Facultad de Matemática, Astronomía, Física y Computación

Universidad Nacional de Córdoba

Estudio de algoritmos para detección de bordes en imágenes SAR satelitales basados en la Divergencia de Jensen Shannon.

Tesis para obtener el título de
Licenciado en Ciencias de la Computación

Autor: Matias Nahuel Pereyra
Director: Dr. Sergio Masuelli

Córdoba, Argentina 2023

Dedicación

A mis padres, hermanos y amigos, quienes me han apoyado incondicionalmente en cada etapa de mi vida y han sido un pilar fundamental en mi formación. A mi pareja Estefanía, por su amor, paciencia y apoyo constante, incluso antes de iniciar mi carrera. A mis adorables mascotas, por su compañía y presencia reconfortante en cada momento de mi investigación. Y por último, pero no menos importante, a mis excepcionales profesores de la Facultad de Matemática, Astronomía y Física (FaMAF), en especial a mi director de tesis, Sergio Masuelli, por su invaluable orientación, paciencia y sabiduría, sin la cual este logro no habría sido posible. Dedico este trabajo con todo mi amor y agradecimiento hacia ustedes.

Resumen

La detección de bordes en imágenes SAR es un tema de interés en diversos campos. El ruido "speckle" dificulta esta tarea, ya que afecta la precisión de la detección. Aunque se ha mejorado la detección de bordes mediante la aplicación de un algoritmo basado en la **Divergencia de Jensen-Shannon (JSD)**, ya sea en su versión 1D o 2D, aún no se ha logrado solucionar completamente este problema. El uso de este algoritmo todavía introduce una cantidad significativa de ruido y no detecta los bordes de manera detallada.

Para abordar esta situación, hemos implementado un enfoque que busca identificar los puntos que realmente conforman el borde que queremos detectar, distinguiéndolos de los puntos de ruido introducidos por el algoritmo de Jensen-Shannon.

Para lograr esto, aplicamos un filtrado repetido a la imagen e interpretamos la imagen JSD como un grafo. Luego, convertimos este grafo en un árbol utilizando el algoritmo de **Union Find Kruskal**. A continuación, seleccionamos el punto inicial y final del borde de la superficie que queremos detectar. Por último, utilizamos el algoritmo de **Depth First Search (DFS)** para encontrar un camino que los conecte y graficamos estos puntos en la imagen para una mejor comprensión visual.

El resultado obtenido cumple con nuestras expectativas. Mediante el uso de los parámetros de filtrado adecuados, la detección de bordes es detallada y proporciona información valiosa sobre la superficie que estamos analizando.

Summary

Edge detection in SAR images is a topic of interest in various fields. The noise "speckle" makes this task difficult because it affects the accuracy of the detection. Although edge detection has been improved by applying an algorithm based on **Jensen's Divergence-Shannon (JSD)**, either in its 1D or 2D version, this problem has not been completely solved. Using this algorithm still introduces a significant amount of noise and does not detect edges in detail.

To address this situation, we have implemented an approach that tries to identify the points that really make up the edge we want to detect, distinguishing them from the noise points introduced by the Jensen-Shannon algorithm.

To achieve this, we apply iterative filtering to the image and interpret the JSD image as a graph. We then convert this graph into a tree using the **Union Find Kruskal** algorithm. Next, we select the start and end points of the surface edge we want to detect. Finally, we use the **Depth First Search (DFS)** algorithm to find a path that connects them, and we plot these points on the image for better visual understanding.

The result obtained corresponds to our expectations. By using the appropriate filtering parameters, the edge detection is detailed and provides valuable information about the surface we are analyzing.

Índice

1. Introducción

En este proyecto trabajaremos con imágenes de tipo *Synthetic Aperture Radar* (SAR).

SAR es un tipo especial de radar que permite obtener imágenes de alta resolución a larga distancia utilizando la tecnología de teledetección. Son muy útiles para medir distancia y proporcionan imágenes independientes de la luz solar y que no se ven afectadas por la capa de nubes [2].

La obtención de las imágenes consiste en procesar, mediante algoritmos, la información capturada por la antena del radar. Combinamos la información obtenida en varios barridos de la antena para recrear un solo "barrido virtual" y así proporcionar un rendimiento equivalente al de una antena mucho más grande [3].

Las imágenes SAR presentan un ruido multiplicativo característico, conocido como *speckle*. Esto hace muy difícil la detección de bordes entre zonas homogéneas en comparación con imágenes ópticas. En las imágenes SAR, las zonas homogéneas presentan alta variabilidad en sus valores píxel a píxel, pero al tomar un entorno alrededor de los píxeles la distribución estadística no cambia. Por eso utilizamos la divergencia de **Jensen Shannon** (JSD) [1].

La JSD es un método no paramétrico que nos permite determinar el grado de similitud entre las distribuciones estadísticas de dos conjuntos de datos. En nuestro caso, aplicamos la JSD para la detección de bordes en imágenes SAR de satélites de témpanos antárticos. Sin embargo, los puntos candidatos a formar parte de un borde (valores altos de JSD) pueden estar dispersos, no ser contiguos o formar aglomeraciones, lo que dificulta el trazado de la línea de borde.Además, también pueden aparecer difusos entre puntos de ruido, lo que añade otro desafío en el proceso de detección de bordes [5] [6].

Para solucionar este problema, aplicamos filtros más específicos con el objetivo de reducir la cantidad de puntos candidatos. Además, interpretamos las aglomeraciones de puntos candidatos como grafos. De esta manera, representamos toda la imagen resultante de JSD como un grafo no conexo compuesto por N componentes [7].

Una vez obtenido el grafo de toda la imagen, aplicamos el algoritmo de **Union Find Kruskal's**. Esto nos permite convertir el grafo no conexo en un bosque, es decir, un grafo no conexo donde todas las componentes son árboles [8] [9].

Por último, obtenemos un camino entre 2 puntos dados (considerados como puntos extremos del borde a identificar) mediante el algoritmo de exploración de grafos **Depth First Search (DFS)**. De esta forma, obtenemos la información coordenada a coordenada (punto a punto) del borde a identificar [10] [11].

1.1. Objetivos

El objetivo principal es estudiar y desarrollar una herramienta para detectar bordes en imágenes satelitales SAR de témpanos antárticos basados en la divergencia de Jensen Shannon (JSD). Además, analizaremos la estructura de datos para conjuntos disjuntos de grafos y aplicaremos algoritmos de grafos, como algoritmos de exploración, para abordar el problema original.

Para lograr estos objetivos, analizaremos detalladamente un caso de estudio y realizaremos el procedimiento necesario posterior a la obtención de la información para el procesamiento de detección de bordes, obteniendo así el resultado final.

1.2. Importancia e impacto

Dentro de la detección de bordes, algo de mucho interés es lograr un seguimiento de los desprendimientos de hielo antártico. Es posible realizar comparaciones entre diferentes imágenes en el tiempo del mismo hielo antártico para obtener más detalle y análisis sobre su crecimiento o disminución. Además, mediante la detección de bordes automática se puede ayudar en el descubrimiento de nuevos desprendimientos de hielo.

Otra utilidad de la detección de bordes es la obtención de indicadores sobre el cambio climático en curso al realizar comparaciones de los frentes de glaciares.

Todas estas comparaciones son altamente eficientes, ya que al contar con información detallada sobre el borde de la superficie a analizar, podemos detectar cambios con gran facilidad y rapidez, sin importar si el cambio es mínimo, ya que cualquier cambio en la superficie impactará directamente en el borde resultante.

1.3. Estructura de la Tesis

La estructura de la tesis consta de seis capítulos. El Capítulo 2 proporciona detalles sobre los datos utilizados y el procesamiento aplicado. En el Capítulo 3 se describen los algoritmos utilizados en el procedimiento y se mencionan las librerías necesarias para su ejecución. El Capítulo 4 presenta el procedimiento completo para la detección del borde, desde la obtención de la imagen JSD generada a partir de los datos SAR hasta llegar al resultado final. En el Capítulo 5 se muestran diversos casos de ejecución y se presentan los resultados obtenidos. Finalmente, el Capítulo 6 incluye las conclusiones, los trabajos futuros y las posibles mejoras que podrían implementarse en la detección de bordes.

2. Datos utilizados y procesamiento aplicado

En esta sección se proporciona información básica sobre la imagen utilizada para la detección de bordes. Se inicia con una breve descripción del problema del *speckle* en imágenes SAR, seguida de una explicación sobre el funcionamiento del algoritmo JSD. Además, se describe el origen de la imagen específica utilizada en este trabajo.

2.1. Características de las imágenes SAR

Todas las imágenes SAR presentan un ruido multiplicativo conocido como *speckle* [1][2], especialmente las imágenes SAR SLC (single look complex) de intensidad. Estas estas imágenes contiene regiones homogéneas con una desviación estándar (σ) cercana a la media (μ), lo que significa que son muy ruidosas. Para reducir este ruido, los proveedores de imágenes utilizan una técnica que consiste en subdividir la información adquirida en diferentes ángulos (*looks*) y promediarlas. A medida que se aumenta el número de estas imágenes, el ruido de tipo *speckle* disminuye [3].

Para este proyecto, utilizamos un recorte de la imagen SAR que se muestra en la Figura 1. Esta imagen representa un iceberg considerado como el más grande del mundo. Se trata de un desprendimiento de la plataforma de hielo de Ronne en el mar de Weddell, con una longitud aproximada de 170 km y una anchura de 25 km. La imagen fue capturada por un satélite el día 03/04/2023 [4].

Figura 1: SAR Original

2.2. JSD y su aplicación a detección de bordes

En este proyecto, exploramos la aplicación del JSD (*Jensen-Shannon Divergence*) a la detección de bordes en imágenes SAR. El JSD es una medida de distancia que se utiliza comúnmente en estadística para comparar dos distribuciones de probabilidad. En el contexto de la detección de bordes, utilizamos el JSD para evaluar la diferencia entre la distribución de intensidad de píxeles en dos regiones adyacentes de una imagen SAR y, por lo tanto, detectamos posibles bordes entre ellas.

Contamos con 2 tipos de JSD que aplicamos en este proyecto:

- **JSD de 1 dimensión**: funciona como un filtro de bordes clásico que detecta bordes transversales a la dirección de barrido. Para obtener bordes de cualquier orientación, aplicamos el JSD de manera independiente por fila y por columna, y luego combinamos los valores tomando el

máximo obtenido en cada fila y columna. De esta manera, obtenemos una sola imagen de JSD que contiene información de bordes en todas las direcciones [5].

- **JSD de 2 dimensiones**: se basa en el cálculo de la JSD, pero utiliza una geometría de imagen para tomar muestras. En esta técnica, tomamos muestras a lo largo de la imagen en dos dimensiones, en lugar de hacerlo de manera independiente por filas y columnas. Esto nos permite detectar bordes en todas las direcciones y obtener una imagen de JSD más precisa y detallada [6].

Ambos métodos se evaluarán y compararán para determinar cuál es el más adecuado para la detección de bordes en imágenes SAR.

2.2.1. Algoritmo JSD de 1 dimensión

Se le llama JSD la distancia entrópica en el espacio de probabilidad. Dadas dos funciones de densidad de probabilidad f1 y f2, la JSD se define como:

$$D\left[f_1, f_2\right] = H\left[\Pi_1 f_1 + \Pi_2 f_2\right] - \Pi_1 H\left[f_1\right] - \Pi_2 H\left[f_2\right] \tag{1}$$

donde Π_1 y Π_2 son pesos arbitrarios restringidos a $\Pi_1 + \Pi_2 = 1$ y

$$H\left[f\right] = -\int dy\ f\left(y\right) \ln\left[f\left(y\right)\right] \tag{2}$$

es la entropía de Gibbs-Shannon para una variable aleatoria de rango continuo con PDF $f\left(y\right)$.
satisface: $[f_1, f_2] \geq 0$, con igualdad si y solo si $f_1 = f_2$. De esta forma D proporciona una medida de disimilitud entre f_1 y f_2.

Esta propiedad permite el desarrollo de un método de detección de bordes en imágenes SAR. Dados dos conjuntos con valores ν, la diferencia estadística entre ellos se evalúa mediante JSD. Para realizar el cálculo, las densidades de probabilidad f_i se aproximan mediante el método de kernel de densidad no paramétrica [5], obteniendo así las densidades de probabilidad estimadas $\hat{f}_i$. De esta forma, el método no depende de ningún modelo particular asumido para las distribuciones de probabilidad, lo que lo hace más generalizable. Con esta aproximación:

$$D\left[f_1, f_2\right] \simeq \frac{1}{2\nu}\left\{\sum_{l=1}^{\nu} \ln\left[\frac{2\hat{f}_1\left(y_l\right)}{\hat{f}_1\left(y_l\right) + \hat{f}_2\left(y_l\right)}\right] + \sum_{j=1}^{\nu} \ln\left[\frac{2\hat{f}_2\left(y_j\right)}{\hat{f}_1\left(y_j\right) + \hat{f}_2\left(y_j\right)}\right]\right\} \tag{3}$$

donde la primera suma se calcula con los valores del primer conjunto y la segunda suma con los

valores del segundo conjunto.

Este algoritmo básico se aplica a conjuntos de datos arbitrarios. Cuando lo aplicamos a una secuencia de datos (por ejemplo, un arreglo unidimensional), el valor máximo de JSD corresponde al punto donde las secuencias son más diferentes, lo cual es equivalente al sentido de un borde. Para obtener una imagen, debemos elegir una dirección de barrido, por ejemplo, por columnas, y luego ejecutarlo sobre todos los puntos de la imagen.

2.2.2. Algoritmo JSD de 2 dimensiones

El algoritmo JSD 2D se basa en el 1D y se diferencia fundamentalmente en cómo tomamos los conjuntos de datos a comparar con la JSD. Para tomar los datos, utilizamos un cuadrado de 9 puntos en cada lado, que se divide en cuatro triángulos iguales con el punto de evaluación en su centro, como se muestra en la Figura 2.

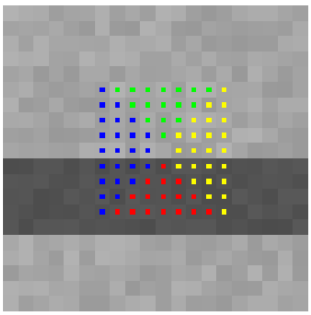


Figura 2: Cuatro conjuntos de puntos involucrados en el JSD 2D, los cuales se distinguen por colores y se encuentran en el centro de los píxeles.

Para el cálculo del JSD en el algoritmo JSD 2D, se comparan los conjuntos de datos contenidos en los triángulos geométricamente opuestos, es decir, el triángulo superior con el inferior y el triángulo derecho con el izquierdo. De esta manera, se obtienen dos valores, de forma similar al barrido por filas y columnas que se aplica en el algoritmo JSD 1D. Es importante destacar que el punto central no forma parte de ninguno de los triángulos comparados. Además, los puntos de las diagonales forman parte de ambos triángulos contiguos, lo que resulta en un conjunto de 24 puntos en cada comparación, similar al número de puntos utilizados en el algoritmo 1D (25 puntos). Con este cambio geométrico en la colección de muestras, esperamos obtener una transición más nítida de los máximos JSD en comparación con el barrido por columna o fila original. Esto se debe a que mover un punto en la dirección transversal de un borde en este algoritmo afecta a 3 puntos, en comparación con solo 1 en las secuencias. Además, su geometría más compacta debería facilitar la captura de pequeñas subestructuras en la imagen, como capas de hielo marino y grietas, lo que resulta útil para nuestra

aplicación [6].

2.3. Características de la imagen a procesar

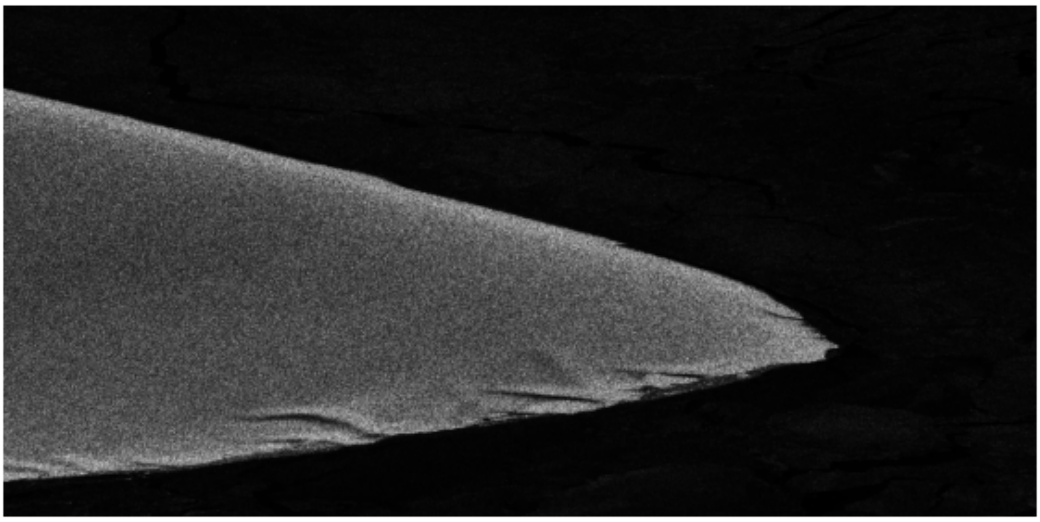
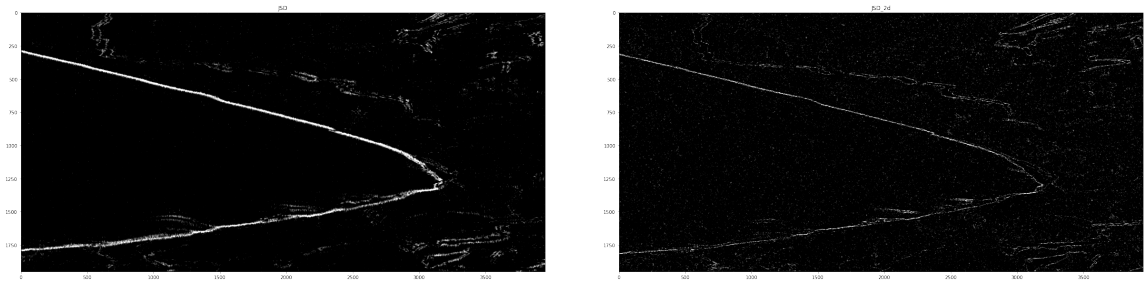


Figura 3: Imagen de entrada SAR

En la Figura 3 se muestra el recorte de la imagen anteriormente mencionada (Figura 1), sobre la cual trabajaremos en esta tesis. En ella se puede observar claramente el ruido presente en la imagen. Antes de poder iniciar el procedimiento adecuado para la detección de bordes, es necesario implementar el algoritmo JSD_{1d} o JSD_{2d} para identificar los puntos candidatos a pertenecer a los bordes, tal como se muestra en las siguientes figuras.

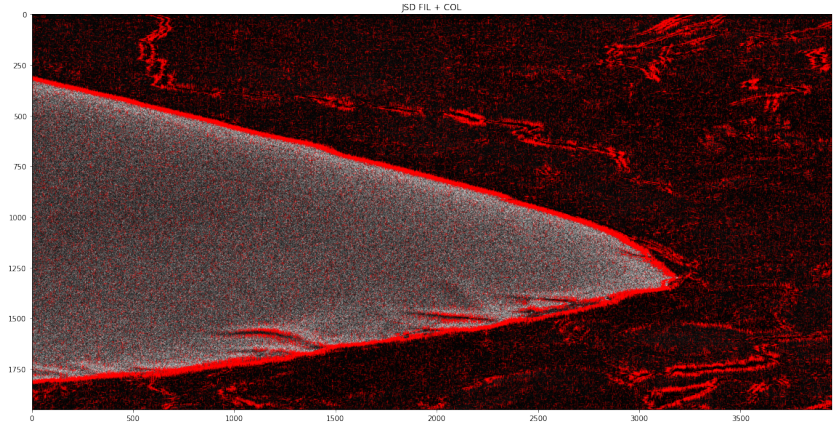


(a) Imagen JSD_{1d} (b) Imagen JSD_{2d}

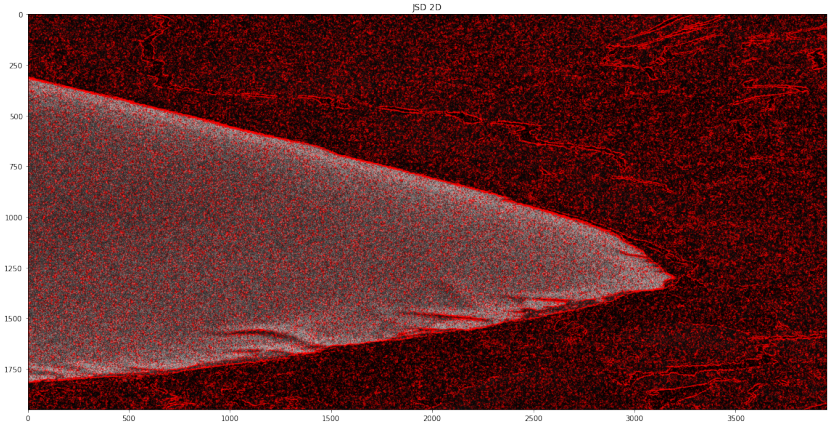
Figura 4: Imágenes JSD

La Figura 4a, que corresponde al resultado del algoritmo JSD de 1 dimensión (JSD_{1d}), y la 4b, que corresponde al resultado del algoritmo JSD de 2 dimensiones (JSD_{2d}). Estas imágenes presentan una diferencia más marcada entre los puntos candidatos y los no candidatos, lo que permitirá, junto con un filtro, reducir el número de candidatos y aplicar adecuadamente la detección de bordes.

Esta diferencia es aún más notable si superponemos las imágenes JSD_{1d} y JSD_{2d} sobre el recorte de la imagen SAR. En la superposición, los puntos de las imágenes JSD_{1d} y JSD_{2d} se resaltan en rojo, mientras que la imagen SAR utiliza una escala de grises para sus puntos.



(a) Imagen SAR y JSD_{1d}



(b) Imagen SAR y JSD_{2d}

Figura 5: Imágenes SAR y JSD

La Figura 5a muestra la imagen SAR original con la superposición de la imagen JSD_{1d}, mientras que en la Figura 5b se puede observar la superposición de la imagen JSD_{2d} en la misma imagen SAR original.

En las zonas más oscuras (agua) de las Figuras se pueden observar agrupaciones de puntos rojos formando aglomeraciones de puntos ligeras. Esto se debe a que el agua contiene el deshielo del iceberg, lo que genera una variación en la temperatura del agua que es detectada e interpretada por el satélite con diferentes niveles de intensidad. Posteriormente, esto es detectado por el algoritmo JSD como un borde. Sin embargo, para nuestro caso de estudio, consideraremos estos bordes detectados por la JSD como ruido, ya que interfieren en la obtención del borde de la superficie buscado. Es importante destacar que estos bordes no son realmente ruido, sino información sobre otras sub-superficies o el deshielo desprendido del iceberg.

3. Algoritmos básicos, librerías y funciones utilizadas

En este trabajo, aplicamos tres algoritmos básicos para la detección y unión de los puntos de borde: **Union Find**, **Kruskal's** y **Depth First Search (DFS)**. Estos algoritmos son ampliamente utilizados en la resolución de problemas de grafos y estructuras de datos [8].

El algoritmo **Union Find** se utiliza para encontrar conjuntos disjuntos en un conjunto de elementos y para unir dos conjuntos en uno solo. **Kruskal's** es un algoritmo para encontrar el árbol de expansión mínima en un grafo conexo, es decir, el subconjunto de aristas que conecta todos los vértices del grafo de manera que la suma de sus pesos sea mínima. Por último, **DFS** es un algoritmo de búsqueda que se utiliza para recorrer o buscar elementos en un grafo o estructura de datos.

A pesar de su simplicidad, estos métodos son esenciales en la resolución de muchos problemas y son la base para algoritmos más avanzados.En este trabajo, exploraremos la implementación y el uso de estos algoritmos, así como sus ventajas y limitaciones. Además, combinaremos **Union Find** y **Kruskal's** en una sola implementación para aprovechar sus ventajas y superar sus limitaciones.

3.1. Union Find Kruskal's

Dado un grafo $G = (V, E)$, donde V representa el conjunto de vértices o nodos en el grafo y E representa el conjunto de aristas o conexiones entre los vértices. El peso de la arista $e \in E$, denotado como *weight*, representa el costo asociado a dicha arista. Se busca encontrar el árbol de expansión mínimo (MST, por sus siglas en inglés de "minimum spanning tree") en el grafo, el cual puede no ser único. Un árbol de expansión mínimo es un subconjunto del grafo que conecta todos los nodos del grafo con el mínimo costo total de aristas basado en sus pesos [7].

El algoritmo comienza con un conjunto de todos los nodos del grafo y un conjunto de aristas vacío. Luego, las aristas se ordenan en orden creciente según su peso. A continuación, se procede a agregar las aristas ordenadas al conjunto de aristas vacío, una por una, en el orden establecido. Se asigna un grupo a cada nodo conectado, representado por los nodos $n1$ y $n2$. Si al añadir una nueva arista se conectan dos nodos que ya pertenecen al mismo grupo, se descarta la arista y se pasa a la siguiente. Este proceso se repite hasta que todos los nodos pertenezcan al mismo grupo o se hayan recorrido todas las aristas. Si el grafo es no conexo y tiene N componentes, habrá N grupos de nodos distintos [9].

3.1.1. Ejemplo Union Find Kruskal's

En la Figura 6, tenemos un grafo $G' = (V', E')$,

$V' = \{A, B, C, D, E, F, G, H\}$,

$E' = \{(A, B, weight = 1), (A, C, weight = 1), (A, F, weight = 1), (B, C, weight = 2),$
$(F, E, weight = 2), (D, C, weight = 3), (B, D, weight = 5), (H, G, weight = 8)\}$

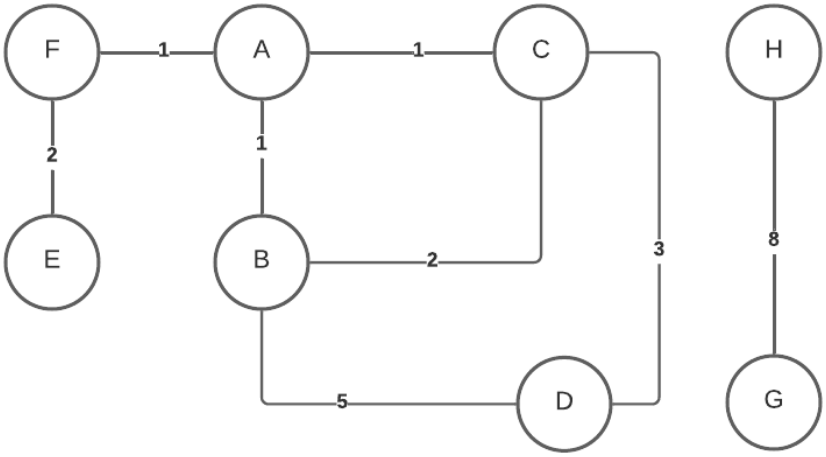


Figura 6: Grafo Inicial

Se puede observar que contamos con un grafo no conexo de 2 componentes, 8 nodos y 8 aristas. Las aristas ya están ordenadas de forma creciente en el conjunto E', por lo que empezamos la ejecución del algoritmo de **Union Find Kruskal's**.

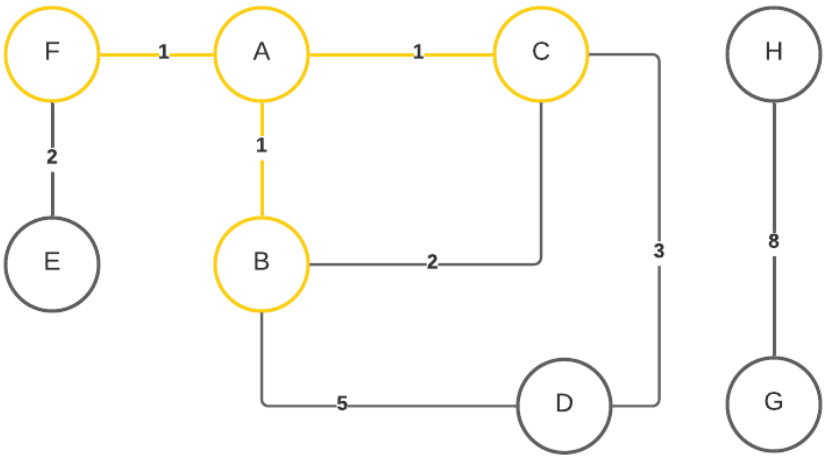


Figura 7: Grafo Kruskal's paso 1

Para la Figura 7, conectamos la arista "$(A, B, \text{peso} = 1)$" y agregamos los nodos A y B al mismo grupo. Repetimos el proceso con las aristas "$(A, C, \text{peso} = 1)$" y "$(A, F, \text{peso} = 1)$". Como los nodos C y F no pertenecen a ningún grupo, los agregamos al mismo grupo que los nodos A y B, unificándolos.

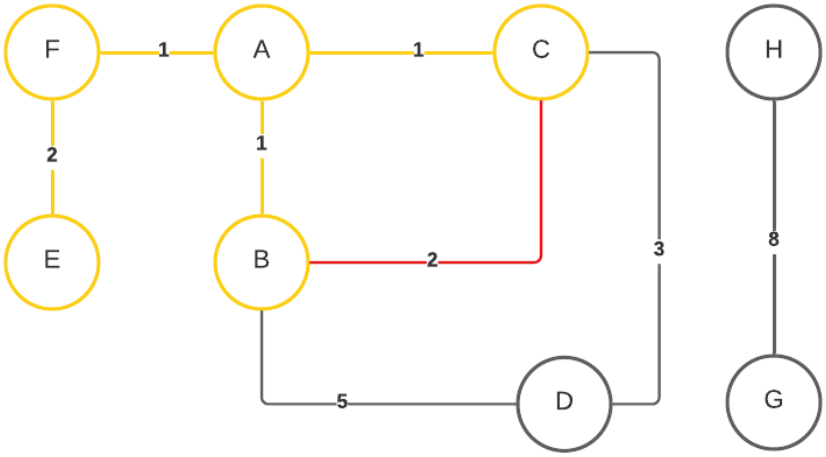


Figura 8: Grafo Kruskal's paso 2

Para la Figura 8, conectamos la arista "$(F, E, \text{peso} = 2)$". Como el nodo E no pertenece a ningún grupo, lo incluimos en el mismo grupo que el nodo F. Luego, conectamos la arista "$(B, C, \text{peso} = 2)$". Esto nos genera un conflicto, ya que los nodos B y C pertenecen al mismo grupo. Por lo tanto, descartamos esta arista y continuamos con las siguientes.

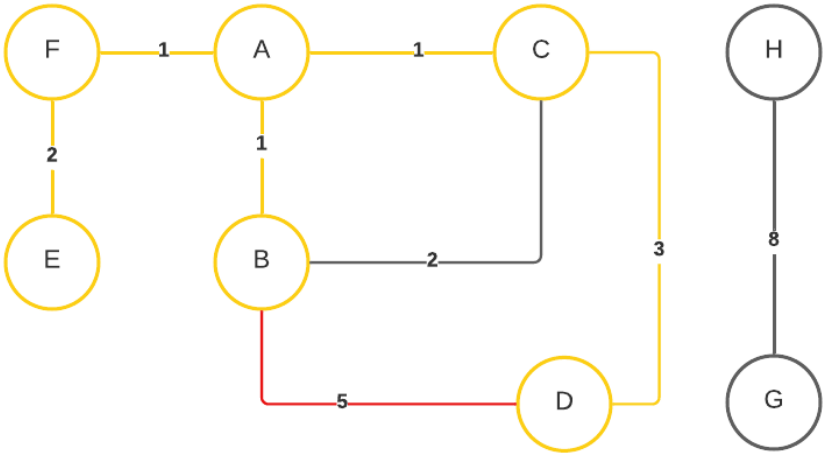


Figura 9: Grafo Kruskal's paso 3

En la Figura 9, conectamos la arista "$(C, D, \text{peso} = 3)$", e incluimos al nodo D en el mismo grupo que C, ya que no pertenece a ningún grupo. Luego, conectamos la arista "$(B, D, \text{peso} = 5)$", pero al pertenecer los nodos B y D al mismo grupo, descartamos la arista.

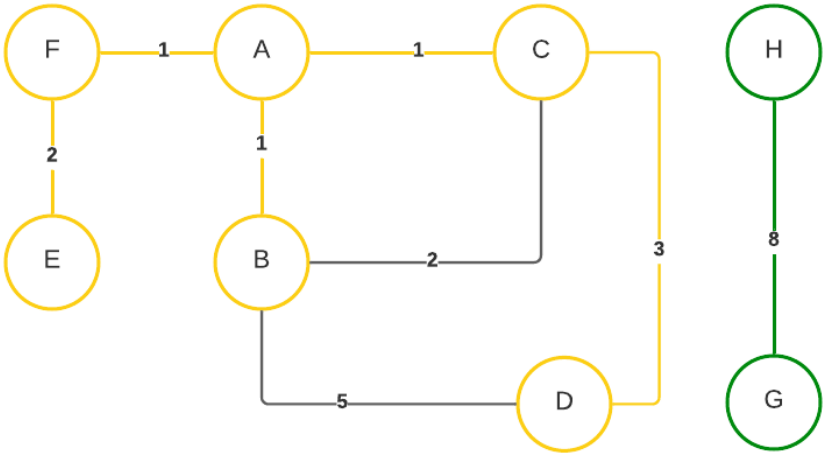


Figura 10: Grafo Kruskal's paso 4

Para la Figura 10, conectamos la arista "$(H, G, \text{peso} = 8)$". Como los nodos H y G no pertenecen a ningún grupo, los agregamos a un nuevo grupo que es diferente al de los demás nodos.

Después de esto, todos los nodos están conectados y hemos recorrido todas las aristas. El resultado es un bosque compuesto por dos árboles: $A'' = (V'', E'')$ y $A''' = (V''', E''')$.

$V'' = \{A, B, C, D, E, F\}$,

$E'' = \{(A, B, weight = 1), (A, C, weight = 1), (A, F, weight = 1), (F, E, weight = 2), (D, C, weight = 3)\}$

$V''' = \{H, G\}$,

$E''' = \{(H, G, weight = 8)\}$

3.2. Depth First Search (DFS)

El algoritmo de **Depth First Search** (Búsqueda en Profundidad) es un método utilizado para explorar grafos. Se inicia en un vértice de partida, también conocido como nodo raíz, y de manera recursiva se exploran los nodos vecinos (los nodos que están directamente conectados por una arista) en profundidad antes de retroceder al nodo anterior. Esto significa que se explora tanto como sea posible a lo largo de cada rama antes de retroceder [10].

Cuando se llega a un nodo que no tiene más vecinos sin visitar, se retrocede al nodo anterior y se continúa explorando el siguiente vecino disponible. Este proceso continúa hasta que se han explorado todos los nodos alcanzables desde el nodo fuente. En ese momento, el algoritmo de Búsqueda en Profundidad termina su ejecución [11].

Este algoritmo puede ser utilizado para resolver varios problemas en grafos, como la búsqueda de caminos, la identificación de componentes conectados, la detección de ciclos, entre otros. Además,

este algoritmo es fácil de implementar y podemos realizarlo de manera iterativa o recursiva.

También es importante destacar que este algoritmo puede generar diferentes árboles de expansión según el orden de exploración de nuestros nodos vecinos. Además, si nuestro grafo no es conexo, puede haber nodos que no sean alcanzables desde nuestra fuente.

3.2.1. Ejemplo Depth First Search

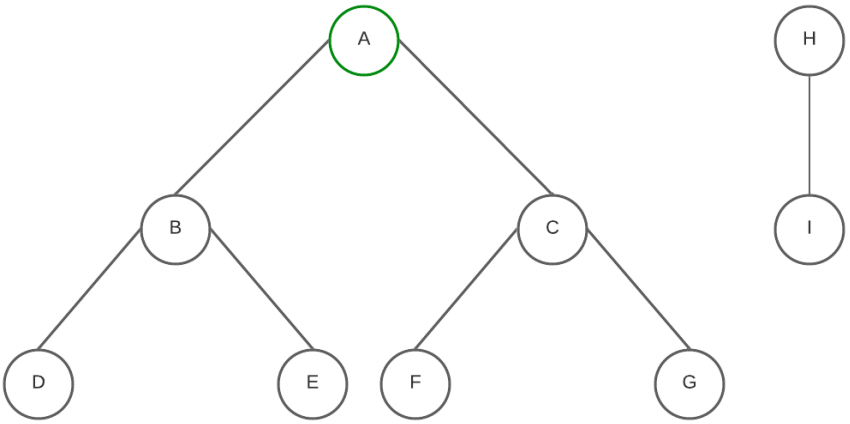


Figura 11: DFS paso 1

En la Figura **??**, elegimos un punto inicial, en este caso el nodo A, y comenzamos a recorrer los nodos en dirección antihoraria (la dirección antihoraria es una convención ampliamente utilizada en la visualización de grafos para establecer un sentido de recorrido en la representación de los nodos, que sigue el sentido contrario a las manecillas del reloj).

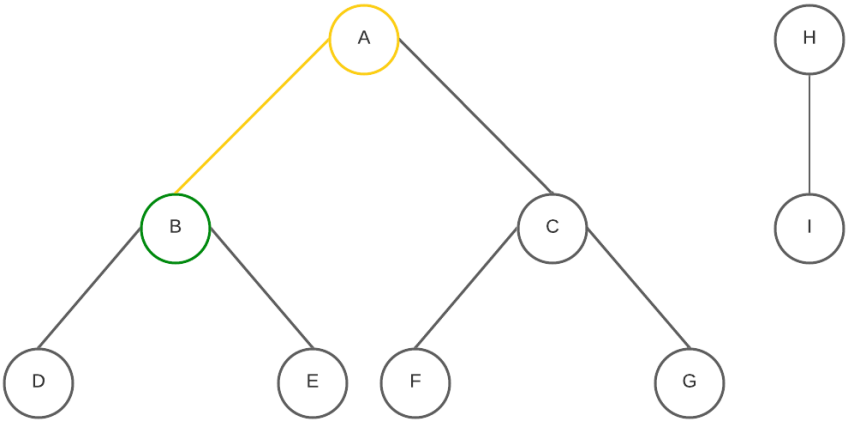


Figura 12: DFS paso 2

En la Figura 12, marcamos como visitado el nodo A y nos desplazamos al nodo B.

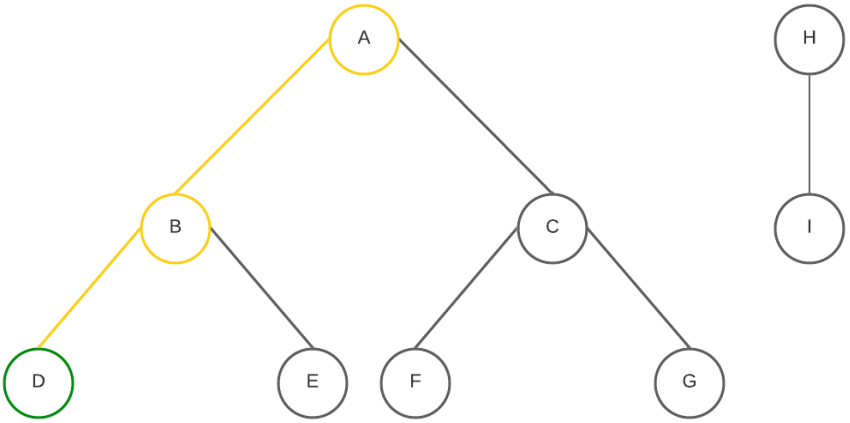


Figura 13: DFS paso 3

En la Figura 13, marcamos como visitado el nodo B y nos desplazamos al nodo D.

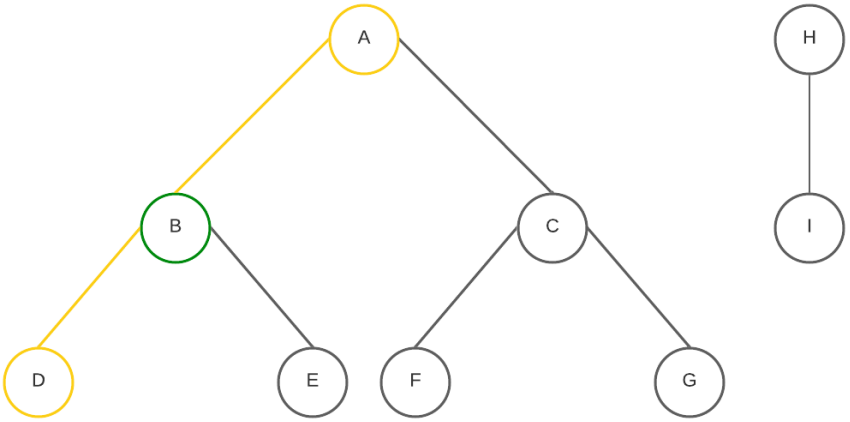


Figura 14: DFS paso 4

En la Figura 14, al no tener más nodos vecinos del nodo D que aún no hayan sido visitados, nos desplazamos hacia atrás al nodo B.

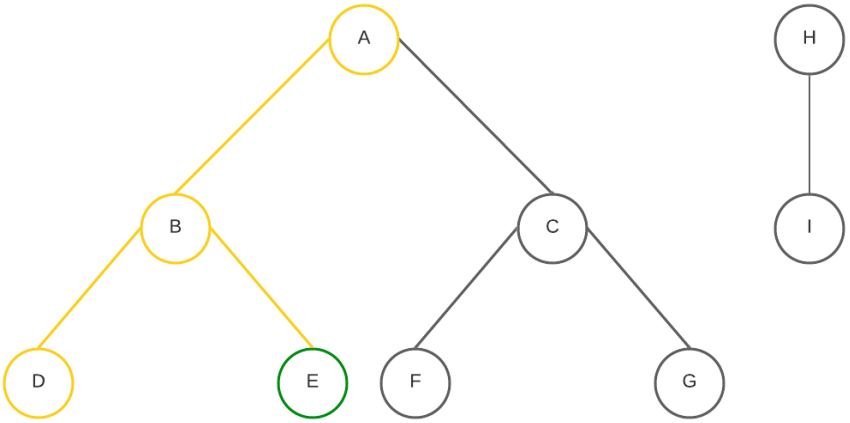


Figura 15: DFS paso 5

En la Figura 15, al ya haber visitado el nodo D, nos desplazamos al nodo E, que es el otro vecino del nodo B.

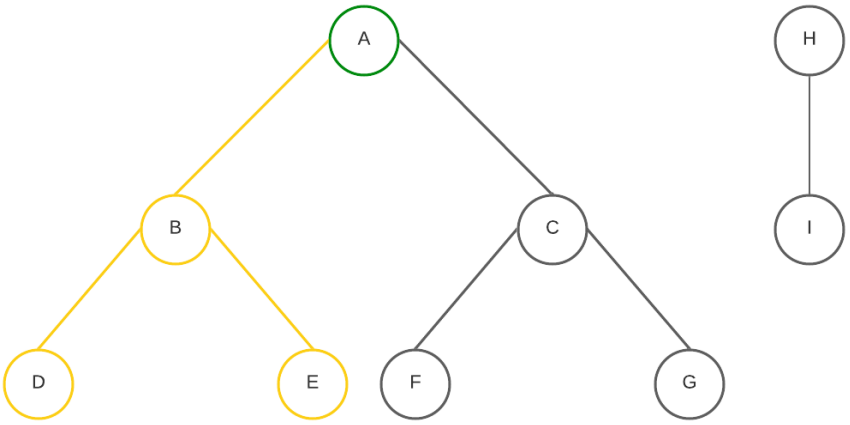


Figura 16: DFS paso 6

En la Figura 16, como el nodo E no tiene ningún vecino que no haya sido visitado, retrocedemos hacia el nodo B, pero como tampoco cuenta con nodos vecinos no visitados, volvemos a retroceder al nodo A.

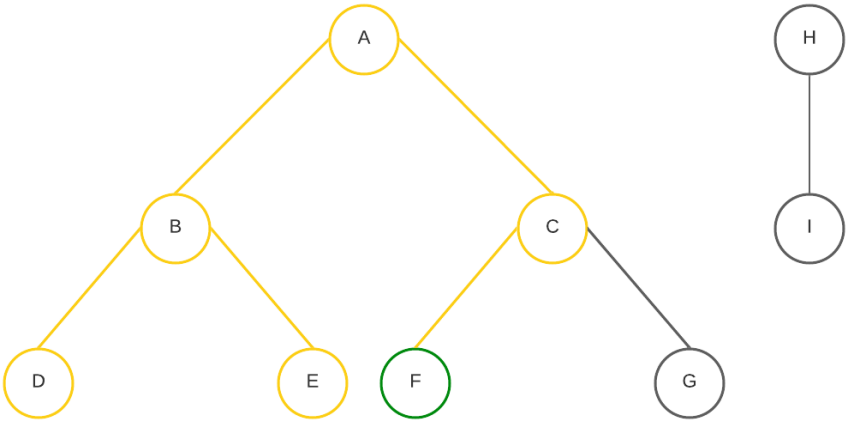


Figura 17: DFS paso 7

En la Figura 17, de manera similar al paso 5, nos desplazamos al otro vecino del nodo A que aún no ha sido visitado, el nodo C, y luego al nodo F, marcándolos como visitados.

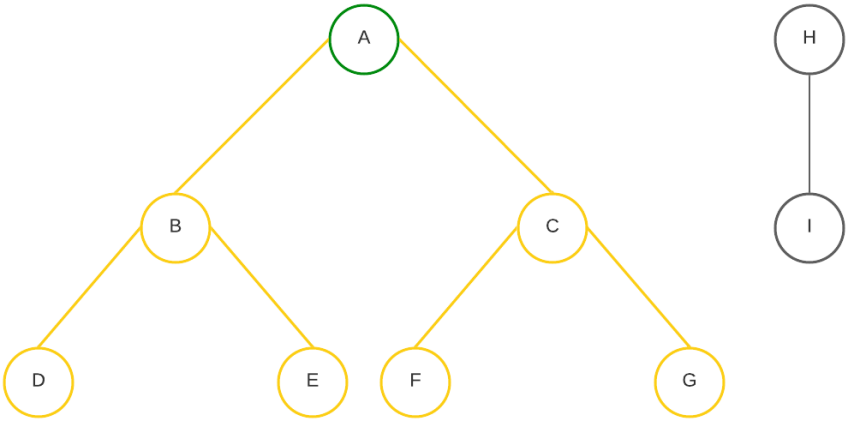


Figura 18: DFS paso 8

Y por último, en la Figura 18, retrocedemos del nodo F al C, luego avanzamos al nodo G, y al no tener más vecinos hacia donde avanzar, retrocedemos hasta volver al nodo inicial A, finalizando la ejecución del algoritmo a pesar de no haber recorrido los nodos H e I, ya que no era posible debido a que el grafo es no conexo.

3.3. Librerías utilizadas

En esta subsección se detallamos las librerías utilizadas para implementar el procedimiento en Python. Hemos elegido Python para facilitar el procesamiento de imágenes SAR, las cuales suelen tener un tamaño considerable (alrededor de $2GB$ cada una), y permitir que diferentes usuarios

interesados en un caso específico puedan realizar el procedimiento de manera sencilla. Además, buscamos hacer accesible el procesamiento a aquellas personas sin conocimientos en programación, evitando la necesidad de realizar instalaciones adicionales o enfrentarse a dificultades innecesarias [16].

- **NumPy**: Es una librería fundamental para la computación científica en Python. Ofrece herramientas para el manejo de matrices y vectores, así como para la realización de operaciones numéricas de alta velocidad. Puedes encontrar más información en `https://numpy.org/` [12].
- **Matplotlib**: Es una librería para la visualización de datos en 2D en Python. Permite crear gráficos, histogramas, gráficos de barras, etc. Puedes encontrar más información en `https://matplotlib.org/` [15].
- **NetworkX**: Es una librería para la creación, manipulación y estudio de estructuras de redes, como grafos y dígrafos. Puedes encontrar más información en `https://networkx.org/` [14].
- **SciPy**: Es una librería que contiene módulos para la optimización, álgebra lineal, integración, interpolación, procesamiento de señales, entre otros. En nuestro caso, utilizaremos el paquete "interpolate" para realizar las interpolaciones necesarias del spline. Puedes encontrar más información en `https://scipy.org/` [13].
- **os**: Es una librería directa de Python que nos permite interactuar con el sistema operativo en el que se está ejecutando Python. Permite realizar operaciones como leer archivos, crear directorios, eliminar archivos, y más.
- **enum**: Es una librería directa de Python para la creación de enumeraciones, que permiten definir un conjunto de valores nombrados.

3.4. Funciones utilizadas

En esta sub-sección se detallamos las principales funciones diseñadas y utilizadas en todo el procedimiento de esta tesis:

- **filter_block**: Realiza un filtro por bloques Rectangulares y Cuadrados.
- **distance_from_origin_within_radius**: Calcula la distancia entre un punto dado y otro punto de origen dado, solo si el punto dado está dentro de un radio dado.
- **img_to_graph**: Crea un grafo uniendo nodos cercanos en un rango de píxeles dado.
- **kruskal**: Implementación del algoritmo de **Kruskal** para encontrar el árbol de expansión mínima de un grafo no dirigido.
- **graph_to_tree**: Convierte un grafo en un árbol de expansión mínima utilizando el algoritmo de Kruskal.
- **complete_list**: Genera una lista completa de todas las aristas en un grafo, incluyendo sus reversos.

- **list_to_dict**: Convierte una lista de tuplas en un diccionario, agrupando los valores por clave.
- **dfs_with_end**: Realiza una búsqueda en profundidad (**DFS**) en un grafo representado como un diccionario de adyacencia, buscando un nodo de inicio hasta encontrar un nodo objetivo.
- **is_existing_road**: Verifica si existe un camino entre dos nodos en un grafo representado como un árbol de expansión mínima utilizando la estructura de conjuntos disjuntos obtenida del algoritmo de **Kruskal**.
- **found_road**: Busca un camino entre dos nodos en un grafo utilizando el algoritmo de búsqueda en profundidad (**DFS**).
- **check_all_roads**: Verifica todas las rutas en un grafo dado, utilizando la representación, nodos iniciales y finales dados.
- **collect_points**: Extrae las coordenadas de los puntos con valor 1 en un área especificada de una imagen binaria.
- **find_valid_extreme_points**: Encuentra los puntos extremos que tengan un camino que los conecte en una imagen binaria.
- **generate_line**: Genera el borde interpolado sobre la imagen original como fondo. También puede graficar solo los puntos sin interpolar o el borde sin interpolación.
- **unificate_imgs**: Unifica dos imágenes JSD_c y JSD_f en una nueva imagen JSD_{1d}.
- **run_jsd**: Ejecuta todo el procedimiento basado en los parámetros y la imagen JSD seleccionada.

4. Procedimiento

En esta sección, analizamos paso a paso el proceso que se debe aplicar a la imagen de entrada JSD (ya sea JSD_{1d} o JSD_{2d}) con el objetivo de obtener con precisión el borde de la superficie de interés en la imagen de entrada JSD. Para ello, aplicamos diferentes técnicas de procesamiento de imágenes y algoritmos de grafos, que nos permiten obtener un resultado óptimo.

En sí, el procedimiento consta de aplicar diferentes filtros para descartar los píxeles de ruido. Para ello, se realizará un filtrado por bloques rectangulares y conservamos únicamente el valor máximo de cada bloque. Luego, se creamos un grafo de la imagen, conectando cada píxel (nodo) con todos los píxeles que se encuentren dentro de su rango. Después, convertimos el grafo en un árbol utilizando el algoritmo **Union Find Kruskal's**. A continuación, se utilizará el algoritmo **DFS** para obtener el camino más corto entre el punto inicial y final ingresado por el usuario. Por último, creamos un spline del camino obtenido y lo representamos en una gráfica.

4.1. Estructura del procedimiento

El procesamiento de la imagen cuenta con 6 pasos principales:

- **Filtrado de la imagen y binarización**: Recibimos la imagen de entrada JSD, la cual puede ser JSD_{1d} o JSD_{2d}, y realizamos un filtrado inicial para descartar los píxeles de ruido más distantes del máximo. Luego, creamos otra imagen binarizada que contiene solo píxeles de valor 0 y 1.
- **Filtrado por bloques**: Realizamos un filtrado por bloques rectangulares [NxM], donde dividimos la imagen en K bloques y conservamos solo el máximo de cada bloque.
- **Creación del grafo**: Creamos un Grafo de la imagen conectando todos los píxeles (Nodos) con todos los píxeles que se encuentren en su rango.
- **Conversión de grafo a árbol**: Convertimos el Grafo en un Árbol mediante el algoritmo de **Union Find Kruskal's**.
- **Búsqueda del camino**: Obtenemos el camino más corto entre el punto "inicial" y "final", ingresados por el usuario, utilizando el algoritmo de **DFS**.
- **Spline y gráfica**: Realizamos un Spline del camino obtenido mediante el algoritmo de **DFS** y lo graficamos.

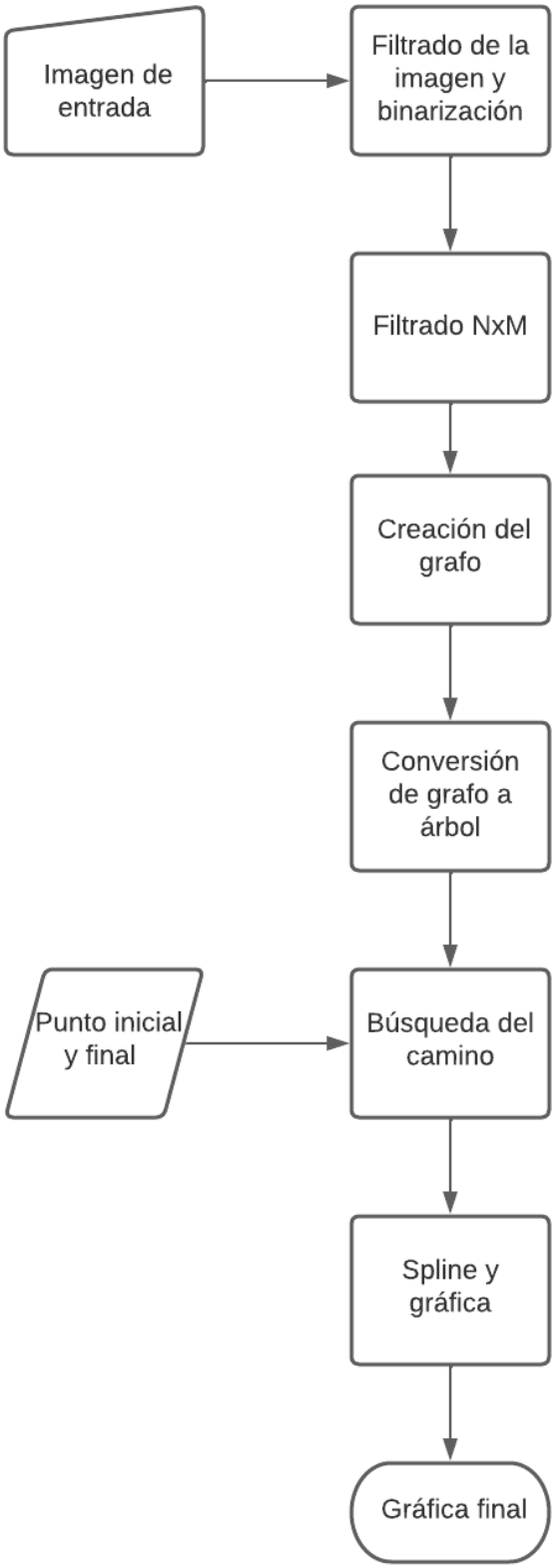


Figura 19: Diagrama de flujo del procesamiento de imagen

4.2. Filtrado de la imagen y Binarización

Las imágenes utilizadas en esta tesis se componen de píxeles que tienen un valor de intensidad. Estas imágenes pueden haber pasado por dos tipos diferentes de preprocesamiento:

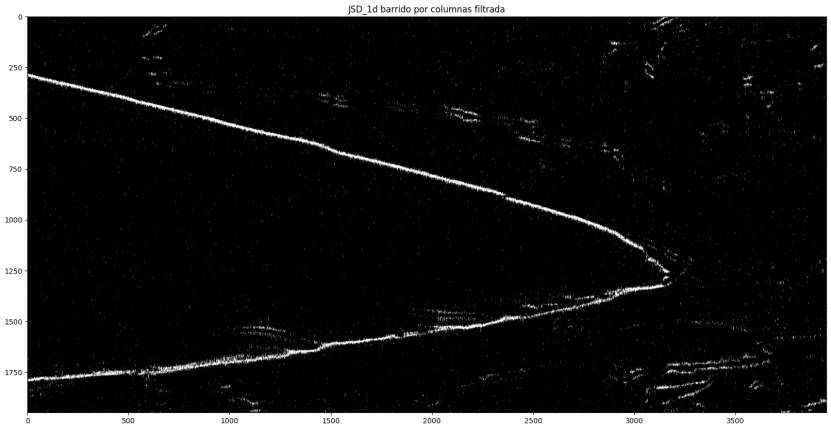
- **Preprocesado JSD_{1d} de Filas y Columnas**
- **Preprocesado JSD_{2d}**

Es importante destacar que, para obtener un resultado más eficiente en el preprocesamiento, utilizamos dos imágenes como entrada: JSD_{1d} de barrido por columnas y JSD_{1d} de barrido por filas. Estas imágenes se unificarán en una sola imagen JSD_{1d} después de aplicar los filtrados correspondientes, lo que nos permitirá continuar correctamente con el procesamiento. La unificación de las imágenes se explicará en la subsección 4.3.1.

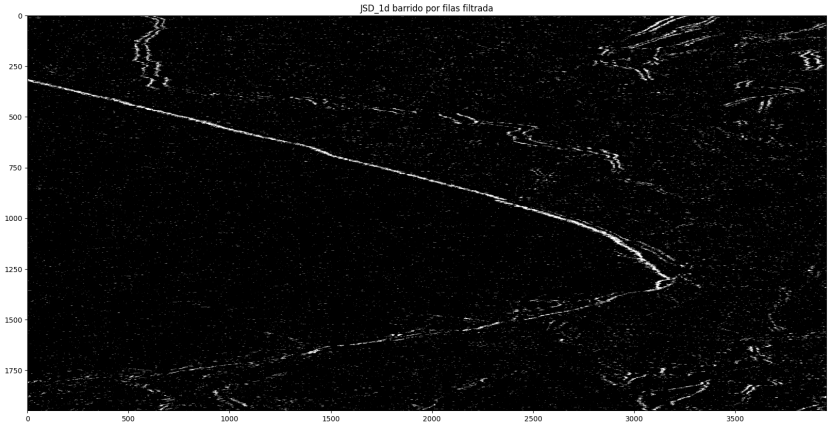
Independientemente del tipo de preprocesamiento elegido, el filtrado y binarización es el mismo. Para comenzar el procesamiento, debemos cargar la imagen en el formato correspondiente utilizando el método adecuado. En el caso del formato "npy", utilizamos la biblioteca "NumPy" con la función "load". Una vez cargada la imagen, se realiza el primer filtrado y binarización, donde solo se conservan los puntos con valores de intensidad mayores al cuádruple de la desviación estándar de la imagen. Esto se hace con el objetivo de obtener más ceros en los valores de intensidad para reducir la complejidad de los cálculos futuros.

Ejemplo de código:

```
JSD_thr = JSD.std() * 2
JSD = np.load('imagenes/JSD/s1a-iw2-slc-hh-20210609INTJSD.npy')
#Imagen Filtrada
imgFiltered = np.where((JSD > JSD_thr*2), JSD, 0)
#Imagen Filtrada y binarizada
imgFilteredBin = np.where((JSD > JSD_thr*2), 1, 0)
```



(a) JSD_{1d} barrido por columnas



(b) JSD_{1d} barrido por filas

Figura 20: Imágenes filtradas y binarizadas JSD_{1d}

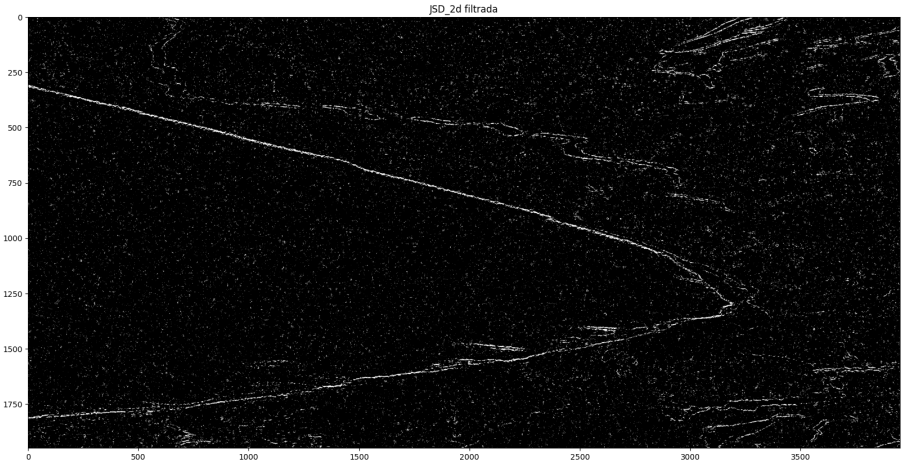


Figura 21: Imagen filtrada y binarizada JSD_{2d}

La imagen de entrada JSD_{1d} para el barrido por columnas contiene un total de 7.702.500 píxeles candidatos a bordes, Después de aplicar el proceso de filtrado y binarización (Figura 20a), la imagen

cuenta con solo 146.518 puntos candidatos a bordes, es decir, píxeles con un valor de intensidad de 1.

La imagen JSD_{1d} de barrido por filas, contiene un total de 7.702.500 píxeles candidatos a bordes, Después de aplicar el proceso de filtrado y binarización (Figura 20b), la imagen cuenta con solo 173.168 puntos candidatos a bordes, es decir, píxeles con un valor de intensidad de 1.

La imagen JSD_{2d} contiene un total de 7.702.500 píxeles candidatos a bordes, Después de aplicar el proceso de filtrado y binarización (Figura 21), la imagen cuenta con solo 232.246 puntos candidatos a bordes, es decir, píxeles con un valor de intensidad de 1.

4.3. Filtrado por bloques

Después de realizar el filtrado inicial, utilizamos la imagen filtrada sin binarización. Esta imagen aún contiene una cantidad significativa de candidatos a bordes, ya que estos píxeles o puntos conservan un valor de intensidad superior a 0. Con el fin de reducir el costo de los cálculos futuros, implementamos un filtro más personalizado que conserva únicamente el punto con el valor de intensidad máximo dentro de un bloque de tamaño específico y asigna un valor de 0 a los demás puntos. En este caso, el término

bloque se refiere a una subdivisión de la imagen original. Para realizar esta subdivisión, utilizamos un tipo de filtrado por bloques que genera subdivisiones rectangulares de tamaño $N \times M$ o cuadradas, donde $N = M$.

En el caso de JSD_{1d}, que consta de dos imágenes separadas (una barrida por columnas y otra por filas), se les aplica el filtrado por bloques de manera individual. Antes de crear el grafo, es necesario unificar las imágenes. Para ello, creamos una nueva imagen que contiene la información más relevante de ambas. En cada posición, tomamos el valor del píxel con mayor valor de las dos imágenes y utilizamos ese valor para la nueva imagen. De esta manera, la nueva imagen resultante contiene los píxeles más importantes de ambas imágenes.

Si hubiera tres imágenes separadas, se aplicaría la misma funcionalidad. Es decir, se filtrarían de manera individual y se unificarían después del filtrado.

4.3.1. Filtrado por bloques

Para aplicar el filtrado por bloques, realizamos una copia de la imagen original y la dividimos en bloques de tamaño $N \times M$. Si la imagen no es divisible por el tamaño de los bloques $N \times M$, la dividimos en la mayor cantidad posible de bloques de ese tamaño y agrupamos el resto en bloques más pequeños que $N \times M$. Por ejemplo, si nuestra imagen tiene un tamaño de 5×10 y buscamos bloques de 5×5, obtenemos como resultado 2 bloques de 5×5 porque la imagen es divisible. Pero si buscamos bloques de 3×2, dividimos la imagen en 5 bloques de 3×2 y 5 bloques de 2×2.

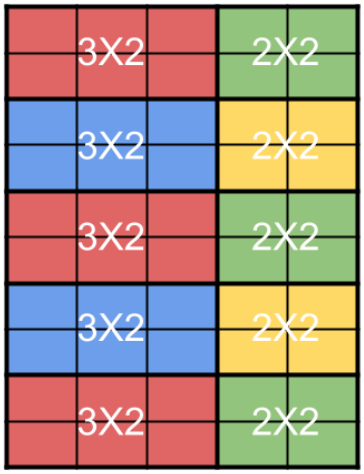


Figura 22: División de imagen 5×10 en bloques 3×2

Luego de la división de la imagen en bloques, realizamos un filtro bloque por bloque conservando el píxel con el máximo valor valor de intensidad y asignamos el valor 0 a los demás píxeles dentro del bloque, lo que resulta en que cada bloque tenga únicamente un píxel con un valor de intensidad mayor a 0. De esta manera, reducimos la cantidad de posibles candidatos a bordes a la cantidad de subdivisiones generadas.

Finalmente, reconstruimos la imagen a partir de los bloques, generando una nueva imagen filtrada por bloques rectangulares. La imagen resultante presenta una disminución en la cantidad de bordes y un aumento en la homogeneidad de las regiones.

Después de completar el proceso de filtrado por bloques, procedemos a aplicar la técnica de unificación de imágenes. Esta técnica implica la creación de una nueva imagen píxel por píxel, donde cada píxel de la nueva imagen se calcula a partir del valor máximo de intensidad entre los píxeles correspondientes de cada imagen original. En otras palabras, seleccionamos el píxel con el valor más alto de intensidad de cada conjunto de píxeles que ocupan la misma posición en las imágenes originales y utilizamos este valor para formar la nueva imagen resultante.

4.3.2. Código filtrado por bloques

```
def filter_block(img_filtered, n, m):
    """
    Realiza un filtro por bloques Rectangulares.
    Args:
      img_filtered (numpy.ndarray): Es la imagen de entrada que se desea filtrar.
↪   Debe ser un arreglo numpy.
      n (int): Es el tamaño del eje x para el filtro.
```

```
    m (int): Es el tamaño del eje y para el filtro.
  Returns:
    img_block_bin (numpy.ndarray): retorna la imagen filtrada.
  """
  # Verifica si n y m son nulos
  if (n is None or m is None):
      raise ValueError('Es necesario tener valores de n y m')
  # Se hace una copia de la imagen de entrada filtrada
  img_block_bin = np.copy(img_filtered)
  # Se itera a través de la imagen de entrada filtrada en bloques.
  for j in range(0, img_filtered.shape[1], n):
      for i in range(0, img_filtered.shape[0], m):
          # Se selecciona un bloque de la imagen de entrada filtrada del tamaño
          ↪  especificado por n y m
          block_array = img_filtered[i: i + m, j: j + n]
          # Se selecciona el elemento máximo del bloque
          max_elem = np.amax(block_array)
          # Si el elemento máximo del bloque es diferente de cero, se binariza
          ↪  el bloque de la imagen
          if max_elem != 0:
              block_bin = np.where(block_array == max_elem, 1, 0)
              img_block_bin[i: i + m, j: j + n] = block_bin
  # Se retorna la imagen filtrada como un arreglo numpy.
  return img_block_bin
```

Codigo 1: Filtrado por bloques

4.3.3. Código Unificación de imágenes

```
def unificate_imgs(JSD_c, JSD_f):
    """
    Unifica dos imágenes JSD_c y JSD_f en una nueva imagen JSD de 1 dimension.
    Args:
        JSD_c (numpy.ndarray): Imagen JSD_c.
        JSD_f (numpy.ndarray): Imagen JSD_f.
    Returns:
```

```
        array: Nueva imagen unificada.
    """
    # Se verifica que las dosm imagenes tengan las mismas dimenciones
    if(JSD_c.shape != JSD_f.shape):
        raise ValueError('Las imágenes deben ser del mismo tamaño.')
    # Corrige la imagen JSD_c desplazandola por 25 pixeles debido a un error en
    ↪  su creación.
    fixed_JSD_c = np.roll(JSD_c, 25, axis=0)
    # Crea la nueva imagen a partir del máximo de cada píxel en ambas imágenes
    ↪  originales.
    new_img = np.maximum(fixed_JSD_c, JSD_f)
    return new_img
```

Codigo 2: Unificación de imágenes

4.4. Creación del grafo

Luego de terminar de filtrar la imagen de entrada, conservamos los puntos candidatos a borde con mayor valor de intensidad y creamos con ellos un grafo único que abarca toda la imagen. Para crearlo, tomamos cada candidato a borde como un "nodo" del grafo y los conectamos entre sí mediante "aristas", asignando un peso determinado por la distancia en píxeles entre esos dos nodos. Por ejemplo, si un nodo se encuentra a 3 píxeles de distancia de otro, al conectarlos, el peso de su arista será de 3.

Continuamos seleccionando cada nodo individualmente y lo conectamos con los demás nodos que se encuentran en su rango, el cual se define mediante el parámetro max_range, que representa el radio de píxeles de búsqueda. Es decir, para un $max\_range = 5$, conectamos con una arista al nodo seleccionado con todos los nodos que se encuentran a una distancia máxima de 5 píxeles.

Para esto, formamos un círculo de 10 píxeles, el cual se define por el parámetro $\max_r ange \max_r ange \max_r ange \max_r ange$, con un diámetro centrado en el nodo seleccionado. Luego, conectamos dicho nodo con los nodos que se encuentran den

```
def distance_from_origin_within_radius(x_origin, y_origin, x, y, radius):
    """
    Calcula la distancia entre un punto dado y otro punto de origen dado, solo si
    ↪  el punto dado está dentro de un radio dado.
    Args:
        x_origin (float): Coordenada x del punto de origen.
```

```
        y_origin (float): Coordenada y del punto de origen.
        x (float): Coordenada x del punto a medir la distancia.
        y (float): Coordenada y del punto a medir la distancia.
        radius (float): Radio dentro del cual se considera que el punto está
↪   dentro del rango.
    Returns:
        float: Distancia entre los puntos si el punto dado está dentro del radio,
↪   -1 en caso contrario.
    """
    # Calcula la distancia entre el punto dado y el punto de origen
    distance_squared = (x - x_origin) ** 2 + (y - y_origin) ** 2
    # Verifica si los puntos está dentro del circulo y devuelve la distancia si
    ↪  es así
    if distance_squared <= radius ** 2:
        return np.sqrt(distance_squared)
```

Codigo 3: Función que determina la distancia entre 2 nodos

```
def img_to_graph(img_block_bin, max_range):
    """
    Crea un grafo uniendo nodos cercanos en un rango de píxeles dado.
    Args:
        img_block_bin (numpy.ndarray): Matriz binaria de la imagen.
        max_range (int): Rango máximo de conexión entre nodos.
    Returns:
        networkx.Graph: Grafo con nodos y aristas creados a partir de la matriz
↪   binaria.
    """
    # Se incia el grafo graph, node_id, i_size y j_size
    graph = nx.Graph()
    node_id = 0
    i_size, j_size = img_block_bin.shape
    # Se obtienen los índices de solo los puntos donde su valor sea 1
    indices = np.argwhere(img_block_bin == 1)
    # Se recorren los índices
    for i, j in indices:
```

```
            # Genero una key única para cada índice
            key = f"{i}-{j}"
            # Se agrega el puntos como nodo al grafo con su id
            graph.add_node(key, coordinates=(i, j), id=node_id)
            node_id += 1
            # Se genera un rango de búsqueda de nodos vecinos
            initial_i, initial_j = max(i - max_range, 0), max(j - max_range, 0)
            final_i, final_j = min(i_size, i + max_range + 1), min(j_size, j +
            ↪  max_range + 1)
            # Se crea una submatriz con el rango para los vecinos
            submatrix = img_block_bin[initial_i:final_i, initial_j:final_j]
            # Se filtran solo los puntos con valor 1 ya que son los únicos posibles
            ↪  candidatos a vecinos
            h_indices, k_indices = np.where(submatrix == 1)
            # Se recorren los candidatos a vecinos
            for h, k in zip(h_indices + initial_i, k_indices + initial_j):
                # Se comprueba que el nodo vecino no sea el mismo
                if i != h or j != k:
                    # Se calcula la distancia entre el nodo y su vecino
                    distance = distance_from_origin_within_radius(i, j, h, k,
                    ↪  max_range)
                    # Si se encuentra dentro del rango se agrega la arista
                    if distance > 0:
                        graph.add_edge(key, f"{h}-{k}", weight=distance)
    # Se retorna el grafo
    return graph
```

Codigo 4: Creación del grafo

4.5. Conversión del grafo a Árbol

En este punto, disponemos de un grafo que cubre toda la imagen, en el cual cada candidato a borde se representa como un nodo con sus respectivas conexiones. Sin embargo, el grafo tiene ciclos y redundancias que dificultan los algoritmos y pueden producir resultados no esperados. Por lo tanto, al convertirlo en un árbol, obtenemos los siguientes beneficios:

- **Simplificación del grafo**: Al convertir el grafo en un árbol, eliminamos todas las aristas que

forman ciclos, lo que simplifica el grafo y hace que sea más fácil de analizar.

- **Identificación de la estructura jerárquica**: Los árboles son estructuras jerárquicas que tienen un nodo raíz y nodos hijos, lo que facilita la identificación de la estructura jerárquica de un grafo.
- **Eficiencia en algoritmos**: Algoritmos específicos, como los algoritmos de búsqueda (como **DFS**) y de optimización, pueden ser más eficientes en árboles que en grafos generales, ya que los árboles tienen una estructura más simple.
- **Reducción de redundancia**: En algunos casos, un grafo puede tener varias aristas que conectan el mismo par de nodos. Al convertir el grafo en un árbol, podemos eliminar todas las aristas redundantes, lo que reduce el tamaño y la complejidad del grafo.

En resumen, un árbol es un tipo de grafo no dirigido y conexo, es decir, un grafo en el que hay una conexión entre cada par de nodos, pero sin ciclos. Para lograr esto, utilizamos el algoritmo de **Union Find Kruskal**, que se menciona previamente en la sección 3.1. Este algoritmo convierte el grafo en un árbol de expansión mínima, es decir, un árbol que mantiene únicamente las aristas de menor peso. Si el grafo no es conexo, lo que obtenemos como resultado es un bosque, que es un conjunto de árboles disjuntos, es decir, árboles que no comparten ningún nodo en común.

4.5.1. Código conversión del grafo a árbol

```
def uf_find(id, rep):
    # Si el representante de id es negativo, significa que id es su propio
    ↪ representante
    if rep[id] < 0:
        return id
    # Si no, se busca recursivamente el representante del padre de id
    rep[id] = uf_find(rep[id], rep)
    # Se actualiza el representante de id para apuntar directamente al
    ↪ representante de su conjunto
    return rep[id]
def uf_join(id1, id2, new_graph, rep, name_list):
    # Se encuentra el representante de cada nodo.
    x = uf_find(id1, rep)
    y = uf_find(id2, rep)
    # Si el representante de x es mayor que el de y, se intercambian.
    if rep[x] > rep[y]:
        x, y = y, x
```

```
    # Se actualiza la representación y el tamaño de la componente.
    rep[x] += rep[y]
    rep[y] = x
    # Se obtienen y agregan los nombres de los nodos al nuevo grafo.
    tuples = (name_list[id1], name_list[id2])
    new_graph.append(tuples)
def kruskal(grafo):
    """
    Implementación del algoritmo de Kruskal para encontrar el árbol de expansión
    ↪ mínima de un grafo no dirigido.
    Args:
        grafo (nx.Graph): Grafo no dirigido representado como un objeto de la
    ↪ clase nx.Graph de la biblioteca NetworkX.
    Returns:
        Tuple(list, np.array): Una tupla que contiene el árbol de expansión
    ↪ mínima representado como una lista de aristas y la estructura de datos de
    ↪ representación de conjuntos disjuntos utilizada por el algoritmo.
    """
    # Se obtienen y ordenar aristas por peso
    distance = sorted(grafo.edges(data=True), key=lambda e: e[2].get('weight', 1)
    ↪ )
    # Se obtienen y ordenar nodos por id
    nodes = sorted(grafo.nodes(data=True), key=lambda n: n[1]['id'])
    # Se obtiene un diccionario del atributo id para cada nodo del grafo
    id_list = nx.get_node_attributes(grafo, 'id')
    # Se inicializa una lista vacía para almacenar el nombre de cada nodo
    name_list = [None] * len(nodes)
    # Se asignan los nombres de cada nodo en una lista de nombres
    for node in nodes:
        name_list[node[1]['id']] = node[0]
    # Se inicializa la estructura de datos de conjuntos disjuntos / representantes
    rep = np.full(len(nodes), -1, dtype=int)
    # Inicializar el árbol de expansión mínima
    new_graph = []
    for elem in distance:
        # Obtener el id del primer nodo en la arista
```

```
            id1 = id_list[elem[0]]
            # Obtener el id del segundo nodo en la arista
            id2 = id_list[elem[1]]
            # Verificar si los nodos ya están conectados en el árbol
            if uf_find(id1, rep) != uf_find(id2, rep):
                # Si no están conectados, se une los nodos en el árbol
                uf_join(id1, id2, new_graph, rep, name_list)
    # Se retorna el árbol de expansión mínima y la estructura de datos de
    ↪  conjuntos disjuntos
    return new_graph, rep
```

Codigo 5: Union Find Kruskal's

En el Código 5, definimos el algoritmo *Union Find Kruskal's*, necesario para encontrar el árbol de expansión mínima del grafo. Sin embargo, este algoritmo solo devuelve una tupla que contiene las aristas que conformarán el árbol y una lista de los representantes de cada nodo. Con esta información, podemos crear un nuevo grafo que será nuestro árbol resultante del algoritmo **Kruskal**.

```
def generate_tree(kruskal_list, G):
    """
    Genera un árbol a partir de una lista de aristas obtenidas mediante el
↪   algoritmo de Kruskal.
    Args:
        kruskal_list (list): Lista de aristas obtenidas mediante el algoritmo de
↪   Kruskal.
        G (nx.Graph): Grafo original representado como un objeto de la clase
↪   nx.Graph de la biblioteca NetworkX.
    Returns:
        nx.Graph: Árbol generado a partir de la lista de aristas, representado
↪   como un objeto de la clase nx.Graph de NetworkX.
    """
    # Se obtiene un diccionario con el atributo 'id' de cada nodo del grafo
    id_dict = nx.get_node_attributes(G, 'id')
    # Se obtiene un diccionario con el atributo 'coordinates' de cada nodo del
    ↪  grafo
    coordinates_dict = nx.get_node_attributes(G, 'coordinates')
    # Se obtiene un diccionario con el 'weight' de cada arista del grafo
```

```
    weight_dict = nx.get_edge_attributes(G, "weight")
    # Se crea un diccionario con los nodos del grafo original que contiene el
    ↪  atributo 'id' y el atributo 'coordinates'
    nodes_dict = {k: {'coordinates': coordinates_dict[k], 'id': id_dict[k]} for k
    ↪  in id_dict}
    # Se crea un grafo vacío para almacenar el árbol
    tree = nx.Graph()
    # Se agregan los nodos del grafo original al grafo del árbol
    tree.add_nodes_from(nodes_dict.items())
    # Se itera sobre las aristas obtenidas por el algoritmo de Kruskal
    for elem in kruskal_list:
        # Se obtienen los nodos que forman la arista
        key1, key2 = elem[:2]
        # Se agrega la arista al árbol con el atributo 'weight' correspondiente
        tree.add_edge(key1, key2, weight=weight_dict[elem])
    # Se devuelve el grafo del árbol generado
    return tree
```

Codigo 6: Creación del Árbol

En el código 6, definimos la función que convierte la lista resultante de **Kruskal** en un árbol.

```
def graph_to_tree(graph):
  """
  Convierte un grafo en un árbol de expansión mínima utilizando el algoritmo de
↪  Kruskal.
  Args:
      graph (nx.Graph): Grafo original representado como un objeto de la clase
↪  nx.Graph de la biblioteca NetworkX.
  Returns:
      Tuple(nx.Graph, np.array): Una tupla que contiene el árbol de expansión
↪  mínima representado como un objeto de la clase nx.Graph de NetworkX, y la
↪  estructura de datos de representación de conjuntos disjuntos utilizada por el
↪  algoritmo de Kruskal.
  """
```

```
    # Se obtiene la lista de aristas del árbol de expansión mínima y la estructura
    ↪  de datos utilizada por Kruskal
    kruskal_list, rep = kruskal(graph)
    # Se genera un árbol a partir de la lista de aristas obtenida
    tree = generate_tree(kruskal_list, graph)
    # Retorna el árbol y la estructura de datos como una tupla
    return tree, rep
```

Codigo 7: Conversión del grafo a Árbol

Una vez hemos definido las funciones necesarias, el código 7 ejecuta las funciones anteriores (5 y 6) para convertir el grafo en un árbol. Como resultado, obtenemos el árbol resultante del algoritmo de *Kruskal* y la lista de representantes de cada nodo. Estos resultados los utilizaremos en algoritmos posteriores.

4.6. Búsqueda del Camino

La búsqueda de caminos se refiere a encontrar la ruta o conexión entre dos nodos específicos en un grafo. En general, existen varios algoritmos que se pueden utilizar para realizar la búsqueda de caminos en grafos, como el algoritmo de búsqueda en profundidad (**DFS**) mencionado anteriormente en la sección 3.2. Para la búsqueda del camino, dividimos el procedimiento en 2 subpartes:

- **Verificación de existencia del camino**.
- **Obtención del camino**.

4.6.1. Verificación de existencia del camino

Antes de obtener un camino, es necesario que definamos el punto de inicio y el punto final, y los verifiquemos, ya que algunos puntos no se conectan entre sí, especialmente en grafos no conexos. Esta verificación la realizamos a través de una lista de representantes obtenida del algoritmo de **Kruskal**. La forma de verificar si dos puntos están conectados es mediante el uso de la lista de representantes, ya que si ambos tienen el mismo representante, entonces existe un camino que los conecta.

4.6.2. Código Verificación de existencia del camino

```
def is_existing_road(graph, rep, start, end):
    """
```

```
    Verifica si existe un camino entre dos nodos en un grafo representado como un
↪   árbol de expansión mínima
    utilizando la estructura de conjuntos disjuntos obtenida del algoritmo de
↪   Kruskal.
    Args:
        graph (nx.Graph): Grafo representado como un objeto de la clase nx.Graph
↪   de la biblioteca NetworkX.
        rep (np.array): Estructura de conjuntos disjuntos obtenida del algoritmo
↪   de Kruskal.
        start (str: int-int): Nodo de inicio del camino.
        end (str: int-int): Nodo de fin del camino.
    Returns:
        bool: True si existe un camino entre los dos nodos, False en caso
↪   contrario.
    """
    # Se obtiene un diccionario de nodos y sus id
    id_dict_tree = nx.get_node_attributes(graph, 'id')
    # Busca los id de los nodos de inicio y fin
    elem1 = id_dict_tree[start]
    elem2 = id_dict_tree[end]
    # Se usa la estructura de conjuntos disjuntos para verificar si los nodos
    ↪  están conectados
    return uf_find(elem1, rep) == uf_find(elem2, rep):
```

Codigo 8: Verificación de existencia del camino

4.6.3. Obtención del camino

Una vez verificada la existencia del camino, podemos utilizar el algoritmo de búsqueda **Depth First Search (DFS)**, mencionado anteriormente en la sección 3.2, con el cual obtendremos una lista de coordenadas de los puntos seleccionados como borde. Estos puntos se conectan entre sí y forman el camino buscado. El algoritmo **DFS** tradicional explora un grafo hasta que ya no puede continuar, sin embargo, nosotros utilizaremos este algoritmo con una pequeña modificación. Detendremos el código en el momento en que encontremos el camino que conecte el punto de inicio con un punto final dado, es decir, la búsqueda del camino se limitará a estos dos puntos.

4.6.4. Código Obtención del camino

```
def dfs_with_end(graph_dic, start, end):
    """
    Realiza una búsqueda en profundidad (DFS) en un grafo representado como un
↪   diccionario de adyacencia, buscando un nodo de inicio hasta encontrar un nodo
↪   objetivo.

    Args:
        graph_dic (dict): Diccionario de adyacencia que representa el grafo.
        start (str: int-int): Nodo de inicio de la búsqueda.
        end (str: int-int): Nodo objetivo a buscar.

    Returns:
        np.array: Un array con la ruta encontrada desde el nodo de inicio hasta
↪   el nodo objetivo, representada como una secuencia de nodos.
    """
    global visited, road, stack
    # Conjunto de nodos visitados
    visited = set()
    # Lista de nodos que forman la ruta encontrada
    road = []
    # Pila que contiene los nodos que se están visitando
    stack = []
    def dfs(x):
        # Si el nodo ya fue visitado o la ruta ya fue encontrada, no se hace nada
        if x in visited or len(road) != 0:
            return
        # Se agrega el nodo actual al conjunto de nodos visitados
        visited.add(x)
        # Se agrega el nodo actual a la pila de nodos visitados
        stack.append(x)
        # Si el nodo actual es el nodo objetivo, se agrega la ruta a la lista de
        ↪  rutas encontradas
        if x == end:
            road.extend(np.copy(stack))
```

```
        # Se itera sobre los nodos adyacentes al nodo actual
        for elem in graph_dic[x]:
            dfs(elem)
        # Se elimina el nodo actual de la pila de nodos visitados
        stack.pop()
    # Llamada inicial a la función dfs
    dfs(start)
    # Se copia la lista de nodos que forman la ruta encontrada
    result = np.copy(road)
    # Se eliminan las variables globales utilizadas
    del visited, stack, road
    # Retorna la lista de nodos que forman la ruta encontrada
    return result
```

Codigo 9: Algoritmo DFS

4.6.5. Código búsqueda del camino

```
def complete_list(G):
    """
    Genera una lista completa de todas las aristas en un grafo G, incluyendo sus
↪   reversos.
    Args:
        G (nx.Graph): Grafo original representado como un objeto de la clase
↪   nx.Graph de la biblioteca NetworkX.
    Returns:
        list: Lista completa de todas las aristas del grafo, incluyendo sus
↪   reversos, ordenadas alfabéticamente.
    """
    # Se crea una lista con todas las aristas del grafo
    edges = list(G.edges)
    # Se crea una lista de las aristas invertidas del grafo
    reversed_edges = [e[::-1] for e in edges]
    # Se crea una lista que contenta las aristas del grafo y su reverso
    all_edges = np.concatenate((edges, reversed_edges), axis=0)
```

```
    # Aplicar np.argsort para obtener los índices que ordenan el array
    ↪ alfabéticamente
    sorted_indexes = np.argsort(all_edges[:, 0])
    # Ordenar la lista utilizando los índices obtenidos
    ordered_list = all_edges[sorted_indexes]
    # Se retorna una lista completa de aristas del grafo ordenadas alfabéticamente
    return ordered_list

def list_to_dict(datalist):
    """
    Convierte una lista de tuplas en un diccionario, agrupando los valores por
↪   clave.
    Args:
        datalist (list): Lista de tuplas con claves y valores.
    Returns:
        dict: Diccionario con las claves como claves y los valores agrupados en
↪   listas.
    """
    # Creamos un diccionario con valores por defecto como listas vacías
    result = defaultdict(list)
    # Iteramos sobre cada tupla en la lista de datos
    for key, value in datalist:
        # Agregamos el valor a la lista correspondiente a la clave en el
        ↪ diccionario
        result[key].append(value)
    # Retornamos el diccionario resultante
    return result
```

Codigo 10: Funciones necesarias para correr correctamente DFS

Las funciones del código 10 son necesarias para el correcto funcionamiento de nuestro algoritmo **DFS**. Necesitamos un diccionario que incluya todas las aristas del árbol, pero para lograrlo debemos completar primero la lista. El algoritmo de **Kruskal** nos proporciona la lista con la arista A conectada a B, pero no incluye la arista B conectada a A, ya que es la misma. Sin embargo, al transformarlo en un diccionario, debemos indicar ambas conexiones.

```
def find_road(graph, rep, start_node, end_node):
```

```
    """
    Busca un camino entre dos nodos en un grafo representado como un objeto de la
↪   clase nx.Graph de NetworkX, utilizando el algoritmo de búsqueda en
↪   profundidad (DFS).

    Args:
        graph (nx.Graph): Grafo original representado como un objeto de la clase
↪   nx.Graph de la biblioteca NetworkX.
        rep (np.array): Estructura de datos de representación de conjuntos
↪   disjuntos utilizada por el algoritmo de Kruskal.
        start_node (str: int-int): Nodo de inicio del camino.
        end_node (str: int-int): Nodo de fin del camino.

    Returns:
        List or None: Una lista que contiene el camino encontrado entre los dos
↪   nodos, si existe. None si no se encuentra un camino entre los nodos o si los
↪   nodos no son válidos.
    """
    # Se obtienen los nodos del grafo
    nodes = graph.nodes()
    # Se verifica que el nodo inicial sea válido
    if not (start_node in nodes):
        print(f'Start node: ({start_node}) no in nodes')
        return
    # Se verifica que el nodo final sea válido
    if not (end_node in nodes):
        print(f'End node: ({end_node}) no in nodes')
        return
    # Si existe una ruta entre el nodos inicial y final
    if is_existing_road(graph, rep, start_node, end_node):
        # Se crea una lista con todos los nodos del grafo
        complete = complete_list(graph)
        # Se convierte la lista de nodos en un diccionario
        graphDict = list_to_dict(complete)
         # Se realiza la búsqueda en profundidad del camino entre el nodos
          ↪  inicial y final
```

```
        return dfs_with_end(graphDict, start_node, end_node)
    print('There is no road between the nodes')
    return
```

Codigo 11: Búsqueda de camino

El Código 11 simplemente ejecuta las funciones 10, 9 y 8, las cuales son necesarias para buscar un camino en un grafo utilizando el algoritmo **DFS**. En resumen, el código 11 se encarga de convertir una lista de aristas en un diccionario, verificar la existencia de un camino entre dos puntos dados y ejecutar el algoritmo **DFS** para obtener la lista de coordenadas que conforman el camino buscado.

4.7. Spline y Gráfica

Una vez obtenida la lista de coordenadas de los puntos que formamos el camino deseado y si nuestra elección de los parámetros fue correcta, ya tenemos la información necesaria para generar la poligonal del borde de la superficie. Para una mejor representación visual, graficaremos el camino obtenido sobre la imagen original. Sin embargo, es importante asegurarnos de que los parámetros seleccionados sean precisos para lograr una representación más precisa del camino.

Sin embargo, la unión de los puntos produce un borde brusco y desordenado debido a los diferentes filtros aplicados. Para obtener un resultado más limpio, aplicamos un spline al camino obtenido. Este spline genera una interpolación y aproximación de los datos, logrando así una imagen más fluida sin cambios bruscos de dirección.

Con esto, podemos graficar el borde de la forma más detallada posible. Para llevar a cabo el spline, empleamos el paquete “interpolate” de la librería “scipy”.

4.7.1. Código spline y gráfica

```
def generate_line(background_img, dotted_road, title='',
↪  plot_interpolate_line=False, plot_line=False, plot_dots=False, ax=None,
↪  imshow_kws=None, dots_kws=None, line_kws=None, interpolate_kws=None,  k=5,
↪  s=10000):
    """
   Genera el borde interpolado sobre la imagen original como fondo. También puede
↪  graficar solo los puntos sin interpolar o el borde sin interpolación.
    Args:
        background_img (numpy.ndarray): Imagen de fondo.
        dotted_road (list): Lista de coordenadas de puntos.
```

```
        title (str, optional): Título de la figura.
        plot_interpolate_line (bool, optional): Indica si se debe trazar la línea
↪   interpolada.
        plot_line (bool, optional): Indica si se debe trazar la línea original de
↪   puntos.
        plot_dots (bool, optional): Indica si se deben graficar los puntos.
        ax (object, optional): Indica el ax que se va a usar, si es None
↪   utilizamos el por defecto.
        imshow_kws (object, optional): Indica las opciones de imshow que vamos a
↪   utilizar, si es None utilizamos el por defecto.
        dots_kws (object, optional): Indica las opciones para la grafica de dots,
↪   si es None utilizamos el por defecto.
        line_kws (object, optional): Indica las opciones para la grafica de line,
↪   si es None utilizamos el por defecto.
        interpolate_kws (object, optional): Indica las opciones para la grafica
↪   interpolate, si es None utilizamos el por defecto.
        k (int, optional): Orden del spline para la interpolación.
        s (int, optional): Valor de suavidad para la interpolación.

    Returns:
        None
    """
    # Obtiene las coordenadas x e y de los puntos
    x_elems = [int(coord.split('-')[1]) for coord in dotted_road]
    y_elems = [int(coord.split('-')[0]) for coord in dotted_road]

    # si no hay un eje usamos el que se usa por defecto
    # gca == "get current axis"
    plt.figure(figsize=(20, 20))
    ax = plt.gca() if ax is None else ax
    # La variable imshow_kws es el diccionario con las configuraciones para
    ↪  imshow, mas algunas cosas por defecto
    imshow_kws = {} if imshow_kws is None else imshow_kws
    imshow_kws.setdefault("cmap", "gray")
    ax.imshow(background_img, **imshow_kws)
    # Grafica los puntos, línea original y línea interpolada si se indica
```

```
if plot_dots:
    dots_kws = {} if dots_kws is None else dots_kws
    dots_kws.setdefault("markersize", 5)
    dots_kws.setdefault("markeredgecolor", "red")
    ax.plot(x_elems, y_elems, marker=".", **dots_kws)
if plot_line:
    line_kws = {} if line_kws is None else line_kws
    line_kws.setdefault("color", "green")
    line_kws.setdefault("linewidth", "3")
    ax.plot(x_elems, y_elems, **line_kws)
if plot_interpolate_line:
    # Interpola la curva con splprep
    tck, u = interpolate.splprep([x_elems,y_elems],k=k,s=s, task=0)
    out = interpolate.splev(u,tck)
    interpolate_kws = {} if interpolate_kws is None else interpolate_kws
    interpolate_kws.setdefault("color", "blue")
    interpolate_kws.setdefault("linewidth", "3")
    plt.plot(out[0], out[1], "b", **interpolate_kws)
# titulo
ax.set_title(title)
# Rotorna la figura
return ax
```

Codigo 12: Spline y gráfica

5. Ejecución y Resultados

Después de haber definido todo el proceso al cual sometemos la imagen de entrada JSD, procederemos a ejecutar las funciones en el orden correspondiente. Para ello, debemos seleccionar cuidadosamente los parámetros de entrada adecuados para obtener los resultados deseados.
Es importante tener en cuenta el tipo de imagen de entrada al seleccionar los parámetros de entrada, ya que elegir valores incorrectos para el filtro por bloques, el rango en la creación del grafo o los puntos iniciales y finales puede resultar en resultados incorrectos o poco precisos.

5.1. Codigos de Ejecución

Para la ejecución, contamos con dos códigos específicos, uno para la imagen JSD_{1d} y otro para la imagen JSD_{2d}. Además, para ambos códigos, definimos funciones que nos ayudan con la elección de los puntos iniciales y finales.

```
def check_all_roads(graph, rep, initials, endings):
    """
    Función que verifica todas las rutas en un grafo dado, utilizando una
↪   representación, nodos iniciales y finales dados.
    Args:
        graph (nx.Graph): Grafo de entrada.
        rep (dict): Representación de nodos en el grafo.
        initials (array): Lista de nodos iniciales.
        endings (array): Lista de nodos finales.
    Returns:
        list (array): Lista de opciones de ruta válidas, representadas como pares
↪   de claves de nodos.
    """
    # Creamos una lista vacía que almacenará todas las opciones de rutas válidas.
    options = []
    # Obtenemos el atributo "id" de cada nodo del grafo y lo almacenamos en un
    ↪   diccionario.
    id_dict_tree = nx.get_node_attributes(graph, 'id')
    # Recorremos cada elemento inicial y final de la lista de nodos iniciales y
    ↪   finales.
    for elem_inicio in initials:
        for elem_final in endings:
```

```
            # Creamos una clave para cada elemento inicial y final a partir de su
            ↪ índice 0 y 1.
            key_1 = f"{elem_inicio[0]}-{elem_inicio[1]}"
            key_2 = f"{elem_final[0]}-{elem_final[1]}"
            try:
                # accedecemos a los elementos 1 y 2 del diccionario a través de
                ↪ sus claves.
                elem1 = id_dict_tree[key_1]
                elem2 = id_dict_tree[key_2]
                # Utilizamos la función "uf_find" para encontrar el conjunto al
                ↪ que pertenecen los elementos.
                a = uf_find(elem1, rep)
                b = uf_find(elem2, rep)
                # Si los elementos pertenecen al mismo conjunto, entonces
                ↪ añadimos las claves a la lista de opciones válidas.
                if a == b:
                    options.append((key_1, key_2))
            # Si los elementos no pertenecen al mismo conjunto, se omite el error
            ↪ y se continúa con la siguiente iteración.
            except:
                warnings.warn(f'The dots {elem1} and {elem2} do not belong to the
                ↪ same group', category=Warning)
                pass
    # Retornamos la lista de opciones de rutas válidas.
    return options

def collect_points(img, x_start, x_end, y_start, y_end):
    """
    Extrae las coordenadas de los puntos con valor 1 en un área especificada de
↪   una imagen binaria.
    Argumentos:
    - img (numpy.ndarray) representa la imagen binaria.
    - x_start (int): Coordenada de inicio en el eje x.
    - x_end (int): Coordenada de fin en el eje x.
    - y_start (int): Coordenada de inicio en el eje y.
    - y_end (int): Coordenada de fin en el eje y.
```

```
    Retorna:
    -numpy.ndarray: contiene las coordenadas de los puntos con valor 1 en el área
↪   especificada,
                ajustadas a las coordenadas reales en la imagen original.
    """
    # Se hace una copia de la sección de la imagen correspondiente al área
    ↪  especificada
    area = np.copy(img[x_start:x_end, y_start:y_end])
    # Se obtienen los índices de los puntos con valor 1
    indices  = np.argwhere(area == 1)
    # Ajusta los índices a las coordenadas reales en la imagen original
    indices [:, 0] += x_start
    indices [:, 1] += y_start
    # Retorna los índices ajustados como un numpy array
    return indices
    def find_valid_extreme_points(imgBlockBin, tree, rep, start_area, end_area):
    """
    Encuentra los puntos extremos que tengan un camino que los conecte en una
↪   imagen binaria.
    Args:
        imgBlockBin (numpy.ndarray): Imagen binaria.
        tree (nx.Graph): Objeto de arbol.
        rep (array): lista de representación.
        start_area (dict): Área de inicio con coordenadas de inicio y fin.
        end_area (dict): Área de fin con coordenadas de inicio y fin.
    Returns:
        list: Lista de opciones de puntos extremos con camino.
    """
    # Lista de posibles inicios
    initials = collect_points(imgBlockBin, start_area['y_start'],
    ↪  start_area['y_end'], start_area['x_start'], start_area['x_end'])
    # Lista de posibles finales
    endings = collect_points(imgBlockBin, end_area['y_start'], end_area['y_end'],
    ↪  end_area['x_start'], end_area['x_end'])
    # Prueba todos los posibles inicios con todos los posibles finales para ver
    ↪  cuales de ellos estan conectados
```

```
    options = check_all_roads(tree, rep, initials, endings)
    return options
```

Codigo 13: Encontrar y validar puntos iniciales y finales

Las funciones definidas en el código 13 se utilizan para encontrar todos los puntos candidatos dadas dos áreas específicas iniciales y finales, y verificamos todos los puntos iniciales con los finales, formando así pares de puntos que cuentan con un camino entre ellos. Con esto, contamos con una forma más automatizada de seleccionar los puntos iniciales y finales para la creación del borde

```
def run_jsd(jsd_type, n, m, range, start_area, end_area, k, s, background_img,
↪  title='', plot_interpolate_line=None, plot_line=None, plot_dots=None,
↪  img_filtered_c=None, img_filtered_f=None, img_filtered_2d=None,
↪  interpolate_kws=None, line_kws=None, dots_kws=None, ax=None, imshow_kws=None):
    """
    Args:
        img_filtered_c (numpy.ndarray): Imagen filtrada de la imagen JSD_c.
        img_filtered_f (numpy.ndarray): Imagen filtrada del la imagen JSD_f.
        img_filtered_2d (numpy.ndarray): Imagen filtrada de la imagen JSD_2d.
        jsd_type (Enum): Es el tipo de JSD a aplicar, representado como un objeto
↪  de tipo JSD. Se espera que sea de tipo enumerado JSD y puede tener los
↪  valores '1d' o '2d'.
        n (int): Es el tamaño del eje x para el filtro de tipo 'box'. No es
↪  requerido si se está utilizando un filtro de tipo 'circle'.
        m (int): Es el tamaño del eje y para el filtro de tipo 'box'. No es
↪  requerido si se está utilizando un filtro de tipo 'circle'.
        range (int): Rango máximo de conexión entre nodos.
        start_area (object): Área en la que se buscarán los puntos de inicio del
↪  camino en la imagen.
        end_area (object): Área en la que se buscarán los puntos finales del
↪  camino en la imagen.
        k (int, optional): Orden del spline para la interpolación.
        s (int, optional): Valor de suavidad para la interpolación.
        plot_interpolate_line (bool, optional): Indica si se debe trazar la línea
↪  interpolada.
        plot_line (bool, optional): Indica si se debe trazar la línea original de
↪  puntos.
```

```
        plot_dots (bool, optional): Indica si se deben graficar los puntos.
        background_img (numpy.ndarray): Imagen de fondo.
        title (string): Título para la figura generada.
        ax (object, optional): Indica el ax que se va a usar, si es None
↪  utilizamos el por defecto.
        imshow_kws (object, optional): Indica las opciones de imshow que vamos a
↪  utilizar, si es None utilizamos el por defecto.
        dots_kws (object, optional): Indica las opciones para la grafica de dots,
↪  si es None utilizamos el por defecto.
        line_kws (object, optional): Indica las opciones para la grafica de line,
↪  si es None utilizamos el por defecto.
        interpolate_kws (object, optional): Indica las opciones para la grafica
↪  interpolate, si es None utilizamos el por defecto.
    Returns:
        None
    """
    # Es necesario para que find_road funcione, ya que excede el límite máximo de
    ↪  recursión permitida en Python
    sys.setrecursionlimit(10000)
    # Verifica si jsd_type es un tipo Filter
    if not isinstance(jsd_type, JSD):
        raise NotJSDEnum("El tipo de filtro no es de tipo JSD")
    # Verifica si jsd_type es de tipo Filter.D1 y si img_filtered_c,
    ↪  img_filtered_f son nulos
    if jsd_type == JSD.D1 and (img_filtered_c is None or img_filtered_f is None):
        raise NotImgsD1("Es necesario tener imagen img_filtered_c y
        ↪  img_filtered_f")
    # Verifica si jsd_type es de tipo Filter.D2 y si img_filtered_2d es nula
    if jsd_type == JSD.D2 and img_filtered_2d is None:
        raise NotImgsD2("Es necesario tener imagen img_filtered_2d")
    if jsd_type == JSD.D1 :
      # Filtrado por bloques de las imágenes de entrada
      img_block_bin_c = filter_block(img_filtered_c, n, m)
      img_block_bin_f = filter_block(img_filtered_f, m, n)
      # Unificación de las imágenes filtradas por bloques
      img_block_bin = unificate_imgs(img_block_bin_c, img_block_bin_f)
```

```
else:
    # Filtrado por bloques de las imágenes de entrada
    img_block_bin = filter_block(img_filtered_2d, n, m)
# Creación del grafo a partir de la imagen filtrada por bloques
G = img_to_graph(img_block_bin, range)
# Creación del árbol generador a partir del grafo
tree, rep = graph_to_tree(G)
# Selección de los pares de puntos iniciales y finales
extreme_points = find_valid_extreme_points(img_block_bin, tree, rep,
↪  start_area, end_area)
# Si hay caminos posibles entre los puntos extremos, elegimos uno y generamos
↪  la línea que los conecta
if extreme_points:
    # Elegimos el punto inicial y final
    start_node = extreme_points[0][0]
    end_node = extreme_points[0][1]
    print('Start:', start_node, 'End:', end_node)
    # Corremos DFS en el árbol para encontrar el camino que conecta los
    ↪  puntos extremos
    dotted_road = find_road(tree, rep, start_node, end_node)
    print("Number of nodes on the road:", len(dotted_road))
    # Generamos la línea que conecta los puntos extremos en la imagen de fondo
    if plot_interpolate_line or plot_line or plot_dots :
        generate_line(background_img, dotted_road, title,
        ↪  plot_interpolate_line=plot_interpolate_line, plot_line=plot_line,
        ↪  plot_dots=plot_dots, k=k, s=s,
                      interpolate_kws=interpolate_kws, line_kws=line_kws,
                      ↪  dots_kws=dots_kws, ax=ax, imshow_kws=imshow_kws)
        return
    else:
        return dotted_road
raise NotValidExtremePointError('There is no possible road between the start
↪  and end points')
```

Codigo 14: Ejecución JSD_{1d} y JSD_{2d}

El Código 14 se utiliza para procesar imágenes JSD_{1d} y JSD_{2d}. En este código, contamos con una variable llamada jsd_type, que nos permite seleccionar qué tipo de JSD procesar. Si la imagen de entrada es JSD_1d, tenemos dos imágenes para filtrar la JSD por barrido por columnas y filas. Después de esto, unificamos las imágenes. En cambio, si es JSD_{2d}, solo filtramos una imagen y no necesitamos unificar. Luego, creamos el grafo, lo convertimos en árbol y buscamos posibles puntos iniciales y finales. Si obtenemos al menos un par de puntos iniciales y finales válidos, obtenemos el camino y lo graficamos. En caso contrario, retornamos un mensaje de error.

5.2. Resultados

En esta subsección se presentaremos los resultados de diferentes pruebas cambiando los parámetros y el tipo de filtrado por bloques, ya sea rectangular o cuadrado. Los resultados dependen considerablemente de la elección de los valores de los parámetros y del tipo de filtrado. En algunos casos, al elegir ciertos valores de los parámetros provoca que el programa no logre encontrar un camino entre el punto inicial y final seleccionados. En otros conjuntos de valores de los parámetros, el programa encuentra un camino, pero éste no representa un borde adecuado. Por ejemplo, en la Figura 23 se muestra el caso en donde aplicamos el filtrado por bloques cuadrados en una imagen JSD_{2d} con los parámetros $\mathbf{N} = \mathbf{M} = 20$ y **range** $= 45$, dando como resultado un camino que no corresponde al borde de la figura, pues siguió una trayectoria entre los puntos de ruido.

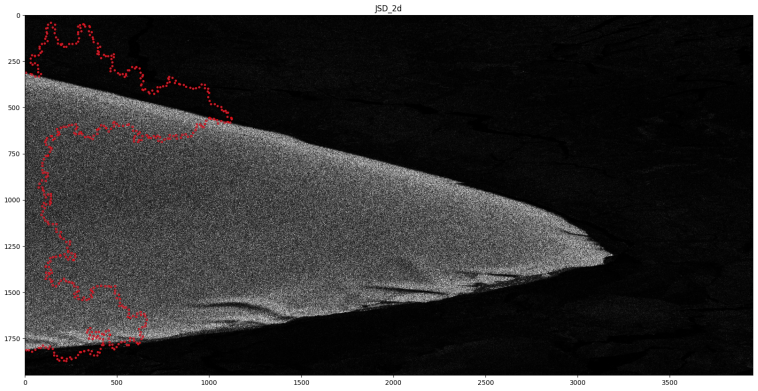


Figura 23: Puntos bordes inadecuados JSD_{2d} filtrado cuadrado con $N = M = 20$ y $range = 45$

En las siguientes subsecciones se detallamos los resultados de tres casos analizados: JSD_{1d} filtrado rectangular, JSD_{2d} filtrado rectangular y JSD_{2d} filtrado cuadrado. Durante estas ejecuciones, debemos elegir varios parámetros, como los valores de **N**, **M**, **range**, **k**, **s**, **background_img**, **initial_area** y **end_area**. Sin embargo, los valores de **k** y **background_img** se mantienen iguales en todas las ejecuciones que ddetallamos.

5.2.1. Resultados JSD_{1d} filtrado Rectangular

A continuación se mostraremos los resultados correspondientes al algoritmo de búsqueda de bordes en la imagen JSD_{1d} con filtrado rectangular. En el Código 15 se presenta el algoritmo con los parámetros seleccionados y los valores asignados a cada uno.

```
### JSD_1d ###
####### area inicial ######
start_area = {
'x_start': 0,
'x_end': 10,
'y_start': 310,
'y_end': 325
}
############################
###### area final #########
end_area = {
'x_start': 0,
'x_end': 10,
'y_start': 1810,
'y_end': 1820
}
############################
run_jsd(img_filtered_c=img_filtered_c, img_filtered_f=img_filtered_f,
↪  img_filtered_2d=None, jsd_type=JSD.D1,n=1, m=30, title='JSD_1d', range=30,
↪  start_area=start_area, end_area=end_area, background_img=JSD_original, k=5,
↪  s=20_000, plot_interpolate_line=False, plot_line=False, plot_dots=True)
```

Codigo 15: Ejecución JSD_{1d} filtrado rectangular

En la figura 24 se mostramos la imagen original superpuesta con la línea de puntos de color azul que representa el borde obtenido después de aplicar el spline.

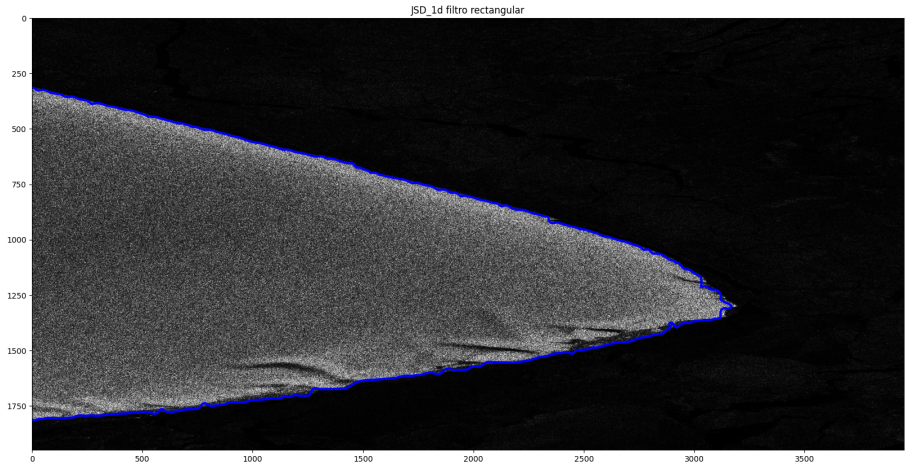


Figura 24: spline JSD_{1d} filtrado rectangular

Para obtener esta Figura, se utilizó una técnica que involucra dos filtros rectangulares unidimensionales: uno de tamaño 1×30 y otro de tamaño 30×1, correspondientes a los cálculos de la JSD por columnas y filas, respectivamente. Posteriormente, se unificaron las imágenes obtenidas para crear una nueva. A partir de ella, procedemos a obtener el grafo.

Al crear el grafo, la variable **range** adquiere una gran importancia pues para bajos valores (en comparación con el tamaño del bloque de filtro, formado por N y M) reducen la cantidad de caminos posibles entre el punto de inicial y final, llegando incluso a no haber ninguna conexión. Por otro lado, una elección de un valor muy alto, aumenta exponencialmente la complejidad del grafo debido al crecimiento del número de conexiones, lo que incrementa el tiempo de cálculo requerido.

Para caso de la Figura 24, hemos elegido un valor de **range** igual a 30 porque el tamaño del bloque de filtro seleccionado busca distanciar los puntos separados por 30 píxeles. Sin embargo, esta distancia de separación entre los puntos podría disminuir debido a la unificación. Al elegir **range** $= 30$, obtenemos un número adecuado de puntos para construir el grafo y aseguramos que el proceso sea computacionalmente eficiente.

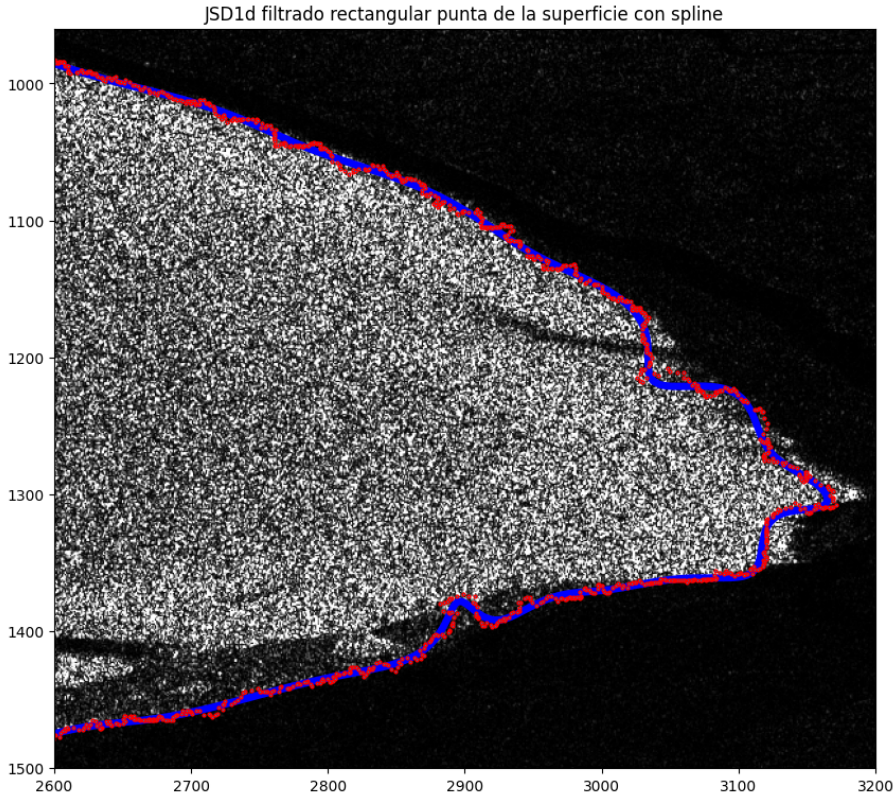


Figura 25: spline y puntos bordes JSD_{1d} filtrado rectangular

En la Figura 25 se puede apreciar un recorte de la Figura 24, en donde comparamos la línea de puntos del borde con el spline, representados por líneas roja y azul, respectivamente. Se ve que hay regiones en donde el borde no encierra la superficie de tonos grises y blancos, pues quedan zonas sin cubrir, especialmente en la punta extrema derecha (entre las coordenadas x=[3000, 3200] e y=[1100, 1400]). A pesar de ello, se puede notar que se conserva la forma general de la superficie, lo que permite obtener información relevante de la misma.
El tiempo de ejecución para este caso es de aproximadamente 40 segundos utilizando Google Colab con el plan gratuito.

5.2.2. Resultados JSD_{2d} filtrado Rectangular

Para el caso de JSD_{2d} con filtrado rectangular, utilizamos el Código 16.

```
### JSD_2d ###
###### area inicial ######
start_area = {
'x_start': 0,
'x_end': 20,
'y_start': 310,
'y_end': 320
}
############################
### JSD de 2 dimensiónes ###
###### area final #########
end_area = {
'x_start': 0,
'x_end': 20,
'y_start': 1810,
'y_end': 1820
}
############################
run_jsd(img_filtered_c=None, img_filtered_f=None,
↪  img_filtered_2d=img_filtered_2d, jsd_type=JSD.D2, n=1, m=30, range=30,
↪  start_area=start_area, end_area=end_area, background_img=JSD_original, k=5,
↪  s=25_000, plot_interpolate_line=False, plot_line=False, plot_dots=True,
↪  title='JSD_2d')
```

Codigo 16: Ejecución JSD_{2d} filtrado rectangular

La Figura 26 muestra la imagen original con el borde obtenido después de aplicar el spline, para este caso.

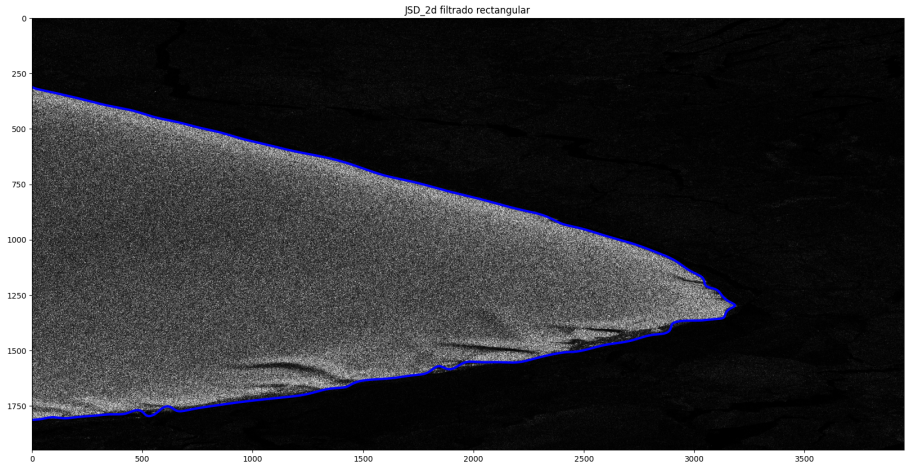


Figura 26: Puntos de borde JSD_{2d} filtrado rectangular

Al igual que antes, llevamos a cabo un proceso de selección de parámetros. En este caso, utilizamos un filtro rectangular unidimensional de tamaño 1×30 y elegimos un **rango** de 30. El tamaño del bloque de filtro seleccionado busca distanciar verticalmente los puntos en 30 píxeles, de esta manera, podemos tener en cuenta todos los puntos de cada bloque.

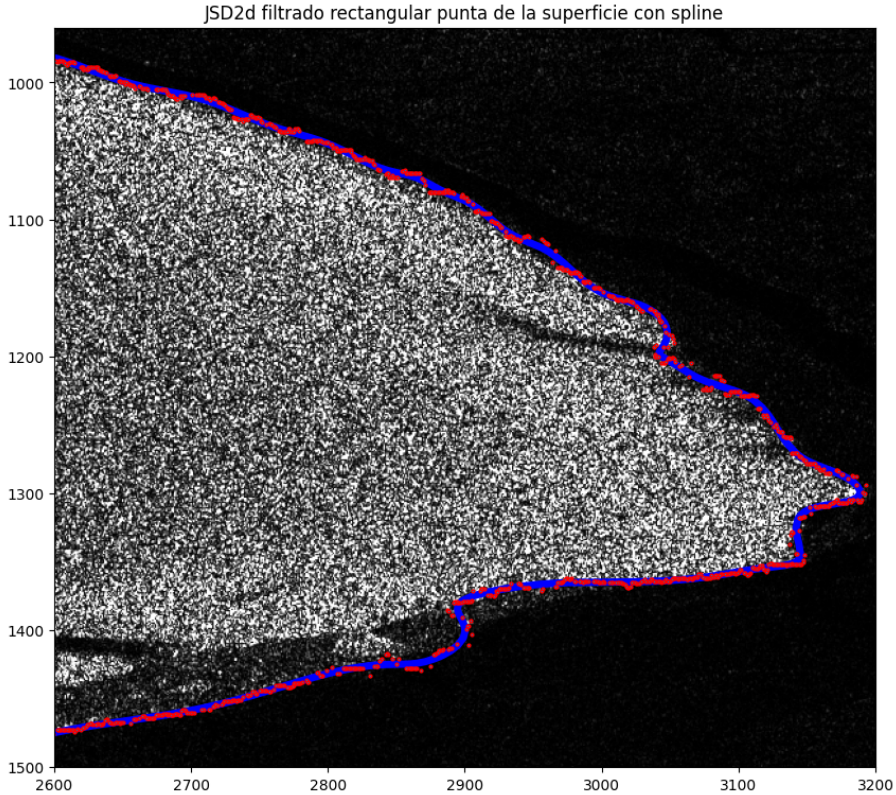


Figura 27: spline y puntos bordes JSD_{2d} filtrado rectangular

La figura 27 muestra un recorte de la figura 26, de la misma zona mostrada en la figura 25. La línea

de borde presenta un alto nivel de detalle, logrando cubrir de manera precisa la forma completa de la superficie de los tonos de grises, lo que permite obtener información mas precisa de la misma. El tiempo de ejecución para este caso es de aproximadamente 50 segudos utilizando Google Colab con el plan gratuito.

5.2.3. Resultados JSD_{2d} filtrado Cuadrado

En esta subsección, se mostrarán los resultados obtenidos al aplicar el algoritmo de búsqueda de bordes en la imagen JSD_{2d} con filtrado cuadrado. El código 16 muestra el algoritmo correspondiente para este nuevo caso.

```
### JSD_2d ###
###### area inicial ######
start_area = {
'x_start': 0,
'x_end': 20,
'y_start': 310,
'y_end': 320
}
############################
### JSD de 2 dimensiónes ###
###### area final #########
end_area = {
'x_start': 0,
'x_end': 20,
'y_start': 1810,
'y_end': 1820
}
############################
run_jsd(img_filtered_c=None, img_filtered_f=None,
↪  img_filtered_2d=img_filtered_2d, jsd_type=JSD.D2, n=8, m=8, range=15,
↪  start_area=start_area, end_area=end_area, background_img=JSD_original, k=5,
↪  s=25_000, plot_interpolate_line=False, plot_line=False, plot_dots=True,
↪  title='JSD_2d')
```

Codigo 17: Ejecución JSD_{2d} filtrado cuadrado

La superposición del borde después de aplicar el spline con la imagen original se muestra en la Figura 28.

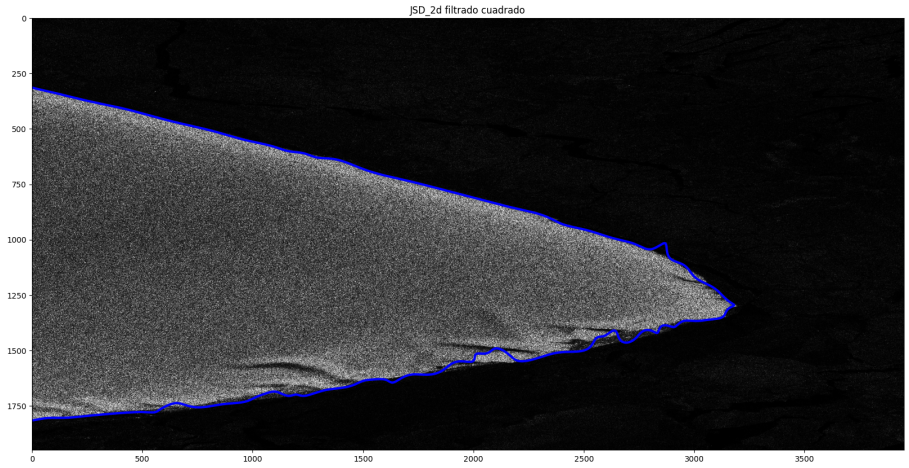


Figura 28: Puntos bordes JSD_{2d} filtrado cuadrado

En este caso, aplicamos un filtro cuadrado de lado 8, es decir $\mathbf{N} = \mathbf{M} = 8$, con un **range** $= 15$. Seleccionamos un **range** de 15, ya que el tamaño del bloque de filtro seleccionado busca distanciar los puntos en aproximadamente entre 8 a 11 píxeles (8 diámetro del cuadrado, 11 diagonal del cuadrado), de manera que contando con un rango de 15 píxeles, se pueden tener en cuenta todos los puntos de cada bloque.

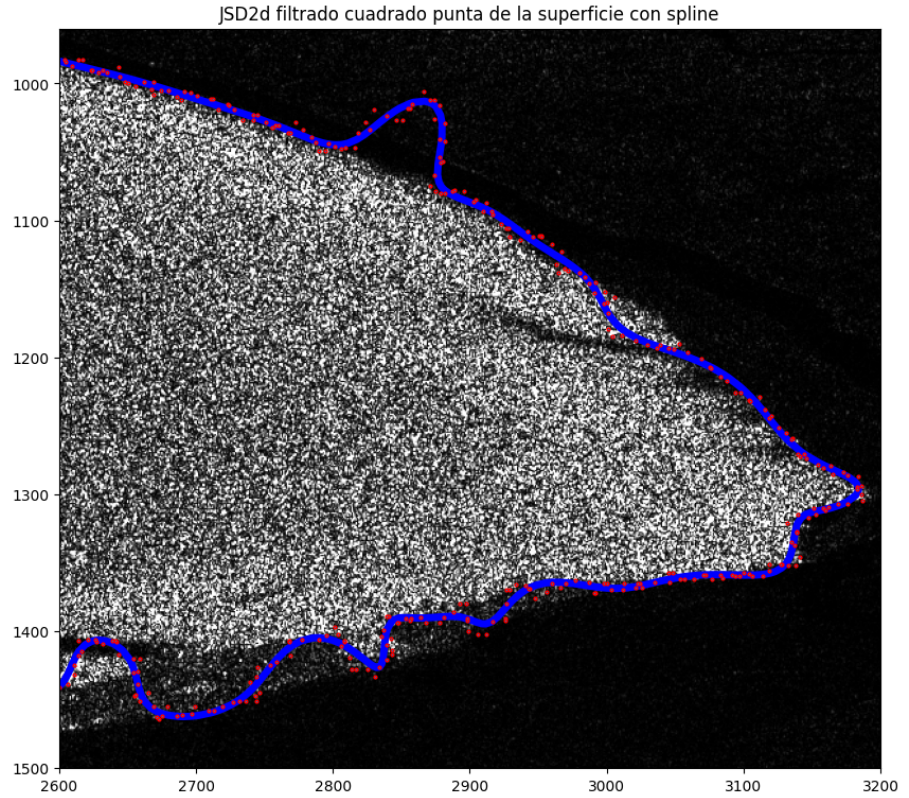


Figura 29: spline y puntos bordes JSD_{2d} filtrado cuadrado

Al igual que en los dos casos anteriores, en la Figura 29, presentamos un recorte de la Figura 28, donde comparamos la línea de puntos del borde con el spline, representados por líneas roja y azul, respectivamente. Podemos observar que en ciertas zonas, la línea del borde muestra errores visibles, especialmente en la sección superior de la imagen (entre las coordenadas x=[2600, 2900] e y=[900, 1100]), donde la línea del borde se aleja de la superficie de los tonos de grises. Esto se debe a la gran cantidad de ruido presente en la imagen, a pesar de los filtros aplicados. Para obtener un mejor resultado, debemos aplicar más filtros, ya sea antes o después de la creación del JSD.
El tiempo de ejecución para este caso es de aproximadamente 15 segundos utilizando Google Colab con el plan gratuito.

5.3. Comparación de resultados

En esta subsección se comparamos los mejores resultados obtenidos en la ejecución de JSD_{1d} y JSD_{2d}. Es evidente que el resultado del filtrado rectangular es superior al cuadrado, pues si comparamos las Figuras 28 e 26 se puede ver que la Figura del filtrado rectangular describe la linea de borde con mayor detalle, por lo que se compararán las imágenes JSD_{1d} y JSD_{2d} obtenidas mediante el filtrado rectangular. Dichas imágenes se muestran en las Figuras 24 y 26, respectivamente. En consecuencia no se tendrá en cuenta la Figura 28
En la Figura 30, podemos observar y comparar las imágenes JSD_{1d} y JSD_{2d} representadas en azul y verde, respectivamente. Como se analizamos en la subsección 5.2.1, la línea de borde generada por JSD_{1d} no cubre completamente la superficie a contornear. En cambio, su contraparte JSD_{2d} sí cubre completamente la superficie, lo que permite obtener una mayor cantidad de información y detalles.
El tiempo de ejecución para JSD_{1d} es de aproximadamente **40 segundos** utilizando Google Colab con el plan gratuito, mientras que para JSD_{2d} es de aproximadamente **50 segundos**. Por lo tanto, no existe una diferencia significativa en el tiempo de procesamiento entre ambas opciones.
En resumen, para este estudio, recomendamos utilizar la técnica JSD_{2d} con filtrado rectangular, ya que observamos una significativa mejora en la calidad de la línea de borde en comparación con la técnica JSD_{1d} con filtrado rectangular. Además, el tiempo de ejecución de JSD_{2d} no es significativamente mayor que el de JSD_{1d}.

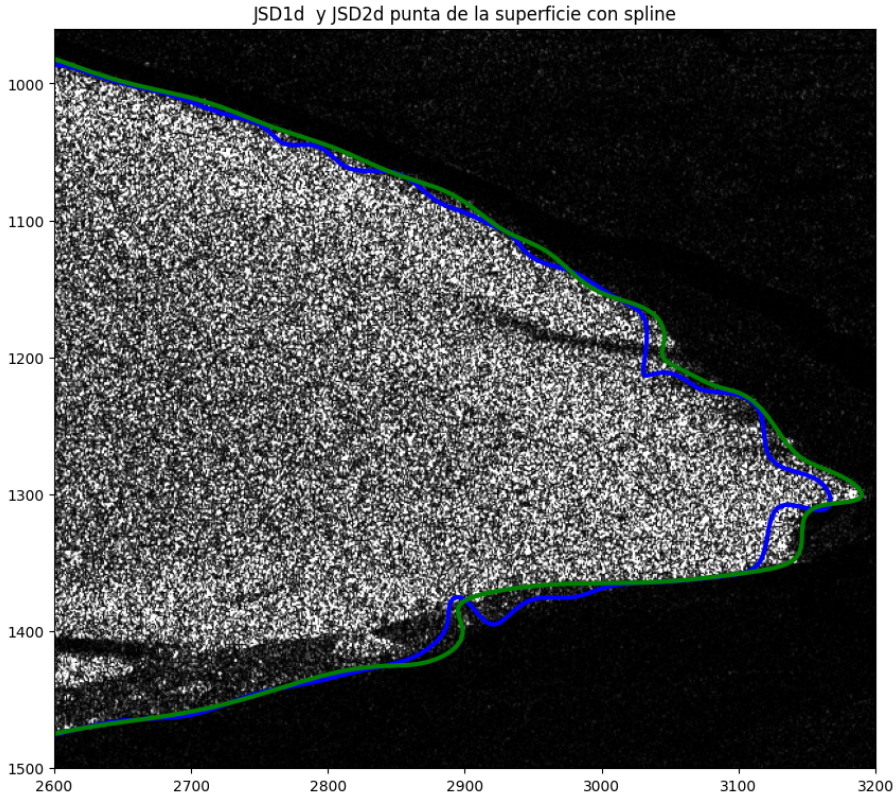


Figura 30: spline de JSD_{1d} y JSD_{2d}

6. Conclusiones

En nuestro trabajo de tesis, estudiamos y analizamos una forma de solucionar el problema de detección de bordes en imágenes SAR en superficies de hielo antárticas mediante el análisis y la aplicación de algoritmos de grafos. Para lograr esto, implementamos un método que transforma el problema en un problema de grafos, donde nuestra solución resulta más clara.

Presentamos una solución semiautomática que se puede aplicar fácilmente a cualquier imagen de entrada SAR generada por la divergencia de Jensen-Shannon, como por ejemplo JSD_{1d} o JSD_{2d}. En esta solución, debemos ingresar manualmente el punto inicial y final del borde que deseamos detectar, junto con los parámetros de entrada que nos permiten elegir el tipo de filtro y el rango de creación del grafo. Esto nos ayuda a evitar la aparición de bordes no deseados en el resultado final. En caso de tener que detectar dos bordes en una misma imagen, debemos ejecutar el algoritmo dos veces, utilizando los puntos de entrada correspondientes para cada borde.

Es importante destacar que el cálculo de la JSD toma actualmente 40 minutos, por lo que el proceso de búsqueda de bordes implementado es óptimo ya que tarda alrededor de 1 minuto.

6.1. Trabajos futuros

Si bien obtenemos un resultado, existen distintos aspectos que podemos mejorar. Por ejemplo, sería posible implementar una detección completamente automática de todos los distintos bordes de la superficie. Es decir, ya no será necesario ingresar un punto inicial y final para la detección del borde. Para imágenes que contengan otros bordes de interés o bordes ramificados, se podría correr el algoritmo presentado. Sin embargo, previamente habría que estudiar la forma apropiada de filtrar los puntos ya presentes en el borde detectado, así como analizar los parámetros apropiados del filtro espacial.

Para mejorar todo el procesamiento, algunos algoritmos podrían implementarse en un lenguaje de más bajo nivel o paralelizar el procesamiento en GPU. De esta manera, se aceleraría el tiempo de respuesta y se podrían utilizar imágenes SAR mucho más complejas o de mayor tamaño, incluida la JSD.[17]

Referencias

[1] M. Tru, K. C. Chin, and J.W. Goodman., When is speckle noise multiplicative?, Applied Optics, 21 (1982), pp.1157-1159.

[2] C. Oliver, and S. Quegan., Understanding Synthetic Aperture Radar Images., Artech House 1998.

[3] M. J. Meester, and A. S. Baslamisli., SAR image edge detection: review and benchmark experiments, International Journal Remote Sens, 14 (2022), pp.1258-1288.

[4] Ivana Kottasová, El iceberg más grande del mundo se desprende de la Antártida, CNN, 19 Mayo (2021).

[5] M. A. Ré, G.G. Aguirre Varela, and S. Masuelli., A proposal for edge detection in SAR images, Actas VII Congreso de Matemática Aplicada, Computacional e Industrial, (2019), pp.485-488.

[6] S. Masuelli, G. Aguirre Varela, and Miguel Re, Edge detection in SAR images by two dimensional Jensen Shannon Divergence, -, - (2023).

[7] Douglas West, Introduction to Graph Theory, West, 2001, Capítulo 9.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein, Introduction to Algorithms, Third Edition, Addison-Wesley, 2009, Capítulo 21.

[9] William Fiset, Union Find Kruskal's Algorithm, 7 abril (2017).

[10] Jon Kleinberg, Eva Tardos, Algorithm Design, Addison-Wesley, 2005, Capítulo 3.

[11] William Fiset, Depth First Search, 1 abril (2018).

[12] Travis E. Oliphant, Guide to NumPy, Trelgol Publishing, 2006.

[13] Eric Jones, Travis Oliphant, Pearu Peterson, SciPy: Open Source Scientific Tools for Python, Trelgol Publishing, Computing in Science Engineering, 2001.

[14] Aric A. Hagberg, Daniel A. Schult, Pieter J. Swart, NetworkX: High-productivity software for complex networks, Computational Science Discovery, 2008.

[15] John D. Hunter, Matplotlib: A 2D Graphics Environment, Computing in Science Engineering, 2007.

[16] Brett Slatkin, Effective Python: 90 Specific Ways to Write Better Python, Addison-Wesley, 2015.

[17] David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach,Morgan Kaufmann, 2010.