I collaborated with: W3023339

Problem

This question explores a data structure that supports a specific set of operations on an string of n bits. Initially, all bits are set to 0. Bits can then individually be set to 1 (but not back to 0), and the value of the i^{th} bit can be obtained. As more bits are set to 1, blocks of consecutive 1s may be formed, separated by 0s (or terminating at the ends of the string). Specifically, the *BitBlocks* data structure supports the following methods:

- $Create(n)$: Initialize the data structure.
- $SetToOne(i)$: Given an index i, set i^{th} bit to 1 (even if it already is 1).
- $GetValue(i)$: Given an index i, return the value of the i^{th} bit.
- $GetBlockSize(i)$: If $GetValue(i) = 1$, return the size of the block containing the i^{th} bit; otherwise return 0.

We would like to ensure that any sequence of these operations, beginning with the initializing of the data structure, will be highly efficient. In particular, we'll allow $Create$ to take $O(n)$ time, but require any subsequent sequence of $SetToOne$, $GetValue$, and $GetBlockSize$ operations to be very fast. The data structure should be of size no worse than $O(n)$.

(a) Describe an implementation of the *BitBlocks* data structure based on a Union-Find data structure. Include descriptions of the implementations of the four methods defined above.

(b) Prove that the methods of your data structure work as specified above.

(c) Analyze the time and space complexity of your data structure and each of its methods. In particular, state and prove a result analogous to statement (4.23) [page 153] of your text. algorithm correctly solves the problem in this amount of time.

Solution

1. Assume sets can have repeating values s.t. a set $\{1, 1, 1, 1\}$ is a valid set.
   (a) $Create(n)$: set up the array Component and initialize it to $Component[i] = I$ where I is a name for a set $\{0\}$ for all i up to n.
   (b) $SetToOne(i)$: do the equivalent of $Find(i)$ to find the name of the set containing i. If this set corresponds to an initial set $\{0\}$, set its element to 1 and if there is an adjacent set in Collection containing a block of 1's then do the equivalent to $Union(A, B)$ where A is the set containing i and B is such adjacent block set. Now, if there is another block set adjacent to our union, we call $Union$ on the adjacent set and on our union. Otherwise, if the set containing i is already a set containing 1's, then leave as is.
   (c) $GetValue(i)$ find the name for set containing the i^{th} bit, and return the value of the first element in that set.
   (d) $GetBlockSize(i)$ find the name of the set containing the i^{th} bit. If the first element is a zero, return 0, else return the size of the set.
2. (a) Creating an array Component of size n initializes the data structure.
   (b) Calling $Find(i)$ will return the name of the set s containing i because of how Union-Find is implemented. If $s = \{0\}$, this means we have not yet called $SetToOne(i)$ on it. Now, we have two further cases: this set is adjacent to two blocks (...1111$\underline{0}$1111...) or this set is adjacent to only one block (...1111$\underline{0}$0000....). In the first case, we must $Union$ the set s to one of its adjacent set after changing its value to 1 and $Union$ the result to the other adjacent set. This will merge both blocks and include i. In the latter case, we only change s to $\{1\}$ and $Union$ it to its adjacent block set to extend the block. Finally, if $s \neq \{0\}$, then s must represent a block of 1's of at least size one. In this case, there is nothing to change.
   (c) If the i^{th} bit happened to be a 1, then it would be part of a set containing a block of 1's due to our implementation of $SetToOne(i)$. Otherwise, if the i^{th} bit happened to be a 0 then the set name at its position would reference a set of the form $\{0\}$. In both cases, the first element of the set corresponds to the value of i.

(d) The implementation of $SetToOne(i)$ separates our data structure into sets containing blocks of 1's and sets containing just zero. Thus, if the name of the set at position i references a set of a block of 1's, we return the size of said set. Otherwise, the set references a set of $\{0\}$ and the size of the block is zero.

3. Claim: Consider the Union-Find implementation of the ***BitBlocks*** data structure for some string S of size n, where unions keep the name of the larger set. The $GetValue(i)$ and $GetBlockSize(i)$ operations take $O(1)$ time, $Create(n)$ takes $O(n)$ time, and any sequence of k $SetToOne(i)$ operations takes at most $O(klogk)$ time.

Proof. $GetValue(i)$ and $GetBlockSize(i)$ take $O(1)$ time because looking up in an array takes constant time. $Create(n)$ takes $O(n)$ because building and populating an array of n elements takes time proportional to the number of elements. Now we must consider $SetToOne(i)$:

- The first time $SetToOne(i)$ refers to a set, pay the set \$1.
- Every time $SetToOne(i)$ changes the set name of a set, pay the set \$1.
- We must show that after k $SetToOne(i)$ operations, total payout is at most \$$O(klogk)$.
- $SetToOne(i)$ only renames the smaller set.
- So every time we pay \$1 to a set, increased the size of the set by at most twice.
- So if a set had its name changed d times, it now has size at most $2d$.
- But $2d \leq 2k$ so $d \leq log(2k)$.
- Thus each set got paid at most \$$log(2k)$ and at most $2k$ sets were paid.
- So total payout is $2klog(2k) = O(klogk)$

□