UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Propuesta de Métodos y Optimización de Algoritmos-Rutinas para la Segmentación de Imágenes

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Actuario

PRESENTA:

Andres Papaqui Notario

TUTOR

M. en C. César Carreón Otañez

Ciudad de México 2021

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

A mi madre, quien siempre me apoyó en mis estudios.

A mis amigos. En especial a Marcos, quien me encaminó positivamente y a Esteban, quien me presionó continuamente.

Introducción

El desarrollo de computadoras/sistemas “inteligentes” es un tema realmente emocionante. Muchos de los avances tecnológicos vivían solo en la imaginación de ciertas personas hace apenas 20 años. Gran parte de los avances fueron inspirados por el mundo que nos rodea, desde aeronaves con fuselaje similar a la la anatomía de insectos y aves, hasta el cambio de paradigma al intentar que las computadoras “aprendan”, en lugar de ser una persona quien programe reglas de decisión para las computadoras. Si tomamos al mundo como referencia, y en específico a los seres vivos, nos tendríamos que preguntar cómo funcionan y cómo han llegado hasta este punto. Es con estas preguntas que la visión y el procesamiento de imágenes en computadores se pueden relacionar.

En el humano, más de la mitad de la corteza cerebral es usada para procesos relacionados a la vista. Incluso la adaptación a la luz se puede ver en casi todos los seres vivos, desde la simple capacidad de detectar luz de las medusas, hasta visión más sofisticada como vemos en los ojos de los caracoles, quienes cuentan con un gran lente esférico que puede enfocar y ajustar cuánta luz pasa. La visión también sirvió como catalizador de la evolución en el periodo conocido como explosión cámbrica, donde hace 540 millones de años la diversidad de las especies tuvo un aumento sin precedentes, intensificando la depredación y la necesidad de las especies de adaptarse para sobrevivir [11].

Siendo precisos, en el ambiente existen radiaciones electromagnéticas de diferentes longitudes de onda, los colores solo existen en nuestra mente. Así es como el ojo, receptor de ondas electromagnéticas, es solo la primera parte de la visión. La segunda consiste en enviar lo que se percibe al cerebro y la tercera es interpretar esos pulsos eléctricos. De la misma manera, las computadoras pueden tener la habilidad de percibir información visual, pero su interpretación es otra historia, en este sentido las computadoras siguen ciegas.

Uno de los primeros pasos que se ha buscado para poder dar habilidad de entender a las computadoras es la detección de objetos en imágenes. En la década de los 70 ya había investigación para desarrollar sistemas que pudieran tener esta capacidad, principalmente basados en la detección de bordes y luego inferir objetos en tres dimensiones. También hubo esfuerzos enfocados en las intensidades de colores y sombras. En la década de los 80, la mayoría de los esfuerzos se enfocaron en técnicas con mayor sustento matemático. Se continuó la investigación para mejorar la detección de

contornos. Hubo avances teóricos importantes para el desarrollo de las redes neuronales convolucionales (CNN), que hoy en día, son uno de los modelos con mayor éxito para el procesamiento de imágenes. También surgió el uso de modelos probabilísticos gráficos, en particular, aplicaciones de campos aleatorios markovianos (MRF). Al igual que las CNN, los modelos basados en MRF sufrirían del poder de cómputo de su época.

Algunas de las publicaciones más importantes que dieron forma a los modelos de procesamiento de imágenes son: Image Segmentation Using Simple Markov Fields Models por Hansen y Elliot, en 1981. Application of the Gibbs Distribution to Image Segmentation por Elliot, Derin, Cristi, Geman en 1983. Bayes Smoothing Algorithms for Segmentation of Images Modelled by Markov Random Fields por Derin, Elliott, Cristi, Geman en 1983. Una de las publicaciones más influyentes y más citadas es Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images por Geman y Geman en 1984.

En la última década, el esfuerzo más importante lo ha dado el proyecto ImageNet, apoyado por la universidad de Stanford que, aparte de proveer una vasta base de datos con imágenes preprocesadas para la investigación, inició el reto para desarrollar un sistema que pudiera reconocer objetos en las imágenes. En su última versión (2017), donde el reto contaba con 22,000 categorías de objetos, se llegó a un mejor resultado que el desempeño promedio de cualquier humano.

Una desventaja de los mejores sistemas actualmente, es la naturaleza de su reconocimiento. El sistema te dice si en un subconjunto rectangular de la imagen existe el objeto buscado, pero no infiere el objeto en un contexto más amplio. Existen otros tipos de sistemas que extraen el objeto sin el ruido de imagen adicional que conlleva contemplar una región rectangular completa. En particular, los sistemas basados en campos aleatorios markovianos (MRF), ofrecen la ventaja de extraer únicamente el objeto de interés, primero segmentando la imagen.

El artículo Iterative MAP and ML Estimations for Image Segmentation publicado en 2007 [3] explica cómo encaja la teoría de las redes markovianas con la práctica y los resultados obtenidos.

La presente tesis busca comprobar la utilidad de métodos similares al expuesto en [3] y buscar alternativas más viables.

El primer capítulo, se dan los conceptos básicos que se utilizan a lo largo de la tesis. El segundo capítulo plantea la segmentación de imágenes en términos de probabilidades. El tercer capítulo replantea lo descrito en el capítulo anterior en un problema equivalente para poder hacer manejable el procesamiento. El cuarto capítulo está dedicado para esclarecer el camino tomado en puntos ambiguos al recrear el proceso. Por último, el quinto capítulo describe un método heurístico para reducir la magnitud de computos necesarios en el método descrito anteriormente y presenta los resultados.

Índice general

Introducción **1**

1. Conceptos previos **7**

1.1. Probabilidad 7

1.2. Programación Lineal 11

1.2.1. Dualidad 13

1.3. Teoría de Gráficas y Redes 14

1.4. Algoritmo K-medias 22

2. Segmentación de imágenes **25**

2.1. Representación de imágenes 25

2.2. Textura de un pixel 26

2.3. Introducción al algoritmo de segmentación 27

2.4. Estimación de f 29

2.5. Estimación de Φ 30

3. MAP como cortadura mínima **35**

3.1. Construcción de la gráfica 37

3.2. Equivalencia gráfica con función de energía 41

4. Consideraciones Tomadas **45**

4.1. Textura 45

4.2. Gráficas 45

4.3. Estimación MAP: $f \mid \Phi$ 46

5. Aportaciones y Resultados **49**

5.1. Reducción de regiones 49

5.2. Pruebas de Convergencia 52

5.3. Segmentación en diferentes escenarios 55

5.4. Comentarios finales 57

Apéndice **59**

Capítulo 1

Conceptos previos

La primera parte de la tesis busca dar una breve introducción en las áreas necesarias para el desarrollo de este texto: Al plantear la segmentación de imágenes, expresamos el problema en términos de probabilidad. Luego, introducimos el problema de optimización, donde utilizamos programación lineal, teoría de redes y gráficas. Por último, para poder escoger una solución inicial en nuestro problema de optimización, usamos el algoritmo k-medias.

1.1. Probabilidad

La probabilidad nos ayuda a medir la incertidumbre de eventos en un espacio definido. Formalmente, al espacio donde viven todos los posibles resultados de un experimento aleatorio se le llama **espacio muestral Ω**.

Para poder aplicar una función de probabilidad en Ω, los elementos a los que se le aplica la función deben pertenecer a una colección de subconjuntos de Ω para que cumplan condiciones necesarias. A la colección de subconjuntos de Ω de interés la llamaremos **σ-álgebra**.

Definición 1.1.1. *Una colección $\mathscr{F}$ de subconjuntos de Ω es una σ-álgebra si cumple las siguientes condiciones:*

1. $\Omega \in \mathscr{F}$.
2. *Si* $A \in \mathscr{F}$, *entonces* $A^c \in \mathscr{F}$.
3. *Si* $A_1, A_2, ... \in \mathscr{F}, entonces \bigcup_{n=1}^{\infty} A_n \in \mathscr{F}$.

A la pareja $(\Omega, \mathscr{F})$ se le denomina espacio medible y los elementos de $\mathscr{F}$ son a los que llamamos eventos.

Dado el espacio muestral y la σ-álgebra, la función de probabilidad se define de la siguiente manera:

Definición 1.1.2. *Sea $(\Omega, \mathscr{F})$ un espacio medible. Una medida de probabilidad es una función $P : \mathscr{F} \longrightarrow [0, 1]$ que satisface*

1. $P(\Omega) = 1$.
2. $P(A) \geq 0$*, para cualquier* $A \in \mathscr{F}$.
3. *Si* $A_1, A_2, ... \in \mathscr{F}$ *y* $A_n \cap A_m = \emptyset$*, para cualquier* $A_n \neq A_m$*, entonces* $P(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} P(A_n)$.

Uno de los conceptos más importantes en la tesis es la probabilidad de eventos condicionados a otros eventos. Por ejemplo, al lanzar dos dados ¿Cuál es la probabilidad de sumar 4, dado que en un dado salió 1? y ¿dado que en un dado salió 5?. Para el último caso nuestra intuición nos dice que la probabilidad debe ser cero porque no es posible sumar 5 más otro número n en $\{1, 2, 3, 4, 5, 6\}$ y obtener 4, para el primer caso necesitamos un poco más que intuición.

A continuación, se presenta la definición de **Probabilidad Condicional** que nos ayuda manejar probabilidades de eventos condicionados:

Definición 1.1.3. *Sean A y B eventos con $P(B) \neq 0$. La probabilidad condicional de A dado B es:*

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}. \tag{1.1}$$

Con la definición 1.1.3 podemos contestar cuál es la probabilidad de sumar 4 con dos dados, dado que en uno salió 5:

Supongamos que el primer dado D_1 tiene valor 5.

$$
\begin{aligned}
&P(D_1 + D_2 = 4 \mid D_1 = 5) \\
&= \frac{P((D_1 + D_2 = 4) \cap (D_1 = 5))}{P(D_1 = 5)} \\
&= \frac{P(\emptyset)}{P(D_1 = 5)} \\
&= \frac{0}{P(D_1 = 5)} \\
&= 0.
\end{aligned}
$$

La intersección $(D_1 + D_2 = 4) \cap (D_1 = 5)$ *es vacía, ya que no hay* $n \in \{1, 2, 3, 4, 5, 6\}$ *que cumpla:* $5 + n = 4$.

Lo único que nos faltaría para resolver la primera pregunta de los dados, es el concepto de independencia.

Definición 1.1.4. *Dos eventos* A *y* B *son independientes cuando:*

$$P(A \cap B) = P(A)P(B).$$

En la mayoría de los casos, la independencia tiene que ser una hipótesis por parte del observador aunque puede llegar a ser difícil de asumir. Por ejemplo, dos eventos independientes no quiere decir que sean ajenos ni al revés, dos eventos ajenos no son necesariamente independientes. Aunque en la mayoría de los casos es evidente el que dos eventos sean independientes.

Con las definiciones de independencia (1.1.4) y probabilidad condicional (1.1.3) podemos deducir la probabilidad de la suma de dos dados sea igual a 4 dado que en uno salió 1:

Sea D_i*, los dos dados,* $i \in \{1, 2\}$*. Entonces:*

$$
\begin{aligned}
&P(D_1 + D_2 = 4 \mid D_1 = 1) \\
&\qquad = \frac{P((D_1 + D_2 = 4) \cap (D_1 = 1))}{P(D_1 = 1)} \\
&\qquad = \frac{P(((D_1 = 1, D_2 = 3) \cup (D_1 = 2, D_2 = 2) \cup (D_1 = 3, D_2 = 1)) \cap (D_1 = 1))}{P(D_1 = 1)} \\
&\qquad = \frac{P(D_1 = 1, D_2 = 3)}{P(D_1 = 1)} \\
&\qquad = \frac{P(D_1 = 1)P(D_2 = 3)}{P(D_1 = 1)} \\
&\qquad = P(D_2 = 3) \\
&\qquad = \frac{1}{6}.
\end{aligned}
$$

La probabilidad condicional es útil para resolver muchas preguntas acerca de un experimento aleatorio pero, ¿qué pasa con otro tipo de preguntas?, como: ¿podríamos saber la probabilidad de haber obtenido 5 en uno de los dos dados lanzados, dado que la suma de ambos es 8?

El siguiente teorema llamado **teorema de Bayes**, nos ayuda a deducir probabilidades más complejas:

Teorema 1.1.5. *Teorema de Bayes*

Sea $(\Omega, \mathscr{F}, P)$ un espacio de probabilidad y sea $A_1, A_2, ...$ una partición de Ω tal que cada elemento de la partición es un evento con probabilidad estrictamente positiva, entonces para cualquier evento B tal que $P(B) > 0$ y $m \geq 1$ fijo,

$$
P(A_m \mid B) = \frac{P(B \mid A_m)P(A_m)}{\sum_{n=1}^{\infty} P(B \mid A_n)P(A_n)}.
$$

Con el Teorema 1.1.5 y toda la teoría anterior tenemos las herramientas suficientes para responder la última pregunta de los dados: dado que el resultado de un lanzamiento de dos dados es 8 ¿Cuál es la probabilidad de haber sacado en uno de ellos 5?

$$
\begin{aligned}
&P(D_1 = 5 \; o \; D_2 = 5 \mid D_1 + D_2 = 8) \\
&\quad = P(D_1 = 5, D_2 \neq 5 \mid D_1 + D_2 = 8) + P(D_1 \neq 5, D_2 = 5 \mid D_1 + D_2 = 8) \\
&\quad\quad + P(D_1 = 5, D_2 = 5 \mid D_1 + D_2 = 5) \\
&\quad = P(D_1 = 5, D_2 \neq 5 \mid D_1 + D_2 = 8) + P(D_1 \neq 5, D_2 = 5 \mid D_1 + D_2 = 8) + 0 \\
&\quad = 2P(D_1 = 5, D_2 \neq 5 \mid D_1 + D_2 = 8) \\
&\quad = 2\frac{P(D_1 + D_2 = 8 \mid D_1 = 5, D_2 \neq 5)P(D_1 = 5, D_2 \neq 5)}{P(D_1 + D_2 = 8 \mid D_1 = 1)P(D_1 = 1) + ... + P(D_1 + D_2 = 8 \mid D_1 = 6)P(D_1 = 6)} \\
&\quad = 2\frac{P(D_1 + D_2 = 8 \mid D_1 = 5, D_2 \neq 5)P(D_1 = 5, D_2 \neq 5)}{0\frac{1}{6} + \frac{1}{6}\frac{1}{6} + \frac{1}{6}\frac{1}{6} + \frac{1}{6}\frac{1}{6} + \frac{1}{6}\frac{1}{6} + \frac{1}{6}\frac{1}{6}} \\
&\quad = 2\frac{\frac{1}{5}\frac{1}{6}\frac{5}{6}}{\frac{5}{36}} \\
&\quad = \frac{10}{25} \\
&\quad = \frac{2}{5}.
\end{aligned}
$$

1.2. Programación Lineal

La programación lineal estudia problemas de optimización para funciones lineales restringidas mediante ecuaciones o desigualdades lineales. Esta rama de las matemáticas tiene aplicaciones muy útiles, iniciando su desarrollo para operaciones militares en la segunda guerra mundial, hoy en día con usos en toda la industria.

En general, cualquier problema de programación lineal puede ser visto de la siguiente manera:

$$
\begin{aligned}
&\textit{Max (o Min)}\ z = c_1x_1 + c_2x_2 + ... + c_nx_n \\
&s.a \\
&a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \leq b_1 \qquad (o\ bien \geq o =) \\
&a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \leq b_2 \qquad (o\ bien \geq o =) \\
&. \\
&. \\
&. \\
&a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \leq b_m \qquad (o\ bien \geq o =) \\
&x_i \geq 0, i = 1, ..., n.
\end{aligned}
$$

A z se le llama *función objetivo*. Es la función que estamos buscando maximizar (o minimizar). Las desigualdades siguientes son las restricciones que debe cumplir una solución factible.

A través de reparametrización de las variables y constantes, así como una representación más compacta, podemos escribir simplemente:

$$
\begin{aligned}
&\textit{Max}\ z = cx \\
&s.a \\
&\qquad Ax \leq b \\
&\qquad x \geq 0.
\end{aligned}
$$

Cualquier problema de programación lineal es resoluble a través del algoritmo *simplex*, suponiendo que está acotado. Aunque el algoritmo simplex es unos de los más sencillos de implementar, existen problemas con una estructura particular de la cual nos podemos apoyar para encontrar de manera más eficiente el máximo (o mínimo), como es el caso del problema de flujo máximo.

El problema del flujo máximo se plantea en una gráfica (1.3) donde tenemos 2 nodos distinguidos, el nodo fuente (s) y el nodo objetivo (t). De este problema queremos encontrar la mayor cantidad de *flujo* posible que podemos llevar desde s a t a través de las aristas y nodos que los conectan. El flujo tiene muchas interpretaciones según el contexto, por ejemplo: supóngase que se quiere transportar cerveza desde su poblado origen s al poblado que la consume t. Se busca llevar la cantidad máxima de cerveza suponiendo que puede pasar por otros poblados, pero entre cada poblado existen capacidades máximas de transporte. Este problema es un problema de flujo máximo donde el poblado donde se produce la cerveza es el origen s, el poblado que consume la cerveza es el nodo objetivo t, el flujo que se quiere maximizar se refiere a la cantidad de cerveza transportada.

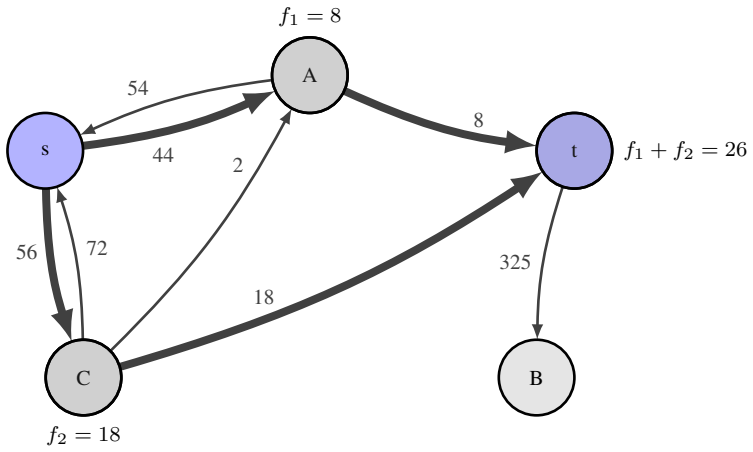


Figura 1.1: Ejemplo de solución para un flujo desde s a t, donde los caminos $(s, C), (C, t)$ y $(s, A), (A, t)$ definen el flujo máximo.

1.2.1. Dualidad

Todo problema de programación lineal tiene un problema dual asociado. Al problema original se le llama primal. Así, tenemos la definición del problema primal y del problema dual:

Problema Primal:

$$\begin{aligned} &\textit{Max } z = cx \\ &s.a \\ &\qquad Ax \leq b \\ &\qquad x \geq 0. \end{aligned}$$

Problema Dual

$$\begin{aligned} &\textit{Min } w = yb \\ &s.a \\ &\qquad yA \geq c \\ &\qquad y \geq 0. \end{aligned}$$

Los dos problemas están fuertemente relacionados y los detalles se pueden encontrar en [8].

Cada problema primal tiene un dual con su respectiva interpretación. Por ejemplo, si consideramos el problema de flujo máximo como problema primal, el problema dual asociado es el problema de la cortadura mínima. En la sección 1.3 se explica más a fondo el problema de la cortadura mínima.

La teoría de dualidad es increíblemente útil para resolver problemas lineales complejos, ya que al resolver el dual podemos saber la solución del primal, y viceversa. El teorema siguiente está demostrado en [8] y da fundamento a la afirmación anterior:

Teorema 1.2.1. *Sean x^* y y^* soluciones factibles del problema primal y dual, respectivamente, tales que: $cx^* = y^*b$.*

Entonces x^* y y^* son soluciones óptimas del problema primal y el dual.

1.3. Teoría de Gráficas y Redes

Definición 1.3.1. *Una Gráfica $G = (V, E)$ es una pareja ordenada donde V es un conjunto no vacío de vértices y E es un conjunto de aristas.*

Existen distintos tipos de gráficas, dependiendo de su uso, las cuales pueden ser dirigidas o no dirigidas, ponderadas o no ponderadas. Las gráficas dirigidas se caracterizan por tener sus aristas E como parejas ordenadas, es decir, cada arista tiene dirección. En contraste, las gráficas no dirigidas tienen sus aristas como parejas no ordenadas, y por tanto, no tienen dirección las aristas. Las gráficas ponderadas tienen pesos en sus aristas a los que podemos llamar también $capacidades$. Al tener una gráfica ponderada, también la podemos representar con $G = (V, E, C)$, donde la letra C representa el conjunto de capacidades ligadas a cada arista.

Definición 1.3.2. *Un corte C_G de una gráfica G es una partición de los vértices de G.*

Si tenemos una gráfica G ponderada, cada corte C_G tiene un costo $\mid C_G \mid$ que es la suma de los costos las aristas en C_G, i.e. son las aristas que definen el corte.

Existen varias aplicaciones que surgen en la teoría de gráficas para encontrar el corte de menor costo. Por ejemplo, el siglo pasado, en época de guerra, se buscaba el corte de las comunicaciones del enemigo con la mayor facilidad. Para la segmentación de imágenes, se trata de encontrar una separación de pixeles que maximice la probabilidad, o bien, que minimice la función de energía como se verá más adelante.

Una *Cortadura mínima s-t* es la cortadura con costo mínimo que separa a los nodos *s* y *t*. En general no haremos distinción entre *cortadura mínima s-t* y *cortadura mínima* si está implícito el papel de *s* y *t* en el contexto.

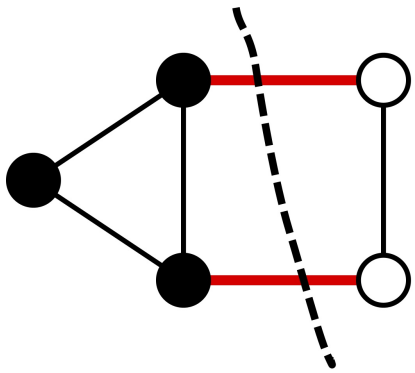


Figura 1.2: Corte de una gráfica. Las aristas en rojo pertenecen a C_G, e inducen la partición de los vértices: vértices negros y vértices blancos.

Existen varios algoritmos para encontrar la cortadura mínima entre dos puntos, la mayoría utilizan el problema dual de cortadura mínima, i.e. flujo máximo. El problema de flujo máximo se define en una gráfica dirigida con aristas e, capacidades c_e definidas por cada arista y con dos nodos distinguidos *s* y *t*, llamados source y sink, o bien, origen y destino.

Supóngase que se quiere enviar la mayor cantidad de flujo posible a través de la gráfica desde el origen hasta el destino. El flujo puede ir de nodo en nodo hasta llegar al nodo destino sin necesariamente cruzar por todos los nodos. También, el flujo debe respetar las direcciones de las aristas y sus capacidades, es decir, el flujo $f \leq c$. El más famoso de los algoritmos de flujo máximo es probablemente el algoritmo ford fulkerson publicado por L.R. Ford, Jr y D. R. Fulkerson en 1956.

Para el trabajo presente ocuparemos el algoritmo *push-relabel* diseñado por Andrew V. Goldberg y Robert Tarjan, presentado en 1986. Por su estructura, el algoritmo cuenta con una de las mejores complejidades en comparación con otros algoritmos existentes para encontrar flujo máximo.

Para poder entender el algoritmo Push-relabel introduciremos conceptos y subrutinas que ayudan a describir el algoritmo.

El primer concepto a introducir es el concepto de preflujo. En otros algoritmos la idea de un flujo se refiere a lo transportado desde el nodo origen y que pasa por los otros vértices para poder llegar al nodo destino. En el algoritmo Preflow-Push se usa la palabra *preflujo* ya que la condición de conservación que se usa no es tan restrictiva, por esto la llamaremos versión "débil".

Definición 1.3.3. *En una gráfica $G = (V, E, C)$ dirigida, llamaremos preflujo a la función $f : V \times V \longrightarrow \mathbb{R}$ que satisface:*

Conservación de flujo (débil)

$\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \geq 0$.

Restricción de capacidad

$\forall u, v \in V, 0 \leq f(u,v) \leq c(u,v)$.

Una de las definiciones más importantes que, además de ayudar a definir el flujo máximo, nos sirve también para definir la cortadura mínima es la *Gráfica residual*.

Definición 1.3.4. *Sea una gráfica $G = (V, E)$ con nodo origen s y nodo destino t, f un preflujo definido en G y un par de vértices $u, v \in V$. La* ***capacidad residual*** *$c_f(u,v)$ está definida por:*

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \textit{si (u,v)} \in E \\ f(v,u) & \textit{si (v,u)} \in E \\ 0 & \textit{Cualquier otro caso.} \end{cases}$$

A la gráfica inducida por $G_f = (V, E_f)$ se le define como **gráfica residual**, donde

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}.$$

La gráfica residual nos ayuda a saber por dónde podemos mandar el preflujo y, al finalizar el algoritmo, nos define la partición que induce el flujo máximo.

Otro concepto que usaremos es el de $exceso$. Intuitivamente, el exceso de un vértice v es el sobrante de (pre)flujo que se queda v después de tomar en cuenta el flujo que sale de v. Formalmente:

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v).$$

También introduciremos otro concepto llamado *función de altura*, o solo *altura*. La altura nos ayuda a etiquetar los vértices y es una guía para la selección de vértices en el algoritmo.

Definición 1.3.5. *Sea $G = (V, E)$ una gráfica con nodos origen s y destino t, f un preflujo en G. Una función $h : V \longrightarrow \mathbb{N}$ es una* ***función de altura*** *si $h(s) = |V|, h(t) = 0$ y $h(u) \leq h(v) + 1$ para todo nodo residual $(u,v) \in E_f$.*

Ahora definiremos dos operaciones utilizadas en el algoritmo Preflow-Push. La primera es un proceso para empujar (push) el preflujo a vértices con "menor altura". La segunda es un proceso

para cambiar la etiqueta a ciertos vértices para que pueda seguir fluyendo el preflujo a través de la operación Push.

Algoritmo 1: Push(u,v)

Aplica cuando: $e(u) > 0$, $c_f(u,v) > 0$ y $h(u) = h(v) + 1$;
$\Delta_f(u,v) = min(e(u), c_f(u,v))$
si $(u,v) \in E$ **entonces**
| $f(u,v) \leftarrow f(u,v) + \Delta_f(u,v)$
en otro caso
| $f(u,v) \leftarrow f(u,v) - \Delta_f(u,v)$
fin
$e(u) \leftarrow e(u) - \Delta_f(u,v)$
$e(v) \leftarrow e(v) - \Delta_f(u,v)$

Hay que notar que la condición $c_f(u,v) > 0$ es equivalente a $(u,v) \in E_f$, es decir, que aún se puede incrementar el preflujo a través de la arista (u,v) y la condición $e(u) > 0$ es equivalente a que el vértice u tenga un exceso que pueda mandar a otros vértices.

Algoritmo 2: Relabel(u)

Aplica cuando: $e(u) > 0$ y $\forall v \in V$ tal que $(u,v) \in E_f$, tenemos que $h(u) \leq h(v)$;
$h(u) \leftarrow 1 + min\{h(v) : (u,v) \in E_f\}$

Con los algoritmos y definiciones anteriores ya tenemos todo lo necesario para definir el algoritmo principal, el cual está expuesto en dos partes, la primera es para inicializar el algoritmo y el segundo es el algoritmo principal:

Algoritmo 3: Inicializar-Preflujo(G,s)

para cada *vértice* $v \in V$ **hacer**
| $h(v) = 0$
| $e(v) = 0$
fin
para cada *arista* $(u,v) \in E$ **hacer**
| $f(u,v) = 0$
fin
$h(s) =\mid V \mid$
para cada *vértice* $v \in \{v \in V \mid (s,v) \in E\}$ **hacer**
| $f(s,v) = c(s,v)$
| $e(v) = c(s,v)$
fin

Algoritmo 4: Push-Relabel(G)

Inicializar-Preflujo(G,s)
mientras *exista un vértice para aplicar Relabel o una arista para aplicar Push* **hacer**
| Aplicar operación
fin

Para pasar del problema de flujo máximo a cortadura mínima, en el caso de usar el algoritmo Push-Relabel, se define directamente usando la gráfica residual G_f resultante después de aplicar el algoritmo. La partición viene dada naturalmente por los nodos a los que se puede llegar desde el origen s en G_f y los que no.

En el caso de tener una gráfica no dirigida, el problema de cortadura mínima sigue teniendo sentido pero el problema de flujo máximo, al menos para el algoritmo descrito, está dado únicamente para gráficas dirigidas. Para poder obtener el flujo máximo de una gráfica no dirigida, y por ende la cortadura mínima, se aplica una transformación a la gráfica: todas las aristas se vuelven dirigidas en ambos sentidos con capacidades iguales a las ponderaciones de la gráfica original.

El proceso de encontrar la cortadura mínima de una gráfica no dirigida y usando el algoritmo Push-Relabel se ve ilustrado abajo.

Ejemplo 1.3.6. *Cortadura mínima de una gráfica no dirigida con algoritmo Push-Relabel.*

Tomemos la gráfica $G = (V, E)$ definida como se muestra en 1.3.

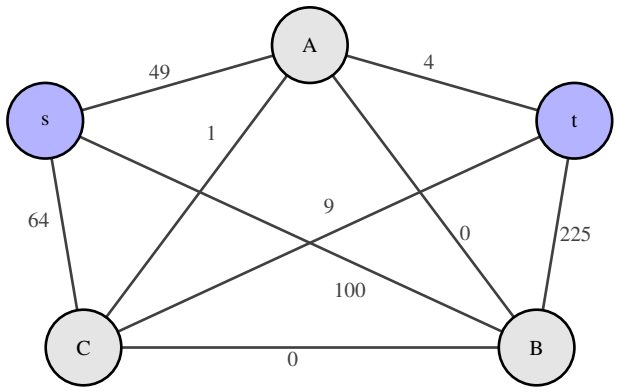


Figura 1.3: Gráfica no dirigida. Los pesos de cada arista se muestran a lo largo de la arista. Los nodos terminales están de color morado y el resto de los nodos en gris.

Para encontrar el flujo máximo desde s a t convertimos la gráfica en una gráfica dirigida 1.4.

Tomando la gráfica residual e inicializando el algoritmo obtenemos la figura 1.5.

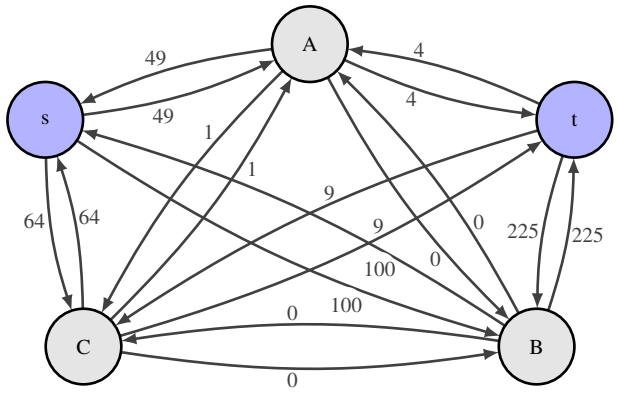


Figura 1.4: Gráfica dirigida creada a partir de la gráfica anterior. Cada arista fue reemplazada por dos arcos dirigidos, cada uno con el mismo peso de la arista original.

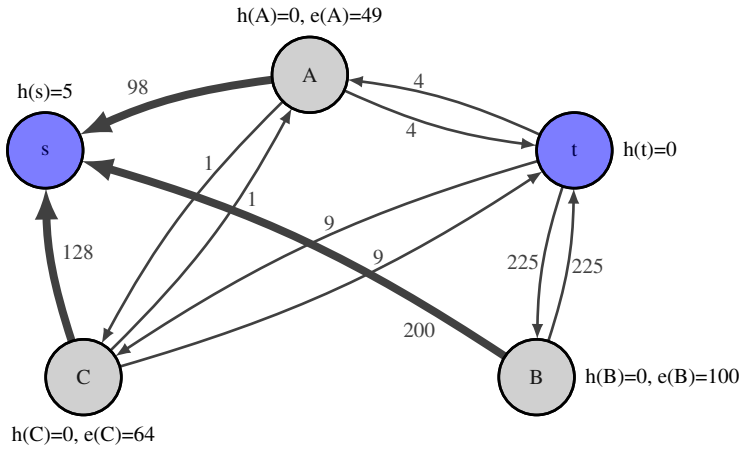


Figura 1.5: Al inicializar el algoritmo se manda todo el (pre)flujo posible a los vecinos de **s**. En particular, el nodo **A** recibe 49 unidades de flujo del nodo origen **s** y esto se ve reflejado en la gráfica residual al sumar 49 unidades del arco que va de **A** a **s** y restando 49 unidades al arco que va de **s** a **A**, que es aplicar la definición de la capacidad residual a la gráfica residual resultante. Al finalizar lo anterior, actualizamos el exceso e(A) = 49.

Seleccionamos al nodo A para aplicarle la operación relabel y push a sus nodos adyacentes con menor altura 1.6.

Volvemos a aplicar las operaciones relabel y push al nodo A y así regresar el excedente al

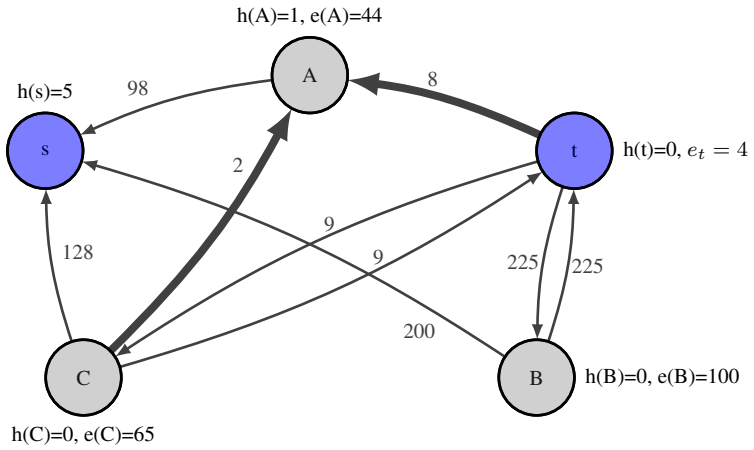


Figura 1.6: El nodo **A** envía todo el exceso que es posible a sus vecinos con altura $h(\cdot) = 0 = h(A) - 1$, en este caso **C** y **t**. Se actualiza las capacidades residuales. Para el caso de **C**, el exceso aumenta en uno y se reetiqueta, para el caso de **t** se agrega una etiqueta ilustrativa e_t para reflejar el flujo que llega al nodo destino **t**.

nodo terminal s, como aparece en la figura 1.7.

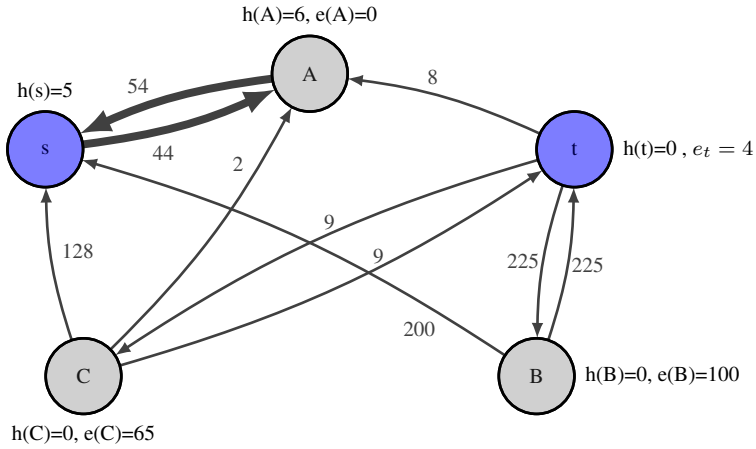


Figura 1.7: Al aplicar la operación relabel al nodo **A**, su altura se modifica a $h(A) = 1 + min\{h(v) : (A, v) \in E_f\} = 1 + h(s) = 6$. Luego, al mandar el exceso a **s**, se modifica $e(A) = 0$ y se actualizan las capacidades residuales.

Ahora que ya no se pueden hacer mas operaciones con A, aplicamos las operaciones relabel a C y Push a los vecinos de C con menor altura 1.8.

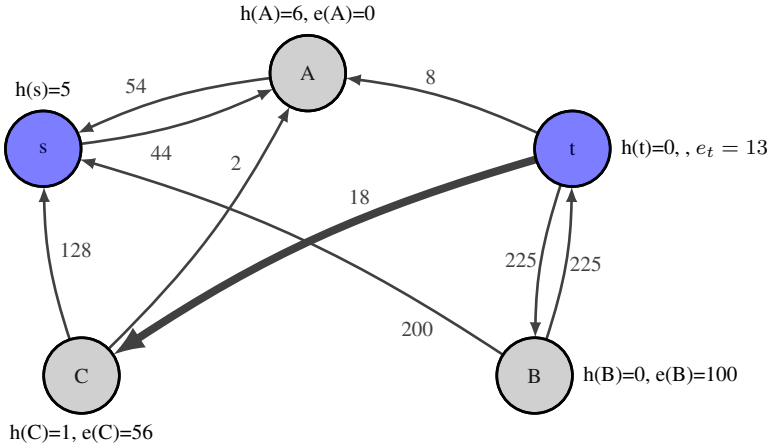


Figura 1.8: El nodo **C** envía su excedente a los vecinos con menor altura.

Volvemos a aplicar las operaciones relabel y push al nodo C y así regresar el excedente al nodo terminal s, como se muestra en la figura 1.9.

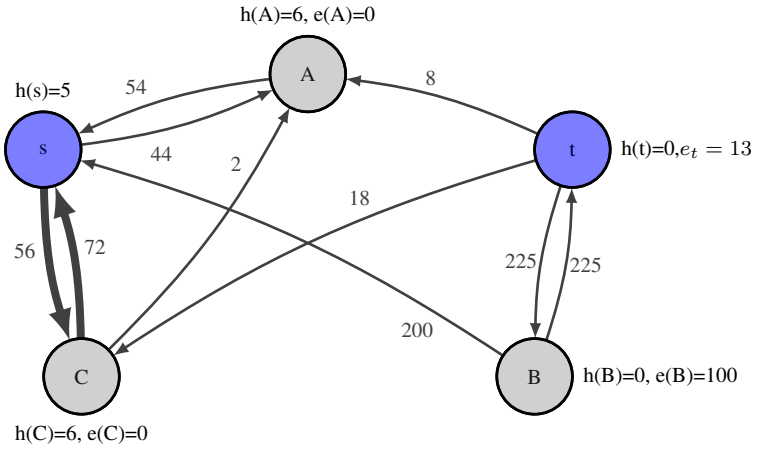


Figura 1.9: Operaciones Push y Relabel al nodo **C**.

Ahora que ya no se puede hacer mas operaciones con C, aplicamos las operaciones relabel a B y Push a los vecinos de B con menor altura 1.10.

Dado que ya no tenemos ningún nodo con exceso $e(\cdot) > 0$ el algoritmo termina. Ahora, para ver cuál sería la cortadura resultante, notemos cuáles nodos son accesibles desde s. Para esta gráfica tan chica, es evidente que los único nodos accesibles desde s a través de las aristas en E_f son C y A, por lo tanto, la cortadura mínima de la gráfica original está dada por la figura 1.11.

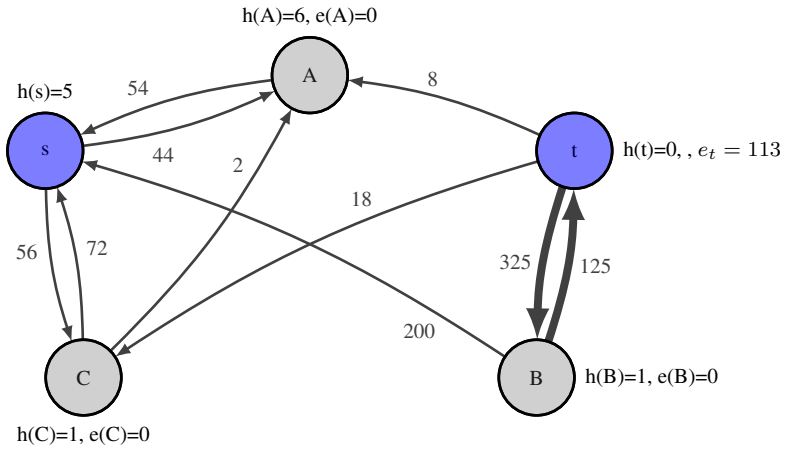


Figura 1.10: Operación relabel al nodo **B** y push a sus vecinos.

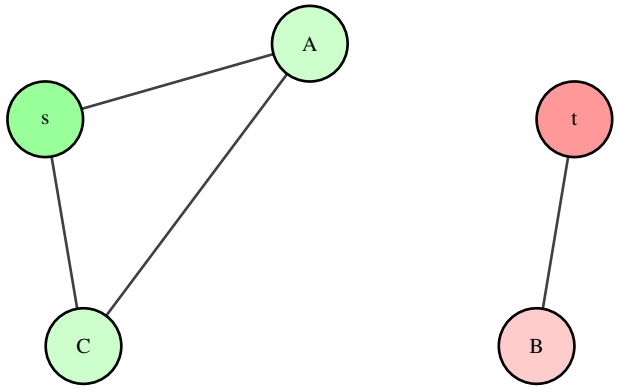


Figura 1.11: En verde y rojo se muestra la partición de vértices inducida por el flujo máximo. Las aristas no dirigidas que pertenecen a C_G son (A, t), (s, B) y (C, t).

1.4. Algoritmo K-medias

Segmentar, o clusterizar, se refiere a la tarea de agrupar objetos de tal manera que elementos en un mismo grupo son más similares entre ellos que a objetos de otros grupos.

Los algoritmos de segmentación tienen muchas aplicaciones, como separar poblaciones, comprimir imágenes, o en el caso de del presente trabajo, ayudar a segmentar una imagen.

El concepto de similitud entre los objetos depende del algoritmo utilizado. Realmente no necesariamente hay una única manera de segmentar un conjunto de objetos dados.

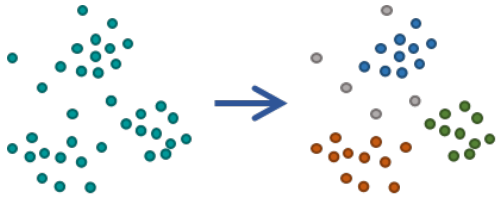


Figura 1.12: Ejemplo de segmentación. Cada color representa un grupo diferente en donde sus miembros son "similares" entre sí.

El algoritmo K-medias es uno de los algoritmos de segmentación más conocidos y usados, tal vez debido a su sencillez de implementación y entendimiento. El algoritmo usa elementos distinguidos que sirven como representantes de cada clúster y encuentra la agrupación que minimice la suma de los errores al cuadrado (SSE) a estos representantes:

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} dist(c_i, x)^2. \tag{1.2}$$

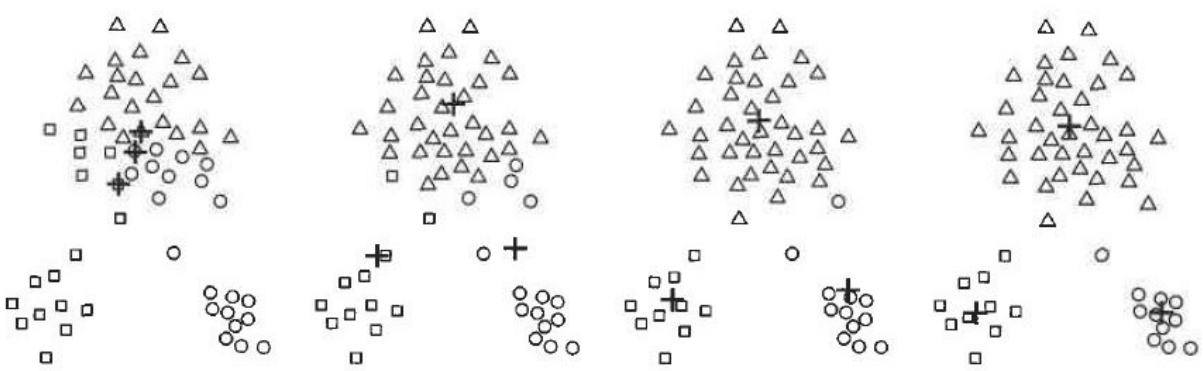
Entonces los c_i que minimizan 1.2 son:

$$c_i = \frac{1}{m_i} \sum_{x \in C_i} x.$$

Entonces, el algoritmo, en cada iteración, busca mejorar las agrupaciones tomando los centroides de cada grupo. El algoritmo está dado de la siguiente manera:

Algoritmo 5: Algoritmo K-medias

Entrada: K, objetos.
1. Selecciona K centroides iniciales.
mientras *Centroides cambien* **hacer**
  3. Asignar cada objeto al clúster con su centroide más cercano.
  4. Recalcular los centroides.
fin
Repetir:
Salida: Objetos agrupados en K clústers.



Iteración 1. Iteración 2. Iteración 3. Iteración 4.

Imagen encontrada en [13]. En cada iteración se van redifiniendo los representantes de cada región, esto se ve ilustrado por las cruces que varían de posición. Al redefinir los representantes en cada iteración implica una reasignación de los elementos de cada grupo.

Capítulo 2

Segmentación de imágenes

2.1. Representación de imágenes

Las imágenes, como las personas las interpretan, tienen gran diferencia a cómo las computadoras las interpretan. Para una persona puede ser una representación de un objeto en el mundo, un concepto, un paisaje, etc. Para una computadora una imagen es una colección de 0's y 1's codificados de una manera particular para ahorrar espacio, esta manera de manejar la imagen es también diferente a como la computadora representa la imagen para que los humanos puedan apreciarla.

La manera en la que una computadora representa una imagen para ser visualizada es a través de pixeles. Un pixel, para el ojo humano, es un punto con un color en particular que, en conjunto con otros pixeles, forman la imagen. Es decir, el pixel es la unidad básica para representar una imagen digital. Para una máquina, un pixel es un arreglo de números que representan el color a desplegar en un punto particular del monitor.

Figura 2.1: Pixeles de una imagen. A la izquierda se aprecia la imagen en tamaño original. A la derecha un acercamiento de una región donde se puede apreciar los elementos más pequeños de toda imagen digital, los pixeles.

La representación de colores en un pixel se da por modelos de color, el más popular siendo el RGB. La representación de un pixel con RGB permite crear un color componiendo otros tres colores: rojo, verde y azul. Los colores en el modelo RGB están definidos por tres números en un

vector $[r, g, b]$. La gama de números que pueden tomar r, g, b está dado por cómo se codifican, en particular, cuántos bits usamos para representar cada entrada. Por ejemplo, si usamos 8 bits para representar cada uno de estos números, entonces los valores van del 0 al $2^8 - 1$, o bien, del 0 al 255.

Aunque el modelo RGB es uno de los más famosos, hay otros modelos muy útiles. Uno de los más interesantes es el espacio de colores L*a*b (también conocido como CIELab), donde L representa el brillo, a la intensidad de verde (-) a rojo (+) y b de azul (-) a amarillo (+). Los valores de L van del 0 al 100, los valores de a y b, en una representación de 8 bits, van del -128 al 127. En el espacio de colores CIELab, los colores están representados independientemente del brillo, lo cual es muy útil para manipular las imágenes.

Actualmente existen muchas implementaciones en varios lenguajes de programación que nos ayudan a pasar de un modelo de color a otro y así poder trabajar en uno y visualizar en otro.

2.2. Textura de un pixel

La textura de un pixel es la propiedad que tiene un pixel con sus vecinos y que refleja de mejor manera la contribución que tiene el pixel en la imagen. Entre otras cosas, la textura nos puede ayudar a saber si un pixel pertenece a una superficie lisa o está contribuyendo a una **textura**, por ejemplo, en un conjunto de pixeles que forman el pelaje de un animal.

El tema de la textura tiene mucha bibliografía con diferentes enfoques de cómo representar propiedades deseadas. En particular, la tesis utilizará el **contraste de la textura** descrito en [2].

La textura utilizada es calculada como se muestra abajo:

$$t = k \times 2 \times \sqrt{\lambda_1 + \lambda_2} \tag{2.1}$$

donde k sirve para escalar la textura, λ_1 y λ_2 son los valores propios de C, definida como:

$$C = \sum_{j \in N_{5x5}(i)} w(j) \nabla I(j) \nabla I(j)^T \tag{2.2}$$

donde $N_{5x5}(i)$ representa los vecinos de i cubiertos por la cuadrícula de 5×5 con centro en i. $w(j)$ es una ponderación que se le da a cada vecino de i y que aproxima a una distribución normal. $\nabla(j)$ es el gradiente de j: $\nabla I = \begin{bmatrix} I_x \\ I_y \end{bmatrix}$.

Es decir, C es una matriz $\in M_{2\times 2}$ formada por la suma de 25 matrices ponderadas por w. Para la ponderación, aplicamos una aproximación binomial a la distribución Normal dada por:

$$w(x,y) = 4^{-(4)}\binom{4}{x}\binom{4}{y}, \quad x,y = 0,1,2,3,4. \tag{2.3}$$

En la ecuación 2.3, w tiene dos entradas, las cuales están dadas como coordenadas en la cuadrícula de 5×5 de los vecinos. En 2.2 la entrada de w es un índice que recorre de izquierda a derecha y de arriba a abajo la cuadrícula y tiene una correspondencia 1 a 1 con las coordenadas. Por ejemplo, $j = 1$ es $x = 1, y = 1$, $j = 12$ es $x = 2, y = 3$, etc.

2.3. Introducción al algoritmo de segmentación

Segmentar una imagen es la tarea de agrupar pixeles de una imagen en regiones relevantes. Esto se puede ver como un problema de clasificación, donde cada pixel tiene que ser clasificado en la región que corresponde.

Figura 2.2: Imagen segmentada. De color amarillo los bordes de las regiones que induce la segmentación.

A un estado de pixeles con clasificaciones ya asignadas le llamaremos configuración, o bien, clasificación de los pixeles. En general, denotaremos a la configuración de los pixeles como: $f = \{f_p \mid p\}$, donde f_p es la clasificación asignada al pixel p.

Para representar cada pixel utilizamos un vector con 4 entradas, denotando en las primeras 3 el espacio de color Lab y en la última la textura como viene descrita en [2]:
$I(p) = (I_L(p), I_a(p), I_b(p), I_t(p))^T$.

Cada pixel $p \in P$ se clasifica en una región y cada región tiene un representante denotado por $\phi(i)$, donde P es el conjunto de pixeles que forman la imagen e i representa el índice de la región.

Al conjunto de representantes lo denotaremos como Φ. Así tenemos: $\Phi = \{\phi(i)\}$.

El proceso que seguiremos para segmentar una imagen es una sucesión de iteraciones en la que comenzamos con un conjunto de representantes iniciales Φ^0, con el conjunto formado estimamos la primera configuración f^0, a partir de esta configuración calculamos los nuevos representantes de las regiones Φ^1 y f^1 e iteramos para encontrar Φ_n, f_n hasta que no cambien de valor entre iteraciones o alcancen un número máximo de iteraciones.

En general, las fórmulas a optimizar son:

$$f^{n+1} = argmax_f Pr(f \mid \Phi^n, P) \tag{2.4}$$

$$\Phi^{n+1} = argmax_\Phi Pr(f^{n+1} \mid \Phi, P). \tag{2.5}$$

Donde f se encuentra por una estimación máxima a posteriori (MAP) y Φ por una estimación máxima verosímil (ML).

El siguiente algoritmo muestra el proceso para la segmentación de imágenes:

Algoritmo 6: Algoritmo de segmentación

Entrada: Imagen RGB.

1. Convertir la imagen al espacio de colores L*a*b y calcular la textura.

2. Inicializar Φ con el algoritmo K-medias.

3. Optimización iterativa:

3.1. Estimación MAP: Estimar f dado Φ.

3.2. Re-etiquetado: Renombrar los segmentos para que no haya 2 regiones disconexas con la misma etiqueta.

3.3. Estimación ML: Recalcular Φ dado f.

4. Si Φ y f no cambian entre dos iteraciones sucesivas o el número máximo de iteraciones es alcanzado finaliza el algoritmo, en otro caso ir al paso 3.

Salida: Múltiples regiones.

2.4. Estimación de f

El objetivo de la sección es saber cómo podemos encontrar $f^{n+1} = argmax_f Pr(f \mid \Phi^n, P)$, es decir, dado los pixeles P y los representantes actuales Φ^n, encontrar la configuración de pixeles que maximice su probabilidad.

Usando probabilidad condicional obtenemos:

$$Pr\left(f \mid \Phi, P\right) = \frac{Pr(\Phi, P \mid f)Pr(f)}{Pr(\Phi, P)}.$$

Dado que $Pr(\Phi, P)$ no depende de f, podemos solo concentrarnos en el resto de la expresión, i.e.

$$Pr(f \mid \Phi, P) \propto Pr(\Phi, P \mid f)Pr(f). \tag{2.6}$$

El primer factor de 2.6 lo podemos reescribir como:

$$Pr(\Phi, P \mid f) \propto \prod_{p \in P} exp\left(-D(p, f_p, \Phi)\right) \tag{2.7}$$

donde $D(p, f_p, \Phi)$ es una función que impone una penalización a la clasificación f_p dada Φ. En este caso:

$$D(p, f_p, \Phi) = || \; I(p) - \phi(f_p) \; ||^2 \; . \tag{2.8}$$

Por otro lado, el segundo factor de 2.9 se expresa de la siguiente forma:

$$Pr(f) \propto exp\left(-\sum_{p \in P} \sum_{q \in N(p)} V_{p,q}(f_p, f_q)\right) \tag{2.9}$$

donde $N(p)$ denota los vecinos de p y $V_{p,q}$ está definida como sigue:

$$V_{p,q}(f_p, f_q) = c \cdot exp\left(\frac{- \mid I_L(p) - I_L(q) \mid}{\sigma}\right) \cdot \mathbb{1}_{(f_p \neq f_q)}$$

donde c, $\sigma > 0$ constantes.

$V_{p,q}$ es una función que penaliza las diferencias entre los vecinos e intuitivamente, mientras más grande sea la diferencia entre pixeles vecinos, más probable será que sean clasificados en regiones diferentes.

Entonces, la expresión completa a maximizar es:

$$\arg\max_f \prod_{p \in P} exp(- \mid\mid I(p) - \phi(f_p) \mid\mid^2) \cdot exp\left(-\sum_{p \in P}\sum_{q \in N8(p)} c \cdot exp\left(\frac{- \mid I_L(p) - I_L(q) \mid}{\sigma}\right) \cdot \mathbb{1}_{(f_p \neq f_q)}\right). \tag{2.10}$$

De manera directa, podríamos encontrar el argumento f que maximice la expresión 2.10 al calcular las probabilidades de todas las posibles configuraciones f y escoger la que tenga la mayor probabilidad. En la sección 2.5.1 se muestra un ejemplo de la búsqueda de f calculando todas las combinaciones y en el capítulo siguiente se propone una manera mucho más eficiente.

2.5. Estimación de Φ

En el algoritmo 6, la primera estimación de Φ está dada por el algoritmo k-medias, descrito en la sección 1.4. En las siguientes estimaciones de Φ_i, el proceso es por una estimación máxima verosímil.

Para la estimación máxima verosímil tomamos la probabilidad conjunta y buscamos el argumento Φ que maximice $Pr(f \mid \Phi, P)$. Lo que es equivalente a encontrar el mínimo de $-log(Pr(f \mid \Phi, P))$:

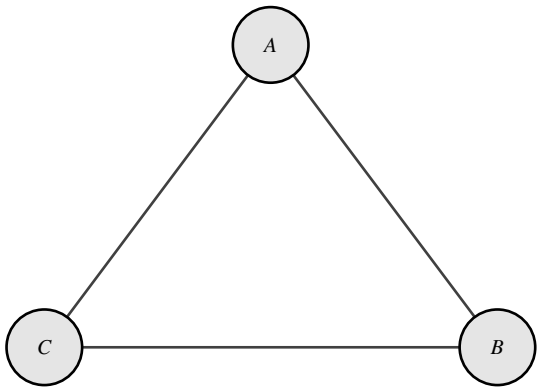
$$\begin{aligned}
0 &= \nabla_\Phi Pr(f \mid \Phi, P) \\
&= \nabla_\Phi \sum_{p \in P} (\mid\mid I(p) - \phi(f_p) \mid\mid^2) + \sum_{p \in P}\sum_{q \in N(p)} c \cdot exp\left(\frac{- \mid I_L(p) - I_L(q) \mid}{\sigma}\right) \cdot \mathbb{1}_{(f_p \neq f_q)} \\
&= \sum_{p \in P} 2 \times (\mid\mid I(p) - \phi(f_p) \mid\mid^2).
\end{aligned}$$

Esto implica que, para cada i:

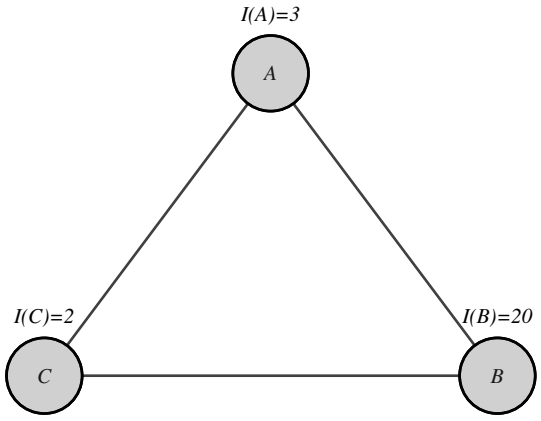
$$\sum_{f_p=i} \phi(f_p) = \sum_{f_p=i} I(p)$$
$$\rightarrow num_i \phi(i) = \sum_{f_p=i} I(p)$$
$$\rightarrow \ \phi(i) = \frac{\sum_{f_p=i} I(p)}{num_i}$$

donde $num_i =\mid \{p \in P \mid f_p = i\} \mid$, es decir, el número de pixeles clasificados como i.

Ejemplo 2.5.1. *Supongamos que $P = \{A, B, C\}$ i.e. tenemos una imagen con 3 pixeles (A, B, C).*



También, para simplificar todo, supongamos que cada pixel está representado por un vector de una dimensión y los valores de los pixeles son 3, 20 y 2 como muestra la gráfica siguiente.



Teniendo únicamente dos regiones denotadas por α y β, los valores de los representantes en la primera iteración son elegidos de diferentes maneras. En este caso:

$$\phi_0(\alpha) = 5$$
$$\phi_0(\beta) = 10.$$

Así, podemos ver todas las configuraciones f posibles de los pixeles:

	A	B	C
f_1	β	β	β
f_2	β	β	α
f_3	β	α	β
f_4	β	α	α
f_5	α	β	β
f_6	α	β	α
f_7	α	α	β
f_8	α	α	α

La idea de la primera iteración es encontrar la configuración f_i que maximice la probabilidad dada de Φ_0. A la probabilidad condicionada le llamaremos probabilidad a posteriori. Maximizar la probabilidad a posteriori y se le denota como MAP y es lo que busca de f_i. En particular:

$$Pr(f \mid \Phi_0, P) \propto Pr(\Phi_0, P \mid f)Pr(f)$$

$$= \left(\prod_{p \in P} exp(- \parallel I(p) - \phi(f_p) \parallel^2) \right) \cdot exp\left(-\Sigma_{p \in P} \Sigma_{q \in N(p)} exp(- \mid I(p) - I(q) \mid) \mathbb{1}_{(f_p \neq f_q)} \right)$$

donde $\mathbb{1}(\cdot)$ es 1 si el argumento es verdadero y 0 en otro caso.

De aquí, calculamos las probabilidades de cada configuración y así saber cuál maximiza la probabilidad a posteriori. Las probabilidades de todas las posibles configuraciones son:

$$
\begin{aligned}
Pr(f_1 \mid \Phi_0, P) &\propto exp(-8^2 - 7^2 - 10^2) \cdot exp(-0) && \approx 3 \cdot 10^{-93} \\
Pr(f_2 \mid \Phi_0, P) &\propto exp(-7^2 - 10^2 - 3^2) \cdot exp(-e^{-1} - e^{-18} - e^{-1} - e^{-18}) && \approx 1.2 \cdot 10^{-69} \\
Pr(f_3 \mid \Phi_0, P) &\propto exp(-7^2 - 8^2 - 15^2) \cdot exp(-e^{-17} - e^{-18} - e^{-17} - e^{-18}) && \approx 0 \\
Pr(f_4 \mid \Phi_0, P) &\propto exp(-7^2 - 3^2 - 15^2) \cdot exp(-e^{-1} - e^{-17} - e^{-1} - e^{-17}) && \approx 0 \\
Pr(f_5 \mid \Phi_0, P) &\propto exp(-8^2 - 10^2 - 2^2) \cdot exp(-e^{-1} - e^{-17} - e^{-1} - e^{-18}) && \approx 5 \cdot 10^{-74} \\
Pr(f_6 \mid \Phi_0, P) &\propto exp(-2^2 - 3^2 - 10^2) \cdot exp(-e^{-17} - e^{-18} - e^{-17} - e^{-18}) && \approx 8 \cdot 10^{-50} \\
Pr(f_7 \mid \Phi_0, P) &\propto exp(-2^2 - 15^2 - 8^2) \cdot exp(-e^{-1} - e^{-18} - e^{-1} - e^{-18}) && \approx 0 \\
Pr(f_8 \mid \Phi_0, P) &\propto exp(-2^2 - 3^2 - 15^2) \cdot exp(0) && \approx 0.
\end{aligned}
$$

De las probabilidades anteriores podemos observar que la configuración f_i que maximiza la probabilidad a posteriori es f_6.

Por lo tanto,

$$
\begin{aligned}
\Phi_0 &= \{\phi_0(\alpha) = 5, \phi_0(\beta) = 10\} \\
f_0 &= \{A : 0, B : 1, C : 0\}.
\end{aligned}
$$

Con lo anterior ahora podemos calcular Φ_1 como:

$$
\{\phi_1(\alpha) = \frac{2+3}{2} = 2.5, \phi_1(\beta) = 20\}
$$

y volvemos a iterar sobre f para calcular el máximo a posteriori.

El ejercicio nos muestra qué hace el algoritmo, pero no cómo lo hace. Para encontrar un estimador MAP de f_i, naturalmente, no podemos estar calculando todas las probabilidades de todas las configuraciones posibles para cada iteración. Esto representa un costo computacional demasiado alto. La estimación MAP de f_i se alcanza convirtiendo el problema en uno de minimización de la energía markoviana dado Φ_{i-1} y resolvemos usando el algoritmo Graph Cut. La explicación del algoritmo se da en las siguientes secciones.

Capítulo 3

MAP como cortadura mínima

En la sección 2.3 se introduce el concepto MAP para hablar de la configuración de pixeles que maximice la probabilidad a posteriori suponiendo que existen solo dos posibles clases de pixeles y en la sección 2.5.1 se mostró cómo, calculando las probabilidades de todas las configuraciones posibles, podemos escoger la que maximice esa probabilidad. Pero, el calcular las probabilidades de todas las configuraciones posibles representa una tarea nada eficiente.

Para dimensionar lo intratable que puede llegar a ser el problema, veamos el siguiente ejemplo: Tomando una imagen de 210x280 pixeles, que representa una imagen muy chica para los estándares actuales, y suponiendo que solo existen dos posibles clases de pixeles, obtenemos 2^{58800} posibles configuraciones. Aún tomando la imagen a una calidad muy inferior al estándar (51x68 pixeles), obtenemos $\sim$3400 pixeles en una imagen, lo que implica 2^{3400} posibles configuraciones.

Sumado a las posibles configuraciones, hay que tomar en cuenta que cada configuración implica el cálculo de su respectiva probabilidad, que representa para una imagen de n x m pixeles: $\sim 4 \cdot (n-2) \cdot (m-2)$ multiplicaciones donde cada factor tiene contenido funciones exponenciales, potencias y sumas.

Para simplificar la estimación primero tomamos de la ecuación 2.6 a la ecuación 2.9 y obtenemos:

$$Pr(f \mid \Phi, P) \propto \left(\prod_{p \in P} exp(-D(p, f_p, \Phi)) \right) \cdot exp\left(- \sum_{p \in P} \sum_{q \in N(p)} V_{p,q}(f_p, f_q) \right). \tag{3.1}$$

Luego definimos la función de energía $E(f, \Phi)$ como menos el logaritmo de 3.1, i.e.

$$E(f, \Phi) = \sum_{p \in P} \left(D(p, f_p, \Phi) + \sum_{q \in N(p)} V_{p,q}(f_p, f_q) \right). \tag{3.2}$$

Figura 3.1: 51 x 68 pixeles.

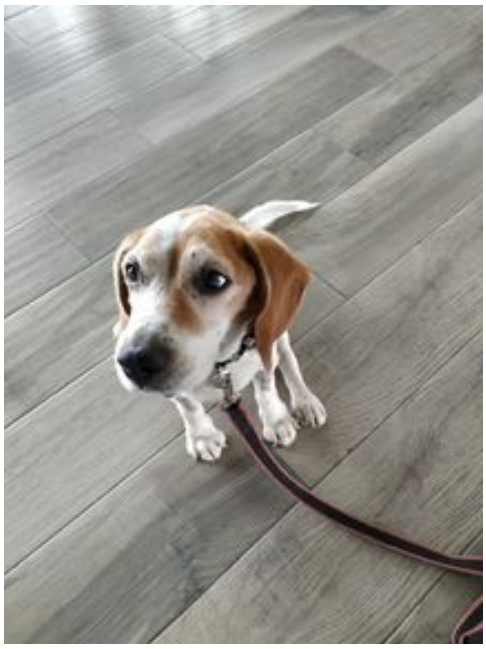


Figura 3.2: 216 x 288 pixeles.

Por lo tanto, encontrar el argumento f que maximice $Pr(f \mid \Phi, P)$ es equivalente a encontrar el argumento f que minimice la función de energía $E(f, \Phi)$.

Las siguientes secciones muestran la equivalencia entre minimizar la función de energía 3.2 y encontrar la cortadura mínima de una gráfica específica.

Primero se mostrará cómo se construye la gráfica que utilizaremos para encontrar la cortadura mínima y luego se demuestra la equivalencia entre encontrar f que minimice $E(f, \Phi)$ y la cortadura mínima de la gráfica construida.

3.1. Construcción de la gráfica

La construcción de la gráfica se basa en minimizar la función de energía de dos clases a la vez, ya que al convertir el problema en uno de cortadura mínima, el algoritmo solo tiene sentido entre dos clases.

Para generalizar el problema a varias clases, el artículo [1] propone dos tipos de movimientos para poder minimizar la función de energía para gráficas con $\mathcal{L} \geq 2$ clases diferentes. El movimiento que usamos es llamado $Swap$.

Dado dos clases fijas α, β, un movimiento de f a f' es llamado un $\alpha - \beta$ $swap$ si $f_l = f'_l$ para toda clase $l \neq \alpha, \beta$. Es decir, la única diferencia entre las configuraciones f y f' es que algunos nodos que tenían la clasificación α ahora tienen la clasificación β y viceversa, algunos nodos que tenían la clasificación β ahora tienen la clasificación α.

La forma de minimizar la función de energía para todos los nodos es con el siguiente algoritmo:

Algoritmo 7: Swap

1. Iniciar con una configuración f arbitraria.
2. Inicializar variable éxito := 0.
3. **para cada** $\{\alpha, \beta\} \subset \mathcal{L}$ **hacer**
   - 3.1 Encontrar $\hat{f} = argminE(f')$ entre todas las posibles configuraciones de las dos clases escogidas α, β.
   - 3.2 Si $E(\hat{f}) < E(f), entonces f := \hat{f}$ y éxito := 1.

   fin
4. Si éxito = 1, ir al paso 2.

Salida: f.

Denotaremos $\mathcal{P}_{\alpha\beta}$ al conjunto de todos los pixeles en dos clases dadas. En nuestro proceso interno llamaremos a las clases α y β. La gráfica final tiene como vértices la unión de todos los pixeles de ambas clases $P_{\alpha\beta} = P_\alpha \cup P_\beta$ y dos vértices α y β, a los que llamaremos representantes de clase. El conjunto de aristas lo forman las aristas $e_{\{p,q\}}$ que conectan todos los $p \in P$ con sus respectivos vecinos $q \in N_p$ y dos aristas t_p^α, t_p^β por cada pixel que conectan a los representantes de clase α, β. Es decir, cada pixel está conectado directamente por una arista a sus vecinos y a los representantes de cada clase. Un ejemplo de esqueleto de gráfica original y transformada se muestra abajo.

Los pesos de las aristas están definidos por los valores que toman los pixeles P y la configuración f_i actual. Los valores explícitos están definidos en el cuadro 3.1.

Donde $V(\alpha, \beta)$ y $D_p(\cdot)$ tiene la misma interpretación que en el artículo *Iterative MAP and ML*

Figura 3.3: Del lado izquierdo la gráfica original con 3 nodos. Del lado derecho la gráfica generada con los nodos terminales α y β añadidos.

Arista	Peso	Aplicable a
t_p^{α}	$D_p(\alpha) + \sum_{\substack{q \in N_p \\ q \notin P_{\alpha\beta}}} V(\alpha, f_q)$	$p \in P_{\alpha\beta}$
t_p^{β}	$D_p(\beta) + \sum_{\substack{q \in N_p \\ q \notin P_{\alpha\beta}}} V(\beta, f_q)$	$p \in P_{\alpha\beta}$
$e_{\{p,q\}}$	$V(\alpha, \beta)$	$\{p, q\} \in P_{\alpha\beta}$

Cuadro 3.1: Definición de los pesos de las aristas.

Estimations for Image Segmentation [3]:

$$D_p(\iota) = D(p, f_p, \iota) = \| I(p) - \iota \|^2 \tag{3.3}$$

$$V(f_p, f_q) = c \cdot exp\left(\frac{-\Delta(p,q)}{\sigma}\right) \cdot \mathbb{1}_{(f_p \neq f_q)} \tag{3.4}$$

donde $f_p, f_q, I(\cdot)$ tienen la misma interpretación que en el capítulo 2. ι es el representante de una etiqueta en $\{\alpha, \beta\}$. c es una constante y $\Delta(p,q) = | I_L(p) - I_L(q) |$ representa la diferencia de brillo que tienen los pixeles.

Ejemplo 3.1.1. *Tomando el ejemplo 2.5.1, tenemos los valores en los vértices como se muestra en 3.4.*

Suponiendo la configuración inicial siguiente:

	A	B	C
f^0	β	β	α

Los pesos de las aristas de la gráfica se obtienen sustituyendo las ecuaciones 3.4 y 3.3, como lo indica el cuadro 3.1. Los cálculos de cada peso se muestran abajo.

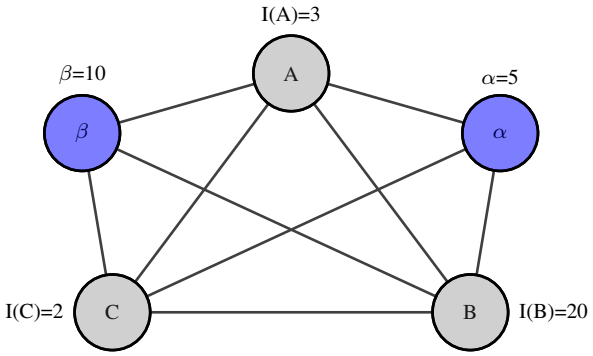


Figura 3.4: Valores obtenidos del ejemplo 2.5.1, posicionados en los vértices correspondientes. En este caso, los valores de α y β son elegidos manualmente.

$$\overrightarrow{\beta A} = t_A^\beta = D_A(\beta) + \sum_{\substack{q \in N_A \\ q \notin P_{\beta\alpha}}} V(\beta, f_q) = || \ 3 - 10 \ ||^2 + 0 = 49$$

$$\overrightarrow{\beta C} = t_C^\beta = D_C(\beta) + \sum_{\substack{q \in N_C \\ q \notin P_{\beta\alpha}}} V(\beta, f_q) = || \ 2 - 10 \ ||^2 + 0 = 64$$

$$\overrightarrow{\beta B} = t_B^\beta = D_B(\beta) + \sum_{\substack{q \in N_B \\ q \notin P_{\beta\alpha}}} V(\beta, f_q) = || \ 20 - 10 \ ||^2 + 0 = 100$$

$$\overrightarrow{\alpha A} = t_A^\alpha = D_A(\alpha) + \sum_{\substack{q \in N_A \\ q \notin P_{\beta\alpha}}} V(\alpha, f_q) = || \ 3 - 5 \ ||^2 + 0 = 4$$

$$\overrightarrow{\alpha C} = t_C^\alpha = D_C(\alpha) + \sum_{\substack{q \in N_C \\ q \notin P_{\beta\alpha}}} V(\alpha, f_q) = || \ 2 - 5 \ ||^2 + 0 = 9$$

$$\overrightarrow{\alpha B} = t_B^\alpha = D_B(\alpha) + \sum_{\substack{q \in N_B \\ q \notin P_{\beta\alpha}}} V(\alpha, f_q) = || \ 20 - 5 \ ||^2 + 0 = 225$$

$$\overrightarrow{AC} = e_{\{A,C\}} = V(\beta, \alpha) = exp(- \mid 5 - 10 \mid) \cdot 1 \approx 0.36$$

$$\overrightarrow{AB} = e_{\{A,B\}} = V(\beta, \alpha) = exp(- \mid 5 - 10 \mid) \cdot 0 = 0$$

$$\overrightarrow{CB} = e_{\{C,B\}} = V(\beta, \alpha) = exp(- \mid 5 - 10 \mid) \cdot 1 \approx 0.36.$$

La gráfica resultante es la que se muestra abajo y que es la misma que la gráfica 1.3, salvo el redondeo de las aristas con decimales.

Ajustando el redondeo, por el ejemplo 1.3.6, vemos cómo la partición que induce el algoritmo

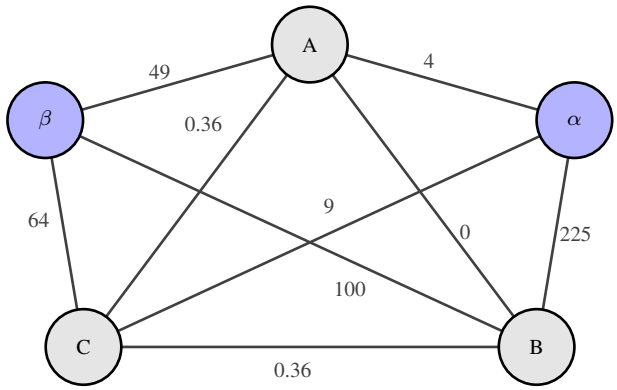


Figura 3.5: Gráfica con nodos terminales α, β añadidos y pesos calculados en cada arista según la tabla 3.1.

es la que define la cortadura mínima: $C_G = \left\{t_A^\alpha, e_{\{A,B\}}, t_C^\alpha, t_B^\beta, e_{\{C,B\}}\right\}$.

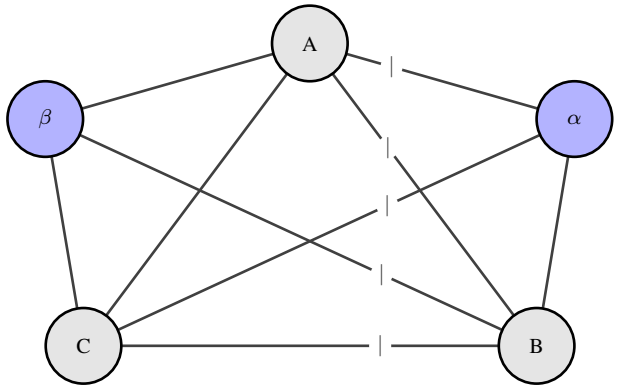


Figura 3.6: Cortadura C_G ilustrada por lineas verticales en las aristas que, al quitarlas, generan la partición buscada de los vértices.

Como $t_A^\alpha, t_B^\beta, t_C^\alpha \in C_G$*, entonces la configuración que minimiza la función de energía es:*

$$\begin{array}{cccc} & A & B & C \\ f' & \alpha & \beta & \alpha \end{array}$$

Que es el mismo resultado que el de la sección 2.5.1.

3.2. Equivalencia gráfica con función de energía

El objetivo de la sección es demostrar que obtener la cortadura mínima de una gráfica con dos clases, como se muestra en la sección 3, es equivalente a minimizar su función de energía.

Como comentario adicional sobre la notación: f_p, f_q son las clases asignadas a p y q respectivamente. f_p^C, f_q^C son las clases asignadas a p y q inducidas por la cortadura C. La notación ayuda a ver si la clasificación en el caso general está dada por una cortadura con un movimiento Swap o la clasificación se mantuvo constante porque los vértices no pertenecían a $P_{\alpha\beta}$. La notación sin superíndice C (f_p, f_q) abarca a los vértices clasificados por la cortadura denotados con C (f_p^C, f_q^C) y por tanto se puede reemplazar la notación con C (f_p^C, f_q^C) por la notación sin C (f_p, f_q).

Los siguientes lemas son necesarios para la demostración.

Lema 3.2.1. *Sea una gráfica G =* $(P_{\alpha\beta} \cup \{\alpha, \beta\}, E)$ *construida según el cuadro 3.1 y* C_G *la cortadura de la gráfica G que separa a los representantes* α *y* β*, entonces:*

$$\mid C_G \cap e_{p,q} \mid = V(f_p^C, f_q^C). \tag{3.5}$$

Demostración. Existen 4 casos:

a) Si $t_p^\alpha, t_q^\alpha \in C$ entonces $e_{\{p,q\}} \notin C$.

b) Si $t_p^\beta, t_q^\beta \in C$ entonces $e_{\{p,q\}} \notin C$.

c) Si $t_p^\beta, t_q^\alpha \in C$ entonces $e_{\{p,q\}} \in C$.

d) Si $t_p^\alpha, t_q^\beta \in C$ entonces $e_{\{p,q\}} \in C$.

La intersección está en los casos con vecinos en diferentes clases, por lo tanto, por construcción

$$\mid C_G \cap e_{p,q} \mid = \mid e_{\{p,q\}} \mid = V(f_p^C, f_q^C) = V(\alpha, \beta).$$ □

La demostración de la equivalencia entre encontrar la cortadura mínima y minimizar la función de energía se basa en que el costo de la cortadura es igual a la función de energía, salvo una constante y esa constante no depende de C.

Teorema 3.2.2. *El costo de una cortadura* C *en* $G_{\alpha\beta}$ *es* $\mid C \mid = E(f^C)$ *más una constante.*

Demostración.

$$\mid C \mid = \sum_{p \in P_{\alpha\beta}} \mid C \cap \left\{ t_p^\alpha, t_p^\beta \right\} \mid + \sum_{\substack{\{p,q\} \in N \\ \{p,q\} \subset P_{\alpha\beta}}} \mid C \cap e_{\{p,q\}} \mid .$$

Para $p \in P_{\alpha\beta}$ tenemos que:

$$\mid C \cap \left\{ t_p^\alpha, t_p^\beta \right\} \mid = \begin{cases} \mid t_p^\alpha \mid, & \text{si } t_p^\alpha \in C. \\ \mid t_p^\beta \mid, & \text{si } t_p^\beta \in C. \end{cases} = D_p(f_p^C) + \sum_{\substack{q \in N_p \\ q \notin P_{\alpha\beta}}} V(f_p^C, f_q). \tag{3.6}$$

Entonces, por la ecuación 3.6 y el lema 3.2.1 tenemos que:

$$\mid C \mid = \sum_{p \in P\alpha\beta} D_p(f_p^C) + \sum_{p \in P_{\alpha\beta}} V(f_p^C, f_q) + \sum_{\substack{\{p,q\} \in N \\ \{p,q\} \subset P_{\alpha\beta}}} V(f_p^C, f_q^C).$$

Notando que

$$\begin{aligned} (N \cap \{p, q \mid p \wedge q \in P\alpha\beta\}) \cup (N \cap \{p, q \mid p \oplus q \in P_{\alpha\beta}\}) \\ = N \cap \{\{p, q \mid p \wedge q \in P_{\alpha\beta}\} \cup \{p, q \mid p \oplus q \in P_{\alpha\beta}\}\} \\ = N \cap \{p, q \mid p \vee q \in P_{\alpha\beta}\} . \end{aligned}$$

Obtenemos la siguiente igualdad:

$$\mid C \mid = \sum_{p \in P_{\alpha\beta}} D_p(f_p^C) + \sum_{\substack{\{p,q\} \in N \\ p \text{ ó } q \in P_{\alpha\beta}}} V(f_p^C, f_q^C).$$

Sea $K = \sum_{p \notin P_{\alpha\beta}} D_p(f_p) + \sum_{\substack{\{p,q\} \in N \\ \{p,q\} \cap P_{\alpha\beta} \neq \emptyset}} V(f_p, f_q)$

$$
\begin{aligned}
\mid C \mid + K &= \sum_{p \in P_{\alpha\beta}} D_p(f_p) + \sum_{\substack{[p,q\} \in N \\ p \text{ ó } q \in P_{\alpha\beta}}} v(f_p, f_q) + \sum_{p \notin P_{\alpha\beta}} D_p(f_p) + \sum_{\substack{\{p,q\} \in N \\ \{p,q\} \cap P_{\alpha\beta} \neq \emptyset}} V(f_p, f_q) \\
&= \sum_{\{p,q\} \in N} V(f_p, f_q) + \sum_{p \in P} D_p(f_p) = E(f^C)
\end{aligned}
$$

donde K es constante para todos los cortes C. □

Corolario 3.2.3. *El mínimo de la función de energía de un movimiento swap es $\hat{f} = f^C$, donde C es la cortadura mínima en $G_{\alpha\beta}$.*

Capítulo 4

Consideraciones Tomadas

Los artículos utilizados como bibliografía tienen un gran enfoque teórico y no expresan explícitamente cómo implementar los métodos que proponen más allá de pseudocódigos. Aquí ilustramos qué consideraciones y métodos fueron ocupados en el código del texto.

4.1. Textura

En el capítulo 2.2, escogimos el **contraste de la textura**, como es citado en el artículo principal [3]. De aquí salen dos consideraciones especiales no explícitas en los artículos [2] [3]:

1. λ_1, λ_2 son calculadas según la clase *Image Processing*, impartida por el autor de [3], *Serge Belongie*.

2. Para la ecuación 2.1, la constante k utilizada para escalar, es calculada empíricamente y no teóricamente. Es decir, de las imágenes utilizadas en la tesis, calculamos de cada pixel la ecuación 2.1 sin la k, luego calculamos el valor de k como el máximo de los valores encontrados para que sirva como constante normalizadora. La k encontrada es $\frac{50}{\sqrt{200}}$.

4.2. Gráficas

En la sección 3.1 se redefine el problema del máximo a posteriori para encontrar la cortadura mínima de una gráfica. Las propiedades de la gráfica son explícitas, solo falta la implementación.

Las gráficas en el código se manejan con la paquetería $NetworkX$, que es una librería especializada para el estudio de gráficas y redes. La librería nos da muchas maneras de definir una gráfica, las dos principales son:

- Matriz de adyacencia con los elementos representando los pesos de las aristas.
- Agregando nodos uno por uno, utilizando el método add_edge y especificando cada peso.

Crear una gráfica utilizando una matriz de adyacencia puede ser más fácil de programar, pero no necesariamente es la manera más eficiente computacionalmente, ya que si tomamos una matriz $M \in M_{n\times m}$, su matriz de adyacencia tiene $n^2 * m^2$ elementos.

Para calcular el número de elementos a definir utilizando el método add_edge, primero hay que considerar que un pixel tiene, a lo más, 4 vecinos: arriba, abajo, derecha e izquierda. Hay que notar también que $2n$ y $2m$ de los pixeles hay que restarles un vecino, ya que están en la orilla de la imagen. Luego, cada arista se debe definir 2 veces, una que vaya del nodo i al j, y otra que vaya del j al i. Entonces, el número de elementos a definir es (| vecinos | $\times$ | pixeles | - |pixeles en orillas|) $\times$ | aristas por vecinos | es decir, $(4 * n * m - 2 * n - 2 * m) * 2$.

Si consideramos una imagen de 500×500 pixeles, que es una imagen chica para los estándares actuales, obtenemos los siguientes elementos para una matriz de adyacencia y para una gráfica definida por el método add_edge respectivamente:

- $500^2 * 500^2 = 62,500,000,000$
- $(4 * 500 * 500 - 2 * 500 - 2 * 500) * 2 = 1,996,000.$

Ambos números son de un orden mayor a 10^6, pero es evidente que, en cuanto a eficiencia de memoria, el segundo método es mejor.

Durante el desarrollo se probó empíricamente ambas propuestas, siendo la segunda la mejor, pues su tiempo de proceso era menor a 10 segundos, mientras que en la primera opción se acababa la memoria sin poder terminar.

4.3. Estimación MAP: $f \mid \Phi$

El artículo *Fast approximate energy minimization via graph cuts* [1] es sin duda, uno de los artículos más importantes de la tesis; junto con el artículo principal [3], explica y demuestra cómo realizar la estimación **MAP** en términos de gráficas.

El algoritmo 7 ilustra el método de estimación de f dado por [1]. En el algoritmo Swap (7), podemos observar que, en el caso donde el número de regiones $\mathcal{L} > 2$, éste no maximiza la probabilidad conjunta, solo la aproxima, maximizando la configuración de 2 en 2 regiones a través de movimientos $Swap$. La manera de abordar la maximización hace que el resultado cambie según el orden en que se escojan los pares de regiones.

La implementación para la tesis usa un orden de regiones simple de seguir: Itera ascendentemente sobre 2 índices i, j , que representan las regiones, y solo procesa los pares que cumplan la condición $\{i < j \mid i, j \in \mathcal{L}\}$. Es decir, si $\mathcal{L} = \{0, 1, 2\}$ entonces el orden sería:

1. $(0, 1)$
2. $(0, 2)$
3. $(1, 2)$.

Capítulo 5

Aportaciones y Resultados

El capítulo detalla los aportes más importantes que se hicieron al algoritmo para mejorar sus tiempos de ejecución y los resultados obtenidos.

5.1. Reducción de regiones

En las pruebas realizadas, el algoritmo de segmentación 6 termina definiendo regiones disconexas gracias al paso 3.2. el cual está diseñado para ir ajustando el número de regiones automáticamente. El problema con dicho enfoque surge con imágenes que no tienen regiones uniformes, aunado a que el algoritmo no tiene un límite sobre cuántas regiones nuevas puede crear.

El costo computacional de generar regiones que no son necesarias es muy alto. Del algoritmo 7 obtenemos que hay que minimizar la función de energía E por cada par de etiquetas $\{\alpha, \beta\}$, es decir, si tenemos n etiquetas, el número de minimizaciones a realizar son $\binom{n}{2} = \frac{n(n-1)}{2}$ lo que implica un crecimiento de orden cuadrático por cada iteración.

Para evitar la generación de muchas nuevas regiones aisladas innecesarias, se agregó un algoritmo después del paso 3.2 del algoritmo 6. El nuevo algoritmo toma las N regiones más grandes y éstas "absorben" las regiones más chicas.

El pseudocódigo del algoritmo 8 muestra la propuesta para reducir el número de regiones.

Después de ejecutar pruebas con las dos implementaciones, i.e., sin y con el algoritmo 8, la diferencia es evidente en una magnitud similar a la esperada; $T \times \binom{\Delta}{2}$, donde T es el tiempo de proceso y Δ es el número de regiones nuevas no sustituidas por el algoritmo 8.

Algoritmo 8: Algoritmo de reducción de regiones

Entrada: Lista de clasificaciones, Imagen, U:= Umbral, N:= Máximo número de regiones

1. Extraer $M := \{i_1, i_2, ..., i_N\}$, las N etiquetas más frecuentes

2. Filtrar M a las etiquetas que cumplan el Umbral, i.e.
$M \leftarrow M - \left\{ i_j \mid \frac{\sum_{p \in P} \mathbb{1}_{(f_p \neq i_j)}}{|P|} \geq U \right\}$

3. Sea $I = \{f_p \mid p \in P\}$ las etiquetas existentes.

para cada $i \in I$ **hacer**
  si i $\notin$ **M:**

  3.1 Extraer la etiqueta más frecuente de los vecinos de $\{p \mid f_p = i\}$. I.e.
  $F = \arg\max_j \mid \{v \mid v \in V_{\{p \mid f_p = i\}}, f_v = j\} \mid$
  3.2 Asignar la etiqueta F a todos los $p \in \{p \mid f_p = i\}$. Es decir, $f_p \leftarrow F$
  $\forall p \in \{p \mid f_p = i\}$

fin

Salida: Múltiples regiones

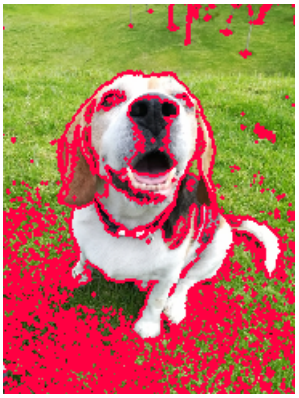
Figura 5.1: Imagen Original sin segmentar.

Figura 5.2: En rojo los bordes de las posibles nuevas regiones.

En la tabla de abajo podemos observar la comparación entre los tiempos de procesamiento para figuras seleccionadas.

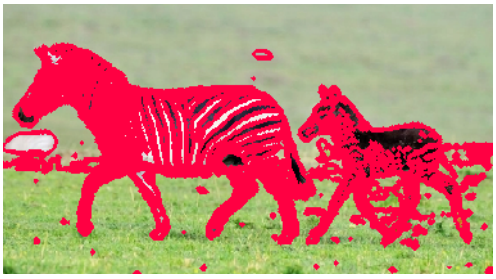
Figura	Minutos **sin** Algoritmo 8	Minutos **con** Algoritmo 8
5.1	3120	12
5.7	2647	35

En todas las pruebas, cuando no usamos la reducción de regiones, la cantidad de regiones resulta intratable, lo que ocasiona tiempos de procesamiento excesivamente largos con resultados no satisfactorios.



Bordes color rojo de todas las regiones generadas al no usar la reducción de regiones en imágen de mascota.

Segmentación generada usando la reducción de regiones propuesta en imagen de mascota.



Bordes color rojo de todas las regiones generadas al no usar la reducción de regiones en imágen de cebras.

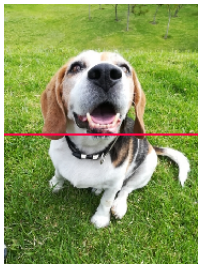
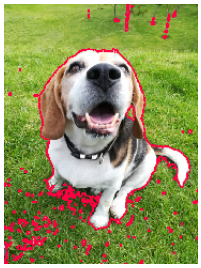
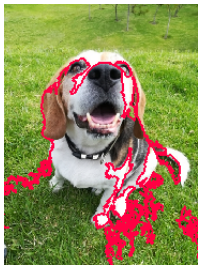
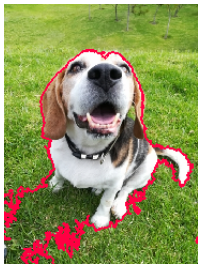
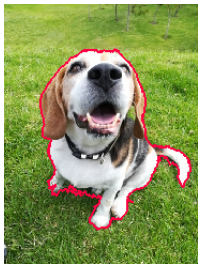
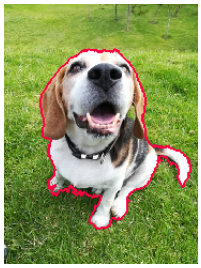
Segmentación generada usando la reducción de regiones propuesta en imagen de cebras.

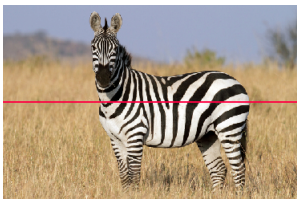
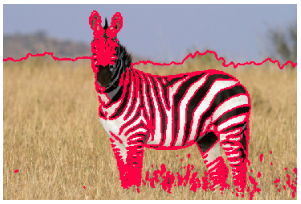
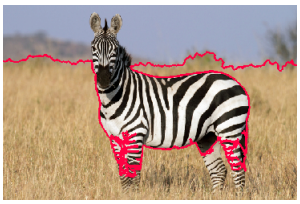
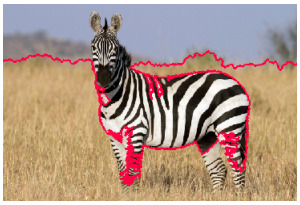
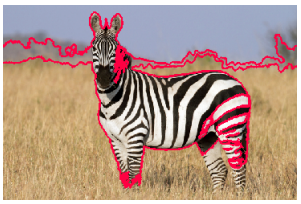
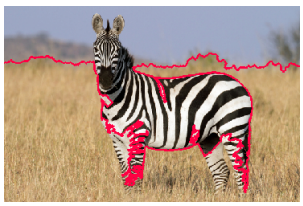
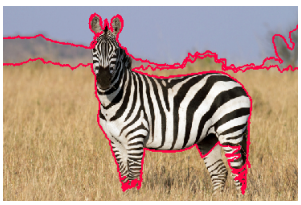
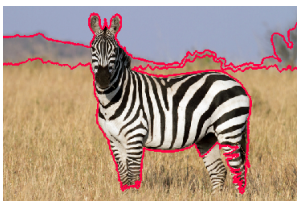
5.2. Pruebas de Convergencia

La motivación de la presente sección surge de las primeras pruebas realizadas al algoritmo. Para algunas imágenes, la configuración inicial que daba el algoritmo de inicialización, junto con el algoritmo 8, era muy cercana a la ideal y el algoritmo principal convergía en una sola iteración. Era inevitable cuestionar la necesidad de todo el algoritmo si la inicialización te deba una configuración muy cercana a la ideal y estaba en duda si el algoritmo principal convergía a una configuración similar al resultado del algoritmo sin inicialización.

Lo que se busca verificar es la capacidad del algoritmo de segmentar, de manera eficiente y correcta, imágenes sin que se tenga una configuración inicial cercana a la correcta. Para ello, sustituimos la configuración dada por el algoritmo de inicialización, por una configuración de pixeles simple, en específico, la mitad superior de la imagen con una etiqueta y la otra mitad con otra etiqueta.

A continuación, se muestran ejemplos del proceso de segmentación sin el algoritmo de inicialización.

Iteración	Sin Algoritmo Inicialización	Con Algoritmo Inicialización
0		
1		
2		
3		
4		NA

Iteración	Sin Algoritmo Inicialización	Con Algoritmo Inicialización
0		
1		
2		
3		
4	NA	
5	NA	

De la última tabla podemos sacar 2 observaciones importantes:

1. La inicialización no garantiza la velocidad de convergencia.
2. Las configuraciones a las que convergen los 2 métodos pueden llegar a diferir de manera importante.

Adicionalmente, si bien las regiones finales no son necesariamente las mismas, al menos hay cierta consistencia en los objetos principales, por ejemplo, en la tabla de arriba, la mayor parte de la cebra siempre está en la misma región.

5.3. Segmentación en diferentes escenarios

Por último, se presentan pruebas más representativas del mundo en diferentes escenarios. El propósito aquí no es ser exhaustivo, si no dar una impresión general de los alcances y limitantes del método.

Figura 5.3: Imagen no segmentada de perros.

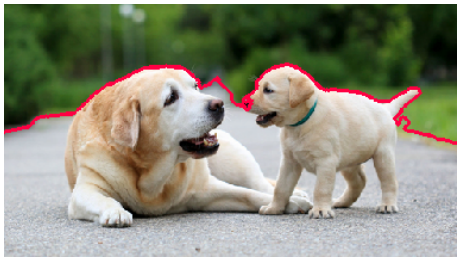


Figura 5.4: Imagen de perros 5.3 Segmentada.

La figura, 5.3, muestra cómo una sola región es dominante y las demás son solo ruido segmentado sin necesidad alguna donde el algoritmo extrae diferentes texturas y se guía mucho por cambios de luz.

En las últimas 2 imágenes se observa el sesgo que el algoritmo tiene por la diferencia en brillo, lo que podría ser mitigado reduciendo la escala a la primera entrada de cada pixel en representación Lab.

En ciertos casos, al tener parámetros de convergencia más estrictos, obtenemos una segmentación mucho más detallada como se ilustra abajo para la imagen 5.8.

Figura 5.5: Imagen Lenna recortada.

Imagen Lenna 5.5 segmentada.

Figura 5.6: Imagen de parque sin segmentar.

Imagen de parque 5.6 segmentada.

Figura 5.7: Imagen de cebras sin segmentar.

Imagen de cebras 5.7 segmentada.

Figura 5.8: Imagen de caballo con jinete sin segmentar.

Imagen de caballo con jinete 5.8 segmentada.

Figura 5.9: Imagen de multitud de personas.

Imagen de multitud 5.9 segmentada.

Figura 5.10: Imagen de varios edificios.

Imagen de edificios 5.10 segmentada.

5.4. Comentarios finales

La modificación propuesta al algoritmo reduce considerablemente los tiempos de procesamiento y al mismo tiempo, mejoramos la calidad de la segmentación. Podemos agregar otros algoritmos basados en heurísticas para mejorar esta implementación.

Otras posibles modificaciones al algoritmo son:

- Compresión de imagen usando algoritmo k-medias para procesar. Dado que la compresión se hace en el espacio de colores y no las dimensiones de la imagen, la segmentación de una es equivalente a la otra.
- Encontrar una función de energía con una complejidad computacional menor.
- Implementar algoritmos en paralelo para encontrar la cortadura mínima.

Apéndice

Código Segmentación de Imágenes

June 3, 2021

1 Librerías

[1]:
```
import sklearn
from sklearn.cluster import KMeans
import os
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from skimage import io, color
import sys
import copy
import time
from IPython.display import display, HTML
```

2 Funciones

2.0.1 Función vecinos

Función utilizada para encontrar los vecinos de un pixel en particular en una imagen con índices numerados de izquierda a derecha y de arriba a abajo.

Input:

1. *Type: int* - índice que representa la posición del pixel.
2. *Type: Int* - 1a dimensión de la imagen (renglones).
3. *Type: int* - 2a dimensión de la imagen (columnas).

Output:

1. *Type: list* - Lista de vecinos.

[2]:
```
def vecinos(index, n, m):
    vecinos = []
    #vecino de arriba
    if index - m >= 0:
        vecinos.append(index - m)
    #vecino de la izquierda
    if (index % m) - 1 >= 0:
```

```
        vecinos.append(index - 1)
    #vecino de la derecha
    if (index % m) + 1 < m:
        vecinos.append(index + 1)
    #vecino de abajo
    if index + m < n*m:
        vecinos.append(index + m)
    return vecinos
```

2.0.2 Función vecinos 5x5

Función para encontrar todos los vecinos a lo más 2 posiciones del pixel de referencia. Es decir, una cuadrícula de 5x5 con pixel central el de referencia. Esto se utiliza para calcular la textura de los pixeles.

Input:

1. *Type: int* - índice que representa la posición del pixel.

2. *Type: Int* - 1a dimensión de la imagen (renglones).

3. *Type: int* - 2a dimensión de la imagen (columnas).

Output:

1. *Type: list* - Lista de vecinos.

[3]:
```
def vecinos5x5(index, n, m):
    vecinos = []
    for i in range(-2,3):
        for j in range(-2,3):
            vecinos.append(index + i*m + j)
    return vecinos
```

2.0.3 Aproximación de pesos Gaussianos

Valores fijos que representan los pesos correspondientes a una gaussiana bivariada en una cuadrícula de 5x5. Estos valores sirven para ponderar el peso al calcular la textura de los pixeles.

[4]:
```
BinGaus=[1,4,6,4,1,4,16,24,16,4,6,24,36,24,6,4,16,24,16,4,1,4,6,4,1]
for i in range(len(BinGaus)):
    BinGaus[i]= BinGaus[i]/256
```

2.0.4 Función gradiente

Gradientes para el cálculo de la textura.

Input:

1. *Type: numpy.ndarrray* - imagen en representación Lab.

2. *Type: Int* - índice del pixel de referencia.

Output:

1. *Type: list* - Lista de gradientes.

[5]:
```
def grads(LabImg, index):
    vecindad=vecinos5x5(index,LabImg.shape[0], LabImg.shape[1])
    gradientes=[]
    for i in range(len(vecindad)):
        gradientes.append(np.array([[(LabImg[(vecindad[i]//LabImg.shape[1]),␣
↪(vecindad[i] % LabImg.shape[1]) + 1, 0] - LabImg[(vecindad[i]//LabImg.
↪shape[1]), (vecindad[i] % LabImg.shape[1])-1 , 0])/2],
        [(LabImg[(vecindad[i]//LabImg.shape[1]) + 1, vecindad[i] % LabImg.
↪shape[1], 0] - LabImg[(vecindad[i]//LabImg.shape[1]) - 1, vecindad[i] % LabImg.
↪shape[1], 0])/(-2)]]))
    return gradientes
```

2.0.5 Función del cálculo de matrix C

Matriz formada por la suma de 25 matrices ponderadas para calcular la textura.

Input:

1. *Type: list* - Lista de gradientes dada por la función grads.

2. *Type: list* - Lista de ponderación de aproximación de pesos gaussianos.

Output:

1. *Type: numpy.array* - Matrix $C \in M_{2\times 2}$.

[6]:
```
def MomentMatrix(Gradientes, GausAprox):
    suma=np.array([[0,0],[0,0]])
    for i in range(len(Gradientes)):
        suma=suma + np.dot(Gradientes[i], np.
↪transpose(Gradientes[i]))*GausAprox[i]
        return suma
```

2.0.6 Función cálculo de la textura de un pixel

Input:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen.

2. *Type: int* - Índice de referencia al pixel.

Output:

1. *Type: float* - textura de un pixel.

[7]:
```
def Textura(Imagen, Indice):
    try:
        a1, a2= np.linalg.eig(MomentMatrix(grads(Imagen, Indice), BinGaus))
    except:
        a1=0
    return np.sqrt(np.sum(a1))*100/np.sqrt(200)
```

2.0.7 Función para añadir textura a la imagen

La función modifica los elementos de una imagen Lab agregando una 4a entrada a la representación de pixeles llamada textura. $M_{n\times m\times 3} \rightarrow M_{n\times m\times 4}$

Input:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen.

Output:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen con una dimensión extra representando la textura.

[8]:
```
def labWithTexture(labImage):
    dim1=labImage.shape[0]
    dim2=labImage.shape[1]
    NewIm=np.zeros((dim1, dim2, 4))
    for i in range(labImage.shape[0]):
        for j in range(labImage.shape[1]):
            NewIm[i][j]= np.append(labImage[i][j], Textura(labImage, i*dim1+ j))
    return NewIm
```

2.0.8 Función para quitar bordes

Función que extrae los bordes de una imagen $M_{n\times m\times d} \rightarrow M_{n-2\times m-2\times d}$.

Input:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen.

Output:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen sin bordes.

[9]:
```
def NoBorders(Img):
    NBImg=Img[3:(Img.shape[0]-3),3:(Img.shape[1]-3),:]
    return NBImg
```

2.0.9 Función inicio de algoritmo

La función inicializa la primera configuración escogiendo las regiones con el algoritmo k-medias.

Input:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen.

2. *Type: int* - Arreglo representante de la imagen.

Output:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen sin bordes.

[10]:
```
def InicAlgo(ImgModif, n):
    vals=ImgModif[0]
    for i in range(1,ImgModif.shape[0]):
        vals=np.concatenate((vals, ImgModif[i]), axis=0)
    labels=KMeans(n_clusters=n, random_state=21).fit_predict(vals)
    try:
        l2=vals[labels==2].mean(axis=0)
        l0=vals[labels==0].mean(axis=0)
        l1=vals[labels==1].mean(axis=0)
    except:
        l0=vals[labels==0].mean(axis=0)
        l1=vals[labels==1].mean(axis=0)

    return labels, l0, l1, l2
```

2.0.10 Función imprimir imagen

La función es una proyección de $M_{m\times n\times 4} \to M_{m\times n\times 3}$ para luego ser impresa.

Input:

1. *Type: numpy.ndarray* - Arreglo representante de la imagen.

Output:

1. Imagen.

[11]:
```
def ImpRep(ImgModif):

    dim1=ImgModif.shape[0]
    dim2=ImgModif.shape[1]
    NewIm=np.zeros((dim1, dim2, 3))
    for i in range(dim1):
        for j in range(dim2):
            NewIm[i][j]= ImgModif[i][j][0:3]

    io.imshow(color.lab2rgb(NewIm))

    return None
```

2.0.11 Función índice a coordenadas

La función toma una lista de índice y regresa una lista de coordenada.

Input:

1. *Type: list* - lista de índices.

Output:

1. *Type: list* - lista de coordenadas.

[12]:
```
def Ind2Cart(arr, dim2):
    coord=[]
    for i in arr:
        coord.append([i//dim2, i%dim2])
    return coord
```

2.0.12 Función algoritmo Swap

Función que aplica el algoritmo Swap a las clases especificadas.

Input:

1. *Type: Int* - Primera etiqueta para seleccionar los pixeles que se tomarán en cuenta para el algoritmo swap.

2.*Type: Int* - Segunda etiqueta para seleccionar los pixeles que se tomarán en cuenta para el algoritmo swap.

3. *Type: numpy.ndarray* - Representante de la etiqueta en la primera entrada del algoritmo actual.

4. *Type: numpy.ndarray* - Representante de la etiqueta en la segunda entrada del alroritmo actual.

5. *Type: numpy.ndarray* - Imagen a la que se le aplicara el algoritmo swap.

6. *Type: list* - Lista de valores con las etiquetas de los pixeles de la imagen ingresada al algoritmo.

Output:

1. *Type: list* - Lista actualizada de etiquetas.

[13]:
```
def Swap(l1,l2, Reps, Img, labels, constant, sigma):

    #verificar si las regiones son conexas
    if VecinosConj4(set(np.where(labels==l1)[0]), Img.shape[0], Img.shape[1]).
↪intersection(set(np.where(etiquetas==l2)[0]))==set([]):
        #print(""Regiones NO conexas"")
        return labels

    #print(""Regiones conexas"")

    #Crear Gráfica
    #print(""CREANDO GRÁFICA..."")
    labelsNew=copy.deepcopy(labels)
    outputGraph=CreateGraph(Img, labelsNew, l1, l2, Reps, constant, sigma)
    #print(""GRÁFICA CREADA"")
```

```
    #Algoritmo de cortadura mínima
    #print(""BUSCANDO CORTADURA MÍNIMA..."")
    cut_value, partition = nx.minimum_cut(outputGraph, "a", "b")

    #print(""CORTADURA MÍNIMA ENCONTRADA"")
    #Encontrar a cuál partición pertenece a para reasignar de manera correcta␣
↪las clases a los pixeles
    #Si en la partición i está ""a"", entonces toda la partición i tiene la␣
↪etiqueta ""b"", o bien, etiqueta l2
    #print(""REETIQUETANDO..."")
    if partition[0].intersection({"a"})==set({"a"}):
        for i in (partition[0]-{"a","b"}):
            labelsNew[i]=l2
        for i in (partition[1]-{"a","b"}):
            labelsNew[i]=l1
    else:
        for i in (partition[0]-{"a","b"}):
            labelsNew[i]=l1
        for i in (partition[1]-{"a","b"}):
            labelsNew[i]=l2
    #print(""REETIQUETADO TERMINADO"")
    return labelsNew
```

2.0.13 Próximos 8 Vecinos (3x3)

Función que regresa los 8 vecinos más próximos; arriba, abajo, derecha, izquierda + arriba izq, arriba der, abajo izq, abajo der.

Input:

1. *Type: int* - Índice del pixel principal.

2. *Type: int* - Dimensión 1 de la imagen, i.e. número de renglones de la imagen.

3. *Type: int* - Dimensión 2 de la imagen, i.e. número de columnas de la imagen.

Output:

1. *Type: list* - lista con a lo más 8 vecinos más cercanos.

[14]:
```
def vecinos8(index, m, n):
    vecinos = []
    ar=False
    ab=False
    iz=False
    de=False
    #vecino de arriba
    if index - m >= 0:
        ar=True
        vecinos.append(index - n)
```

```
    #vecino de la izquierda
    if (index % m) - 1 >= 0:
        iz=True
        vecinos.append(index - 1)
    #vecino de la derecha
    if (index % m) + 1 < m:
        de=True
        vecinos.append(index + 1)
    #vecino de abajo
    if index + m < n*m:
        ab=True
        vecinos.append(index + n)

    #vecino arriba izq
    if ar and iz:
        vecinos.append(index - n - 1)
    #vecino arriba der
    if ar and de:
        vecinos.append(index - n + 1)
    #vecino abajo izq
    if ab and iz:
        vecinos.append(index + n - 1)
    #vecino de abajo der
    if ab and de:
        vecinos.append(index + n + 1)
    return vecinos
```

2.0.14 Vecinos de conjunto

Función que regresa los vecinos de un conjunto.

Input:

1. *Type: set* - Conjunto al que se le van a buscar los vecinos.

2. *Type: int* - Dimensión 1 de la imagen, i.e. cuántos renglones de pixeles tiene la imagen.

3. *Type: int* - Dimensión 2 de la imagen, i.e. cuántas columnas de pixeles tiene la imagen.

Output:

1. *Type: set* - vecinos encontrados del conjunto original. Puede tener intersección con el conjunto original.

[15]:
```
def VecinosConj(conj, m, n):
    conjFinal=set([])
    for i in conj:
        conjFinal=conjFinal.union(set(vecinos8(i,m,n)))
    return conjFinal
```

2.0.15 Vecinos de conjunto (4)

Función que regresa los vecinos de un conjunto.

Input:

1. *Type: set* - Conjunto al que se le van a buscar los vecinos.

2. *Type: int* - Dimensión 1 de la imagen, i.e. cuántos renglones de pixeles tiene la imagen.

3. *Type: int* - Dimensión 2 de la imagen, i.e. cuántas columnas de pixeles tiene la imagen.

Output:

1. *Type: set* - vecinos encontrados del conjunto original. Puede tener intersección con el conjunto original.

[16]:
```
def VecinosConj4(conj, m, n):
    conjFinal=set([])
    for i in conj:
        conjFinal=conjFinal.union(set(vecinos(i,m,n)))
    return conjFinal
```

2.0.16 Pixeles conexos

Función que encuentra todos los pixeles conexos con la misma etiqueta.

Input:

1. *Type: set* - Conjunto de índices conexos a los que se les buscarán todos los pixeles conexos y con la misma etiqueta

2. *Type: set* - Superconjunto de la primera entrada que sirve para rutinas internas del algoritmo. La misma entrada a la anterior basta

3. *Type: list* - Lista de etiquetas de cada pixel

4. *Type: int* - Dimensión 1 de la imagen, i.e. cuántos renglones de pixeles tiene la imagen

5. *Type: int* - Dimensión 2 de la imagen, i.e. cuántas columnas de pixeles tiene la imagen.

Output:

1. *Type: set* - Conjunto de pixeles. conexos con pixi (conjunto de pixeles dados en la primera entrada)

[17]:
```
def ConjuntoConexo(pixi, superpixi, labels, m, n):
    #verify if the pixi set is null
    if pixi == set([]):
        Result = set([])
    else:
        #get any item from pixi set
        for i in pixi:
            aux=i
            break
```

```
        #get neighbours from pixi set
        V=VecinosConj(pixi, m, n)
        #intersect V set with pixels of the same label
        V.intersection_update(set(np.where(np.array(labels)==labels[i])[0]))
        #substrat de pixels already in superpixi
        V.difference_update(superpixi)
        #update output variable
        Result = superpixi.union(V)
        Result = Result.union(ConjuntoConexo(V, superpixi.union(V), labels, m,␣
↪n))
    return Result
```

2.0.17 Reetiquetar pixeles disconexos

Función que reetiqueta pixeles de una misma clase a diferentes clases si son disconexos.

Input:

1. *Type: int* - etiqueta a analizar.
2. *Type: list* - Lista de etiquetas de cada pixel.
3. *Type: int* - Dimensión 1 de la imagen, i.e. cuántos renglones de pixeles tiene la imagen.
4. *Type: int* - Dimensión 2 de la imagen, i.e. cuántas columnas de pixeles tiene la imagen.

Output:

1. *Type: list* - lista de etiquetas actualizada.

[18]:
```
def Reetiqueta(label, labels, m, n):
    #Extraer los índices de los pixeles con la etiqueta label
    Restantes=set(np.where(np.array(labels)==label)[0])
    labelsAux=copy.deepcopy(labels)
    ConteoEtiquetas=0
    #mientras no hayamos reetiquetado (y extraído) los índices de Restantes...
    while Restantes != set([]):
        #extraer un elemento del conjunto Restantes y asignarlo a aux
        for i in Restantes:
            aux=i
            break
        #Obtener conjunto conexo de aux
        NewLabels=ConjuntoConexo({aux}, {aux}, labels, m, n)
        #Validar que no sean conexos
        if NewLabels==Restantes:
            return labelsAux
        #Asignar nuevas etiquetas al subconjunto conexo
        NewLabel=max(labelsAux)+1
        ConteoEtiquetas=ConteoEtiquetas+1
        #print(""Nueva etiqueta: "", NewLabel)
```

```
        for j in NewLabels:
            labelsAux[j]=NewLabel
        Restantes.difference_update(NewLabels)
    print(ConteoEtiquetas, "nuevas etiquetas creadas")
    return labelsAux
```

2.0.18 Gráfica para Swap

Función que crea gráfica para ser utilizada en el algoritmo swap. Dada una imagen, representantes, lista de etiquetas totales y etiquetas te regresa la Gráfica lista para usarse en el algoritmo min_cut.

Los valores de las aristas entre vecinos están definimos por:

$$e_{\{p,q\}} = V(f_p, f_q) = c \cdot exp(\frac{- \mid I_L(p) - I_L(q) \mid}{\sigma})$$

donde σ, c son iguales a 1.

Los pesos de los nodos terminales t_p^α y t_p^β se definen como:

Para t_p^β: $D_p(\beta) + \sum_{\substack{q \in N_p \\ q \notin P_{\alpha\beta}}} V(\beta, f_q)$

Para t_p^α: $D_p(\alpha) + \sum_{\substack{q \in N_p \\ q \notin P_{\alpha\beta}}} V(\alpha, f_q)$

donde

$D_p(\iota) = D(p, f_p, \iota) = \mid\mid I(p) - \iota \mid\mid^2$

$V(f_p, f_q) = c \cdot exp(\frac{-\Delta(p,q)}{\sigma}) \cdot 1_{(f_p \neq f_q)}$.

Input:

1. *Type: narray* - Imagen.
2. *Type: list* - Lista de etiquetas actuales.
3. *Type: int* - Etiqueta 1.
4. *Type: int* - Etiqueta 2.
5. *Type: dic* - diccionario con el número de clase como índice y el representante como valor.

Output:

1. *Type: nx.graph* - objeto gráfica definido en la paquetería networkx.

[19]:
```
def CreateGraph(Image, Labels, labela, labelb, Reps, sigma, constant):

    #L es el conjunto de índices con etiquetas alpha y beta
    L=set(np.where(etiquetas==labela)[0]).union(set(np.
↪where(etiquetas==labelb)[0]))
```

```
   #e es $V(\apha , \beta)$
   e=constant*np.exp(-abs(Reps[labela][3]-Reps[labelb][3])/sigma)

   #vecinos de los alpha intersección beta. Sirve para saber cuáles betas son␣
↪vecinos de la clase alpha
   intersecb = VecinosConj4(set(np.where(etiquetas==labela)[0]), Image.
↪shape[0], Image.shape[1])
   intersecb.intersection_update(set(np.where(etiquetas==labelb)[0]))
   #vecinos de las betas intersección alpha. Sirve para saber cuáles alphas son␣
↪vecinos de la clase beta
   interseca = VecinosConj4(set(np.where(etiquetas==labelb)[0]), Image.
↪shape[0], Image.shape[1])
   interseca.intersection_update(set(np.where(etiquetas==labela)[0]))

   #Asignación a G como gráfica
   G = nx.DiGraph()
   #print(""Primera asignación de nodos e..."")
   #Asignación de aristas (con peso e) a las parejas de pixeles vecinos con␣
↪etiquetas distintas
   for i in interseca:
       for j in vecinos(i, Image.shape[0], Image.shape[1]):
           if Labels[j]!=Labels[i]:
               G.add_edge(i,j, capacity = e)
               G.add_edge(j,i, capacity = e)
   #print(""Primera asignación de nodos e terminada"")

   #print(""Segunda asignación de nodos e..."")
   #Asignación de aristas (con valor 0) a las parejas de pixeles vecinos con␣
↪misma etiqueta
   conjuntoAux=set()
   for i in L:
       conjuntoAux.add(i)
       for j in vecinos(i, Image.shape[0], Image.shape[1]):
           if Labels[i]==Labels[j] and not j in conjuntoAux:
               G.add_edge(j,i, capacity = 0)
               G.add_edge(i,j, capacity = 0)
   #print(""Segunda asignación de nodos e terminada"")

   #print(""Inicio de asignación nodo terminal alpha..."")
   #Asignación de aristas a nodos terminales
   for i in L:
       t= np.linalg.norm(Reps[labela]-Image[i//Image.shape[1], i%Image.
↪shape[1]])
       for j in vecinos(i, Image.shape[0], Image.shape[1]):
           if j not in L:
```

```
                t=t+(constant*np.exp(-abs(Reps[Labels[j]][3] - Reps[labela][3])/
↪sigma))
        G.add_edge("a",i, capacity = t)
        G.add_edge(i,"a", capacity = t)
    #print(""Asignación de nodo terminal alpha terminada"")

    #print(""Inicio de asignación nodo terminal beta"")
    for i in L:
        t= np.linalg.norm(Reps[labelb]-Image[i//Image.shape[1], i%Image.
↪shape[1]])
        for j in vecinos(i, Image.shape[0], Image.shape[1]):
            if j not in L:
                t=t+(constant*np.exp(-abs(Reps[Labels[j]][3] - Reps[labelb][3])/
↪sigma))
        G.add_edge("b",i, capacity = t)
        G.add_edge(i,"b", capacity = t)
    #print(""Asignación de noto terminal beta terminada"")
    return G
```

2.0.19 Encontrar Representantes: Φ^n

Función que calcula los representantes de clase.

Input:

1. *Type: narray* - Imagen (procesada con 4a entrada).

2. *Type: list* - Lista de etiquetas actuales.

Output:

1. *Type: dict* - Diccionario con el número de clase como índice y el representante como valor.

[20]:
```
def CalcReps(Img, Labels):
    print("Estimación ML de Phi")
    shape1= Img.shape[1]
    Phi={}
    for i in np.unique(Labels):
        RepAux=[0,0,0,0]
        #print(""Calculando representante "", i, ""..."")
        for j in np.where(Labels==i)[0]:
            RepAux=RepAux+Img[j//shape1, j%shape1]
        RepAux=RepAux/len(np.where(Labels==i)[0])
        Phi[i]=RepAux
        #print(""Cálculo "", i, "" terminado"")
    print("Estimación Phi terminada")
    return Phi
```

2.0.20 Iteración Swap Todas las clases

Función que calcula la etiqueta tras aplicar el algoritmo swap a todas las clases actuales.

Input:

1. *Type: narray* - Imagen (procesada con 4a entrada).

2. *Type: list* - Lista de etiquetas actuales.

3. *Type: dict* - Diccionario con el número de clase como índice y el representante como valor.

Output:

1. *Type: list* - arreglo con etiquetas resultantes.

[21]:
```
def SwapVsAll(Representatives, Image, Labels, constant, sigma):
    AuxLabels=copy.deepcopy(Labels)
    for i in np.unique(Labels):
        #print(""Swap de "", i, "" vs "", sum(np.unique(Labels)<=i), ""␣
↪etiquetas"")
        for j in np.unique(Labels):
            if i < j:
                #print(""Swap "", i, "" vs "", j)
                AuxLabels=Swap(i, j, Representatives, Image, AuxLabels,␣
↪constant, sigma)
    print("Swap vs all terminado")
    return AuxLabels
```

2.0.21 Función Eliminación de Regiones Aisladas V2

Función que cambia de etiqueta las regiones "aisladas" para que al final queden N regiones conexas. N es el parámetro de inicialización del algoritmo que nos dice el número inicial de etiquetas.

Podemos ver este algoritmo como la rutina que hace que las N regiones más grandes absorban a las más chicas.

Input:

1. *Type: numpy.ndarray* - arreglo etiquetas modificadas después de aplicar el algoritmo "Reetiqueta"

2. *Type: numpy.ndarray* - imagen.

3. *Type: float* - umbral $\in (0,1)$ representando el mínimo de cobertura que tiene que tener una región para ser creada.

4. *Type: int* - número máximo de regiones totales.

Output:

1. *Type: numpy.ndarray* - arreglo de etiquetas corregido.

[22]:
```
def LabelCutV2(Labels1, Img, Threshold, MaxNumberOfRegions):

    #copiamos las etiquetas para irlas modificando
    LabelsAux=copy.deepcopy(Labels1)
    #Sumamos agrupando por etiqueta
    uniques1, counts1 = np.unique(Labels1, return_counts=True)

    #Extraemos las N ocurrencias más grandes
    TopCountsV0 = np.sort(counts1)[::-1][:MaxNumberOfRegions]

    #Nos quedamos con las ocurrencias más grandes que cumplan el umbral mínimo
    TopCounts=np.array([])
    for i in range(len(TopCountsV0)):
        if (TopCountsV0[i]/len(Labels1))>=Threshold:
            TopCounts = np.append(TopCounts, TopCountsV0[i])

    #print(""Top N labels: "")

    TopN = list([])
    #Inicializar variables de control para el próximo loop
    k=0
    h=0
    #loop para extraer el TopN (Las clases más representativas en la imagen)
    for i in TopCounts:
        #variables de control por si existen etiquetas con mismo número de␣
↪pixeles
        if i==h:
            k=k+1
        else:
            k=0
        #agrega a TopN la clase que tenga i ocurrencias
        TopN.append(uniques1[np.where(counts1==i)[0][k]])
        h=copy.deepcopy(i)
    #print(TopN)

    #loop para eliminar las regiones a excepción de las N más representativas
    for j in uniques1:
        #condición para no eliminar una región representativa
        if j not in TopN:

            #Extraer los índices de la región actual
            CurrentSet=set(np.where(Labels1==j)[0])
            #Extraer los vecinos de la región actual
            CurrentNeighbours = VecinosConj4(CurrentSet, Img.shape[0], Img.
↪shape[1])
            #Restar el conjunto original para que los vecinos no sea un␣
↪superconjunto de la región j, sino pixeles que rodeen la región
```

```
            CurrentNeighbours.difference_update(CurrentSet)

            #Sumamos agrupando por etiqueta
            uniques2, counts2 = np.unique(Labels1[list(CurrentNeighbours)],␣
↪return_counts=True)
            #print(uniques2)

            #Extraer, de las etiquetas originales, la que mas se repita
            Label2b = uniques2[np.where(counts2==counts2.max())[0][0]]

            #Reetiquetado de la región aislada
            LabelsAux[list(CurrentSet)]=Label2b
            #print(""Región "", j, "" reetiquetada"")
    return LabelsAux
```

2.0.22 Función Orillas de Regiones

Función destinada a detectar las orillas y sus vecinos para poder representarlos visualmente,

Input:

1. *Type: numpy.ndarray* - arreglo de etiquetas que segmentan una imagen.
2. *Type: int* - Dimesión 1 de la imagen, i.e. Cuántos renglones tiene.
3. *Type: int* - Dimensión 2 de la imagen, i.e. Cuántas columnas tiene.

Output:

1. *Type: set* - arreglo de etiquetas corregido.

[23]:
```
def Orillas(ImageLabels, Dim1, Dim2):
    UniqueLabels=np.unique(ImageLabels)
    output=set()
    for i in UniqueLabels:
        R=set(np.where(ImageLabels==i)[0])
        VR=VecinosConj4(R, Dim1, Dim2)
        VR.difference_update(R)
        #PRUEBA
        VR=VecinosConj4(VR, Dim1, Dim2)
        ##
        output=output.union(VR)
    output=Ind2Cart(output, Dim2)
    return output
```

2.0.23 Función Reseteado de Etiquetas

Función destinada a reetiquetar un arreglo con tal de tener siempre elementos con el menor valor natural posible.

Input:

1. *Type: numpy.ndarray* - arreglo de etiquetas a reetiquetar.

2. *Type: int* - número mínimo de valor de etiqueta.

Output:

1. *Type: numpy.ndarray* - arreglo de etiquetas.

[24]:
```
def relabelLabels(labels2Relabel, minimumLabel):
    distinctLabels=np.unique(labels2Relabel)
    #print(distinctLabels)
    outputArray=np.zeros(len(labels2Relabel))
    labels2Relabel=labels2Relabel+len(distinctLabels)+minimumLabel
    distinctLabels=np.unique(labels2Relabel)
    for i in range(len(distinctLabels)):
        #print(i)
        #print(distinctLabels[i])
        outputArray[labels2Relabel==distinctLabels[i]]=int(i+minimumLabel)
    return outputArray.astype(int)
```

3 PRUEBAS

1. Importar imagen, convertir a L*a*b y calcular textura.

2. Inicializar Φ con K-medias.

3. Optimización iterativa:

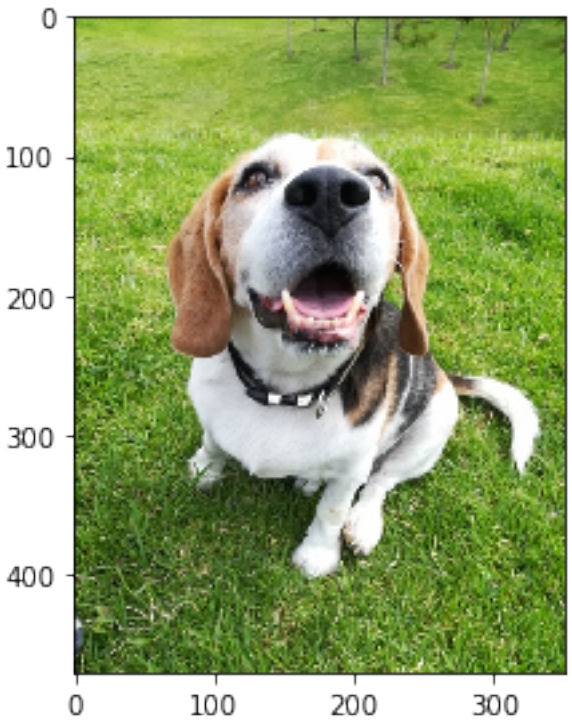
3.1. Estimación MAP: Estimar $f \mid \Phi$.

3.2. Re-etiquedato: Renombrar los segmentos para que no haya 2 regiones disconexas con la misma etiqueta

4. Si no hay cambio en las iteraciones de 3. o el número máximo de iteraciones es alcanzado, finalizar el algoritmo, en caso contrario, regresar al paso 3.

1. Importar imagen, conversión a rgb, a lab y luego calcular textura.

[25]:
```
rgb = io.imread('SmallImage.png')
lab = color.rgb2lab(rgb)

ImgPrueba=NoBorders(labWithTexture(lab))

ImpRep(ImgPrueba)
```



2. Inicializar algoritmo con K-medias.

[26]:
```
t=time.time()
etiquetas, alpha0, alpha1, alpha2 = InicAlgo(ImgPrueba,3)
segundos=time.time()-t
print('segundos de inicialización: %.2f'%(segundos))
```

```
segundos del segundos de inicialización: 1.36
```

[27]:
```
t=time.time()

for i in range(3):
    etiquetas = LabelCutV2(etiquetas, ImgPrueba, 0.05, 4)

segundos=time.time()-t

print('minutos %.2f'%(segundos/60))
```

```
minutos 0.00
```

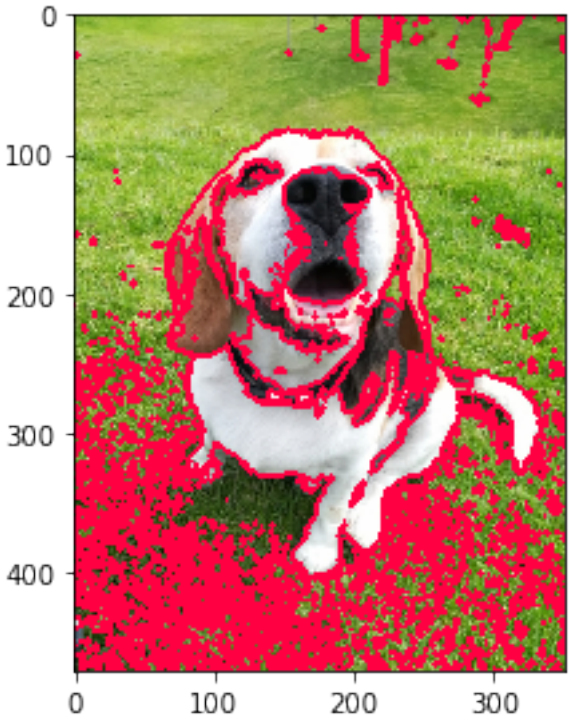
[28]:
```
#guardamos hora de inicio
t=time.time()
#copiamos la imagen original
segmented=copy.deepcopy(NoBorders(lab))
```

```
#extraemos orillas
frontiers=Orillas(etiquetas, ImgPrueba.shape[0], ImgPrueba.shape[1])
#Coloreamos las orillas de rojo en nuestra imagen copiada
for i in frontiers:
    segmented[i[0],i[1],0]=100
    segmented[i[0],i[1],1]=150
    segmented[i[0],i[1],2]=100
#Imprimimos el tiempo de proceso
segundos=time.time()-t
print('minutos: %.2f'%(segundos/60))
```

```
minutos: 2.50
```

[29]:
```
io.imshow(color.lab2rgb(segmented))
```

[29]:
```
<matplotlib.image.AxesImage at 0x1bd2d9b6250>
```



3 & 4. optimización iterativa.

[30]:
```
t=time.time()

etiquetasRef0=copy.deepcopy(etiquetas)
MaxIter=5
```

```
print("Inicio optimización iterativa")
print("Número máximo de iteraciones: " + str(MaxIter))
#número máximo de iteraciones:
for i in range(MaxIter):
    t1=time.time()
    print(" ")
    print("Iteración: ", i)
    representantes=CalcReps(ImgPrueba, etiquetas)
    etiquetas=SwapVsAll(representantes, ImgPrueba, etiquetas, 100, 100)
    if sum(etiquetas!=etiquetasRef0)==0:
        print("sin cambio de etiquetas")
        break
    segundos=time.time()-t1
    print('minutos del Algoritmo Swap: %.2f'%(segundos/60))

    for j in np.unique(etiquetas):
        etiquetas=Reetiqueta(j, etiquetas, ImgPrueba.shape[0], ImgPrueba.
 ↪shape[1])
    print("Iteración ", i, " terminada")

    #reducción de etiquetas
    t1=time.time()
    for w in range(15):
        etiquetas = LabelCutV2(etiquetas, ImgPrueba, 0.05, len(np.
 ↪unique(etiquetasRef0))+2)
        if (len(np.unique(etiquetas))<=(len(np.unique(etiquetasRef0))+2)) and␣
 ↪w>2:
            break
    print("Etiquetas resultantes:")
    print(np.unique(etiquetas))
    #break
    print("Segundos de la reducción de etiquetas:")
    segundos=time.time()-t1


    #Ajuste y reetiquetado para verificar condición de paro
    etiquetasRef1=copy.deepcopy(etiquetas)
    etiquetas=relabelLabels(etiquetas,len(np.unique(etiquetasRef0)))
    createdLabels=set(np.unique(etiquetas))
    previousLabels=set(np.unique(etiquetasRef0))
    for h in previousLabels:
        if len(createdLabels)>0:
            currentMax=0
            label2Replace=-1
            for l in createdLabels:
                if sum(np.logical_and(etiquetasRef0==h, etiquetas==l)) >␣
 ↪currentMax:
```

```
                    currentMax=sum(np.logical_and(etiquetasRef0==h,␣
↪etiquetas==l))
                    label2Replace=l
            if label2Replace!=-1:
                etiquetas[etiquetas==label2Replace]=h
                createdLabels.remove(label2Replace)

    #Definición de umbral de 2%
    print("etiquetas a comparar:")
    print(np.unique(etiquetas))
    print("etiquetas referencia:")
    print(np.unique(etiquetasRef0))
    print("porcentaje de coincidencia: ")
    print(1-sum(etiquetas!=etiquetasRef0)/len(etiquetas))
    if (sum(etiquetas!=etiquetasRef0)/len(etiquetas))<0.02:
        print("Condición de paro al 2% alcanzada en iteración " + str(i))
        print()
        break

    #si sigue, resetear etiquetasRef0 +reetiquetado
    etiquetas=relabelLabels(etiquetas,0)
    etiquetasRef0=copy.deepcopy(etiquetas)
    print("iteración " + str(i) +" terminada")
    #print(time.time()-t)

segundos=time.time()-t
print('minutos totales %.2f'%(segundos/60))

print("Optimización iterativa terminada")
```

```
Inicio optimización iterativa
Número máximo de iteraciones: 5

Iteración:  0
Estimación ML de Phi
Estimación Phi terminada
Swap vs all terminado
minutos del Algoritmo Swap: 8.46
Iteración  0  terminada
Etiquetas resultantes:
[   3 1151 1289]
Segundos de la reducción de etiquetas:
etiquetas a comparar:
[0 1 2]
etiquetas referencia:
[0 1 2]
porcentaje de coincidencia:
```

```
0.6783475783475783
iteración 0 terminada

Iteración:  1
Estimación ML de Phi
Estimación Phi terminada
Swap vs all terminado
minutos del Algoritmo Swap: 9.75
Iteración  1  terminada
Etiquetas resultantes:
[  3 141 357]
Segundos de la reducción de etiquetas:
etiquetas a comparar:
[0 1 2]
etiquetas referencia:
[0 1 2]
porcentaje de coincidencia:
0.9379584166818209
iteración 1 terminada

Iteración:  2
Estimación ML de Phi
Estimación Phi terminada
Swap vs all terminado
minutos del Algoritmo Swap: 8.39
Iteración  2  terminada
Etiquetas resultantes:
[  3 109 828]
Segundos de la reducción de etiquetas:
etiquetas a comparar:
[0 1 2]
etiquetas referencia:
[0 1 2]
porcentaje de coincidencia:
0.9723161786991574
iteración 2 terminada

Iteración:  3
Estimación ML de Phi
Estimación Phi terminada
Swap vs all terminado
minutos del Algoritmo Swap: 7.86
Iteración  3  terminada
Etiquetas resultantes:
[  3 324 453]
Segundos de la reducción de etiquetas:
etiquetas a comparar:
[0 1 2]
```

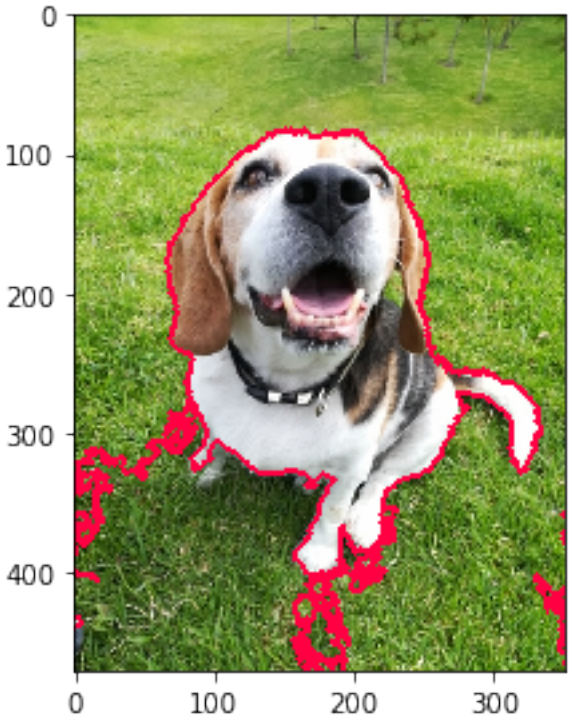
```
etiquetas referencia:
[0 1 2]
procentaje de coincidencia:
0.9800448566406014
Condición de paro al 2% alcanzada en iteración 3

minutos totales 51.22
Optimización iterativa terminada
```

[31]:
```
#guardamos hora de inicio
t=time.time()
#copiamos la imagen original
segmented=copy.deepcopy(NoBorders(lab))
#extraemos orillas
frontiers=Orillas(etiquetas, ImgPrueba.shape[0], ImgPrueba.shape[1])
#Coloreamos las orillas de rojo en nuestra imagen copiada
for i in frontiers:
    segmented[i[0],i[1],0]=100
    segmented[i[0],i[1],1]=150
    segmented[i[0],i[1],2]=100
#Imprimimos el tiempo de proceso
segundos=time.time()-t
print('minutos: %.2f'%(segundos/60))
```

```
minutos: 2.17
```

[32]:
```
io.imshow(color.lab2rgb(segmented))
```

[32]:
```
<matplotlib.image.AxesImage at 0x1bd2e6ee490>
```



[33]:
```
np.unique(etiquetas)
```

[33]:
```
array([0, 1, 2])
```

Bibliografía

[1] Boykov, Y., Veksler, O., Zabih, R. Fast approximate energy minimization via graph cuts. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 11, noviembre de 2001, pp. 1222-39. DOI.org (Crossref), doi:10.1109/34.969114.

[2] Carson, C., Bolongie, S., Greenspan, H. Malik, J. Blobworld: Image Segmentation Using Expectation-Maximization and Its Application to Image Querying. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, no. 8, agosto de 2002, pp. 1026-38. DOI.org (Crossref), doi:10.1109/TPAMI.2002.1023800.

[3] Chen, S., cao, L., Liu, J., Tang, X. Iterative MAP and ML Estimations for Image Segmentation. IEEE Computer Vision and Pattern Recognition, 2007, pp. 1-6, doi:10.1109/CVPR.2007.383007.

[4] Cormen, T., Leiserson, C., Rivest, R., Stein, C. Introduction to algorithms. 3rd ed, MIT Press, 2009.

[5] Geman, S., Donald, G. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-6, no. 6, noviembre de 1984, pp. 721-41. DOI.org (Crossref), doi:10.1109/TPAMI.1984.4767596.

[6] Gonzalez, R., Woods, R. Digital image processing. 2nd ed, Prentice Hall, 2002.

[7] Greig, D., Porteous, B. Seheult, A. Exact Maximum A Posteriori Estimation for Binary Images. Journal of the Royal Statistical Society: Series B (Methodological), vol. 51, no. 2, enero de 1989, pp. 271-79. DOI.org (Crossref), doi:10.1111/j.2517-6161.1989.tb01764.x.

[8] Hernández, M. Introducción a la programación lineal. UNAM, 2010.

[9] Hernández, M. Introducción a la teoría de redes. 2. ed, Sociedad Matemática Mexicana, 2005.

[10] Li, S. Markov Random Field Modeling in Computer Vision. Springer Japan, 1995. Open WorldCat, http://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=3099350.

[11] Parker, A. In the Blink of an Eye. Basic Books, 2004.

[12] Rincón, L. Curso intermedio de probabilidad. UNAM, Facultad de Ciencias, 2007.

[13] Tan, P., Steinbach, M., Kumar, V. Introduction to Data Mining. Pearson Education, 2006.