DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Algoritmos sobre grafos

15 de julio de 2019 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Antola, Andrés Nahuel	233/17	andresnahuel35@gmail.com
Cifuentes, Santiago	624/17	santiagocifuentes66@hotmail.com
Faillace Mullen, Simon	475/14	sfaillacemullen@gmail.com
Recalde Campos, Julián	502/17	recaldej@hotmail.com.ar

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

1. Introducción

En el siguiente informe presentamos dos problemas distintos que pueden ser modelados con grafos. Para cada uno hacemos una descripción precisa del problema y ofrecemos distintas formas para resolverlo, junto a un análisis de las características de cada solución. Específicamente nos concentramos en las complejidades temporales de los algoritmos. Luego también realizamos un análisis experimental de los distintos algoritmos para poder determinar si la cota teórica se relaciona con los resultados empíricos. También nos interesa poder determinar en qué situaciones un método de resolución resulta más conveniente que otro.

Los dos problemas que desarrollamos se llaman 'Segmentation is my fault' y 'Llenalo con súper', y se estudian en la sección 2 y 3 respectivamente.

Finalmente presentamos algunas conclusiones respecto al desarrollo de estos algoritmos sobre grafos.

2. Segmentation is my fault

2.1. El problema

Uno de los problemas clásicos en procesamiento de imágenes y visión computacional es el de segmentar una imagen. Segmentar una imagen significa dividir los distintos píxeles en regiones que tengan una cierta coherencia entre sí. A grandes rasgos una segmentación apropiada debería obtener resultados similares a las segmentaciones que los humanos hacemos cotidianamente para reconocer los objetos dentro de una imagen, como en el siguiente ejemplo:

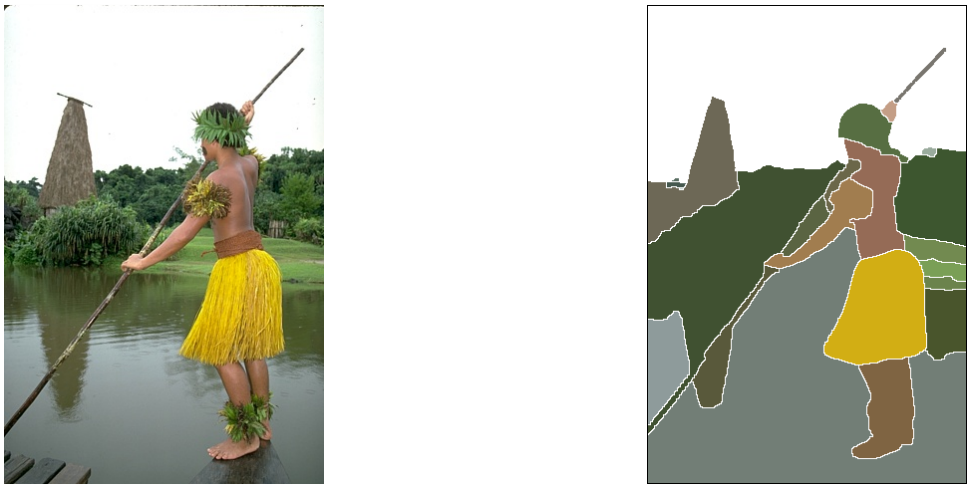


Figura 1: Segmentación de una imagen

La segmentación anterior distingue al cuerpo humano del resto de la imagen (en particular también distingue la vestimenta) y también separa el agua, el cielo y el bosque. En general los procesos de segmentación tienen que dividir las regiones en los objetos que un ojo humano podría identificar con un cierto nivel mínimo de detalle pero sin excederse en el mismo. Es decir, si hiciéramos una segmentación más fina (que intentara encontrar más regiones dentro de las regiones ya definidas) podríamos separar los arbustos del resto del bosque o reconocer distintos grados de intensidad del agua debido a los reflejos del fondo. También podríamos separar las hojas que tiene el chico cerca del pie, las cuales entraron en la región de la pierna.

El último caso llevaría a una segmentación que podríamos considerar más fina, pero separar los arbustos del resto del fondo no parecería lo apropiado. Esto ya define dos características fundamentales de toda segmentación:

- Toda segmentación sufre de un *Trade off* entre la precisión con la que se busca distinguir ciertas regiones y la necesidad de abstraer otras, las cuales a pesar de ser distintas en cuanto a los colores o formas representan un mismo objeto. Son los casos de la vestimenta y el arbusto respectivamente.

- Es fundamental un análisis cualitativo de los resultados, ya que la definición de una segmentación *apropiada* depende totalmente de la observación humana y podría variar en cada caso particular. Sin embargo se debe llegar a algún tipo de concenso para el desarrollo de los algoritmos.

Antes de definir formalmente el problema de la segmentación vamos a enunciar el enfoque que utilizamos en este trabajo. Para poder procesar una imagen usamos una representación de la misma como un grafo: en este grafo los vértices son los píxeles y los ejes representan relaciones que ciertos píxeles tiene entre sí (como por ejemplo cercanía) con un cierto peso dependiendo de las características de los píxeles que unen (color, brillo, posición). De esta forma se puede definir el problema de segmentación sobre grafos:

Dado un grafo $G = (V, E)$ queremos encontrar una segmentación S, la cuál es una partición de V en componentes de tal forma que todo conjunto $C \in S$ corresponda a una componente conexa en $G' = (V, E')$ con $E' \subseteq E$. En otras palabras una segmentación es inducida por un subconjunto de E. Nuestro objetivo es que estas componentes conexas disjuntas en E' tengan una cierta correlación entre si de tal forma que representen una segmentación apropiada.

Los criterios que utilizamos para generar esta partición se corresponden con los de [1], en donde también se demuestra formalmente que los algoritmos propuestos son correctos y el output cumple con los criterios. Veamos cuales son estos.

Una primera intuición nos dice que si un conjunto de píxeles pertenece a una misma región entonces deben ser de un color similar. Es por eso que para definir los costos de los ejes del grafo incorporaremos la información de cada píxel. Debido a que nuestros algoritmos trabajan sobre imágenes en blanco y negro el costo asociado al eje entre dos vértices v y w se define como:

$$c(vw) = |I(v) - I(w)|$$

donde $I(v)$ representa el valor de la intensidad asociada a v. Para procesar imágenes a color se puede trabajar con la norma entre ambos píxeles (considerando a sus colores **RGB** como puntos en $\mathbb{R}^3$) o bien procesando 3 segmentaciones con cada componente y luego tomando la intersección entre todas.

Como dijimos antes, un conjunto de píxeles pertenece a la misma región si sus colores o intensidades son similares. Pero podría haber componentes donde las diferencias de intensidades sean fuertes pero aún así representen un mismo objeto en nuestra interpretación humana. Esto ocurre debido a que las diferencias de intensidades son uniformes. A esto le llamamos *coherencia*, y es una característica local de cada región. Mirando la figura 1 vemos que el fondo no es necesariamente de un mismo color, sino que es una gran región de píxeles donde las diferencias entre ellos se mantienen constantes (las mayores diferencias son entre verdes y negros). Algo similar ocurre con el río, que a pesar de cambiar de tonalidades sigue siendo mayormente un mismo objeto. Convendría tener un predicado que permita decidir si un píxel fuera de la región es coherente con los que ya están en una componente. Entonces vamos a definir la *diferencia interna* de una componente $C \in S$ como:

$$Int(C) = \max_{\forall e \in C} c(e)$$

donde $c(e)$ es el peso de la arista e. $Int(C)$ es el peso de la arista de mayor peso dentro de una región C (si no hay ninguna arista en C se le asigna 0), por lo que sabemos que para toda arista e dentro de C su costo será menor a $Int(C)$. Entonces si un vértice v de C tiene un eje con un vértice w fuera de la región y $c(vw) < Int(C)$ agregar el vértice w a C no representa un gran cambio en la coherencia de los ejes dentro de la región, pues ya hay vértices con mayor diferencia. Interpretando lo anterior con píxeles queremos decir que w es un píxel cuya diferencia de color o intensidad no es muy grande teniendo en cuenta las diferencias ya presentes en C. Por ende agregar w a la región de C parecería correcto. Sin embargo w podría ya pertenecer a una segunda región C', por lo que unir w y v implicaría juntar ambas regiones C y C'. Tenemos que definir un segundo predicado que defina los casos en que vale la pena unificar dos componentes distintas. Esto es, en qué casos unir dos regiones no afecta la coherencia de la segmentación final. Esta es una característica global de la imagen.

Uno haría esto cuando ambas componentes tienen una coherencia similar, pero a la vez deberían estar cerca en términos de valor absoluto. Por ejemplo, el río y el fondo podrían tener una coherencia interna similar, pero la diferencia entre las tonalidades del río y las del fondo son muy grandes, o por lo menos muy grandes teniendo

en cuenta la coherencia interna de cada una. Si bien ambas componentes tienen una variabilidad interna similar, los ejes de su frontera tienen un peso muy alto en comparación a sus coherencias. Vamos a definir:

$$Dif(C_1, C_2) = \min_{\forall v_i \in C_1, \forall v_j \in C_2, v_i v_j \in E} w(v_i v_j)$$

Para ver qué tan grandes son las diferencias entre ambas regiones tomamos el eje de peso mínimo en su frontera. Si no hay ningún eje definimos $Dif(C_1, C_2) = \infty$. Con estos elementos ya podemos escribir un predicado D que dadas dos regiones C_1 y C_2 nos diga si están correctamente separadas (si es correcto que sean dos componentes distintas).

$$D(C_1, C_2) = (Dif(C_1, C_2) > MInt(C_1, C_2))$$

En el predicado anterior $MInt$ se define como:

$$MInt = \min(Int(C_1) + \tau(C_1), Int(C_2) + \tau(C_2))$$

A $MInt$ lo llamamos *diferencia interna mínima* y representa la menor entre las dos diferencias internas sumando una función τ que controla el valor final de $MInt$. La utilizamos porque hay casos en que Int no refleja apropiadamente el nivel de coherencia interna dentro de una componente (como cuando C tiene un solo vértice). Entonces podríamos definir τ como:

$$\tau(C) = k/|C|$$

con $k > 0$. De esta forma salvamos los casos en que Int subestima el valor de la coherencia interna, pues cuando haya pocos elemento en C la función τ reevaluará a $MInt$ para darle más sentido. Nótese que puede usarse cualquier función no negativa para τ y que ello definirá la preferencia por un tipo de componentes dentro de la segmentación, y que variar el k decide la preferencia por componentes grandes o chicas (valores de k mayores o menores respectivamente).

El predicado D devuelve *true* cuando la diferencia entre componentes es más grande que la mínima diferencia interna entre las dos (agregando la función τ). Por ende si para todo par de regiones C_1 y C_2 en S se cumple $D(C_1, C_2)$ podríamos considerar que S es una segmentación suficientemente *fina*, ya que para todo par de componentes estas son considerablemente diferentes (siguiendo el criterio de D el cual resulta razonable) y por ende no hay objetos de la imagen que hayan sido divididos en varias partes.

Por otro lado, nos gustaría estar seguros que la segmentación no es demasiado *gruesa*. Esto quiere decir que no haya otras segmentaciones que tengan las mismas componentes C excepto por algunas, las cuales fueron subdivididas en más regiones que cumplen D, consiguiendo una nueva segmentación que no es demasiado fina y a la vez es menos gruesa. En ese caso la segmentación original habría unido componentes que debían estar separadas.

Entonces queremos diseñar un algoritmo que tome un grafo y genere una partición de tal forma que no sea ni muy fina ni muy gruesa de acuerdo a D. Para eso vamos a utilizar una variación del algoritmo de árbol generador mínimo de Kruskal. Este procedimiento tiene como invariante un bosque con arboles mínimos parciales. Podemos interpretar a estos arboles como a las distintas componentes de nuestra partición final, pero en cada iteración de Kruskal veremos si unir las componentes de acuerdo a nuestro predicado D. De esta forma nos aseguramos que las segmentaciones generadas no serán ni muy finas ni muy gruesas (la demostración de esto está en [1]. El pseudocódigo del mismo es el siguiente:

Algorithm 1

```
1:  procedure SEGMENTATE(G,τ)
2:      Sort(E)                                   ▷ Ordeno los ejes en función de su peso
3:      ds ← DisjointSet(|V|)                     ▷ Un Disjoint Set de tamaño |V|)
4:      Int ← Array(|V|, 0)                       ▷ Un arreglo de tamaño |V| inicializado en 0
5:      for i=0...|E|-1 do
6:          (v, w, c(cw)) ← E[i]
7:          id_v ← ds.find(v)
8:          id_w ← ds.find(w)
9:          if id_v ≠ id_w and !D(id_v, id_w, τ) then
10:             ds.Unite(id_v, id_w)
11:             newInt ← max(Int(id_v), Int(id_w), c(vw))
12:             Int(v) = newInt
13:             Int(w) = newInt
14:     return ds
```

El vector Int se encarga de almacenar el valor de Int para cada componente, y para acceder a la coherencia interna de la componente de un elemento v tenemos que buscar en el arreglo Int la posición correspondiente a su identificador. Es por eso que cuando unimos dos componentes C_1 y C_2 tenemos que revisar cual será el nuevo valor de Int: como el eje de mayor costo era $Int(C_1)$ o bien $Int(C_2)$ y agregamos un solo eje tomamos el máximo entre los 3 (en particular sabemos que eje recién agregado es mayor a uno de los dos Int ya que sino no se habría ejecutado la unión). Gracias a que tenemos los valores de Int y los tamaños de las componentes guardados (este se verá más adelante con *DisjointSet*) la implementación del predicado D es directa.

Vemos que las únicas variaciones respecto al algoritmo de Kruskal son las líneas dedicadas a actualizar Int y la inclusión del predicado D en el *If*. La complejidad del algoritmo puede ser descrita entonces como:

$$T(n+m) = O(s + mf + nu)$$

donde s es el costo de ordenar las aristas, f el costo de los *find* y u el costo de los *Unite*. Sabemos que la cantidad de *Find* es $O(m)$ pues hay dos *find* por cada iteración. Por otro lado puede haber como mucho n *Unite*, pues con esa cantidad ya se unen todas las componentes disconexas iniciales.

El sorting se puede hacer con complejidad $O(mlog(m))$ con m la cantidad de aristas. Pero la complejidad de las otras operaciones depende de la implementación que se utilice de *DisjointSet*. Recordemos que esta estructura de datos representa un conjunto de conjuntos disjuntos y debe ofrecer los métodos *find* y *Unite* que permiten obtener un identificador de un conjunto dentro de la estructura y unir dos conjuntos disjuntos dentro de la estructura respectivamente. También incluiremos dentro de los métodos de *DisjointSet* una función que dado un elemento i nos devuelva el tamaño del conjunto al que pertenece. Esto lo utilizaremos para calcular el valor de τ (si bien también sirve para optimizar las operaciones del mismo *DisjointSet*, como se verá más adelante). Si mantenemos los tamaños actualizados en la misma estructura esta última función toma tiempo constante. Para este trabajo realizamos 3 implementaciones distintas:

- *DisjointSet* con arreglo de representación.
- *DisjointSet* con árbol de representación.
- *DisjointSet* con árbol de representación y *Path Compression*

La primera es la más simple, pues consiste de un vector donde almacenamos el identificador de cada conjunto. De esta forma para implementar *find* solo hay que acceder al vector y devolver el identificador. Sin embargo el Unite es lento pues para unir dos conjuntos x e y hay que recorrer todo el vector de identificadores y asignarle a cada conjunto que tenga el identificador de x el identificador de y (o viceversa). También hay actualizar los tamaños de las componentes unidas. Esto tiene una complejidad de $O(n)$. Entonces el *for* del algoritmo de *Kruskal* tendrá una complejidad de:

$$T(n+m) = O(mf + nu) = O(m + n^2)$$

Con la segunda implementación vamos a representar cada componente disjunta como un árbol. Inicialmente todos tendrán como padre a si mismos, por lo que serán arboles con un solo nodo. A medida que se ejecuten uniones de conjuntos los arboles se harán más profundos, uniendo arboles entre sí. De esta forma para obtener el identificador de su componente un elemento deberá fijarse el valor de su padre. Si su padre es él mismo entonces él es su identificador, y si no entonces su identificador será el identificador de su padre. Luego la complejidad de la operación *find* será equivalente a la profundidad que tome el árbol de cada componente. Como no hacemos nada para balancear al árbol podría pasar que su profundidad alcance valores proporcionales a n, por lo que *find* tendrá una complejidad $O(n)$. Por otro lado para hacer los *Union* solo hay que enlazar dos árboles, y esto se hace fácil estableciendo a la raíz de un árbol como padre de la raíz del otro. Sin embargo para encontrar la raíz habrá que hacer un *find*, por lo que en el peor caso también será $O(n)$. El pseudocódigo de estas dos funciones es el siguiente:

Algorithm 2

```
1: procedure FIND(i)
2:     if padre(i) = i then
3:         return i
4:     return find(padre(i))
5: procedure UNITE(i, j)
6:     if find(i) ≠ find(j) then
7:         size(find(j)) ← size(find(j)) + size(find(i))
8:         padre(find(i)) ← padre(find(j))
```

Las lineas relacionadas al *size* fueron agregadas para la utilización del predicado D pero no afectan a la complejidad final, pues su costo es asintóticamente idéntico al del resto de las operaciones. Con estas funciones la complejidad del *For* de *Kruskal* es:

$$T(n+m) = O(mf + nu) = O(mn + n^2)$$

Vale notar que si bien la complejidad es peor que la de la primera implementación esto se debe a que tomamos en cuenta los peores casos posibles, los cuales son muy pocos en este caso (cuando los arboles se arman de tal forma que tengan una profundidad $O(n)$) mientras que si usamos el arreglo de representación todos los casos tendrán un desempeño similar. Todo esto se estudia con mayor precisión en la experimentación.

Por último implementamos una estructura de representación con un árbol pero intentando evitar que el árbol se 'estire' demasiado. Para hacer esto utilizamos dos métodos: *Path Compression* y *Union by sizes*. Lo que vamos a hacer es reasignar los padres cada vez que realizamos un *find*, y cuando hagamos los *Unite* vamos a intentar unir los arboles de manera inteligente.

En el *find* anterior estamos recorriendo el árbol cada vez que buscamos al padre, pero desaprovechando el hecho de que podríamos modificar la sección del árbol que estamos atravesando. Entonces lo que haremos es reasignar el padre de los nodos por los que vamos pasando, y de esta forma reduciremos la profundidad del árbol. Cuando lleguemos a la raíz del árbol vamos a asignarle a todos los nodos por los que pasamos ese nodo como padre, por lo que todos los nodos que atravesamos quedarán a distancia 1 de la raíz, manteniendo la altura del árbol lo más pequeña posible.

Para hacer los *Unite* aprovecharemos el hecho de que conocemos los tamaños de cada árbol, por lo que pondremos al árbol más grande a menor distancia de la raíz. Esto tiene sentido ya que como el árbol es más grande entonces hay una mayor probabilidad de que en el futuro se haga un *find* a uno de sus elementos. El pseudocódigo de esto es el siguiente:

Algorithm 3

```
1: procedure FIND(i)
2:     if padre(i) ≠ i then
3:         padre(i) = find(padre(i))
4:     return padre(i)
5: procedure UNITE(i, j)
6:     p_i ← find(i)
7:     p_j ← find(j)
8:     if p_i ≠ p_j then
9:         if size(p_i) < size(p_j) then
10:            swap(p_i, p_j)
11:        size(p_i) ← size(p_i) + size(p_j)
12:        padre(p_j) = p_i
```

Como antes la complejidad depende de los *find*, y a su vez estos dependen de la máxima profundidad que puedan tomar los árboles. Está probado en [2] que realizar k operaciones *find* o *Union* bajo esta implementación da una complejidad amortizada de $O(k\alpha(n)))$. Luego la complejidad final es:

$$T(n+m) = O(mf + nu) = O(m\alpha(n) + n\alpha(n))$$

donde α es la inversa de la función de Ackerman (la cual crece extremadamente rápido, por lo que en la práctica $\alpha(n) = O(1)$). Por ende la implementación más eficiente parecería ser esta última.

Una última consideración que falta tomar es la forma en que vamos a disponer los ejes del grafo. Para esto vamos a considerar un modelo *8-connected* donde cada píxel va a tener un eje a los 8 píxeles circundantes. Es interesante considerar que usando este método cada vértice no va a tener más de 8 aristas, por lo que $m = O(n)$. Otras opciones que se pueden tomar son *4-connected*, en donde a cada píxel se lo conecta con su vecino superior, inferior y a ambos lados, o bien conectar los píxeles que cumplan una cierta función dependiendo de la cercanía y la similitud de sus valores, lo que se conoce como *Nearest Neighbor*.

2.2. Experimentacion

Para iniciar con la etapa de experimentación, hemos planteado un caso de estudio inicial. Nuestro objetivo es analizar, comparar y concluir, que estructura de datos es la mas rápida en tiempo para ejecutar el algoritmo descripto en la sección anterior.

Tomando imágenes distintas, de tamaño variado, con representaciones de figuras distintas, queremos comparar el tiempo de ejecución de las tres estructuras distintas, el **arreglo**, el **arbol** y el **arbol** con **Path Compression**.

Las imágenes elegidas fueron tomadas de `https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds`, sitio donde se encuentran una colección variadas de fotos para el estudio de segmentación de imágenes. Con el objetivo de poder ejecutar los casos de tests presentados en este trabajo en un tiempo razonable, hemos decidido tomar una muestra de 12 imágenes de esta colección, las cuales, en nuestra opinión, muestran figuras y tamaños variados para el correcto estudio del algoritmo.

(a) 38092.jpg

(b) 14037.jpg

(c) 108005.jpg

(d) 21077.jpg

(e) 220075.jpg

(f) 296059.jpg

(g) 3096.jpg

(h) 300091.jpg

(i) 37073.jpg

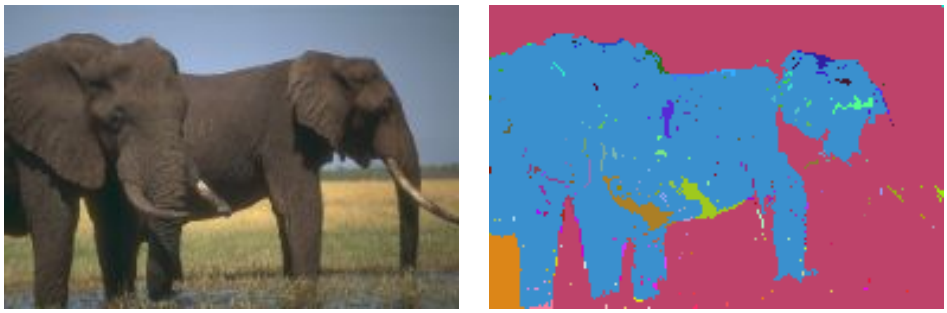
(j) 167083.jpg

(k) 69015.jpg

(l) 86000.jpg

2.2.1. Performance en distintas imágenes

Para comenzar el análisis del tiempo de ejecución de cada estructura de datos, hemos decidido tomar las 12 imágenes presentadas anteriormente, y fijando un valor para la variable K, ejecutar para cada imagen, el algoritmo implementado con el array de representación, con el árbol de representación y con el árbol de representación con Path Compression. Elegimos fijar el valor K en 10000, debido a que este mismo genera imágenes con una cantidad de segmentos razonables, lo cual podrían describir un caso promedio del uso de este algoritmo.



Original $K = 10000$

Figura 2: Ejemplo de imagen aplicando el algoritmo con un valor de K = 10000

Hemos implementado este test en el archivo ***main.cpp***, dentro de la carpeta ***Ej1***. La función correspondiente denominada **benchmark_functions_same_K**, toma las imágenes guardadas en la carpeta ***images***, las convierte a un archivo de extensión ***.pgm***, es tomado y transformado en un vector de tuplas de enteros, y luego, tomando el tiempo de ejecución promedio para cada uno, repitiendo el proceso 5 veces, imprime los resultados en el archivo ***benchmark_functions_same_k.csv***, localizado en la carpeta ***Ej1/tests/benchmark_same_K***. Por ultimo, gracias un script programado en Python, se convierten los resultados de nuevo a imágenes, para una mejor visualización.

A continuación, presentamos los resultados, medidos en milisegundos, de ejecutar esta función en una computadora con un procesador Intel Core i3 4330, con una velocidad de clock de 3.50GHz.

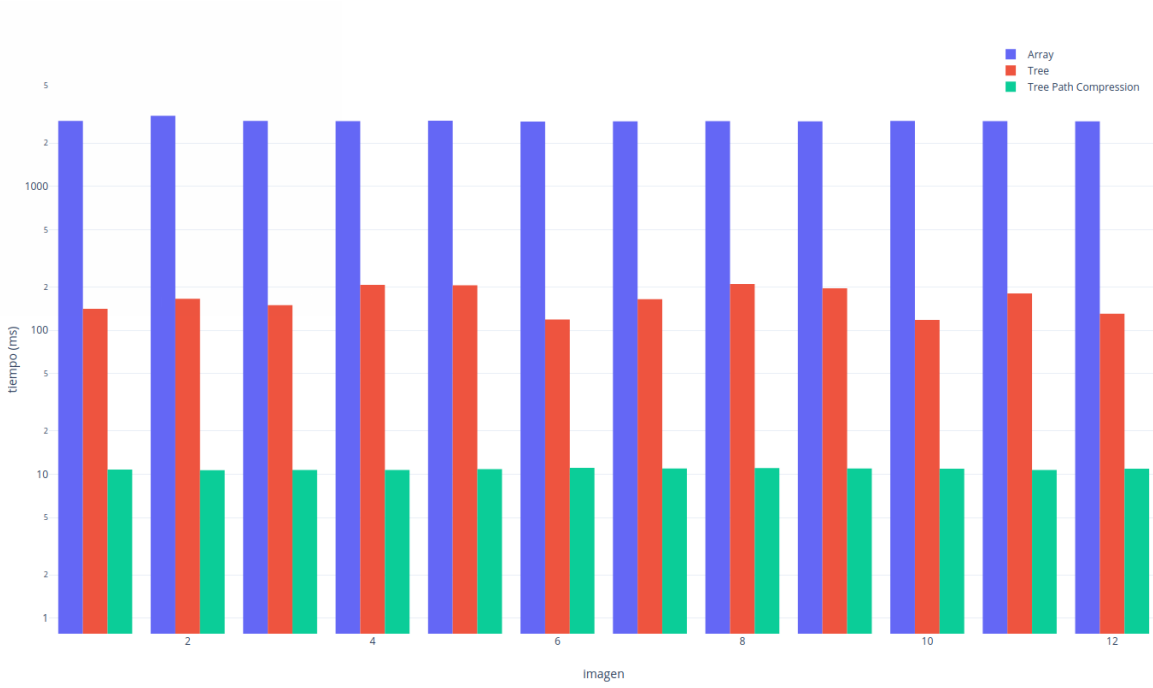
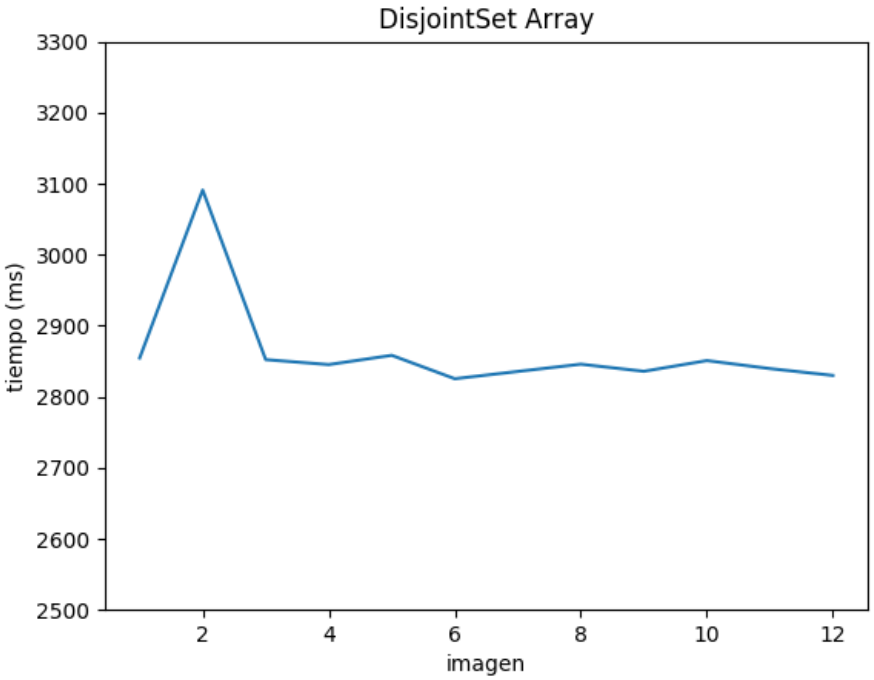


Figura 3: Diferencia en el DisjointSet implementado en un Array, un Tree con y sin Path Compression

Como podemos ver a simple vista, el algoritmo sobre la estructura de datos del Array es notablemente mucho mas lenta que las otras dos. En el mejor de los casos, procesando la imagen f, es un %2394 mas lento que su ejecución en un Tree.

Empezamos analizando la estructura del Array, la cual, según hemos estudiado en la sección 2,1, posee una complejidad de $O(m + n^2)$, con **m** la cantidad de aristas y **n** la cantidad de píxeles. Como podemos ver en el grafico, y también a continuacion, no hay una significante variación, entre el peor y el mejor de lo casos presentada por esta estructura de datos.



Esto se debe a que la cantidad de píxeles (n) es siempre la misma, y como, el valor de m es fijo en todos los casos, excepto en los bordes, como hemos visto en la complejidad, n^2 en este caso, aumenta de forma muy rápida, y m representa un menor crecimiento.

Ahora, si analizamos la ejecución del algoritmo implementado en el Tree, vemos una variación notable entre los distintos resultados, como por ejemplo entre la imagen f y la h, las cuales tienen una diferencia del casi el doble de tiempo de ejecución. Si comparamos la diferencia de las imágenes resultantes del algoritmo, en una primera instancia, podríamos notar que en la imagen h hay una mayor cantidad de segmentos, y a su vez, estos no son tan uniformes en comparación con la imagen f.

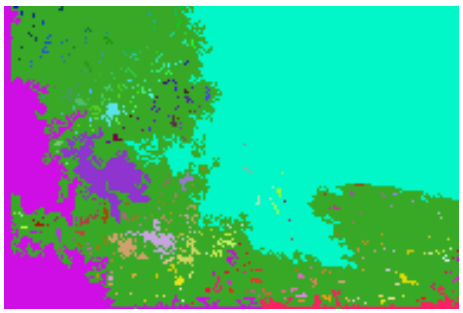


imagen (h)

imagen (f)

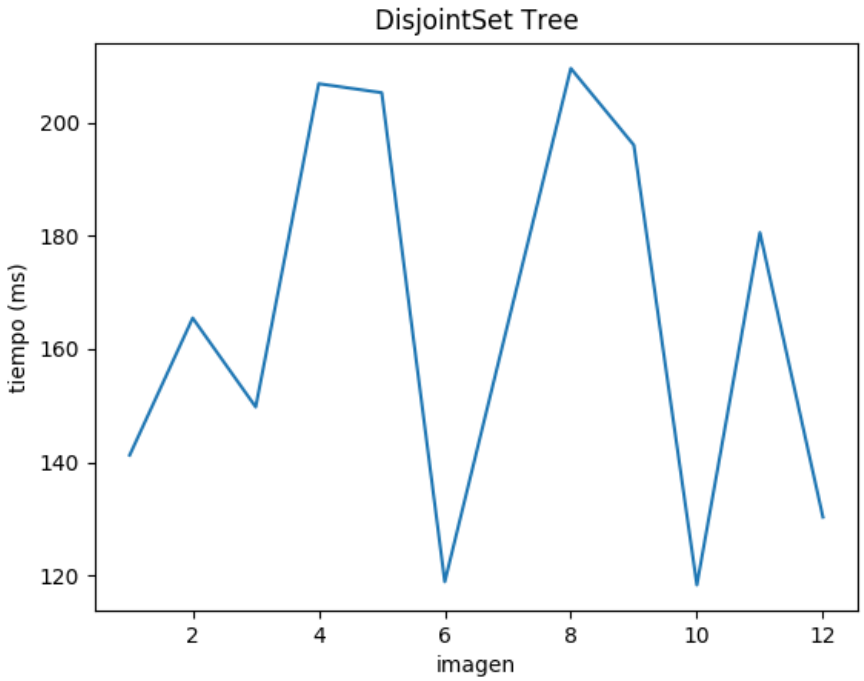


Figura 4: Tiempo de ejecución en un Tree

Hemos descripto en la sección anterior, esta estructura de datos se ve muy afectada por sus casos promedios y su peores casos. Como podemos ver en la imágenes resultantes, la imagen h cuenta con grandes segmentaciones del mismo color, por lo tanto, estos segmentos se verán representados con arboles de mayor profundidad. Esto afecta de forma negativa a la función ***find***, y como consecuencia directa, aumenta el tiempo de ejecución.

Por ultimo, si comparamos la estructura de Tree con su implementación añadiendo la técnica de *Path Compression*, podemos notar diferencias notables entre uno y el otro.

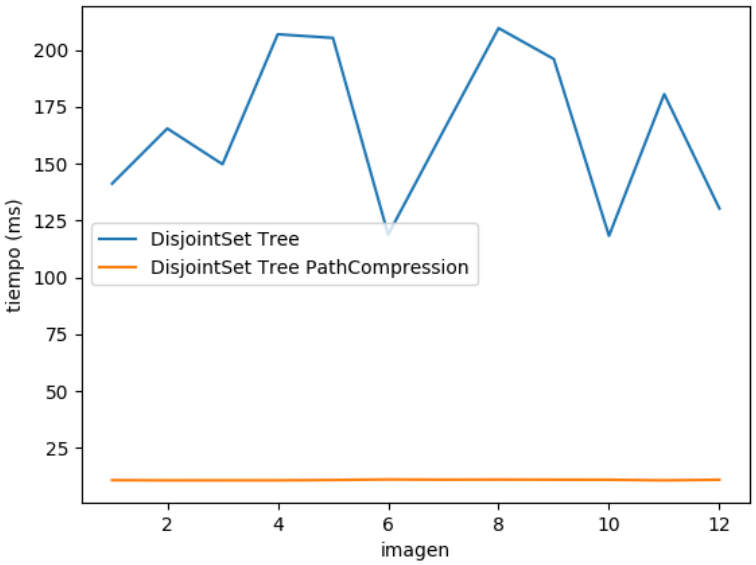


Figura 5: Diferencia en el DisjointSet implementado en un Tree con y sin Path Compression

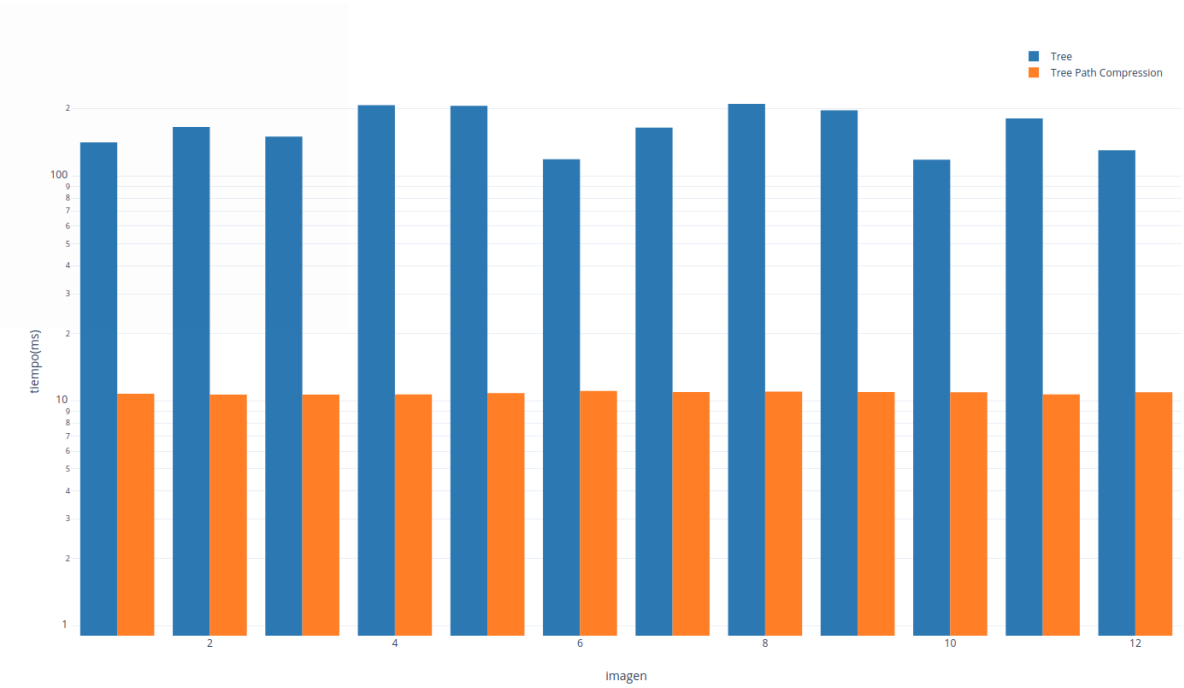
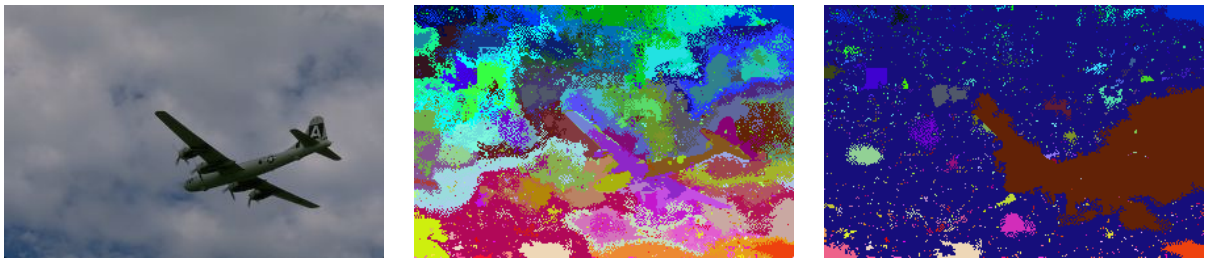


Figura 6: Resultados expresados en un gráfico de barras

Esto se debe a que al achicar el árbol en cada ***find***, estamos contrarrestando los efectos negativos de la representación sobre un árbol. Es decir, según lo visto por la complejidad en la sección ***2.1***, solo estamos recorriendo la estructura una vez por píxel, y encontrarlo en el árbol es mucho menos costoso que la implementación del Tree, ya que acortamos la distancia del nodo a la raíz.

2.2.2. Performance variando k.

Como pudimos ver en la sección ***2.2.1***, la implementación del algoritmo con un Tree (sin *Compression Path*), tiene resultados muy variados con respecto a la imagen resultante. Para estudiar este fenomeno de manera mas profunda, hemos implementado la función **benchmark_functions_diff_K**, en el archivo ***main.cpp***. A diferencia del experimento anterior, esta función toma solo una imagen, la imagen (g), y repite el mismo procedimiento que la función anterior, pero, en vez de iterar sobre un conjunto de imágenes, itera sobre un valor de k. Una vez terminado, imprime los resultados en el archivo ***benchmark_functions_diff_k.csv***, dentro de la carpeta ***Ej1/tests/benchmark_diff_K***. Al terminar, se generaran dentro de la capeta ***image_results***, las imagenes resultantes de variar el valor k sobre la misma imagen. Se ha decidido variar el valor de k entre 500 y 18000. Esto se debe a que notamos que los resultados con un valor menor a 500, eran muy similares, al igual que las imágenes resultantes de aplicar el algoritmo con un valor mayor a 18000.



original $k = 500$ $k = 18000$

Figura 7: Imagen original, al comienzo del algoritmo y al final

Si nuestra hipótesis anterior, que la estructura del Tree se ve afectado por sus segmentaciones, y como observamos en las imágenes superiores, y si nuestro análisis de la complejidad de las otras dos estructuras es correcta, los resultados de este experimento deberían demostrar una diferencia de tiempo notable entre el Array y el Tree y Tree con Path Compression, y a su vez, el algoritmo implementado en el Tree, debería ser el menos constante a lo largo de la ejecución del algoritmo.

A continuación, los resultados del experimento.

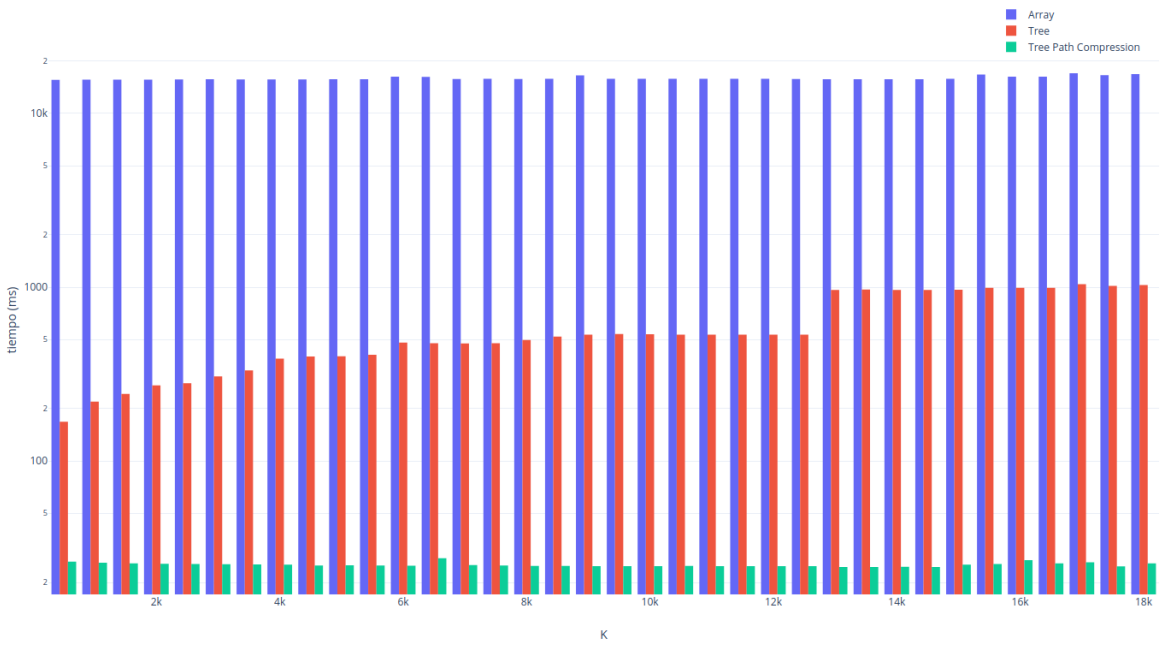
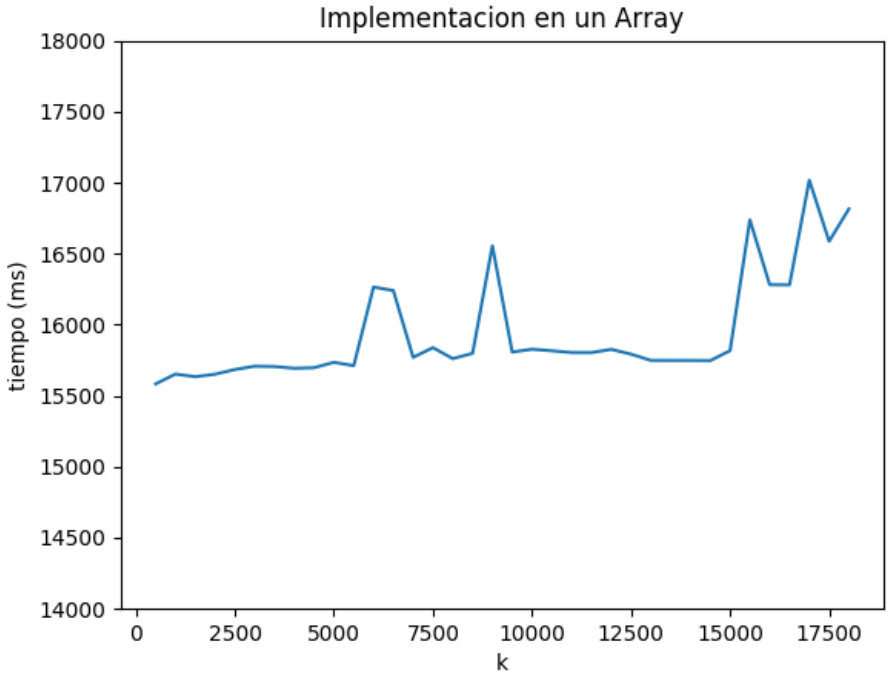


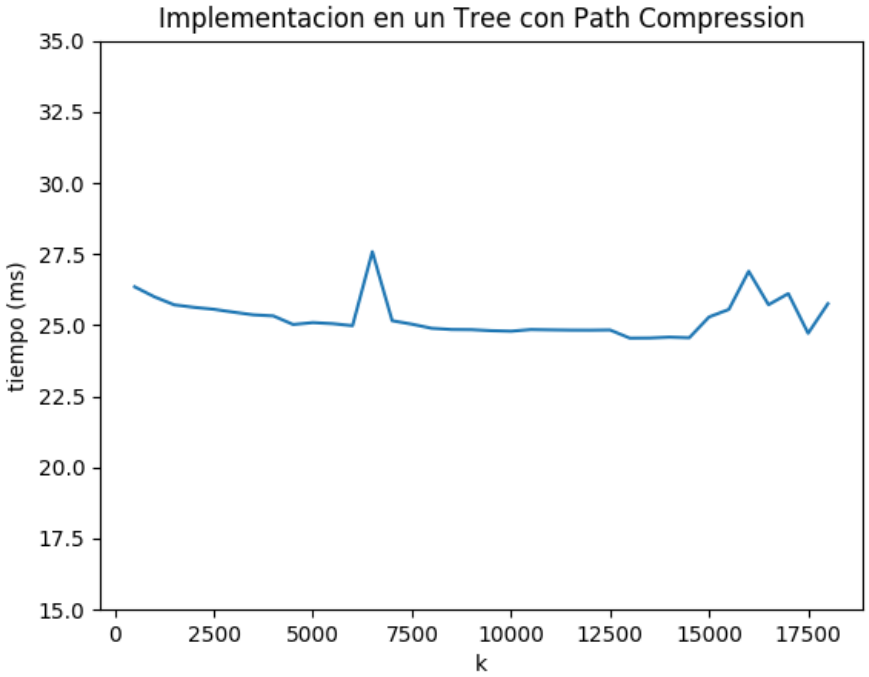
Figura 8

Como podemos ver en el grafico, se puede apreciar que Array tiene una variación muy leve, mientras que Tree posee la variación mas grande. Al contrario de Tree con Path Compression, su tiempo de ejecución crece constantemente.

Viendo el gráfico, generado con los resultados, el Array posee una muy leve variación a lo largo del experimento. Esto, como ya hemos visto, se puede atribuir a que la complejidad temporal de este se ve muy afectado por la cantidad de píxeles, y no tan afectado por los bordes elegidos en el algoritmo.



Con aun menor variación, podemos observar que la implementación sobre un Tree con Path Compression cumple con las mismas expectativas que en el experimento anterior, anulando los efectos negativos del Tree sin Path Compression y siendo en complejidad temporal, el mas eficiente.



Luego, si analizamos el caso del algoritmo ejecutado sobre un Tree, encontramos que su tiempo de ejecución crece a lo largo del experimento. Como podemos ver claramente en la figura 6, el tiempo de ejecución crece a medida que crece el valor de k. También vemos un fuerte crecimiento entre los valores 12500 y 15000. Mas específicamente, como podemos ver en la tabla, el aumento repentino ocurre entre 12500 y 13000.

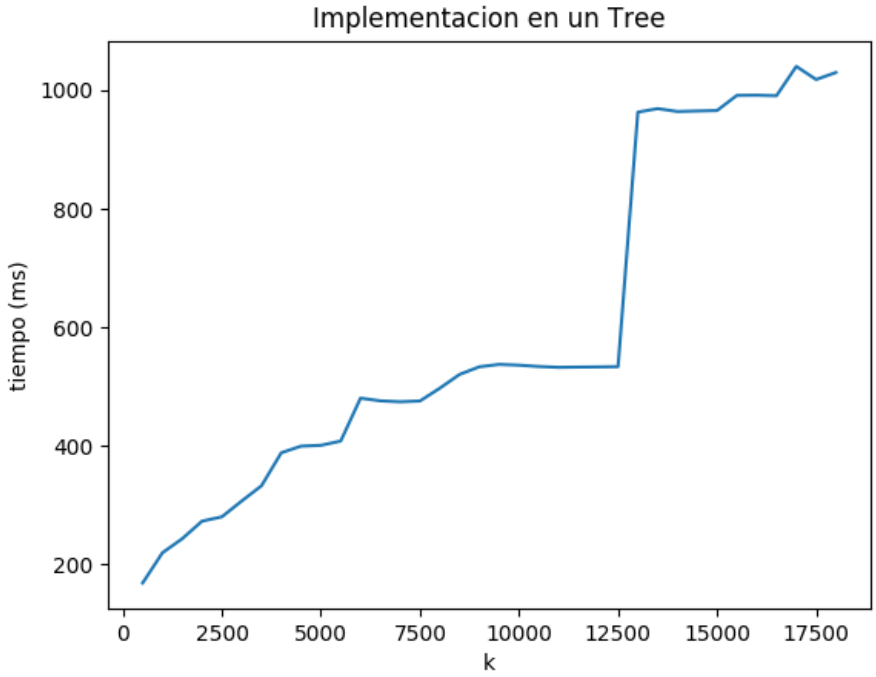
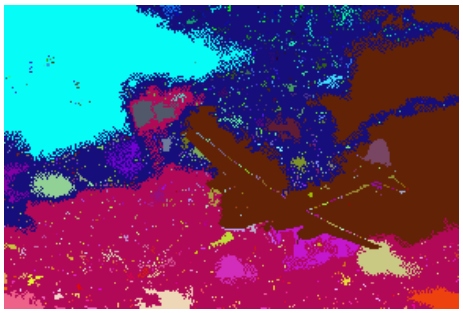
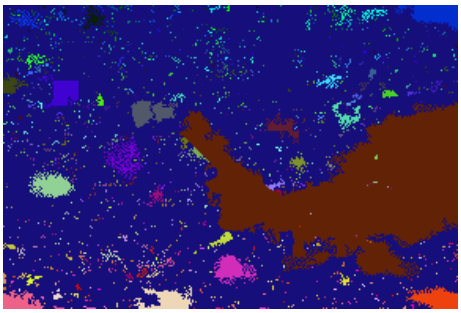


Figura 9

Si analizamos estas dos ultimas imágenes, podemos ver que hay un cambio repentino entre la cantidad de segmentaciones. Esto produce un efecto negativo en nuestra estructura de datos. Por ejemplo, en la imagen de la derecha, el segmento de color azul oscuro representa la mayor parte de la imagen. Como consecuencia, el árbol de este segmento es el mas profundo de todos, por lo tanto, en el algoritmo de Kruskal, la función ***find*** tardara un tiempo mucho mayor, que por ejemplo, en el segmento representado con el color marrón.



$k = 12500$



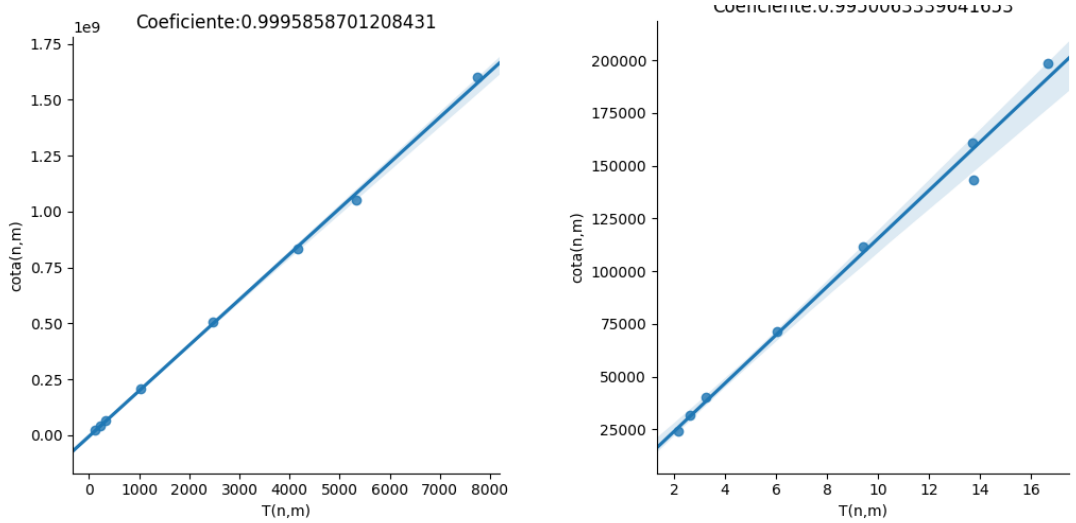
$k = 13000$

2.2.3. Complejidades

A continuación, analizaremos las complejidades pautadas en la sección ***2.1***, y las vistas en la experimentación. Para ello, implementamos la función **complexity**, que toma 7 imágenes distintas, de tamaño distinto, y guarda los resultados de ejecutar el algoritmo en las distintas estructuras de datos. En el archivo ***complexity.csv***, dentro de la carpeta ***tests/complexity***, podemos analizar el tiempo que tarda en ejecutarse el algoritmo en cada caso.

Como hemos visto a lo largo de este trabajo, la implementación de Tree es la que ofrece mas inconsistencias, comparadas con las otras. Como habíamos planteado, esperamos que tanto Tree con Path Compression, como Array, se vean afectadas correlativamente con el tamaño de la imagen, mientras que Tree, se ve afectado por la cantidad de peores casos de segmentación junto con el tamaño de imagen.

Si estudiamos el coeficiente de Pearson de las estructuras de Array y Tree con Path Compression, con sus complejidades temporales, vemos que ambas son superiores a 0.99.



Array Tree con Path Compression

Figura 10: Correlacion de Pearson de ambas estructuras de datos.

Mientras que en el caso de Tree, si bien, el coeficiente es alto, no es tan exacto como los dos anteriores.

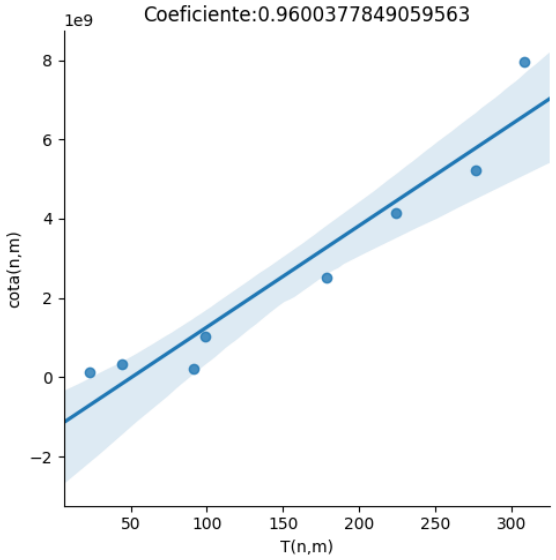


Figura 11: Correlación de Pearson de la estructura Tree

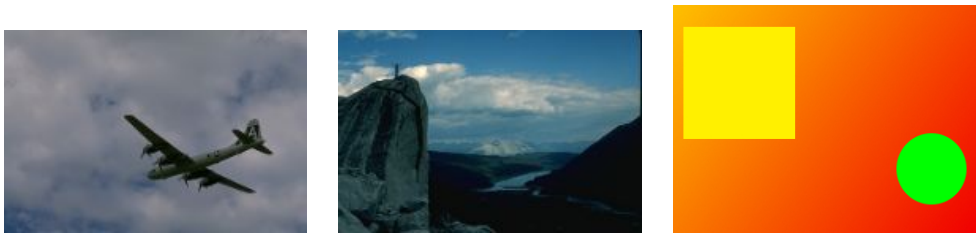
En conclusión, hemos podido demostrar las complejidades establecidas en la sección ***2.1***. A su vez, hemos

podido demostrar que en las experimentaciones realizadas, el algoritmo implementado en un Tree con Path Compression es mucho mas eficiente, y hemos estudiado las distintas variables que afectan a la estructura del Tree.

2.3. Análisis cualitativo

En la primera parte de la experimentación, hemos estudiado la performance del algoritmo implementado en diferentes estructuras, y como las variables perjudican a este mismo. Si bien hemos podido analizar como afecta variar la segmentación de una imagen, aun no hemos observado el aspecto empírico mas importante de este trabajo, el cual es, con que valores logro tener la "mejor"segmentación. Aclaramos que la mejor segmentación es aquella en la cual podemos distinguir los objetos mas distintivos entre ellos de forma clara.

Para este experimento, proponemos estudiar un conjunto de tres imágenes, dos fotografías y una imagen creada digitalmente. La $imagen(a)$, se ha elegido por la razón en la cual el objeto y el fondo están fuertemente distinguidos por su intensidad, pero también, el fondo presenta áreas donde su intensidad varia levemente (en las nubes) y varia mayormente (en el cielo despejado). Esta foto nos servirá para analizar que valor de **k** se necesita para lograr el mejor resultado. A su vez elegimos la instantánea de la $imagen(b)$, la cual presenta poca variabilidad entre sus objetos. Esta nos sera de utilidad a la hora de demostrar la capacidad del algoritmo de segmentar objetos con poca variabilidad de intensidad entre ellos. Por ultimo la ultima imagen comprende dos figuras geométricas, y un fondo en degrade. Se ha elegido colorear el cuadrado de color amarillo, el cual nos servirá para testear la segmentación en cuanto al cambio de intensidad entre esta figura y el fondo, ya que en el lugar que se posiciona, la diferencia de intensidad es notable, pero no lo es tanto en comparación con el circulo, el cual posee la mayor intensidad de toda la imagen. En la figura 12 se muestran las fotografías utilizadas en este experimento.



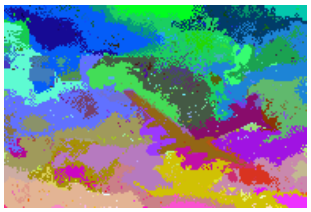
$imagen(a)$ $imagen(b)$ $imagen(c)$

Figura 12: imágenes usadas en el experimento

Para poder llevar a cabo esta indagación, hemos implementado la función **qualitative_experiment**, en el archivo ***main***, el cual, tomadas las tres imágenes, las somete al algoritmo implementado en un Tree con Path Compression (debido a su velocidad de ejecución estudiada anteriormente), y se ejecutan variando el valor de **k**. Como se ha explicado en la ***2.1***, y se extiende en [2], el valor **k** afecta directamente la segmentación. Luego, si comenzamos con un valor pequeño, en este caso 500, esperaríamos tener una mayor cantidad de segmentos en las imágenes resultantes. En cambio, al aumentar **k**, tendremos menos segmentación, y con valores muy grandes, los resultados esperados deben ser imágenes con uno o dos segmentos. Hemos establecido iterar hasta 48500, ya que hemos visto en los experimentos anteriores, que estos valores alcanzan para producir la evidencia necesaria. Como hemos usado en secciones anteriores, utilizaremos del script **generateTxtToImage.py** para transformar el output de la función en imágenes de colores, lo cual nos sera de utilidad a la hora de comparar empíricamente los resultados obtenidos.

2.3.1. Imagen A

Como esta imagen ya la hemos analizado en el experimento ***2.2.2***, podemos anticiparnos a los resultados que obtendremos en este caso. Ya hemos visto que con un valor $\mathbf{k} = 13000$ el resultado es muy cercano a lo esperado. Y a su vez, esperaremos una imagen con a lo sumo dos divisiones para el final del ciclo de la función, cuando segmentemos la imagen con un valor de **k** igual a 48500.



$K = 500$

$K = 22500$

$K = 49500$

Afirmativamente, podemos observar que a la mitad del ciclo, hemos obtenido la mejor segmentación de la fotografía, en la cual, el avión se distingue completamente del fondo, y este mismo no muestra pequeñas segmentaciones en comparación con el primer resultado. A su vez, la ultima imagen esta completamente vacía, debido a que el valor de **k** produce una indistinción absoluta entre el objeto y el fondo.

2.3.2. Imagen B

En el segundo caso de estudio, esperamos conseguir dos resultados distintos. Uno, en el cual, al principio del ciclo, la segmentación producida pueda distinguir entre el fondo, la montaña y la roca. Y otro, en el cual, por la poca diferencia de intensidad, la roca y la montaña sean distinguidos como un solo objeto, al avanzar el ciclo.

$K = 6500$

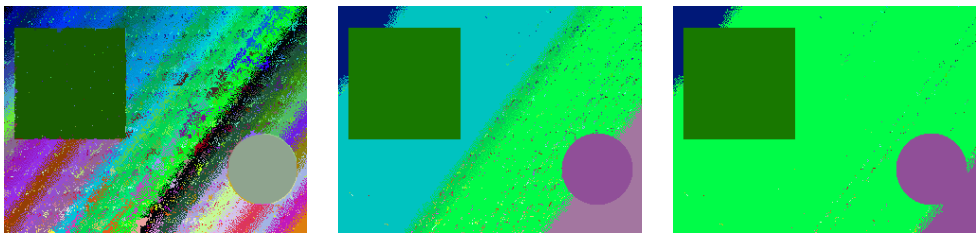
$K = 12000$

A diferencia del caso anterior, hemos logrado segmentar la imagen en sus objetos mas distinguidos con un valor **k** mas chico. Como vemos en la primer imagen, se pueden distinguir la roca, la montaña, el cielo, la nube y el río. Pero la roca y el horizonte se segmentan como uno solo, y al observar mas detenidamente la foto, podemos notar que estas dos áreas son la de mayor intensidad, junto con las nubes. Luego, es esperado que ambos objetos sean identificados como uno en el proceso del algoritmo. También obtenemos una segmentación

mínima muy temprano en el ciclo, lo cual indica que el valor elegido para esta imagen como iteracion maxima es excesivo.

2.3.3. Imagen C

El resultado que se espera en la primera imagen es tener segmentadas las dos figuras, y el fondo fuertemente segmentado debido al valor pequeño de **k**. En un paso intermedio del experimento, esperamos notar un fondo menos segmentado, y al finalizar el ciclo, tener una imagen de color uniforme, como ha sucedido en los casos anteriores.



$K = 500$ $K = 20000$ $K = 48500$

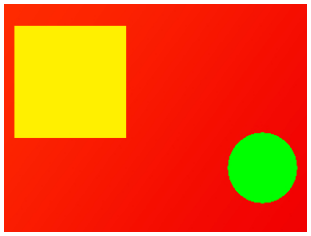
Si bien la hipótesis es correcta sobre los dos primeros resultados, notamos que aun no hemos conseguido la segmentación esperada en la ultima imagen. Como podemos ver, las dos secciones con mayor cambio de intensidad del fondo siguen siendo evaluadas como segmentos distintos en el algoritmo. Debido a este fenómeno, hemos decidido implementar la función ***qualitative_experiment_adhoc***, que simplemente toma esta imagen, y ejecuta el algoritmo sobre ella con un valor de $\mathbf{k} = 75000$, aproximadamente 1,5 el valor máximo del ciclo de la función usada a lo largo de este experimento. El resultado es el siguiente

Finalmente, hemos logrado el resultado esperado. Como podemos observar, el fondo es uniforme y las dos figuras se distinguen perfectamente. En la sección siguiente, analizaremos con mayor profundidad los resultados obtenidos con este experimento.

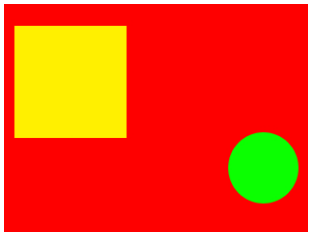
2.3.4. Conclusión

Los resultados del experimento nos demuestran a simple vista que no hay un valor fijo para **k**, en el cual, obtengamos la mejor segmentación para todas las imágenes. Pero por la elección de casos de estudio podemos indagar un poco mas en las características de las imágenes y un valor aproximado en el cual el algoritmo implementado nos devuelva los mejores resultados.

En consecuencia hemos decidido estudiar las siguientes tres instantáneas.



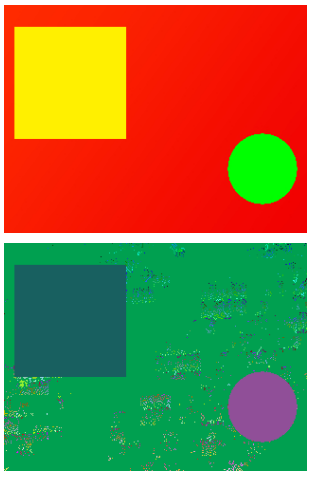
$imagen(d)$



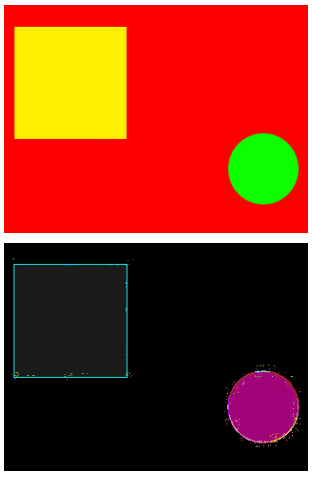
$imagen(e)$

$imagen(f)$

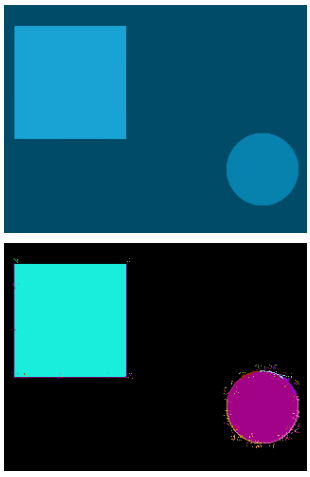
La primera imagen se distingue de la $imagen(c)$ por su fondo. Este presenta menos cambios de intensidad, pero mantiene un degrade oblicuo de naranja a rojo. En este caso, si nuestra hipótesis es correcta, debería llegar a un resultado óptimo con un valor **k** mas chico. La $imagen(d)$ posee un fondo de color solido, lo cual ya nos permitiría observar un resultado óptimo con un valor mucho mas reducido de **k**. Y por ultimo, incluimos la ultima imagen para demostrar que si bien el cambio de intensidades de los objetos es menor, el resultado óptimo se debería obtener con los mismos parámetros que el caso de la imagen segunda. Para esto implementamos la ultima función ***qualitive_experiment_same_image***. En esta, utilizaremos las tres imágenes planteadas anteriormente, y las someteremos a la mismas condiciones del algoritmo que en el primer experimento de la sección ***2.3***, y luego observaremos los distintos resultados obtenidos.



$K = 48500$



$K = 500$



$K = 500$

Finalmente, hemos podido demostrar nuestra hipótesis. Como se puede apreciar en las imágenes de arriba, la primer imagen ha necesitado un valor **k** de casi el %50 menor que la imagen original para alcanzar el resultado esperado. Mientras que las dos siguientes imágenes, demuestran que el cambio de intensidad de los colores con respecto al fondo no afecta el valor necesario para obtener el mismo resultado. Por lo tanto, podemos concluir que necesitamos un valor **k** mayor para imágenes con cambios de intensidades mas leves dentro de sus objetos, mientras que para aquellas con figuras fuertemente distinguidas, el valor necesario es mucho menor.

3. Llenalo con súper

3.1. El problema

Este problema captura una situación mucho más cotidiana que la del problema anterior: dado un mapa con ciudades y rutas queremos calcular cuál es el precio mínimo que hay que pagar para movernos de una ciudad a otra en nuestro auto, sabiendo el consumo de nafta en cada ruta, los distintos costos que tiene el litro de nafta en las diferentes ciudades, y teniendo en cuenta la capacidad limitada del tanque del auto. Es un conflicto que se presenta siempre que hay que desplazarse entre ciudades, por lo que desarrollar algoritmos eficientes que lo resuelvan es relevante.

Definiendo formalmente el problema vamos a contar con n ciudades diferentes, m rutas bidireccionales conectando pares de ciudades distintas a y b (cada una con su consumo l de litros de nafta) y un conjunto de costos c_i con $1 <= i <= n$ representando el costo de cargar un litro de nafta en cada ciudad. Lo que queremos encontrar es, para cada par de ciudades v y w, el costo mínimo que tiene trasladarse de v a w. Para empezar podemos hacer algunas observaciones respecto a estas variables:

- Asumimos que los costos de cargar nafta en cada ciudad c_i serán mayores a 0. No consideramos que esta restricción sea severa ya que estamos interesados en resolver el problema para casos prácticos y relacionados con el mundo real. Luego tiene sentido que la nafta nunca sea gratis (o con precio negativo).
- Por el mismo motivo que lo anterior asumimos que los consumos l_i de cada ruta $1 <= i <= m$ serán mayores o iguales a 0. En particular consideramos que son números enteros positivos. Esta simplificación puede parecer un poco más pesada que las otras pero es fundamental para la resolución del problema, y en términos generales la información relacionada a las longitudes de rutas y al consumo de vehículos no suele ser tan puntillosa como para utilizar números racionales no enteros.
- Tiene sentido considerar que no hay dos rutas que conecten las mismas ciudades, ya que en ese caso podemos quedarnos solamente con la de consumo l_i mínimo e ignorar totalmente la otra. De todas formas podríamos filtrarlas con un algoritmo de preprocesamiento que elimine las rutas de este estilo. Bien implementado se podría hacer de forma lineal.
- Asumimos también que inicialmente el tanque esta vacío. Más adelante mostramos que se puede considerar cualquier cantidad inicial de nafta, pero para simplificar el desarrollo la consideramos cero.
- Asumimos que no hay rutas que vayan de una ciudad a a la misma ciudad a.

Resulta natural visualizar este problema con un grafo con pesos asociados a sus ejes donde las ciudades son los nodos y las rutas son los ejes. Podemos asignarle a cada ciudad un número entre 1 y n. La única dificultad resulta involucrar los costos c_i y el tamaño del tanque del auto (de ahora en más llamado T) al grafo. A modo de ejemplo está el siguiente grafo:

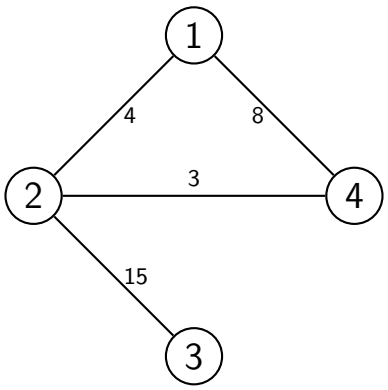


Figura 13: Ejemplo de grafo donde $n = 4$, $m = 4$, y los costos l_i de cada ruta están representados sobre las aristas. Falta agregar las variables c_i y T.

El grafo es no dirigido ya que las rutas son bidireccionales, por lo que un camino de a a b también es un camino de b a a con un mismo consumo asociado. Entonces dado un n, m y la lista de rutas que unen cada ciudad a_i con otra ciudad b_i de consumo l_i con $1 <= i <= m$ se arma el grafo de la siguiente forma:

1. Se numeran las n ciudades de 1 a n.
2. Se crean n vértices.
3. Dos vértices a y b tiene una arista entre ellos si y solo si existe una ruta que una a a y b. El peso asociado a esa arista es el consumo l de esa ruta.

Como dijimos antes este modelo es incompleto al no incluir las variables c_i ni a T al grafo. Pero lo que es peor es que no captura la totalidad de estados del problema. Recordemos que el grafo debería representar en cada vértice un estado posible del auto a través del viaje, mientras que los ejes representan nexos entre estados posibles. La representación anterior considera los estados como 'estar en la ciudad a', cuando en realidad no siempre que estemos en una ciudad estaremos en un mismo estado, pues podríamos tener más o menos nafta al haber llegado allí.

Lo que hicimos entonces es redefinir nuestro grafo: si asumimos que el combustible restante del vehículo es siempre un número entero positivo (o cero) entonces podemos considerar que el nodo de una ciudad a se extiende a $T + 1$ nodos distintos que representan el estado de estar en la ciudad a con un combustible restante t, con $0 <= t <= T$. Ahora podemos considerar a los nodos como un par (a, t) donde a es la ciudad y t el combustible restante. De esta forma representamos la totalidad de estados posibles del auto. Ahora falta redefinir los ejes de nuestro grafo, y lo hicimos de la siguiente forma: el nodo (a, t) tiene una arista hacia (a', t') si y solo si:

- Las ciudades a y a' tienen una ruta que las una.
- El vehículo puede llegar de a a a' y tener t' litros de combustible restantes, habiendo comenzado con t litros, sin cargar en a' y gastando solamente l litros, donde l es el consumo de la ruta del punto anterior. Esto quiere decir que suponiendo que la ruta entre a y a' existe y $T = 60$, $t' = 59$, $t = 0$ y $l = 2$; entonces (a, t) no estará conectado a (a', t'). Esto tiene sentido pues suponiendo el mejor caso en el que el auto llene el tanque en a, entonces partirá con 60 litros y llegará con 58 litros, asi que nunca podrá llegar con 59 litros restantes. Esto impone condiciones a la nafta máxima que el auto puede tener tras un cambio de estado, y también a la mínima (pensando el caso en que $T = 60$, $t' = 1$, $t = 58$ y $l = 2$).

Así definidos los ejes se puede pensar que cuando llego a un estado estoy 'en la puerta de la ciudad', por lo que mi nafta actual es la que tenía antes (t) más la que cargué (llamémosla x) menos la que gasté para llegar hasta la nueva ciudad (l). Luego es fácil ver que

$$x = t' + l - t$$

y que todas las condiciones que debe cumplir un eje para ser válido son

$$x >= 0$$
$$x + t <= T$$

Si $x < 0$ entonces el vehículo debe gastar más de l litros para llegar al nuevo estado, lo que no es óptimo pues estaría desperdiciando nafta (le sería más conveniente pasar a un estado con mayor nafta restante), asi que descartamos ese eje. Si $x + t > T$ entonces el auto va a partir de la ciudad a con más nafta que la que puede portar su tanque, lo que es absurdo.

Entonces nuestro grafo modela los cambios de estados de un vehículo donde los vértices son pares (a, t) con a una ciudad y t la nafta del auto cuando llega a la ciudad, y en el cual dados dos vértices v y w estos tienen una arista que va de v a w si el auto puede cargar la suficiente nafta en la ciudad de v (sumada a la que ya tenía) para poder llegar hasta la puerta de la ciudad de w y tener la cantidad de combustible correspondiente a w. Nótese que ahora tenemos un digrafo.

Con las definiciones anteriores la función de costo para los ejes ya queda determinada, pues para transitar entre dos ciudades conectadas hay que cargar x litros de nafta en la ciudad origen. Luego se define:

$$c(vw) = xc_v$$

donde x está definido en las ecuaciones anteriores y c_v es el costo por litro de nafta en la ciudad de v. Como $x >= 0$ y $c_i > 0$ los pesos asociados a los ejes son todos mayores o iguales a cero.

Ahora ya logramos reducir el problema a encontrar los caminos mínimos en un digrafo con pesos asociados a sus ejes, los cuales en particular son positivos. Este digrafo tendrá como vértices a todos los pares (a, t) de ciudades y cantidades enteras de nafta posibles en el tanque. Por ende habrá $n(T + 1)$ nodos. El cálculo de la cantidad de aristas es un poco más difícil.

Sea v un vértice antes de agregar los vértices duplicados. Llamemos $d(v)$ al grado de v en ese grafo G. Cuando expandamos la cantidad de vértices agregando los distintos estados de la nafta la cantidad de vecinos de v será

$$d'(v) <= (T + 1)d(v)$$

pues como mucho lo uniremos a todas las réplicas de cada uno de sus vecinos (en particular se puede conseguir una cota más fina, pero no nos interesa en nuestro caso). Para cada vértice que creamos copiando esa ciudad (pero con una cantidad de nafta distinta) va a valer la misma desigualdad en el nuevo grafo G'. Con esto ya podemos encontrar una cota para la cantidad de arista m' de G':

$$2m' = \sum_{w \in V(G')} d(w) = \sum_{i=1}^{n} \sum_{w \in R_i} d(w) <= \sum_{i=1}^{n} \sum_{w \in R_i} (T + 1)d(v_i) =$$

$$= \sum_{i=1}^{n} (T + 1)^2 d(v_i) = (T + 1)^2 2m$$

Hicimos una partición de los vértices de G' en conjuntos disjuntos R_i con $1 <= i <= n$ donde R_i contiene a todos los vértices que surgen de replicar el vértice que representa a la ciudad i. Usando algunas propiedades básicas de grafos ya se puede obtener la desigualdad anterior.

Nos interesa resolver el caso particular en que $T = 60$, por lo que las diferencias en tamaños entre ambos grafos se puede definir de la siguiente forma:

$$n' = O(n) \text{ y } m' = O(m)$$

Ahora que tenemos la caracterización del grafo podemos aplicar algoritmos conocidos a nuestro problema y analizar su complejidad. Si queremos encontrar el costo mínimo entre dos ciudades a y b comenzando con 0 litros de nafta tenemos que ver cual es el costo del camino de menor costo entre $(a, 0)$ y $(b, 0)$. El vértice de origen es el de $(a,0)$ pues es el estado en el que comenzamos, mientras que el vértice de destino es $(b, 0)$ ya que queremos llegar a la ciudad b, y el estado menos costoso al que llegar será siempre el que tenga una cantidad de nafta nula. Miremos con más cuidado esto último.

Supongamos que tenemos el costo del camino de costo mínimo entre $(a, 0)$ y (b, t') con $t' \neq 0$. Si en el vértice anterior a (b, t') el auto cargó nafta entonces podríamos hacer un camino en el que cargue menos y

llegue a (b, t') de todas formas (pues le sobraron t' litros). Si no cargó en ese nodo puedo fijarme en el anterior y así sucesivamente. Seguro voy a llegar a algún estado en el que el auto haya cargado nafta ya que comenzó con cero y finalizó con más de cero, y como los costos de nafta son siempre positivos puedo armar un camino con menor costo sacando los t' litros que cargó de más y llegar a (b, 0) pagando menos. Nótese que esta última afirmación depende de iniciar con cero. Si quisiéramos iniciar en otro valor podríamos revisar todos los costos de los caminos hasta (b, t') para cualquier t' y quedarnos con el menor.

Analicemos cuanto nos cuesta construir este nuevo grafo. Inicialmente nos dan una lista de n ciudades, las m aristas y los n costos asociados a cada ciudad. Para armar el grafo con lista de adyacencias, conservando junto a cada arista su costo, se hace lo siguiente:

- Inicializamos un vector con $60n$ posiciones, donde la posición p representa a la ciudad **p mod n** con una cantidad de combustible asociada de $\frac{p}{n}$ litros.
- Para cada arista (a, b, l) que conecta la ciudad a y b con un consumo l vemos cuáles vértices (a, t) están conectados con algunos (b, t'). Para eso por cada eje que corresponda a la ciudad de origen a y de destino b calculamos x y vemos si cumple las condiciones antes declaradas. Si las cumple lo agregamos al grafo junto a su costo. En total hay que revisar $61 * 61 * m = 3721m$ pares. También hacemos la inversa para armar ejes entre vértices que representan a los caminos de b hacia a.

Esta construcción puede generalizarse para situaciones en las que el tamaño del tanque de combustible es una variable, pero afecta al cálculo de la complejidad. Escrito en pseudocódigo tenemos el siguiente procedimiento:

Algorithm 4

```
1:  procedure BUILDGRAPH(n,M,C)
2:      G ← NewGraph(60n)                          ▷ Inicializa grafo de 60n vértices
3:      for i=0...M-1 do
4:          (a, b, l) ← M[i]
5:          for j=0...60 do
6:              for k=0...60 do
7:                  t ← k
8:                  t' ← j
9:                  x ← t' + l − t
10:                 if x >= 0 and x + t <= 60 then
11:                     p ← ((b, t'), x * c_a)
12:                     Edges((a, t)).Add(p)
13:                 swap(t, t')
14:                 x ← t' + l − t
15:                 if x >= 0 and x + t <= 60 then
16:                     p ← ((a, t'), x * c_b)
17:                     Edges((b, t)).Add(p)
18:     return G
```

El input es la cantidad de ciudades n, un vector con las aristas M y un vector con los costos C. Esta algoritmo tiene una complejidad temporal de $O(n + m)$ pues los ciclos internos tienen todos un tiempo de ejecución constante y G tiene un tamaño múltiplo de n por 60. Luego el grafo se arma en tiempo lineal y esta complejidad será despreciable en relación al resto de los procedimientos para calcular los caminos.

En este contexto podemos utilizar algoritmos conocidos para calcular los caminos de costo mínimo. En la siguiente sección enumeraremos los que implementamos y analizamos las diferencias de rendimiento. En todos los casos usamos las versiones clásicas de los algoritmos, pues gracias a la forma en que logramos modelar nuestro problema podemos utilizarlos sin mayores modificaciones. Gracias a esto sabemos que la correctitud viene dada, siempre y cuando el camino de costo mínimo sobre este grafo que pasamos como input sea lo que estamos buscando. Veamos esto último.

El camino que realiza el vehículo hasta un destino consiste en una secuencia de ciudades, y en cada ciudad carga una cantidad de combustible (si no carga nada podemos considerar que carga cero). Para calcular el costo total del recorrido se debe multiplicar cada litro cargado por el precio que tomó cargarlo, el cual depende de la ciudad en que se está en el momento de la carga. Luego supongamos que existe un camino con costo mínimo que va de la ciudad v_0 a v_n de la forma:

$$v_1, v_2, v_3, ..., v_n$$

con un conjunto de valores de nafta cargados de la forma:

$$l_1, l_2, l_3, ..., l_n$$

Esto quiere decir que en la ciudad v_i el vehículo cargó l_i litros de nafta a un costo de c_i x l_i. El modelo que representa el grafo que construimos con la función *BuildGraph* se corresponde con estos datos. Veamos que este camino existe en el grafo y que su costo es el mismo. Para esto analicemos el eje entre una ciudad v_i y v_{i+1}:

Como existe una ruta entre v_i y v_{i+1} sabemos que se entra al segundo *for* del algoritmo *BuildGraph*. Para llegar hasta el vértice v_{i+1} el vehículo cargó l_i litros de nafta en v_i a los que ya tenía antes (llamemos t a esa cantidad). Luego el costo del traslado fue de c_i x l_i, y al llegar a v_{i+1} el auto cuenta con t' litros, lo cual equivale a $t + l_i - d$ donde d es lo que gastó para realizar el traslado (la distancia entre las ciudades). Como este movimiento corresponde a un camino real sabemos que se respetan las restricciones razonables (que el tanque de nafta nunca supera su capacidad máxima y que su contenido de combustible nunca es menor que cero). Por ende sabemos que se cumplen las condiciones necesarias para que se cree el eje entre el estado (v_i, t) y (v_{i+1}, t') con un costo asociado de l_i (en el pseudocódigo l_i es el valor x que se calcula viendo las diferencias entre la nafta final, la inicial y el costo de recorrer la ruta en cada desplazamiento). Entonces el camino v_i, v_{i+1} con esos valores de combustible asociados es equivalente al eje (v_i, t),(v_{i+1}, t') en el grafo. Podemos reconstruir todo el camino $v_1, v_2, ..., v_n$ en el grafo como:

$$(v_1, 0),(v_2, t_2),...,(v_n, t_n)$$

donde t_i corresponde a la nafta restante al llegar a v_i, la cual se puede calcular recursivamente usando la distancia entre la ciudad anterior y la cantidad de nafta cargada l_{i-1} (y obsérvese que t_1 es cero pues todos nuestros camino inician en cero). Con esto probamos que todo camino sobre el 'mapa' se corresponde con uno de mismo costo en el grafo al cual podemos darle un sentido para que se relacione con el camino real. Ahora inversamente veamos que todo camino sobre el grafo se corresponde con uno del mismo costo en el mapa. Tomemos un camino de costo mínimos sobre el grafo de la forma:

$$(v_1, 0),(v_2, t_1),...,(v_n, t_n)$$

Recordemos que t_n vale cero por lo que se dijo unos párrafos atrás. Queremos encontrar una secuencia de ciudades y una secuencia de valores de litros de nafta cargados que se corresponda con este camino. La secuencia de ciudades será $v_1, v_2, ..., v_n$ y la de litros será $l_1, l_2, ..., l_n$ donde $l_i = t_{i+1} - t_i + d(v_i, v_{i+1})$ con d la distancia entre dos ciudades. Esta última fórmula es idéntica a la que se usa para asignarle pesos a los ejes en el grafo, por lo que sabemos que el costo de este camino sobre el grafo es la sumatoria de c_i x l_i. Por otro lado el camino real $v_1, ..., v_n$ también tendrá ese mismo costo, pues es el valor asociado a la nafta que se carga en cada ciudad. Solo falta ver que ese camino existe efectivamente en el 'mapa'.

Sabemos que los ejes v_i, v_{i+1} existen pues de otra forma no estaría en el grafo el eje (v_i, t_i),(v_{i+1}, t_{i+1}). También podemos ver que la cantidad de litros que carga en cada ciudad no genera que se supere la capacidad del tanque, pues en ese caso el eje no hubiera sido creado entre esos dos ejes (es una de las condiciones que se revisa en el pseudocódigo antes de generar un eje). De la misma forma sabemos que con esa cantidad cargada el tanque no tendrá menos de cero litros en ningún momento, pues en ese caso el eje tampoco existiría en el grafo (como antes, es una de las condiciones que se revisa en la construcción del grafo). Luego probamos que un camino del grafo está asociado a un camino en el 'mapa' con un mismo costo.

Con esto probamos que el modelo representa correctamente los caminos que puede tomar el vehículo, y junto a la correctitud ya conocida de los algoritmos de camino mínimo probamos la correctitud general del algoritmo. Ahora procedemos a la experimentación.

3.2. Experimentación

3.2.1. Observaciones e hipótesis

Se utilizaron 4 algoritmos distintos para calcular los caminos entre todas las ciudades:

- Algoritmo de *Dijkstra* con cola de prioridad.
- Algoritmo de *Dijkstra* con cola *FIFO*.
- Algoritmo de *Bellman-Ford*.
- Algoritmo de *Floyd-Warshall*.

Los primero 3 algoritmos solo calculan los caminos desde un vértice a todos los demás, por lo que para calcular los caminos entre todos los vértices habrá que hacer n ejecuciones de los mismos (siendo n la cantidad de ciudades que recibimos de input). Podemos observar que estos algoritmos tienen complejidades diferentes, y por ende esperamos que dependiendo del escenario algunos sean más eficientes que otros. Uno de nuestros objetivos es lograr caracterizar cada uno de estos tipos de escenarios.

Las complejidades temporales de dichos algoritmos son respectivamente: $O(n.m.log(n))$, $O(n^2m)$, $O(n^2m)$ y $O(n^3)$. Ya se ve que las complejidades no son solo distintas, sino que incluso hay una (la del algoritmo de *Floyd*) que ni siquiera depende de la cantidad de aristas m del grafo. Entonces uno esperaría que el algoritmo de *Floyd* sea más eficiente en grafos densos ($m = \Omega(n^2)$).

Resumiendo, las principales incógnitas que nos gustaría discernir mediante la experimentación son:

- Que las complejidades teóricas se ajusten a los resultados observacionales, variando los valores de n y m para estudiar las diferencias en tiempos de ejecución.
- Definir cuál es el tipo de grafo óptimo para cada algoritmo, distinguiendo los grafos entre ralos, densos y generales (o aleatorios).
- Analizar si alguno de los algoritmos tiene un desempeño mucho mejor que el resto más allá de su complejidad teórica.

Antes de comenzar la experimentación hicimos algunas hipótesis en cuando a las incógnitas que acabamos de presentar. Por un lado suponemos que la complejidad teórica de estos algoritmos estará correlacionada con los resultamos que midamos, pues los cálculos de las mismas no son complejos y están muy estudiados al ser los mismos ampliamente utilizados. Guiándonos por las complejidades anteriores el algoritmo de *Dijkstra* con cola de prioridad debería ser el que tenga un mejor desempeño en la mayoría de los casos, si bien cuando los grafos son densos la complejidad de *Floyd* es mejor. Hay que remarcar que por la lógica del algoritmo para resolver el problema el grafo sobre el que se aplican estos algoritmos de camino mínimo siempre tiene al menos unos 60 vértices. Con esto queremos decir que siempre se trabaja con grafos grandes, lo que va a limitar el tamaño de la experimentación.

3.2.2. Casos de test

Antes de mostrar la experimentación haremos algunas aclaraciones:

- Los valores de n fueron elegidos mayores a 2 al ser los grafos con $n = 1$ y $n = 2$ casos trivial.
- Los valores máximos de n están limitados de acuerdo a la capacidad de procesamiento del ordenador.
- Cada evaluación se ejecuta varias veces y se promedian los resultados para minimizar el impacto de interrupciones del SO. Como en el punto anterior la cantidad de repeticiones está ligada a la capacidad de procesamiento del ordenador.
- Los valores de l de las rutas están entre 0 y 60 y fueron generados por una distribución uniforme. No se eligen rutas con valores mayores a 60 pues no serían atravesables por el vehículo.
- Los precios de la nafta son entre 0 y 60 generados por una distribución uniforme para intentar obtener distintas instancias.
- Utilizamos el mismo ordenador que usamos para la experimentación de segmentación.

Primero vamos a presentar las experimentaciones comparativas entre los 4 algoritmos y luego los resultados relacionados con la correlación y complejidad. Los casos de comparación que establecimos son:

- **Caso random**: Generamos 10 instancias con $n = 3$ y $n = 6$ eligiendo m al azar con $n - 1 \leq m \leq \frac{n(n-1)}{2}$ (utilizando una distribución uniforme). Cada uno de estos casos fue ejecutado 25 veces.
- **Caso ralo**: Hicimos tests con cada n tal que $3 \leq n \leq 10$ y con $m = n$ (para tener $m = O(n)$ y ejecutamos 10 veces cada uno.
- **Caso completo**: Hicimos tests para cada n $3 \leq n \leq 10$ y $m = \frac{n.(n-1)}{2}$. Ejecutamos 10 veces cada uno..

3.2.3. Análisis de resultados

Caso random

Antes de comenzar el testeo aleatorio experimentamos con valores fijos de m para obtener una idea de los algoritmos y su *performance*:

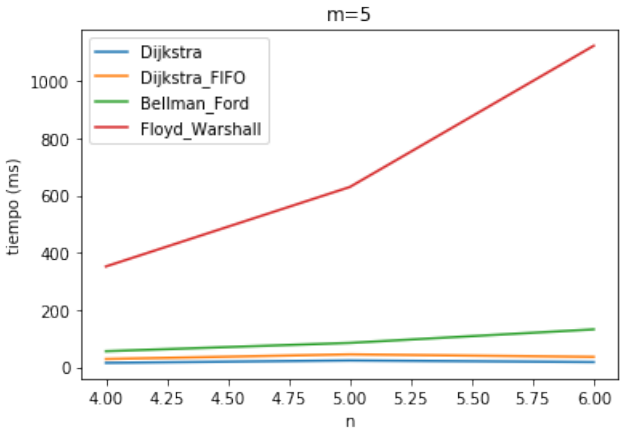


Figura 14: Caso random, tiempo/n

El tiempo de ejecución de *Floyd* aumentó notablemente a medida que aumentaba el n, lo cual es razonable pues su complejidad depende del cubo de n. Los otros algoritmos no se vieron tan afectados. Como uno esperaría, con valores de m pequeños los otros algoritmos tienen un desempeño mucho mejor que el de *Floyd*. Para discernir con facilidad las diferencias entre *Dijkstra* y *Dijkstra* FIFO tenemos el siguiente gráfico.

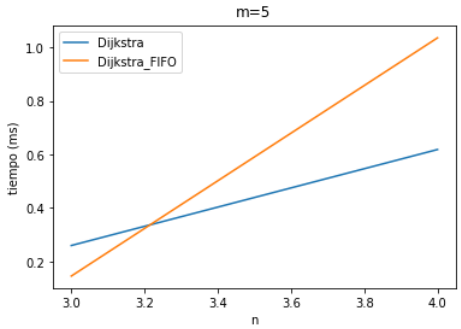


Figura 15: Caso random, tiempo/n

Dijkstra con la cola FIFO depende más de n que el *Dijkstra* con cola de prioridad, por lo que es razonable lo que vemos en el gráfico: los tiempos de ejecución de la implementación con la cola FIFO se vuelve mayores a los de *Dijkstra* con cola de prioridad a medida que aumenta el valor de n. En general parece que *Dijkstra* con cola de prioridad es el más eficiente de los 4. Ahora veamos qué pasa cuando fijamos n en 6 y tomamos valores de m aleatorios:

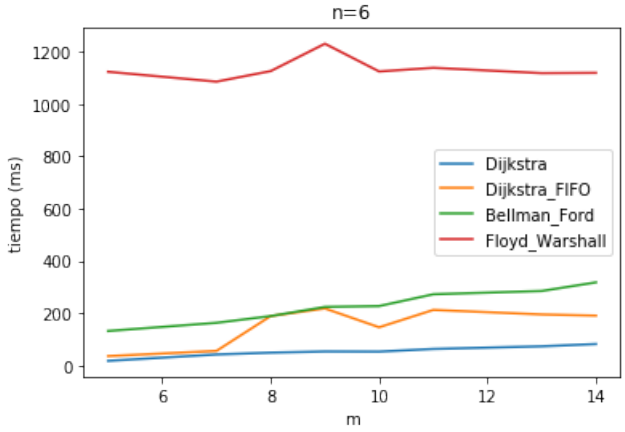


Figura 16: Caso random, tiempo/m

Los resultados son similares a los anteriores, pero ahora está claro que *Dijkstra* con cola de prioridad es mucho más eficiente que el resto. Por último hicimos un gráfico de barras usando el total de tiempo que tardaron en ejecutarse todas las instancias por cada algoritmo.

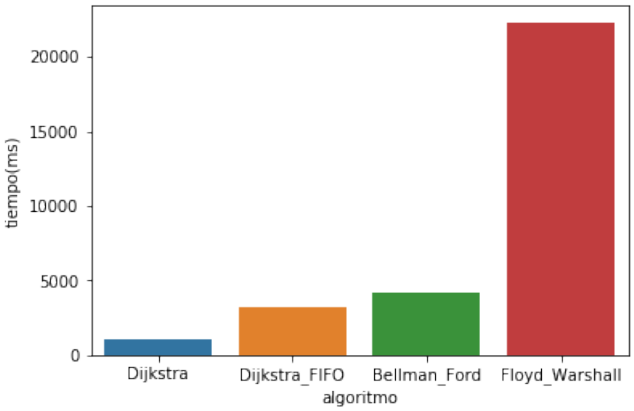


Figura 17: Caso random, tiempo total por algoritmo

Con estos datos podemos concluir en grafos aleatorios que la *performance* de los algoritmos ordenados del mejor al peor es: *Dijkstra*, *Dijkstra* con cola FIFO, *Bellman−Ford* y *Floyd-Warshall*.

Caso ralo

Como antes tenemos un gráfico de líneas para tener una primera idea de los tiempos de ejecución de los algoritmos:

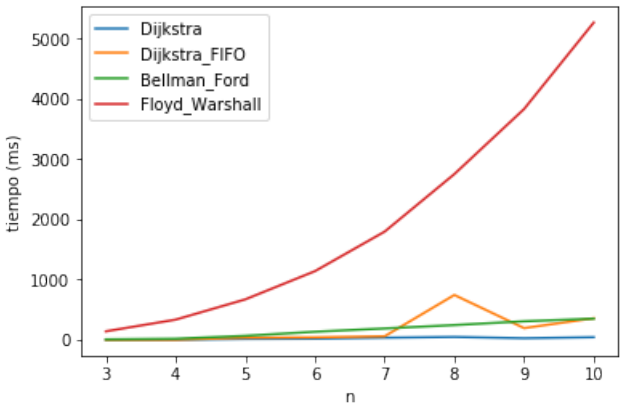


Figura 18: Caso ralo, tiempo/n

Se puede ver que *Floyd−warshall* es el algoritmo más lento y que más escala en este caso para todas las instancias. Esto tiene sentido teniendo en cuenta que los valores de m son siempre pequeños y constantes. Pero es más difícil discernir entre los demás. En el siguiente gráfico se los puede ver con más detalle:

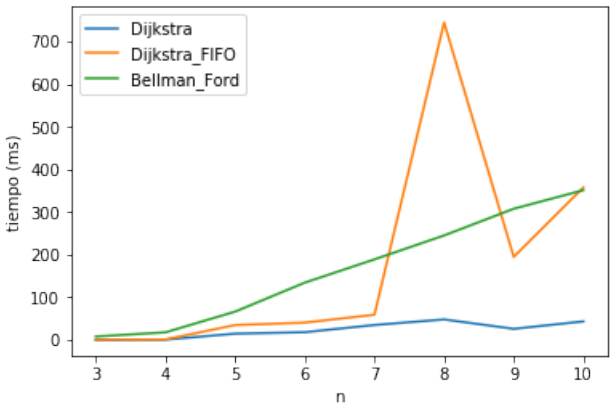


Figura 19: Caso ralo, tiempo/n sin Floyd

En el gráfico anterior se ve que *Dijkstra* es el más rápido entre los 3, y que todos crecen en una proporción similar (esto se corresponde con las cotas teóricas). Hubo un *outlier* en el desempeño de *Dijkstra* con FIFO pero puede ser ignorados pues es solo un caso. En general *Dijkstra* con FIFO tiene una *performance* similar a *Bellman−Ford*, siendo un poco más eficiente. Por último podemos ver las diferencias en el gráfico de barras:

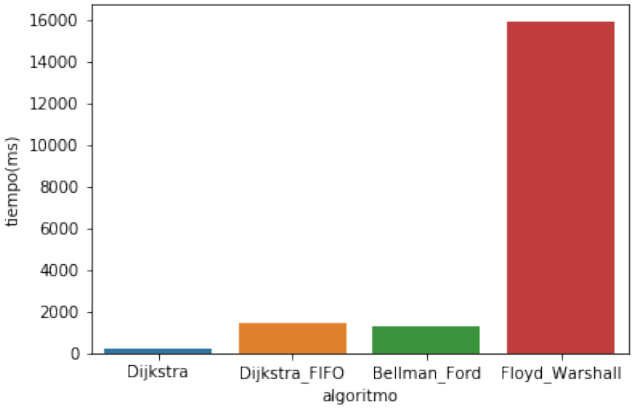


Figura 20: Caso ralo, tiempo total por algoritmo

Dijkstra con FIFO parece más lento que *Bellman−Ford*, pero esto solo es debido a el *outlier*. El orden de eficiencia similar al anterior, y se ve claramente que en los casos ralos el algoritmo de *Floyd* es poco recomendable.

Caso completo

Como antes, comenzamos con una mirada rápida a todos los algoritmos:

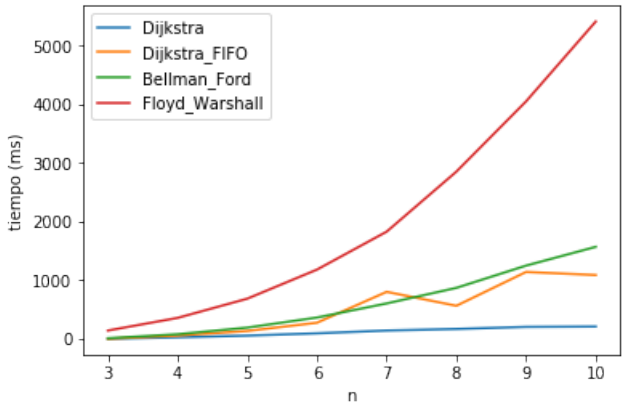


Figura 21: Caso completo, tiempo/n

Ahora el algoritmo de *Floyd* no está tan alejado del resto, si bien sigue siendo lento en comparación. También podemos ver que como antes el algoritmo de *Dijkstra* con cola de prioridad es el más eficiente, y los otros tienen un desempeño similar.

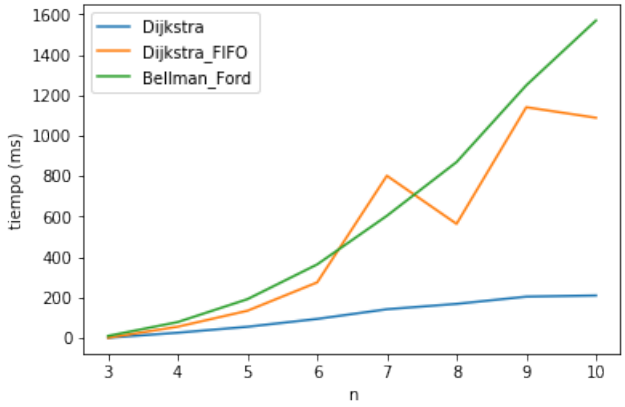


Figura 22: Caso completo, tiempo/n sin Floyd

Ahora podemos distinguir entre los dos algoritmos y ver que *Bellman−Ford* es un poco más lento. En un grafo de barras tenemos lo siguiente:

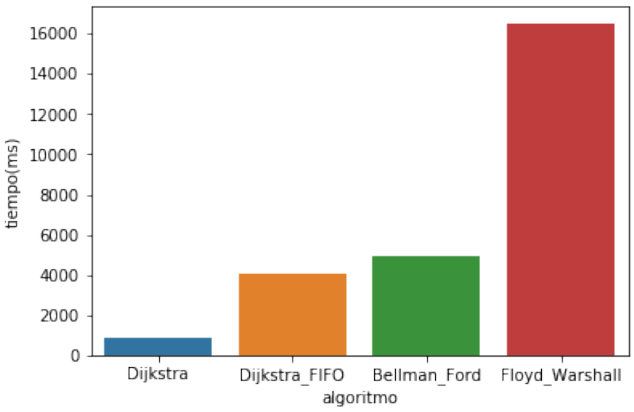


Figura 23: Caso completo, tiempo total por algoritmo

Se puede ver claramente en el gráfico que para esta instancia el orden en cuanto a tiempo de ejecución es (de menor a mayor): *Dijkstra*, *Dijkstra* FIFO, *Bellman−Ford* y por último *Floyd− Warshall*, igual que antes.

3.2.4. Complejidad y correlacion

En esta sección se utilizan los casos de tests anteriores para determinar la correlación entre los tiempos de ejecución experimentales y la cota teórica correspondiente a cada algoritmo.

Dijkstra con cola de prioridad

Considerando los casos anteriores armamos el siguiente gráfico:

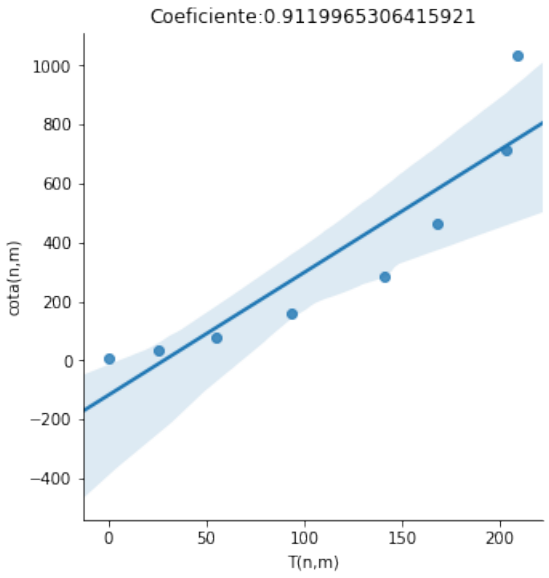


Figura 24: Correlación caso completo Dijkstra

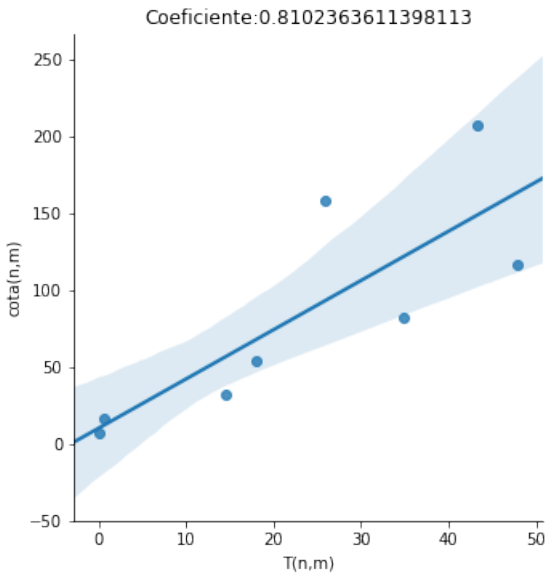


Figura 25: Correlación caso ralo Dijkstra

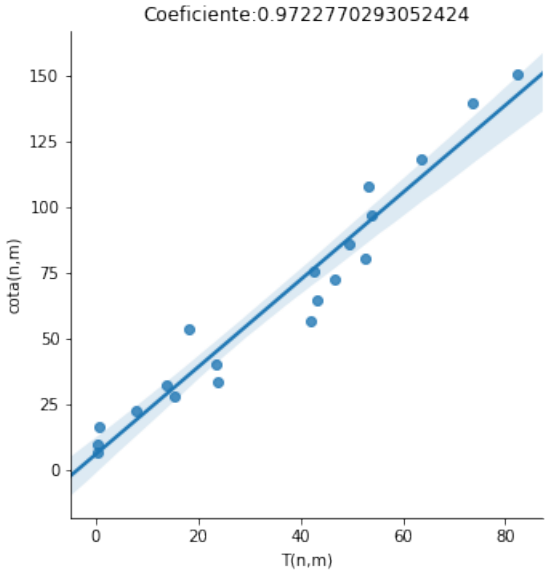


Figura 26: Correlación caso random Dijkstra

En términos generales el coeficiente de *Pearson* tiene un valor elevado en los 3 casos (siempre está por arriba del 0.8) y hay una correlación evidente es el caso aleatorio. Podemos entender los valores más bajos en los otros tests teniendo en cuenta que la complejidad teórica es del peor caso, por lo que puede que en los tests ralos y densos haya habido casos óptimos en los que el tiempo de ejecución haya sido menor. De todas formas concluimos que la cota teórica se corresponde con lo observado.

Dijkstra sin cola de prioridad

A continuación están los resultados de la correlación para los distintos casos:

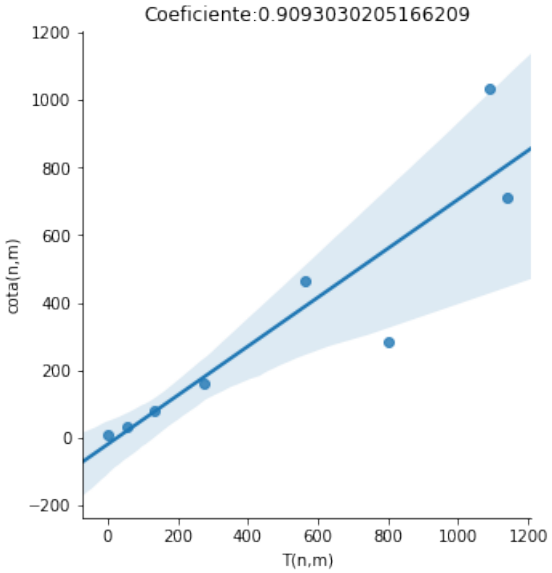


Figura 27: Correlación caso completo Dijkstra FIFO

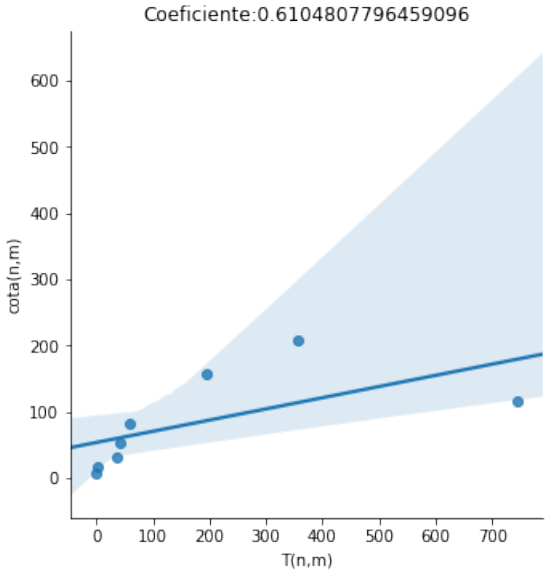


Figura 28: Correlación caso ralo Dijkstra FIFO

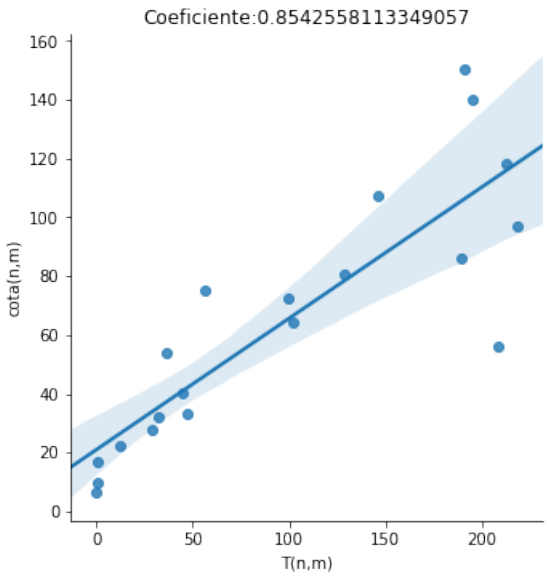


Figura 29: Correlación caso random Dijkstra FIFO

En los casos aleatorios y densos el coeficiente de *Pearson* es mayor a 0.9, por lo que la correlación es efectiva. Por otro lado, en el caso ralo la correlación no se cumplió. Eso se debe a que la implementación sobre FIFO puede terminar mucho antes que el peor caso. El peor caso se alcanza cuando el algoritmo tiene que pisar varias veces las mismas aristas debido a que encuentra una y otra vez caminos menos costosos. Cada vez que pisa una arista agrega nuevas aristas a la cola, pero eventualmente se detiene ya que las nuevas aristas serán por lo menos más costosas que la recién agregada. En los casos ralos, donde hay pocas aristas, podría pasar que el algoritmo terminara en pocas ejecuciones debido a que no hay necesidad de pisar nuevas aristas ni de rellenar la cola FIFO. Por ende es esperable que los resultados no se ajusten del todo a la cota teórica. En los otros test al haber más aristas estos casos son menos frecuentes.

Concluimos entonces que la cota de complejidad teórica está correlacionada, si bien hay muchos casos en los que esta cota no es fina.

Bellman−Ford

A continuación están los gráficos de correlación para los distintos casos:

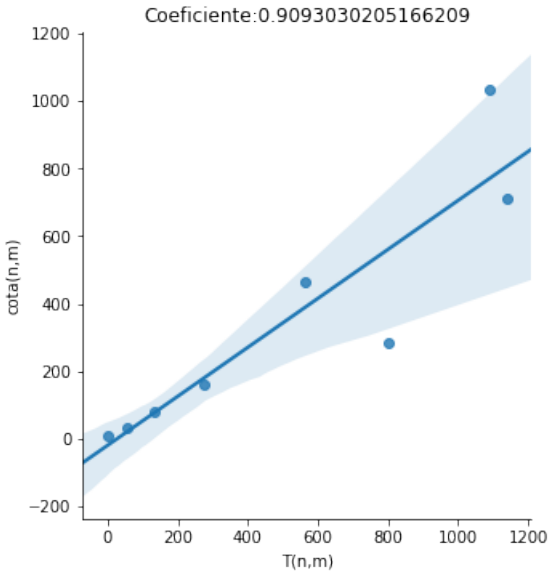


Figura 30: Correlación caso completo Bellman−Ford

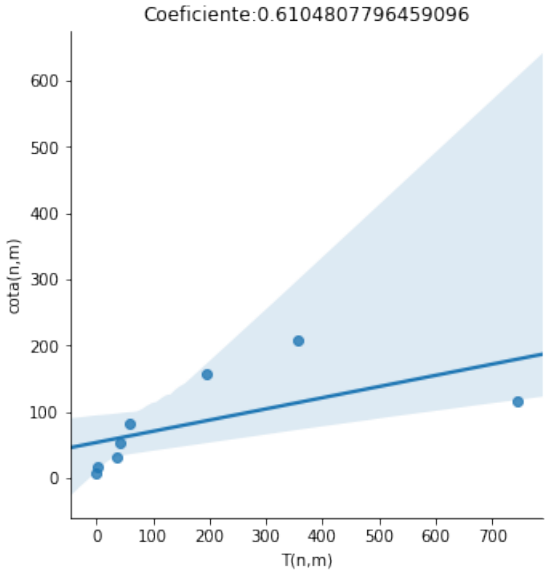


Figura 31: Correlación caso ralo Bellman−Ford

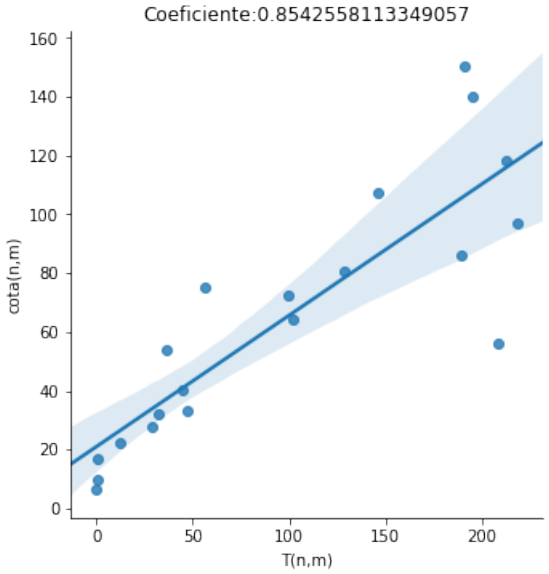


Figura 32: Correlación caso random Bellman−Ford

Como antes los resultados fueron buenos en los testeos random y completos (el coeficiente de *Pearson* tiene un valor alto) pero en los casos ralos esta no está tan clara. Esto se puede entender con el mismo argumento que antes, pues el algoritmo de *Dijkstra* con cola FIFO es más bien una versión optimizada del algoritmo de *Bellman−Ford*. Por ende en los casos ralos el algoritmo no tendrá el desempeño esperado. En particular en el caso de *Bellman−Ford* el algoritmo podría ejecutar muchas más iteraciones de las necesarias si la cantidad de aristas es muy pequeña. En particular esto se nota mucho cuando el grafo es disconexo.

Concluímos como en el algoritmo anterior que la cota teórica es correcta pero no tan fina.

Floyd−Warshall

A continuación están los gráficos de complejidad para los distintos casos, y además hicimos un test para verificar que la complejidad no depende en absoluto de m. Los resultados son los siguientes.

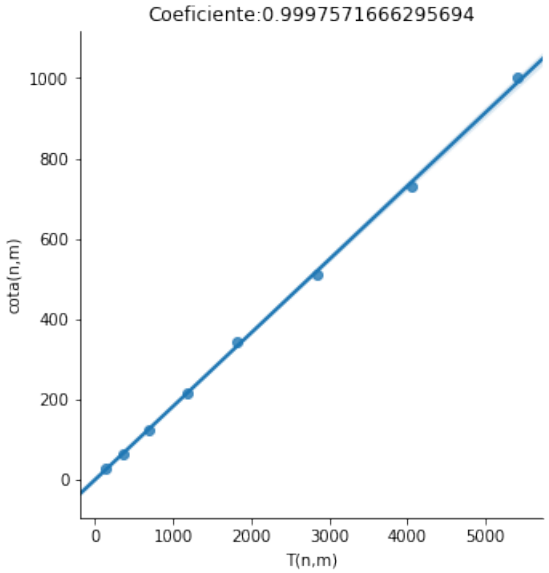


Figura 33: Correlación caso completo Floyd−Warshall

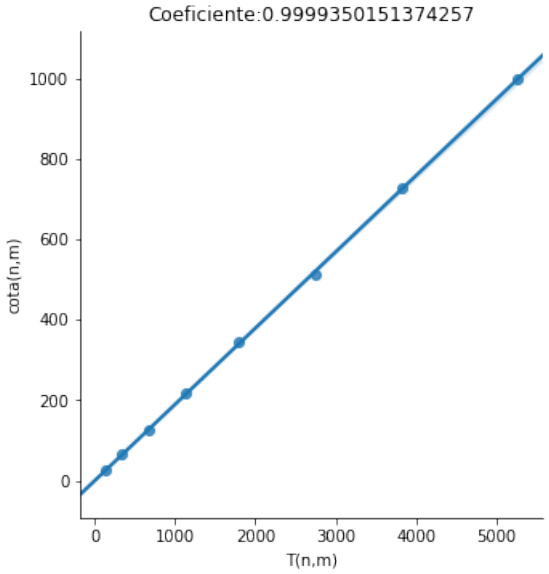


Figura 34: Correlación caso ralo Floyd−Warshall

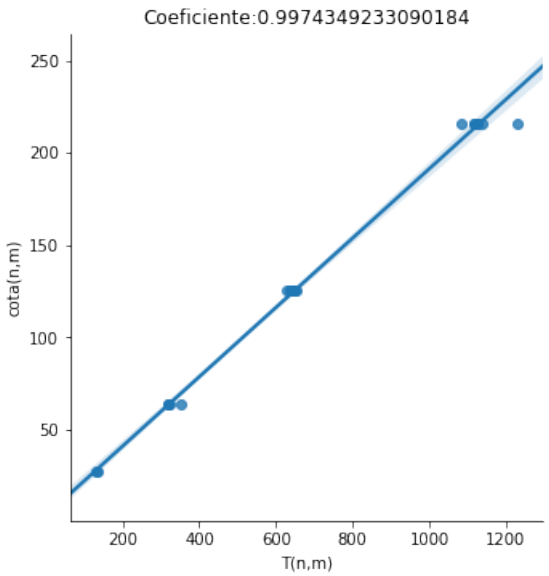


Figura 35: Correlación caso random Floyd−Warshall

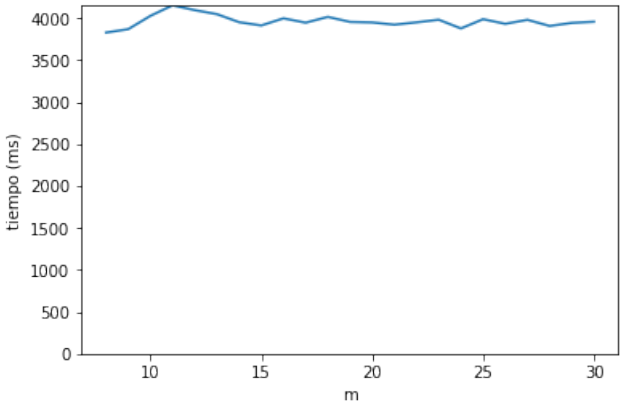


Figura 36: Floyd tiempo/m en un caso de N fijo

Con estos gráficos es evidente que la complejidad del algoritmo de *Floyd* es $O(n^3)$. Como la implementación de este algoritmo depende de 3 *for* anidados con n valores es esperable que la cota sea fina, lo que se ve reflejado en el coeficiente de *Pearson* cercano a 1. En el último gráfico se muestra que el valor de m no afecta al algoritmo.

Con esto conluímos las experimentaciones.

3.2.5. Conclusiones de la experimentación

Las conclusiones respecto a la complejidad ya fueron realizadas en la sección anterior, por lo que ahora enunciaremos algunas conclusiones respecto a las comparaciones de *performance*:

- En los casos probados el algoritmo más rápido fue la implementación de *Dijkstra* con cola de prioridad, y el más lento fue la implementación de *Floyd− Warshall*. Se podrían hacer más experimentaciones entre la implementación de *Dijkstra* con FIFO y la de *Bellman−Ford* para lograr diferencias con mayor precisión cuál es más eficiente.

- Hay una diferencia significativa a la hora de ejecutar estos algoritmos ya que en todos los casos probados *Dijkstra* superó ampliamente a las otras cuatro, con *Floyd− Warshall* siendo particularmente lento, aunque esto podría cambiar con valores de m mucho más elevado. En particular vale afirma que si bien *Dijkstra* es el más rápido entre los 4, solo puede aplicarse a grafos donde los pesos de los eje son positivos, mientras

que *Bellman−Ford* y *Floyd* pueden aplicarse en una mayor variedad de casos. Esta muestra el *Trade off* que hay entre la eficiencia de un algoritmo y su alcance para resolver problemas diferentes.

4. Conclusiones

Como conclusión queremos remarcar la importancia de desarrollar distintos algoritmos para un mismo problema. Esto se nota particularmente en el problema de camino mínimos, donde algunos algoritmos dependen de variables que otros no. De esta forma uno puede ajustar el algoritmo a utilizar de acuerdo a las características particulares del problema al que se enfrenta.

En cuanto a los algoritmos de segmentación rescatamos la importancia del análisis cualitativo y extensivo de los resultados que se obtienen con los algoritmos. De esta forma se pueden encontrar los parámetros necesarios (en nuestro caso el k) para que los procedimientos obtenga el resultado más óptimo posible (en este caso una segmentación lo más próxima a la interpretación humana).

Referencias

[1] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient graph-based image segmentation. *International journal of computer vision*, 59(2):167–181, 2004.

[2] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.