Fast Implementation of the Progressive Edge-Growth Algorithm

Lin Chen and Da-Zheng Feng

ABSTRACT—A computationally efficient implementation of the progressive edge-growth algorithm is presented. This implementation uses an array of red-black (RB) trees to manage the layered structure of check nodes and adopts a new strategy to expand the Tanner graph. The complexity analysis and the simulation results show that the proposed approach reduces the computational effort effectively. In constructing a low-density parity check code with a length of 10^4, the RB-tree-array-based implementation takes no more 10% of the time required by the original method.

Keywords—LDPC code, progressive edge-growth algorithm, red-black tree.

I. Introduction

The progressive edge-growth (PEG) algorithm, introduced by Hu and others [1], is an effective approach to constructing low-density parity check (LDPC) codes with large girth. The PEG construction builds the Tanner graph for an LDPC code by establishing edges between the variable nodes and the check nodes in an edge-by-edge manner and maximizing the local girth in a greedy fashion. The key point of this algorithm is the search for the most distant check node. In [1], an indicator-based tree expansion of a Tanner graph is used to find the check nodes which are furthest from the root node. Then, a linear search is adopted to locate the check nodes with the lowest degree among them. This approach is simple but not efficient enough. In the case of long codes, the construction of LDPC codes with this approach requires long time consumption [2].

Inspired by the computationally optimal metric-first stack algorithm presented in [3], we propose a computationally efficient implementation of the PEG algorithm. An array of red-black (RB) trees [4] is used to manage the layered structure of the check nodes, and dynamic adjustments of this structure are adopted to partly replace the tree expansion of the Tanner graph. Compared with the existing approach in [1], the complexity of this RB-tree-array-based implementation is significantly lower.

Manuscript received Nov. 11, 2008; revised Jan. 15, 2009; accepted Feb. 18, 2009.

Lin Chen (phone: + 86 29 8820 0000, email: lchen30@gmail.com) and Da-Zheng Feng (email: dzfeng@mail.xidian.edu.cn) are with the National Laboratory of Radar Signal Processing, Xidian University, Shaanxi, China.

II. RB-Tree-Array-Based Implementation

1. Layered Structure of Check Nodes

As shown in the left part of Fig. 1, the tree expansion [1] of a Tanner graph from the variable node s_j partitions the check nodes into layers according to their respective shortest distances from s_j. The layered structure is represented by the set $L_{s_j} = \{L_{s_j}^{-1}, L_{s_j}^{0}, \cdots, L_{s_j}^{N-1}\}$, where the subscript s_j denotes the variable node from which the structure results, and the superscript l ($l = -1, 1, \cdots, N-1$) identifies the layer so that $L_{s_j}^{l}$ represents the set of the check nodes in the layer l. Note that the check nodes that are not connected to s_j in the current Tanner graph constitute a special layer with the label –1. Assume that the variable node s_j has d_{s_j} edges linked to it, and let $E_{s_j}^{k}$ $(0 \le k < d_{s_j})$ denote the k-th edge. The layered structure L_{s_j} is built up after the first edge $E_{s_j}^{0}$ has been established. However, L_{s_j} is changed by the newly added edge $E_{s_j}^{k}$ $(1 \le k < d_{s_j})$, which introduces new paths into the graph. Whenever a check node gets a shorter path from s_j, its location should be correspondingly adjusted from one layer to another. Therefore, set $L_{s_j}^{l}$ varies dynamically and may be null after adjustments.

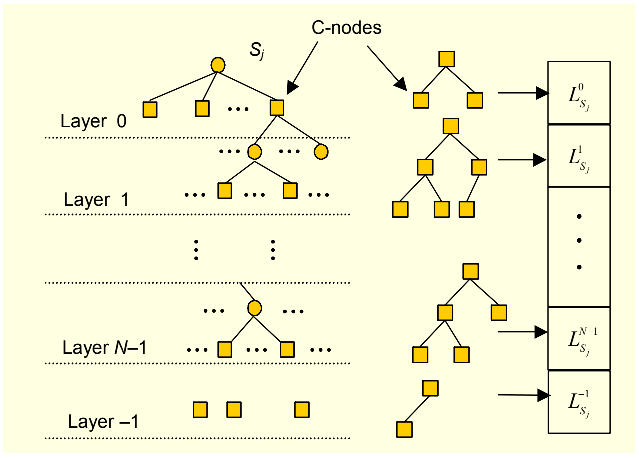

Fig. 1. Layered structure from S_j and the RB-tree array.

2. Data Structure

To assist the management of L_{s_j} with an RB-tree array, the following data structure is adopted to represent a check node. A check node c_i includes four fields: an identifier number (*id*), the distance from the current root node (*dist*), the current link degree (*deg*), and a pointer table (*vTable*) which records the variable nodes that are linked to c_i in the current Tanner graph. The dynamic set $L^l_{s_j}$ is then implemented by the insertion of all check nodes within layer l into an RB tree, and all these RB trees are arranged into an array according to the label l to represent L_{s_j} as shown in the right part of Fig. 1. The RB-tree array sorts its elements (RB trees) according to the *dist* since all the check nodes within an RB tree have the same *dist*. The check nodes in an RB tree are sorted mainly according to their *deg*. Considering the fact that the check nodes within the same layer may have the same *deg* but different *id*, the following "less than" comparison rule is defined to sort the check nodes during the insertion process. For two check nodes, c_i and c_j, we can judge whether c_i<c_j by the following procedure:

```
if $(c_i.deg \;==\; c_j.deg)$
    return $c_i.id < c_j.id$;
else
    return $c_i.deg < c_j.deg$.
```

Also, to trace the *deg* variation of every check node, another RB tree T_g is built to sort all the check nodes globally according to the *deg* attribute using the same comparison rule.

3. Implementation of the PEG algorithm

We describe the RB-tree-array-based implementation of the PEG algorithm for constructing a Tanner graph with m check nodes and n variable nodes in the following C-style pseudo-code:

```
Initialize $T_g$ as a null tree;
for $i$=0 to $m-1$ {
    $c_i.id = i,\ c_i.dist = -1,\ c_i.deg = 0$;
    Insert $c_i$ into $T_g$;
}
for $j$=0 to $n-1$ {
  Step 1. Link $c_i$ to $s_j$, where $c_i$ is a check node in $T_g$ with the
          lowest degree, and update the location of $c_i$ in $T_g$;
  Step 2. Expand the graph from $s_j$ to construct the RB-tree
          array $L_{s_j}$ and record every check node's shortest
          distance from $s_j$ in its *dist* field;
  Step 3. For $k = 1$ to $d_j-1$ {
        Step 3.1. Link $c_i$ to $s_j$, where $c_i$ is a check node with
                  the lowest degree in the furthest layer, and
                  update the location of $c_i$ in $T_g$;
        Step 3.2. From $c_i$, recursively adjust $L_{s_j}$;
  }
  Step 4. Clear the RB tree array $L_{s_j}$, and reset the *dist*
          field of every check node to–1;
}
Destroy the tree $T_g$;
```

There are several subtle points in this algorithm that need further comment. The update of T_g in step 1 and step 3.1 is achieved through the following steps: locate and delete c_i from T_g, let $c_i.deg = c_i.deg + 1$, and then re-insert c_i into T_g. The adjustment of L_{s_j} in step 3.2 is performed in a similar manner. If a check node c_i in $L^l_{s_j}$ gets a new distance l' and $l' < l$, locate and delete c_i from $L^l_{s_j}$, let $c_i.dist = l'$, and then re-insert c_i into $L^{l'}_{s_j}$. Whenever we encounter multiple choices of c_i in step 1 or step 3.1, we can adopt the same strategy as that adopted in [1].

III. Complexity Analysis and Comparison

Since the PEG algorithm builds a Tanner graph in an edge-by-edge manner, both the RB-tree-array-based implementation presented here and the approach in [1] have linear complexity with respect to the total number of edges in the Tanner graph. However, the average effort of placing one edge differs in the two methods. Compared with the approach in [1], the RB-tree-array-based implementation reduces the computational effort in two respects. First, the RB-tree data structure guarantees that the basic dynamic-set operations, such as searching, inserting, and deleting take $O(\log m)$ time in the worst case [4], where m is the number of the check nodes. The arrangement of the RB trees into an array further reduces the searching complexity as it decouples the sorting of check nodes according to the *dist*

ETRI Journal, Volume 31, Number 2, April 2009 Lin Chen et al. 241

from that according to the *deg*. The operation of choosing one check node has logarithmic complexity rather than linear complexity in the indicator-based implementation. Note that the adoption of an RB-tree array to manage L_{s_j} also facilitates dynamic adjustment of L_{s_j}. The adjustment of a single check node can be finished in $O(\log m)$ time since it is realized through three operations on a RB tree. Second, in the RB-tree-array-based implementation, the tree-expansion of the Tanner graph needs to be performed only once (in step 2) for every variable node. When L_{s_j} is changed by the newly added edge, we perform the dynamic adjustment of L_{s_j} rather than re-expanding the Tanner graph from the beginning. Because only some of the check nodes whose locations are influenced by the newly added edges $E_{s_j}^k \left(1 \le k < d_{s_j}\right)$ and the adjustment of a single node have logarithmic complexity, the dynamic adjustment of L_{s_j} is more efficient than the reconstruction of L_{s_j} through a new tree-expansion of the Tanner graph [1]. These two measures effectively reduce the complexity of the PEG algorithm without any performance loss.

To compare the complexity of the RB-tree-array-based implementation with the indicator-based one in [1], we constructed two groups of rate-1/2 LDPC codes with different degree distributions. These two degree distributions, with maximum variable node degrees of 20 and 15, respectively, are all from table II in [5]. For a fair comparison, we always chose the check node with the smallest *id* in both implementations when there existed multiple choices for c_i, which ensured that the Tanner graphs constructed by these two methods would be the same. A comparison of the construction time is shown in Fig. 2. The curves marked with hollow symbols (circle or triangle) are for the indicator-based method in [1], and the ones with solid symbols are for our method. The curves of our presented RB-tree-array method increase much more slowly than those of the indictor-based method. For the construction of the 10^4 long LDPC codes with the degree distributions of 20 and 15, it takes 2,393 seconds and 1,888 seconds, respectively, with the method in [1]. By contrast, only 153 seconds and 141 seconds are needed with our method.

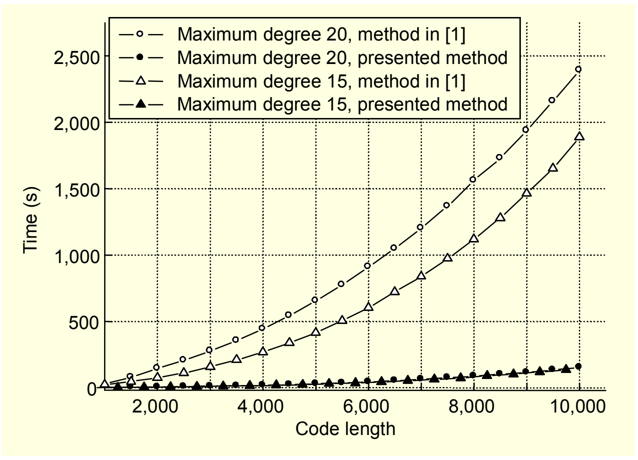

Fig. 2. Construction time comparison of different implementations of PEG algorithm.

IV. Conclusion

We presented a novel low-complexity implementation of the PEG algorithm which takes advantage of the RB-tree-array data structure. On the basis of this data structure, an efficient tree-expansion strategy was adopted. The complexity analysis and simulation results demonstrated that the proposed implementation reduces the computational effort effectively, and it can be used to quickly construct long LDPC codes with large girth.

References

[1] X.-Y. Hu, E. Eleftheriou, and D.-M. Arnold "Regular and Irregular Progressive Edge-Growth Tanner Graphs," *IEEE Trans. Inform. Theory*, vol. 51, no. 1, Jan. 2005, pp. 386-398.

[2] M. Andersson, *Graph Optimization for Sparse Graph Codes*, M.S. thesis, KTH, Stockholm, Sweden 2007, Available online: http://www.ee.kth.se/php/modules/publications/reports/2007/IR-SB-XR-EE-KT%202007:001.pdf

[3] S. Mohan and J. Anderson, "Computationally Optimal Metric First Code Tree Search Algorithms," *IEEE Trans. Comm.*, vol. 32, no. 6, June 1984, pp. 710-717.

[4] T. Cormen et al., *Introduction to Algorithms (Second Edition)*, MIT Press, Cambridge, Massachusetts London, 2001.

[5] T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke, "Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, Feb. 2001, pp. 619-637.