

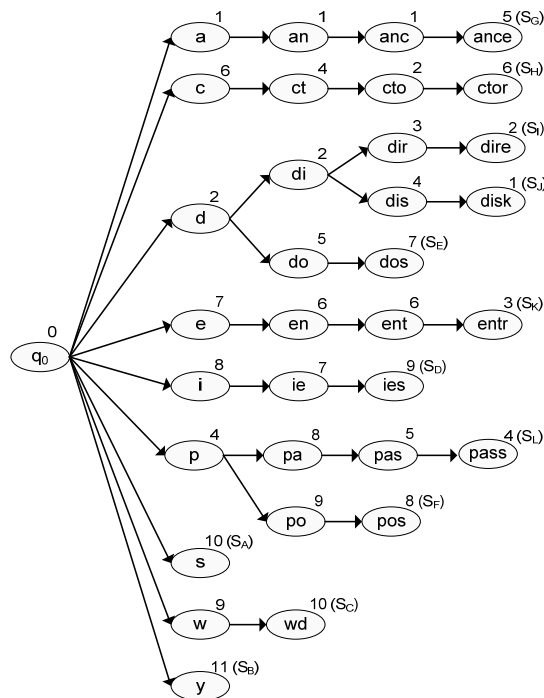
Construction of lookup tables for the P-AC string matching engine

Derek Pao
 Electronic Engineering Department
 City University of Hong Kong

This explanation notes aim to provide more details on the procedure to construct the lookup tables required by the P-AC string matching engine [1].

In the following discussion we assume the state transition graph of the pipeline unit (PU) $M_p(\Gamma)$ and the transition graph of the aggregation unit (AU) $M(A)$ have been constructed.

We shall first describe the steps to construct tables LT_0 to LT_2 of the PU. The example given in [1] is used to illustrate the process.



$M_p(\Gamma_k)$ for the sample signature set. The number next to a node represents the assigned physical state ID (physical segment ID), and the logical segment ID is placed inside the bracket.

Table LT_0 stores the out-going transition edges from the root node q_0 . The program will first traverse all the level-1 nodes in $M_p(\Gamma)$, i.e. direct descendants of q_0 . For each level-1 node, the program will compute the corresponding mask-vector and the required block size (number of entries required in table LT_1). For example, for the node $\langle a \rangle$, its fanout is 1. Hence, the mask-vector is '0000 0000' and the required block size is 1. For node $\langle d \rangle$, its fanout is 2. The mask-vector is '0000 0010', and the required block size is 2.

After finding out the required block sizes of all level-1 nodes, the program will process the level-1 nodes in a second pass ordered by the block size in descending order. A node will be mapped to a vacant block of the required size in LT_1 such that the starting address of the allocated block is an integral multiple of the block size. The physical state ID will be equal to the starting address of the allocated block.

So, based on the above procedure, node <d> is mapped to address 2 (occupies locations 2 and 3 in LT_1), node <p> mapped to address 4 (occupies locations 4 and 5), node <a> mapped to address 1, node <c> mapped to address 6, and so on.

The same procedure is used to construct lookup tables LT_1 and LT_2 . There is only one additional constraint needed to be taken into consideration in the assignment of the state ID to output nodes (nodes that represent a pattern with 1 to 3 characters). The state IDs of these output nodes should be distinct. Since the number of patterns with 1 to 3 characters is very small, this requirement can be fulfilled easily.

The more difficult task will be the construction of table LT_3 . The next-state (*ns*) value in LT_3 represents the segment ID and/or pattern ID of a 4-character pattern. The assignment of segment IDs should facilitate the construction of memory efficient lookup tables for the AU. Hence, the assignment of the state IDs to level-4 nodes of $M_p(\Gamma)$ will be delayed after the program analyses the transition graph of the AU $M(A)$.

The AU will maintain 2 lookup tables TA_0 and TA_1 . TA_0 stores the out-going transition edges from the root q_0 , and TA_1 stores the remaining transition edges. Initially nodes in $M(A)$ are assigned a logical state ID starting from 0. The program traverses $M(A)$ and puts the logical segment IDs into groups, $G_i = \{\text{logical segment IDs appearing in output edges of node } i\}$. One can see that the membership of the groups may not be disjoint, i.e. a segment ID may be member of more than 1 group.

G_0 is the group of segment IDs that appear on the transition edges of the root node. TA_0 is a direct-indexed table accessed using the segment IDs. So, what we want to do is to assign physical IDs to members of G_0 in the range of 1 to $|G_0| + \Delta$ (where $|G_0|$ is the size of G_0 , and Δ is the number of patterns with 4 characters or less). The program maintains 3 arrays `IDmap[]`, `finalized[]` and `avail[]`, such that `IDmap[i]` = physical ID of logical segment `i`, `avail[j]` = 1 if physical ID `j` has not been taken up, `finalized[i]` = 1 if physical ID of segment `i` has been finalized.

To simplify the discussion, let's assume $\Delta = 0$. The program will reserve physical IDs 1 to $|G_0|$ for members of G_0 . The program will then process the other groups in descending order by size. Table TA_1 is implemented using the DIBS approach. Hence, we try to assign physical IDs to members of a group such that the physical IDs can be differentiated by the least significant bits of the ID values. The pseudo code for processing group G_i is outlined below:

```

//pusedo code to assign physical IDs to members in Gi, where |Gi| > 2

k = size of group Gi;
k2 = k rounded up to the nearest value that is a power of 2;
//let k2 >= k and k2 = power(2, d)

s = starting address of a vacant block of size k2 in avail[] such that
    s > |G0| and s is an integral multiple of k2;

//first pass
for each member m of Gi
{
    if (finalized[m] == 1) //physical ID of m has been finalized
    {
        r = value of the least significant d bits of IDmap[m];
        avail[s+r] = 0; //reserve this physical ID
    }
}

//second pass
for each member m of Gi
{
    if (finalized[m] == 0) //physical ID not yet finalized
    {
        for (r = 0; r < k2 && avail[s+r] == 1; r++)
            ;

        avail[s+r] = 0;

        if (m is a member of G0)
        {
            //least significant bits of the physical ID of m = r
            if (r > 0)
                p = r;
            else
                p = k2;

            while (p < |G0| && avail[p] == 1)
                p += k2;

            if (p < |G0|)
            {
                finalized[m] = 1;
                IPmap[m] = p;
                avail[p] = 0;
            }
        }
        else
        {
            finalized[m] = 1;
            IPmap[m] = s+r;
        }
    }
}

```

The assignment of physical IDs to groups with 1 or 2 members is not constrained. After assigning the physical IDs to members of other groups, the program will go back to process members of G_0 . For each member of G_0 that has not been assigned a physical ID, it will be assigned an available value within the range 1 to $|G_0|$.

After the physical segment IDs have been finalized, table LT_3 can be constructed. Before tables TA_0 and TA_1 can be built, we need to compute the physical state ID for the nodes in $M(A)$. The program will determine the mask-vector and required block size for each node in $M(A)$, except the root. Nodes in $M(A)$ is assigned physical state ID in the order specified in the paper [1].

1. internal node, output state, $CMG = 0$
2. terminal node, output state, $CMG = 0$
3. internal node, $CMG = 0$
4. terminal node, $CMG = 0$
5. internal node, output state, $CMG > 0$
6. terminal node, $CMG > 0$
7. internal node, output state, $CMG < 0$
8. internal node, non-output state, $CMG < 0$
9. terminal node, output state, $CMG < 0$

There are, however, two minor refinements to the data structures. The format of the state ID is refined. In [1], the state ID format is T-CT-00-address_offset for conditional output states, and T-CT-bb-address for other states (where the address value is equal to CT-bb-address_offset, and $bb \neq 00$). We propose to make two refinements such that we can have better flexibility in the assignment of physical state ID.

First, we add one more status bit (CO) to each entry in tables TA_0 and TA_1 . If $CO = 1$, then the node is a conditional output state, and the lower-order bits will be used to access one or more conditional match tables. If $CO = 0$, then the state is not a conditional output state, and the state ID will not be used to access any conditional match table.

Second, the position of the 2 CT bits is swapped with the pair of bits bb. The new state ID format is T-bb-CT-address_offset for conditional output state, and T-address for other states. As long as the state IDs are unique, there is no additional requirement on the bit values of the address field. By making these two refinements, we can have more compact lookup tables.

Reference

1. Derek Pao, Wei Lin and Bin Liu, "A memory efficient pipelined implementation of the Aho-Corasick string matching algorithm", ACM Trans. on Architecture and Code Optimization.