Aplicación de Segment Tree para Modificar Resistencias y Calcular Resistencia Equivalente entre Nodos en un Circuito Eléctrico

Manuel Alejandro Loaiza Vasquez
Departamento de Matemáticas
Pontificia Universidad Católica del Perú
Lima, Perú
Email: manuel.loaiza@pucp.edu.pe

***Abstract*—Having multiple modifications of resistor values in a series circuit and queries for the equivalent resistance between two nodes falls into the category of an update and range query problem. These problems are intuitively solved in $O(n)$ per update or query. Our study includes the implementation of a Segment Tree where we have queries in ranges with the associative operation of the sum in $O(\lg n)$ and also updates in $O(\lg n)$.**

1. Introducción

Un multímetro es un dispositivo que es utilizado para medir la resistencia eléctrica entre dos nodos en un circuito eléctrico. Asimismo, algunos circuitos cuentan con resistores que tienen la posibilidad de variar el valor de la resistencia. Cuando en un laboratorio se realizan experimentos, generalmente se tiene que modificar la resistencia en los resistores múltiples veces, así como calcular la resistencia equivalente entre distintos nodos dependiendo de lo que necesitemos. En un circuito con resistencias conectadas en serie tenemos que la resistencia equivalente R_{eq} de los resistores en el rango $[l, r]$ es

$$R_{eq} = R_l + R_{l+1} + \cdots + R_{r-1} + R_r.$$

El problema de medida de Klee en un espacio m-dimensional consistía en conseguir un algoritmo eficiente para obtener la medida del conjunto formado por la unión de n hiperrectángulos de dimensión m. Asimismo, se podían eliminar como añadir hiperrectángulos al conjunto y consultar la nueva medida cuántas veces querramos. Este problema fue resuelto con la creción de una estructura de datos llamada Segment Tree, la cual podemos utilizar en nuestro caso particular en dimensión uno.

2. Segment Tree

Estructura de datos que nos permite contestar consultas en rangos en $O(\lg n)$ bajo algún operador binario o función que cumpla con la propiedad asociativa. En esta estructura podemos realizar actualizaciones puntuales en $O(\lg n)$ y en ciertos casos hacer actualizacinoes en rangos en $O(\lg n)$ con una técnica especial llamada Lazy Propagation. El paradigma que utiliza esta estructura es el de **Divide y Vencerás**.

En esta aplicación utilizaremos la versión más sencilla del Segment Tree en donde tenemos consultas en rangos de la operación asociativa de la suma entre números reales y actualizaciones puntuales.

- Consultar$(l, r) : A[l]+A[l+1]+\cdots+A[r-1]+A[r]$
- Actualizar$(pos, val) : A[pos] \leftarrow val$

Para poder responder consultas rápidamente, se hará un preprocesamiento de la respuesta en algunos intervalos. Utilizaremos el paradigma Divide y Vencerás cuando querramos hallar el resultado en el segmento $[0 \ldots n-1]$, el cual lo calcularemos a partir de los resultados en los segmentos $[0 \ldots \lfloor \frac{n-1}{2} \rfloor]$ y $[\lfloor \frac{n-1}{2} \rfloor + 1 \ldots n-1]$. De esta forma se irá creando un árbol de segmentos con las respuestas precalculadas, las cuales usaremos más adelante para responder las consultas eficientemente.

2.1. Construcción

En el Segment Tree, cada nodo tendrá dos hijos, a excepción de las hojas, y el valor de cada nodo del Segment Tree se calculará a partir del valor de sus hijos. La primera interrogante que puede surgir es respecto a la cantidad de nodos que podríamos necesitar en el peor de los casos. Un límite superior lo podemos calcular de la siguiente forma:

- En el nivel cero tenemos un solo nodo.
- En el nivel uno tenemos a lo más dos nodos.
- En el nivel dos tenemos a lo más cuatro nodos.

Así sucesivamente, podremos llegar hasta el nivel $\lceil \lg n \rceil$.

$$\begin{aligned} \#\text{nodos} &\leq 1 + 2 + 2^2 + \cdots + 2^{\lceil \lg n \rceil} \\ &= 2^{\lceil \lg n \rceil + 1} - 1 \\ &\leq 2^{\lceil \lg n \rceil + 1} \\ &< 2^{\lg n + 2} \\ &= 2^{\lg n} 2^2 \\ &= 4n \end{aligned}$$

De esta manera, solo necesitaremos $O(n)$ memoria para almacenar los nodos que representarán nuestros segmentos.

La segunda interrogante que surge es respecto a cómo vamos a implementarlo. Para implementarlo recursivamente al estilo Divide y Vencerás debemos primero recordar que en cada nodo guardaremos la respuesta del segmnto que representa, por lo tanto, necesitaremos alguna forma de indexar los nodos para guardarlos en un arreglo. Al nodo raíz le asignaremos el índice 1. A partir de ahí, supongamos que un nodo tiene el índice i, entonces asignaremos a su hijo izquierdo el índice $2i$ y a su hijo derecho el índice $2i + 1$. La construcción del Segment Tree para la operación suma quedaría así:

Algorithm 1: Construir

Data: $A, id, tl, tr, tree$.
Result: Resultado precalculado en el segmento $[tl \dots tr]$ y en todos sus descendientes.
begin
  if $tl = tr$ **then**
    $tree[id] \leftarrow A[tl]$
  else
    $tm \leftarrow \lfloor \frac{tl+tr}{2} \rfloor$
    Construir$(A, 2 * id, tl, tm)$
    Construir$(A, 2 * id + 1, tm + 1, tr)$
    $tree[id] \leftarrow tree[2 * id] + tree[2 * id + 1]$
  end
end

A esta función la llamaríamos de la siguiente manera: Construir$(A, 1, 0, n - 1)$.

2.2. Actualización

La forma más simple de actualización es realizar una actualización puntual; es decir, solo modificar un elemento. Supongamos que una actualización modifica al elemento de la posición pos. Entonces simplemente nos dirigimos hacia la hoja del Segment Tree que contenga al intervalo $[pos \dots pos]$, modificamos ese nodo y luego modificamos a todos sus ancestros. Como la altura del Segment Tree es igual a $\lceil \lg n \rceil + 1$, entonces solo visitaremos $O(\lg n)$ nodos.

Algorithm 2: Actualizar

Data: $pos, val, id, tl, tr, tree$.
Result: Actualización del nodo que representa al segmento $[pos \dots pos]$ y de todos sus ancestros en $tree$.
begin
  if $tl = tr$ **then**
    $tree[id] \leftarrow val$
  else
    $tm \leftarrow \lfloor \frac{tl+tr}{2} \rfloor$
    if $pos \leq tm$ **then**
      Actualizar$(pos, val, 2 * id, tl, tm)$
    else
      Actualizar$(pos, val, 2 * id + 1, tm + 1, tr)$
    end
    $tree[id] = tree[2 * id] + tree[2 * id + 1]$
  end
end

A esta función la llamaríamos de la siguiente manera: Actualizar$(pos, val, 1, 0, n - 1)$.

2.3. Consulta

Ahora supongamos que queremos consultar en rango

$$\text{Consultar}(l, r) = A[l] + A[l + 1] + \cdots + A[r - 1] + A[r].$$

La idea es dividir el segmento $[l, r]$ en segmentos consecutivos que estén presentes en el Segment Tree. Podemos ir haciendo esto recursivamente. Resumamos el algoritmo de la consulta en tres casos:

1) El segmento del nodo en el que estamos está completamente incluido en el rango consultado. En este caso simplemente retornaremos el valor de este segmento como parte de la respuesta.
2) En caso contrario, solo parte del segmento representado por el nodo actual está dentro del segmento de la consulta. Aquí tenemos dos casos:

- Solo uno de sus hijos es parte del segmento de la consulta. En este caso solo es necesario ir hacia este hijo, realizando una llamada recursiva.
- Ambos hijos son parte del rango de la consulta así que debo ir recursivamente a ambos nodos, realizando dos llamadas recursivas.

Lema 1. *La máxima cantidad de nodos que puede visitar la consulta en un nivel del Segment Tree es* 4.

Prueba. Probaremos el lema por inducción en el número de niveles n:

- Caso base: Sabemos que para $n = 1, n = 2$ y $n = 3$, existen como máximo 4 nodos en el nivel, de modo que a lo más podemos visitar 4 nodos en esos niveles.
- Hipótesis inductiva: : Supongamos que en el n-ésimo nivel se han visitado como máximo 4 nodos.
- Tesis inductiva: Debemos probar que en nivel $(n+1)$ podemos visitar como máximo 4 nodos. Sabemos que los nodos que se visitarán en el nivel $(n + 1)$ han sido llamados por al menos uno de los nodos visitados en el nivel anterior n. Denotemos p a la cantidad de nodos visitados en el n-ésimo nivel, donde $1 \leq p \leq 4$. Existen dos casos:

1) Caso $p \leq 2$: Sabemos que un nodo tiene dos hijos; por lo tanto, en el nivel $(n + 1)$ se pueden llamar recursivamente a máximo $2p$ nodos; es decir, a lo más $2 \cdot 2 = 4$ nodos.
2) Caso $p \geq 3$: En este caso, el n-ésimo nivel tiene 2 nodos extremos y por lo menos un nodo intermedio, cada uno de ellos tiene la posibilidad de ser retornado en caso estar completamente contenido o llamar recursivamente a su(s) hijo(s). Supongamos que algún nodo intermedio no es retornado, esto implica que tiene que llamar recursivamente a algún hijo; es decir, existe un subsegmento no vacío del

rango (representado por el nodo) que no pertenece al rango inicial consultado. Debido a que el rango consultado no tiene vacíos intermedios, entonces alguno de los nodos extremos no tiene intersección con el rango y no debió ser visitado en primer lugar, lo cual es una contradicción. Por lo tanto, los nodos intermedios necesariamente se retornan y los únicos nodos con la posibilidad de hacer una llamada recursiva a sus hijos son los extremos. Y dado que solo existen 2 extremos, como máximo se pueden visitar $2 \cdot 2 = 4$ nodos en el nivel $(n + 1)$.

En conclusión, el algoritmo de consulta visita a lo más 4 nodos en cualquier nivel. □

Algorithm 3: Consultar

```
Data: l, r, id, tl, tr, tree
Result: Resultado calculado en el segmento
        [tl ... tr] en tree.
begin
    if l <= tl and tr <= r then
        return tree[id]
    end
    tm ← ⌊(tl+tr)/2⌋
    if r <= tm then
        return Consultar(l, r, 2 * id, tl, tm)
    end
    if tm < 1 then
        return Consultar(l, r, 2 * id + 1, tm + 1, tr)
        b
    end
    return Consultar(l, r, 2 * id, tl, tm) +
      Consultar(l, r, 2 * id + 1, tm + 1, tr)
end
```

Esta función la utilizaríamos de la siguiente manera: $R_{eq} \leftarrow$ Consultar$(l, r, 1, 0, n - 1)$.

Tenemos $\lceil \lg n \rceil + 1$ niveles en el Segment Tree y la consulta visitará a lo más 4 nodos por nivel. Por lo tanto, en el peor de los casos se visitarán $4(\lceil \lg n \rceil + 1)$ nodos. Finalmente, la complejidad de la consulta sería $O(\lg n)$. La implementación de la consulta sería de la siguiente forma:

3. Aplicación

En un laboratorio se tiene un circuito con n resistores en serie con valores iniciales $R_0, R_1, \ldots, R_{n-1}$. Asimismo, se realizarán q acciones que pueden ser de dos tipos:

- 1 pos val: Modificar el valor de la resistencia en la posición pos por val.
- 2 l r: Hallar la resistencia equivalente entre los resistores $R_l, \ldots, R_r$, incluyendo a los extremos.

Para cada consulta del tipo 2 se quiere saber el valor de la resistencia equivalente medida.

3.1. Solución $O(qn)$

La manera más intuitiva de solucionar el problema presentado es mediante un arreglo en el cual guardaremos los valores de las resistencias el cual denotaremos con el nombre res y tendrá n elementos, variable que representa la cantidad de resistencias; esto implica una complejidad en memoria de $O(n)$. Trabajaremos con este arreglo realizando consultas descriras de acuerdo a un $tipo$ dado. Sin embargo, aunque este método sea simple de razonar no es el más eficiente puesto que en el peor de los casos la complejidad en tiempo es $O(qn)$.

Algorithm 4: Solución ineficiente

```
Data: n, q, res.
Result: Respuesta a las consultas tipo 2.
begin
    while q > 0 do
        Leer tipo
        if tipo = 1 then
            Leer pos, val
            res[pos] ← val
        else
            Leer l, r
            R_eq ← 0
            for i ← l to r do
                R_eq ← R_eq + res[i]
            end
            Imprimir R_eq
        end
        q ← q − 1
    end
end
```

3.2. Solución $O(q \lg n)$

Utilizando la estructura de datos Segment Tree y sus métodos previamente detallados se logra obtener una solución más eficiente en tiempo con complejidad $O(q \lg n)$ y con la misma complejidad en espacio de memoria $O(n)$.

La función **Construir** utiliza el arreglo de resistencias res, seguido del valor que representa al segmento inicial y los dos últimos valores representan los extremos del segmento $[0 \ldots n - 1]$.

El uso de la función **Actualizar** y la función **Consultar** se ejecutan en el primer y segundo caso dependiendo del tipo de consulta descrita en el problema.

Algorithm 5: Solución eficiente

```
Data: n, q, res.
Result: Respuesta a las consultas tipo 2.
begin
    st.Construir(res, 1, 0, n − 1)
    while q > 0 do
        Leer tipo
        if tipo = 1 then
            Leer pos, val
            st.Actualizar(pos, val, 1, 0, n − 1)
        else
            Leer l, r
            R_eq ← st.Consultar(l, r, 1, 0, n − 1)
            Imprimir R_eq
        end
        q ← q − 1
    end
end
```

De las secciones 2.2 y 2.3, la función **Actualizar** tendrá complejidad en tiempo de $O(\lg n)$ y la función **Consultar** también tendrá complejidad en tiempo de $O(\lg n)$, por lo que la complejidad en tiempo total es de $O(q \lg n)$, puesto que la solución detallada en este apartado llama a estas funciones dentro de un bucle que se ejecutará q veces.

4. Discusión

Otra estructura de datos llamada Fenwick Tree es capaz de resolver este problema respondiendo las consultas y realizando las actualizaciones en $O(\lg n)$. Asimismo, otro posible candidato a solución es el algoritmo de Mo, el cual nos permite precalcular las consultas y responder todo lo solicitado tras un procesamiento total en $O((n+q)\sqrt{n})$.

La técnica puede ser útil para resolver problemas más generales, no solo en dimensiones superiores sino también para realizar actualizaciones en rango sin verse afectada la complejidad en tiempo.

Referencias

[1] J. L. Bentley y D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," en *IEEE Transactions on Computers*, vol. C-29, no. 7, pp. 571-577, Julio 1980, doi: 10.1109/TC.1980.1675628.

[2] M. Ivanov, *e-maxx::algo*. Dominio público, 2012.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein. *Introduction to Algorithms (3rd ed.)*. Cambridge, Massachussets: The MIT Press, 2009.