, which fills the children array of a node with its children ids. It takes as input the address of a node and the string to parse.

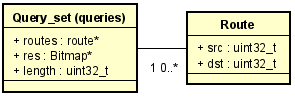
The array “root\_nodes” contains the node ids of root nodes of the graph

Interval Generation

1. Run *n* threads.
2. The i-th thread is tasked to generate the i-th interval
   1. For each root node choosen randomly call “randomized\_label”

Randomized\_label()

1. Return if the node has been already visited
2. Mark the node as visited
3. For each child of the node, call randomized label
4. Compute the label’s left and right value
   * Left value represents the smallest rank present in the subtree rooted at the current node
   * Right value is the rank of the current node



Query phase

1. Populate the struct “queries” from the file
2. Divide the queries in *n* blocks. Assign each block to a thread
3. For each query of its block, the i-th thread calls find\_path\_reachability:
   1. If curr.source == dest the query is reachable, then return “query reachable”
   2. Mark the curr.source as visited
   3. For each child of the current source
      1. If it has not been visited and the dest.labels[i-th] are contained in the curr.source\_node.labels[i-th]
         1. Calls find\_path\_reachability
   4. The current source can’t reach dest, then return “query non reachable”
4. Save the result in “queries->res”