Data Structures and Algorithms for Calculating Reuse Distance

Date: 11/16/2012 • GTID: 902818088

Definition: Reuse Distance

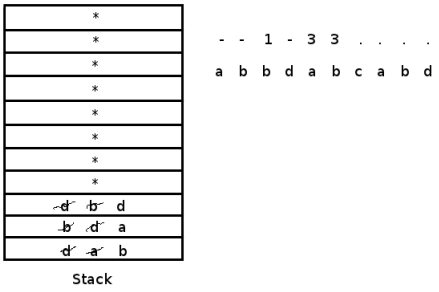
Reuse distance is the number of distinct memory references that have happened between two memory references to the same location. In short, it tells you when was the last time you used this address.

Method 1: Stack based Naive Method

As the heading suggests, we use a stack for calculating reuse distace in this method. As and when we encounter an address, we search for that address in the stack. If not found, we assume that the reuse distance is infinity since this is the first time we have encountered this memory address. We also push this new distinct address on top of our stack.

If while searching for this address in the stack, we encounter that a same address is present at some location in the stack, we delete that address from the stack and push our new address on top of the stack. The deletion is done three steps. The first step is to temporarily pop all the elements that are above our searched element in the stack. In the second step, the searched address itself is popped out and discarded and the final step, all the poppod elements are pushed back in the stack in a reversed order to their popping to maintain the original order of those addresses.

The reuse distance is calculated by incrementing a counter for every element/address that was temporarily popped from the stack. The following example illustrates this algorithm:

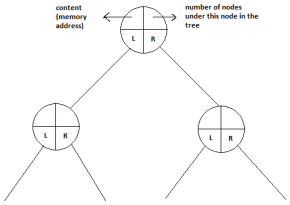


Consider the stack shown in the figure. Here, * indicated addresses that have already been pushed in the stack and are all distinct. Now consider that the algorith encounters following stream of memory addresses. First it encounters 'a' and searches the stack for address 'a'. Since 'a' is not already present in the stack, it is pushed at the top of the stack and its reuse distance is marked as '-' meaning infinity. Similar thing happens with 'b'. Now one more 'b' come and the seach in the stack yields a match. In this case, the 'b' matched in the stack and all elements above 'b' (none in this case) are popped from the stack and the number of elements that are popped is calculated (in this case 1). So, 1 is the reuse distance of 'b'. The new 'b' gets pushed on the top of the stack. If you follow this algorithm, you will end up with the stack shown in the figure.

Output of this algorithm is reuse distance at any time and colocation of all these distances gives you a reuse distance histogram. This naive method takes O(N) time and hence is too slow.

Method 2: Tree based method

This is another data structure for calculating reuse distance which is more efficient than stack. The tree that use stores four variables in the strcture of its node. Two of these four variables are basic pointers to the left and right child. One of these variables is the content i.e. the memory address. And the last and most important information that we store at each node is the number of nodes under this node in the tree.

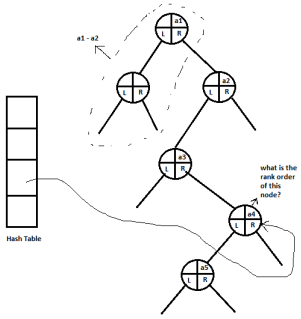


But, to be able to calculate reuse distance, the tree must satisfy an invariant at all time. This invariant is as follows:

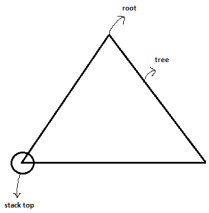
At any time t, the result of in-order traversal of this tree should be same as the result of listing all the addresses in the stack from bottom up.

You should maintain the tree in such a way to satisfy the above invariant. The invariant is used to ensure that both data structures (stack and tree) give the same output.

One thing to remember here is that, even though it looks like one, this tree is not a binary search tree based on the key (content field or memory address) but it just happens to support our invariant. hence to search in the tree, we can not use BST properties. We can have something like a separate hash table that points to nodes in the tree. Once we are at a particular node, we will also need a parent pointer to go up and travel up and down in the tree. This is shown in the diagram below.

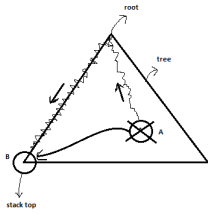


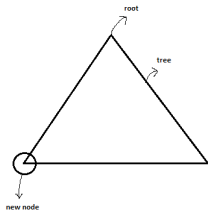
To calculate the reuse distance once a particular node is found, we only need to follow the parent pointer to go back all the way to root and compute distances (ranks) along the way. There is no need to walk down again from the root. Once the rank (distance) is found, delete that node from the tree and move it to the extreme left end of the tree which actually corresponds to the top of the stack.



The tree needs rebalancing once we delete any node from the tree. This is also called L-R dance. While rebalancing the tree, we need to change the "number of nodes under this node" field of all the nodes that are affected by this deletion.

To move node from point A to point B, there will be two paths of destructions. One going up and other coming down. But if the address is unique, then we can just add it to the tree at the extreme left and there will be just one path of destruction which is shown below.





This is a much more efficient algorithm which works in O(log N) time. This concludes out discussion on augmented data structures.

Associative Rule Mining

Assiciative Rule Mining (ARM) is a database concept that derives its solution from network algorithmics. The problem at hand is as follows:

Suppose you have a huge database table with 100,000 columns and billions of rows. For example, lets assume that it is a table from the database of walmart. The table has columns such as milk, cheese, serials etc. i.e. all the products that are sold at walmart and each cell is either 1 or 0 depending on whether that particular product was purchased in that transaction.

In this case our objective is to find for every pair of products, is there a definitive correlation or trend between the two? For example, for products like milk and cereals, is there an association between the two, does majority customers buy these two products together, does one go with the other?

This is a simple case of bit-wise and of these two columns and then finding out how many ones are present in the result. But to do this for every pair of columns and for billions of rows is a very time consuming task and would take literally years. But with our smart algorithm adapted from newtwork algorithmics, it takes onyl few hours.

We will look at that algorithm in the next class.