Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Segmentación basada en grafos

Algoritmo de Felzenszwalb

Alumno: Daniel Soto
Profesores: José M. Saavedra
Mauricio Cerda Villablanca
Ayudante: Andrés Ferrada L.
Fecha de entrega: 26 de agosto de 2018
Santiago, Chile

Resumen

En este paper, se detalla una implementación del algoritmo de Felzenszwalb de segmentación de imágenes basada en grafos. Luego se discuten los resultados y se proponen posibles mejoras a la implementación realizada.

1. Introducción

El algoritmo de Felzenszwalb se basa en interpretar una imagen como un grafo no direccionado con pesos. Cada pixel de la imagen se traduce en un vértice del grafo, y existen aristas entre todos los pixeles vecinos (puede ser en el sentido 4-conectado o 8-conectado, en este paper se utilizó el caso 8-conectado.). Luego se utilizan métricas de diferencia interna dentro de un cluster de vértices y de diferencia interna mínima entre clusters para tener una condición de mezcla de clusters. Iterando sobre las aristas del grafo se evalúa esta condición y se genera una segmentación de la imagen dada.

2. Desarrollo

La implementación de este algoritmo se realizó mediante 3 clases, una que contiene información sobre la imagen y que realiza en algoritmo en sí, y una clase para representar un vértice, soportando operaciones de la estructura de datos Union-Find, y una otra para representar una arista en el grafo.

La clase que representa un vértice, `Vertex`, define los siguientes campos y métodos:

- Campos:
  - `coordinates`: Coordenadas del vértice en la imágen. Usado para verificar la igualdad entre dos vértices.
  - `parent`: Define el padre del nodo en la estructura Union-Find.
- Métodos:
  - `find`: Realiza la búsqueda de la raíz del cluster de la estructura Union-Find al que pertenece el nodo. Se realiza además compresión de camino, para que las próximas búsquedas desde ese nodo y sus ancestros tomen tiempo constante.
  - `unite`: Recibe como parámetro otro nodo. Hace que el padre del nodo entregado sea el nodo que recibe el método. Se asume que el usuario entrega las raíces de dos distintos clusters para evitar generación de ciclos en la estructura.

Estas operaciones de Union-Find necesitan que todos los vértices relevantes se mantengan en memoria al mismo tiempo, para no perder las conexiones en la estructura Union-Find.

La clase principal tiene los siguientes campos:

- `image`: Campo que guarda la imagen sobre la cual se realizará la segmentación.
- `vertices`: Matriz del mismo ancho y alto que la imágen, donde cada coordenada almacena un objeto de la clase `Vertex`, para poder mantenerlos en memoria y que no sean borrados al perder sus referencias.
- `clusters`: Diccionario donde las llaves son 2-tuplas que corresponden a coordenadas de la imágen, y los valores son una lista de 2-tuplas que tienen como raíz en la estructura Union-Find, la llave que les apunta.
- `edges`: Diccionario donde las llaves son 2-tuplas que corresponden a un par de coordenadas de la imágen, y los valores son objetos Edge, los cuales simplemente almacenan el peso correspondiente a esa arista y los dos vértices que la forman.

La inicialización de la clase comienza llenando la matriz `vertices` con cada coordenada teniendo un objeto `Vertex` con la coordenada de aquella celda. El diccionario clusters comienza inicializado con todas las coordenadas de la imágen, y una lista conteniendo la misma coordenada, como pares llave-valor. Finalmente, el diccionario edges fue el más problemático, pues este ocupaba mucha memoria si se almacenaban todas las aristas de la imágen. Para resolver esto se decidió realizar durante el parsing de estos bordes la unión de vértices donde la arista que los conectaba tenía peso 0. Esto es válido según el algoritmo, pues la evaluación de condición para unir distintos clusters se realiza en el set de aristas, en orden no creciente, por lo que las aristas con peso 0 siempre irán primero (notar que ninguna arista puede tener peso negativo). Además de esto, cualquier arista de peso 0 siempre fuerza una unión, para cualquier valor de $k > 0$. De este modo, en la inicialización de la clase sólo se guardan los valores de las aristas no nulos, y se unen todos los nodos con aristas de peso 0.

Otro punto importante es que para evitar la duplicación de elementos en el diccionario de aristas, para un par de vértices v_i, v_j, sólo se almacenaba en el diccionario la tupla (v_i, v_j), y se implementó una función que buscaba si existía un par dado en el diccionario, buscando las dos posibles permutaciones como llave. Si no se encontraba se retornaba 0.

Esta clase define además métodos para el cálculo de las métricas de cada cluster, incluyendo el cálculo del MST de un cluster y la diferencia interna mínima entre dos clusters.

3. Resultados

La solucuón implementada logra calcular las aristas de la imágen relativamente rápido, pero al iterar sobre las aristas no-cero, suele frenarse bastante luego de cierto tiempo. Se sospecha que esto es debido a falta de memoria, por lo que la colección de basura frena la ejecución del programa.

Además, la segmentación generada no es muy buena, de hecho, no se logran ver los objetos originales de la imágen, por lo que probablemente exista una falla en el cálculo de distancias en la imágen o en la obtención del MST de ésta, por lo que se evalúa mal la condición de mezcla de clusters.

A continuación se presentan los resultados de la segmentación variando algunos parámetros y para las imágenes entregadas, y las buscadas.

3.1. Efecto de el parámetro k

Se probó la segmentación sobre la imágen 1 para $k \in (150, 500, 1000)$, obteniendo los siguientes resultados.

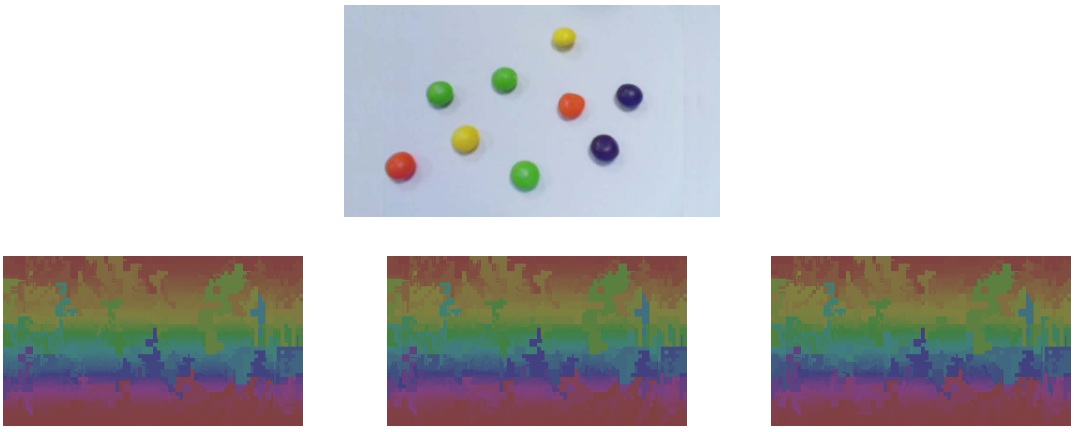

Figura 1: Resultados obtenidos para la imagen 1. De derecha a izquiera, $k = 150$, $k = 500$ y $k = 1000$.

No se logra ver mucho cambio entre los distintos valores de k, esto probablemente se deba al mismo error que causa que las segmentaciones generadas no tengan mucho sentido. Cada una de las segmentaciones tomó alrededor de 1 hora en generarse.

3.2. Resultados para imágenes entregadas

Sólo se logró generar segmentaciones sólo para las imágenes 1, 2 y 3, debido a quela resolución de las imágenes 4 y 5 era demasiado grande, lo que hacía poco viable realizar este análisis sobre ellas, dado que tomarían sobre 6 horas cada una.

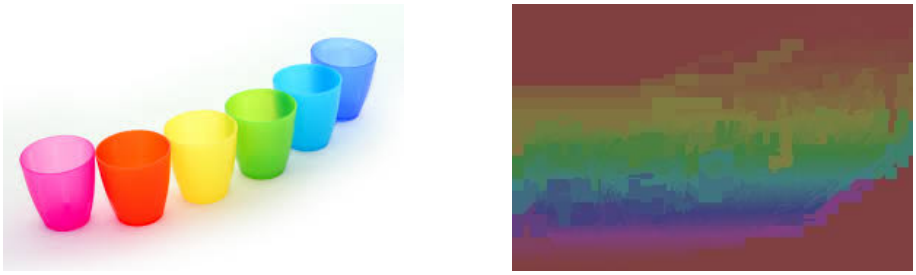

Figura 2: A la izquierda la imágen 2, y la derecha el resultado de su segmentación. Esta tomó 8 minutos en generarse.


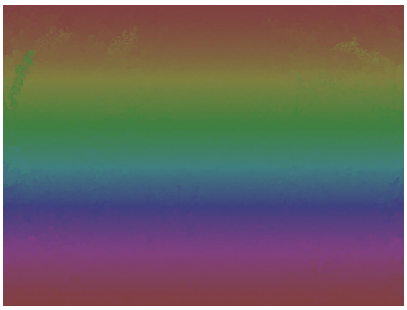

Figura 3: A la izquierda la imágen 3, y la derecha el resultado de su segmentación. Esta tomó 4 horas en generarse.

3.3. Resultados para imágenes buscadas

Debido a la limitación en el tamaño impuesta por la implementación poco eficiente, se utilizaron imágenes de baja resolución (256x256 pixéles).


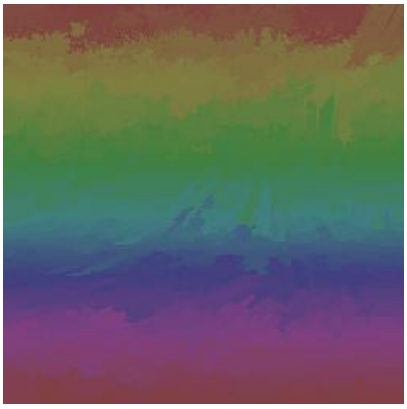

Figura 4: A la izquierda la imágen 6, y la derecha el resultado de su segmentación. Esta tomó 13 minutos en generarse.


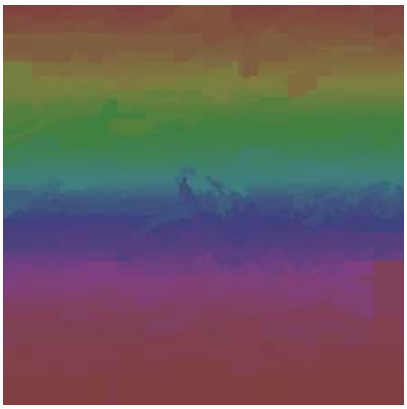

Figura 5: A la izquierda la imágen 7, y la derecha el resultado de su segmentación. Esta tomó 4 minutos en generarse.


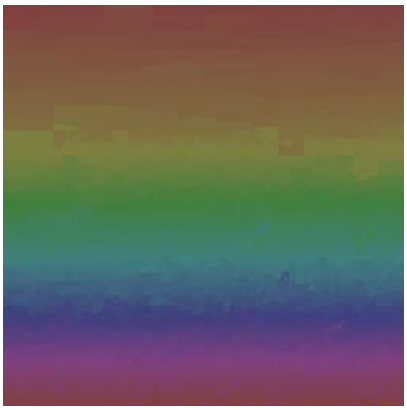

Figura 6: A la izquierda la imágen 8, y la derecha el resultado de su segmentación. Esta tomó 5 minutos en generarse.

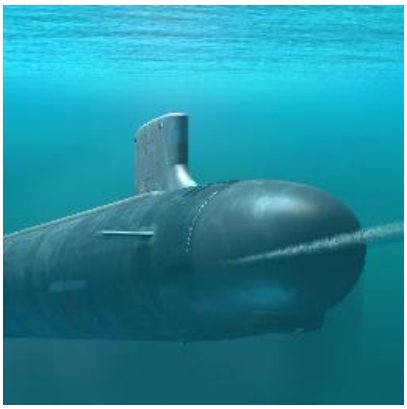
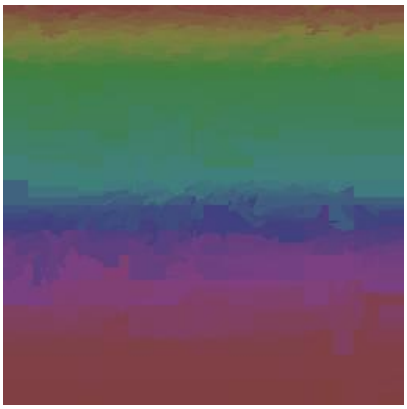

Figura 7: A la izquierda la imágen 9, y la derecha el resultado de su segmentación. Esta tomó 8 minutos en generarse.


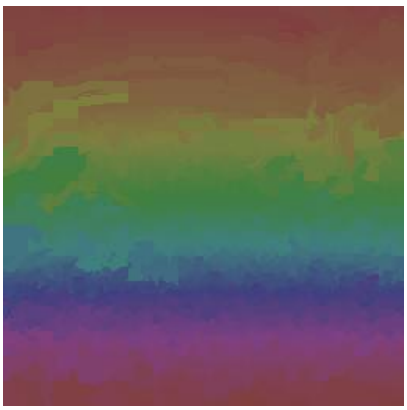

Figura 8: A la izquierda la imágen 10, y la derecha el resultado de su segmentación. Esta tomó 8 minutos en generarse.

Todas las segmentaciones fueron calculadas en un computador corriendo Ubuntu 18.04, con

16gb de RAM y reloj de CPU a 3.2 GHz.

4. Conclusiones

El algoritmo propuesto para esta tarea era más costoso que los anteriores, por lo que la optimización era extremadamente importante. Debido a esto, al no conseguir una versión óptima, se hizo difícil el testeo de los resultados, lo que dificultó mucho el poder encontrar bugs en el funcionamiento y una posible razón de porqué el algoritmo entregaba segmentaciones tan malas.

Una posible falta en el algoritmo implementada es la búsqueda del MST de un cluster. En la implementación actual estos se calculan cuando se necesitan para cada nodo. Esto implica conseguir la lista de todos los nodos en el cluster correspondiente, y para cada pixel buscar sus vecinos, ver si cada vecino pertenece al cluster usando Union-Find, y luego agregar el peso en el diccionario de pesos a una lista, la cual luego es ordenada por pesos no-crecientes y se comienza el algoritmo de Kruskal para encontrar el MST. Esto podría ser cambiado por tener pre-calculados los MST de cada cluster, y al haber una mezcla de clusters, sólo uniéndolos con la arista de menor peso que conecte a los dos cluster (la cual siempre será la arista que causó la unión). De este modo no se deben recalcular en todas las iteraciones, y se la mezcla de distintos MST es mucho más rápida.