DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Segmentation is my fault

Algoritmos y Estructura de Datos III

Primer Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Amigo, Martín Ignacio	368/17	martinamigo@protonmail.com
de Renteria, Dago Martín	036/17	momasosa1@gmail.com
Festini, Santiago Alberto	311/17	festini.santiago21@gmail.com
Regnier, Melissa	052/17	melissaregnier.98@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo práctico, buscamos utilizar los algoritmos aprendidos de generación de árboles generadores mínimos en una aplicación concreta de un problema útil en la vida real. Para esto, una vez modelado el problema en forma de grafo, aplicaremos una adaptación del algoritmo de Kruskal en conjunto con distintas implementaciones de la estructura de datos Disjoint-Set para así poder comparar sus rendimientos.

1.1. Contexto y motivación

Nuestro objetivo es estudiar el problema de segmentación de imágenes, éste consiste en el proceso de etiquetar todos los píxeles de una imagen de tal manera que aquellos con la misma etiqueta compartan ciertas características entre sí. Su aplicación presenta utilidad en diversas ramas científicas, desde medicina para identificación de tumores por nombrar un ejemplo, hasta reconocimiento de caras en problemas de seguridad o aplicaciones móbiles de entretenimiento; lo que lo convierte en un problema de gran interés.[1]

Para estudiar este problema usaremos como herramienta el algoritmo de segmentación de imágenes propuesto por Pedro F. Felzenszwalb en su paper “Eficient Graph-Based Image Segmentation”[2]. Éste se basa en representar imágenes como grafos a los cuales se les puede aplicar una variante del algoritmo de Kruskal que eficientemente genera diversas componentes no conexas. Los elementos de estas componentes presentan cierta similitud entre ellos por lo que manifiestan los distintos segmentos de la imagen.

El centro de nuestro estudio será analizar la eficiencia de este algoritmo utilizando distintas implementaciones de la estructura de datos disjoint-set, al igual que experimentar con sus resultados y realizar conclusiones sobre cuáles son los grupos de imágenes que más se benefician por este método, y para cuáles este método no sirve.

1.2. Abstracción y simplificación

Para centralizar nuestro estudio en el problema principal, segmentación de imágenes, y no agregar dificultades innecesarias, decidimos considerar unicamente imágenes en escala de grises y con al menos 2 píxeles de ancho y 2 píxeles de alto.

Además, para facilitar el entendimiento y poder abstraernos de los detalles particulares, definiremos las siguientes notaciones:

- h : Altura de la imagen (h proviene de height: altura en inglés), h $\geq$ 2.
- w : Anchura de la imagen (w proviene de width: anchura en inglés), w $\geq$ 2.
- n : Número de píxeles de la imagen y cantidad de vértices del grafo, dado por h*w.
- m : Cantidad de aristas del grafo.
- k : Parámetro constante usado para computar la función τ definida en la sección de desarrollo. k toma un rol importante en la política de unión de componentes, el predicado D que definiremos luego.

1.3. Ejemplificación

Para poder entender mejor de qué se trata la segmentación, veremos un ejemplo. Para ello, utilizaremos la imagen de la izquierda que es de los creadores de este trabajo práctico cuando sus vidas no dependían de una sucesión infinta de $deadlines$. Observemos las diferentes componentes que se pueden observar en la imagen: las personas, el lago, la tierra, la montaña y las nubes. Además vemos que éstas en la imagen segmentada si bien pueden estar compuestas por más de un color distinto no son unificados con otras componentes observables distintas (notar que el verde de las caras no es el mismo verde que el del lago).

Imagen Original

Imagen Segmentada

2. Desarrollo

2.1. Algoritmo

Para aquellos no familiarizados con el trabajo de Felzenszwalb y que no planean leer su paper[2]; y con el fin además de ser autocontenidos pasamos a explicar su algoritmo de forma breve, concisa y sin lujo de detalle. Ciertamente, sin embargo, recomendamos su lectura para una mayor comprensión del problema al igual que la presentación de un segundo algoritmo que no cubriremos.

Para resolver el problema de segmentación de imágenes, Felzenszwalb plantea modelar la imagen con un grafo en el cual cada vértice representa a un píxel y las aristas, que pesan lo que el módulo de la diferencia de color de los vértices involucrados, unen a cada vértice con sus 8 vecinos. Una vez modelada la imagen, el objetivo es formar subconjuntos de vértices que tengan sentido juntos. Para lograr esto se usa el algoritmo de Kruskal, conocido por construir árboles generadores mínimos mediante la unión de componentes disconexas, con una modificación que le permite no agregar la arista y dejar separadas las componentes si estas no tienen verdadera relación entre ellas. Esta modificación toma forma como el predicado D.

D es quién decide si dos componentes disjuntas C_1 y C_2 se relacionan entre ellas o no identificando si hay evidencia de un límite entre las mismas:

Si el menor de los pesos de las aristas que unen C_1 con C_2 (Dif) es mayor a la diferencia interna mínima (MInt) de ambas componentes, entonces hay evidencia suficiente para no unir ambas componentes.

$$\mathrm{D}(C_1, C_2) = \begin{cases} \text{Verdadero} & \text{si Dif}(C_1, C_2) > \text{MInt}(C_1, C_2) \\ \text{Falso} & \text{de otra manera} \end{cases}$$

Donde la diferencia interna mínima (MInt) esta dada por el mínimo valor entre la suma de la diferencia interna (Int) de cada componente con su función umbral τ.

$$\text{MInt}(C_1, C_2) = \min(\text{Int}(C_1) + \tau\ (C_1), \text{Int}(C_2) + \tau\ (C_2)).$$

La diferencia interna (Int) de una componente se define como la arista de mayor peso existente en esa componente.

La función umbral τ se define como k dividido la cantidad de elementos de la componente. Ésta sirve para permitir agregar los primeros elementos y luego para definir un sistema de unión mas relajado (cuando k es grande) o más estricto (cuando k es chico).

Dicho esto, el algoritmo es el siguiente:

$AlgoritmodeSegmentación$ (grafo G = (V,E) con n vértices y m aristas)
Ordenar E en $\pi = (o_1, \ldots, o_m)$, en orden creciente. O(m)[a]
Empezar con una segmentación S_0, donde cada vértice v_i está en su propia componente. O(create(n))
Para q = 1, . . . , m. O(m)
Construir S_q dado S_{q1} de la siguiente manera.
Sean v_i y v_j los vértices conectados por la $q - ésima$ arista, o_q, en π. O(1)
Buscar en que componentes están v_i y v_j. O(find)
Si v_i y v_j están en componentes disjuntas de S_{q1} y el predicado D(C_{v_i}, C_{v_j}) permite la unión:
Entonces se unen[b]. (Notar que o_q es la arista de menor peso que une ambas componentes) O(unite)
Si no, no pasa nada.
Devolver S_m.

[a]Ordenar las aristas cuesta unicamente O(m) porque podemos usar counting sort aprovechando el hecho de que su peso esta acotado por 255.

[b]Hacemos unite de los representantes de las componentes calculados previamente con los finds al fijarnos si estaban o no en componentes distintas, esto evita volver a buscar los representates

Podemos expresar entonces la complejidad del algoritmo en función a la complejidad de los métodos create, find y unite dependiente de cada implementación de union-find.

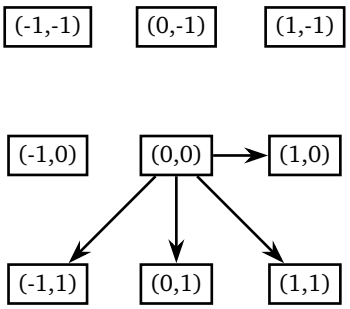
Siendo esta O(m + create(n) + m*(find + unite))

2.2. Modelización

Para poder usar el algoritmo recién descripto, tenemos que modelar los parámetros de entrada en un grafo adecuado que nos permita resolver nuestro problema.

Nuestros parámetros de entrada consisten en los enteros w y h que determinan el ancho y el alto de la imagen, seguidos por h líneas con w números enteros entre 0 y 255, cada uno de estos números representa un píxel de la imagen en escala de grises.

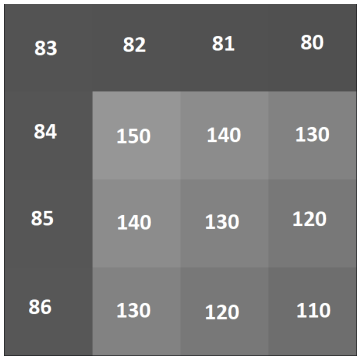
Una vez procesada la imagen y almacenada en una matriz de manera tradicional, recorremos cada elemento de la matriz (cada píxel) y agregamos las aristas válidas a sus vecinos como se indica en el gráfico de la derecha. Donde el (0,0) representa a la fuente y el resto referencia a pocisiones relativas al (0,0) dentro de la matriz.
Por válidas nos referimos a las aristas cuyo destino esta bien definido, ó en rango dentro de la matriz.
Definiendo las aristas de esta manera evitamos definir 2 veces la misma arista ya que nuestro modelo es un grafo no dirigido.
Cada arista conlleva un peso asociado que representa al módulo de la diferencia entre los vértices involucrados.

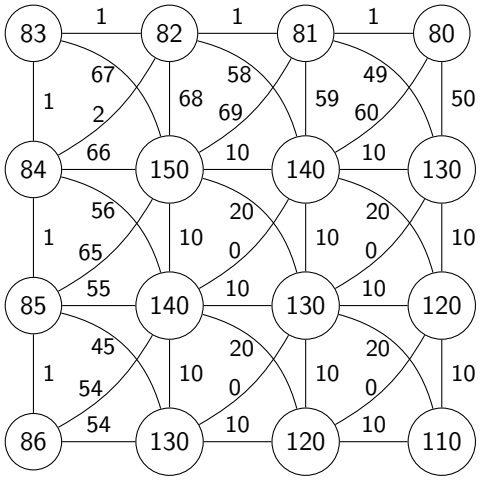


Podemos ver que la complejidad de modelar el problema con un grafo de la manera propuesta es O(n) = O($h * w$). Ésto es porque recorremos una vez todos los píxeles(vértices) O(n) y agregamos a lo sumo 4 aristas por cada uno O(1).

2.3. Ejemplificación

Para facilitar la comprensión del modelado vemos a continuación un ejemplo sencillo de una imagen 4x4 (h = 4 y w = 4). Los números dentro de cada píxel en la imagen representan su valor en escala de grises.





2.4. Estructuras de datos: Disjoint-Set

Disjoint-Set es el tipo de datos que represemta un grupo de conjuntos disjuntos, donde cada conjunto representa una partición $V_1, \ldots, V_k$ del conjunto U = 1, . . . , n y tal que $V_1 \cup \ldots \cup V_k$ = U $\wedge \forall$ i j / 1 $\leq$ i < j $\leq$ k : $V_i \cap V_j = \emptyset$. Éste tipo de datos se basa en 3 operaciones:

• $Create(n)$: Crea la partición inicial $V_1, \ldots, V_n$ donde $V_i = \{i\}$. [1]

• $Find(A)$: Determina a cuál conjunto pertenece el elemento A usando un único identificador por componente, ésto permite determinar si 2 elementos están o no en el mismo conjunto.

• $Unite(A, B)$: Une todo el conjunto al que pertenece A con todo el conjunto al que pertenece B, dando como resultado un nuevo conjunto basado en los elementos tanto de A como de B.

Disjoint-Set se representa con la estructura de datos union-find,

A continuación implementaremos diversas versiones de union-find usadas para representar al tipo Disjoint-Set, todas comparten la misma estructura base de dos diccionarios.

• $pertenencias$[2] : Diccionario cuyas claves son elementos y cuyos significados el identificador de la componente en el caso de array o el padre o si mismo en los casos de trees.

• $tamaños$[3] : Diccionario cuyas claves son componentes y cuyos significados la cantidad de elementos que contiene tal componente.

Es importante notar que aunque cambien las estructuras, la acción de correr el mismo algoritmo sobre ellas resultará en el mismo resultado ya que todas satisfacen al mismo tipo de datos.

2.4.1. Array union-find

La representación de union-find con arrays usa dos vectores como estructuras de datos para pertenencias y tamaños, en donde la posición i, 0 $\leq$ i < n : denota a la componente a la que pertenece el elemento i y a la cantidad de elementos de la componente i respectivamente.

• $create(n)$: Para todo elemento i, 0 $\leq$ i < n : definimos pertenencias[i] = i y tamaños[i] = 1, porque originalmente cada elemento está en su propia componente y está solo. Éste procedimiento conlleva una complejidad O(n) ya que agregar elementos a un vector lleva tiempo O(1) amortizado y por cada elemento (n total) agregamos 2 elementos.

• $find(i)$: Obtener la componente a la que pertenece i es simplemente indexar pertenecientes en i. Complejidad O(1) ya que indexar en un vector es O(1)

• $unite(i, j)$: Para unir las componentes i y j debemos iterar sobre todo pertenencias actualizando los elementos pertenecientes a j por i, o viceversa. Notemos que da lo mismo si decidimos unir ambas componentes en i o en j ya que de todas formas tenemos que recorrer todos los elementos definidos en pertenencias por lo que la complejidad de este método es O(n)

Reemplazando estas complejidades en las calculadas para el algoritmo genérico nos queda

[1] Existen varias definiciones del método Create, como en nuestro caso la imagen no varía, decidimos optar por la versión que inicializa todas las componentes al principio y no permite añadir nuevas posteriormente.

[2] En el código lo nombramos _name

[3] En el código lo nombramos _size

O(m + n + m*(1 + n)) = O(m + n + m*n).

2.4.2. Tree union-find

La representación de union-find con tree usa dos vectores como estructuras de datos para pertenencias y tamaños, en donde la posición i, $0 \leq i < n$: denota al padre de i en el árbol de la componente ó en caso de i ser igual a pertenencias[i] entonces i es la raíz de la componente y por ende el elemento representativo; y a la cantidad de elementos de la componente i respectivamente.

• $create(n)$: Para todo elemento i, $0 \leq i < n$: definimos pertenencias[i] = i y tamaños[i] = 1, porque originalmente cada elemento está en su propia componente y está solo. Éste procedimiento conlleva una complejidad O(n) ya que agregar elementos a un vector lleva tiempo O(1) amortizado y por cada elemento (n total) agregamos 2 elementos.

• $find(i)$: Para obtener la componente a la que pertenece i debemos iterar sobre sus padres hasta llegar a la raíz de la componente, es decir, recorrer el camino i = x(1), . . . , x(k) = r, donde r es la raíz de la componente que contiene a i, lo que daría complejidad de O(k). En peor caso, un árbol degenerado cuya hoja es i por ejemplo, este método podría llegar a requerir recorrer todo el árbol. Su complejidad es entonces O(n).

• $unite(i, j)$: Como siempre que utilizamos al método unite lo hacemos con los representantes de las componentes que queremos unir, lo único que debe hacerse es que el representante de la componente a ser integrada defina como su padre al otro representante y actualizar la cantidad de elementos de las componentes, ambas operaciones toman tiempo constante. Por ende, la complejidad de unite es O(1).

Reemplazando estas complejidades en las calculadas para el algoritmo genérico nos queda
O(m + n + m*(n + 1)) = O(m + n + m*n).

2.4.3. Tree path-relinking union-find

La representación de union-find con path-relinking usa dos vectores como estructuras de datos para pertenencias y tamaños, en donde la posición i, $0 \leq i < n$: denota al padre de i en el árbol de la componente ó en caso de i ser igual a pertenencias[i] entonces i es la raíz de la componente y por ende el elemento representativo; y a la cantidad de elementos de la componente i respectivamente. Se caracteriza por actualizar constantemente pertenencias en las llamadas al método find para que sus llamadas subsecuentes sean más sencillas.

• $create(n)$: Para todo elemento i, $0 \leq i < n$: definimos pertenencias[i] = i y tamaños[i] = 1, porque originalmente cada elemento está en su propia componente y está solo. Éste procedimiento conlleva una complejidad O(n) ya que agregar elementos a un vector lleva tiempo O(1) amortizado y por cada elemento (n total) agregamos 2 elementos.

• $find(i)$: Find de path-relinking funciona igual que el de tree (véase arriba) con una modificación, en la cual a todo x(i) $\in$ camino x(1),...,x(k) se define su padre como la raíz de la componente. Ésto simplifica y reduce el costo de finds subsecuentes logrando así una complejidad amortizada de O(1).

• $unite(i, j)$: Como siempre que utilizamos al método unite lo hacemos con los representantes de las componentes que queremos unir, lo único que debe hacerse es que el representante de la componente a ser integrada defina como su padre al otro representante y actualizar la cantidad de elementos de las componentes, ambas operaciones toman tiempo constante. Por ende, la complejidad de unite es O(1).

Reemplazando estas complejidades en las calculadas para el algoritmo genérico nos queda
O(m + n + m*(1 + 1)) = O(m + n + m).

3. Experimentación

En esta sección, procederemos a experimentar con el algoritmo descripto en la sección anterior con sus distintas implementaciones de $union - find$, para lograr responder las preguntas que nos surgen a partir de las mismas. En un primer lugar, nos parece pertinente corroborar las cotas teóricas del peor caso calculadas en la sección de desarrollo para las tres posibles implementaciones. Por otra parte, queremos llevar a cabo una evaluación comparativa entre ellas. Además, realizamos un análisis cualitativo del

resultado de la segmentación de ciertas imágenes para distintos valores de k con el fin de tratar de estimar qué valor de k resulta apropiado para diversos tipos de imágenes.

3.1. Análisis de complejidad para el peor caso

En lo relativo al primer objetivo planteado, para poder corroborar la cota teórica del peor caso, debemos analizar en un primer lugar cuál es este peor caso para cada implementación de la estructura $union - find$. Recordemos que las complejidades de cada una de sus operaciones son las siguientes cuando la cantidad de elementos es igual a $h * w$:

Operación	Implementación sobre $array$	Implementación sobre $tree$	Implementación sobre $tree$ con $path\ relinking$
Create	$O(h * w)$	$O(h * w)$	$O(h * w)$
Find	$O(1)$	$O(h * w)$	$O(1)\ amortizado$
Unite desde el representante	$O(h * w)$	$O(1)$	$O(1)$

Vale aclarar que sólo consideramos el costo de $unite$ realizado sobre el representante de la componente porque sólo se llama bajo esas condiciones.

La complejidad de nuestro algoritmo en el peor caso es de $O(h * w * (c_{find} + c_{unite}))$. Si bien $find$ se realiza en todas las $O(h * w)$ iteraciones, $unite$ sólo se realiza para una arista ab cuando se cumple la guarda necesaria para unir dos componentes. Tanto para $tree$ como para $tree$ con $path\ relinking$, el costo del $unite$ desde el representante es de $O(1)$. Es por esto que podemos decir que su peor caso no depende del $unite$. Esto no se cumple para la implementación sobre $array$, donde la operación de $unite$ es la más costosa ($O(h * w)$). Por dicha razón, buscaremos maximizar la cantidad de veces que se cumple la guarda que precede al $unite$ para llegar al peor caso de $array$. Recordemos la guarda:

$$find(a) \neq find(b) \quad \wedge \quad weight(ab) < MInt(find(a), find(b))$$

Como vemos, para que la guarda se cumpla la mayor cantidad de veces se deben cumplir ambas partes del and. En cuanto a la primera parte, la mayor cantidad de veces que puede cumplirse es la mayor cantidad de veces que los $find$ de dos distintos vértices me den distinto. Esto es la mayor cantidad de $unite$ que se puedan llevar a cabo antes de tener todos los elementos en una misma componente, ya que una vez todos los elementos estén en la misma componente los $find$ siempre me darán igual y por tanto no se realizará ningún otro $unite$. Si pensamos a cada componente como la componente conexa en un grafo al que pertenecen n vértices donde las aristas representan cada $unite$ que permite que dos componentes conexas sean una, podemos ver que la máxima cantidad de $unite$ a realizar antes de que todos los vértices pertenezcan a la misma componente es la mínima cantidad de aristas a definir antes de que el grafo sea conexo. Es decir, son $n - 1$ aristas y por tanto $O(h * w)\ unite$.

En cuanto a la segunda condición, como $weight$ está acotado por 255 si $MInt$ tiene un valor mayor a 255 entonces la condición es trivialmente válida para cualquier arista. Para lograr que esto suceda, recordemos que $MInt$ se define como:

$$MInt(C_1, C_2) = min(maxEdge(C_1) + \frac{k}{size(C_1)}, maxEdge(C_2) + \frac{k}{size(C_2)})$$

Como $maxEdge$ varía, basta ver que $\frac{k}{size(C)}$ sea mayor a 255. Por tanto, sabiendo que $size(C)$ es a lo sumo tan grande como todos los vértices ($h * w$), con un valor de k de $k = h * w * 256$ siempre es mayor que 255. De esta forma tanto la primer como segunda parte de la guarda se cumplen la mayor cantidad de veces posible, por lo que tomando dicho valor de k encontramos un peor caso para la implementación sobre $array$.

En cuanto al peor caso de la implementación sobre $tree$, ya mencionamos que no depende de la cantidad de veces que se realice el $unite$. Sin embargo, siempre se realiza el $find$ que en su peor caso tiene complejidad de $O(n)$. Veamos si podemos lograr que siempre la complejidad sea la del peor caso. Como

recordaremos, la complejidad del peor caso surge cuando realizo un $find$ de un elemento que se encuentra a distancia $O(n)$ de su representante, lo que le lleva $O(n)$ consultas para lograrlo. Si quisiéramos que esto siempre sucediese, todos los elementos deberían encontrarse a $O(n)$ distancia de su representante, lo cual es imposible. Por otra parte, como nuestro algoritmo realiza la operación de $unite$ sobre los representantes, los árboles de cada componente terminan adosándose entre raíces. Si bien podría existir un caso particular en el cual la unión entre componentes se diese de manera tal que el árbol terminase siendo una lista enlazada, este dependería exclusivamente de la manera en que las aristas de los píxeles se definen y el orden en el cual se pasan como parámetros al $unite$. Para que esto suceda, el $unite$ debería tomar siempre como primer parámetro vértices aislados. Este caso no sólo nos parece patológico sino difícil de ver y de generar ya que depende del orden en el cual se definen las aristas, teniendo que encontrarse una relación entre el orden topológico en el cual se definen las aristas y los valores de las mismas. Además, de todas formas $find$ no costaría $\theta(n)$ para todos los vértices. Por otra parte, como nuestra manera de agregar aristas pone primeros a los vértices de menor h y menor w, no existe manera de que se forme una lista enlazada. Sin embargo, sí podemos intentar aumentar el costo del $find$ dentro de lo posible. Es por esto que decidimos considerar el mismo caso que para $array$, ya que si se ejecutan más $unite$s esto resultará en tamaños de componentes mayores y por lo tanto árboles de mayor altura. Es decir, de esta forma el $find$ termina costando más ya que debe recorrer a partir del nivel del nodo en cada llamada, por lo que en promedio el costo aumenta.

En el caso de la implementación sobre $tree$ con $path\ relinking$, como el $find$ cuesta $O(\alpha(n)^4)$ tiempo amortizado cuando se realizan $O(h * w)$ operaciones y el $unite$ también es $O(1)$, no existe un peor caso teórico para esta estructura.

Teniendo en cuenta lo ya mencionado acerca de la existencia de peores casos, para corroborar las complejidades utilizamos el valor de k mencionado para el caso de $array$ e imágenes con valores aleatorios uniformes entre 0 y 255. De esta forma, podemos analizar la validez de las cotas teóricas y a su vez comparar entre sí las distinas implementaciones. Vale notar que como $tree$ y $tree$ con $path\ relinking$ no poseían un peor caso, utilizamos el de $array$ que de todas formas era válído para estos.

El experimento consistió en correr 30 veces los algoritmos para las imágenes generadas y el k correspondiente para poder obtener un promedio y que no influya el manejo de scheduler y memoria en los tiempos resultantes. Se generaron imágenes variando tanto h como w entre los siguientes valores: $[2^5, 52, 102, 152, 202, 302, 402, 502]$. Estos valores fueron escogidos ya que pensamos que nos otorgarían un rango suficiente de tamaños de imágenes en el que se pueda apreciar la relación entre el tiempo de ejecución y las variables de las cual depende la complejidad sin suponer un costo de tiempo y recursos mayor al que disponíamos.

Nuestra hipótesis acerca del resultado es que los tiempos de ejecución cumplirán con la complejidad calculada.

A continuación, presentamos los resultados de la experimentación para cada algoritmo.

[4] $\alpha(n)$: inversa de la función de Ackermann

[5] Comienza en 2 ya que no tiene sentido considerar imágenes de un sólo píxel

3.1.1. Array

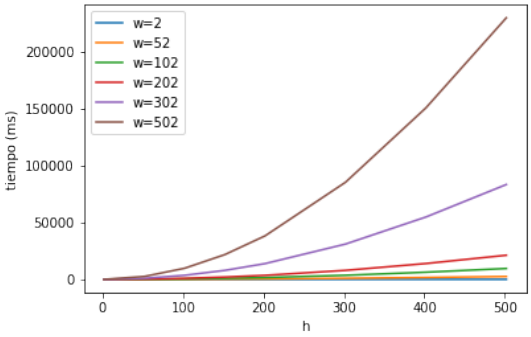


Imagen 1: Tiempo en función de h para distintos valores de w para el caso de implementación sobre array.

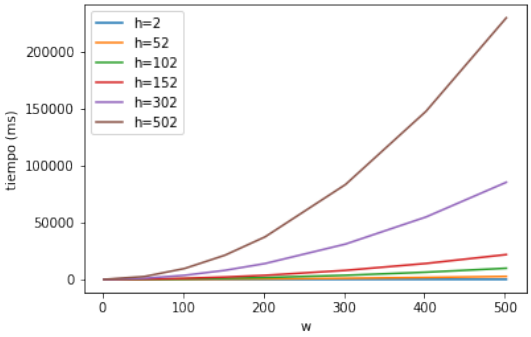


Imagen 2: Tiempo en función de w para distintos valores de h para el caso de implementación sobre array.

Mientras que la imagen 1 grafica cómo varía el tiempo de ejecución en función de h cuando w se encuentra fijo, la imagen 2 grafica la relación inversa. En ambos gráficos es posible apreciar que cuando la segunda variable se deja fija, el tiempo de ejecución aumenta a medida que la primera lo hace. Además, para mayores valores de la segunda variable también observamos mayores valores de tiempos de ejecución. Es más, las curvas esbozadas parecen tener una forma de parábola, lo cual indicaría una relación cuadrática del tiempo con respecto a cada variable por separado. Esto se condice con la cota teórica planteada de $O(h^2 * w^2)$.

Para poder verificar que la intuición adquirida luego del análisis es correcta, analizaremos la correlación existente entre los tiempos obtenidos y la cota teórica calculada. El gráfico resultante puede verse en la imagen 3, donde podemos ver que los puntos parecen alinearse casi perfectamente en una recta. Además, su coeficiente de correlación de Pearson es 0.9999231445299285 lo cual apunta a una fuerte correlación y confirma nuestra hipótesis.

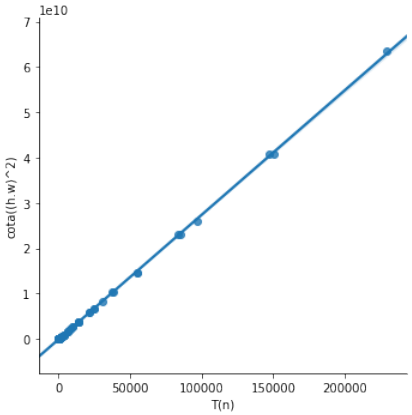


Imagen 3: Gráfico de correlación entre el tiempo de ejecución y la cota $(h * w)^2$ para el caso de $array$.

3.1.2. Tree

A continuación presentamos los resultados obtenidos.

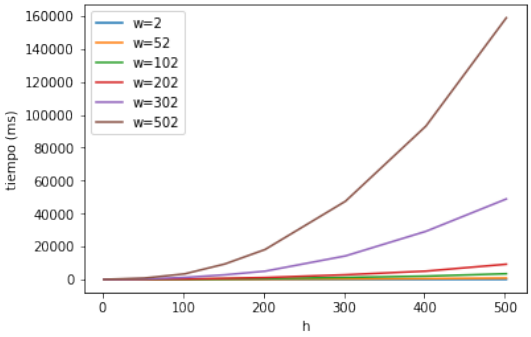


Imagen 4: Tiempo en función de h para distintos valores de w para el caso de implementación sobre tree.

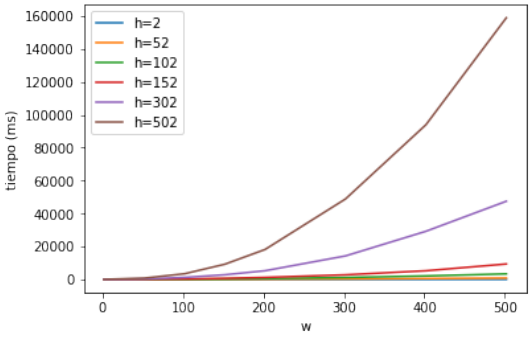


Imagen 5: Tiempo en función de w para distintos valores de h para el caso de implementación sobre tree.

Tanto en la imagen 4 como la imagen 5 vemos cómo, cuando una de las variables es fijada, el tiempo de ejecución va aumentando a medida que la otra variable aumenta, indicando la existencia de una relación entre el tiempo y cada una de las variables. Además, volvemos a notar como en el caso de $array$ que las curvaturas asemejan una relación cuadrática, aunque en este caso pareciera haber ligeras anomalías, lo cual podemos atribuir a que el $find$ en $tree$ no es necesariamente $\theta(n)$, como ya mencionamos.

Para poder analizar con mayor exactitud la cota, graficamos la correlación entre la cota teórica y los tiempos de ejecución para los distintos valores de h y w. Vemos en la imagen 6 cómo, si bien los puntos se encuentran mayormente alineados, existen mayores disidencias con respecto a la recta principal, hecho que no sucedía en $array$. Esto guarda relación con lo antes mencionado: no es en todas las iteraciones que el $find$ cuesta $O(n)$. Sin embargo, el coeficiente de correlación con valor aproximado de 0.996 nos indica que se trata aún así de una cota apropiada, si bien es menor que para la anterior estructura.

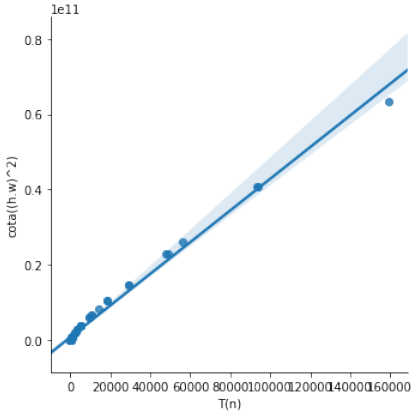


Imagen 6: Gráfico de correlación entre el tiempo de ejecución y la cota de $(h * w)^2$ para el caso de implementación sobre tree.
$Coeficiente\ de\ Pearson = 0{,}9957143735435156$

3.1.3. Tree con path relinking

Los resultados obtenidos se graficaron de la misma manera que para las otras dos implementaciones. Como podemos ver, en este caso si bien se mantiene que para un w o h fijo observamos que el tiempo de ejecución aumenta cuando lo hace h o w, las curvaturas ya no son similares a la curva de una parábola. Se mantiene también que a mayor h o w mayores tiempos de ejecución pueden observarse.

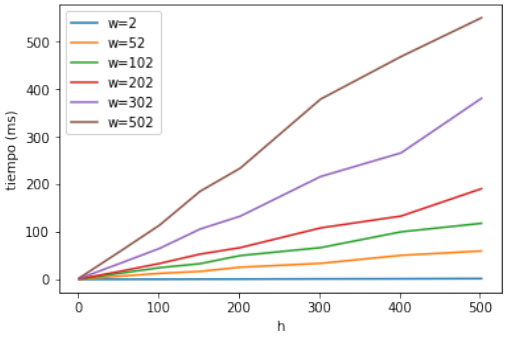


Imagen 7: Tiempo en función de h para distintos valores de w para el caso de implementación sobre tree con path relinking.

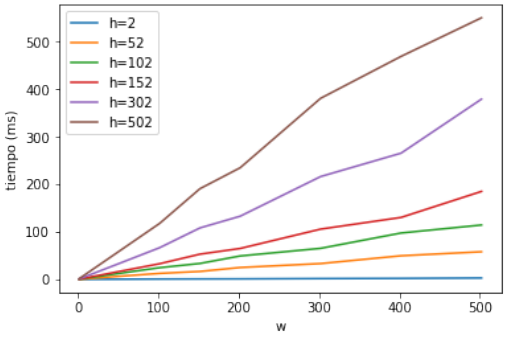


Imagen 8: Tiempo en función de w para distintos valores de h para el caso de implementación sobre tree con path relinking.

En cuanto al comportamiento de las curvas, este parece ser lineal si bien existen ciertos altibajos. Estos se observan simétricamente en ambos gráficos lo cual nos lleva a pensar que se relaciona con el tamaño de la imagen (ya que si es simétrico, $h * w$ tiene un mismo valor). Debido a esto, consideramos que la razón de estas anomalías puede yacer en una cuestión de utilización de memoria, lo cual explicaría la relación con el tamaño de imagen. Relacionado con esto, revisamos el código lo que nos llevó a observar en el counting sort que si bien el vector principal tiene tamaño fijo de 256, en cada posición del mismo se define un vector que varía de tamaño con respecto a la cantidad de aristas que existan del valor de la posición. Como nuestro experimento utiliza valores uniformes de píxeles, la distribución de las aristas dentro del vector también lo será, por lo cual tendrá tamaño promedio de $\frac{8*h*w}{256}$. Para comprobar nuestra incipiente hipótesis, realizamos un experimento en donde variamos h y w con valores mayores[6].

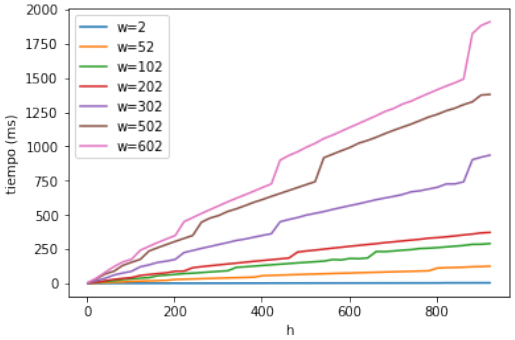


Imagen 9: Tiempo en función de h para distintos valores de w para el caso de implementación sobre tree con path relinking.

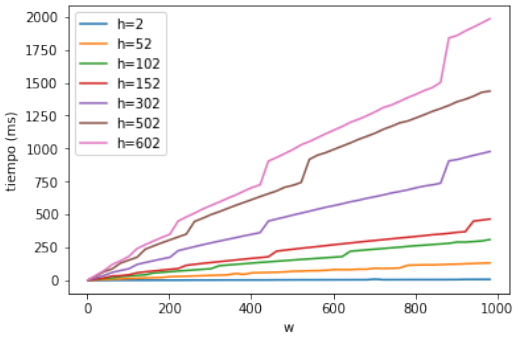


Imagen 10: Tiempo en función de w para distintos valores de h para el caso de implementación sobre tree con path relinking.

En las imágenes 9 y 10, podemos ver que los resultados parecen condecirse con lo predicho por nuestra hipótesis, ya que existen pequeños saltos que se realizan de manera periódica, donde la distancia entre dos saltos distintos parece duplicarse a medida que crecen h y w, lo cual se relaciona con el alojamiento en memoria de los vectores, que duplican el tamaño reservado en memoria a medida que se quedan sin espacio.

Este mismo comportamiento no era fácilmente observable en los casos anteriores ya que la complejidad del resto del algoritmo era mayor que $O(h * w)$.

Finalmente, analizaremos la correlación existente entre la cota teórica propuesta y el tiempo de ejecución del algoritmo.

[6]Variamos h y w entre 2 y 1002 con saltos de 20

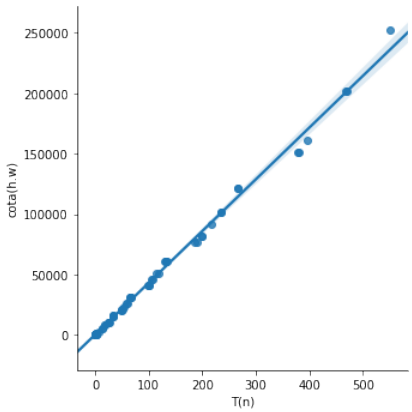


Imagen 11: Gráfico de correlación entre el tiempo de ejecución y la cota de $h * w$ para el caso de implementación sobre tree con path relinking.
$Coeficiente\ de\ Pearson = 0{,}9984631234760327$

Como vemos graficado, parece existir una correlación entre las dos variables, lo cual es corroborado por el coeficiente de Pearson de un valor aproximado de $0{,}998$.

3.1.4. Comparación entre estructuras

Una vez comprobado que las cotas planteadas se ajustan a los resultados obtenidos, nos parece pertinente analizar la eficiencia de las tres implementaciones comparativamente entre ellas. Basándonos en las cotas teóricas, suponemos que la eficiencia de $tree$ con $path\ relinking$ será la mejor estructura. En cuanto a $tree$ y $array$, debido a que la ejecución de $find$ en el primero no es siempre $\theta(n)$, podemos suponer que será mejor que $array$. Veamos entonces en un gráfico comparativo los resultados:

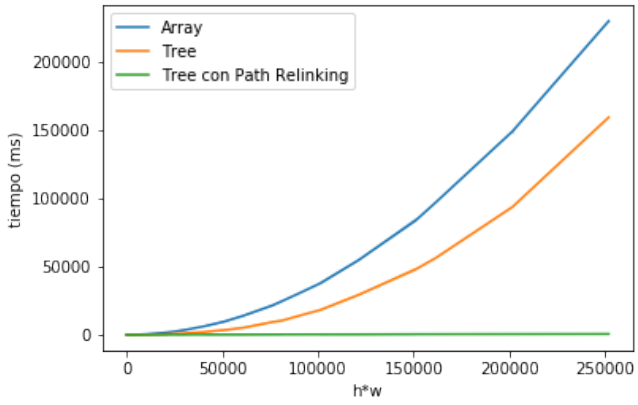


Imagen 12: Gráfico de comparación entre las tres implementaciones con tiempo de ejecución en función del tamaño de la imagen, $h * w$.

Como podemos observar, nuestras hipótesis resultan ciertas: de entre las tres estructuras, la más eficiente resulta ser $tree$ con $path\ relinking$, seguida por $tree$ con $array$ en la retaguardia. Vale notar la amplia diferencia observable entre $tree$ con $path\ relinking$ y las dos estructuras restantes, lo cual se condice con la diferencia entre una cota lineal y otra cuadrática.

3.2. Análisis complejidad con k fijo

En la subsección anterior, analizamos la complejidad para las tres implementaciones presentados para el peor caso. Sin embargo, nos surge la incógnita de la eficiencia de nuestro algoritmos frente a casos

que resulten mejores para alguna implementación con el fin de poder realizar una nueva comparación entre ellas. Es de nuestro interés averiguar si para estos mejores casos la relación entre las estructuras no cambiaría.

Por lo tanto, basándonos en que la experimentación anterior se basó en un peor caso que razonamos común a las tres estructuras, en esta nueva experimentación nos pareció interesante considerar un mejor caso para $array$ y observar si esto afecta el balance de eficiencia entre las implementaciones.

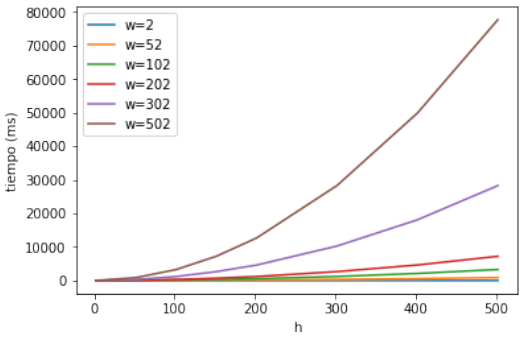
Como vimos que la cantidad de $unite$s realizados afecta al tiempo de ejecución de la implementación de $array$, fijamos un valor de k bajo para disminuir la cantidad de operaciones de $unite$ que se realizan y así buscar mejorar el tiempo de ejecución del algoritmo. Si recordaramos el caso peor utilizado en la experimentación anterior, una mayor cantidad de $unite$s provocaba un peor caso para la implementación sobre $tree$ ya que la altura promedio de los árboles de cada componente aumentaba. Basándonos en esto, podríamos conjeturar que una menor cantidad de operaciones $unite$ resultará en una baja altura promedio del árbol de cada componente, por lo que también mejorará la eficiencia de esta implementación. En cuanto al caso de $tree$ con $path\ relinking$, este caso no debería variar su eficiencia.

La experimentación consistirá, entonces, en llevar a cabo el mismo procedimiento descripto en la subsección anterior con la excepción de que en este caso k se tratará de un valor fijo de 15.

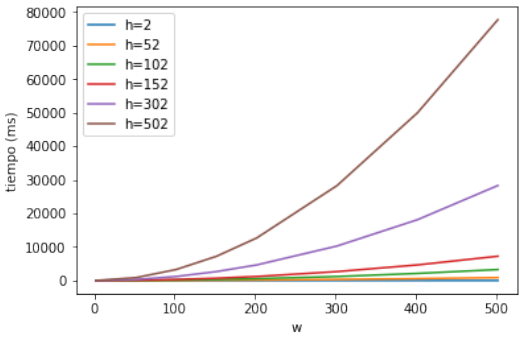
Por otra parte, como el análisis de la complejidad ya lo realizamos en la sección anterior, en este nuevo experimento nos tomaremos la libertad de no analizar con tanto lujo de detalle cada gráfico, deteniéndonos exclusivamente en aquellos análisis que consideremos pertinentes en lo relativo a su diferencia con el anterior caso y por lo tanto sean signos de mención.

3.2.1. Implementación sobre array

Comencemos analizando los resultados para el caso de implementación sobre $array$.



$Imagen$ 13: Tiempo en función de h para distintos valores de w para el caso de implementación sobre $array$ con $k = 15$.



$Imagen$ 14: Tiempo en función de w para distintos valores de h para el caso de implementación sobre $array$ con $k = 15$.

Como podemos observar en ambas imágenes, la relación parece seguir siendo cuadrática por lo que la cota debería seguir siendo la misma. Esto implica que a pesar de hacerse una mucho menor cantidad de $unite$s el costo de los mismos es suficiente como para resultar en la misma cota. Para corroborar que de hecho se trate de la misma cota, realizamos el gráfico de correlación y calculamos el coeficiente de Pearson tanto para la cota original $(h * w)^2$ (imagen 15) como para la cota $h * w$, que debería tener en caso de que $unite$ costase $O(1)$(imagen 16). En las gráficos podemos ver cómo la primer cota sigue resultando adecuada con un coeficiente de Pearson muy elevado, mientras que la segunda cota resulta no tan acertada (valor menor de su coeficiente de Pearson correspondiente).

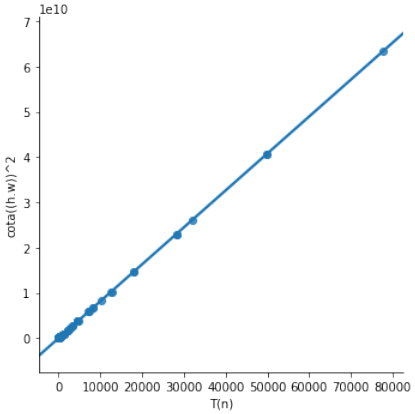


Imagen 15: Gráfico de correlación entre el tiempo de ejecución y la cota $(h * w)^2$ para el caso de implementación sobre $array$.
$Coeficiente\ de\ Pearson =$ 0,9999957831855851

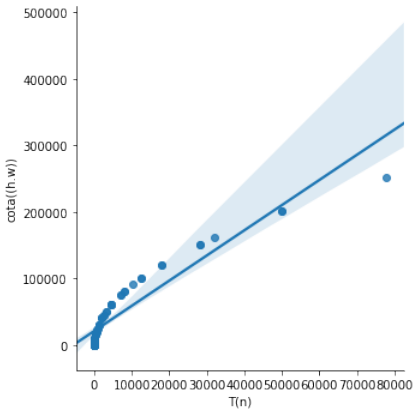


Imagen 16: Gráfico de correlación entre el tiempo de ejecución y la cota $(h * w)$ para el caso de implementación sobre $array$.
$Coeficiente\ de\ Pearson =$ 0,9345478200813492

3.2.2. Implementación sobre tree

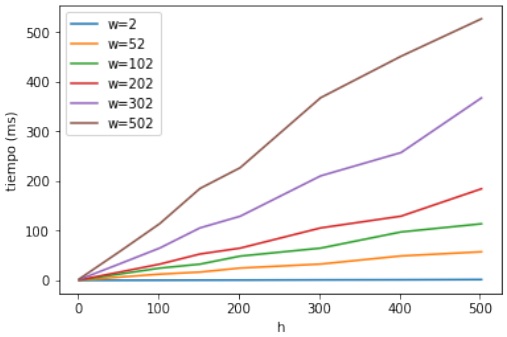


Imagen 17: Tiempo en función de h para distintos valores de w para el caso de implementación sobre $tree$ con $k = 15$.

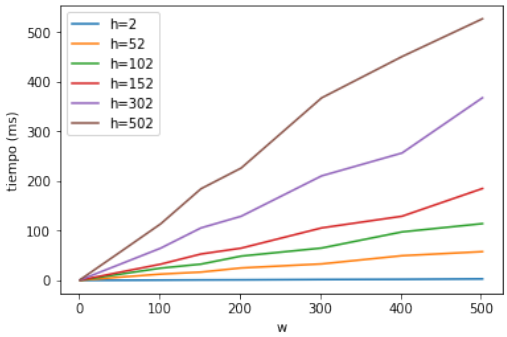


Imagen 18: Tiempo en función de w para distintos valores de h para el caso de implementación sobre $tree$ con $k = 15$.

En las imágenes 17 y 18, vemos cómo la relación entre el tiempo de ejecución y h para w fijo y entre el tiempo de ejecución y w para un h fijo ya no parece ser cuadrática, sino más bien lineal. Recordando nuestra hipótesis inicial, esto se condice con lo predicho ya que en este caso la operación $find$ parecería estar costando una complejidad constante en promedio, por lo que la cota pasaría de ser $\theta((h*w)^2)$ a $\theta(h*w)$. Sin embargo, vemos ciertas anomalías en la recta; si volvemos a los resultados de la experimentación anterior, este gráfico nos recordará al obtenido en la implementación sobre $tree$ con $path\ relinking$. A estas anomalías ya les habíamos encontrado una causa, relacionada con el uso de vectores en nuestra implementación y la manera en que los mismos se asignan en memoria. Debido a que la cota en este experimento pasa a ser de carácter lineal, estas anomalías pasan a ser distinguibles, lo mismo que sucedía en el caso de $tree$ con $path\ relinking$, mientras que no influía en la cota mayor ya que esta era cuadrática.

Para corroborar que la cota lineal es más apropiada en este caso, realizamos, al igual que en el caso anterior para $array$, los gráficos de correlación pertinentes con sus correspondientes coeficientes de Pearson.

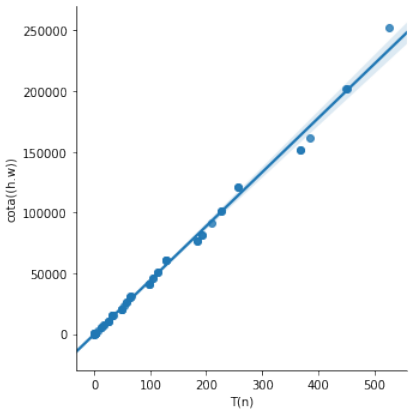


Imagen 19: Gráfico de correlación entre el tiempo de ejecución y la cota $(h * w)^2$ para el caso de implementación sobre $tree$.
$Coeficiente\ de\ Pearson =$ 0,9975083833812181

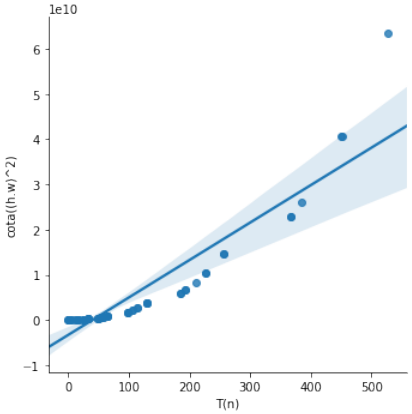


Imagen 20: Gráfico de correlación entre el tiempo de ejecución y la cota $(h * w)^2$ para el caso de implementación sobre $tree$.
$Coeficiente\ de\ Pearson =$ 0,9210351250871439

En las imágenes 19 y 20, vemos cómo la cota de $h * w$ resulta más apropiada que la cota de $(h * w)^2$, resultando en un coeficiente de correlación de Pearson mayor para la primera cota que para la segunda.

3.2.3. Implementación sobre $tree$ con $path\ relinking$

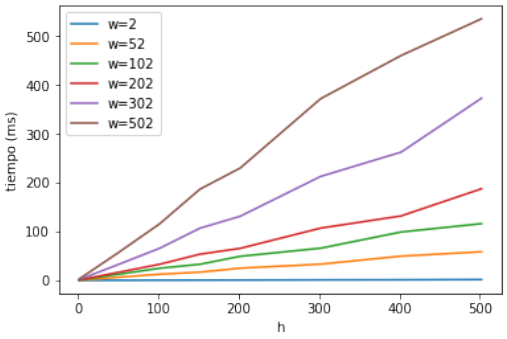


Imagen 21: Tiempo en función de h para distintos valores de w para el caso de implementación sobre $tree$ con $path\ relinking$ con $k = 15$.

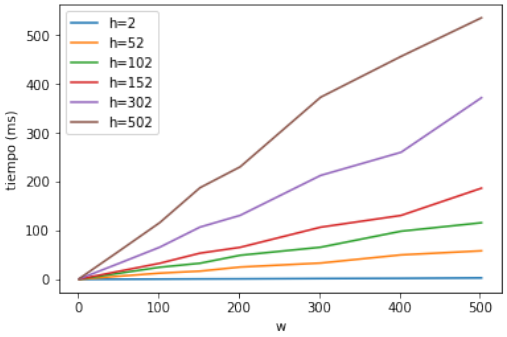


Imagen 22: Tiempo en función de w para distintos valores de h para el caso de implementación sobre $tree$ con $path\ relinking$ con $k = 15$.

Podemos observar a simple vista que los gráficos no difieren de los obtenidos para la misma estructura en la experimentación anterior. También resultan similares a los recién analizados en la estructura de $tree$ en este mismo experimento, lo que nos indica que comparten su carácter lineal así como también las anomalías ya mencionadas (lo cual tiene sentido ya que estas las atribuimos al algoritmo de ordenamiento, que no se relaciona con la implementación de cada estructura).

El gráfico de correlación que observamos en la imagen 20 se condice con lo obtenido anteriormente para esta misma estructura y corrobora que se mantiene la cota ya propuesta de $h * w$.

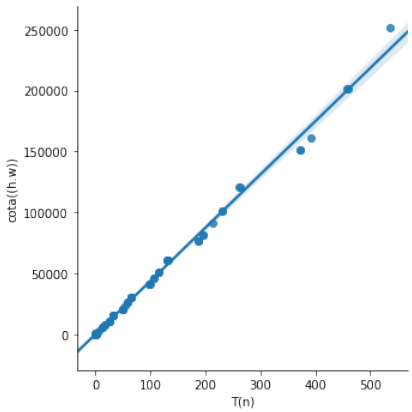


Imagen 23: Gráfico de correlación entre el tiempo de ejecución y la cota $(h * w)^2$ para el caso de implementación sobre $tree$ con $path\ relinking$.
$Coeficiente\ de\ Pearson = 0{,}9984631234760327$

3.2.4. Comparación entre experimentos

Como mencionamos anteriormente, realizamos una comparación para cada estructura entre los resultados obtenidos para un k grande y un k fijo.

Implementación sobre array

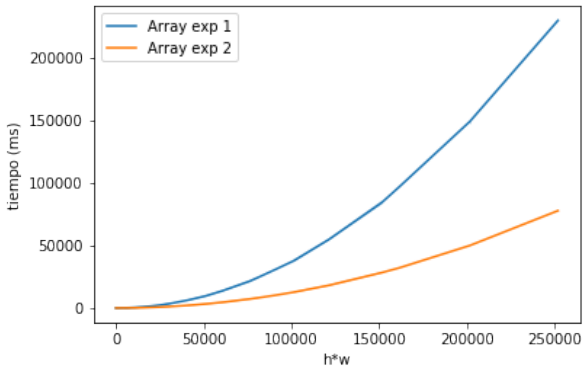


Imagen 24: Gráfico de comparación entre el primer y segundo experimento del tiempo de ejecución en función del tamaño de la imagen $h * w$ para el caso de implementación sobre $array$.

En el gráfico (imagen 24), podemos ver cómo los valores de tiempos de ejecución para el experimento 2 se encuentran bastante por debajo de los obtenidos en el primer experimento. Por lo tanto, podemos confirmar que la cantidad de operaciones $unite$ que se llevan a cabo afecta a la eficiencia del algoritmo y a los tiempos de ejecución involucrados en el mismo. A pesar de tratarse de relaciones cuadráticas, una curva se encuentra marcadamente por encima de la otra.

Implementación sobre tree

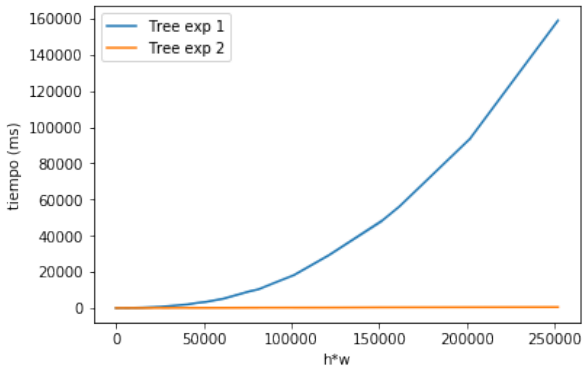


Imagen 25: Gráfico de comparación entre el primer y segundo experimento del tiempo de ejecución en función del tamaño de la imagen $h * w$ para el caso de implementación sobre $tree$.

En esta comparación observamos claramente la diferencia en tiempos de ejecución entre uno y otro experimento. Se puede observar como la experimentación del peor caso resulta cuadrática, provocando que la experimentación de un mejor caso con complejidad lineal se encuentre muy por debajo de ésta. Mientras que la segunda experimentación no llega a los $1000ms$, la primer experimentación muestra valores muy por encima de este valor, llegando a los $16000ms$, dieciséis veces mayor que el valor anterior.

Implementación sobre tree con path relinking

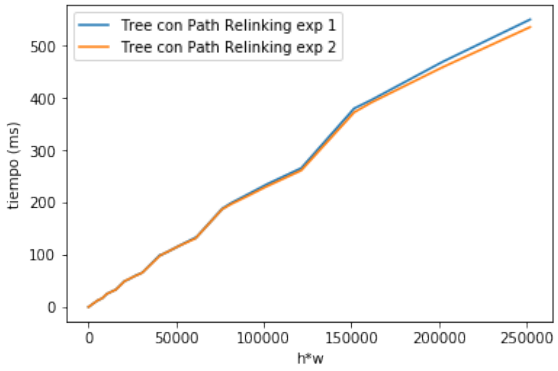


Imagen 26: Gráfico de comparación entre el primer y segundo experimento del tiempo de ejecución en función del tamaño de la imagen $h * w$ para el caso de implementación sobre $tree$ con $path\ relinking$.

En la imagen 26 encontramos el gráfico comparativo de los resultados obtenidos en uno y otro experimento. Destacamos que, como habíamos predicho, ambos casos resultan en tiempos de ejecución muy similares, apenas pudiéndose notar la diferencia entre las dos curvas. Esto se condice con las complejidades de las operaciones de la estructura y cómo no existe un peor caso para la misma. Algo a resaltar es cómo para valor mayores de tamaños de imagen notamos que la curva correspondiente a la experimentación 1 se encuentra ligeramente por encima de la curva correspondiente a la experimentación 2. Esto lo atribuimos a la mayor cantidad de operaciones $unite$ realizadas en el caso de la experimentación 1; si bien son operaciones que cuestan $O(1)$, aún así se trata de una mayor cantidad de operaciones que se realizan en la primera experimentación y no en la segunda, pudiendo resultar esto en la diferencia que notamos entre tiempos de ejecución.

Comparación entre estructuras

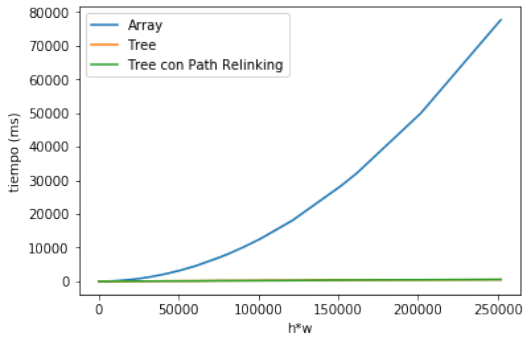


Imagen 27: Gráfico de comparación del tiempo de ejecución en función del tamaño de la imagen $h * w$ entre las implementaciones de las tres estructuras para el segundo experimento.

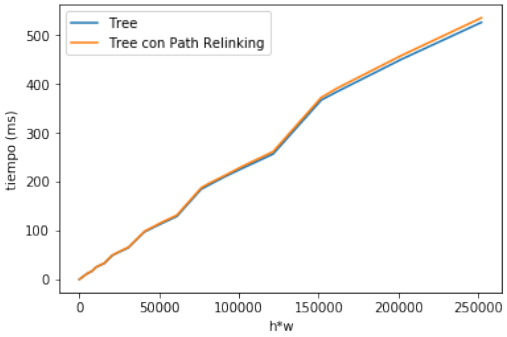


Imagen 28: Gráfico de comparación del tiempo de ejecución en función del tamaño de la imagen $h * w$ entre $tree$ y $tree$ con $path\ relinking$ para la segunda experimentación.

En la imagen 27, observamos cómo claramente la implementación sobre array resulta la menos eficiente, con tiempos de ejecución remarcablemente mayores a los observados en las otras dos estructuras. La naturaleza cuadrática de la complejidad de la implementación utilizando esta estructura en comparación con la naturaleza lineal de las otras dos es la causante de esta brecha. Debido a esta gran diferencia, no nos era posible analizar con claridad la diferencia entre $tree$ y $tree$ con $path\ relinking$, por lo que confeccionamos otro gráfico en la que apareciesen sólo estas dos estructuras con el fin de poder analizarlas. Este gráfico es el que observamos en la imagen 28, donde volvemos a notar que la diferencia entre ellas es casi inexistente. Es remarcable que la curva perteneciente a $tree$ se encuentra ligeramente por debajo de la curva de $tree$ con $path\ relinking$. Esto implica que la complejidad del $find$ en el caso de $tree$ está resultando igual o incluso más eficiente que el caso del $find$ en $tree$ con $path\ relinking$, que si bien recordamos tenía un costo amortizado de $O(\alpha(n))$ siendo α la inversa de la función de Ackermann. Podemos atinar a decir que si bien las complejidades están resultando ser similares, se debe tratar de diferencias entre constantes que están resultando en esta diferencia observable. Incluso podríamos relacionarlo con la dependencia de la cota de $tree$ con $path\ relinking$ de la inversa de la función de Ackermann; si bien esta tiene un crecimiento extremadamente chico, puede estar influyendo mínimamente en los tiempos de ejecución de esta estructura, provocando la diferencia que observamos entre las curvas de las dos estructuras mencionadas.

3.3. Análisis cualitativo

Es de nuestro interés, como mencionamos al principio, poder realizar un análisis cualitativo en cuanto a las imágenes segmentadas obtenidas a partir del algoritmo y establecer una aproximación de valores de k convenientes en relación con características distinguibles en una imagen. Con este propósito, primero definiremos conjuntos de imágenes que compartan ciertas propiedades para poder analizar las segmentaciones resultantes para distintos valores de k y así decidir qué valores son convenientes para cada conjunto.

Conociendo la naturaleza del algoritmo, primero nos pareció interesante considerar como propiedad a la diferencia de matices dentro de cada componente distinguible. Por un lado, consideraremos imágenes tal que sus componentes sean relativamente homogéneas, es decir, que no tengan grandes diferencias de matices. Por otro lado, tomaremos imágenes cuyas componentes sean más heterogéneas, es decir, con una amplitud de matices mayor.

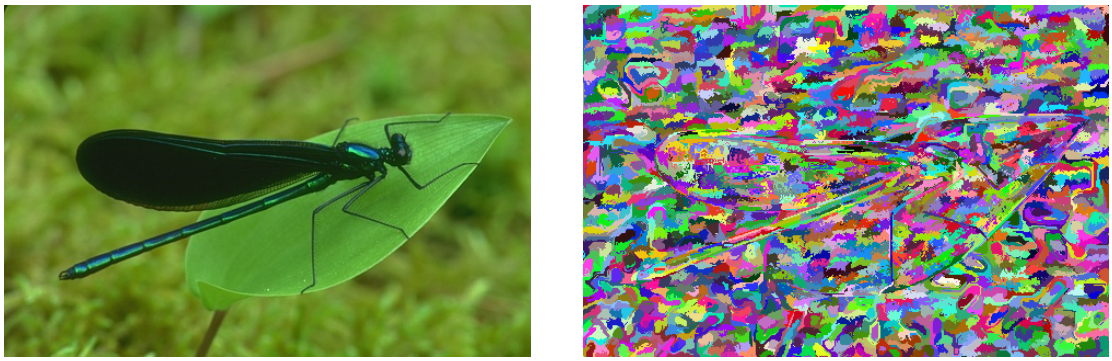
En segundo lugar, otra característica que destacamos es la cantidad de componentes existentes en la imagen. Es por esto que estaremos tratando imágenes de pocas componentes así también como imágenes con muchas componentes. Como menciona el paper[2], a medida que el valor de k aumenta, también lo hará el tamaño de las componentes en las cuales se segmenta la imagen y, por ende, disminuirá la cantidad total de las mismas (ya que el tamaño no varía).

Una vez elegidas nuestras características distintivas, presentaremos imágenes que cumplan las distintas combinaciones de las dos propiedades: tanto imágenes con pocas como muchas componentes, de

pocos o muchos matices.

Para llevar a cabo el experimento, utilizaremos un script de python que tranforma las imagenes en color de formato $.bmp$ a escala de grises utilizando un filtro gaussiano para suavizar ligeramente la imagen con $\sigma = 0{,}5$ (ya que nos pareció un valor intermedio) y luego a formato $.txt$ para utilizarlo como $input$ del programa.

El experimento consistirá en aplicar el algoritmo para dos imágenes dentro de cada grupo,. Utilizaremos los siguientes valores de k: 1000, 10000 y 100000. Estos valores nos resultaron razonables considerando que la mayoría de las imágenes utilizadas tienen un tamaño aproximado de 150000, por lo que con dichos valores de k podemos ver distintos tamaños razonables de componentes, siendo el último valor de k el que supone componentes que abarcan la mayor parte de la imagen. Si bien consideramos utilizar un valor de k de 100, los resultados (como se puede apreciar en la imagen 29) siempre implicaban una segmentación en demasiadas componentes, no siendo de interés este comportamiento para ninguna imagen considerada.



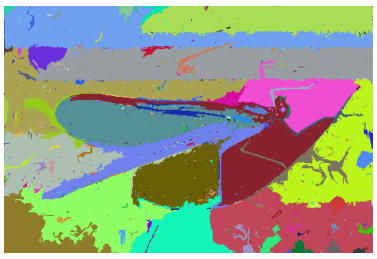
$Imagen$ 29: Comparación entre imagen original y segmentada con $k = 100$.

Como consideración general adaptada, consideramos más correctas a aquellas segmentaciones que no unen aquellos elementos de la imagen que deberían estar en componentes distintas.

3.3.1. Pocas componentes homogéneas

Nuestra hipótesis para este grupo de imágenes es que, debido a la homogeneidad de las componentes, con valores relativamente bajos de k la segmentación obtenida unirá los elementos de la misma componente ya que no habrá tanta diferencia entre píxeles dentro de la misma componente. Sin embargo, como se tratan de pocas componentes un valor de k elevado es necesario para unificar componentes que tienen un tamaño mayor al tratarse de pocas. Por lo tanto, deberíamos utilizar un valor de k intermedio: lo suficientemente alto para unir componentes de gran tamaño (recordemos que k se divide por el tamaño de cada componente) y lo suficientemente bajo para no provocar la unión no deseada de componentes distintas.

$Imagen$ 30: Imagen original



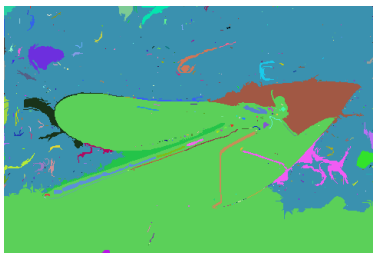


Imagen 31: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Observemos las imágenes 30 y 31. Considerando que una segmentación óptima sería aquellas que separa la libélula, la hoja y el fondo (sólo tres componentes), podemos decir que la segmentación que más se acerca a la deseada es aquella con $k = 10000$. Podemos ver como separa el cuerpo de la libélula de la hoja asi como también es capaz de distinguir su ala. El fondo queda segmentado en una mayor cantidad de componentes quizás que la deseada, pero si observamos la imagen original (imagen 30) vemos que es en el fondo en donde más diferencia de matices podemos distinguir, por lo que esta mayor segmentación tiene sentido. Por otra parte, logra separar las partes de la hoja delimitadas por las patas de la libélula sin separar cada parte por demás. Por el contrario, la imagen con $k = 100000$, unifica tanto a la libélula como la hoja y parte del fondo en una misma componente, característica que no deseamos en una segmentación. A su vez, la imagen con $k = 1000$ presenta demasiadas subdivisiones dentro de cada componente buscada como para poder ser considerada adecuada.

Veamos otro ejemplo de imagen dentro de este mismo grupo, que posee dos componentes pero que la diferencia de matices promedio entre ambas componentes es mayor que en el ejemplo anterior observado.

Imagen 32: Imagen original

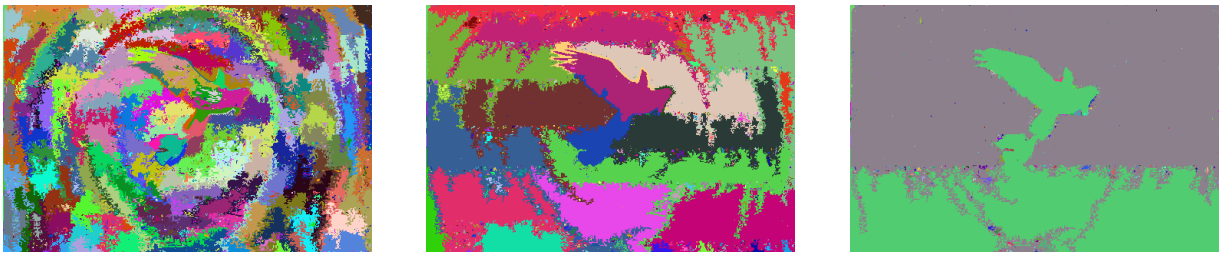


Imagen 33: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

En este caso, veremos cómo el valor de k más adecuado difiere. En esta imagen, nos interesa segmentar al ave del cielo ya que nos parecen las únicas dos componentes destacables.

Podemos ver que para el valor de k menor, hay una cantidad de componentes exageradamente superior a la deseada, por lo que este valor de k puede ser descartado rápidamente. Vemos que para el valor de $k = 10000$, que en el ejemplo anterior era el más apropiado, en este caso sigue dividiendo quizás por demás de lo buscado a las únicas dos componentes existentes. Mayormente, es reprochable la división del cielo en una gran cantidad de componentes, considerando la reducida amplitud de matices que hayamos en el mismo, por lo que podríamos teorizar que debería existir un valor de k satisfactorio que una más al cielo. En el tercer valor de k vemos cómo mejora la cantidad de componentes existentes: el cielo se encuentra más unificado así como también lo está el pájaro. Sin embargo, vemos como el pájaro se encuentra unido con parte del cielo, lo cual no es óptimo. Si observamos el tamaño de las componentes definidas, podemos decir que desearíamos que la componente del cielo sea aún más grande de lo que actualmente es. Por lo tanto, suponemos que si utilizamos un k más grande podríamos lograr que se termine unificando todo el cielo; sin embargo no debe ser lo suficientemente grande como para unificar el pájaro también. Es por esto que vamos a probar con un $k = 200000$ para ver si esto permite definitivamente unificar la componente del cielo y distinguir así el ave, en caso de que este k resulte excesivo y solo se observe una componente probaremos con valores de k intermedios entre 100000 y 200000.

Imagen 34: Segmentación obtenida para $k = 200000$

Con este nuevo k podemos ver que el ave se encuentra perfectamente segmentada, lo cual podría generar en el lector la incógnita de ¿por qué con $k = 100000$ une el cielo con el ave mientras que un k mayor, que debería implicar que se unan más píxeles, el cielo no se une con el ave? Esto lo atribuimos a que como un k más grande se admiten más uniones entre componentes al permitirse un tamaño mayor de las mismas. Con esto en cuenta y recordando la función "$D(C_1, C_2)$"

$$MInt(C_1, C_2) = min(maxEdge(C_1) + \frac{k}{size(C_1)}, maxEdge(C_2) + \frac{k}{size(C_2)})$$

podemos ver que el nuevo vecindario entre el ave y la nueva componente cielo (que ahora se encuentra unificada) no admite una unión debido a la diferencia de matices promedio entre el ave y el cielo.

Es destacable que dentro del mismo grupo debimos considerar distintos valores de k, lo cual nos inclina a pensar que no podemos englobar a todas las imágenes que cumplen esta característica para poder utilizar un mismo valor de k. Sin embargo, podemos encontrar una explicación a lo sucedido. Mientras que en el segundo caso la diferencia de matices promedio entre una componente y la otra es considerablemente grande, no sucede lo mismo en el primer caso: hay una menor diferencia entre los matices del $background$, los matices de la libélula y los matices de la hoja. Si tomamos esto en consideración, tiene sentido que en el primer caso con un valor menor de k logremos una segmentación aceptable mientras que si aumentamos el valor de k esto termine provocando la unión de las componentes, ya que se necesita una menor diferencia para unificarlas debido a la distancia menor entre matices promedio. Al contrario, en el caso del pájaro, hay una mayor distancia entre el matiz del pájaro y el del cielo. Por lo tanto, al utilizar un k mayor somos capaces de lograr componentes de tamaño mayor pero que no unifican las dos componentes.

3.3.2. Pocas componentes heterogénas

Repetimos el procedimiento descripto para este grupo, obteniendo los resultados que se pueden ver en las imágenes 35, 36, 37 y 38. Las componentes distinguibles en la imagen 35 son los dos caballos y el $background$ con pasto. En cuanto a la imagen 37, las componentes a segmentar serían la pirámide, la parte del cielo nublada y la parte del cielo despejada.

Imagen 35: Imagen original

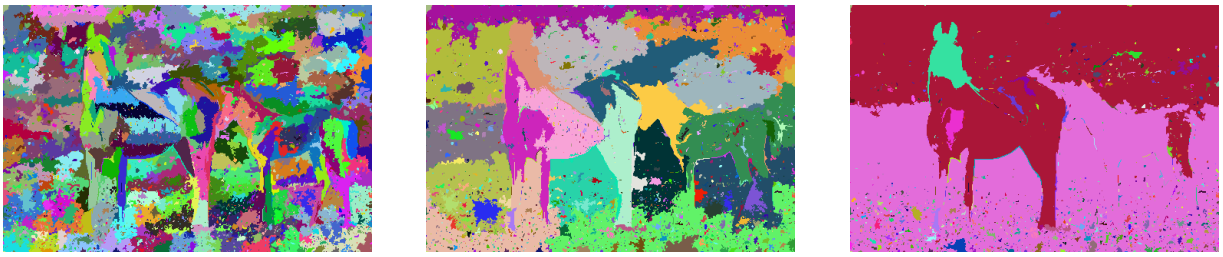


Imagen 36: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Imagen 37: Imagen original

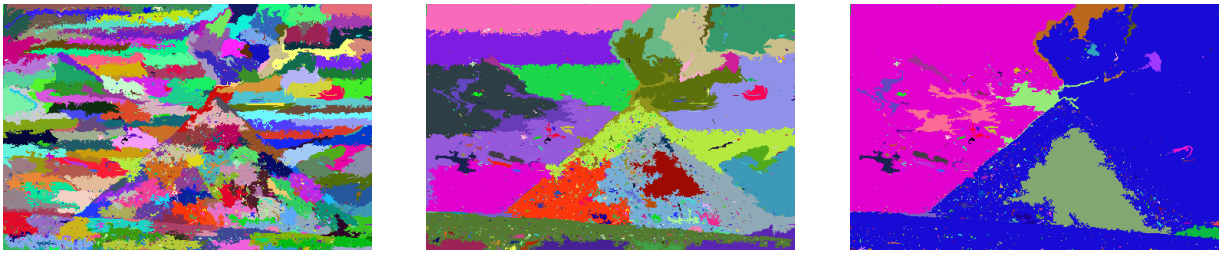


Imagen 38: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Vemos en ambos ejemplos cómo el k más óptimo resulta ser el de $k = 10000$, ya que para el valor mayor termina unificando componentes no deseadas y para el valor menor segmenta por demás las componentes observables. Sin embargo, si bien esto se condice con la hipótesis inicial de que para menor cantidad de componentes necesitaríamos un k grande que permita tamaños grandes de componentes, pareciera contradecirse con la otra hipótesis planteada en lo relativo a la diferencia de matices dentro de la misma componente. Si bien en este caso tenemos una mayor amplitud de matices dentro de la misma componente, un k más alto termina generando la unión insatisfactoria de componentes mientras que habíamos supuesto que la mayor cantidad de matices exigiría un k mayor que permitiese su unión. En un principio, esto nos generó confusión. Pero antes de sucumbir ante la duda existencial de nuestras hipótesis, observamos con más detenimiento ambas imágenes. En el caso del caballo, el algoritmo unifica al potrillo con el $background$ mientras que en el caso de la pirámide unifica las nubes de la derecha con parte de pirámide. Esto puede atribuirse a que en ambos casos existe una zona en la frontera entre ambas componentes donde la diferencia de matices no es tan alta y por lo tanto los límites se desdibujan. En el ejemplo de la primera imagen, las patas del potrillo no presentan gran diferencia de matiz con su sombra en el pasto. En el ejemplo de la segunda imagen, la parte oscura de las nubes no presenta gran diferencia de matiz con la parte sombreada superior de la pirámide. Esta poca diferencia de matices en cierta zona de la frontera provoca su unión que luego se extiende al resto de la componente que al tener menor diferencia de matices también puede terminar siendo incluida en la componente que se forma.

Para ver si podemos lograr una mejor segmentación variando el valor de k, probaremos un valor ligeramente mayor al considerado, ya que buscamos unificar más las componentes dentro de la pirámide y el cielo en la segunda imagen y los caballos en la primera.

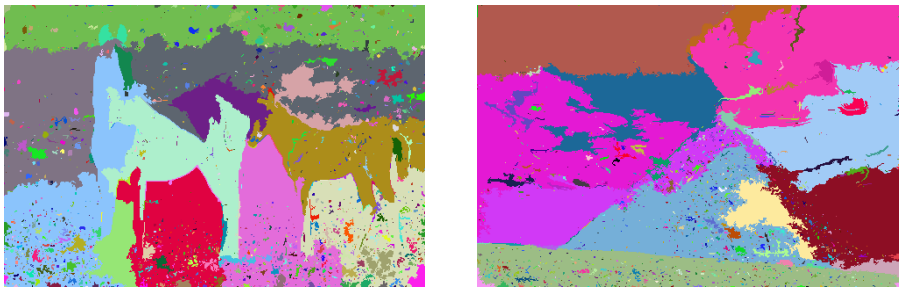


Imagen 39: Segmentaciones obtenidas para $k = 20000$

En la imagen 39 vemos cómo si bien en el caso de la imagen de los caballos obtenemos una segmentación más adecuada ya que lograr unificar más a las componentes de cada caballo por separado, en el caso de la imagen de la pirámide, no conseguimos lo que buscábamos. Esto es así ya que dos partes de la pirámide son confundidas con las nubes.

3.3.3. Muchas componentes homogéneas

Este grupo de imágenes posee muchas componentes con matices homogéneos. Veremos dos imágenes representativas de este subgrupo. La primer imagen (imagen 40) puede dividirse en las distintas piedras, el pasto, las montañas y el cielo. En el caso de la segunda imagen (imagen 42), buscamos distinguir los distintos pares de zapatos y el *background*.

Imagen 40: Imagen original

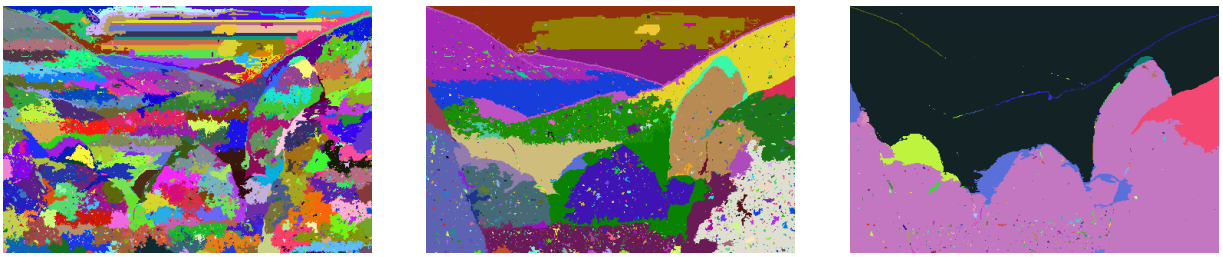


Imagen 41: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Podemos ver en la imagen 41 que el k que mejor segmenta las distintas piedras del *background* es aquel con valor 10000. Esto es así ya que para la imagen con $k = 1000$ no se puede distinguir ninguna piedra y sus componentes son demasiado pequeñas mientras que, por el contrario, en la imagen con $k = 100000$ todas las piedras quedan unificadas en una misma componente junto con el pasto de la misma forma que el cielo está unificado con las montañas. Esta segmentación no es satisfactoria ya que unifica demasiadas componentes que debieran ser distinguidas entre sí.

Imagen 42: Imagen original

Imagen 43: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Podemos ver como en este caso se repite el valor de k óptimo, ya que para $k = 10000$ se logran separar los pares de zapatos claramente. Algo a destacar es que si bien el valor óptimo se sigue manteniendo, el valor de $k = 1000$ deja de ser una segmentación poco adecuada, ya que podría ser útil si es de nuestro interés distinguir entre las distintas zonas dentro de cada par de zapatos, a ser: el interior, el exterior, las partes sombreadas, etc. Nuevamente vemos como el valor mayor de k produce una mayor unión de componentes a la deseada. Sin embargo, en este caso esto es más pronunciado ya que al tratarse de una mayor cantidad de componentes a distinguir, si considero componentes de mayor tamaño esto inevitablemente provocará una por demás incorrecta segmentación. También es destacable que al tratarse de componentes que poseen colores homogéneos distintivos muy distintos separados por barreras finas de color más oscuro, la segmentación obtenida resulta ser muy buena, ya que el algoritmo logra separar con mayor precisión cada componente.

3.3.4. Muchas componentes heterogénas

Pasamos a analizar el último caso de los que destacamos en la introducción de experimentación cualitativa. Para ello, en base a las experimentaciones ya realizadas, nuestra hipótesis sobre cuál será un valor de k adecuado es la siguiente. Primero, nos basamos en que son muchas componentes para decir que el valor de k no deberá ser muy alto ya que si no, quedarán en la imagen segmentada pocas componentes. Y en segundo lugar, como presentan muchos matices, resulta complicado dentro de lo que percibimos como una misma componente que ésta sea representada en la segmentación como una única componente, ya que al cada componente no ser homogénea, se necesita de un valor de k alto. Pero utilizar un valor de k alto perjudicaría lo ya mencionado causando que componentes observablemente distintas se vean representadas en la segmentación con una misma componente.

Veamos si se cumplen nuestras hipótesis en las siguientes imágenes sabiendo que en la imagen del puerto queremos distinguir las siguientes componentes observables: el muelle, el barco en el muelle, las personas en el muelle, la montaña, el cielo, el agua y las casas. Y en la imagen de las mujeres las distintas componentes serian: sus caras, sus pelos, los adornos en el sombrero de la muchacha de la izquierda y el *background*.

Imagen 44: Imagen original

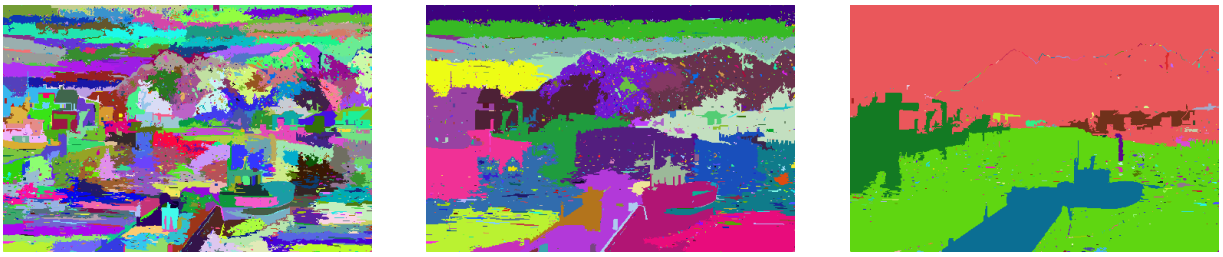


Imagen 45: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Imagen 46: Imagen original

Imagen 47: Segmentaciones obtenidas para $k = 1000$, $k = 10000$ y $k = 100000$.

Volvemos a notar que el valor de $k = 10000$ resulta el más adecuado, ya que un valor mayor unifica por demás las componentes mientras que un valor menor resulta en una mayor segmentación de las componentes que la buscada. Notamos que si bien $k = 10000$ resulta el más adecuado, no logra unificar todo lo que desearíamos a cada componente pero sin embargo un valor mayor ya no nos es útil; en el primer caso, unifica el cielo con las montañas y el barco con el muelle mientras que en la segunda imagen, directamente unifica las caras de las mujeres con la mayor parte del $background$. Esto comprueba la hipótesis planteada anteriormente.

4. Discusión y conclusiones

Teniendo en cuenta el análisis llevado a cabo a lo largo de todo este trabajo, podemos llegar a ciertas conclusiones basándonos en los resultados obtenidos. Desde el punto de vista del análisis cuantitativo (es decir, análisis de eficiencia y complejidades del algoritmo), podemos llegar a la conclusión de que la estructura de $tree$ con $path\ relinking$ resulta más eficiente en el peor caso (de todas las estructuras) mientras que si se considera un caso algo mejor para las primeras dos estructuras, $tree$ resulta tener la misma complejidad que $tree$ con $path\ relinking$ incluso siendo en la práctica ligeramente más rápido. Por lo tanto, si se realizan muchas operaciones $unite$ (es decir, no encontramos frente a un peor caso), conviene utilizar la estructura de $tree$ con $path\ relinking$ mientras que si las operaciones $unite$ realizadas son acotadas, puede utilizarse también la estructura de $tree$, que en la práctica puede resultar ligeramente más eficiente. Como el uso de memoria es el mismo en las tres estructuras, no se nos ocurre un caso en el que sea conveniente utilizar la implementación sobre $array$.

En cuanto al análisis cualitativo llevado a cabo, es decir, los resultados obtenidos de segmentaciones de imágenes, debemos notar diversas cosas. Teniendo en cuenta los análisis realizados para cada grupo planteado de imágenes, podemos concluir que si bien en algunos casos es posible aproximar un k general que sirva para la totalidad del grupo (caso de muchas componentes homogéneas con $k = 10000$), la diversidad de las imágenes junto con sus matices, fronteras entre componentes, entre otras, vuelve

muy dificultosa esta generalización. En la mayor parte de los casos, un $k = 10000$ logra segmentar las componentes deseadas, si bien puede contener subdivisiones dentro de cada componente. Esto en algunos casos puede ser arreglado por un valor de k mayor, pero sólo en donde exista una mínima cantidad de componentes, ya que de existir muchas componentes el k mayor resulta en componentes de mayor tamaño por lo que se unifican componentes observablemente distintas. Por otra parte, aquellas imágenes que presentan fronteras claras con una gran diferencia de matices resultan en una mejor segmentación resultante, ya que el algoritmo puede detectarlas con mayor facilidad.

A pesar de que es dificultosa una generalización, se puede llevar a cabo un proceso de aproximación $ad\ hoc$ variando los valores de k hasta obtener una segmentación satisfactoria que cumpla con un set de características deseada.

Referencias

[1] Er. Anjna, Er.Rajandeep Kaur: Review of Image Segmentation Technique, `http://www.ijarcs.info/index.php/Ijarcs/article/view/3691/3183`

[2] Pedro F. Felzenszwalb: Eficient Graph-Based Image Segmentation, `http://people.cs.uchicago.edu/ pff/papers/seg-ijcv.pdf`