Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructura de Datos y Algoritmos

# Tarea 1

15 de octubre de 2021

2º semestre 2021 - Bartolomé Peirano

---

## 1. Component Tree

A grandes rasgos un component tree es la forma de representar una lista de valores (matriz) en forma de arbol. En el caso de esta tarea, se utilizaba una lista de pixeles. El arbol se construye a partir de un nodo raiz, el cual contiene todos los valores de la lista, luego empieza a diverisficarse en nodos más pequeños segun los vecindarios que se forma al aumentar el valor del umbral (cada vecindario es un nodo). Cada nodo hijo guarda todos los valores del arreglo que cumplen con $valor \geq umbral$.

Para poder explicar si es conveniente usar un Component Tree o no, es necesario tener claro su diferencia con el Max Tree. La principal diferencia que existe entre ambos arboles, es que los nodos del Component Tree guarda todos los valores del vecindario tal que $valor \geq umbral$, por tanto $vecindario\_nodo\_hijo_i \subset vecindario\_nodo\_padre$. En cambio el Max Tree, guarda en sus nodos solo los valores del vecindario que son iguales a ese umbral, es decir que, $valor = umbral$.

### Ventajas Component Tree

La principal ventaja que se puede destacar del component tree en comparación al max tree, es que acceder a la información de un vecindario es mucho más directo, ya que esta se encuentra contenida en un solo nodo. En cambio en el Max Tree para acceder al vecindario es más complejo, ya que se tiene que ir bajando nodo a nodo para recolectar la información.

### Desventajas de un Component Tree

La principal desventaja que tiene un Component Tree respecto al Max Tree es que se puede deducir que este utiliza más memoria, ya que en el component tree entre nodos se pueden repetir valores de los vecindarios, en cambio en el max tree no hay valores repetidos sino que cada nodo guarda un solo nivel. Por tanto, se puede conlcuir que al guardar datos repetidos este utiliza más memoria.

Otra desventaja podría ser que si se quiere aplicar un filtro especial a un solo umbral de pixeles el Component Tree es más lento que el máx Tree, ya que en se tendría que revisar en el vecindario que pixeles cumplen con el umbral, en cambio en el Max Tree uno sabe directamente que los pixeles de ese nodo corresponden al nivel del filtro.

### Conclusion

Por lo mencionado antes, creo que para esta tarea es más conveniente utilizar un Max tree, ya que en términos de memoria es más eficiente al igual que para implementar el filtro.

### Para más información se puede revisar:

1. A Comparative Review of Component Tree Computation Algorithms
2. Interactive Segmentation Based on Component-trees

## 2. Complejidad

En primer lugar cabe destacar que el algoritmo que desarrolle para la tarea es recursivo. Por tanto, calcular su complejidad no es tan simple.

Sea $n$ el número de pixeles, $m$ la profundidad del arbol y $u$ el numero de umbrales.

### Construccion del arbol

En la primera parte del código hay un $for$ que va de 0 a $n$, el cual nos permite convertir los valores de del arreglo de numeros pixeles en un arreglo de estructura de pixeles, además de identificar los distintos umbrales presente. Ya que es solo un $for$ esta parte del codigo tiene comlejidad $\mathcal{O}(n)$.

Luego, hay otro $for$ de 0 a $u$ para poder obtener el valor de los umbrales ordenado. Por tanto se tendría una notación $\mathcal{O}(u)$.

En la siguiente parte del código hay otro $for$ de 0 a $n$ el cual se encarga de recorrer cada pixel y para unirlo con sus pixeles vecinos (arriba, abajo, derecha e izquierda). La complejidad de esta parte del codigo es $\mathcal{O}(n)$

Finalmente en el código se llama a la función *recursion* la cual se encarga de hacer llamados recursivos para crear el arbol, desde el nodo raiz hasta las hojas. En esta parte trataré de calcular una estimación de la complejidad del algortimo aunque en realidad no esta muy fácil ni claro, ya que es una recursion dentro de otra recursión.

En primer lugar la función parte recorriendo un $for$ que va de 0 hasta el largo del vecindario del nodo (número de pixeles), en el peor de los casos esto equivale a $n$, luego si el valor del pixel que se esta evaluando es mayor al umbral, entonces se procede a llamar a la función

recursiva drs_pixel la cual nos permite encontrar los vecindario del siguiente nivel del nodo (para posteriormente poder crear el nodo), en el peor de los casos esta recursion será de $4n^2$, ya que se recorre los n pixeles en 4 direccions por cada iteración y que puede durar hasta profundizar en los n pixeles. Luego, una vez encontrado los vecindarios (y terminada dicha recursión interna), se procede a crear el nodo y despues a seguir con la recursion *recursion* para ver y crear los siguientes nodos.Esto puede durar un tiempo de profundiad $m$ por la cantidad promedio $v$ de nodos hijos que tiene cada nodo, por lo que la complejidad de esta parte quedaría como $m * v$

Esto nos da una comlejidad de $\mathcal{O}(n^2) * (\mathcal{O}(4n^2) + \mathcal{O}(m * v))$, lo cual nos deja una complejidad total de $\mathcal{O}(4n^3) + \mathcal{O}(m * v * n)$. En este caso no sabemos cual de los dos terminos es de mayor complejidad, debido a que tienen distitnas variables. Por tanto, la complejidad final de crear el arbol sería: $\mathcal{O}(4n^3) + \mathcal{O}(m * v * n)$.

### Implementación del filtro

En el caso del filtro también se utiliza una función recursiva basada en un dfs, en donde se recorre el arbol de manera recursiva desde el nodo raiz hasta las hojas para poder calcular los costos y modificar las pixeles de la imagen. El costo de modificar los pixeles es de $\mathcal{O}(1)$, en cambio el de calcular el costo esta dado por la recursión. En este caso la profundad del arbol es $m$ y se estima que cada nodo tiene una cantidad promedio $v$ de hijos por lo que la comlejidad de esta parte sería de $\mathcal{O}(m * v)$ pero en cada una de las recursiones se recorre el vecindario que contiene cada nodo, el cual en el peor de los casos tiene un largo $n$. Por tanto, la complejidad del filtro estaría dada por $\mathcal{O}(m * v) * \mathcal{O}(n)$ lo que nos da una comlejidad final de $\mathcal{O}(m * v * n)$

Por tanto la complejidad final del algoritmo estaría dada por Complejidad[Crear arbol] + Complejidad[Aplicar filtro] por lo que la comlejidad final del algoritmo estaría en el orden de $\mathcal{O}(4n^3) + \mathcal{O}(m * v * n) + \mathcal{O}(m * v * n)$ lo que es igual a $\mathcal{O}(4n^3) + 2 * \mathcal{O}(m * v * n)$

## 3. Últimos comentarios

En verdad le dedique mucho tiempo a la tarea pero no pude solucionar el stackoverflow que se formaba en los casos más grandes. Mi hipótesis es que esto ocurría por la cantidad de recursiones que usa mi algoritmo, ya esto implica un gran gasto de memoria según lo que converse con los ayudantes. Creo que una mejor solución pudo haber sido usar más iteraciones en vez de recursiones. Me impresionó harto el tiempo que puede cambiar el programa agregandole o sacandole una iteración como lo es un for dentro de otro for dentro de otro for.