Notes 22

We now consider a natural problem that arises in many applications, particularly in conjunction with suffix trees, which we will study later. Suppose we have a rooted tree T with n nodes. We would like to be able to answer questions of the following form: what is the lowest common ancestor of nodes u and v; that is, what is the common ancestor of u and v closest to the root?

In this setting, we will not be answering a single questions, but many questions on the same fixed tree T. If we are given the tree T in advance, we can design an appropriate data structure for answering future queries. Our algorithm will therefore be measured on several criteria. Of course one important criterion is the *query time*, or the time to answer a specific query. However, a second consideration is how much *preprocessing time*, or time to set up the data structure, is required to answer the questions. A third related aspect to study is the memory required to store the data structure.

For example, a trivial algorithm for the problem is to consider each pair of vertices, and compute their lowest common ancestor by following both paths toward the root until the first shared vertex is found. Then all the the answers can be stored in a table. There are $\binom{n}{2}$ pairs of vertices, so our table will require $\Theta(n^2)$ space. Queries can be answered by a table lookup, which is constant time. Preprocessing, however, can require $\Theta(n^3)$ time.

The problem of designing an appropriate data structure for this is called the Lowest Common Ancestor (LCA) Problem. We will show that there is an algorithm for LCA that require only linear preprocessing time and memory, but still answers any query in constant time! This result is as efficient as we could hope for.

We will reduce the LCA problem to a seemingly different but in fact quite related problem, called the Range Minimum Query (RMQ) Problem. The RMQ problem applies to an array A of length n of numbers. We would like to be able to answer questions of the following form: given two indices i and j, what is the *index* of the smallest element in the subarray $A[i \ldots j]$? Again, we may prepocess the array A to derive some alternative data structure to answer the questions quickly. There is a trivial solution for the RMQ problem completely similar to the one above for the LCA problem.

22.1 Reduction: From LCA to RMQ

How to we convert an LCA problem to an RMQ problem? Note that we must do the conversion in linear time, if we are going to totally complete the preprocessing in linear time for the LCA problem.

Linear time suggests that we want to do a tree traversal. In fact, the observation we will use is that the LCA of nodes u and v is just the shallowest node encountered between visiting u and v during a depth first search of the tree starting at the root. So let us do a DFS on the tree, and we can record in an array V the nodes we visit. An example is shown in Figure 1. Notice each node can appear multiple times, but the total length of the array is $2n-1$, where n is the number of nodes in the tree. Each of the $n-1$ edges yields two values in the array, one when we go down the edge and one when we go up the edge. The first value is the root. Also, from now on we will refer to each node by its number on the DFS search.

We will also require two further arrays. The *Level Array* is derived from V; $L[i]$ is the distance from the root of $V[i]$. Adjacent elements in L can only differ by $+1$ or -1, since adjacent steps in the DFS are connected by an edge. Finally, $R[i]$ is the representative array; $R[i]$ contains the first index of V that contains the value i. (Actually, any occurrence of i can be stored in $R[i]$, but we might as well choose a specific one.)

Clearly, to compute $LCA(u,v)$ it suffices to compute $RMQ(R[u],R[v])$ over the array L. This gives us the index of the shallowest node between u and v, and the array V can be used to determine the actual node from the index.

22.2 Solutions for RMQ

We first note that we can do better than the naive $\Theta(n^3)$ preprocessing time for RMQ on an array A by doing a trivial dynamic programming, using the recurrence

$$RMQ(i,j) = A^{-1}[\min(A[RMQ(i,j-1)],A[j])].$$

Here we are using convenient notation. Clearly $\min(A[RMQ(i,j-1)],A[j])$ gives the value $A[k]$, where $A[k]$ is the smallest value that in the subarray $A[i\dots j]$. However, we want the index of this value. We use the notation A^{-1} to represent that we want the index of this value; note that if multiple indices have this value, we do not particularly care which index we obtain. Each table entry can be calculated in constant time by building the table in order of ranges $[i,j]$ of increasing size, leading to preprocessing time $\Theta(n^2)$.

In fact, we can reduce our table size and memory using a different dynamic program, and by using a few additional operations per query. Let us create a table $M(i,j)$ such that $M(i,j) = A^{-1}[\min_{k\in[i,i+2^j)} A[k]]$. That is,

$M(i,j)$ contains the location of the minimum value over the 2^j positions starting from i. This table has size $O(n\log n)$, and it can easily be filled in $O(n\log n)$ step by using dynamic programming, based on the fact that $M(i,j)$ can be determined from $M(i,j-1)$ and $M(i+2^{j-1},j-1)$.

How do we use the $M(i,j)$ to compute $RMQ(i,j)$, if j is not a power of 2? We may use two overlapping intervals that cover the range $[i,j]$ as follows. Let $k=\lfloor\log(j-i+1)\rfloor$, so that 2^k is the largest power of 2 such that $i+2^k\le j+1$. Then $RMQ(i,j)=A^{-1}[\min A[M(i,k)],A[M(j-2^k,k)]]$, and this can be computed in constant time from the M.

We have shown that we can achieve preprocessing time and memory size $\Theta(n\log n)$ while maintaining constant query time. Interestingly, this method can be enhanced so as to require preprocessing time and memory size $\Theta(n\log\log n)$ through a recursive construction. (This will be an exercise.) In practice, such a result would probably be good enough – $\log\log n$ is quite small for reasonable values of n. By continuing the recursive construction for further levels, we could even achieve $\Theta(n\log\log\log n)$ preprocessing time and memory size, and so on for any fixed number of logs, while maintaining constant query time. However, this recursive construction would add significant complexity to an actual program, and it still would not lead us to a linear preprocessing time solution.

22.3 ± 1 RMQ

In order to achieve linear preprocessing, we will use an additional fact about the RMQ problem we obtain from the reduction from LCA. Recall that our RMQ problem is on the Level Array obtained from the LCA problem. The Level Array has one additional property that we are not yet taking advantage of: each entry differs from the previous entry by $+1$ or -1. We can take advantage of this fact to split the RMQ problem into a different set of small subproblems in such a way that we can avoid some work by doing table look-ups.

The split works as follows: partition A into blocks of size $\frac{\log n}{2}$. Let $X[1,\ldots,2n/\log n]$ and $Y[1,\ldots,2n/\log n]$ be arrays such that $X[i]$ stores the minimum element in the ith block of A, and $Y[i]$ stores the position in the ith block where the element $X[i]$ occurs. Now to answer an RMQ query for indices i and j with $i<j$ on the array A, we can do the following:

1. If i and j are in the same block, we can perform an RMQ on this block. Notice that this requires that each block be preprocessed.
2. If i and j are in different blocks, we have to compute the following values, and take the minimum of them:

(a) The minimum from position i to the end of i's block.

(b) The minimum from the beginning of j's block to position j.

(c) The minimum of all blocks strictly between i's block and j's block.

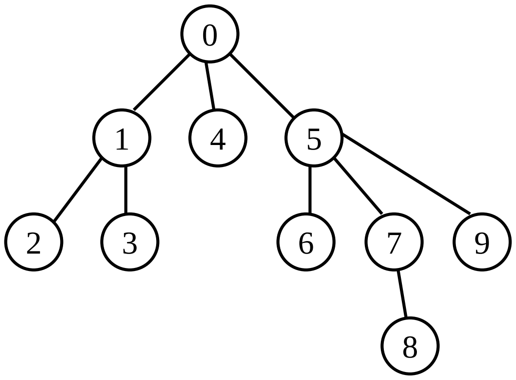
Steps 2a and 2b also require that we preprocess for RMQ queries on each block. Step 2c requires that we perform an RMQ over the array X. Assuming we have done all this preprocessing, the total query time is still constant. However, if we preprocess each block in order to do RMQ's, we have not saved on the running time. We need a faster way to deal with preprocessing each block.

How can we possibly avoid preprocessing each block separately? We use the following observation. Consider two arrays X and X'. Suppose that these two arrays differ by a constant at each position; for example, the arrays might be $1,2,3,4,3,2\dots$ and $3,4,5,6,5,4\dots$ and. Then the RMQ answers, which give the *index* of the minimum element, will be the same for these two arrays. Hence we can "share" the preprocessing used for these two arrays!

Another way to explain this is that in the ± 1 RMQ problem, the initial value of the array does not matter, only the sequence of $+1$ and -1 values are necessary to determine the answer. Now, how many different such sequences are there? Since there are only $\log n/2$ elements in a block, there are only $(\log n/2) - 1$ values in the sequence of $+1$ and -1 values. Hence there are only $2^{(\log n/2)-1} = \sqrt{n}/2$ possible sequences. This number is so small, we can afford to compute and store tables for every possible sequence! Even if we use quadratic preprocessing time and memory, these tables would take time $O(\sqrt{n}\log^2 n)$ to preprocess and $O(\sqrt{n}\log^2 n)$ memory. For each block in A, we have to determine which table to use; this can easily be done in linear time.

22.4 Back to the standard RMQ

We have shown that ± 1 RMQ problems can be solved with linear time preprocessing, and therefore we have a linear time preprocessing solution for LCA. What about the general RMQ problem? It turns out that we can also reduce the RMQ problem to the LCA problem in linear time. So we can obtain a linear time solution the general RMQ problem, by turning it into an LCA problem, and solving that as a ± 1 RMQ problem! The details of this reduction are omitted here.



V: 0 1 2 1 3 1 0 4 0 5 6 5 7 8 7 5 9 5 0
L: 0 1 2 1 2 1 0 1 0 1 2 1 2 3 2 1 2 1 0
R: 0 1 2 4 7 9 10 12 13 16

Figure 1: Changing an LCA problem into an RMQ problem.