Range Minimum Query I

Study Notes by Ruijie Fang

1 Preliminaries

Problem 1 (Range Minimum Query (RMQ))**.** Given an array A of size n and m queries L_i and R_i, $1 \leq i \leq m$, report the minimum element in $A[L_i...R_i]$ for each query.

Problem 2 (Lowest Common Ancestors (LCA))**.** Given a rooted tree τ with nodes labeled $1...n$, find a shared common ancestor c of a and b that is farthest from root (or, has the maximum depth).

Why are these problems important? RMQ was first proposed by J. L. Bently in the 1980s. LCA was a classic problem in theoretical CS for a very long time, and it was known to be quite difficult because there was no algorithm matching its theoretical lower-bound (until Farach-Colton and Bender published a famous paper which reduced LCA to ± 1-RMQ in early 2000s). There are numerous competitive programming problems that involve either online or offline RMQ/LCA processing.

2 RMQ to LCA using the Cartesian tree

The Cartesian tree is a min-heap whose in-order traversal returns the original array, A. A Cartesian tree can be constructed in $O(n)$-time using the all nearest smallest values algorithm. For each index i, the left child is the smallest largest value that that is to its left in the original array, and its right child will be added later.

Algorithm 3 (Cartesian tree construction)**. Initialize** $P[1...n]$ *to 0; // parent array,* $P[i]$ *is the parent of node* i *in the Cartesian tree*

Initialize $S := Stack()$*; // a stack of indices*

For $i := 1; i \leq n; i+ = 1$

$l := 0;$

While $\neg S.empty() \wedge A[S.top()] \geq A[i]$

$l := S.top(); S.pop();$

If $\neg S.empty()$

$P[i] := S.top();$

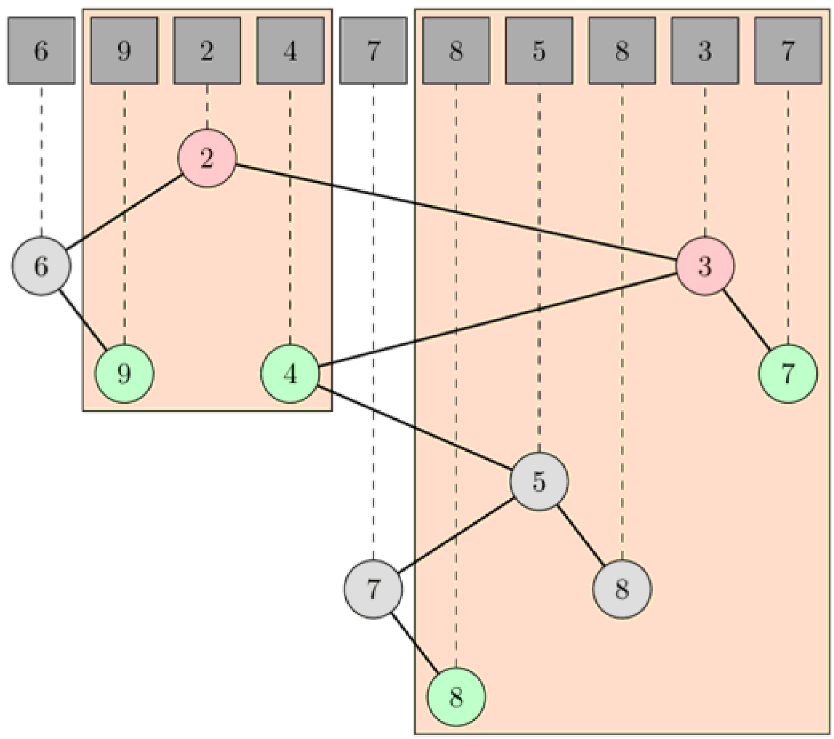
If $l > 0$

$P[l] := i;$

$S.push(i);$

Since only n elements are pushed onto the stack, the algorithm runs in $O(n)$. We may find the root by iterating through P and finding the entry that has value 0 (the root has no parent).

Since small values are towards the top of the Cartesian tree, it is not hard to see that the value of RMQ(L,R) on a cartesian tree is equal to finding the LCA of L and R.



3 Offline RMQ using All Nearest Values

This interesting algorithm achieves near-linear-time in offline range minimum query using the Union-Find structure. The Union-Find structure has an operation called $findSet(i)$ which retrieves the parent of element i. We assume that the $findSet(\cdot)$ operation on a Union-Find structure of size n takes $\alpha(n)$-time, the $\alpha(\cdot)$ function being the inverse Ackermann function. This is called Arpa's Trick in the competitive programming community. The algorithm is as follows:

Algorithm 4. *ArpaRangeMinimumQuery(*$B[\cdot]$*,* $A[\cdot]$*,* $P[\cdot]$*)*

Precondition *$B[\cdot]$ is a bucket for queries. Each query is a pair $\langle L, idx\rangle$. The queries stored in $B[i]$ has right endpoint $R = i$. idx specifies the index which the answer to the query will be stored in. The parent array $P[\cdot]$ stores the union-find structure; $P[i]$ records the parent of i.*

Postcondition *A correctly computed $A[\cdot]$. Entry i in $A[i]$ stores the result of query $\langle L, R, i\rangle$.*

1. *Initialize a stack S.*
2. *For $i := 0$ to n:*
   (a) *While $\neg S.empty() \land A[S.top()] > A[i]$:*
      i. *$P[S.top()] := i$;*
      ii. *$S.pop()$;*
   (b) *$S.push(i)$;*
   (c) *For Each $\langle L, idx\rangle$ in $B[i]$:*
      i. *$A[idx] := A[findSet(L)]$;*

The correctness of the Algorithm 1 relies on the behaviour of the union-find structure. At step i in the loop, the parent of each element $j < i$ in the Union-Find structure is set to the minimum element in range $[j, i]$. Therefore finding the parent of $L < i$ results in finding the minimum value in range $[L, i]$. Since the outer loop is monotone, each index of A is only stored in S once and each query in $B[i], 1 \leq i \leq n$ is only processed once. This results in the $O(\alpha(n)n+m)$ runtime.

The preprocessing phase involves filling the bucket $B[\cdot]$ with queries. This is done in $O(n + m)$-time.

What's interesting: This algorithm basically merged together Tarjan's offline Lowest Common Ancestors algorithm with the all-nearest-smaller-values algorithm. Tarjan's LCA algorithm works by operating a union-find set on top of a DFS spanning tree, with the guarantee that LCA(u,v) can be answered once the algorithm had already visited u and is processing v, and that the representative of u and v will be their lowest common ancestor.

The preprocessing phase involves filling the bucket $B[\cdot]$ with queries. This is done in $O(n + m)$-time.

Algorithm 5. *PrecomputeBuckets($Q[\cdot]$, $B[\cdot]$)*

Precondition *$B[\cdot]$ is a bucket for queries. $Q[\cdot]$ is an array of queries of form $\langle L, R\rangle$.*

Postcondition *Query i at $Q[i] = \langle L_i, R_i\rangle$ will be stored in $B[R_i]$ as $\langle L, i\rangle$.*

1. *Initialize $B[\cdot]$ to hold n buckets;*
2. *For Each $\langle L_i, R_i\rangle$ in Q:*
   (a) *$B[R_i].add\,\langle L_i, idx\rangle$;*

Step 1 takes $O(n)$-time, and step 2 takes $O(m)$-time. This concludes the runtime of this algorithm as $O(m+n)$ preprocessing and $O(\alpha(n)m+n)$.

The Union-Find structure isn't completely necessary in the idea of the algorithm, and combined with the stack, binary search may be used to result in an $\langle O(m+n), O(n \log m)\rangle$-time offline RMQ algorithm.

4 Sparse Table for $\langle O(n \log n), O(1)\rangle$ Offline RMQ

Tarjan's Sparse Table algorithm is a dynamic programming technique for solving RMQ in $O(n \log n)$ preprocessing time and spends $O(1)$ time for each query. Tarjan's idea is based on the following fact:

Fact 6. *A sequence $[i...i+2^k-1]$ can be split into two sequences of length 2^{k-1}: $[i...i+2^{k-1}-1]$ and $[i+2^{k-1}...2^k-1]$.*

Let $T[i,j]$ denote the minimum value of $A[i...i+2^j]$:

$$T[i,j] := \begin{cases} \min\{T[i,j-1], T[i+2^{j-1}, j-1]\} & j > 0 \\ A[i] & j = 0 \end{cases}$$

Since the maximum j value is $\log_2 n + 1$, the dynamic programming recurrence works in $O(n \log n)$-time. This results in the following preprocessing algorithm:

Algorithm 7. *InitSparseTable($A[\cdot]$, $T[\cdot]$, n)*

Precondition *$A[\cdot]$ is an array of n elements and T is a 2D table of size $n \times (\log_2(n)+1$.*

Postcondition *Constructs $T[\cdot]$ table, entry $T[i,j]$ denotes the minimum value in $A[i...i+2^j-1]$.*

1. *For $i := 0$ to $n-1$:*
   (a) *$T[i,0] := A[i]$;*
2. *For $j := 1; 2^j \leq n; j := j+1$:*
   (a) *For $i := 0; i+2^j-1 < n; i := i+1$:*
      i. *$T[i,j] := \min\{T[i,j-1], T[i+2^{j-1}, j-1]\}$;*

Having preprocessed the table, we can now answer queries. Recall that each query takes the form of $(L,R), R \geq L$. Let $k := \log_2(R-L+1)$. The precomputation naturally leads us to the fact that $[L,R]$ is covered by $[L, L+k]$ and $[R-k, R]$. Since we're finding the minimum in the region, repeated computation doesn't matter.

Algorithm 8. *QuerySparseTable($T[\cdot]$, L, R):*

Precondition *Constructed $T[\cdot]$ sparse table with $T[i,j]$ denoting the minimum value in range $[i...i+2^j]$, query parameters $R \geq L$.*

Postcondition *Returns the minimum value in range* $[L...R]$.

1. $k := 0$;
2. *While* $2^{k+1} \leq R - L + 1$*:* $k := k + 1$;
3. *Return* $\min\{T[L, k], T[R - 2^k + 1, k]\}$; *(Takes the minimum in ranges* $[L...2^k - 1]$ *and* $[R - 2^k + 1...R]$*)*

The while loop in step 2 takes $O(\log(n))$-time. We can precompute the logarithms in $O(\log(n))$ to make this constant time:

Algorithm 9. *CalculateLog2(*$\log[\cdot]$*, n):*

1. $\log[1] := 0$;
2. *For* $i := 2$ *to* $\log_2(n) + 1$*:*
   (a) $\log[i] := \log[i/2] + 1$;

This concludes our description for Tarjan's Sparse Table algorithm.

5 Segment Trees

Segment trees support dynamic RMQ/RSQ operations with point and range updates. A segment tree $\tau[\cdot]$ recursively divides each interval into a left interval and a right interval evenly. Smaller intervals are stored further down the tree while the root of τ is the original interval $[1...n]$. Each node in τ is associated with information that will help compute RSQ/RMQ problems.

We number the nodes in τ from top to bottom, left to right, starting from 0. The left child of a node i is $2i + 1$ and the right child is $2i + 2$.

Algorithm 10. *SegTreeRMQ(*τ*, i, L, R,* Q_L*,* Q_R*)*

Precondition τ *is a segment tree with an array* $\min_v[\cdot]$ *storing RMQ information. Node i stores interval [L, R]. The query is* $[Q_L, Q_R]$. *Call with* $i := 0$*,* $L := 0$*,* $R := n$.

Postcondition *Returns the RMQ result of query* (Q_L, Q_R).

1. $M := L + (R - L)/2$; $ans := +\infty$;
2. *If* $L \geq Q_L \wedge R \leq Q_R$: *Return* $\tau.\min_v[i]$;
3. *If* $Q_L \leq M$*:* $ans := \min\{ans, SegTreeRMQ(2i + 1, L, M)\}$;
4. *If* $M < Q_R$*:* $ans := \min\{ans, SegTreeRMQ(2i + 2, M + 1, R)\}$;
5. *Return ans;*

The SegTreeRMQ procedure works in $O(\log n + (R - L + 1))$.

Algorithm 11. *UpdateSegTreeRMQ(*τ*, i, L, R, p, v)*

Precondition *τ is a segment tree. i is the current node index. $[L, R]$ is the current node interval, p the index and v the value of the update operation.*

Postcondition *UpdateSegTreeRMQ($\cdot$) updates τ in range $[L...R]$ after replacing element p's value with v.*

1. $M := L + (R - L)/2$;
2. *If* $L = R$:
   (a) $\min_v[i] := v$; *Return;*
3. *Else:*
   (a) *If* $p \leq M$: *UpdateSegTreeRMQ(τ, $2i + 1$, L, M, p, v);*
   (b) *Else: UpdateSegTreeRMQ(τ, $2i + 2$, $M + 1$, R, p, v);*
   (c) $\tau.\min_v[i] := \min\{\tau.\min_v[2i+1], \tau.\min_v[2i+2]\}$; *(Update current node)*

The update procedure also works in $O(\log n)$. Given the update procedure, we can build a segment tree of n nodes in $O(n \log n)$. We can, though, write a special build procedure that builds the tree in $O(n)$.

Algorithm 12. *BuildSegTreeRMQ(τ, A, i, L, R)*

Precondition *Tree τ is an empty segment tree. $A[L...R]$ is the target array; i is the root index for $A[L...R]$.*

Postcondition *BuildSegTreeRMQ($\cdot$) builds a segment tree for $A[L...R]$.*

1. $M := L + (R - L)/2$;
2. *If* $L = R$:
   (a) $\tau.\min_v[i] := A[L]$; *Return;*
3. *BuildSegTreeRMQ(τ, A, $2i + 1$, L, M);*
4. *BuildSegTreeRMQ(τ, A, $2i + 2$, $M + 1$, R);*
5. $\tau.\min_v[i] := \min\{\tau.\min_v[2i + 1], \tau.\min_v[2i + 2]\}$;

Since each index of $A[1...n]$ is only visited once, we have $T(n) = 2T(n/2) + 1 = O(n)$ as the resulting runtime of BuildSegTreeRMQ($\cdot$).

The description for segment trees above results in a $\langle O(n), O(\log n)\rangle$-time dynamic RMQ algorithm supporting node updates.