PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

ACELERACIÓN EN LA PLANIFICACIÓN DE RUTAS EN 3D PARA ROBOTS AUTÓNOMOS EN MINERÍA SUBTERRÁNEA

Tesis para obtener el título profesional de Ingeniero Electrónico

AUTOR:

Alvaro Huiman Tocto

ASESOR:

César Alberto Carranza De La Cruz

Lima, Agosto, 2025

Informe de Similitud

Yo, **Cesar Alberto Carranza de la Cruz**, docente de la Facultad de **Ciencias e Ingeniería** de la Pontificia Universidad Católica del Perú, asesor de la tesis titulada "**ACELERACIÓN EN LA PLANIFICACIÓN DE RUTAS EN 3D PARA ROBOTS AUTÓNOMOS EN MINERÍA SUBTERRÁNEA"**, del autor **Alvaro Huiman Tocto**,

dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de 15 %. Así lo consigna el reporte de similitud emitido por el software *Turnitin* el 22/07/2025.
- He revisado con detalle dicho reporte y la Tesis o Trabajo de Suficiencia Profesional, y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

Lugar y fecha: Lima, 22 de julio de 2025

Apellidos y nombres del asesor: Carranza de la Cruz, Cesar Alberto	
DNI: 09641576	Firma
ORCID: 0000-0003-1222-0118	

Dedico esta tesis a mis padres, quienes siempre
me apoyaron y confiaron en mí, y a mis hermanos, que
estuvieron a mi lado en todo momento.

Resumen

Cada año, el sector minero sigue innovando con el uso de nuevas tecnologías para mejorar la producción y la eficiencia. Además, debido al incremento de accidentes en las áreas mineras, se están utilizando robots para realizar las funciones de los obreros y así evitar accidentes. Estos robots también se utilizan para analizar mejor las áreas subterráneas, proporcionando información detallada sobre las zonas donde se encontrarán los trabajadores. Por ello, los robots se están convirtiendo en una parte esencial del trabajo en minería, ya que pueden generar mapas tridimensionales mediante sensores LiDAR y otros dispositivos.

Además, los algoritmos de planificación de rutas permiten a los robots recorrer zonas desconocidas y recopilar información para generar mapas en tiempo real. Sin embargo, estos robots todavía presentan ineficiencias, ya que tardan en generar mapas y no lo hacen con la precisión deseada. Por ello, esta tesis busca optimizar ciertos algoritmos que conforman la funcionalidad del robot.

En concreto, se enfocará en la serialización de datos para la comunicación de nodos en el sistema ROS y en la planificación de rutas que se basa en algoritmos para encontrar las distancias más cortas en un grafo, como Dijkstra o Bellman-Ford. La tesis se centra en optimizar estos dos algoritmos para mejorar el tiempo de computación. Esto se logrará mediante el uso de recursos como la GPU, que permite paralelizar los algoritmos para obtener un mejor tiempo de ejecución, haciendo así que el robot sea más eficiente.

Abstract

Each year, the mining sector continues to innovate through the use of new technologies to improve production and efficiency. Furthermore, due to an increase in accidents in mining areas, robots are being utilized to perform the functions of human workers, thereby preventing accidents. These robots are also used to better analyze underground areas, providing detailed information about where workers will be located. Consequently, robots are becoming an essential part of mining operations, as they can generate three-dimensional maps using LiDAR sensors and other devices.

Additionally, route planning algorithms allow robots to navigate unknown areas and collect information to generate real-time maps. However, these robots still exhibit inefficiencies, as they are slow to generate maps and do not do so with the desired precision. Therefore, this thesis aims to optimize certain algorithms that constitute the robot's functionality.

Specifically, the focus will be on data serialization for node communication in the ROS system and route planning based on algorithms for finding the shortest distances in a graph, such as Dijkstra or Bellman-Ford. The thesis centers on optimizing these two algorithms to improve computation time. This will be achieved by utilizing resources like the GPU, which allows for the parallelization of algorithms to obtain better execution times, thus making the robot more efficient.

Índice General

Índice de Figuras

Capítulo 1

Introducción

1.1 Marco problemático

La seguridad y el bienestar de los trabajadores son cuestiones de suma importancia en la industria minera. Los accidentes y enfermedades laborales pueden acarrear consecuencias graves tanto para los empleados como para las empresas que operan en este sector. Dichos incidentes resultan en costos significativos, ya sea de forma directa, mediante el financiamiento de gastos médicos y compensaciones a los trabajadores afectados, o de manera indirecta, debido a la interrupción de la producción y posibles sanciones regulatorias, como se menciona en el estudio [1].

Las estadísticas indican que la mayoría de los accidentes en la industria minera se originan por errores humanos, falta de capacitación adecuada o la limitada experiencia de los trabajadores en este entorno especializado [2]. Además, un estudio realizado en China en 2016, que se presenta en la Figura 1.1, resalta que las principales causas de estos incidentes incluyen la explosión de gases inflamables y el colapso de techos, siendo estos los factores predominantes en la ocurrencia de accidentes en la minería [3].

En el mismo contexto, en el año 2023, el Ministerio de Energía y Minas (MINEM) en Perú llevó a cabo un análisis de estadísticas acerca de las causas más comunes de los accidentes en la industria minera, como se muestra en la Figura 1.2.

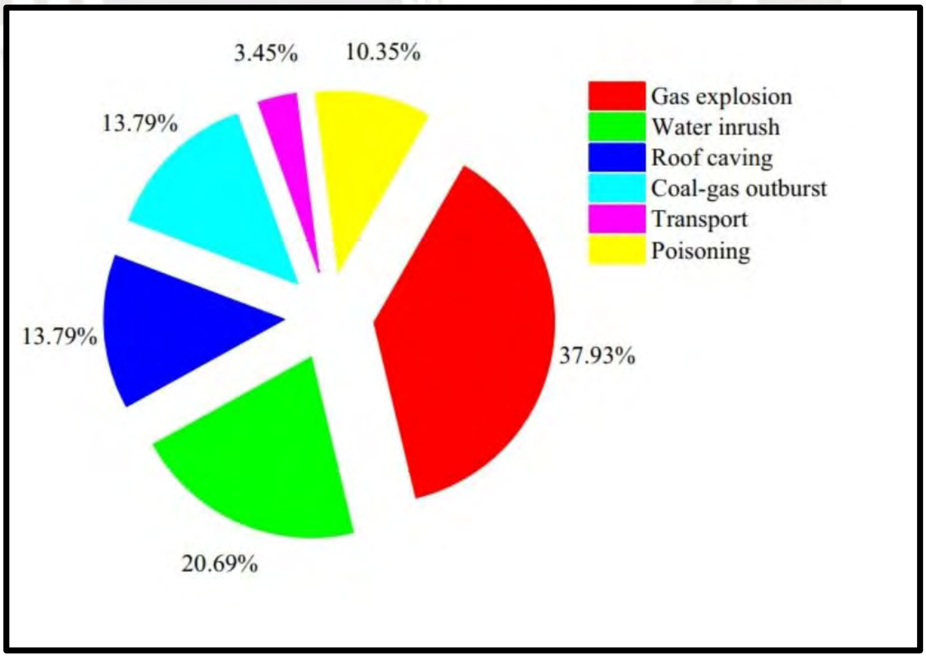


Figura 1.1: Estadísticas de la minería china en el 2016 [3].

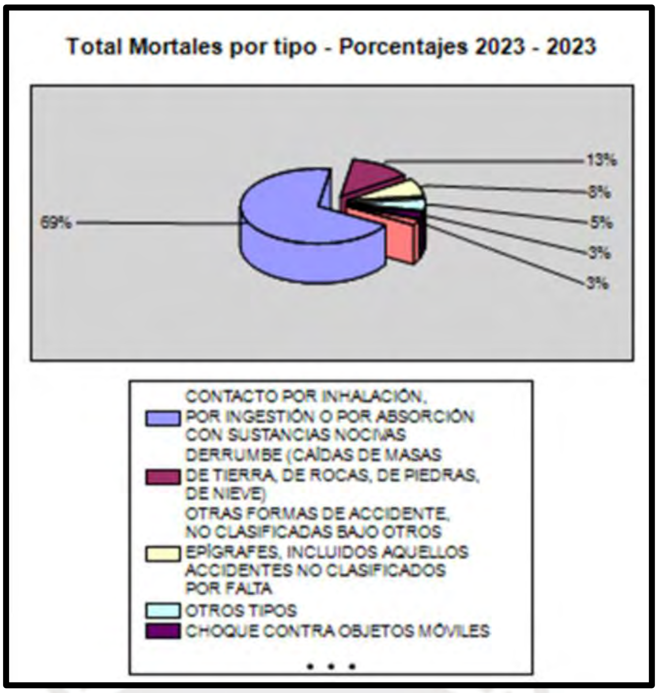


Figura 1.2: Estadísticas del MINEM del 2023 [4].

Por otro lado, se encontró que un comportamiento que carece de suficiente seguridad puede ocasionar errores por parte de las personas, mientras que la falta de seguridad en el equipo provoca fallos en su funcionamiento. Al mismo tiempo, la falta de seguridad en el entorno puede aumentar la vulnerabilidad de dicho entorno, y los errores de gestión conducirán a una gestión poco eficiente [5]. En relación con este tema, los incidentes que ocurren en las minas representan un motivo de preocupación, especialmente cuando se considera la relación entre las operaciones de extracción de carbón y el desarrollo económico en el contexto del sistema de producción social [6]. Dentro del marco de la regulación ambiental para reducir las emisiones de carbono al máximo, la discrepancia entre la extracción segura de carbón y el desarrollo económico se está volviendo un asunto cada vez más destacado. Esta situación podría aumentar el riesgo de accidentes mineros graves y resultar en pérdidas tanto económicas como humanas [6]. Asimismo, la teoría de la economía de la seguridad argumenta que el estado actual de una producción segura refleja el nivel de progreso social y económico de un país, junto con la habilidad de los gobiernos en la gestión de esta situación [7].

En este contexto, se requiere vehículos autónomos capaces de reemplazar a los operarios en zonas de alto riesgo, permitiéndoles navegar de forma autónoma en tiempo real en túneles. Esto es crucial para crear mapas actualizados de los túneles, lo que permite evaluar su estado y tomar medidas preventivas para evitar accidentes. Además, es esencial contar con algoritmos que faciliten esta navegación autónoma y en tiempo real, contribuyendo así a la seguridad y eficiencia en la operación subterránea.

1.2 Estado del Arte

- *FAST-SLAM (Fast Simultaneous Localization and Mapping):*

  El algoritmo FAST-SLAM combina de manera eficaz las ventajas de la asociación de datos de hipótesis múltiples y una complejidad computacional lineal, lo que le permite abordar con éxito el desafío de la localización y mapeo simultáneos en entornos robóticos [8]. Para lograr esto, FAST-SLAM hace uso de la estrategia de filtrado de partículas Rao-Blackwellized para estimar con precisión la trayectoria del robot, al mismo tiempo que emplea el método EKFSLAM para estimar la ubicación de los puntos de referencia [8]. En la Figura 1.3 se aprecia el entorno y los puntos de referencia que se utilizan como base para trazar una ruta, empleando el algoritmo FAST-SLAM.

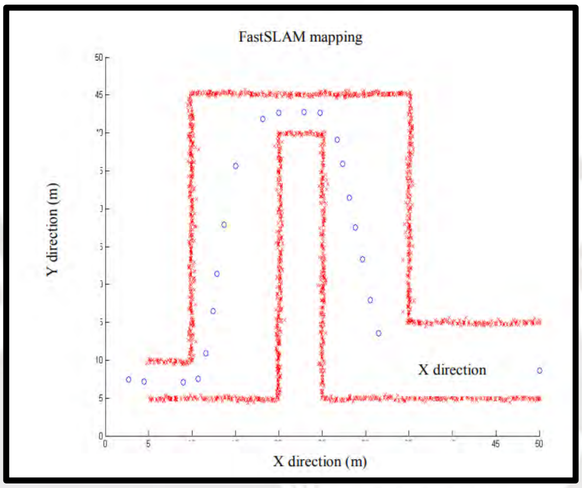


Figura 1.3: Mapeo con FAST-SLAM [8].

En el contexto de su representación, es importante resaltar que la complejidad computacional de FAST-SLAM se mantiene en un tiempo lineal en función del número de puntos de referencia, multiplicado por el número de partículas asociadas durante el cálculo de la trayectoria del robot [9]. Además, FAST-SLAM demuestra propiedades avanzadas de asociación de datos, lo que significa que solo incorpora y fusiona en el mapa los datos más precisos [10]. Cabe destacar que este versátil algoritmo es compatible con una variedad de sensores, incluyendo LiDAR y cámaras.

Para este algoritmo existen otras 3 versiones como el FAST-SLAM 1.0, FAST-SLAM 2.0 y L-SLAM. En la Tabla 1.1, se presenta una comparación entre estas tres versiones en función del número de partículas, y se concluye que L-SLAM es la opción preferida debido a su mayor precisión y su menor costo computacional en comparación con FAST-SLAM 2.0 [11].

Algorithm	Number of particles	Time/step (ms)	Position error (m)	Features error (m)
FastSLAM 1.0	4	7.1 ms	0.55 m	0.65 m
FastSLAM 2.0	4	19.0 ms	0.15 m	0.22 m
L-SLAM	4	7.9 ms	0.19 m	0.26 m
FastSLAM 1.0	40	58.0 ms	0.23 m	0.28 m
FastSLAM 2.0	40	198.0 ms	0.13 m	0.15 m
L-SLAM	40	64.0 ms	0.13 m	0.15 m

Tabla 1.1: Comparación de las 3 versiones [11].

- *ORB-SLAM (Oriented and Rotated BRIEF SLAM):*

  El algoritmo ORB-SLAM es una solución SLAM altamente versátil y precisa. Desde dispositivos portátiles hasta vehículos en movimiento a través de áreas urbanas extensas en tiempo real, este algoritmo es capaz de lograr la recuperación de la trayectoria de la cámara y una reconstrucción fragmentaria de entornos 3D. Su capacidad para cerrar bucles significativos y relocalizarse globalmente en tiempo real es destacable [12]. Además, el algoritmo ORB-SLAM se basa en tres procesos fundamentales. Estos procesos operan en paralelo y se encargan de obtener los fotogramas secuenciales para estimar el movimiento de la cámara y mapear el entorno [12]. En la Figura 1.4 se muestra un diagrama de flujo de estos pasos del algoritmo.

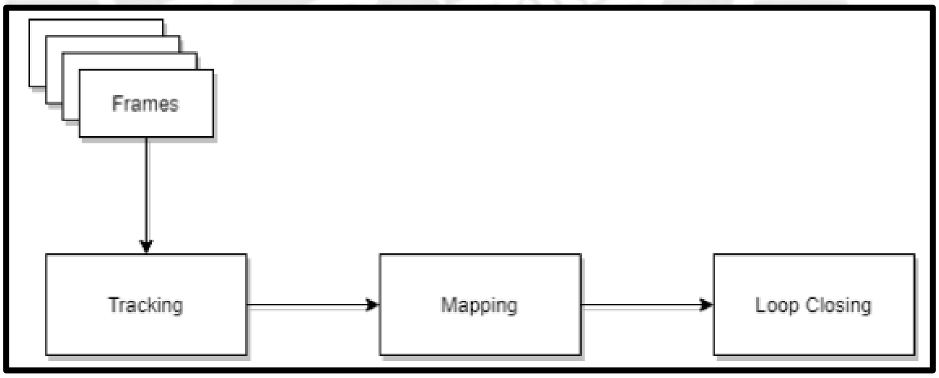


Figura 1.4: Diagrama de flujos del ORB-SLAM [12].

Por otro lado, este método SLAM visual, ORB-SLAM, se distingue por su robustez y se beneficia de la invariancia de velocidad y rotación. Esto le permite extraer características de las imágenes en tiempo real. Está diseñado específicamente para aplicaciones de procesamiento en tiempo real y aprovecha la eficiente búsqueda visual de características ORB en el espacio de la superficie de la imagen [12]. Además de esto, ORB-SLAM opera utilizando conjuntos de imágenes consecutivas para construir mapas y localizarse en ellos. Este proceso puede llevarse a cabo en tiempo real, ya que el agente procesa cada imagen tan pronto como se genera, o de manera posterior utilizando un conjunto de datos. En cada imagen, conocida como "marco", se extraen puntos visuales de relevancia [13].

- *RBPF-SLAM (Rao-Blackwellized Particle Filter Simultaneous Localization and Mapping):*

  El RBPF-SLAM es una solución basada en el filtro de partículas que se enfoca en la optimización del tiempo de ejecución. Esto se logra mediante la implementación de una distribución de propuestas precisa y una estrategia de remuestreo selectivo, lo que conduce a una reducción significativa en la cantidad de partículas requeridas [14]. Por otro lado, en este enfoque, la predicción de la función de distribución propuesta en el RBPF-SLAM se basa en datos de odometría. Sin embargo, esta dependencia en la odometría puede complicar la integración de información adicional en el proceso [15]. La idea central de este algoritmo se basa en que cada partícula lleva consigo un mapa local del entorno. En un primer paso, las partículas son extraídas del modelo de movimiento. A continuación, se procede a volver a muestrear basado en sus pesos de importancia. Por último, se estima el mapa mediante el seguimiento de las trayectorias de las partículas y el historial de observaciones [16]. En la Figura 1.5 se muestra el diagrama de flujo de este algoritmo.

  El resultado de este algoritmo es la creación de un mapa global del entorno que rodea al robot. El proceso de generación del mapa global se divide en cuatro etapas: actualización en movimiento, actualización de mapas locales, ponderación de importancias y actualización de mapas globales [16].

  En la Figura 1.6 se muestra el mapa global utilizando este algoritmo para una trayectoria como se muestra en la Figura 1.7.

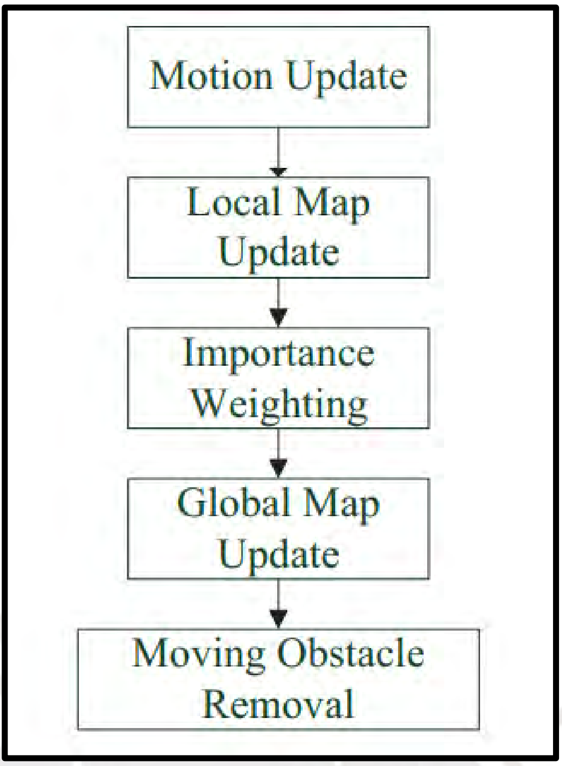


Figura 1.5: Diagrama de flujo de RBPF-SLAM [16].

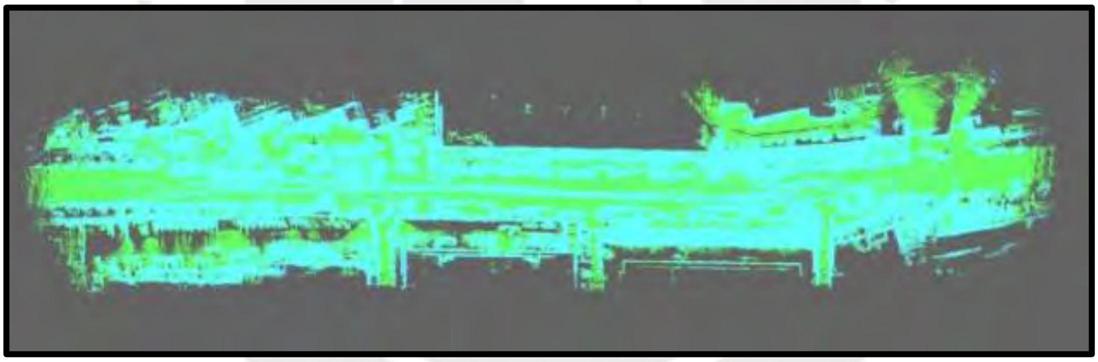


Figura 1.6: Mapa global de RBPF-SLAM [16].

Figura 1.7: Trayectoria del experimento [16].

FAST-SLAM, ORB-SLAM y RBPF-SLAM son tres algoritmos utilizados en el campo del Simultaneous Localization and Mapping (SLAM). Por un lado, FAST-SLAM emplea un filtro de partículas para mantener una distribución de creencias sobre la ubicación del robot y el mapa del entorno, siendo adecuado para entornos con muchas características y alta incertidumbre. ORB-SLAM, por su parte, se basa en características ORB para detectar y seguir puntos de referencia en el entorno, siendo eficiente y adecuado para entornos con estructuras planas y texturas distintivas.

Por último, RBPF-SLAM es una variante de FAST-SLAM que utiliza el filtro de partículas Rao-Blackwellizado para una estimación más precisa de la ubicación del robot y puede ser más eficiente en términos de recursos computacionales cuando la incertidumbre en la ubicación del robot es alta. La elección entre estos algoritmos depende de las características del entorno y los recursos disponibles, así como de la precisión requerida en la estimación de la posición y el mapa del robot.

1.3 Justificación

Justificación para el desarrollo de tesis:

- Optimizar la planificación de rutas en 3D para robots autónomos en minería subterránea permitiría una gestión más eficiente de las operaciones mineras. Los robots autónomos pueden navegar de manera más rápida y precisa, lo que conduce a una mayor productividad y ahorro de tiempo en la ejecución de tareas críticas.
- La navegación autónoma utilizando robots reduce el riesgo para los trabajadores al minimizar su exposición a entornos potencialmente peligrosos y permitir que los robots realicen tareas de inspección y exploración en áreas de difícil acceso o riesgo elevado. Esto contribuye a un entorno laboral más seguro.
- Una planificación de rutas más eficiente puede llevar a una disminución en los costos operativos. Los robots autónomos pueden evitar rutas menos eficientes, lo que resulta en un menor consumo de energía y una mayor vida útil de los equipos. Además, se

obtiene información valiosa sobre el estado de la mina, la geología subterránea y otros datos críticos.

Para lograr esto se necesita navegación autónoma en tiempo real lo cual hace necesario contar con algoritmos paralelos.

1.4 Objetivos

1.4.1 Objetivo general

- El objetivo fundamental de esta tesis es potenciar la aceleración del tiempo de ejecución de los algoritmos empleados en la planificación de rutas para robots autónomos. El propósito es crear un mapa en tiempo real de una mina, haciendo uso de los datos suministrados por sensores LiDAR y unidades de medición inercial (IMU), con el propósito de identificar y seleccionar las mejores rutas para maximizar las características deseables del entorno. De esta forma, se posibilita que el robot pueda navegar de manera más eficiente en términos de consumo de energía y latencia.

1.4.2 Objetivo específicos

- Realizar un análisis de la captura de datos del robot (LiDAR, IMU) y su procesamiento para ser usado como insumo en la navegación autónoma.
- Estudio y selección de algoritmos de navegación autónoma paralelizables.
- Implementación paralela del algoritmo de navegación seleccionado.
- Análisis de la performance del algoritmo implementado en términos de tiempo de ejecución. Comparación con otros algoritmos.

Capítulo 2

Marco teórico

2.1 Sensor LiDAR (Light Detection And Ranging) y diseño de voxel

Un sensor LiDAR se utiliza para calcular con precisión la distancia a un objeto y se compone de elementos clave que incluyen un diodo láser (LD), un fotodiodo de avalancha (APD) y un conversor de tiempo a formato digital (TDC) [17]. En la Figura 2.1.1 se muestra el funcionamiento de un sensor LiDAR.

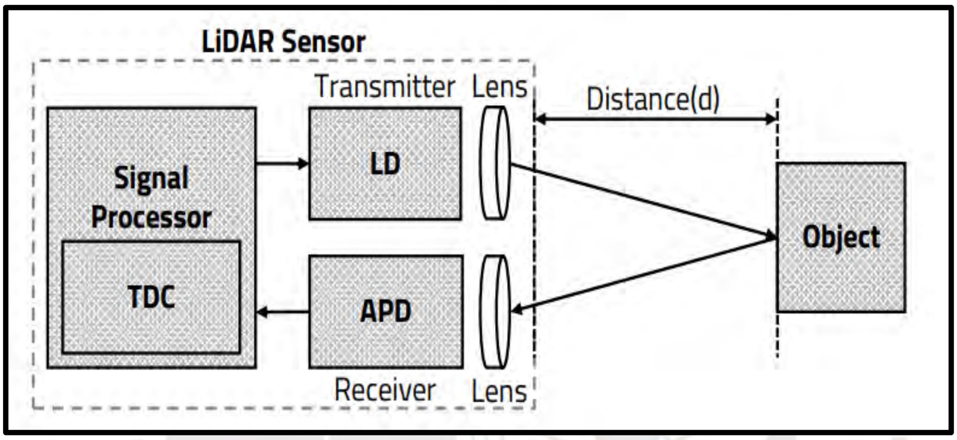


Figura 2.1.1: Diagrama de funcionamiento del sensor LiDAR [18].

El diodo láser (LD) emite un rayo láser que se enfoca utilizando una lente transmisora de luz. Este láser se refleja en el objeto y es capturado por el fotodiodo de avalancha (APD) a través de una lente receptora de luz. Luego, el conversor de tiempo a formato digital (TDC) mide la diferencia entre el momento en que el diodo láser emite el láser y el momento en que el APD lo recibe, transformando esta diferencia en un tiempo de vuelo (ToF) [18].

La unidad de procesamiento de señales, o microprocesador (MP), recibe el valor de Tiempo de Vuelo (ToF) del Convertidor de Digital a Tiempo (TDC). Con este dato, el microprocesador calcula la distancia entre el sensor LiDAR y el objeto [18].

Por otra parte, el LiDAR produce una nube de puntos en un espacio tridimensional, y esta está vinculada a cuatro parámetros clave del sensor: resolución angular vertical (VAR), resolución angular horizontal (HAR), campo de visión horizontal (HFoV) y campo de visión vertical (VFoV). A modo de ilustración, en la Figura 2.1.2 se presenta un sensor LiDAR de 16 canales con un VFoV de 9.6° y un HFoV de 145°.

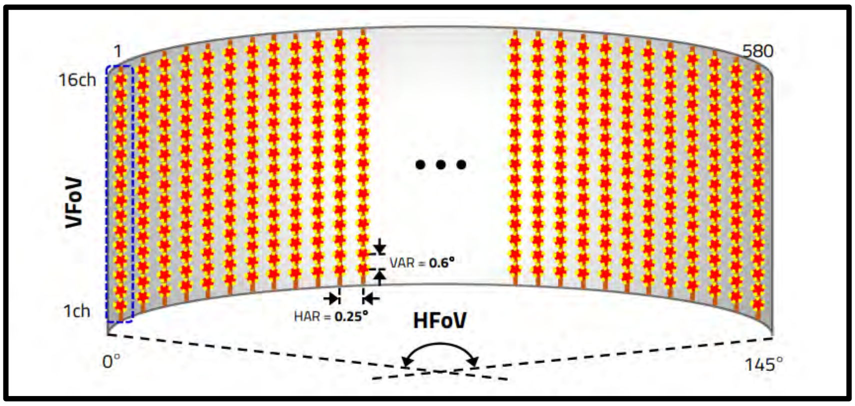


Figura 2.1.2: Sensor LiDAR de 16 canales [18].

Por otro lado, la nube de puntos capturada por el sensor LiDAR debe ser convertida en voxels para crear el mapa tridimensional. Este proceso se lleva a cabo mediante el método de cuadrícula multiescala, que comprende tres fases. En la primera etapa, se utiliza un algoritmo de filtrado para identificar puntos terrestres y no terrestres en una rejilla a gran escala, eliminando la influencia del relieve topográfico en la detección de edificaciones. En la segunda etapa, se emplea un método de interpolación morfológica jerárquica en una rejilla a pequeña escala para generar una imagen de profundidad de alta resolución. Luego, se utiliza un algoritmo de segmentación para detectar características precisas y detalladas de los edificios. En la tercera etapa, se detectan las características detalladas de los techos de los edificios y se reconstruyen modelos 3D de los techos según la elevación de las características. Se utiliza la interpolación de nubes de puntos no organizadas en dos conjuntos de rejillas con diferentes resoluciones espaciales, una a gran escala y otra a pequeña escala, para evitar la pérdida de precisión de datos [19]. En la Figura 2.1.3 se muestra el diagrama de flujo de este método.

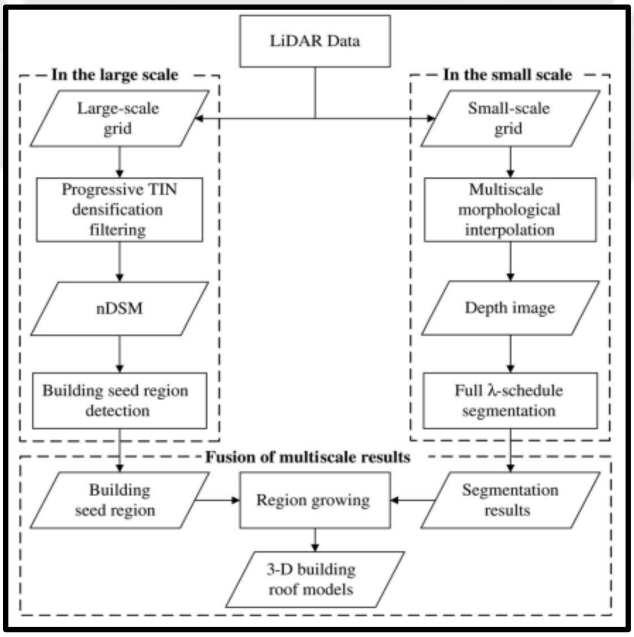


Figura 2.1.3: Diagrama de flujo de cuadrícula multi escala [19].

Para el campo de la navegación autónoma, un sensor LiDAR ampliamente utilizado en algoritmos SLAM es el Ouster. Posee un campo de visión vertical de 90°, campo de visión horizontal de 360° y puede detectar objetos a una distancia de hasta 50 metros. En la Figura 2.1.4, se representa el modelo OS0-32, el cual desempeña un papel crucial en la percepción del entorno para la planificación de rutas.

Figura 2.1.4: Sensor LiDAR Ouster [20].

2.2 Sensor IMU (Inertial Measurement Unit)

El IMU es empleado principalmente en dispositivos destinados a la medición de velocidad, orientación y fuerza gravitatoria. Se puede dividir en dos componentes principales. La primera tecnología comprende dos tipos de sensores, conocidos como acelerómetros y giroscopios. El acelerómetro se utiliza para cuantificar la aceleración inercial, mientras que el giroscopio se encarga de medir la rotación angular. Ambos sensores suelen disponer de tres grados de libertad para medir a lo largo de tres ejes distintos [21].

Por lo general, cada sensor está configurado con alrededor de dos a tres grados de libertad asignados a los ejes ‘x’, ‘y’, y ‘z’. Por lo tanto, la combinación de ambos sensores proporciona un rango total de cuatro a seis grados de libertad. Los valores de aceleración obtenidos del acelerómetro y la velocidad angular registrada por los giroscopios se mantienen de manera independiente. Por otro lado, los ángulos medidos por los dos sensores se pueden calibrar para obtener un valor más preciso [21]. En la Figura 2.1.5 se muestra un diagrama de la distribución de las señales de cada sensor.

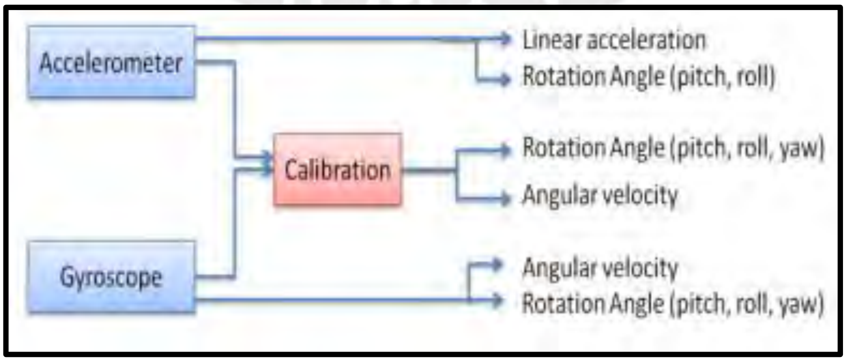


Figura 2.1.5: Distribución de señales de sensor IMU [21].

2.3 Teoría de Grafos y sus tipos

Un grafo (G) está formado por un conjunto de vértices o nodos (V) y un conjunto de aristas (E), donde un vértice puede representar un objeto individual, y si se trata de dos objetos relacionados, se representarían mediante una arista [22]. Por lo tanto, un grafo puede utilizarse para representar cualquier información que pueda modelarse como objetos y las relaciones entre esos objetos [22]. En la Figura 2.1.6 se presenta un grafo compuesto por pequeños círculos que representan los vértices o nodos, mientras que las líneas y curvas representan las aristas.

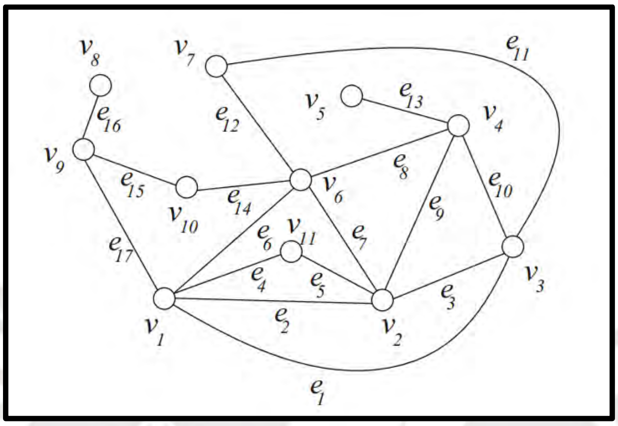


Figura 2.1.6: Ejemplo de grafo [22].

Por otro lado, un grafo puede recibir dos tipos de denominación: como un "grafo ponderado" o como un "grafo dirigido" (también conocido como "dígrafo"). En el primer escenario, se habla de un "grafo ponderado" cuando se le asigna un peso a sus aristas o vértices, tal como se representa en la Figura 2.1.7. En el segundo caso, se refiere a un "grafo dirigido" cuando cada arista se orienta en una dirección específica, como se ejemplifica en la Figura 2.1.8.

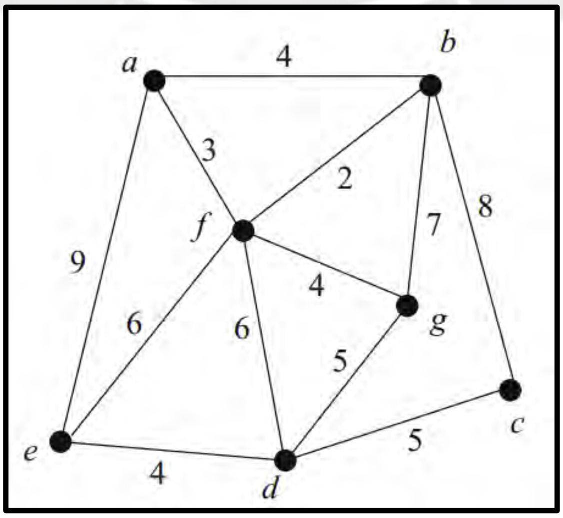


Figura 2.1.7: Grafo ponderado [22].

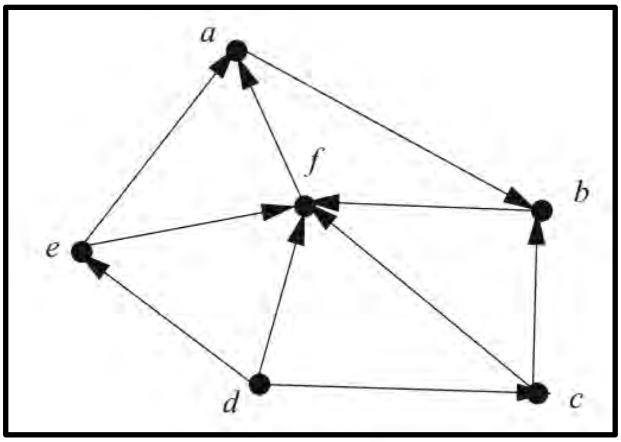


Figura 2.1.8: Grafo dirigido [22].

2.4 GBPLANNER (Graph-Based exploration path Planner)

Es un método que permite la exploración y el mapeo autónomo de lugares subterráneos. Hasta la actualidad se usan dos versiones que son el GBPlanner1 y GBPlanner2. Esta última revisión se ha optimizado para abordar desafíos en el manejo de geometrías subterráneas complicadas y pendientes pronunciadas, al mismo tiempo que ofrece un mejor rendimiento computacional [23].

El método GBPlanner se sustenta en una arquitectura bifurcada que comprende la planificación de rutas a nivel global y local, ambas adaptadas a un dominio subterráneo [24].

2.4.1 Planificador de exploración local

El diseño busca una exploración ágil de entornos subterráneos usando un algoritmo de grafos aleatorios [24]. El planificador local calcula la ganancia volumétrica para cada vértice, que representa el volumen no mapeado que un sensor de rango S percibirá, basándose en la proyección de rayos y siendo adaptable a múltiples sensores de profundidad [24]. Esta ganancia, junto con otras funciones de distancia y dirección, se usa para calcular la ganancia de exploración de cada ruta más corta, como se muestra en la Ecuación 2.1.

$$ExplorationGain(\sigma_i) = e^{-\gamma s S(\sigma_i, \sigma_{exp})} \sum_{j=1}^{m_i} VolumetricGain(v_j^i) e^{-\gamma D D(v_1^i, v_j^i)} \qquad (2.1)$$

Donde:

- $S(\sigma_i, \sigma_{exp})$, $D(v_1^i, v_j^i)$ son funciones de peso
- $\gamma s, \gamma D$ son factores sintonizables
- $D(v_1^i, v_j^i)$ es la distancia acumulada euclidiana

En la Figura 2.1.9 se representa un escenario específico en el que se observa un robot equipado con un sensor LiDAR. En este contexto, el algoritmo se encarga de determinar la cantidad de vóxeles desconocidos que pueden ser penetrados por los rayos del sensor, lo que a su vez permite calcular la ganancia volumétrica.

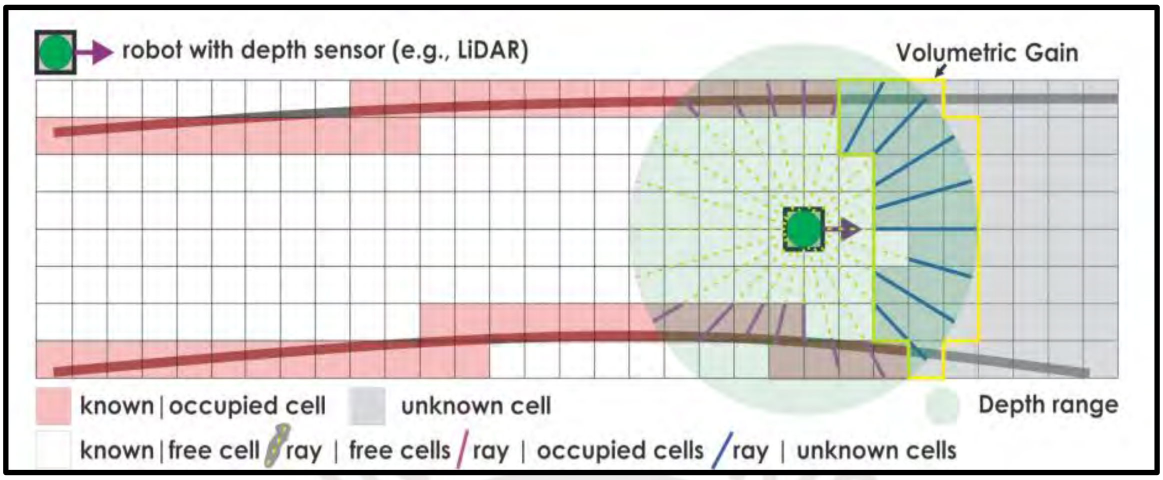
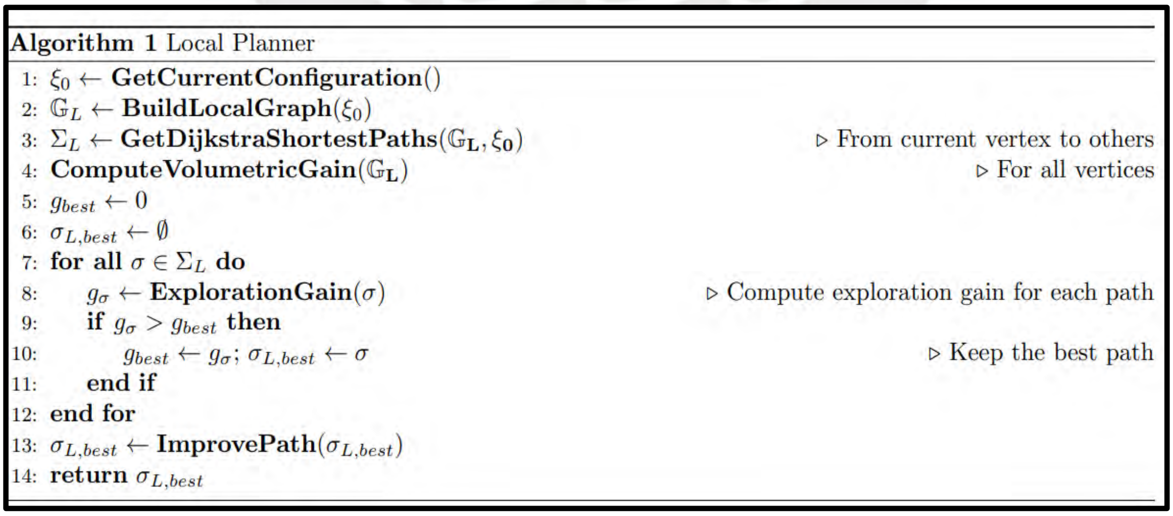


Figura 2.1.9: Escenario para cálculo de la ganancia volumétrica [24].

El algoritmo del planificador de exploración local tiene como objetivo determinar las ganancias volumétricas de cada vértice en un grafo local, con el fin de encontrar la ruta más adecuada para la exploración del robot. Este algoritmo se basa en la obtención de la configuración actual del robot y la construcción de un grafo local utilizando dichos parámetros. A partir de este grafo, se generan múltiples trayectos cortos por medio del algoritmo de Dijkstra. Además, se evalúa la ganancia de exploración asociada a cada uno de estos caminos, y se utiliza este valor para identificar la mejor ruta de exploración libre. La Figura 2.2.1 presenta el pseudocódigo del proceso de planificación local, mientras que la Figura 2.2.2 ofrece una visualización detallada de cada etapa de dicho proceso.



Algorithm 1 Local Planner

```
$\xi_0 \leftarrow$ GetCurrentConfiguration()
$\mathbb{G}_L \leftarrow$ BuildLocalGraph($\xi_0$)
$\Sigma_L \leftarrow$ GetDijkstraShortestPaths($\mathbb{G}_L, \xi_0$)    ▷ From current vertex to others
ComputeVolumetricGain($\mathbb{G}_L$)    ▷ For all vertices
$g_{best} \leftarrow 0$
$\sigma_{L,best} \leftarrow \emptyset$
for all $\sigma \in \Sigma_L$ do
    $g_\sigma \leftarrow$ ExplorationGain($\sigma$)    ▷ Compute exploration gain for each path
    if $g_\sigma > g_{best}$ then
        $g_{best} \leftarrow g_\sigma$; $\sigma_{L,best} \leftarrow \sigma$    ▷ Keep the best path
    end if
end for
$\sigma_{L,best} \leftarrow$ ImprovePath($\sigma_{L,best}$)
return $\sigma_{L,best}$
```

Figura 2.2.1: Pseudocódigo de planificador de exploración local [24].

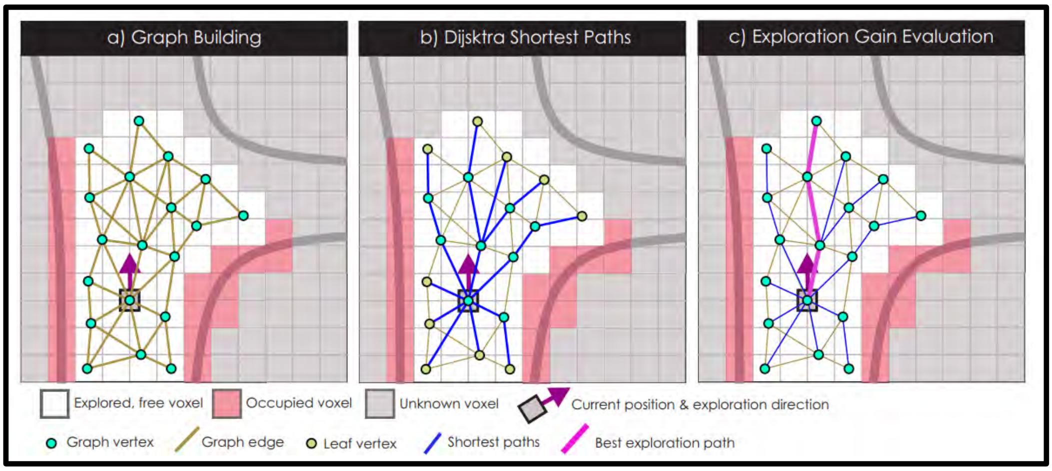


Figura 2.2.2: Etapas de planificador de exploración local [24].

2.4.2 Planificador de exploración global

El planificador global examina el espacio global del entorno actualmente investigado y ofrece dos funciones fundamentales que aseguran la capacidad de expansión de la autonomía de exploración en entornos subterráneos. En primer lugar, busca rutas alternativas en áreas no exploradas cuando el planificador local no puede sugerir caminos efectivos debido a limitaciones locales. En segundo lugar, gestiona la localización crítica cuando el tiempo restante se acerca al límite establecido, como la duración de la batería o el tiempo asignado por el usuario [24].

Asimismo, se emplea la ganancia volumétrica para calcular la ganancia de exploración global, tal como se indica en la Ecuación 2.2.

$$GlobalExplorationGain(V_{G,i}^{F}) = \tau(v_{G,cur}, v_{G,i}^{F})VolumetricGain(v_{G,i}^{F})e^{-\varepsilon D(v_{G,cur}, v_{G,i}^{F})} \quad (2.2)$$

Donde:

- $\tau(v_{G,cur}, v_{G,i}^{F})$ el tiempo de exploración restante estimado
- $D(v_{G,cur}, v_{G,i}^{F})$ la longitud de ruta más corta desde la ubicación actual hasta la frontera
- $(v_{G,i}^{F})$ la frontera

La Figura 2.2.3 ilustra el funcionamiento del planificador global. En la subfigura b), se presentan dos instancias del robot utilizando el planificador local, identificando los principales caminos de exploración y estableciendo los nodos finales como posibles fronteras. Estas rutas se almacenan en el planificador global. Luego, en la sub figura a), se observa que, cuando el robot se encuentra sin salida, recurre al planificador global para seleccionar una ruta de navegación. En este caso, tiene tres opciones: la mejor ruta en fucsia, la ruta intermedia en azul o la ruta de regreso a casa en verde. La elección de la ruta se basa en la ganancia de exploración global calculada.

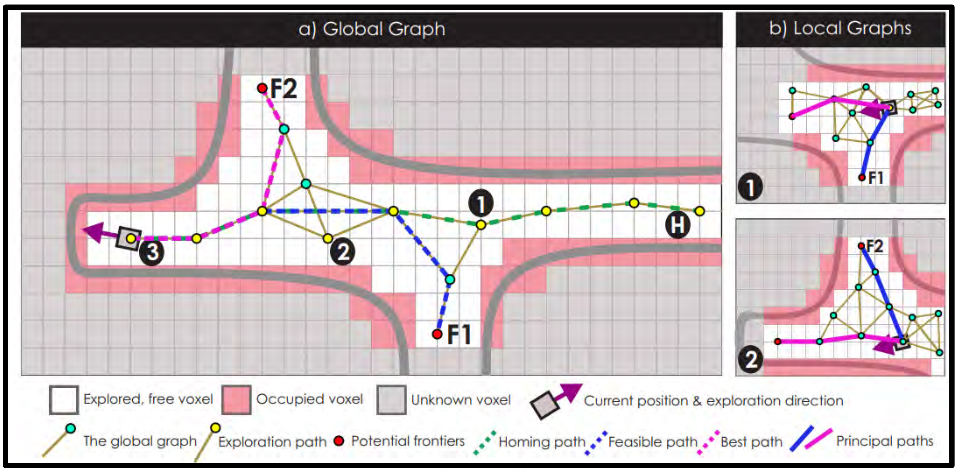
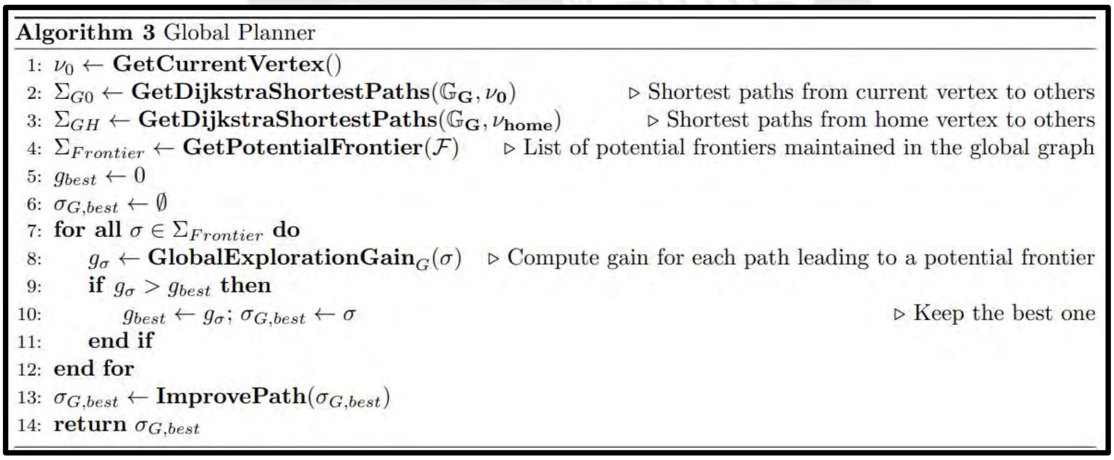


Figura 2.2.3: Etapas de planificador de exploración global [24].

El algoritmo del planificador de exploración global inicia obteniendo la configuración actual del robot, presentándola como un vértice en el gráfico global. Posteriormente, utiliza el algoritmo de Dijkstra para determinar los caminos más cortos desde el vértice actual hacia otros vértices o de regreso a casa. Además, genera una lista de todas las fronteras potenciales. A continuación, calcula la ganancia de exploración global para cada posible camino que lleva a las fronteras potenciales, seleccionando así la ruta óptima para el robot. La distribución detallada de este algoritmo se presenta en el pseudocódigo de la Figura 2.2.4.

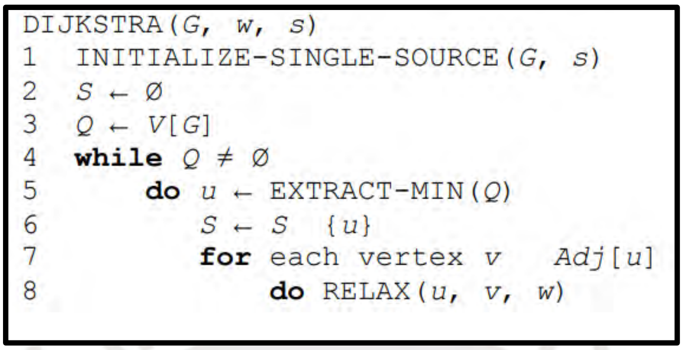


```
Algorithm 3 Global Planner
$\nu_0 \leftarrow \textbf{GetCurrentVertex}()$
$\Sigma_{G0} \leftarrow \textbf{GetDijkstraShortestPaths}(\mathbb{G}_{\mathbf{G}}, \nu_0)$    ▷ Shortest paths from current vertex to others
$\Sigma_{GH} \leftarrow \textbf{GetDijkstraShortestPaths}(\mathbb{G}_{\mathbf{G}}, \nu_{\mathbf{home}})$    ▷ Shortest paths from home vertex to others
$\Sigma_{Frontier} \leftarrow \textbf{GetPotentialFrontier}(\mathcal{F})$    ▷ List of potential frontiers maintained in the global graph
$g_{best} \leftarrow 0$
$\sigma_{G,best} \leftarrow \emptyset$
for all $\sigma \in \Sigma_{Frontier}$ do
    $g_\sigma \leftarrow \textbf{GlobalExplorationGain}_G(\sigma)$    ▷ Compute gain for each path leading to a potential frontier
    if $g_\sigma > g_{best}$ then
        $g_{best} \leftarrow g_\sigma$; $\sigma_{G,best} \leftarrow \sigma$    ▷ Keep the best one
    end if
end for
$\sigma_{G,best} \leftarrow \textbf{ImprovePath}(\sigma_{G,best})$
return $\sigma_{G,best}$
```

Figura 2.2.4: Pseudocódigo de planificador de exploración global [24].

2.5 Algoritmo DIJKSTRA

El algoritmo de Dijkstra resuelve el problema de los caminos más cortos desde una única fuente en un grafo ponderado y dirigido G = (V, E), en el caso en que todos los pesos de las aristas son no negativos [25]. En la Figura 2.2.5 se muestra en pseudocódigo la estructura de este algoritmo.



```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← Ø
3  Q ← V[G]
4  while Q ≠ Ø
5       do u ← EXTRACT-MIN(Q)
6           S ← S  {u}
7           for each vertex v   Adj[u]
8               do RELAX(u, v, w)
```

Figura 2.2.5: Pseudocódigo del Algoritmo Dijkstra [25].

En la Línea 1, se inician los valores y la Línea 2 establece S como vacío. El algoritmo mantiene la invariante Q = V - S al comenzar cada iteración del bucle while de las Líneas 4-8. La Línea 3 inicializa Q con todos los vértices en V, y en cada iteración del bucle, se extrae un vértice u de Q = V - S y se añade a S, cumpliendo la invariante. Luego, las Líneas 7-8 relajan cada arista (u, v) que sale de u, actualizando si el camino más corto a v puede mejorarse. Los vértices no se insertan en Q después de la Línea 3, y cada vértice se extrae de Q y se agrega a S exactamente una vez, asegurando que el bucle while de las Líneas 4-8 itere exactamente |V| veces [25].

La Figura 2.2.6 muestra un grafo donde el punto de partida (vértice 's') está a la izquierda junto con otros vértices que se conectan por medio de aristas que tienen un peso. Los números dentro de cada punto indican la distancia más corta estimada. Las líneas azules muestran el camino que se ha tomado para llegar a ese punto, y los puntos azules son los que ya se han explorado completamente [25].

La eficiencia del algoritmo de Dijkstra está estrechamente ligada a la implementación de la cola de prioridad mínima Q. En consecuencia, su complejidad computacional se expresa como $O((V+E)log\,V)$, siendo V el número de vértices y E el número de aristas del grafo teniendo en cuenta que se usa una cola de prioridad mínima. En cambio si no existe una cola de prioridad su complejidad sería $O(V^2+E)=O(V^2)$ resultando poco eficiente [25].

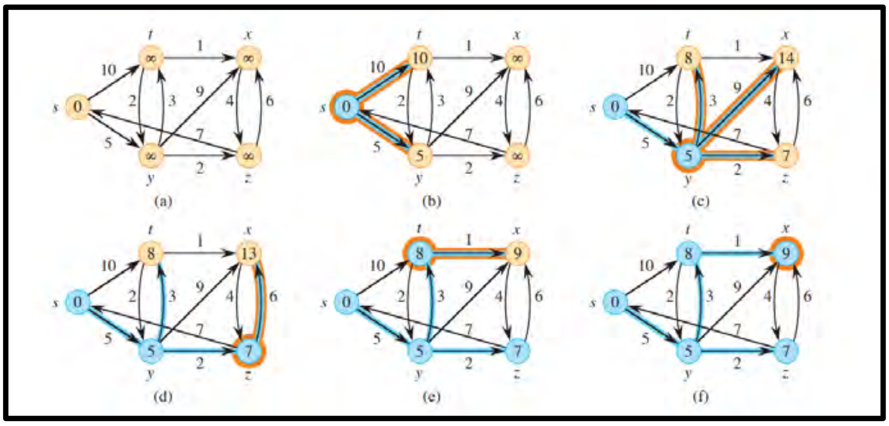
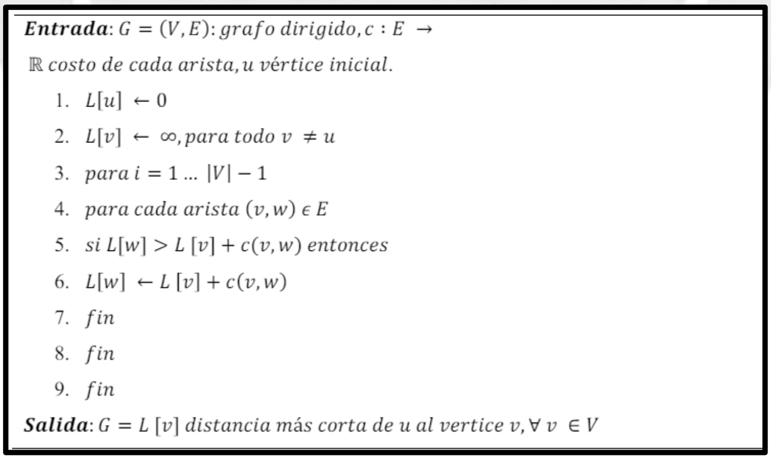


Figura 2.2.6: Configuración de un grafo [25].

2.6 Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford, al igual que el de Dijkstra, busca obtener la ruta más corta desde un vértice fuente a través de un grafo. Sin embargo, a diferencia del algoritmo de Dijkstra, el algoritmo de Bellman-Ford puede trabajar con aristas que tienen pesos negativos. Este algoritmo también maneja el concepto de ciclos de peso negativo. Si detecta un ciclo de este tipo, el algoritmo concluye que no existe una solución válida para la búsqueda de caminos cortos, ya que los ciclos negativos permiten reducir indefinidamente la longitud de la ruta [26]. En ausencia de ciclos de peso negativo, el algoritmo de Bellman-Ford encuentra las rutas más cortas mediante la relajación de aristas, un proceso iterativo que ajusta las estimaciones de las distancias más cortas [26]. La Figura 2.2.7 muestra el pseudocódigo del algoritmo de Bellman-Ford.



```
Entrada: $G = (V,E)$: grafo dirigido, $c : E \to \mathbb{R}$ costo de cada arista, u vértice inicial.
  $L[u] \leftarrow 0$
  $L[v] \leftarrow \infty$, para todo $v \neq u$
  para $i = 1 \dots |V| - 1$
    para cada arista $(v, w) \in E$
      si $L[w] > L[v] + c(v,w)$ entonces
        $L[w] \leftarrow L[v] + c(v,w)$
      fin
    fin
  fin
Salida: $G = L[v]$ distancia más corta de u al vertice v, $\forall v \in V$
```

Figura 2.2.7: Pseudocódigo del Algoritmo Bellman-Ford [26].

Por otro lado, la complejidad del algoritmo cuando un grafo está representado por una lista de adyacencia es de $O(V^2 + VE)$ donde V son los vértices del grafo y E representa a las aristas. En el caso que $|E| = \Omega(V)$ la complejidad se podría expresar de la siguiente manera $O(VE)$ [25].

2.7 FAST-LIO2 (FAST Direct LiDAR-Inertial-Odometry)

Para que un robot autónomo pueda navegar de manera segura en un entorno desconocido, es fundamental que lleve a cabo el proceso de Simultaneous Localization and Mapping (SLAM). Este procedimiento implica la creación de un mapa tridimensional (3D) en tiempo real, permitiendo la detallada representación de los entornos, mientras simultáneamente determina la ubicación del robot en dicho mapa.

La implementación de este procedimiento se lleva a cabo mediante la aplicación del método FAST-LIO2, el cual se fundamenta en un filtro de Kalman iterativo altamente eficiente y compacto, contribuyendo así a la mejora de la precisión en la navegación [27]. Este método se distingue por incorporar dos innovaciones clave que posibilitan una navegación veloz y precisa [27]. La primera innovación consiste en incorporar directamente puntos no procesados en el mapa, sin la necesidad de extraer características en un primer momento. Luego, se realiza una actualización del mapa, es decir, su mapeo. Este enfoque aprovecha las características sutiles del entorno, lo que resulta en un aumento de la precisión [27]. La segunda mejora sobresaliente consiste en la representación de un mapa de nube de puntos mediante una estructura de datos conocida como árbol k-d incremental o ikd-Tree. Esta innovación permite llevar a cabo actualizaciones de manera gradual, como la inserción o eliminación de puntos, y facilita un equilibrio dinámico. Además, su idoneidad para la odometría con LiDAR lo convierte en una elección especialmente apropiada [27].

Como ilustra la Figura 2.2.8, en un primer momento, los puntos LiDAR sin procesar, adquiridos de forma secuencial, se acumulan durante un intervalo que oscila entre 10 ms y 100 ms. Luego, con el fin de llevar a cabo la estimación del estado, se registran los puntos de un nuevo escaneo en puntos del mapa (es decir, odometría) que están mantenidos en un extenso mapa local mediante un marco de filtro de Kalman que también incorpora la calibración de los parámetros del LiDAR-IMU [27]. Este algoritmo Kalman, permite estimar el estado de un sistema dinámico a partir de una serie de mediciones ruidosas. Por otro lado, los puntos del mapa se estructuran en un árbol ikd que se expande de manera dinámica al integrar un nuevo escaneo de nube de puntos a la velocidad de la odometría [27].

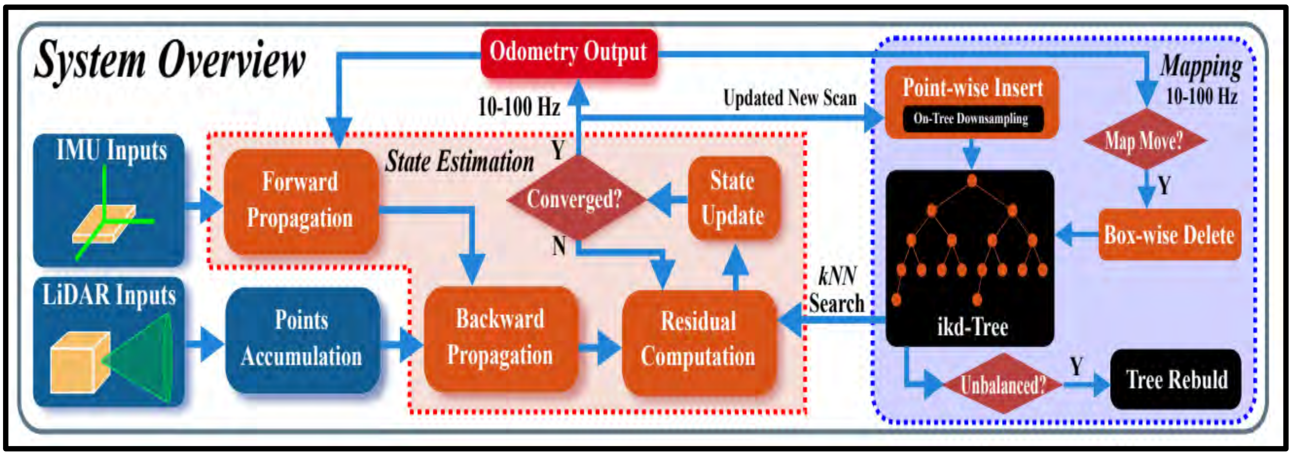


Figura 2.2.8: Descripción general de Fast-Lio2 [27].

2.8 Estructura de ejecución de códigos en CUDA

CUDA permite que el código se ejecute simultáneamente en dos procesadores diferentes: la CPU, que actúa como host principal, y la GPU, diseñada para realizar cálculos masivos y paralelos. Esta distribución de tareas optimiza el rendimiento general del programa.

El rol de la CPU es preparar y cargar datos para el procesamiento en la GPU. También se encarga de la ejecución de kernels, que son bloques de código CUDA que se ejecutan en la GPU. Por último, una vez finalizado el procesamiento en la GPU, la CPU recolecta los resultados generados por los kernels [28]. Por otro lado, la GPU realiza cálculos complejos de forma paralela utilizando sus miles de núcleos y ejecutando los kernels [28]. En la Figura 2.2.9 se muestra la estructura de ejecución de la implementación en CUDA.

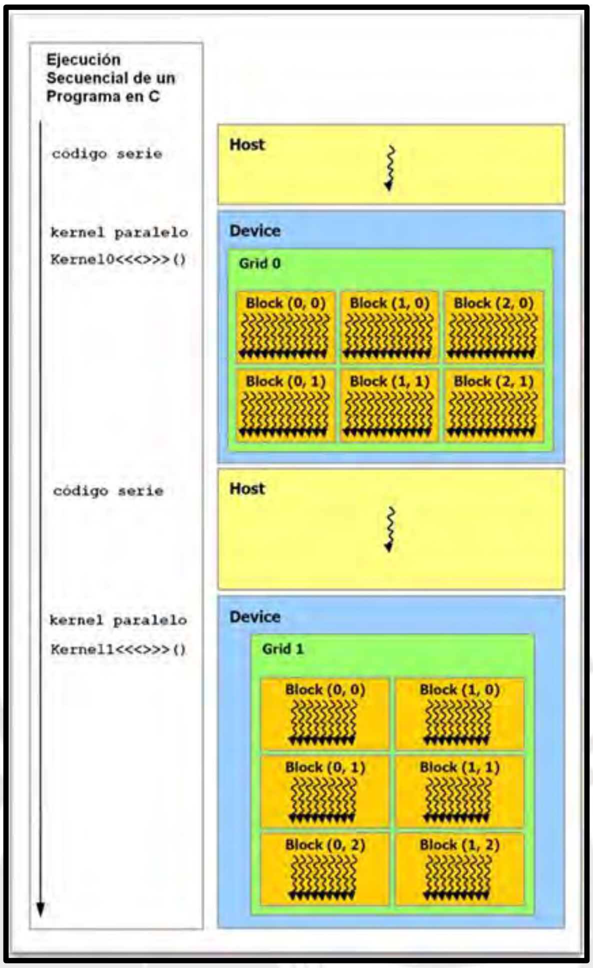


Figura 2.2.9: Estructura de ejecución CUDA [28].

Capítulo 3

Diseño e implementación de algoritmos de voxelización y planificación de rutas

Este capítulo comenzará con una explicación sobre dos secciones cruciales en la programación de un robot autónomo para minería: la etapa de voxelización y la planificación de rutas. Posteriormente, se analizará la lógica de manera serial de los códigos de programación que cubren estas dos secciones. Por último, se explorará la solución paralela de estos códigos mediante un diagrama de flujo.

3.1 Descripción de etapas de voxelización y planificación de rutas

3.1.1 Etapa de voxelización

Antes de la etapa de voxelización, el proceso se inicia con la generación de una nube de puntos del entorno por parte del sensor LiDAR, que captura la geometría del entorno circundante al robot. Luego, estos puntos pasan por un proceso de alineación mediante un algoritmo ICP, cuyo objetivo es encontrar una transformación geométrica que permite superponer dos nubes de puntos, lo que es fundamental para la detección y reconocimiento de objetos en 3D. Finalmente, tras estas etapas, se inicia el proceso de voxelización. Este consiste en la creación de pequeños cubos tridimensionales que representan el entorno del robot, lo que facilita su percepción y navegación en ambientes complejos. En esta fase, se comienza definiendo el tamaño de cada voxel, lo que se refleja en su resolución espacial. Luego, se genera una rejilla tridimensional que abarca la nube de puntos como se muestra en la Figura 3.1, seguida de un proceso de clasificación de puntos. Cada punto de la nube se asigna a un voxel de la rejilla mediante la técnica de hashing espacial. Posteriormente, a cada voxel se le calcula un valor que representa sus características, como la presencia o ausencia de objetos, la distancia a la que se encuentran o la intensidad del reflejo del láser del LiDAR. Por último, se aplican técnicas de relleno de huecos para eliminar los voxels vacíos. En la Figura 3.2 se muestra el proceso de mapeo, que incluye la etapa de voxelización descrita anteriormente.

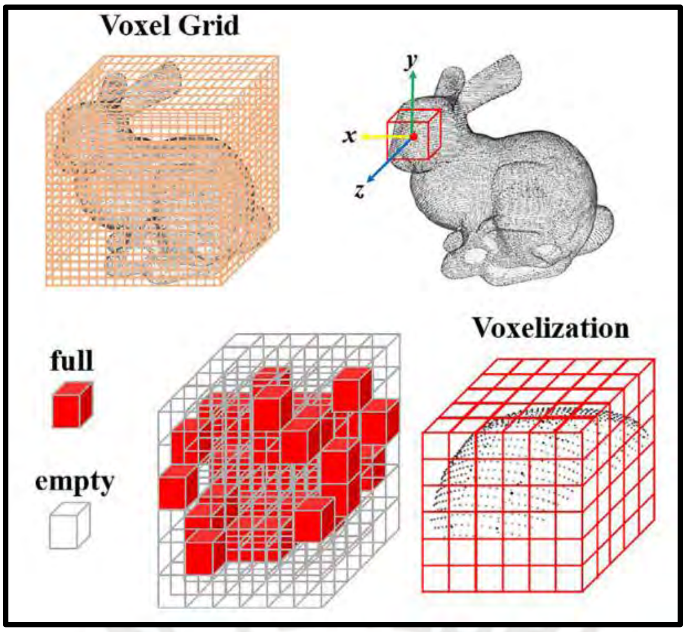


Figura 3.1: Creación de rejilla tridimensional [29].

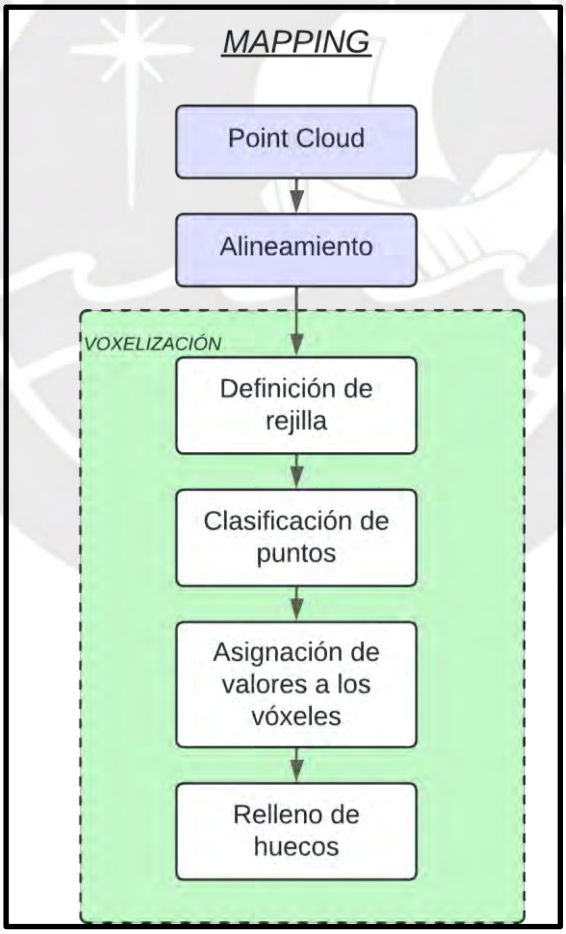


Figura 3.2: Descripción de proceso Mapping.

3.1.2 Etapa de planificación de rutas

Esta etapa comienza con los voxels creados previamente. Luego, se procede a generar un grafo local compuesto por vértices y aristas utilizando el algoritmo de muestreo aleatorio. Posteriormente, se calculan los caminos más cortos por los cuales el robot podría circular, empleando el algoritmo de Dijkstra. Finalmente, se evalúa la ganancia de exploración de cada ruta corta encontrada para seleccionar la más óptima para el robot. A medida que avanza, el robot registra la ruta seguida y los posibles caminos alternativos en caso de estar encerrado o tener poca energía. Toda esta información se almacena en el grafo global. En la Figura 3.3 se ilustra el proceso de planificación de rutas (global y local) mediante una simulación, mientras que en la Figura 3.4 se presenta la lógica de este proceso a través de un diagrama de bloques.

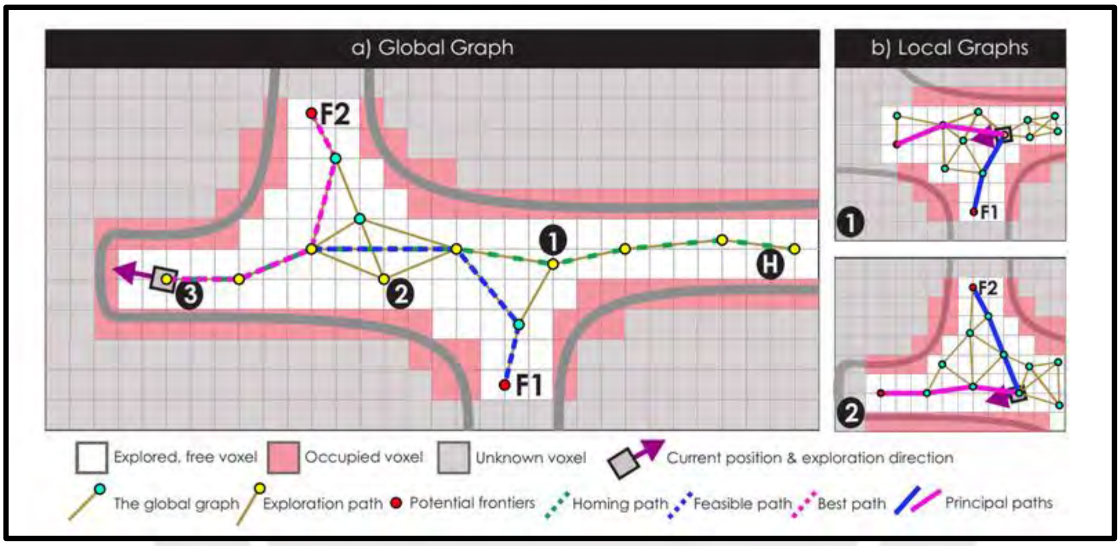


Figura 3.3: Construcción de grafo local y global [24].

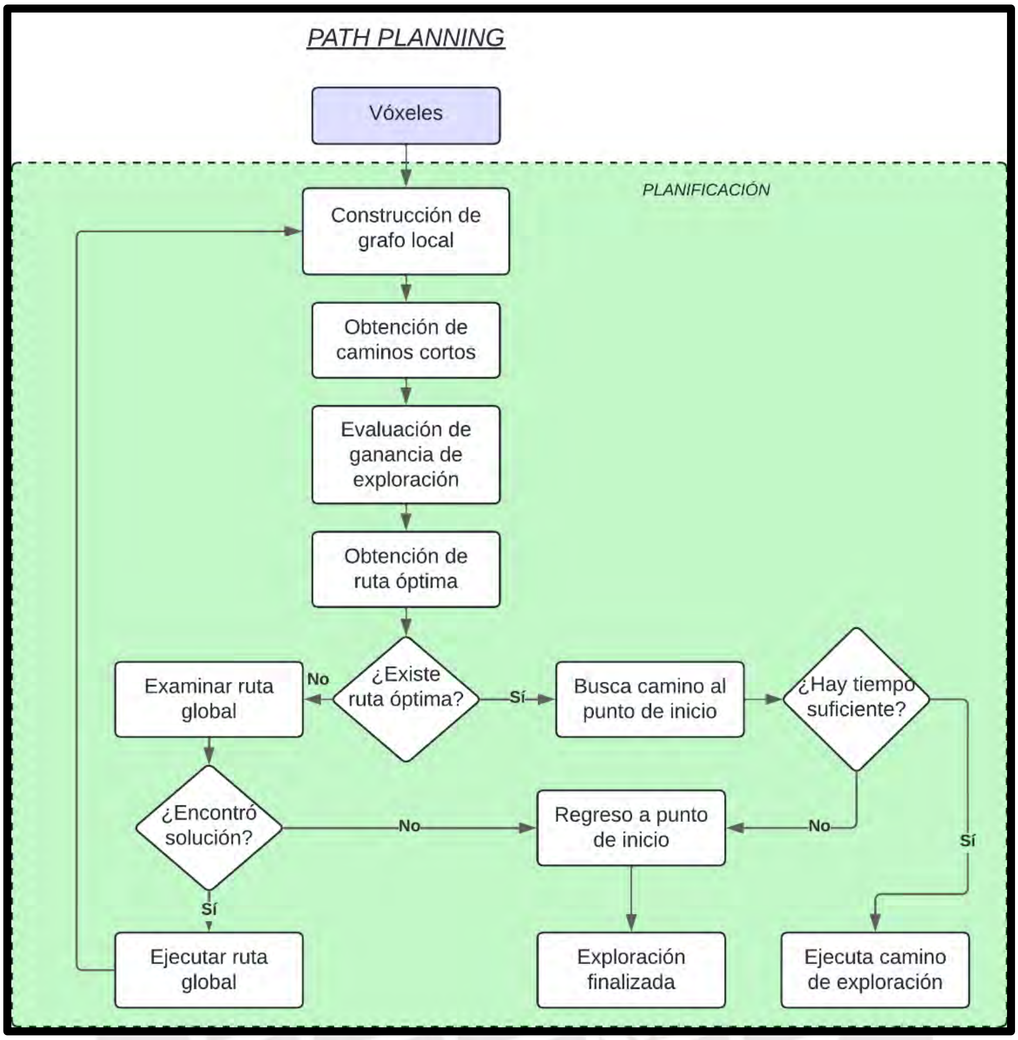


Figura 3.4: Descripción de proceso Planning.

3.2 Códigos secuenciales usando CPU

En esta sección se analizarán los códigos de estas dos etapas mediante un modo de ejecución secuencial, utilizando los métodos y funciones necesarios para imitar el proceso que seguiría un robot autónomo al ejecutarlas. El objetivo es determinar el tiempo de procesamiento requerido, lo cual será útil para posteriores pruebas comparativas.

3.2.1 Código secuencial de asignación de valores a vóxeles

El código se encarga de la serialización y deserialización de datos de vóxeles. Comienza con la definición de librerías y de la clase Block, que agrupa propiedades y métodos para trabajar con los vóxeles de un bloque.

Primero, se crea un bloque conformado por varios vóxeles del tipo Esdf (Euclidean Signed Distance Field). Se define la cantidad de vóxeles por lado del bloque, el tamaño de estos y el punto de origen, para luego calcular la cantidad total de vóxeles que conforman el bloque.

Posteriormente, se modifican los valores de los vóxeles, como la distancia, el parent y el estado observed.

Luego, se realiza la tarea de serialización del bloque, convirtiendo la estructura de datos de los vóxeles (incluyendo la dirección del voxel parent) en una secuencia de bytes para su almacenamiento o transmisión. Finalmente, se lleva a cabo el proceso de deserialización, convirtiendo el vector de bytes que contiene la información serializada de los vóxeles de vuelta a una estructura de datos comprensible. En la Figura 3.5 se muestra el diagrama de flujo del proceso de clasificación de voxels.

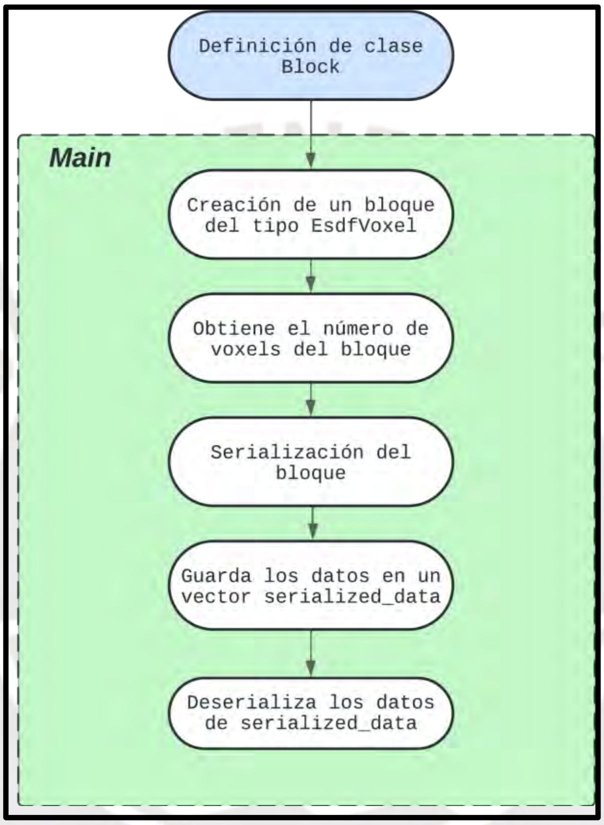
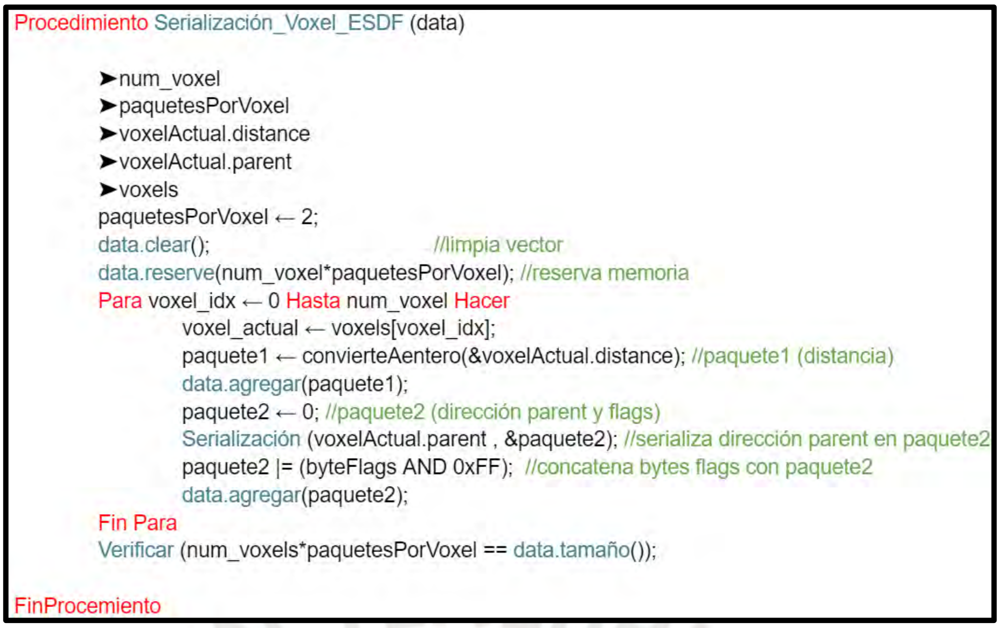


Figura 3.5: Diagrama de clasificación de voxels serial.



```
Procedimiento Serialización_Voxel_ESDF (data)

        ➤num_voxel
        ➤paquetesPorVoxel
        ➤voxelActual.distance
        ➤voxelActual.parent
        ➤voxels
        paquetesPorVoxel ← 2;
        data.clear();                    //limpia vector
        data.reserve(num_voxel*paquetesPorVoxel); //reserva memoria
        Para voxel_idx ← 0 Hasta num_voxel Hacer
                voxel_actual ← voxels[voxel_idx];
                paquete1 ← convierteAentero(&voxelActual.distance); //paquete1 (distancia)
                data.agregar(paquete1);
                paquete2 ← 0; //paquete2 (dirección parent y flags)
                Serialización (voxelActual.parent , &paquete2); //serializa dirección parent en paquete2
                paquete2 |= (byteFlags AND 0xFF);  //concatena bytes flags con paquete2
                data.agregar(paquete2);
        Fin Para
        Verificar (num_voxels*paquetesPorVoxel == data.tamaño());

FinProcemiento
```

Figura 3.6: Algoritmo de Serialización.

3.2.2 Código secuencial de planificación de rutas

Este código sigue un proceso secuencial para describir el proceso de elaboración de un grafo hasta la selección de caminos cortos. Inicia con la definición de la clase Graph, que incluye una variedad de métodos para agregar y eliminar vértices y aristas. Además, incorpora funciones para encontrar los caminos más cortos, utilizando tanto el algoritmo de Dijkstra como el algoritmo de Bellman-Ford, ambos diseñados para este propósito. Luego, en la función principal (main), se inicia con la creación del objeto del grafo dirigido, donde se almacenarán tanto los vértices como las aristas. Luego, se define un vértice fuente que actuará como punto de partida para los caminos cortos. A continuación, se determina aleatoriamente una cantidad de vértices. Una vez hecho esto, se procede a crear dicha cantidad de vértices y se almacenan en el objeto correspondiente.

Posteriormente, se generan las aristas y se conectan aleatoriamente con los vértices, asignándoles un peso. Luego, se inicia el cálculo de los caminos más cortos desde el vértice fuente hacia los demás vértices, teniendo en cuenta los pesos de las aristas. En la Figura 3.7 se muestra un diagrama de flujo del código serial.

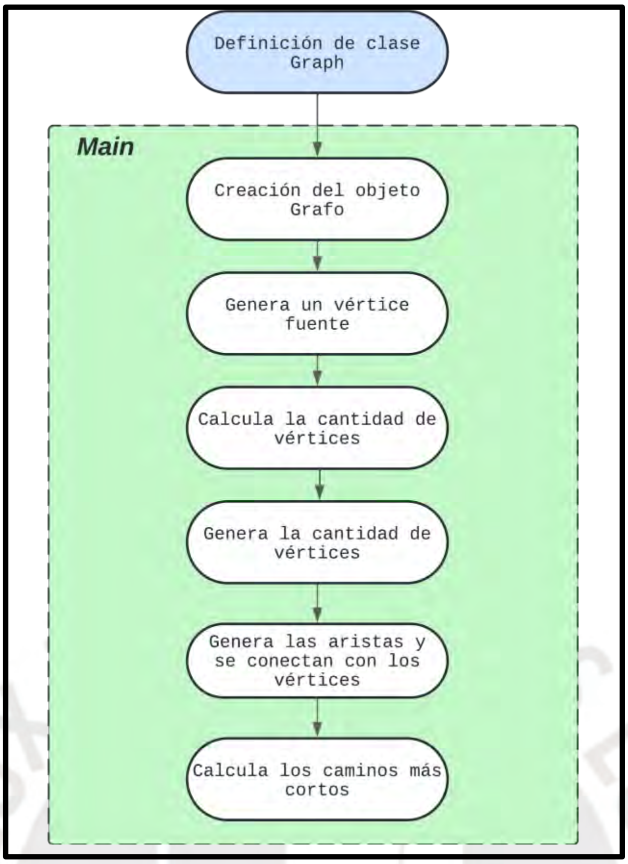
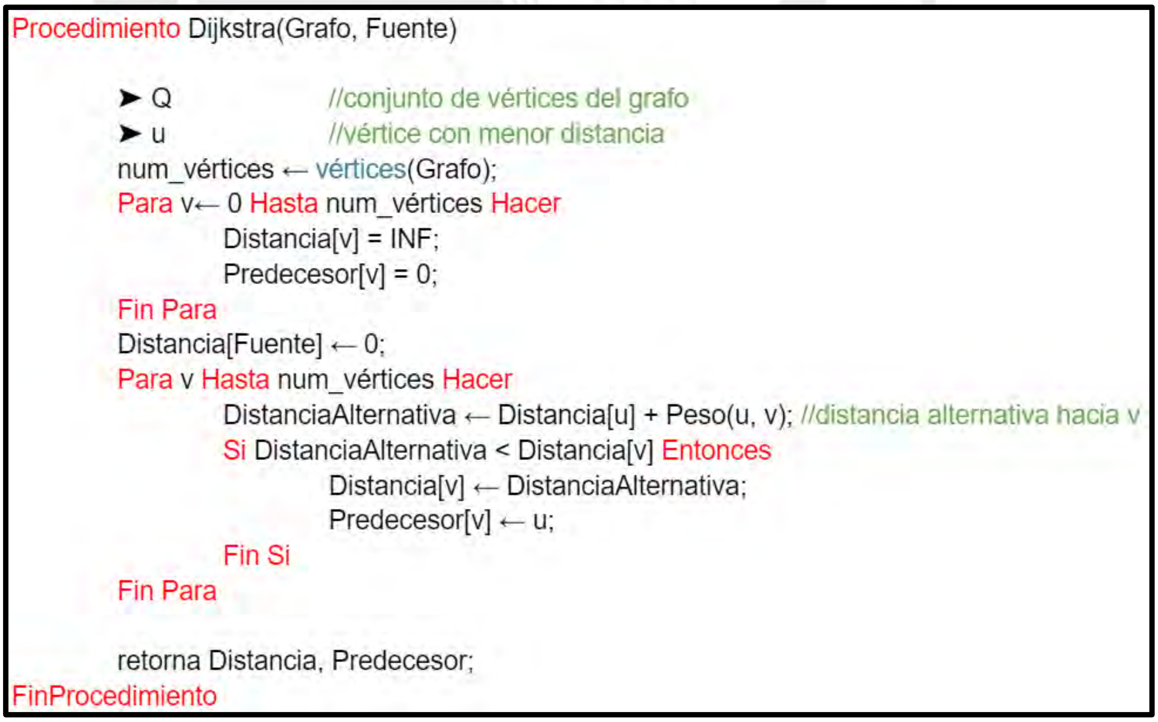


Figura 3.7: Diagrama de flujo path planning serial.

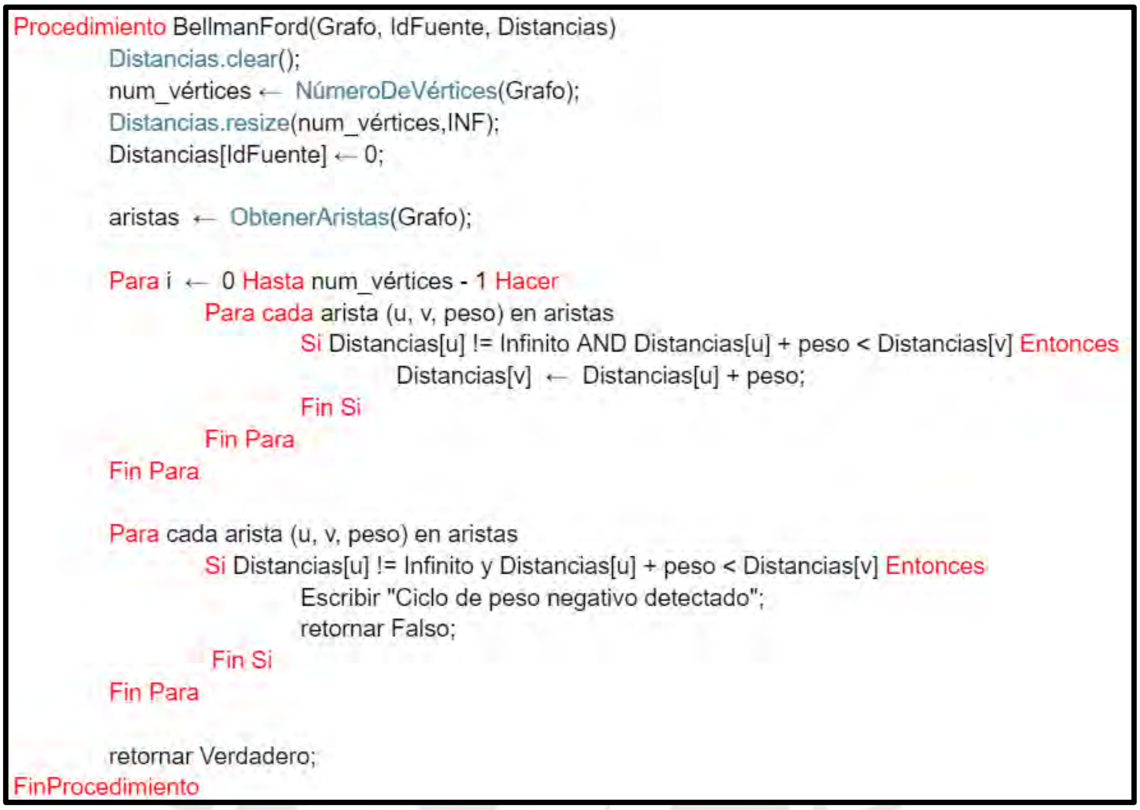


```
Procedimiento Dijkstra(Grafo, Fuente)

        ➤ Q                //conjunto de vértices del grafo
        ➤ u                //vértice con menor distancia
        num_vértices ← vértices(Grafo);
        Para v← 0 Hasta num_vértices Hacer
                Distancia[v] = INF;
                Predecesor[v] = 0;
        Fin Para
        Distancia[Fuente] ← 0;
        Para v Hasta num_vértices Hacer
                DistanciaAlternativa ← Distancia[u] + Peso(u, v); //distancia alternativa hacia v
                Si DistanciaAlternativa < Distancia[v] Entonces
                        Distancia[v] ← DistanciaAlternativa;
                        Predecesor[v] ← u;
                Fin Si
        Fin Para

        retorna Distancia, Predecesor;
FinProcedimiento
```

Figura 3.8: Algoritmo Dijkstra.



```
Procedimiento BellmanFord(Grafo, IdFuente, Distancias)
    Distancias.clear();
    num_vértices ← NúmeroDeVértices(Grafo);
    Distancias.resize(num_vértices,INF);
    Distancias[IdFuente] ← 0;

    aristas ← ObtenerAristas(Grafo);

    Para i ← 0 Hasta num_vértices - 1 Hacer
        Para cada arista (u, v, peso) en aristas
            Si Distancias[u] != Infinito AND Distancias[u] + peso < Distancias[v] Entonces
                Distancias[v] ← Distancias[u] + peso;
            Fin Si
        Fin Para
    Fin Para

    Para cada arista (u, v, peso) en aristas
        Si Distancias[u] != Infinito y Distancias[u] + peso < Distancias[v] Entonces
            Escribir "Ciclo de peso negativo detectado";
            retornar Falso;
        Fin Si
    Fin Para

    retornar Verdadero;
FinProcedimiento
```

Figura 3.9: Algoritmo Bellman-Ford.

3.3 Códigos paralelizados usando GPU

En esta sección se explicará cómo se paralelizarán los códigos secuenciales del proceso de serialización de datos y del algoritmo de Bellman-Ford para encontrar las distancias más cortas en un grafo. Para la paralelización de estos dos algoritmos se utilizará la interfaz CUDA, que permitirá aprovechar la GPU de la computadora para realizar las tareas de manera paralela.

3.3.1 Código paralelizado de asignación de valores a vóxeles

Como se muestra en la Figura 3.10, el algoritmo de serialización de datos recibe una estructura de datos que consiste en un bloque tridimensional conformado por N vóxeles. Cada voxel contiene información que debe ser serializada individualmente para su transmisión de un nodo a otro. Para ello, se utilizan los recursos de la GPU, donde se crea un kernel encargado de serializar un voxel específico, y todos los vóxeles se procesan de manera paralela. Finalmente, la información serializada de cada voxel se almacena en una sola variable para su transmisión.

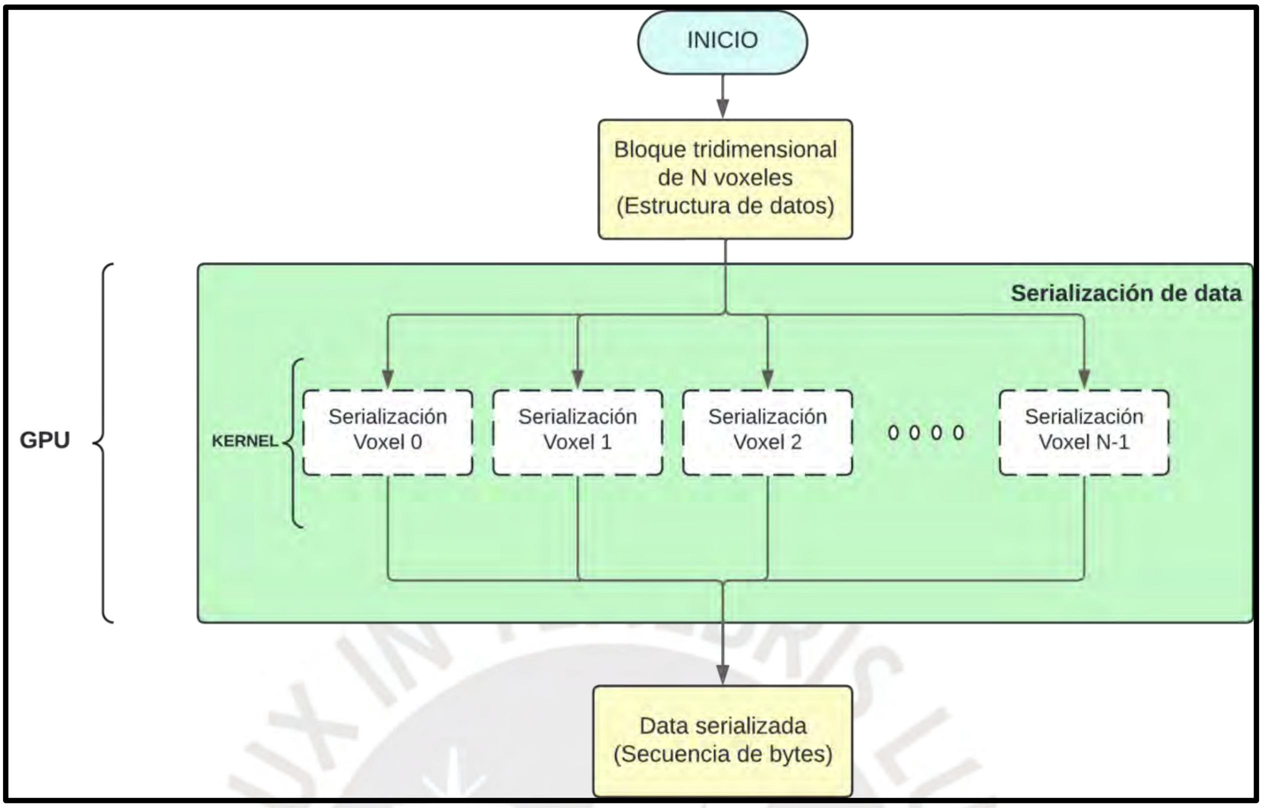


Figura 3.10: Serialización de datos en GPU.

3.3.2 Código paralelizado de planificación de rutas

En la Figura 3.11 se muestra cómo se paralelizará el algoritmo de Bellman-Ford utilizando los recursos de la GPU. Primero, el algoritmo recibe como entrada un grafo ponderado con N vértices y E aristas, cada una con distintos pesos. Este algoritmo debe iterarse N-1 veces, donde en cada iteración se realiza un proceso de relajación de aristas. Este proceso consiste en encontrar una distancia más corta desde el vértice fuente hasta un vértice específico que la distancia originalmente conocida. Si se encuentra una distancia menor, esta se actualiza. La relajación se aplica a todas las aristas del grafo, por lo que se decide asignar esta tarea a un kernel para que se ejecute de manera paralela las N-1 veces que se deba iterar este algoritmo.

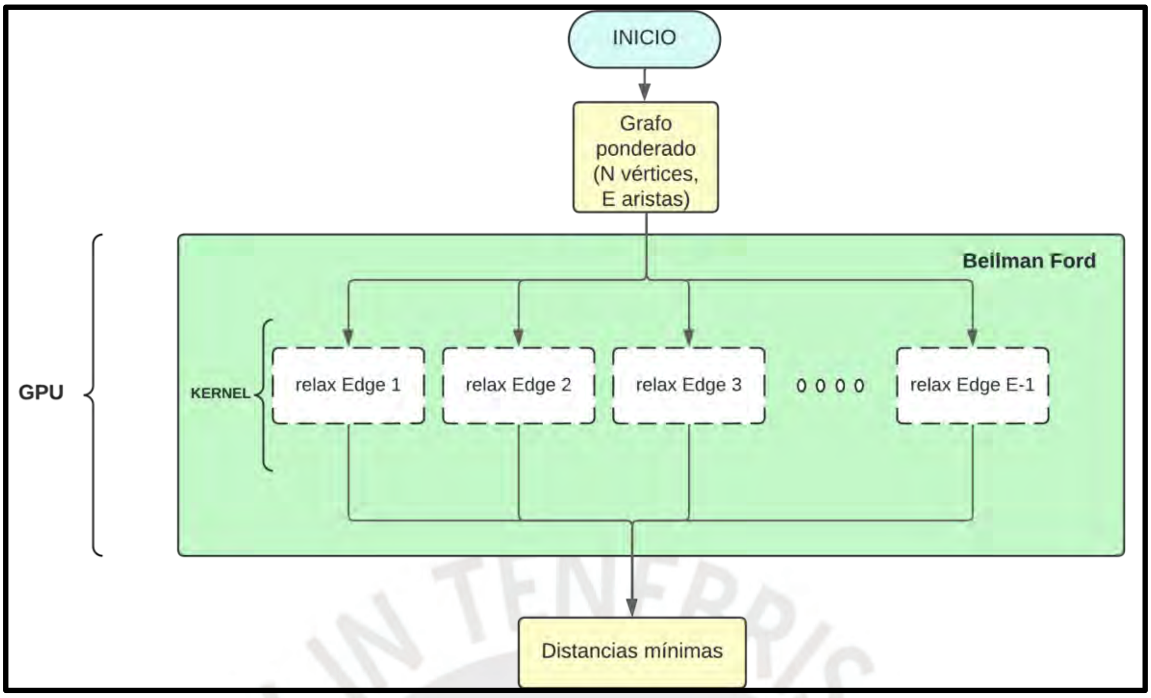


Figura 3.11: Bellman-Ford en GPU.

Capítulo 4

Resultados

En este capítulo se presentan los resultados obtenidos a partir de las simulaciones de los algoritmos de Dijkstra y Bellman-Ford, tanto en su implementación serial como en su versión paralela. Además, se incluye una sección dedicada al código de serialización de datos, tanto en su forma serial como paralelizada.

Primero, se detallará el entorno de desarrollo utilizado para paralelizar los códigos, así como las especificaciones del equipo en el que se realizaron las simulaciones. Finalmente, se procederá a analizar los resultados, destacando las optimizaciones y mejoras de rendimiento obtenidas tras la paralelización de los algoritmos.

4.1 Recursos usados para la simulación

4.1.1 Entorno de desarrollo

Los códigos fueron desarrollados en C++. Para la simulación de los códigos seriales originales, se utilizó Visual Studio Code (VS Code), en el cual se instalaron extensiones de C++ para poder compilar. La versión de VS Code utilizada fue la 1.90.1.

Para la codificación en modo paralelizado, se instaló CUDA de Nvidia, una plataforma que permite paralelizar códigos en C++ utilizando los recursos de la GPU. La versión de CUDA utilizada fue la 12.4. Adicionalmente, se empleó Visual Studio 2022 para codificar, aprovechando la integración con CUDA para utilizar los recursos de la GPU.

En la codificación paralela se utilizaron librerías como cuda_runtime y device_launch_parameters, las cuales permiten asignar memoria en la GPU de acuerdo a las variables que se van a utilizar. Estas librerías también incluyen funciones para transferir datos entre la CPU y la GPU y viceversa. Además, permiten la creación y configuración de kernels que se ejecutan de manera paralela.

4.1.2 Especificaciones técnicas

Las simulaciones se llevaron a cabo en un CPU equipado con un procesador Intel Core i7-9750HF de 2.60GHz, que dispone de 6 núcleos. El sistema cuenta con 16 GB de RAM y utiliza una tarjeta gráfica Nvidia GeForce GTX 1050. Esta GPU posee 3GB de memoria y cuenta con 768 núcleos CUDA a una frecuencia de 1392 MHz. Además, tiene un ancho de interfaz de memoria de 96 bits y una memoria de banda ancha de 84 GB/s.

4.1.3 Datos de entrada

Para los algoritmos de búsqueda de caminos de distancia más corta, específicamente Dijkstra y Bellman-Ford, se utilizaron grafos ponderados con pesos positivos. El algoritmo de Dijkstra está diseñado para trabajar exclusivamente con pesos positivos, lo que garantiza que siempre encontrará el camino más corto en un grafo ponderado positivamente. Por otro lado, Bellman-Ford es más flexible y puede manejar tanto pesos positivos como negativos, lo que lo hace adecuado para casos donde pueden existir ciclos de peso negativo.

En las simulaciones, se varió el número de vértices en los grafos para obtener diferentes muestras y analizar cómo el tiempo de ejecución de estos algoritmos se ve afectado por el tamaño del grafo. Este enfoque permite evaluar la escalabilidad y eficiencia de los algoritmos bajo diferentes condiciones, proporcionando una visión más completa de su desempeño. En la Figura 4.1 se muestra un grafo de 10 vértices ponderado con pesos positivos.

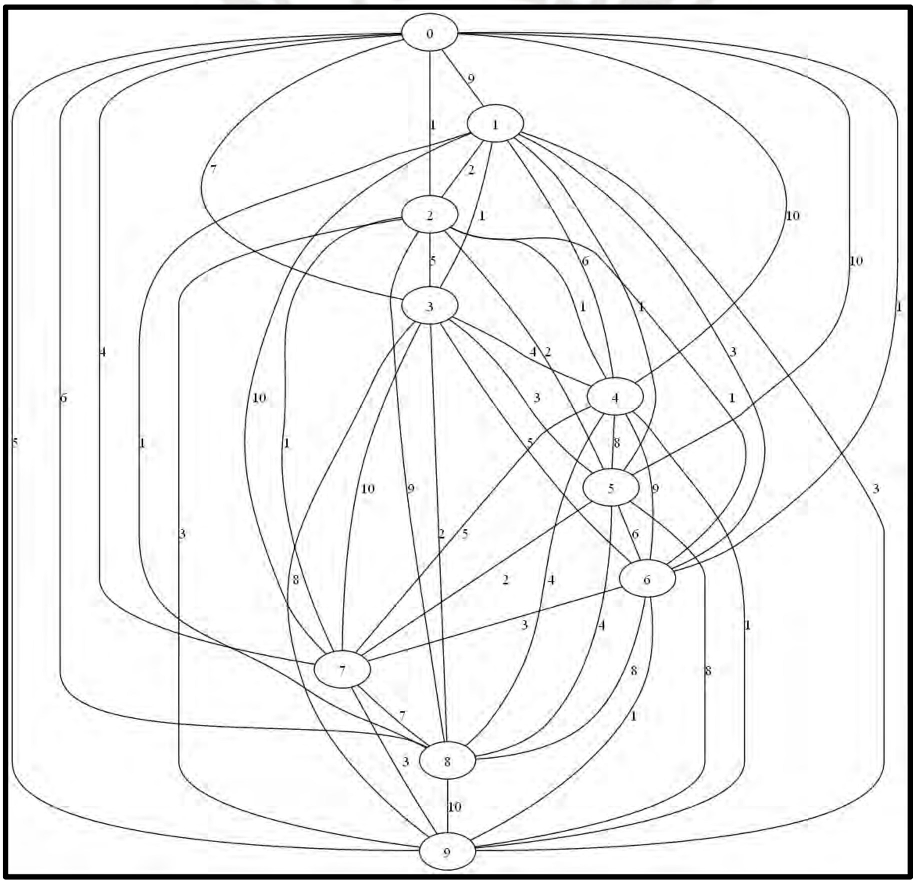


Figura 4.1: Grafo ponderado de 10 vértices.

Por otro lado, para el algoritmo de serialización de datos, se consideró como entrada un bloque tridimensional compuesto por vóxeles, que son pequeños cubos del tipo ESDF (Euclidean Signed Distance Fields) de tamaño unitario. Estos vóxeles contienen información como la distancia a la que se encuentran, si han sido detectados por el sensor y cuáles son sus vóxeles parent. Además, este bloque posee un origen tridimensional.

4.2 Resultados de algoritmos simulados

4.2.1 Tiempo de ejecución Serial

Para la ejecución del algoritmo de Dijkstra de manera secuencial, se realizó una simulación sobre un rango de 0 a 500 vértices. Además, se consideraron pesos positivos dentro del rango de 0 a 10. La simulación se llevó a cabo en el software Visual Studio Code, utilizando el lenguaje C++. Este es el código actualmente implementado en el robot Cerberus y con el cual opera.

En la Figura 4.2, se observa un comportamiento cuadrático en el tiempo de ejecución, alcanzando 81.66 ms para 500 vértices. La gráfica tiene un coeficiente de determinación de 0.989, lo que indica una fuerte correlación entre el tamaño del grafo y el tiempo de ejecución del algoritmo.

El comportamiento cuadrático del algoritmo de Dijkstra en este caso se debe a que no se está usando una cola de prioridad mínima, ya que si la tuviera tendría un comportamiento logarítmico.

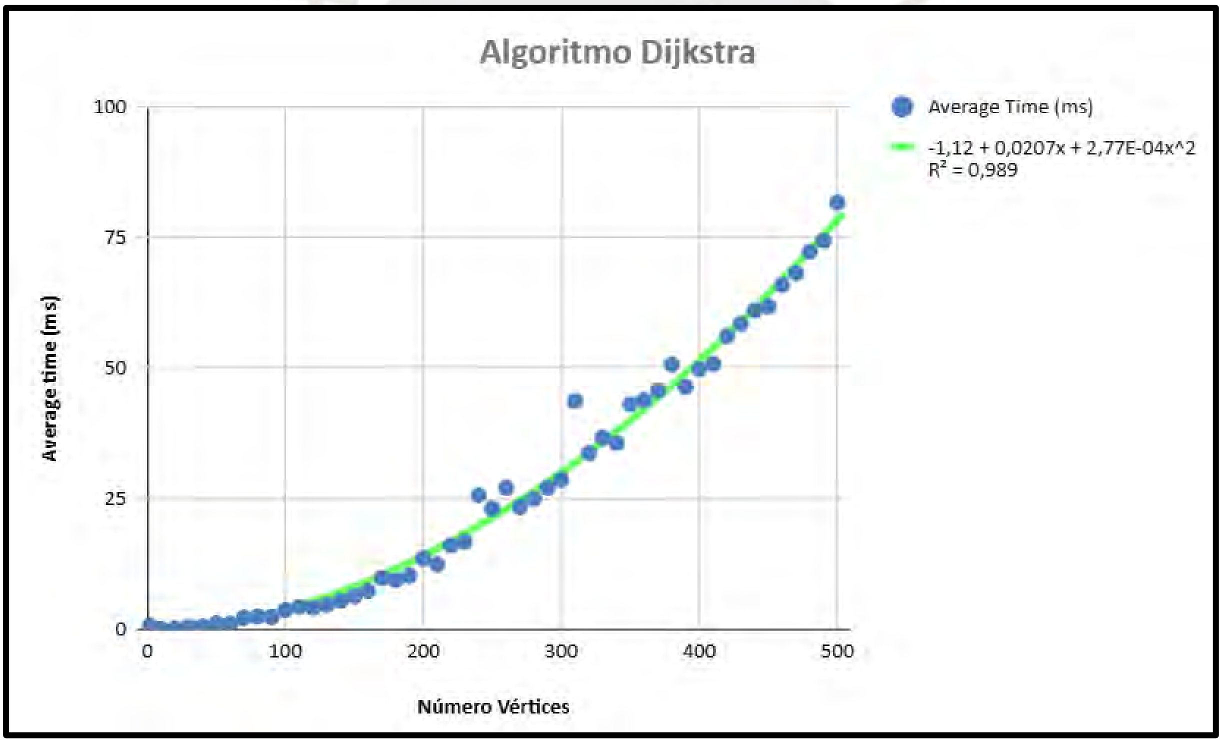


Figura 4.2: Algoritmo Dijkstra modo serial.

Por otro lado, en la ejecución del algoritmo Bellman-Ford de manera secuencial, se realizó también una simulación sobre un rango de 0 a 500 vértices. De igual manera, se consideraron pesos positivos dentro del rango de 0 a 10. Para la simulación se usó código en C++. Este algoritmo permite pesos negativos y se optó por él como una alternativa para la detección de rutas cortas debido a su naturaleza no secuencial, lo que facilita su paralelización.

A diferencia del algoritmo de Dijkstra, que posee un comportamiento secuencial y no permite paralelizar el código de la manera deseada. Sin embargo, como se aprecia en la figura, el algoritmo Bellman-Ford presenta mayores tiempos de ejecución que luego con la paralelización mejorarán.

Como se muestra en la Figura 4.3, el algoritmo Bellman-Ford alcanza tiempos de ejecución de hasta 125000 ms, superando a otros métodos. Esto se evidencia por un coeficiente de determinación de 0.998.

La Figura 4.3 muestra un comportamiento cúbico debido a que el algoritmo ejecutado considera un grafo en donde la cantidad de aristas es un valor mucho mayor que la cantidad de vértices. Además, el grafo está representado mediante una lista de adyacencia, lo que implica que el tiempo de procesamiento aumenta rápidamente a medida que crece el tamaño del grafo, ya que cada nodo puede estar conectado a un número variable de nodos vecinos.

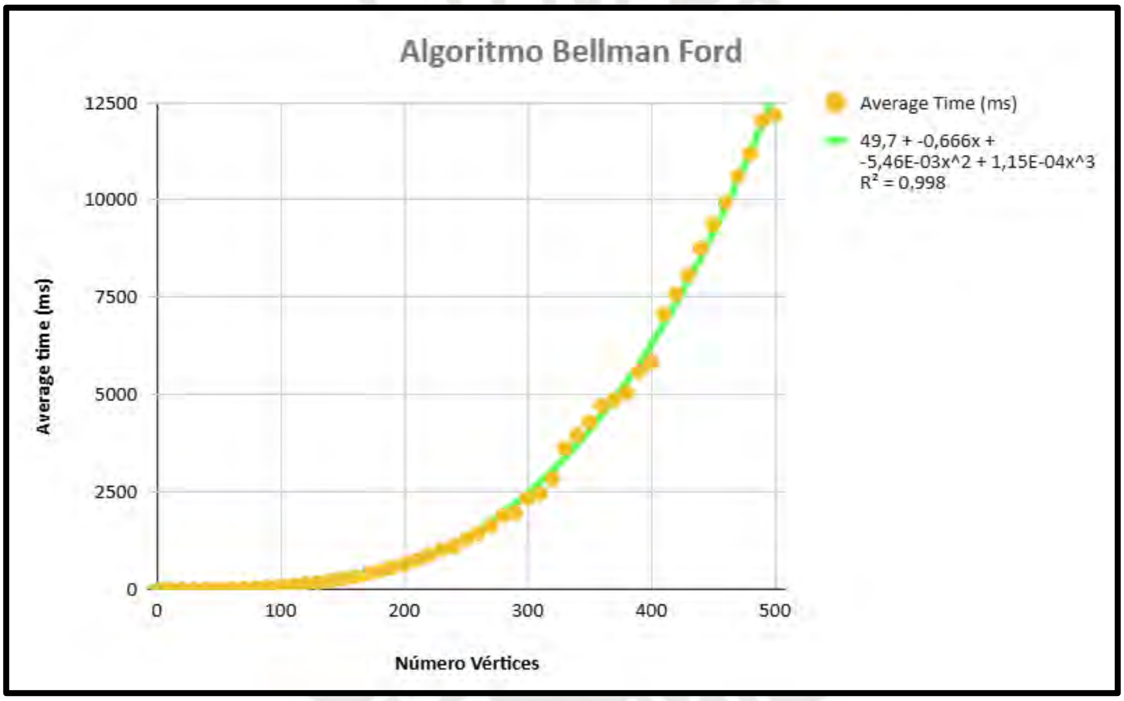


Figura 4.3: Algoritmo Bellman-Ford modo serial.

Finalmente, se muestra el modo serial de la etapa de serialización de datos, implementado en C++. Como se muestra en la Figura 4.4, ante un bloque conformado por 500 vóxeles por lado de una unidad de tamaño, el tiempo de ejecución puede llegar a los 20000 ms, lo cual es un tiempo muy significativo considerando que este proceso debe realizarse de manera rápida para transferir los datos de los vóxeles eficientemente. También se observa que presenta un coeficiente de determinación de 0.98 y un comportamiento exponencial.

Esta simulación también se llevó a cabo para un rango de vóxeles por lado, proporcionando mayor información acerca de la relación con el tiempo de ejecución. Las dimensiones de los vóxeles que conforman el bloque general (cuadrilla tridimensional) se mantienen constantes para representar una resolución fija. Disminuir las dimensiones implicaría una mayor resolución y, por tanto, un mayor esfuerzo computacional.

Figura 4.4: Algoritmo Serialización modo serial.

4.2.2 Tiempo de ejecución Paralelizada

Debido a la naturaleza no secuencial del algoritmo Bellman-Ford, se ha decidido paralelizar. En contraste, el algoritmo de Dijkstra es secuencial, lo que impide su paralelización.

Para la simulación del algoritmo se utilizaron parámetros específicos como el tamaño y el número de bloques que se ejecutarán en la GPU. Después de múltiples simulaciones del algoritmo, se determinó que un tamaño de bloque de 128 es óptimo, ya que minimiza el consumo de recursos computacionales y reduce el tiempo de ejecución. Este valor no se puede calcular de manera exacta mediante un enfoque matemático, por lo que su selección se basa en pruebas experimentales del código. La Figura 4.5 ilustra la relación entre el tiempo de ejecución y el tamaño del bloque. Posteriormente, el número de bloques se determinó en función del tamaño del bloque y la cantidad de aristas que conforman el grafo. El kernel ejecutado en la GPU se encargó de la relajación de aristas, una operación crucial para encontrar los caminos más cortos en el grafo.

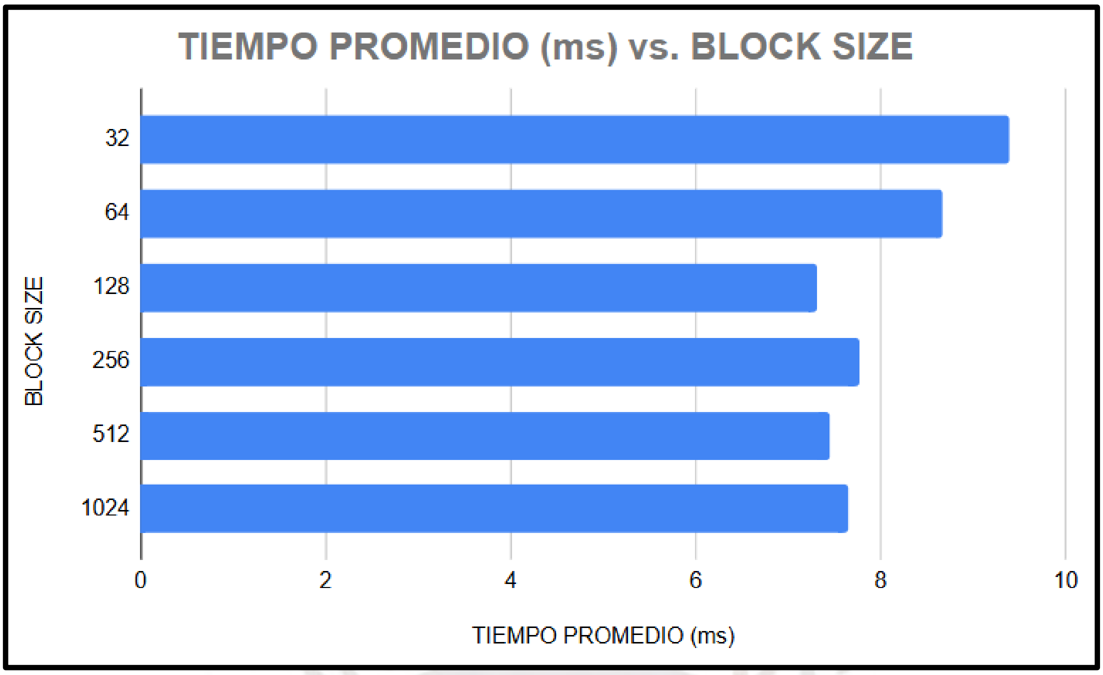


Figura 4.5: Relación del tamaño del bloque con el tiempo de ejecución.

En la Figura 4.6 se puede apreciar un comportamiento cuadrático a medida que aumenta el número de vértices. Para un grafo con 500 vértices, el tiempo llega aproximadamente a 5.7 ms, lo cual es un valor muy optimizado en comparación con la versión secuencial de Bellman-Ford. Además, se muestra un coeficiente de determinación de 0.979 considerando un tamaño de bloque de 128 para la simulación.

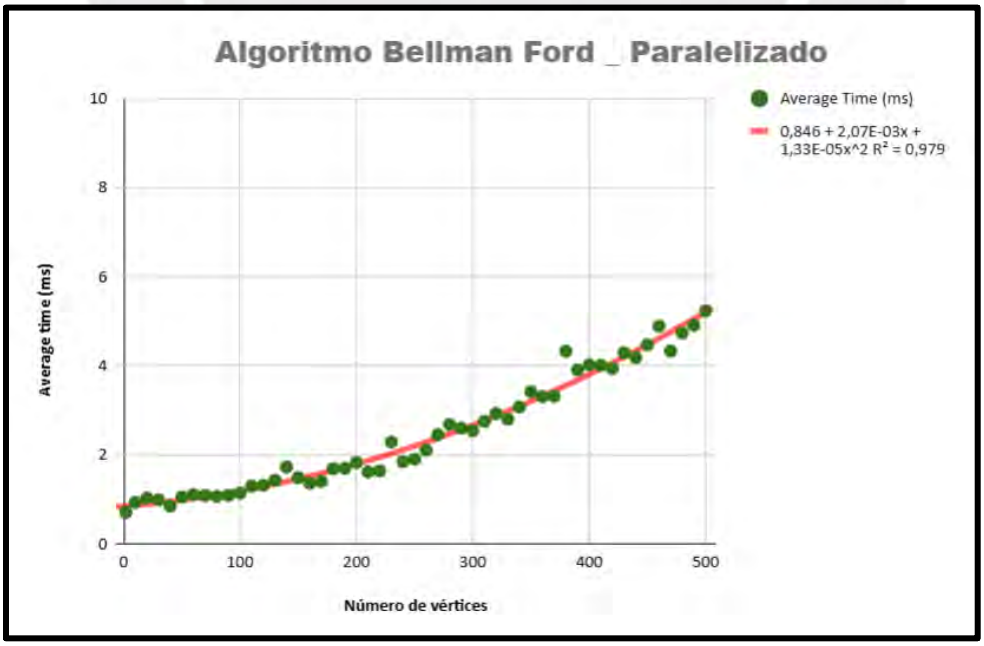


Figura 4.6: Algoritmo Bellman-Ford modo paralelizado.

Por otro lado, para la simulación del proceso de serialización de datos, también se consideró un tamaño de bloque de 128, es decir, que habría 128 hilos ejecutándose de manera paralela. El número de bloques también depende del tamaño del bloque y del número de vóxeles que conforman el bloque tridimensional.

En la Figura 4.7 se puede observar un comportamiento exponencial y que los valores de tiempo en milisegundos se habrían reducido para cada número de vóxeles por lado que conformaría el bloque. La versión paralelizada muestra que para valores de 500 vóxeles por lado, este tendría un tiempo de ejecución de casi 6000 ms, un valor optimizado en comparación con su versión serial. Además, este gráfico presenta un coeficiente de determinación de 0.907, demostrando la relación que existe entre el tiempo de ejecución y el número de vóxeles por lado.

Figura 4.7: Algoritmo Serialización modo paralelizado.

4.2.3 Speed Up

Para optimizar el algoritmo de Bellman-Ford, se comparará su rendimiento con su versión secuencial y con el algoritmo de Dijkstra. En esta comparación, se analizarán las ecuaciones exponenciales de la línea de tendencia para evaluar su desempeño.

Primero, se puede analizar que en la Ecuación 4.1, la versión paralelizada es significativamente mejor, ya que su tiempo de ejecución es más de 100 veces más rápido que el de la versión secuencial. Esto se debe a que los procesos de relajación de aristas se realizan de manera paralela, aprovechando los recursos de la GPU, lo que optimiza considerablemente el tiempo de ejecución.

La Figura 4.8 muestra que el "speed up" incrementa a medida que aumenta la variable x, que representa la cantidad de vértices del grafo generado.

$$S(x) = \frac{0.000115x^3 - 0.00546x^2 - 0.666x + 49.7}{0.0000133x^2 + 0.00207x + 0.846} \tag{4.1}$$

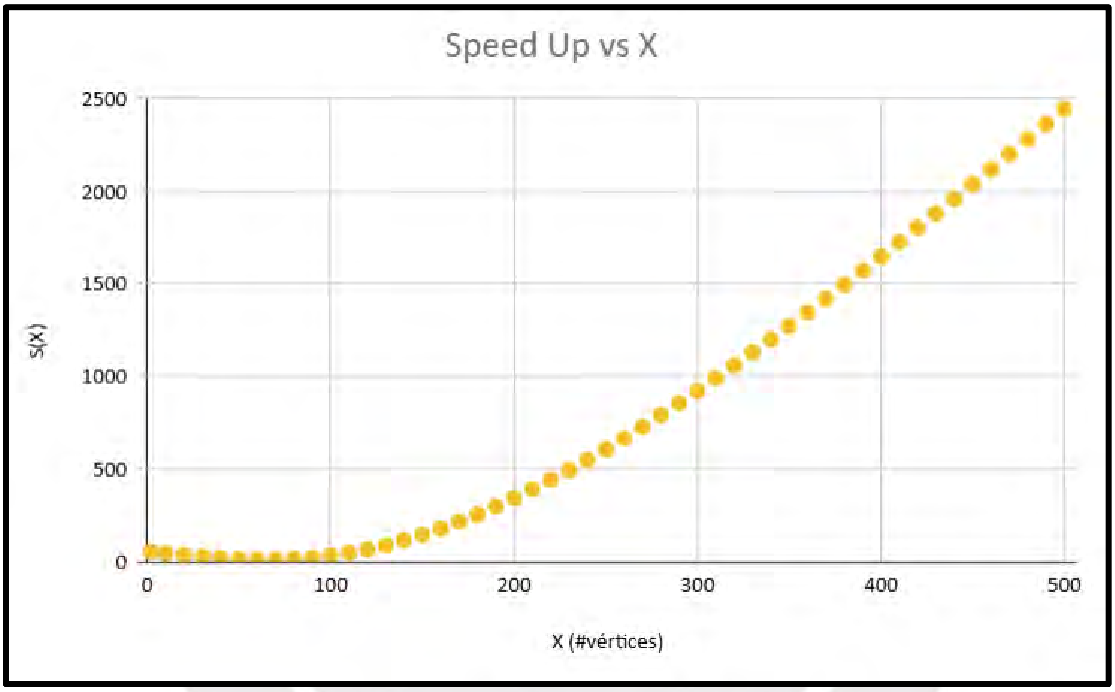


Figura 4.8: Speed Up de versión serial y paralela de Bellman-Ford.

Por otro lado, en la Ecuación 4.2 se presenta la comparación con el algoritmo de Dijkstra, que también se utiliza para encontrar las distancias mínimas en un grafo. Se observa que la versión paralelizada del algoritmo de Bellman-Ford es superior, ya que su tiempo de ejecución es una cuarta parte del que requeriría el algoritmo de Dijkstra para hallar las distancias más cortas.

En la Figura 4.9 se observa que el valor de 'speed up' aumenta a medida que incrementa la variable x que representa el número de vértices del grafo. Aunque este incremento es menor en comparación con el aumento entre la versión serial y paralelizada de Bellman, sigue siendo un algoritmo optimizado.

$$S(x) = \frac{0.000277x^2 + 0.0207x - 1.12}{0.0000133x^2 + 0.00207x + 0.846} \tag{4.2}$$

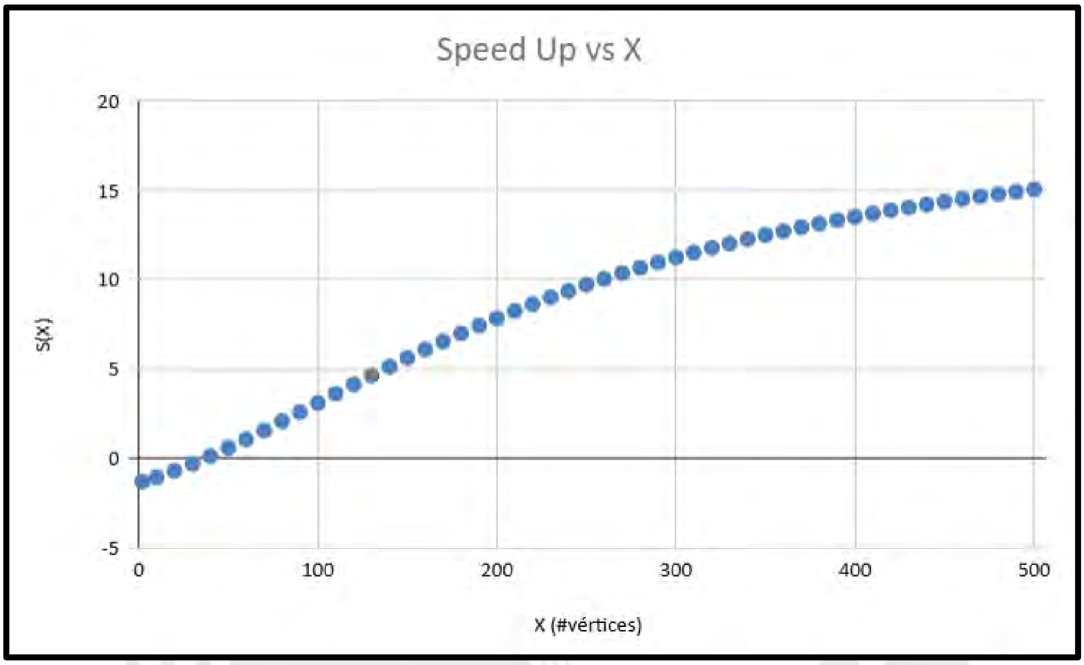


Figura 4.9: Speed Up de Dijkstra con Bellman-Ford.

Por otra parte, en el algoritmo de serialización de datos también se logró una optimización significativa en los tiempos de ejecución. Como se muestra en la Ecuación 4.3, la versión paralela de este algoritmo supera considerablemente a la versión secuencial a medida que aumenta el número de vóxeles en un bloque tridimensional.

En la Figura 4.10 se muestra el comportamiento del 'speed up' en relación con el número de vóxeles por lado del bloque tridimensional. Este valor disminuye a medida que aumenta el número de bloques por lado, tendiendo a 0, aunque nunca alcanzará ese valor.

$$S(x) = \frac{289e^{0,00847x}}{10,5e^{0,0125x}} \tag{4.3}$$

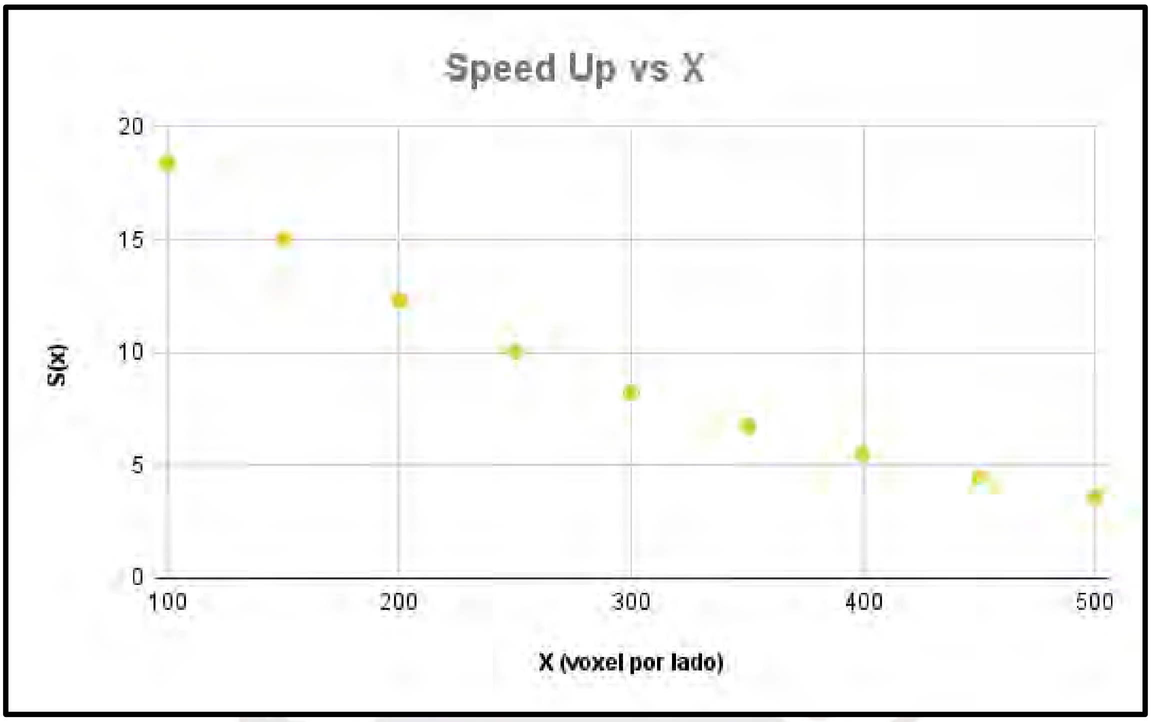


Figura 4.10: Speed Up de serialización de datos.

4.3 Optimizaciones en códigos

4.3.1 Bellman-Ford

En el código serial de este algoritmo, se itera la cantidad de vértices del grafo menos uno para encontrar las distancias más cortas desde el vértice fuente hacia los demás vértices. Debido a esto, esta versión no es óptima en comparación con el algoritmo de Dijkstra. Para mejorar esto, en la versión paralela se decidió realizar la relajación de aristas en un kernel. Este kernel se ejecuta de manera paralela en la GPU, lo que resulta en una reducción significativa del tiempo de ejecución.

Anteriormente se configuró el tamaño del bloque en 256, que representa la cantidad de hilos que se ejecutarán de manera paralela. Además, la cantidad de bloques se determina en función del tamaño del bloque y de la cantidad de aristas del grafo, ya que la relajación de aristas depende de este último parámetro.

Estas optimizaciones han conducido a una mejora notable en el tiempo de ejecución, alcanzando una cuarta parte del tiempo que tomaría el algoritmo de Dijkstra, utilizado por el robot para la planificación de rutas.

4.3.2 Serialización de datos

Este algoritmo recibe un bloque tridimensional y los parámetros de cada voxel. Luego, analiza los datos de cada voxel, recogiendo información sobre la distancia, el voxel padre, entre otros, y serializa estos datos. La información recibida se estructura de manera que, al ser serializada, se convierte en una secuencia de bytes.

En el algoritmo de serialización, se itera sobre cada voxel. Para paralelizar esta tarea, se selecciona esta iteración y se coloca en un kernel. Antes de esto, se considera el tamaño del bloque, que es de 128, y la cantidad de bloques, que depende de la cantidad de vóxeles que contiene el bloque. Ahora, cada iteración sobre un voxel se realiza de manera paralela utilizando los 128 hilos generados por cada bloque y los hilos de los bloques restantes. Con esta solución, se logró optimizar el tiempo de ejecución del algoritmo, reduciéndolo 20 veces.

Conclusiones

- La versión paralela del algoritmo de Bellman-Ford supera con creces a la versión serial, gracias al uso de los recursos de la GPU. Los hilos de la GPU permiten configurar funciones para que se ejecuten en paralelo como kernels, aprovechando al máximo la memoria y los núcleos de la GPU.

- La versión optimizada de Bellman-Ford, aunque superior a la versión serial, tiene un menor speed up en comparación con Dijkstra, ya que Bellman-Ford requiere V-1 iteraciones para encontrar las distancias mínimas. Por ello, Dijkstra es mejor en tiempos de ejecución en su versión serial. Esto se debe a que el algoritmo de Bellman-Ford tiene una complejidad lineal, mientras que Dijkstra tiene una complejidad de logarítmica.

- Para la serialización de datos, es esencial que este proceso sea óptimo para permitir una rápida compartición de la información contenida dentro de un voxel. La paralelización del proceso de serialización de datos es eficaz hasta cierto número de vóxeles por lado de un bloque tridimensional, ya que más allá de ese punto, el speed up comienza a decrecer y deja de ser óptimo. Aunque el speed up presenta un comportamiento exponencial negativo, sigue siendo un algoritmo paralelizado y optimizado hasta un valor de 500 vóxeles por lado.

- Se ha logrado acelerar significativamente el algoritmo de planificación de rutas del robot Cerberus mediante la paralelización utilizando la plataforma CUDA y aprovechando la capacidad de procesamiento de una GPU. Los resultados de las simulaciones muestran una reducción en los tiempos de ejecución en comparación con la versión serial original. Las gráficas de speed up evidencian esta mejora, lo que se traduce en una mayor eficiencia y capacidad de respuesta del robot en tareas de navegación.

Bibliografía

[1] L. Sanmiquel, J. M. Rossell y C. Vintró, "Study of Spanish mining accidents using data mining techniques," Safety Science, vol. 75, pp. 49-55, 2015, doi: 10.1016/j.ssci.2015.01.016.

[2] S. N. Ismail, A. Ramli y H. A. Abdul, "Research trends in mining accidents study: A systematic literature review," Safety Science, vol. 143, p. 105438, 2021, doi: 10.1016/j.ssci.2021.105438.

[3] X. Wang y F. B. Meng, "Statistical analysis of large accidents in China's coal mines in 2016," Nat Hazards, vol. 92, pp. 311–325, 2018, doi: 10.1007/s11069-018-3211-5.

[4] MINEM, "Coyuntural de Accidentes Mortales - Año 2023," minem.gob.pe, [Gráfica]. Disponible en: https://www.minem.gob.pe/_estadistica.php?idSector=1&idEstadistica=12464. [Accedido: 21 de septiembre de 2023].

[5] W. Qiao, "Analysis and measurement of multifactor risk in underground coal mine accidents based on coupling theory," Reliability Engineering & System Safety, vol. 208, p. 107433, 2021, doi: 10.1016/j.ress.2021.107433.

[6] C. Lu, S. Li, K. Xu y J. Liu, "Coal Mine Safety Accidents, Environmental Regulation and Economic Development—An Empirical Study of PVAR Based on Ten Major Coal Provinces in China," Sustainability, vol. 14, no. 21, p. 14334, 2022, doi: 10.3390/su1421334.

[7] S. Li, C. He, C. Y. Chen y W. C. Wang, The Theory and Practice of Double Prevention Mechanism of Production and Management Unit Safety. Xuzhou, China: Chinese Mining University Press, 2021.

[8] A. Monjazeb, J. Z. Sasiadek y D. Necsulescu, "Autonomous navigation among large number of nearby landmarks using FastSLAM and EKF-SLAM - A comparative study," en 2011 16th International Conference on Methods & Models in Automation & Robotics, Miedzyzdroje, Poland, 2011, pp. 369-374, doi: 10.1109/MMAR.2011.6031375.

[9] M. Montemerlo, S. Thrun, D. Koller y B. Wegbreit, "FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that converges," en Proc. of IJCAI, 2003.

[10] J. Hartmann, D. Forouher, M. Litza, J. H. Kluessendorff y E. Maehle, "Real-Time Visual SLAM Using FastSLAM and the Microsoft Kinect Camera," en ROBOTIK 2012; 7th German Conference on Robotics, Munich, Germany, 2012, pp. 1-6.

[11] N. Zikos y V. Petridis, "L-SLAM: Reduced dimensionality FastSLAM with unknown data association," en 2011 IEEE International Conference on Robotics and Automation, Shanghai, China, 2011, pp. 4074-4079, doi: 10.1109/ICRA.2011.5979921.

[12] A. Yusefı, A. Durdu y C. Sungur, "ORB-SLAM-based 2D Reconstruction of Environment for Indoor Autonomous Navigation of UAVs," Avrupa Bilim ve Teknoloji Dergisi, Ejosat Special Issue 2020 (ICCEES), pp. 466-472, 2020, doi: 10.31590/ejosat.819620.

[13] D. Kiss-Illés, C. Barrado y E. Salamí, "GPS-SLAM: An Augmentation of the ORB-SLAM Algorithm," Sensors, vol. 19, no. 22, art. 4973, 2019, doi: 10.3390/s19224973.

[14] Y. Dai, M. Zhao y L. Fortuna, "Grey Wolf Resampling-Based Rao-Blackwellized Particle Filter for Mobile Robot Simultaneous Localization and Mapping," Journal of Robotics, vol. 2021, p. 4978984, 2021, doi: 10.1155/2021/4978984.

[15] J. Sun, J. Zhao, X. Hu, H. Gao y J. Yu, "Autonomous Navigation System of Indoor Mobile Robots Using 2D LiDAR," Mathematics, vol. 11, no. 6, art. 1455, 2023, doi: 10.3390/math11061455.

[16] M. Fu, H. Zhu, Y. Yang, M. Wang y Z. Deng, "A navigation map building algorithm using refined RBPF-SLAM," en 2016 IEEE Chinese Guidance, Navigation and Control Conference (CGNCC), Nanjing, 2016, pp. 2483-2487, doi: 10.1109/CGNCC.2016.7829183.

[17] B. Behroozpour, P. A. M. Sandborn, M. C. Wu y B. E. Boser, "LiDAR System Architectures and Circuits," IEEE Communications Magazine, vol. 55, no. 10, pp. 135-142, Oct. 2017, doi: 10.1109/MCOM.2017.1700030.

[18] S. Lee, D. Lee, P. Choi y D. Park, "Accuracy–Power Controllable LiDAR Sensor System with 3D Object Recognition for Autonomous Vehicle," Sensors, vol. 20, no. 19, p. 5706, 2020. Disponible en: https://doi.org/10.3390/s20195706.

[19] Y. Chen, L. Cheng, M. Li, J. Wang, L. Tong y K. Yang, "Multiscale Grid Method for Detection and Reconstruction of Building Roofs from Airborne LiDAR Data," IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 7, no. 10, pp. 4081-4094, Oct. 2014, doi: 10.1109/JSTARS.2014.2306003.

[20] "Ouster OS0-32 LIDAR Sensor." [En línea]. Disponible en: https://atyges.es/tienda/ouster-os0-32-64-128-canales/. [Accedido: 26 de octubre de 2023].

[21] N. Ahmad, R. A. R. Ghazilla, N. M. Khairi y V. Kasi, "Reviews on various inertial measurement unit (IMU) sensor applications," International Journal of Signal Processing Systems, vol. 1, no. 2, pp. 256-262, 2013.

[22] S. Rahman, Basic Graph Theory. Springer Cham, 2017.

[23] M. Kulkarni et al., "Autonomous Teamed Exploration of Subterranean Environments using Legged and Aerial Robots," en 2022 International Conference on Robotics and Automation (ICRA), Philadelphia, PA, USA, 2022, pp. 3306-3313, doi: 10.1109/ICRA46639.2022.9812401.

[24] T. Dang, M. Tranzatto, S. Khattak, F. Mascarich, K. Alexis y M. Hutter, "Graph-based subterranean exploration path planning using aerial and legged robots," J. Field Robotics, vol. 37, pp. 1363-1388, 2020, doi: 10.1002/rob.21993.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, Introduction to Algorithms. MIT Press, 2022.

[26] J. C. Y. Cazar, E. G. S. Álvarez y J. E. V. Sampedro, "Algoritmo de Bellman-Ford para solucionar el problema de la ruta más corta entre nodos," Polo del Conocimiento: Revista científico-profesional, vol. 7, no. 7, pp. 1288-1302, 2022.

[27] W. Xu, Y. Cai, D. He, J. Lin y F. Zhang, "FAST-LIO2: Fast Direct LiDAR-Inertial Odometry," IEEE Transactions on Robotics, vol. 38, no. 4, pp. 2053-2073, Aug. 2022, doi: 10.1109/TRO.2022.3141876.

[28] J. A. Raya Vaquera, "Aceleración con CUDA de Procesos 3D," Universitat Politècnica de Catalunya, 2009. Disponible en: http://hdl.handle.net/2099.1/6805.

[29] E. Özbay y A. Çinar, "A voxelize structured refinement method for registration of point clouds from Kinect sensors," Engineering Science and Technology, an International Journal, vol. 22, Oct. 2018, doi: 10.1016/j.jestch.2018.09.012.