Concurrent tree exploration algorithm

Mateusz Kobos

September 24, 2011

1 Introduction

We introduce a tree exploring algorithm with concurrent threads (or processes) traversing a tree.

The main assumption of the algorithm is:

- moving between nodes and processing them is time-consuming (expensive).

This assumption is particularly true in the application domain of our interest. In this domain the threads are independent web crawlers which explore tree-like web page document structure.

All of the threads share a common tree data structure which represents the part of the whole tree that has been already explored by the threads. The main task of this tree data structure is to share information between threads about nodes that were already visited or are processed at the moment. Out of convenience, in the rest of this paper we reference this tree data structure simply as "**the tree**"; on the other hand, the tree that is explored by the algorithm is referenced as "**the domain tree**".

The general structure of the tree data structure is shown in Fig. 1. We can see that the root node which normally has no parent, here has a parent called "**sentinel**". The sentinel is a purely technical node devised to make the crawling algorithm easier to write down. The sentinel node has no parent and only one child – the "**root**".

2 General description of the algorithm

During the algorithm execution, new nodes are added to the tree; the nodes already in the tree are never removed. Each node has a state and this state is changed by the threads that visit the node. When a node is added to the tree, it has an initial state called OPEN, which means that it has not been entered/visited by any thread.

Let us consider the algorithm executed by a single thread. We assume that the thread is currently in a certain node and wants to descend into one of its children nodes. First, it fetches children nodes from the domain tree and adds appropriate nodes corresponding to them to the tree. Then, it checks if there is any OPEN (non-visited) node available. If this

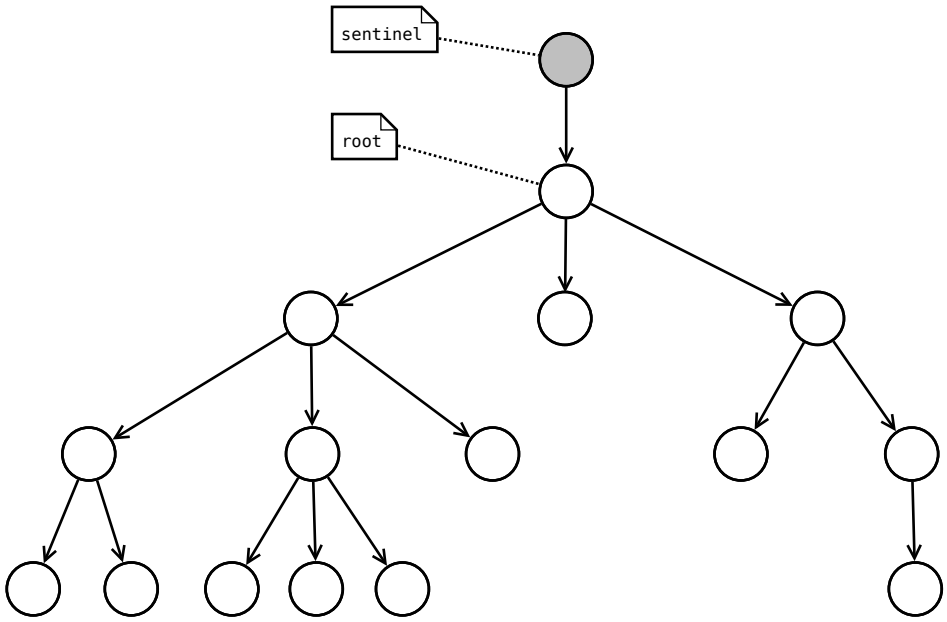

Figure 1: Traversed tree.

is the case, it descends into this node. We explore the OPEN (non-visited) nodes in the first place because when exploring such a node, the thread does not get into the way of other threads that might be exploring the same part of the tree. If there were no OPEN nodes available, it checks if there is any VISITED node available. If this is the case, it descends into this node. The node is in the VISITED state if it is an internal node and some thread has already visited it and is currently in some direct or indirect child of this node. If there is no such node available, it checks if there are any PROCESSING nodes available. The node is in the PROCESSING state if some thread is currently analyzing it. If there are such nodes, the thread waits until at least one of them is processed then checks again the state of the children. If each of these tests failed, it means that there are no children that can be explored. In such case, the thread moves upwards, to the parent of the node it is currently in.

While visiting a node, the thread changes the node state. See Fig. 2 for an overview how these states are changed. The thread changes the node's state to CLOSED when the whole subtree with a root in this node has been explored and processed and, as a consequence, there is no use for other threads to descend into this node. The thread changes the node's state to ERROR if some error occurred in one of its direct and indirect children and all other direct and indirect children are CLOSED (or are in the ERROR state). The advantage of this way of marking the erroneous nodes is that we can easily check after the algorithm's execution which part of the tree was affected by the error. What is more, if we want to try to analyze again the error nodes, it is sufficient to change all of the ERROR states in the tree to VISITED states and run the algorithm once again.

Note, that after the node is processed, its state is changed by the thread either to

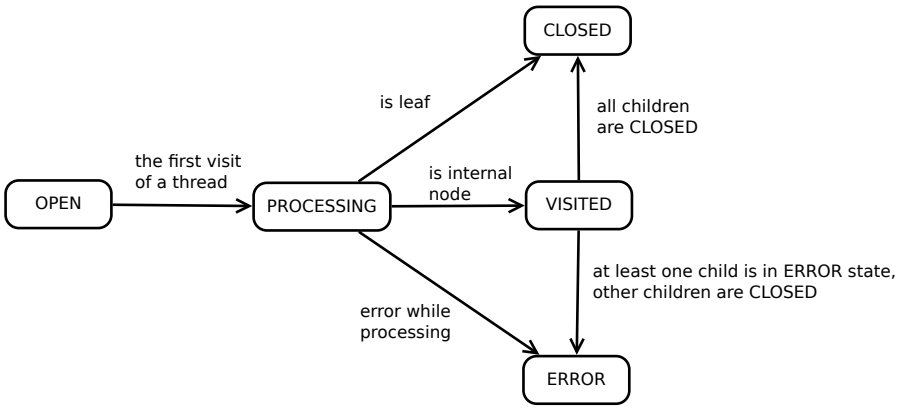

Figure 2: Possible states of a single node and passages between them.

CLOSED, ERROR, or VISITED. In case of the change to CLOSED and ERROR states we have to check the states of the node's siblings to see if wee should also change the state of the parent node. If all of the children of node's parent are now in CLOSED state, the parent's state has also be set to CLOSED. If at least one child is in ERROR state and other children are in CLOSED state, the parent's state has to be set to ERROR. These updates have to be done recursively.

3 Detailed description of the algorithm

In Sect. 2 we describe how the algorithm works from a high-level perspective. That perspective did not take into consideration challenges that are present when a single tree data structure is accessed concurrently. Here we present a more detailed description in a form a Python-like pseudocode.

```
class Node: # the tree's node structure
    state # state of the node
    parent # parent node
    children # children nodes
    # a condition object used for blocking access to node's children
    children_cond

while True:
    node = get_sentinel() # obtain the sentinel node
    try:
        while True:
            node = analyze_children_and_move_to_next_node(node)
            if node is None: # The whole tree has been traversed
                return
    except Exception as ex:
        pass
```

```
17
18 # return the node we're currently in
19 def analyze_children_and_move_to_next_node(node):
20     (child, action) = get_available_child(node)
21     if child is None:
22         return node.parent
23     else:
24         node = child
25         if action == TO_PROCESS:
26             # the following function returns True iff the given node is
                   a leaf
27             is_leaf = process_node_and_check_if_is_leaf(node)
28             set_node_type(node, is_leaf)
29             if is_leaf:
30                 return node.parent
31             else:
32                 return node
33         elif action == TO_VISIT:
34             return node
35
36 def get_available_child(node):
37     while True:
38         node.children_cond.acquire()
39         child = internal_get_available_child(node)
40         if child is None: # No accessible children are available
41             node.children_cond.release()
42             return (None, None)
43         if child.state == OPEN:
44             child.state = PROCESSING
45             node.children_cond.release()
46             return (child, TO_PROCESS)
47         elif child.state == VISITED:
48             node.children_cond.release()
49             return (child, TO_VISIT)
50         elif child.state == PROCESSING:
51             node.children_cond.wait()
52             node.children_cond.release()
53
54 def internal_get_available_child(node):
55     # If there is a child in OPEN state available, return it. Otherwise
       , check
56     # if there is a child in VISITED state. If there is, return it.
       Otherwise,
57     # check if there is a child in PROCESSING state. If there is,
       return it,
```

```
58      # otherwise return None.
59
60  def set_node_type(node, is_leaf):
61      node.parent.get_children_cond().acquire()
62      if is_leaf:
63          node.state = CLOSED
64          internal_update_node_state(node.parent)
65      else:
66          node.state = VISITED
67      node.parent.children_cond.notify_all()
68      node.parent.children_cond.release()
69
70  def internal_update_node_state(node):
71      if node == sentinel:
72          # The state of the sentinel is undefined and not used
73          # in the algorithm, it should not be changed
74          return
75      new_state = None
76      if all_children_are_in_one_of_states(node, {CLOSED}):
77          new_state = CLOSED
78      elif all_children_are_in_one_of_states(node, {ERROR, CLOSED}):
79          new_state = ERROR
80      # Node state does not have to be changed
81      if new_state is None:
82          return
83      node.parent.children_cond.acquire()
84      node.state = new_state
85      internal_update_node_state(node.parent)
86      node.parent.children_cond.notify_all()
87      node.parent.children_cond.release()
88
89  def all_children_are_in_one_of_states(node, set_of_states):
90      # return True iff all of the node's children are in one of the
            given states
91
92  def handle_error(node):
93      # This function is called when one an error occurs while visiting a
94      # certain node.
95      set_error(node)
96      raise Exception()
97
98  def set_error(node):
99      node.parent.children_cond.acquire()
100     node.state = ERROR
101     internal_update_node_state(node.parent)
```

```
102     node.parent.children_cond.notify_all()
103     node.parent.children_cond.release()
```

See also Fig. 3 which shows how an example update of a node state is propagated upwards the tree.

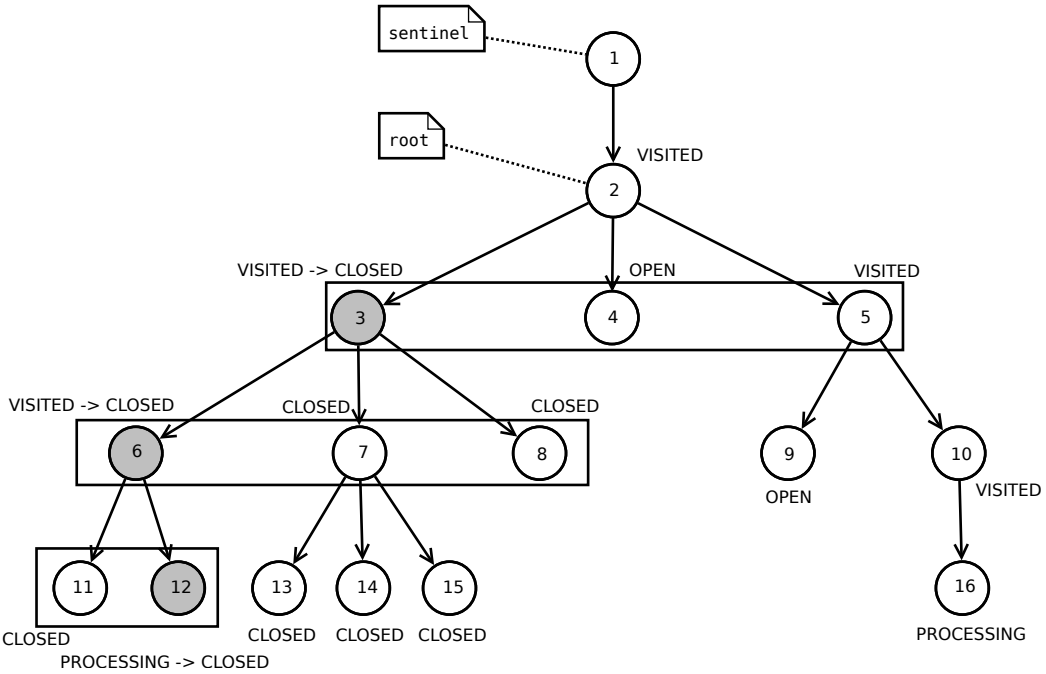

Figure 3: Updating the state of the nodes from the bottom to the top. Here a state of the node 12 changed from PROCESSING to CLOSED. As a consequence, the state of appropriate direct and indirect parent nodes' is changed. The rectangle embracing sibling nodes denotes that the condition object corresponding to them is locked during the update.