Generación de Rutas Seguras en Entornos Dinámicos

Miguel Alejandro Yáñez Martínez C-411

Resumen

El problema de optimización de rutas seguras en entornos dinámicos es un desafío fundamental en múltiples áreas, como el transporte, la logística y la planificación urbana. En este trabajo, se propone un enfoque basado en el algoritmo *Threshold Multiobjective Dijkstra Algorithm* (T-MDA), que permite calcular caminos eficientes en grafos dinámicos donde las aristas presentan pesos variables asociados a diferentes criterios de optimización, tales como el tiempo de recorrido, la distancia y la seguridad.

La metodología desarrollada integra técnicas avanzadas de poda y heurísticas multiobjetivo para reducir el número de soluciones exploradas, asegurando un rendimiento computacional adecuado. Además, se incorpora un mecanismo de actualización dinámica que permite ajustar las rutas sin necesidad de recomputar desde cero, lo que resulta fundamental en escenarios donde las condiciones del entorno cambian constantemente. Asimismo, se implementa un sistema de selección adaptativa de rutas basado en funciones de utilidad ponderadas, permitiendo ajustar las preferencias del usuario en tiempo real.

Los resultados experimentales muestran que el enfoque propuesto mejora significativamente la eficiencia en la búsqueda de caminos óptimos en comparación con métodos tradicionales, al tiempo que mantiene la calidad y diversidad de las soluciones obtenidas. Se presentan visualizaciones y análisis de desempeño que demuestran la capacidad del algoritmo para responder de manera eficiente a cambios en la estructura del grafo. Finalmente, se discuten posibles extensiones y mejoras futuras, incluyendo la aplicación de técnicas de aprendizaje automático para la predicción de cambios en el entorno y la optimización de la selección de rutas.

Palabras clave: Optimización de rutas, grafos dinámicos, algoritmos multiobjetivo, T-MDA, búsqueda de caminos mínimos, heurísticas, actualización dinámica.

Índice

1. Introducción

En el contexto de los sistemas de transporte modernos y el auge de las ciudades inteligentes, la planificación de rutas eficientes y seguras se ha convertido en un desafío crítico. La creciente complejidad de los entornos urbanos, caracterizada por condiciones cambiantes de tráfico, clima y eventos imprevistos, exige soluciones innovadoras capaces de garantizar tanto la seguridad de los usuarios como la eficiencia en el transporte. Este proyecto aborda el problema de generación de rutas seguras en entornos dinámicos, donde los factores que afectan a las rutas varían en tiempo real, lo que añade un nivel significativo de dificultad al problema de optimización.

La seguridad en las rutas no solo depende de minimizar el tiempo o la distancia, sino también de considerar variables dinámicas como la congestión del tráfico, la presencia de zonas peligrosas, accidentes o condiciones climáticas adversas. Además, la capacidad de adaptar las rutas en respuesta a eventos inesperados es esencial para reducir los riesgos y garantizar que las decisiones tomadas sean válidas bajo las condiciones actuales. Este enfoque es particularmente relevante en aplicaciones prácticas como la logística de entrega, el transporte público y la gestión de servicios de emergencia.

Las definiciones, conceptos y análisis presentados en este trabajo, incluyendo los Problemas de Caminos Más Cortos Multiobjetivo, las Relaciones de Dominancia, el Problema de Búsqueda Multiobjetivo, el Problema de Camino Más Corto Multiobjetivo de Uno-a-Uno, las Heurísticas y Cotas de Dominancia, así como la descripción del Algoritmo de Dijkstra Multiobjetivo Dirigido (T-MDA), su corrección, la complejidad de las búsquedas de nuevos caminos de cola, el impacto en la complejidad temporal, y la obtención de heurísticas y cotas de dominancia, han sido obtenidos del artículo [1]. Este artículo, escrito por Pedro Maristany de las Casas, Luitgard Kraus, Antonio Sedeño-Noda y Ralf Borndörfer, propone el T-MDA, una extensión del algoritmo de Dijkstra multiobjetivo que emplea técnicas inspiradas en el algoritmo A* para resolver de manera eficiente el problema de caminos más cortos multiobjetivo de uno-a-uno. Los análisis teóricos y resultados computacionales descritos en este trabajo constituyen la base fundamental para este estudio.

2. Formulación del Problema

El problema de generación de rutas seguras en entornos dinámicos se formula considerando un modelo de grafo, donde las aristas representan calles que pueden ser de uno o dos sentidos, y los costos se actualizan dinámicamente tras cada paso del trayecto. Este enfoque permite capturar las relaciones espaciales, las características dinámicas del entorno, y calcular la ruta en partes para adaptarse a las condiciones cambiantes. Los datos de entrada del problema son:

Grafo: $G = (V, E)$, donde V es el conjunto de nodos que representan ubicaciones, tales como intersecciones, calles o puntos de interés y E es el conjunto de aristas que conectan los nodos, representando los caminos entre ubicaciones. Cada arista puede ser bidireccional si ambas direcciones de la calle son transitables o unidireccional si la calle sólo es transitable en una dirección.

Cada arista $e_{ij} \in E$ está asociada con:

- t_{ij}: Tiempo estimado de viaje para recorrer la arista.

- d_{ij}: Distancia física de la arista.
- s_{ij}: Puntaje de seguridad de la arista, que refleja el nivel de riesgo asociado (por ejemplo, debido a zonas peligrosas, tráfico o condiciones climáticas).

Los valores de t_{ij}, d_{ij} y s_{ij} pueden variar dinámicamente en función de factores externos como tráfico, clima o eventos inesperados.

Punto de inicio y destino: Dos nodos $O, D \in V$, que representan el origen y el destino de la ruta deseada. La solución del problema es una ruta R, que es una secuencia de nodos calculada de manera iterativa:

$$R = \{O, v_1, v_2, ..., D\},$$

tal que:

- La ruta conecta O con D a través de nodos intermedios, respetando las restricciones de conectividad en el grafo y la dirección de las aristas. El nodo v_{i+1} debe ser accesible desde v_i mediante una arista válida en G.La dirección de las aristas debe respetarse, si $(v_i, v_{i+1}) \in E$ pero $(v_{i+1}, v_i) \notin E$, la ruta no puede tomar ese camino en reversa.
- La ruta minimiza el costo total, que combina tiempo de viaje, distancia y nivel de riesgo.

El objetivo principal es minimizar el costo que combina tiempo de viaje, distancia y seguridad, o sea, minimizar la función de costo:

$$\text{Costo}(R) = \alpha \sum_{e_{ij} \in R} d_{ij} + \beta \sum_{e_{ij} \in R} t_{ij} + \gamma \sum_{e_{ij} \in R} s_{ij},$$

donde α, β, γ son parámetros de ponderación que determinan la importancia relativa de cada criterio.

La ruta se calcula por partes, generando el siguiente nodo v_{i+1} desde el nodo actual v_i, utilizando los valores actuales de t_{ij}, d_{ij}, s_{ij} para determinar la arista óptima.Los valores de t_{ij}, d_{ij}, s_{ij} se actualizan dinámicamente en cada paso, por lo que una arista viable en un momento dado puede dejar de serlo más adelante.Tras el paso a un nuevo nodo, se actualizan dinámicamente los costos asociados a las aristas restantes en el grafo en función de la información en tiempo real (por ejemplo, cambios en tráfico o condiciones climáticas).

3. Análisis de Algoritmos

La optimización de rutas en grafos ha sido abordada con diversos enfoques, desde algoritmos clásicos hasta técnicas modernas basadas en inteligencia artificial y optimización evolutiva. En este apartado, se presentan diferentes algoritmos que han sido propuestos para resolver problemas de planificación de rutas, resaltando sus ventajas y desventajas en el contexto de sistemas dinámicos y multiobjetivo. Finalmente, se justifica la elección del algoritmo **T-MDA**, no como el mejor en términos generales, sino como el más adecuado para las necesidades específicas del problema tratado.

3.1. Algoritmo de Dijkstra

El algoritmo de Dijkstra [9] es un método clásico para encontrar la ruta más corta en un grafo con pesos no negativos. Su eficiencia lo ha convertido en una herramienta fundamental en redes de transporte y telecomunicaciones. Sin embargo, su rendimiento se ve afectado en entornos dinámicos donde los pesos de las aristas cambian frecuentemente, ya que debe recalcular rutas desde cero en cada actualización.

3.2. Algoritmo A*

El algoritmo A* [10] introduce una heurística en la búsqueda de caminos, mejorando la eficiencia en comparación con Dijkstra. Cuando la heurística es bien definida, A* encuentra rutas óptimas explorando menos nodos. No obstante, su desempeño depende críticamente de la calidad de la heurística utilizada, lo que puede ser una limitación en escenarios dinámicos.

3.3. NSGA-II

El *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) [11] es un algoritmo evolutivo ampliamente utilizado en optimización multiobjetivo. Su capacidad para encontrar múltiples soluciones Pareto-óptimas lo hace adecuado para problemas con múltiples criterios, como la planificación de rutas considerando tiempo, distancia y seguridad. Sin embargo, su alto costo computacional y la necesidad de evaluar múltiples generaciones de soluciones lo hacen menos viable para aplicaciones en tiempo real.

3.4. Deep Reinforcement Learning para Rutas Multiobjetivo

El *Deep Reinforcement Learning* (DRL) [12] ha sido aplicado en la optimización de rutas en entornos dinámicos, permitiendo que un agente aprenda políticas de decisión basadas en experiencias pasadas. Algoritmos como *Deep Q-Networks* (DQN) y *Proximal Policy Optimization* (PPO) [13] han demostrado su capacidad para adaptarse a cambios en tiempo real, lo que los hace atractivos para sistemas de tráfico inteligente y redes logísticas. Sin embargo, requieren un tiempo de entrenamiento considerable y una gran cantidad de datos históricos para lograr una generalización efectiva.

3.5. Redes Neuronales de Grafos (GNNs)

Las *Graph Neural Networks* (GNNs) [14] han emergido como una alternativa para la optimización de rutas al permitir la representación eficiente de grafos complejos. Estas redes pueden predecir rutas óptimas sin realizar una búsqueda exhaustiva en el grafo, lo que reduce significativamente los tiempos de cómputo. Sin embargo, su rendimiento depende de un entrenamiento supervisado con datos de tráfico, lo que limita su aplicabilidad en entornos con información insuficiente o en escenarios altamente dinámicos.

3.6. Optimización de Flujo en Redes mediante Programación Convexa

La *programación convexa* [15] se ha utilizado para modelar problemas de flujo en redes, minimizando los tiempos de tránsito y evitando la congestión mediante la distribución

óptima del tráfico. A diferencia de los algoritmos de búsqueda de caminos individuales, este enfoque optimiza toda la red simultáneamente. Aunque su aplicación ha sido exitosa en el diseño de infraestructuras de transporte y telecomunicaciones, la resolución de los sistemas de ecuaciones involucrados puede ser computacionalmente costosa, lo que dificulta su uso en escenarios donde se requiere una respuesta inmediata.

3.7. Elección del Algoritmo T-MDA

El algoritmo **T-MDA** ha sido seleccionado para este estudio no porque sea el mejor en términos absolutos, sino porque se adapta de manera eficiente a la planificación de rutas en entornos dinámicos y multiobjetivo. A diferencia de NSGA-II, que requiere múltiples generaciones para converger a soluciones óptimas, T-MDA encuentra caminos eficientes en tiempo real sin necesidad de simulaciones extensivas. En comparación con DRL y GNNs, no necesita una fase de entrenamiento previa ni depende de datos históricos, lo que lo hace más versátil en escenarios cambiantes. Además, su capacidad para incorporar técnicas de poda reduce significativamente el número de soluciones dominadas, optimizando el proceso de búsqueda de rutas en grafos dinámicos.

Entre sus ventajas se destacan su eficiencia computacional y su capacidad para manejar múltiples criterios simultáneamente sin la sobrecarga de los algoritmos evolutivos. No obstante, su desempeño puede degradarse en grafos extremadamente grandes donde la cantidad de soluciones no dominadas crece exponencialmente, lo que podría requerir estrategias adicionales para la reducción del frente de Pareto. A pesar de esta limitación, T-MDA se presenta como una solución efectiva para el problema tratado, permitiendo la planificación de rutas dinámicas con tiempos de cómputo razonables y sin comprometer la calidad de las soluciones obtenidas.

4. Análisis de Complejidad

Para demostrar que el problema de generación de rutas seguras en entornos dinámicos es al menos NP-Hard, se realiza una reducción desde el problema de la *Mochila Binaria* (0/1 Knapsack Problem), el cual es un problema NP-completo ampliamente estudiado [6], lo que implica que no existe un algoritmo polinómico conocido para resolverlo en el caso general.

El problema de la mochila binaria se define de la siguiente manera: dado un conjunto de n objetos, cada uno con un peso w_i y un valor v_i, y una mochila con una capacidad máxima W, el objetivo es seleccionar un subconjunto de objetos tal que la suma de los pesos no exceda W y el valor total sea máximo. Formalmente, se busca un conjunto de índices $S \subseteq \{1, 2, ..., n\}$ tal que:

$$\sum_{i \in S} w_i \leq W, \quad \text{maximizando} \quad \sum_{i \in S} v_i.$$

Este problema es NP-completo cuando los pesos y valores son números enteros y se requiere una solución exacta. Se ha demostrado [5] que el problema de caminos más cortos con múltiples criterios (MOSP) puede transformarse en una instancia del problema de la mochila binaria, lo que implica que MOSP es NP-Hard.

La reducción se construye de la siguiente manera. Dada una instancia del problema de la mochila binaria con n objetos y una capacidad W, se define un grafo $G = (V, E)$ donde

cada arista representa la inclusión o exclusión de un objeto en la mochila. El grafo se organiza en capas: para cada objeto O_i, existe una capa C_i con dos nodos, representando la selección o exclusión de O_i. Desde cada nodo de C_i parten dos aristas hacia la siguiente capa, correspondientes a cada decisión posible. La capa C_0 contiene un único nodo de origen, mientras que C_{n+1} tiene el nodo de destino.

Los pesos de las aristas se asignan de modo que la primera función objetivo represente la suma de los pesos de los objetos seleccionados (peso total), y la segunda función objetivo represente el opuesto de la sumatoria de sus valores (valor total). Para las $|K|-2$ características restantes, se asigna un valor de cero, garantizando su optimización trivial.

El objetivo en esta instancia de MOSP es encontrar un conjunto de caminos en el grafo que minimicen tanto el peso total como el opuesto del valor total. Esto es análogo al problema de la mochila, ya que minimizar el opuesto del valor total equivale a maximizar el valor total. Dado que la construcción del grafo involucra $O(2n)$ vértices y $O(4n)$ aristas, su complejidad temporal es $O(n)$. En consecuencia, MOSP conserva la complejidad del problema de la mochila binaria, lo que confirma que es NP-Hard.

En el caso de grafos dinámicos, donde los pesos de las aristas varían con el tiempo, se puede reducir MOSP en un grafo estándar a su equivalente en un grafo dinámico sin modificaciones estructurales. Un grafo estático puede interpretarse como un caso particular de un grafo dinámico donde, en cada instante de tiempo, los parámetros de las aristas se les adiciona 0, por lo que permanecen constantes.

Como resultado de esta reducción, se concluye que el problema de generación de rutas seguras en entornos dinámicos es al menos NP-Hard.

5. Problemas de Caminos Más Cortos Multiobjetivo

El problema de la búsqueda del camino más corto multiobjetivo es una extensión del problema del camino más corto, donde las aristas están etiquetadas con vectores de costos. Cada componente en un vector de costos representa un atributo relevante diferente a minimizar, por ejemplo, distancia, tiempo, riesgo. Estos problemas rara vez tienen una solución óptima única. Con mayor frecuencia, se encuentra un conjunto de soluciones no dominadas (óptimas de Pareto), cada una de las cuales presenta un compromiso particular entre los objetivos bajo consideración. Se han desarrollado múltiples técnicas para resolver problemas de caminos más cortos multiobjetivo.

5.1. Relaciones de dominancia

En el contexto de los problemas de caminos más cortos multiobjetivo, los conceptos de **caminos dominados** y **caminos no dominados** son fundamentales para evaluar las soluciones cuando se consideran múltiples objetivos. Estos conceptos se basan en la **relación de dominancia** entre los vectores de costos asociados a los caminos.

Caminos dominados

Sea $G = (N, A, \vec{c})$ un grafo dirigido, donde las aristas están etiquetadas con vectores de costos $\vec{c}(n, n') \in \mathbb{R}^q$. El costo de un camino P se define como un vector q-dimensional:

$$\vec{C}_P = \sum_{(n,n')\in P} \vec{c}(n, n').$$

Dado un conjunto de vectores de costos, se dice que un camino domina a otro si cumple la siguiente relación de dominancia:

Sean dos vectores $\vec{v}, \vec{v}' \in \mathbb{R}^q$, decimos que $\vec{v}$ domina a $\vec{v}'$ ($\vec{v} \preceq \vec{v}'$) si y solo si:

$$\forall i \in \{1, \ldots, q\},\ v_i \leq v'_i \quad \text{y} \quad \vec{v} \neq \vec{v}'.$$

Esto significa que el vector $\vec{v}$ es estrictamente mejor o igual en todos los objetivos y estrictamente mejor en al menos uno de ellos.

Por otro lado, un camino se considera **dominado** si existe al menos otro camino cuyo vector de costos es mejor o igual en todos los objetivos y estrictamente mejor en al menos uno. Es decir, sea P el conjunto de todos los caminos posibles en G, un camino $P_D \in P$ es dominado si:

$$\exists P' \in P \quad \text{tal que} \quad \vec{C}_{P'} \preceq \vec{C}_{P_D}.$$

Los caminos dominados no son soluciones eficientes, ya que siempre existe al menos un camino alternativo que es igual o mejor en todos los objetivos.

Caminos no dominados

Un camino se considera **no dominado** si su vector de costos no es dominado por el vector de costos de ningún otro camino dentro del mismo conjunto. Formalmente, sea P el conjunto de todos los caminos posibles en G que conectan un nodo fuente s con un nodo objetivo t. Un camino $P_E \in P$ es no dominado si:

$$\nexists P' \in P \quad \text{tal que} \quad \vec{C}_{P'} \preceq \vec{C}_{P_E}.$$

Los caminos no dominados representan un conjunto eficiente de soluciones en el cual cada camino ofrece un compromiso óptimo entre los objetivos. Este conjunto de caminos no dominados se conoce como el **conjunto de Pareto**.

Sea P un problema de optimización multiobjetivo, se dice entonces que una solución S_1 es pareto-óptima cuando no existe otra solución S_2 tal que mejore en un objetivo sin empeorar al menos uno de los otros.

En términos formales, dado un conjunto de soluciones S, una solución S_1 se considera *Pareto-óptima* si no existe otra solución S_2 tal que S_2 sea mejor en al menos un criterio sin empeorar en otro. Es decir, una solución S_1 es Pareto-óptima si no existe S_2 tal que:

$$\forall i, f_i(S_2) \leq f_i(S_1) \quad \text{y} \quad \exists j, f_j(S_2) < f_j(S_1),$$

donde f_i representa la función objetivo para el criterio i.

5.2. El Problema de Búsqueda Multiobjetivo

El problema de caminos más cortos multiobjetivo se define en un grafo dirigido $G = (V, A)$, donde V es el conjunto de nodos y A es el conjunto de arcos. Cada arco $(i, j) \in A$ conecta dos nodos $i, j \in V$, y está asociado con $|K|$ costos, representados por c_{ij}^k, con $k \in K$, donde K es el conjunto de criterios. Suponemos que todos los costos son no negativos, es decir, $c_{ij}^k \geq 0\, \forall (i, j) \in A,\ \forall k \in K$.

El objetivo del problema es determinar un conjunto de caminos P que conecten un nodo fuente s con un nodo destino t, minimizando una función objetivo de tipo suma

para cada criterio. Formalmente, el costo total de un camino P se define como:

$$\vec{C}(P) = \left(\sum_{(i,j)\in P} c_{ij}^1, \sum_{(i,j)\in P} c_{ij}^2, \ldots, \sum_{(i,j)\in P} c_{ij}^k \right),$$

donde cada componente del vector corresponde al costo acumulado de un criterio k a lo largo del camino P.

La solución del problema es un conjunto de caminos estrictamente no dominados, o sea, el **conjunto de Pareto**.

Este conjunto tiene dos propiedades fundamentales: primero, ningún camino en el frente de Pareto puede ser mejorado en un criterio sin empeorar al menos otro criterio (principio de no dominancia). Segundo, cada camino en el frente de Pareto representa un compromiso óptimo entre los diferentes objetivos, permitiendo a los usuarios seleccionar soluciones dependiendo de sus prioridades específicas.

6. Formalización de la solución para el problema de generación de rutas seguras en entornos dinámicos

La solución del problema consiste en resolver la Búsqueda Multiobjetivo para $|K| = 3$ y construir el **conjunto de Pareto** a partir del cual se elige la solución que minimice la función de coste. El problema se resuelve de manera iterativa para adaptarse a las condiciones dinámicas del entorno. En cada paso, se evalúan las aristas salientes del nodo actual v_i utilizando los valores dinámicos de t_{ij}, d_{ij} y s_{ij}. Una vez seleccionado el siguiente nodo v_{i+1}, los costos de las aristas restantes en el grafo se actualizan en función de información en tiempo real, como cambios en el tráfico, clima u otros eventos inesperados,actualizando el conjunto de Pareto. Este enfoque garantiza que todas las restricciones del grafo sean respetadas, incluyendo las direcciones de las aristas (unidireccionales o bidireccionales). Además, se asegura que todos los nodos en la ruta estén conectados y que la ruta comience en el nodo O y termine en el nodo D.

7. Subproblemas y Estrategias de Solución

El problema de optimización de rutas seguras en entornos dinámicos requiere un enfoque capaz para manejar múltiples objetivos de manera simultánea, al tiempo que se adapta a cambios en tiempo real en el grafo subyacente. La estrategia que proponemos se fundamenta en el uso del algoritmo T-MDA [1] (Threshold Multiobjective Dijkstra Algorithm). Como se explica en el **Anexo C**, es un algoritmo diseñado para encontrar caminos óptimos en grafos con múltiples criterios de evaluación, tales como tiempo de recorrido, distancia y seguridad. Además, se complementa con un mecanismo de actualización dinámica que permite incorporar cambios en los pesos de las aristas sin necesidad de recalcular completamente la solución desde cero, garantizando así una respuesta eficiente en entornos cambiantes.

El problema de *Generación de Rutas Seguras en Entornos Dinámicos* puede descomponerse en varios subproblemas fundamentales, cada uno de los cuales aborda un aspecto clave de la optimización de rutas en un grafo dinámico con múltiples criterios. A continuación, se presentan las definiciones formales de estos subproblemas.

7.1. Subproblema 1: Búsqueda de Caminos Mínimos Multiobjetivo

En el problema de caminos mínimos multiobjetivo, el enfoque tradicional basado en algoritmos de caminos más cortos, como Dijkstra o Bellman-Ford no es adecuado, ya que dichos métodos están diseñados para minimizar un único criterio. La resolución del problema requiere un método capaz de manejar múltiples dimensiones de optimización simultáneamente, preservando la eficiencia computacional y evitando la generación innecesaria de soluciones subóptimas. Además, debido a la naturaleza dinámica del problema, el algoritmo debe ser capaz de adaptarse a cambios en el grafo en tiempo real sin necesidad de recalcular todas las soluciones desde cero.

Para abordar este subproblema, se propone la aplicación del algoritmo *Threshold Multiobjective Dijkstra Algorithm* (T-MDA).Este algoritmo extiende el enfoque clásico de Dijkstra incorporando técnicas de poda y heurísticas avanzadas que permiten reducir el número de soluciones exploradas y mejorar la eficiencia en la construcción del frente de Pareto. El frente de Pareto se construye mediante la evaluación iterativa de las rutas generadas, comparando sus vectores de costos acumulados. Si una ruta P_1 no es dominada por ninguna otra ruta P_2 previamente explorada, se agrega al frente de Pareto.

T-MDA emplea consultas léxicas de caminos más cortos para calcular heurísticas que guían la búsqueda, generando puntos ideales que representan las soluciones óptimas individuales para cada criterio. A partir de estos puntos, se construyen cotas de dominancia que permiten descartar rutas ineficientes de manera temprana, reduciendo la cantidad de soluciones almacenadas sin comprometer la calidad del resultado. La selección de caminos se realiza mediante una estrategia iterativa en la que se comparan costos acumulativos y costos reducidos, evaluando si una nueva ruta domina o es dominada por rutas previamente almacenadas.

Una de las principales ventajas de T-MDA es su capacidad para manejar grafos dinámicos en los que los pesos de las aristas pueden cambiar en tiempo real. En lugar de reiniciar la búsqueda desde el inicio ante cada modificación del grafo, el algoritmo incorpora un mecanismo de actualización incremental que ajusta únicamente las soluciones afectadas por los cambios. Este mecanismo de propagación de modificaciones permite una respuesta eficiente a variaciones en el entorno, asegurando que las rutas calculadas reflejen las condiciones actuales de la red.

El resultado de la ejecución del algoritmo es un conjunto de rutas eficientes que conforman el frente de Pareto, proporcionando al usuario múltiples opciones optimizadas en función de sus necesidades. Dado que en entornos reales las preferencias pueden variar según la situación, la solución final no se basa en un único camino, sino en una selección adaptativa que permite ajustar dinámicamente la ponderación de los criterios de optimización. Este proceso de selección se realiza mediante la función de utilidad ponderad, que asigna pesos a cada criterio y elige la ruta más adecuada según la configuración establecida en cada momento.

La función de utilidad ponderada se define como:

$$U(P) = \alpha \cdot T(P) + \beta \cdot D(P) + \gamma \cdot S(P),$$

donde α, β, γ son los pesos que el usuario puede ajustar en tiempo real según sus prioridades. Por ejemplo, en situaciones de emergencia, el peso α (tiempo de viaje) puede ser mayor, mientras que en condiciones normales, el peso γ (seguridad) podría tener mayor relevancia.

En términos de complejidad, T-MDA mejora significativamente el rendimiento en comparación con métodos multiobjetivo tradicionales, gracias a su enfoque basado en poda y heurísticas. Sin embargo, la complejidad sigue estando influenciada por el número de criterios y la estructura del grafo, lo que implica que en casos con un gran número de soluciones no dominadas, puede ser necesario aplicar estrategias adicionales de reducción del espacio de soluciones para mantener la eficiencia.

La solución propuesta mediante T-MDA permite abordar de manera efectiva el problema de búsqueda de caminos mínimos multiobjetivo en entornos dinámicos. Su combinación de poda, heurísticas y actualización incremental garantiza un equilibrio entre calidad de las soluciones y eficiencia computacional, proporcionando un enfoque robusto para la optimización de rutas seguras en escenarios cambiantes.

Dado que el subproblema de búsqueda de caminos mínimos multiobjetivo se resuelve utilizando el algoritmo *Time-based Multiobjective Dijkstra Algorithm* (T-MDA), es necesario diseñar estrategias complementarias para abordar los desafíos asociados con la actualización dinámica de pesos, la selección óptima de rutas en tiempo real y la reducción del espacio de soluciones. Estas soluciones deben integrarse con la estructura de T-MDA para garantizar la escalabilidad del sistema y su aplicabilidad en entornos dinámicos.

7.2. Subproblema 2: Actualización Dinámica de Pesos en el Grafo

El problema de actualización dinámica de pesos surge debido a la variabilidad de las condiciones externas que afectan las aristas del grafo. En un entorno dinámico, los valores de los pesos $w(e_{ij})$ pueden cambiar con el tiempo. Se requiere un mecanismo para actualizar los valores de $w(e_{ij}, t)$ en cada instante de tiempo t, sin necesidad de recomputar todas las soluciones desde cero, lo que sería computacionalmente ineficiente.Para dar solución a lo anterior, se propone un modelo de actualización basado en eventos. En este enfoque, los cambios en los pesos de las aristas se procesan de manera incremental, afectando únicamente los caminos que incluyen dichas aristas.

Formalmente, sea $G_t = (V, E, W_t)$ la instancia del grafo en el instante t, donde W_t representa la función de pesos en cada arista. Si un evento externo modifica el peso de una arista $e_{ij} \in E$, el peso en el instante $t + 1$ se actualiza como:

$$w(e_{ij}, t + 1) = f(w(e_{ij}, t), X_t),$$

donde X_t representa la información externa que afecta el sistema. El desafío es desarrollar estructuras de datos eficientes que permitan estas actualizaciones sin comprometer la eficiencia del algoritmo de búsqueda de caminos.Para mantener la eficiencia, se emplea una estructura de datos basada en colas de prioridad y listas de aristas activas, permitiendo que T-MDA solo recalcule los caminos que han sido afectados por los cambios en los pesos. Esto evita la recomputación global de rutas y mejora la escalabilidad del algoritmo.

7.3. Subproblema 3: Selección Óptima de Rutas bajo Cambios en Tiempo Real

Dado que los pesos de las aristas pueden cambiar en tiempo real, es necesario un método para determinar en cada instante cuál es la mejor ruta a seguir, considerando la información más reciente. Sea $P_t(O, D)$ el conjunto de rutas de Pareto en el instante t, el objetivo es definir una estrategia de selección S tal que:

$$P_t^{\text{seleccionado}} = S(P_t(O, D), C_t)$$

donde C_t representa las condiciones actuales del sistema y restricciones específicas del usuario, como una prioridad sobre la seguridad o la minimización del tiempo de viaje. La estrategia S puede basarse en métodos heurísticos o modelos predictivos que anticipen cambios futuros en los pesos de las aristas.

La estrategia de selección S puede combinar métodos heurísticos con modelos predictivos basados en datos históricos de tráfico, condiciones climáticas y eventos pasados para anticipar futuros cambios en el entorno. Por ejemplo, un modelo de regresión basado en series temporales podría predecir las condiciones de tráfico en función del horario del día.

7.4. Subproblema 4: Reducción del Espacio de Soluciones

El crecimiento exponencial del número de soluciones no dominadas en el problema multiobjetivo impone la necesidad de aplicar estrategias de poda para reducir el número de soluciones a considerar sin perder información relevante. Se requiere un método eficiente para eliminar soluciones dominadas o poco relevantes mientras se mantiene la diversidad en el conjunto de soluciones de Pareto.

Formalmente, el problema consiste en encontrar un subconjunto $P_r^*(O, D) \subseteq P^*(O, D)$ tal que:

$$|P_r^*(O, D)| \ll |P^*(O, D)|$$

sujeto a la condición de que la calidad de la aproximación del conjunto de Pareto original se mantenga dentro de un margen aceptable, medido por una métrica de dispersión o cobertura.

En cada instante de tiempo, el algoritmo T-MDA genera un conjunto de soluciones de Pareto, lo que implica que, en lugar de una única ruta óptima, se obtiene un conjunto de caminos no dominados que representan diferentes compromisos entre los criterios de tiempo, distancia y seguridad. La selección de la ruta a seguir en cada momento depende de restricciones operativas o preferencias específicas del usuario.

Para resolver este subproblema, se introduce una función de utilidad U que permite evaluar y seleccionar el camino más adecuado según las condiciones actuales y las preferencias del usuario.

Dada una ruta P con vector de pesos $w(P) = (t_P, d_P, s_P)$, la función de utilidad se define como:

$$U(P) = \alpha \cdot t_P + \beta \cdot d_P + \gamma \cdot s_P,$$

donde α, β, γ son parámetros de ponderación que pueden ajustarse en tiempo real para reflejar las prioridades del usuario.. En cada instante, se selecciona la ruta P^* que minimiza $U(P)$, asegurando que la decisión sea adaptativa a los cambios dinámicos del sistema. Esta estrategia permite flexibilidad en la toma de decisiones y puede combinarse con modelos predictivos para anticipar futuras alteraciones en los pesos de las aristas.

8. Algoritmo de Búsqueda de Caminos Multiobjetivo en Grafos Dinámicos

El algoritmo implementado es una extensión del algoritmo de Dijkstra que permite la búsqueda de rutas óptimas en grafos dirigidos con múltiples criterios de optimización. Su característica principal es la capacidad de adaptarse dinámicamente a cambios en los pesos de las aristas, permitiendo la actualización de rutas en tiempo real sin necesidad de recomputar todas las soluciones desde cero.

Para resolver este problema, el algoritmo mantiene un frente de Pareto dinámico de caminos eficientes, actualiza los pesos en tiempo real y aplica heurísticas para guiar la búsqueda de soluciones óptimas en un entorno dinámico. La ejecución del algoritmo se realiza mediante la función `navigate` de la calse `DynamicTMDA`, que devuelve iterativamente los nodos del camino óptimo a seguir.

8.1. Estructura General del Algoritmo

El algoritmo sigue una estructura basada en cuatro componentes principales:

1. **Inicialización**: Se establece el grafo dinámico, los criterios de optimización y los nodos de origen y destino. Se precalculan heurísticas y cotas de dominancia para guiar la búsqueda.

2. **Exploración del Grafo**: Se emplea una variante multiobjetivo del algoritmo de Dijkstra para encontrar caminos eficientes utilizando una cola de prioridad.

3. **Actualización Dinámica**: Se actualizan los pesos de las aristas conforme cambian las condiciones del entorno, evitando recomputaciones globales y recalculando solo las rutas afectadas.

4. **Selección Óptima y Predicción**: Se elige el mejor camino en cada iteración basándose en heurísticas y modelos de predicción de cambios futuros en los pesos.

Cada uno de estos componentes se implementa mediante una serie de subalgoritmos que permiten la ejecución del proceso de navegación.

9. Algoritmo de Búsqueda de Caminos Multiobjetivo en Grafos Dinámicos

El algoritmo implementado en la clase `DynamicTMDA` se basa en una variante del algoritmo T-MDA [1] (Threshold Multiobjective Dijkstra Algorithm). Su propósito es encontrar rutas óptimas en un grafo dirigido donde cada arista tiene múltiples atributos de costo, como tiempo, distancia y seguridad. Además, el algoritmo es capaz de adaptarse dinámicamente a cambios en los pesos de las aristas, permitiendo la actualización de rutas en tiempo real sin necesidad de recomputar desde cero.

El algoritmo inicia con la creación de una instancia de la clase `DynamicTMDA`, donde se proporciona el grafo, la lista de dimensiones a optimizar junto con sus respectivas preferencias, el nodo origen y destino, además de un componente encargado de actualizar

dinámicamente los pesos de las aristas. Durante la inicialización, se calculan heurísticas basadas en consultas léxicas de Dijkstra, las cuales permiten guiar la búsqueda y establecer cotas de dominancia para reducir el espacio de soluciones.

Para la exploración del grafo, se utiliza una cola de prioridad ordenada lexicográficamente según los costos reducidos de los caminos explorados. Cada nodo procesado almacena un conjunto de soluciones de Pareto que representan rutas eficientes sin ser dominadas por otras. A medida que se expanden los caminos, se descartan aquellos que son dominados o que exceden las cotas establecidas, reduciendo significativamente la cantidad de rutas a considerar. Esta poda de soluciones es clave para la eficiencia del algoritmo.

La actualización de pesos en tiempo real se realiza a través de la clase `DynamicEdgeUpdater`. Cuando una arista es modificada, su nuevo peso se almacena en un historial y se utiliza un modelo de regresión lineal para predecir futuras variaciones. Basándose en esta predicción, el algoritmo ajusta dinámicamente su criterio de selección de rutas, favoreciendo caminos que minimicen el impacto de variaciones inesperadas en el costo de las aristas.

El proceso de navegación se realiza mediante el método `navigate`, el cual devuelve iterativamente los nodos que se deben visitar para completar el camino. En cada iteración, el algoritmo evalúa el estado actual del grafo y decide si continuar con el camino previamente calculado o si es necesario recalcularlo debido a modificaciones en los pesos de las aristas. Para ello, se comparan los valores históricos de las aristas con los cambios recientes y, si la diferencia es significativa, se ejecuta nuevamente el cálculo de caminos eficientes desde el nodo actual.

El algoritmo también incorpora una función de selección óptima de rutas basada en una combinación de heurísticas y predicciones. Utilizando una función de utilidad ponderada, se evalúan los caminos candidatos según las preferencias del usuario y las condiciones del entorno, seleccionando aquel que minimiza la suma ponderada de sus costos ajustados por los factores de predicción. De esta manera, se logra una adaptación en tiempo real que mejora la eficiencia y seguridad de la navegación en entornos dinámicos.

A nivel de implementación, el algoritmo emplea estructuras de datos eficientes como colas de prioridad para la selección de caminos, diccionarios para el almacenamiento de soluciones de Pareto y listas de doble enlace para gestionar el historial de cambios en los pesos de las aristas. La combinación de estas estructuras permite una ejecución eficiente incluso en grafos de gran escala.

En conclusión, el algoritmo desarrollado en la clase `DynamicTMDA` implementa una solución avanzada para la búsqueda de caminos en grafos dinámicos con múltiples criterios de optimización. Gracias a su capacidad de adaptación y su enfoque basado en heurísticas y predicciones, es capaz de ofrecer rutas eficientes en tiempo real, minimizando la recomputación y garantizando decisiones óptimas a lo largo del recorrido.

9.1. Inicialización y Preprocesamiento

La inicialización del algoritmo se realiza en el constructor de la clase `DynamicTMDA`. Durante este proceso, se almacenan los parámetros de entrada y se construyen las estructuras de datos necesarias para la ejecución del algoritmo.

Una parte clave de esta etapa es el cálculo de heurísticas mediante consultas léxicas de Dijkstra. Para cada nodo en el grafo, se ejecuta el algoritmo `dijkstra_lexicographic`, que calcula el costo mínimo hacia el nodo destino según diferentes órdenes de prioridad de las dimensiones. Los resultados de estas consultas se combinan para obtener puntos ideales que sirven como heurísticas, mejorando la eficiencia en la exploración del grafo.

Además, se establece una cota de dominancia basada en los valores heurísticos, la cual se utiliza para descartar caminos ineficientes durante la ejecución.

9.2. Algoritmo de Navegación - `navigate`

El algoritmo `navigate` es el núcleo del sistema de búsqueda de caminos en entornos dinámicos. Su objetivo principal es encontrar una secuencia óptima de nodos desde un nodo fuente s hasta un nodo destino t, considerando múltiples criterios de optimizació. Este proceso se realiza de manera iterativa, adaptándose dinámicamente a cambios en los pesos de las aristas.

El algoritmo inicia ejecutando `run`, que calcula un conjunto de caminos eficientes utilizando el método de optimización multiobjetivo. A partir de este conjunto, se selecciona el mejor camino disponible mediante `select_best_path`. Se establece el nodo inicial de la ruta y se inicializa una estructura para almacenar la ruta completa y sus costos acumulados.

El proceso de navegación ocurre en un bucle donde, en cada iteración, se actualizan dinámicamente los pesos de las aristas del camino actual con la función `update_weights`. A continuación, se evalúan todas las aristas salientes desde el nodo actual, explorando posibles alternativas de ruta. Para cada vecino del nodo actual, se ejecuta `recalculate`, lo que permite obtener un nuevo conjunto de caminos eficientes considerando los cambios recientes en el grafo.

Cada camino candidato se evalúa en función de sus costos y se selecciona el que minimiza la función de costo total. Una vez determinado el mejor camino local, el algoritmo avanza al siguiente nodo y actualiza los costos acumulados en la estructura de la ruta. En cada paso, se devuelve el nodo actual, lo que permite una ejecución incremental del algoritmo.

Si en algún momento no se encuentra un camino válido desde el nodo actual, se genera un error, indicando que la navegación no puede continuar. Al finalizar el proceso, se obtiene la ruta completa seguida y los costos asociados a cada dimensión de optimización.

Algorithm 1: Algoritmo de Navegación - `navigate`

Input: Grafo dinámico $G(V, E)$, nodo fuente s, nodo objetivo t, criterios de optimización `dimensions`
Output: Secuencia de nodos que forman la ruta óptima
Ejecutar `run()` para calcular caminos eficientes;
Seleccionar el mejor camino P con `select_best_path()`;
$current\_node \leftarrow P[0]$;
$previous\_node \leftarrow$ None;
Inicializar el historial de costos y la ruta completa;
while $current\_node \neq t$ **do**
    Actualizar pesos de las aristas con `update_weights(P)`;
    Obtener vecinos de $current\_node$;
    $best\_local\_path \leftarrow$ None, $best\_local\_cost \leftarrow \infty$;
    if *Se modificaron aristas o el frente de Pareto cambió en* $current\_node$ **then**
        Eliminar caminos inválidos del frente de Pareto;
        Recalcular caminos eficientes desde $current\_node$ con `recalculate(current_node)`;
    foreach *vecino* v *de* $current\_node$ **do**
        if $v = previous\_node$ **then**
            (*Evitar retrocesos*);
        Evaluar caminos desde v con `select_best_path()`;
        $candidate\_cost \leftarrow$ Suma de costos de $candidate\_path$;
        if $candidate\_cost < best\_local\_cost$ **then**
            $best\_local\_path \leftarrow candidate\_path$;
            $best\_local\_cost \leftarrow candidate\_cost$;
    if $best\_local\_path$ *es None* **then**
        Error: No se encontró un camino válido;
    $previous\_node \leftarrow current\_node$;
    $current\_node \leftarrow$ siguiente nodo en $best\_local\_path$;
    Acumular costos en la ruta completa;
    $current\_node$;
return Ruta completa seguida;

Este enfoque permite que el algoritmo se adapte a cambios dinámicos en el grafo sin necesidad de recalcular completamente todas las soluciones desde cero, mejorando la eficiencia y permitiendo su aplicación en entornos con restricciones de tiempo real. Además, al utilizar estrategias de poda y heurísticas basadas en aprendizaje de cambios previos, se optimiza la selección de rutas minimizando recomputaciones innecesarias.

9.3. Algoritmo de Ejecución - `run`

El método `run` es el núcleo del proceso de búsqueda de caminos eficientes en el grafo dinámico. Su objetivo es encontrar un conjunto de rutas óptimas desde el nodo origen s hasta el nodo destino t mediante la exploración de múltiples dimensiones de costos. Para ello, el algoritmo emplea una estrategia basada en optimización multiobjetivo, utilizando un enfoque de Pareto para mantener múltiples soluciones no dominadas.

El algoritmo comienza inicializando la cola de prioridad con un camino trivial que solo contiene el nodo fuente. Este camino tiene costos iniciales de cero y su costo reducido está determinado por la heurística precomputada del nodo fuente. A partir de esta configuración inicial, el algoritmo entra en un bucle en el que extrae repetidamente el camino con el menor costo reducido de la cola de prioridad y lo evalúa.

Cada camino extraído es almacenado en el frente de Pareto del nodo correspondiente, lo que garantiza que solo se mantengan caminos eficientes. Posteriormente, el algoritmo intenta extender la exploración buscando caminos alternativos menos eficientes que pueden ser útiles en iteraciones futuras. Para ello, se ejecuta `next_queue_path`, que selecciona un camino previamente descartado si este sigue siendo viable en la nueva iteración.

El proceso de propagación ocurre expandiendo el nodo actual hacia todos sus vecinos en el grafo. Para cada vecino, el algoritmo genera un nuevo camino extendido y verifica si este es eficiente en comparación con las soluciones ya encontradas. Si el camino no está dominado por soluciones previas, se agrega a la cola de prioridad para futuras iteraciones.

Este proceso continúa hasta que la cola de prioridad se vacía, momento en el cual el algoritmo devuelve el conjunto de caminos óptimos encontrados hasta el destino. La estructura del método `run` garantiza que el algoritmo sea capaz de adaptarse a cambios dinámicos en el grafo, minimizando la recomputación y preservando la eficiencia del sistema.

Algorithm 2: Algoritmo `run`

Input: $G(V, E)$ Grafo dirigido con múltiples dimensiones de costo
Input: s Nodo origen, t Nodo destino
Input: heuristic[v] Heurística precomputada para cada nodo
Output: $P_t(s, t)$ Conjunto de caminos eficientes desde s hasta t
Inicializar cola de prioridad priority_queue $\leftarrow \emptyset$;
Inicializar frente de Pareto pareto_front $\leftarrow \emptyset$;
Crear camino inicial $p_{\text{init}} \leftarrow \{s\}$ con costos 0 y costo reducido heuristic[s];
Insertar p_{init} en priority_queue;
while *priority_queue* $\neq \emptyset$ **do**
    Extraer camino p_{current} con menor costo reducido de priority_queue;
    $v \leftarrow$ último nodo en p_{current};
    Almacenar p_{current} en pareto_front[v];
    $p_{\text{queue}} \leftarrow$ next_queue_path(v);
    if $p_{queue} \neq None$ **then**
        Insertar p_{queue} en priority_queue;
    end
    foreach $(v, w) \in E$ **do**
        propagate($p_{\text{current}}, (v, w)$);
    end
end
return *pareto_front*[t] ;

9.4. Algoritmo de Recalculación - `recalculate`

El método `recalculate` es fundamental para la adaptabilidad del algoritmo en entornos dinámicos. Su función principal es actualizar los caminos eficientes después de que se han modificado los pesos de las aristas en el grafo, garantizando que la búsqueda de rutas se mantenga óptima y actualizada en todo momento.

El algoritmo recibe como entrada un nodo v desde el cual debe recalcular las rutas. Se asume que el grafo ha sufrido cambios en los pesos de las aristas, por lo que las estructuras internas del algoritmo deben reinicializarse. Para lograr esto, el método limpia la cola de prioridad y los conjuntos de caminos eficientes y no eficientes, eliminando cualquier dato obsoleto que pueda haber sido afectado por los cambios en los pesos de las aristas.

Tras esta limpieza, el algoritmo ejecuta nuevamente el método `run`, que se encarga de volver a calcular todas las rutas eficientes a partir del nuevo nodo fuente. De esta manera, se garantiza que el algoritmo sigue operando con información actualizada y que las rutas generadas continúan siendo óptimas bajo las nuevas condiciones del grafo.

Al permitir la actualización incremental de rutas sin necesidad de recalcular todo el grafo desde el nodo de origen inicial, se logra una mejora significativa en la eficiencia computacional del sistema.

Algorithm 3: Algoritmo `recalculate`

Input: $G(V, E)$ Grafo dirigido con múltiples dimensiones de costo
Input: v Nodo desde el cual se debe recalcular la búsqueda
Output: Actualización de los caminos eficientes después de cambios en el grafo
Actualizar el nodo fuente: $s \leftarrow v$;
`// Reiniciar estructuras afectadas por los cambios en los pesos`;
Vaciar priority_queue;
Vaciar pareto_front;
Vaciar non_queue_paths;
`// Ejecutar nuevamente la búsqueda de caminos eficientes`;
run();

9.5. Algoritmo de Propagación - `propagate`

El método `propagate` es el encargado de extender un camino actual hacia un nodo vecino y evaluar si dicho camino sigue siendo eficiente. Su función principal es explorar nuevas posibilidades de rutas manteniendo el frente de Pareto optimizado y evitando caminos redundantes o dominados.

El algoritmo recibe como entrada un camino `current_path` y una arista `edge` que conecta el nodo actual con un nodo vecino. Primero, se genera una nueva ruta extendiendo el camino actual a través de la arista proporcionada. Esto se realiza mediante la función `extend_path`, que calcula los nuevos costos acumulativos y los costos reducidos del camino extendido.

Una vez generado el nuevo camino, se aplica un mecanismo de poda basado en dominancia. Si el nuevo camino está dominado por la cota de dominancia o por otros caminos eficientes previamente almacenados en el frente de Pareto, se descarta automáticamente. Esto evita mantener rutas innecesarias en la cola de prioridad y mejora la eficiencia del algoritmo.

Si el camino no es descartado, se verifica si existe ya un camino hacia el mismo nodo en la cola de prioridad. Si dicho camino existe y el nuevo es mejor, se actualiza el camino en la cola, reorganizándola para preservar su orden óptimo. En caso contrario, el nuevo camino es añadido a la cola de prioridad para ser procesado en iteraciones futuras.

El método `propagate` permite que el algoritmo encuentre múltiples rutas óptimas sin evaluar caminos innecesarios. Su combinación con la estructura de datos de Pareto y la cola de prioridad garantiza que el algoritmo sea escalable incluso en entornos dinámicos

y de gran tamaño.

Algorithm 4: Algoritmo `propagate`

Input: *current_path* Camino actual representado por {path, cost, reduced_cost}
Input: $edge = (u, v)$ Arista que conecta el nodo actual con su vecino
Output: El camino propagado se añade a la cola de prioridad si es eficiente
`// Extender el camino actual utilizando la arista` (u, v);
$new\_path \leftarrow$ extend_path($current\_path, edge$);
$reduced\_cost \leftarrow new\_path.reduced\_cost$;
`// Verificar si el camino es dominado por la cota de dominancia o el frente de Pareto`;
if $dominates(dominance\_bound, reduced\_cost)$ **then**
  return `// Se descarta el camino por estar dominado`
end
foreach $permanent \in pareto\_front[v]$ **do**
  if $dominates(permanent.reduced\_cost, reduced\_cost)$ **then**
    return `// Se descarta el camino`
  end
end
`// Verificar si ya existe un camino hacia` v `en la cola de prioridad`;
foreach $item \in priority\_queue$ **do**
  $existing\_path \leftarrow item.path\_data$;
  if $existing\_path.path[-1] == v$ **then**
    if $dominates(reduced\_cost, existing\_path.reduced\_cost)$ **then**
      $existing\_path.reduced\_cost \leftarrow reduced\_cost$;
      $existing\_path.path \leftarrow new\_path.path$;
      $existing\_path.cost \leftarrow new\_path.cost$;
      heapify($priority\_queue$);
    end
    return `// Ya existía un camino, se actualizó o se descarta el nuevo`
  end
end
`// Si no hay conflicto, añadir el nuevo camino a la cola de prioridad`;
$new\_item \leftarrow$ PriorityQueueItem($reduced\_cost, new\_path$);
heappush($priority\_queue, new\_item$);

9.6. Registro y Predicción de Cambios en los Pesos de las Aristas

El algoritmo implementado en la clase `DynamicTMDA` incorpora un mecanismo de adaptación a entornos dinámicos a través del registro y la predicción de cambios en los pesos de las aristas del grafo. Para ello, se utilizan dos funciones clave: `save_future_changes` y `predict_future_changes`. La primera función, `save_future_changes`, se encarga de almacenar los cambios observados en los costos de las aristas en una estructura de datos denominada `weight_history`. Esta estructura mantiene un historial de valores recientes para cada dimensión de costo, como el tiempo, la distancia y la seguridad, lo que permite disponer de información sobre la evolución de los pesos a lo largo del tiempo. Esta función se ejecuta inmediatamente después de la actualización de pesos en el méto-

do `update_weights`, asegurando así que los datos almacenados reflejen con precisión las modificaciones más recientes en el grafo.

Por otro lado, la función `predict_future_changes` analiza el historial de cambios registrados y, cuando se dispone de suficientes datos, aplica regresión lineal para estimar la tendencia futura de los costos de cada arista. Si la predicción indica un aumento en el costo de una arista, el algoritmo ajusta su ponderación penalizándola en la selección de rutas. Este ajuste se incorpora dentro del método `select_best_path`, donde cada camino recibe un factor de corrección basado en la proyección de los cambios futuros. De esta manera, el algoritmo no solo toma en cuenta el estado actual del grafo, sino que también anticipa posibles modificaciones y adapta su comportamiento en consecuencia.

Gracias a la combinación de estos dos mecanismos, el algoritmo logra aprender de los cambios históricos en el grafo y ajustar su toma de decisiones de manera proactiva. Esto le permite evitar rutas que probablemente se vuelvan más costosas en el futuro y seleccionar caminos óptimos a largo plazo en lugar de simplemente considerar el costo en el instante actual. Como resultado, el algoritmo se vuelve más robusto y eficiente en entornos dinámicos, donde las condiciones pueden cambiar de manera impredecible. La integración de este modelo de predicción mejora significativamente la calidad de las soluciones obtenidas, permitiendo un mejor equilibrio entre eficiencia y adaptabilidad en la búsqueda de rutas óptimas.

10. Implementación y Tecnologías

La implementación del proyecto se ha realizado en Python, además, se han utilizado bibliotecas especializadas NetworkX y Matplotlib que facilitan el desarrollo de algoritmos de grafos y visualización de resultados. En cuanto a la estructura del código fuente, se ha adoptado un enfoque modularizado, que divide el proyecto en diversas clases y funciones, cada una encargada de una parte específica del proceso.

La eficiencia y la optimización del código han sido una prioridad en esta implementación. Debido a la naturaleza dinámica del problema, se ha utilizado un enfoque de actualización incremental, en el que se recalculan solo las rutas afectadas por los cambios en los pesos. Este mecanismo de propagación de cambios se complementa con el uso de colas de prioridad y listas de aristas activas, que permiten gestionar los cambios en el grafo. La combinación de estas técnicas contribuye a la reducción de la complejidad computacional, evitando la sobrecarga de cálculos innecesarios y garantizando un rendimiento adecuado incluso en entornos con múltiples cambios dinámicos.

Además, se han considerado técnicas de paralelización en la evaluación de los caminos, lo que permite distribuir el procesamiento de las rutas de manera más eficiente y aprovechar los recursos de hardware disponibles. La implementación de estas técnicas de paralelización asegura que el sistema pueda escalar adecuadamente a medida que aumenta el tamaño del grafo o el número de criterios a optimizar.

10.1. Explicación del Diseño del Algoritmo: Clases `DynamicTMDA`, `DynamicEdgeUpdater` y `DynamicGraph`

El diseño del sistema de optimización de rutas seguras en entornos dinámicos se basa en la extensión y adaptación del algoritmo *Threshold Multiobjective Dijkstra Algorithm* (T-MDA). El sistema está compuesto por tres clases principales: `DynamicTMDA`,

`DynamicEdgeUpdater`, y `DynamicGraph`. Cada una de estas clases tiene un papel crucial en la implementación del algoritmo y su adaptación a entornos dinámicos, donde los pesos de las aristas cambian en tiempo real. A continuación, se proporciona una explicación detallada de cada clase y su interacción dentro del sistema.

Clase `DynamicTMDA`: Implementación del Algoritmo Principal

La clase `DynamicTMDA` es la implementación principal del algoritmo de optimización de rutas. Esta clase extiende el algoritmo T-MDA original, adaptándolo para trabajar con grafos dinámicos, donde los costos de las aristas pueden cambiar en tiempo real debido a factores externos como el tráfico, las condiciones climáticas o incidentes imprevistos.

El papel principal de la clase `DynamicTMDA` es gestionar la búsqueda de rutas multiobjetivo en el grafo, considerando los tres criterios principales: tiempo, distancia y seguridad. Utiliza el enfoque de consultas léxicas para obtener heurísticas que guían la búsqueda de rutas en función de estos criterios. Además, la clase implementa una estrategia de poda basada en cotas de dominancia, lo que permite eliminar rutas ineficientes antes de ser exploradas completamente, reduciendo así la complejidad computacional.

Una característica clave de `DynamicTMDA` es su capacidad de adaptación dinámica. La clase es capaz de actualizar las rutas existentes sin necesidad de recalcular todas las soluciones desde cero, lo que es crucial en entornos dinámicos donde las condiciones cambian continuamente. Para lograr esto, la clase integra el mecanismo de actualización incremental, donde solo las rutas afectadas por los cambios en los pesos de las aristas son reevaluadas.

Clase `DynamicEdgeUpdater`: Actualización Dinámica de Pesos

La clase `DynamicEdgeUpdater` es responsable de gestionar la actualización dinámica de los pesos de las aristas en el grafo. Este componente permite que el algoritmo responda a cambios en tiempo real, como la congestión del tráfico o la aparición de nuevos riesgos en determinadas zonas.

El objetivo principal de `DynamicEdgeUpdater` es proporcionar un mecanismo eficiente para modificar los pesos de las aristas en función de los eventos del entorno, como el tráfico, las condiciones meteorológicas o los incidentes de seguridad. Esta clase está diseñada para actualizar solo aquellas aristas que han sido modificadas, evitando así la necesidad de recalcular los costos de todas las rutas en el grafo. Esto reduce la carga computacional y mejora la eficiencia del sistema, especialmente cuando los cambios son locales y afectan solo a una pequeña parte del grafo.

Para actualizar los pesos, `DynamicEdgeUpdater` recibe la información externa X_t (por ejemplo, datos de tráfico o clima) y aplica la función de actualización $f(w(e_{ij}, t), X_t)$ a las aristas relevantes. El resultado es la modificación de los costos de las aristas en tiempo real, lo que permite que el sistema de optimización de rutas se ajuste de manera continua a las condiciones cambiantes.

DynamicGraph

: Representación del Grafo Dinámico

La clase `DynamicGraph` es responsable de la representación del grafo dinámico, donde se almacenan los nodos y las aristas. Cada arista en el grafo está asociada con tres

atributos: tiempo de viaje, distancia y seguridad, que se actualizan en tiempo real a través de la clase `DynamicEdgeUpdater`.

El grafo está diseñado para ser flexible, permitiendo tanto aristas unidireccionales como bidireccionales, dependiendo de las características de la red de transporte. Además, `DynamicGraph` proporciona métodos para actualizar los pesos de las aristas, añadir nuevas aristas y consultar los costos de las aristas. Estos métodos son utilizados por la clase `DynamicTMDA` para realizar la búsqueda de caminos multiobjetivo y gestionar las actualizaciones dinámicas de los pesos.

El uso de la clase `DynamicGraph` permite que el grafo refleje de manera precisa las condiciones cambiantes del entorno. Por ejemplo, si un camino se vuelve más peligroso debido a un accidente, el puntaje de seguridad de la arista correspondiente se incrementará, lo que influirá directamente en la selección de rutas por parte de `DynamicTMDA`. De esta forma, la clase asegura que el grafo sea siempre una representación precisa del entorno de transporte en tiempo real.

Interacción entre las Clases

La interacción entre estas tres clases permite que el sistema funcione de manera eficiente y flexible. `DynamicGraph` proporciona la estructura básica del grafo y gestiona los datos de las aristas, mientras que `DynamicEdgeUpdater` se encarga de la actualización dinámica de los pesos de las aristas en función de los eventos del entorno. Finalmente, `DynamicTMDA` ejecuta el algoritmo principal de búsqueda de rutas, utilizando la estructura del grafo y los cambios dinámicos en los pesos para encontrar el conjunto de rutas óptimas bajo múltiples criterios.

La modularización de estas funciones en tres clases independientes permite que el sistema sea fácilmente escalable y adaptable a nuevos requerimientos. Además, al separar las responsabilidades de cada clase, el código es más mantenible y flexible, lo que facilita la implementación de futuras mejoras, como la incorporación de modelos predictivos para anticipar cambios en el entorno o la optimización de rutas en función de nuevas preferencias del usuario.

11. Evaluación Experimental y Visualización de Resultados

Para evaluar el rendimiento y la eficiencia del algoritmo **DynamicTMDA**, se ha diseñado un conjunto de pruebas de estrés que permiten analizar su comportamiento bajo diferentes condiciones estructurales del grafo y variaciones dinámicas en los pesos de las aristas. Estas pruebas tienen como objetivo medir la capacidad del algoritmo para adaptarse a cambios en la topología de la red, la eficiencia computacional en términos de tiempo de ejecución y consumo de memoria, así como el impacto en el uso del CPU durante su ejecución.

El proceso de validación se inicia con la generación de grafos de distintas características estructurales. Para ello, se emplean tres modelos bien establecidos en la teoría de grafos: *Barabasi-Albert*, que modela redes con crecimiento preferencial; *Grid-2D*, que representa sistemas de navegación sobre estructuras espaciales discretizadas; y *Erdos-Renyi*, en el que las conexiones entre nodos se establecen de manera aleatoria con una probabilidad definida.

El modelo **Barabasi-Albert** genera grafos de tipo libre de escala, donde los nodos se agregan de manera progresiva y las nuevas conexiones siguen un principio de preferencia de enlace. En este tipo de grafo, algunos nodos actúan como *hubs* altamente conectados, mientras que la mayoría de los nodos tienen un grado bajo. Esta distribución heterogénea de conexiones implica que las rutas óptimas suelen depender de un subconjunto reducido de nodos altamente centrales, lo que puede generar congestión en estos puntos críticos. Un desafío importante en este tipo de grafo es que pequeñas modificaciones en los pesos de las aristas que involucran estos *hubs* pueden provocar cambios significativos en la estructura de las rutas eficientes, lo que obliga al algoritmo a recalcular las soluciones con frecuencia. Sin embargo, su estructura jerárquica también permite que el algoritmo encuentre caminos óptimos con relativa rapidez, ya que la cantidad de rutas alternativas entre pares de nodos es menor en comparación con otros modelos más homogéneos.

El modelo **Grid-2D** representa un grafo en forma de cuadrícula bidimensional, donde cada nodo está conectado a sus vecinos inmediatos en una malla estructurada. Este modelo es ampliamente utilizado para simular entornos de navegación y planificación de rutas en espacios físicos, como redes urbanas o sistemas de tránsito. A diferencia de Barabasi-Albert, la distribución de grados en Grid-2D es homogénea, ya que cada nodo interno tiene exactamente cuatro conexiones, mientras que los nodos en los bordes tienen tres o dos conexiones. La principal dificultad en este tipo de grafo radica en la redundancia de caminos alternativos, lo que puede incrementar la cantidad de soluciones Pareto-óptimas y aumentar la complejidad del procesamiento. Además, los cambios en los pesos de las aristas tienden a afectar las rutas locales de manera gradual en lugar de provocar modificaciones drásticas en toda la estructura de caminos, lo que puede ser ventajoso para la estabilidad del algoritmo, pero también desafiante en términos de eficiencia computacional, ya que debe gestionar un gran número de soluciones equivalentes.

El modelo **Erdos-Renyi** genera grafos aleatorios en los que cada par de nodos tiene una probabilidad fija de estar conectado. Esto da lugar a estructuras altamente variables en términos de densidad y distribución de grados, con algunas instancias generando grafos altamente conectados y otras produciendo estructuras fragmentadas con componentes desconectados. Esta variabilidad introduce un desafío significativo para el algoritmo, ya que en algunos casos las rutas óptimas pueden ser escasas y difíciles de encontrar, mientras que en otros pueden existir múltiples caminos con costos similares. En particular, cuando el grafo es disperso, el algoritmo puede verse obligado a explorar grandes porciones del espacio de búsqueda antes de encontrar rutas viables. Por otro lado, en grafos densos, la cantidad de combinaciones de caminos posibles puede ser considerablemente alta, aumentando la cantidad de soluciones en el frente de Pareto y, en consecuencia, el consumo de memoria y tiempo de cómputo.

Cada uno de estos modelos de grafo ofrece un conjunto distinto de características y desafíos que influyen en la efectividad del algoritmo **DynamicTMDA**. El modelo *Barabasi-Albert* enfatiza la importancia de los *hubs* en la conectividad de la red y plantea el reto de gestionar cambios en los nodos centrales. El modelo *Grid-2D* proporciona una estructura altamente regular que permite evaluar la eficiencia en entornos con múltiples caminos equivalentes. Finalmente, el modelo *Erdos-Renyi* introduce incertidumbre en la estructura del grafo, obligando al algoritmo a manejar tanto escenarios de baja conectividad como configuraciones altamente densas. Evaluar el algoritmo en estos tres tipos de redes permite obtener una comprensión más completa de su rendimiento y adaptabilidad en distintos contextos de aplicación.

Cada uno de estos modelos permite evaluar el algoritmo en diferentes escenarios,

desde redes con propiedades de mundo pequeño hasta estructuras espaciales regulares y sistemas altamente aleatorios.

Los experimentos se realizan sobre grafos con tamaños que oscilan entre 100 y 5000 nodos, lo que permite observar el impacto del escalamiento en el rendimiento del algoritmo. Además, para simular la naturaleza dinámica de los entornos en los que **DynamicTM-DA** será aplicado, se modifican los pesos de las aristas en distintos grados. Se consideran cinco niveles de alteración del grafo, variando entre el 10 % y el 90 % de las aristas, lo que permite evaluar la sensibilidad del algoritmo a cambios abruptos en la red y su capacidad de respuesta ante modificaciones en tiempo real.

Cada prueba se ejecuta en múltiples iteraciones para garantizar la estabilidad de los resultados y minimizar el efecto de posibles fluctuaciones en la ejecución. Antes de la ejecución del algoritmo, se mide el uso del CPU para obtener un punto de referencia sobre el impacto computacional del procesamiento de las estructuras de datos y la exploración de caminos eficientes. Posteriormente, se mide el tiempo de ejecución del algoritmo, que se inicia en un nodo origen y busca los caminos eficientes hacia un nodo destino, considerando criterios de optimización multiobjetivo como el tiempo, la distancia y la seguridad. Durante la ejecución, se registran las modificaciones realizadas en las aristas del grafo, cuantificando el número total de cambios procesados.

Finalizada la ejecución, se recopilan métricas clave para el análisis del desempeño del algoritmo. Se mide la memoria utilizada en tiempo real mediante la herramienta *tracemalloc*, lo que permite conocer tanto el consumo actual como el pico máximo de memoria durante la ejecución. Asimismo, se realiza una segunda medición del uso del CPU para evaluar la carga computacional impuesta por el algoritmo. Además de estos parámetros de rendimiento, se almacenan los caminos obtenidos antes y después de la ejecución del algoritmo, permitiendo evaluar la estabilidad de las soluciones generadas y la efectividad del proceso de optimización.

Los resultados de cada prueba son impresos en consola y almacenados en un archivo CSV, lo que facilita el análisis posterior y la comparación entre distintos escenarios de prueba. Esta metodología de evaluación permite obtener una visión integral del comportamiento del algoritmo, identificando sus fortalezas y posibles limitaciones en términos de escalabilidad, eficiencia y capacidad de respuesta en entornos dinámicos.

11.1. Evaluación de Rendimiento del Algoritmo

En esta sección se presentan los resultados obtenidos a partir de la ejecución del algoritmo sobre diferentes tipos de grafos y configuraciones. A continuación, se incluyen las figuras que ilustran los principales aspectos analizados.

Análisis del Tiempo de Ejecución

El tiempo de ejecución del algoritmo varía considerablemente según el tipo de grafo y el tamaño del problema. En grafos con estructuras jerárquicas como *Barabasi-Albert*, el algoritmo presenta tiempos de ejecución significativamente menores en comparación con los grafos de tipo *Erdos-Renyi* y *Grid-2D*. Esto se debe a que la estructura jerárquica de *Barabasi-Albert* permite reducir el espacio de búsqueda, minimizando la cantidad de caminos evaluados. En cambio, en grafos aleatorios como *Erdos-Renyi* y en estructuras de cuadrícula, la cantidad de caminos no dominados crece exponencialmente, lo que incrementa considerablemente la carga computacional.

Cuadro 1: Tiempo de ejecución del algoritmo en segundos según el tipo de grafo y porcentaje de aristas modificadas.

Tipo de Grafo	Tamaño del Grafo	10 %	30 %	50 %	70 %	90 %
Barabasi-Albert	100	0.17	0.09	0.1	0.02	0.04
	250	0.21	0.09	0.02	0.13	0.15
	500	0.14	0.49	0.59	0.2	0.07
	1000	0.94	0.17	0.13	0.26	0.87
Erdos-Renyi	100	0.06	0.09	0.46	0.33	0.25
	250	0.12	3.83	0.56	3.23	0.4
	500	7.61	8.53	22.89	3.23	8.75
	1000	58.55	157.09	83.3	161.89	121.32
Grid-2D	100	4.29	4.02	4.18	4.22	4.01
	250	28.3	28.87	33.86	30.22	29.37
	500	138.18	168.2	163.55	182.32	185
	1000	1426.67	1346.62	1321.07	1006.65	1125.23

Se observa que los tiempos de ejecución en *Grid-2D* aumentan de manera exponencial con el tamaño del grafo, lo que indica que el número de soluciones no dominadas en este tipo de estructura crece rápidamente. Esto sugiere que el algoritmo necesita una estrategia más agresiva de poda de caminos en grafos estructurados como cuadrículas.

Análisis de la Cantidad de Recalculaciones

El número de recalculaciones necesarias para encontrar rutas óptimas depende de la cantidad de aristas modificadas y de la estructura del grafo. En grafos de tipo *Barabasi-Albert* y *Erdos-Renyi*, el número de recalculaciones es bajo, ya que los cambios en los pesos de las aristas no afectan significativamente el conjunto de soluciones de Pareto. Sin embargo, en *Grid-2D*, las modificaciones en las aristas suelen generar múltiples caminos alternativos, lo que incrementa la necesidad de recalculaciones.

Cuadro 2: Cantidad de recalculaciones necesarias según el tipo de grafo y porcentaje de aristas modificadas.

Tipo de Grafo	Tamaño del Grafo	10 %	30 %	50 %	90 %
Barabasi-Albert	100	7	6	5	5
	250	4	4	3	4
	500	5	7	9	4
	1000	5	4	4	7
Erdos-Renyi	100	5	5	6	6
	250	3	5	5	4
	500	5	5	4	5
	1000	5	5	5	5
Grid-2D	100	36	38	38	36
	250	58	56	62	56
	500	86	88	90	92
	1000	126	120	120	126

En los casos donde la cantidad de recalculaciones es alta, se sugiere optimizar la estrategia de actualización del frente de Pareto, evitando la recomputación completa cuando solo un subconjunto de caminos se ve afectado.

Cuadro 3: Comparación Unificada por Tipo de Grafo y Porcentaje de Aristas Modificadas

Tipo de Grafo	**Aristas Modificadas**	**Tiempo Ejecución**	**Aristas Modificadas**	**Recalculos**	**Memoria**
Barabasi-Albert	10 %	0.36	4.38	5.25	1.57
	30 %	0.21	3.12	5.25	1.59
	50 %	0.21	3.88	5.25	1.58
	70 %	0.15	4.25	5.5	1.54
	90 %	0.28	3.12	5.0	1.55
Grid-2D	10 %	399.36	462.25	76.5	7.8
	30 %	386.93	446.5	75.5	7.31
	50 %	380.67	454.62	77.5	7.71
	70 %	305.85	434.38	75.0	6.01
	90 %	335.9	455.38	77.5	7.21
Erdos-Renyi	10 %	16.59	2.12	4.5	25.45
	30 %	42.38	3.25	5.0	16.78
	50 %	26.8	3.5	5.0	15.59
	70 %	42.17	2.75	4.25	16.06
	90 %	32.68	3.88	5.0	16.74

Cantidad de Aristas Modificadas

La siguiente tabla muestra la cantidad de aristas modificadas durante la ejecución del algoritmo en función del tipo de grafo, su tamaño y el porcentaje de modificaciones aplicado. Se puede observar que los grafos tipo **Grid-2D** presentan un número significativamente mayor de aristas modificadas en comparación con **Erdos-Renyi** y **Barabasi-Albert**, lo que sugiere una mayor sensibilidad a los cambios estructurales.

Cuadro 4: Cantidad de Aristas Modificadas por Tipo de Grafo y Tamaño

Tipo de Grafo	**Tamaño del Grafo**	**10 %**	**30 %**	**50 %**	**70 %**	**90 %**
Barabasi-Albert	100	7.5	4.5	4	2	4
	250	3	2	1	6	2
	500	3.5	5	7	3.5	2
	1000	3.5	1	3.5	5.5	4.5
Erdos-Renyi	100	2.5	3.5	3.5	3.5	5.5
	250	0.5	2.5	3.5	2.5	2
	500	2	4	3.5	2.5	4
	1000	3.5	3	3.5	2.5	4
Grid-2D	100	93.5	100	102.5	95	91
	250	221.5	235	245.5	219	208.5
	500	510	505.5	514	470	504.5
	1000	1024	945.5	956.5	953.5	1017.5

Pico de Memoria Utilizada (MB)

A continuación, se presenta la cantidad máxima de memoria utilizada durante la ejecución del algoritmo en diferentes tipos de grafos. Se puede notar que los grafos **Grid-2D** y **Erdos-Renyi** requieren significativamente más memoria que los **Barabasi-Albert**, lo que sugiere que su estructura influye en el consumo de memoria.

Cuadro 5: Pico de Memoria (MB) por Tipo de Grafo y Tamaño

Tipo de Grafo	**Tamaño del Grafo**	**10 %**	**30 %**	**50 %**	**70 %**	**90 %**
Barabasi-Albert	100	0.44	0.42	0.42	0.39	0.4
	250	0.89	0.83	0.84	0.84	0.85
	500	1.58	1.69	1.71	1.62	1.56
	1000	3.37	3.41	3.35	3.32	3.4
Erdos-Renyi	100	39.76	0.56	0.66	0.66	0.58
	250	2.97	3.45	3.09	3.53	3.03
	500	11.9	11.83	13.32	11.53	11.88
	1000	47.18	51.29	45.29	48.5	51.48
Grid-2D	100	0.84	0.72	0.69	0.82	0.72
	250	2.39	2.24	2.51	2.1	2.59
	500	5.35	5.59	5.79	6.66	6.72
	1000	22.62	20.7	21.87	14.47	18.82

Análisis del Uso de Memoria

El uso de memoria del algoritmo está fuertemente correlacionado con la cantidad de caminos almacenados en el frente de Pareto. En grafos como *Erdos-Renyi*, donde la estructura es más caótica, se observa un consumo de memoria mayor debido a la necesidad de mantener múltiples rutas óptimas en memoria. Por otro lado, en *Barabasi-Albert*, el consumo de memoria es significativamente menor, ya que la estructura jerárquica del grafo reduce la cantidad de rutas alternativas.

Cuadro 6: Pico de memoria alcanzado en MB según el tipo de grafo y porcentaje de aristas modificadas.

Tipo de Grafo	Tamaño del Grafo	10 %	50 %	90 %
Barabasi-Albert	100	0.44	0.42	0.4
Erdos-Renyi	1000	47.18	45.29	51.48
Grid-2D	1000	22.62	21.87	18.82

Análisis Visual de los Resultados

Las tablas anteriores presentan un resumen detallado de los valores obtenidos en las pruebas realizadas sobre distintos tipos de grafos. Para complementar esta información, a continuación se muestran una serie de gráficos que permiten visualizar de manera más clara las tendencias y patrones en los datos.

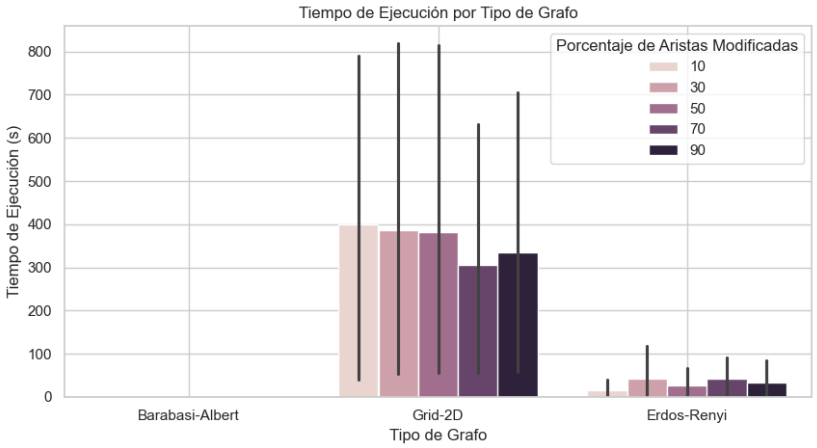

Figura 1: Comparación del tiempo de ejecución del algoritmo en distintos tipos de grafos y porcentajes de aristas modificadas. Se observa cómo el tiempo varía dependiendo de la estructura del grafo y las modificaciones aplicadas.

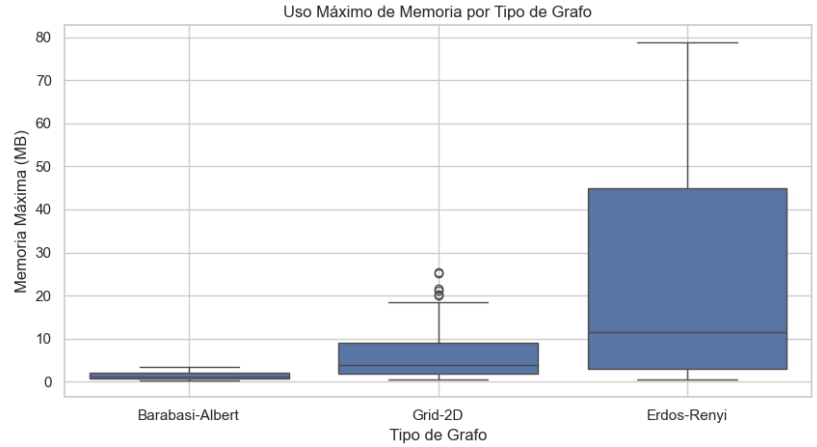

Figura 2: Análisis del uso máximo de memoria en función del tipo de grafo. Se observa que algunos grafos requieren mayor cantidad de memoria debido a la estructura de sus caminos óptimos.

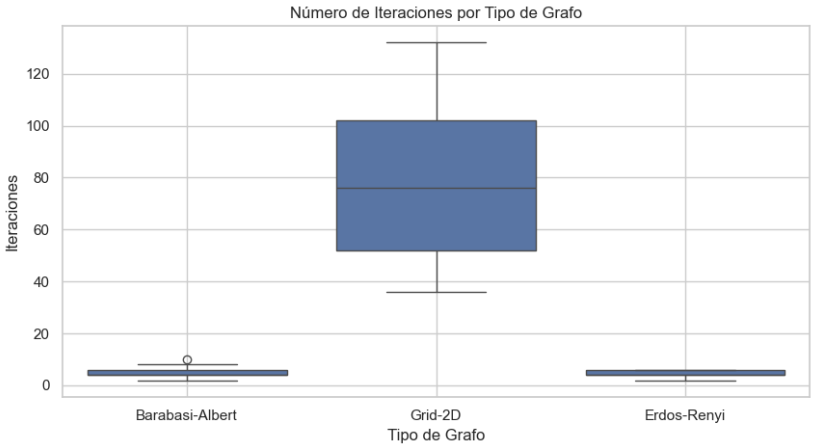

Figura 3: Número promedio de recalculos realizados en cada tipo de grafo. Un mayor número de recalculos indica una mayor cantidad de cambios en los pesos de las aristas durante la ejecución del algoritmo.

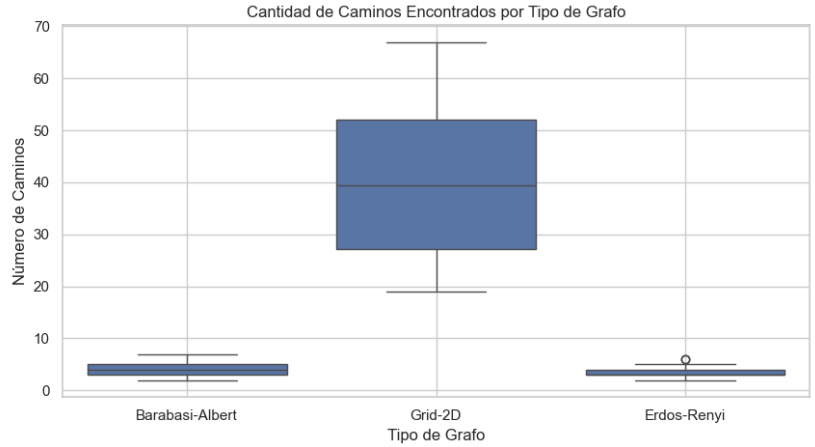

Figura 4: Número de caminos eficientes encontrados en cada tipo de grafo. Se analiza cómo la estructura del grafo afecta la cantidad de soluciones óptimas obtenidas.

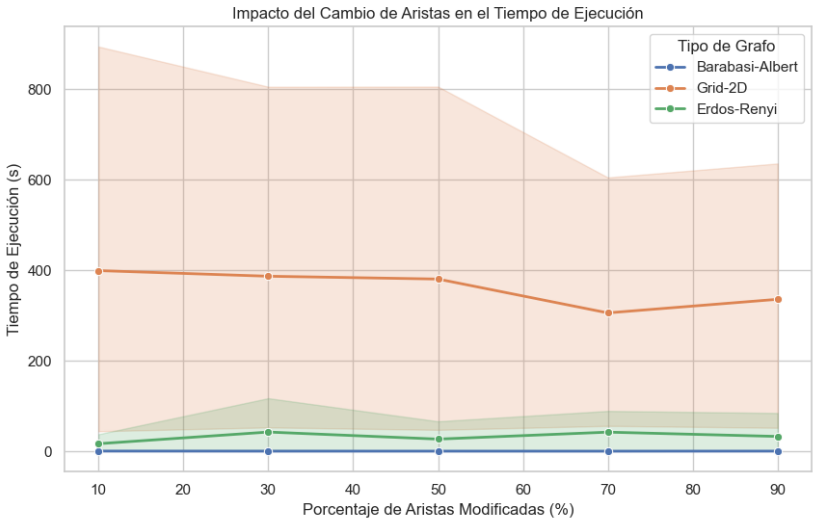

Figura 5: Relación entre el porcentaje de aristas modificadas y el tiempo de ejecución del algoritmo en distintos tipos de grafos.

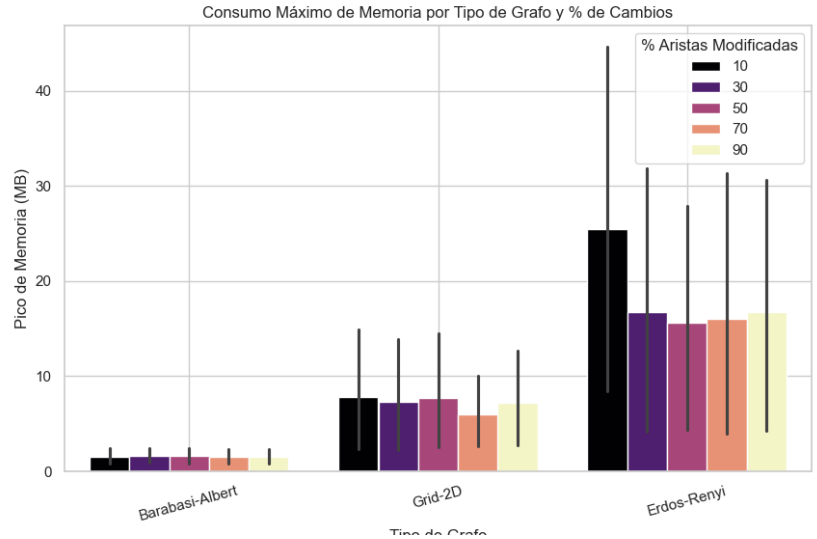

Figura 6: Consumo máximo de memoria en función del tipo de grafo y el porcentaje de cambios en las aristas.

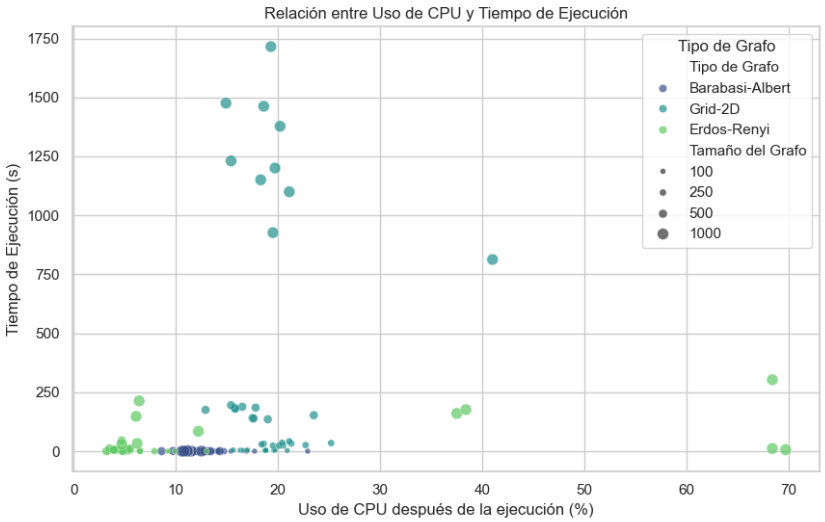

Figura 7: Comparación del uso de CPU antes y después de la ejecución del algoritmo en diferentes tipos de grafos.

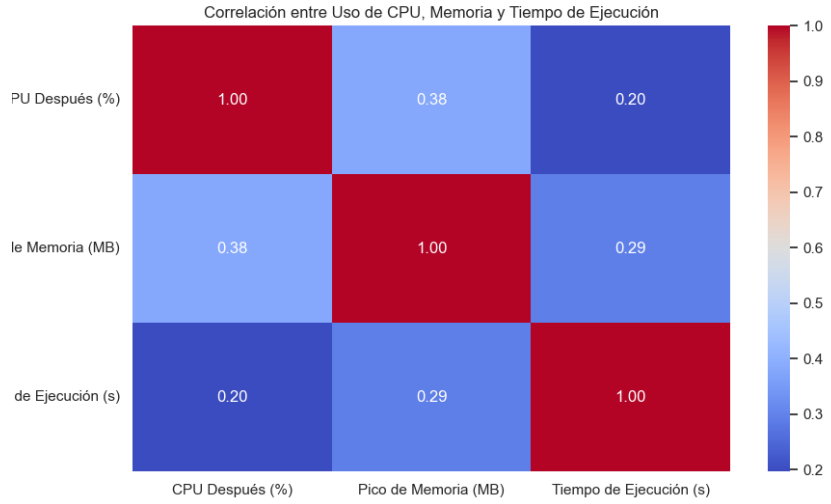

Figura 8: Análisis de la correlación entre el uso de CPU, el consumo de memoria y el tiempo de ejecución del algoritmo.

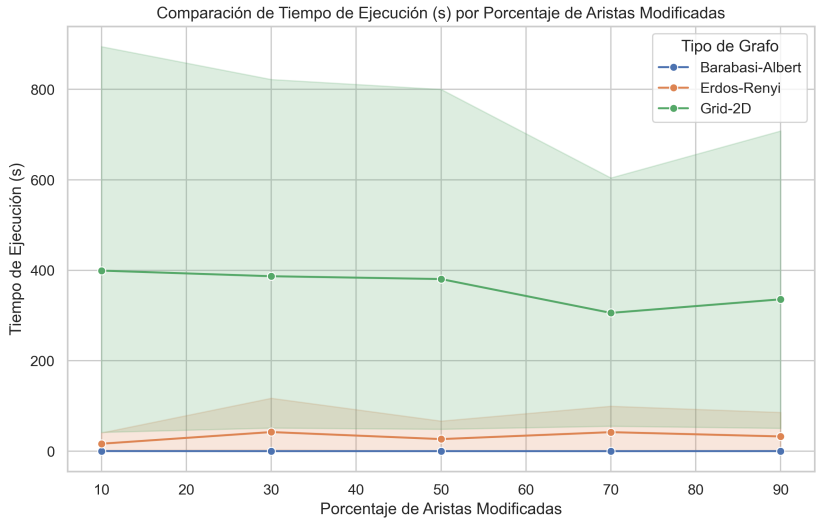

Figura 9: Tiempo de ejecución en función del tamaño del grafo para cada tipo de grafo. Se observa cómo el tiempo crece con el número de nodos, con diferencias entre los distintos modelos de grafos.

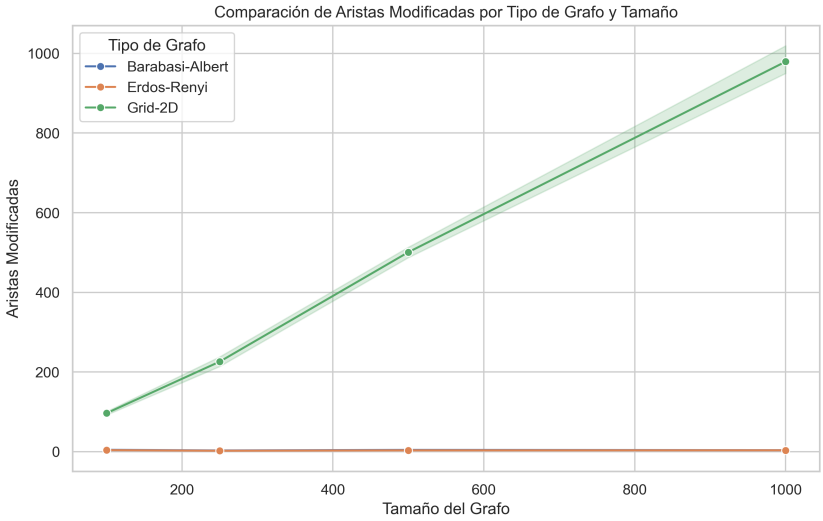

Figura 10: Cantidad de aristas modificadas en función del tamaño del grafo. Se visualiza cómo varía el impacto de los cambios estructurales en cada tipo de grafo.

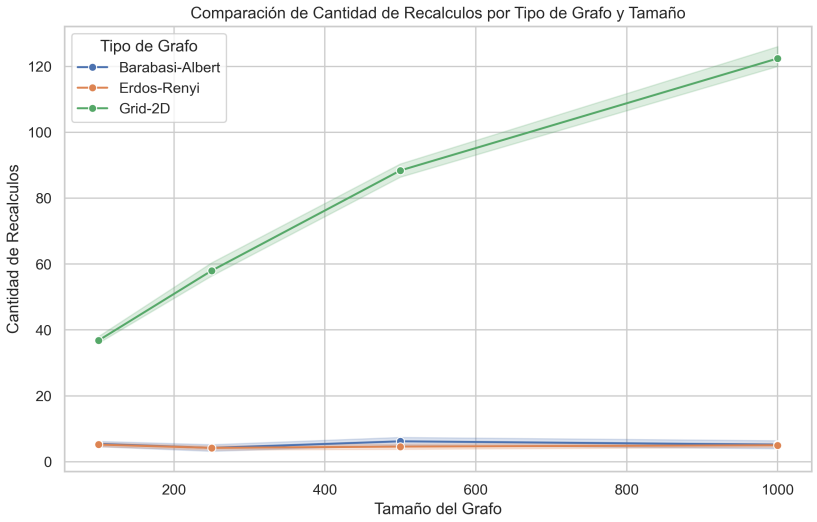

Figura 11: Número de recalculos realizados dependiendo del tamaño del grafo. Los resultados muestran que ciertos modelos de grafos requieren más ajustes que otros.

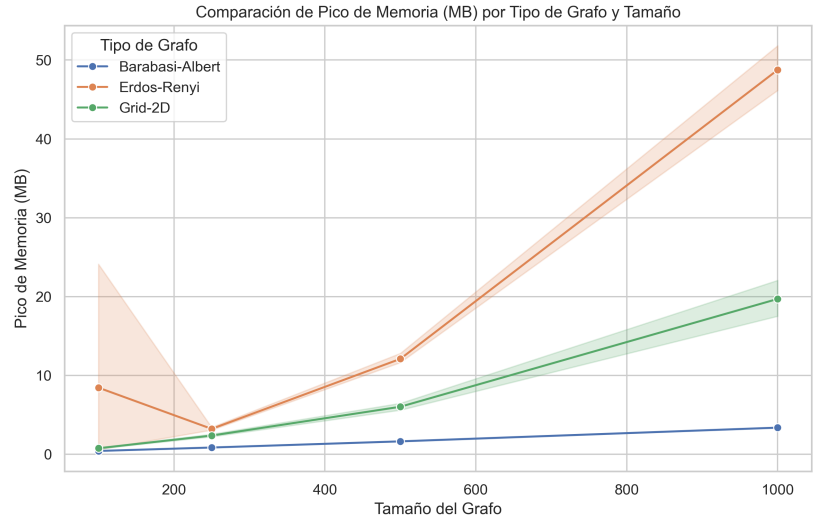

Figura 12: Consumo máximo de memoria en función del tamaño del grafo. Se evalúa el impacto de la complejidad del grafo en la utilización de memoria.

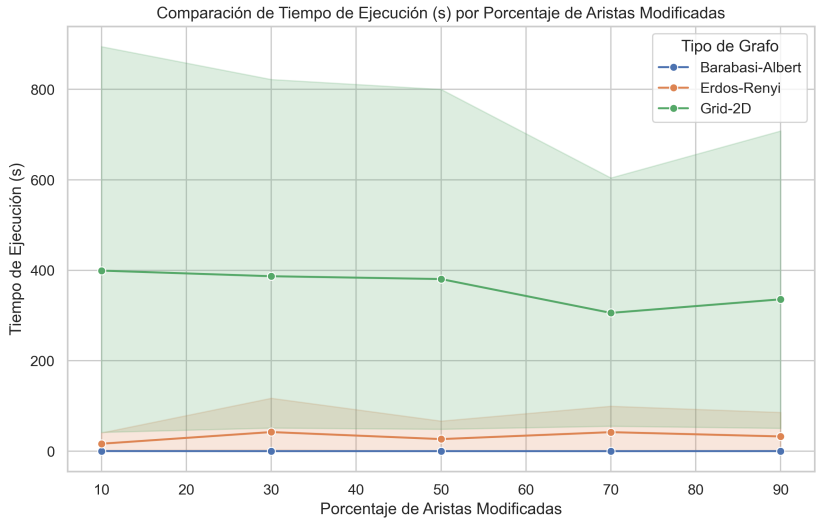

Figura 13: Impacto del porcentaje de aristas modificadas en el tiempo de ejecución del algoritmo. Se observa una tendencia creciente en ciertos tipos de grafos.

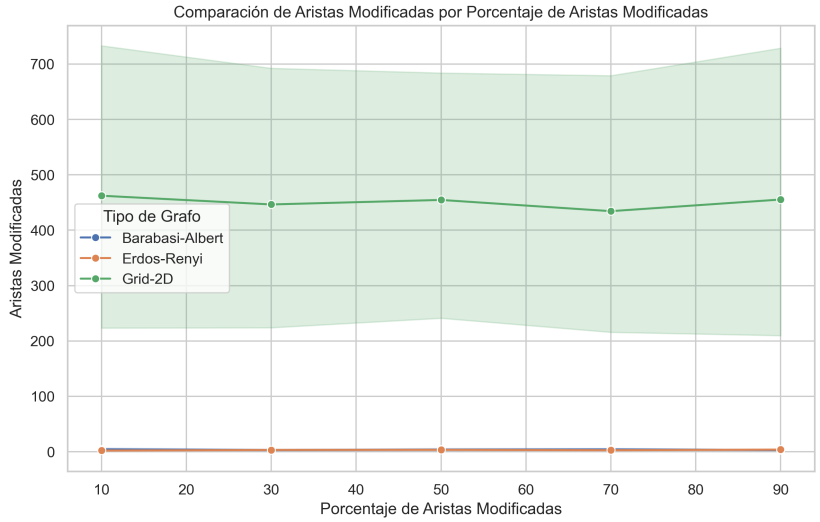

Figura 14: Cantidad de aristas modificadas según el porcentaje de cambios aplicados. Los resultados indican una relación esperada entre la cantidad de modificaciones y los valores registrados.

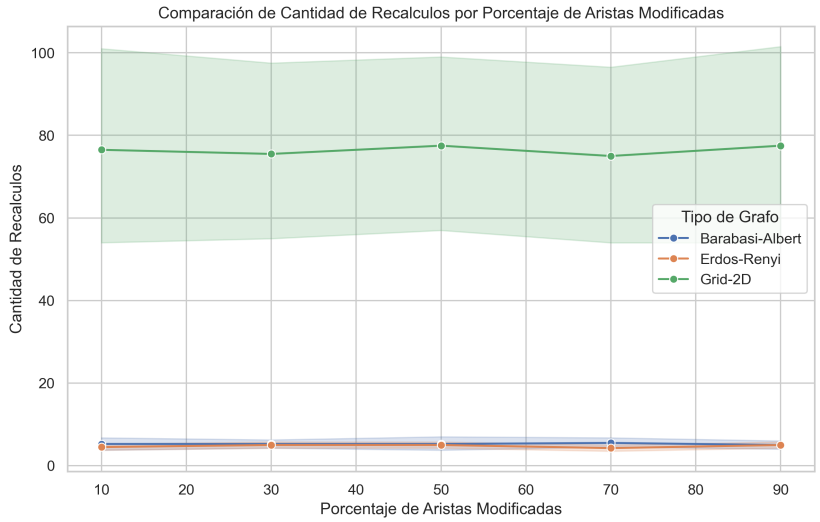

Figura 15: Número de recalculos realizados según el porcentaje de aristas modificadas. En algunos grafos, los cambios abruptos requieren más recalculaciones.

Figura 16: Consumo máximo de memoria según el porcentaje de aristas modificadas. El análisis indica que ciertos tipos de grafos generan un mayor consumo con cambios estructurales.

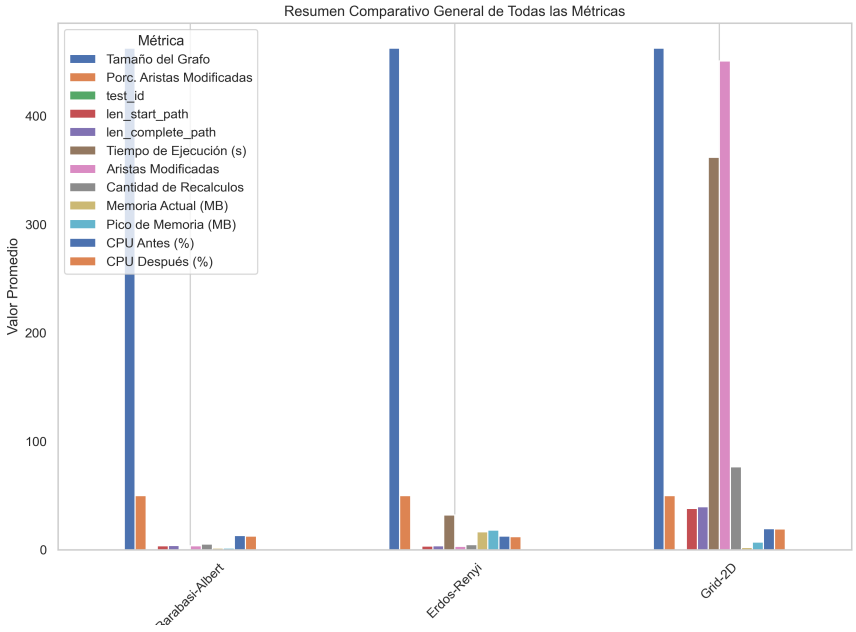

Figura 17: Resumen comparativo general de todas las métricas por tipo de grafo. Este gráfico muestra las diferencias en desempeño y consumo de recursos según la estructura del grafo.

11.2. Análisis de Resultados

Los resultados obtenidos en las pruebas de estrés realizadas sobre el algoritmo de generación de rutas seguras en entornos dinámicos permiten evaluar su desempeño en diferentes estructuras de grafos y configuraciones de actualización de aristas. Uno de los hallazgos más significativos es la diferencia en los tiempos de ejecución entre los distintos tipos de grafos. En los grafos Barabasi-Albert, caracterizados por la presencia de nodos altamente conectados, el algoritmo mantiene tiempos de ejecución reducidos incluso para grafos de gran tamaño. En contraste, en los grafos Erdos-Renyi y Grid-2D, el crecimiento en el tiempo de ejecución es mucho más pronunciado a medida que aumenta la cantidad de nodos y el porcentaje de aristas modificadas. El caso más extremo se observa en los grafos Grid-2D, donde los tiempos de ejecución alcanzan valores significativamente altos, con una ejecución máxima de 1426.67 segundos en grafos de 1000 nodos con un 10 % de aristas modificadas. Esto sugiere que la naturaleza estructurada y altamente interconectada de estos grafos genera una explosión combinatoria en el número de rutas posibles, lo que impacta de manera significativa en la carga computacional del algoritmo.

En cuanto a la cantidad de recalculaciones necesarias para mantener la actualización dinámica de rutas, se observa un patrón similar. En los grafos Barabasi-Albert, el número de recalculaciones es relativamente bajo, reflejando la estabilidad de las conexiones dominantes en este tipo de estructura. Sin embargo, en los grafos Erdos-Renyi y Grid-2D, el número de recalculaciones aumenta considerablemente debido a la mayor cantidad de caminos no dominados que deben ser evaluados y ajustados en tiempo real. En los casos de mayor tamaño en los grafos Grid-2D, las recalculaciones pueden superar las 120 iteraciones, lo que indica que el algoritmo enfrenta dificultades al gestionar cambios dinámicos en entornos altamente estructurados.

El uso de memoria es otro factor crítico en el rendimiento del algoritmo. En los grafos Barabasi-Albert, el consumo de memoria se mantiene en niveles relativamente bajos, con un pico máximo de aproximadamente 3.5 MB en las pruebas más exigentes. En contraste, en los grafos Erdos-Renyi y Grid-2D, el uso de memoria se incrementa de manera notable. En particular, los grafos Erdos-Renyi presentan picos de memoria que superan los 50 MB en configuraciones de 1000 nodos con altos niveles de modificación de aristas. Este comportamiento sugiere que la estructura aleatoria de estos grafos genera una mayor cantidad de soluciones intermedias que deben ser almacenadas y gestionadas en memoria. Por su parte, en los grafos Grid-2D, el consumo de memoria muestra un crecimiento más controlado, pero aún así presenta valores elevados debido a la gran cantidad de caminos evaluados simultáneamente.

Otro aspecto relevante en el análisis de los resultados es la cantidad de caminos encontrados en cada prueba. En los grafos Barabasi-Albert y Erdos-Renyi, el número de caminos óptimos descubiertos se mantiene en valores relativamente bajos, reflejando una menor cantidad de soluciones viables dentro del conjunto de Pareto. En contraste, en los grafos Grid-2D, el número de caminos eficientes aumenta significativamente, alcanzando valores promedio cercanos a 40 caminos en los experimentos realizados. Esta diferencia sugiere que en estructuras altamente organizadas y con alta conectividad local, el algoritmo encuentra múltiples soluciones óptimas en términos de los criterios de optimización establecidos.

Finalmente, el análisis del uso de CPU antes y después de la ejecución muestra que, en general, el algoritmo mantiene un consumo estable de recursos computacionales. En los grafos Barabasi-Albert y Erdos-Renyi, el consumo de CPU se mantiene en niveles re-

lativamente bajos, sin cambios significativos tras la ejecución del algoritmo. Sin embargo, en los grafos Grid-2D, el consumo de CPU experimenta un aumento notable, reflejando la mayor carga de procesamiento requerida para manejar la complejidad de estas estructuras. Este comportamiento sugiere que, si bien el algoritmo es eficiente en escenarios con grafos menos estructurados, su rendimiento se ve afectado de manera considerable en entornos con una organización espacial más rígida y una mayor cantidad de conexiones locales.

En conclusión, el análisis de los resultados muestra que el algoritmo es eficiente en la mayoría de los escenarios evaluados, pero su desempeño se ve afectado por la estructura del grafo y la cantidad de modificaciones dinámicas en las aristas. En grafos altamente estructurados como Grid-2D, el crecimiento exponencial del número de rutas evaluadas genera un aumento significativo en los tiempos de ejecución, el consumo de memoria y la utilización de CPU. En cambio, en grafos menos estructurados como Barabasi-Albert y Erdos-Renyi, el algoritmo logra mantener una ejecución estable con tiempos reducidos y un consumo moderado de recursos. Estos resultados resaltan la importancia de considerar la estructura del grafo al aplicar el algoritmo en escenarios reales, así como la necesidad de desarrollar estrategias adicionales de optimización para mejorar su desempeño en entornos altamente interconectados.

12. Anexos

A. Complejidad del Problema de Búsqueda Multiobjetivo

El problema de búsqueda multiobjetivo es conocido por ser un problema intratable en muchos casos, debido a que el número de soluciones no dominadas puede crecer exponencialmente en función del tamaño del grafo o de la cantidad de objetivos. La naturaleza multiobjetivo del problema introduce una complejidad significativa debido al crecimiento exponencial del número de soluciones no dominadas en función del tamaño del grafo y del número de criterios.

A diferencia del problema de caminos más cortos con un único objetivo, el MOSP no tiene una solución única, sino un conjunto de soluciones conocidas como el frente de Pareto. Este conjunto contiene todas las rutas no dominadas, donde cada ruta representa un compromiso óptimo entre los múltiples criterios. En el caso de $q = 1$, el problema se reduce al problema clásico de caminos más cortos, que puede resolverse en tiempo polinómico utilizando algoritmos como el de Dijkstra [2] ($O(n \log n + m)$, donde $n = |V|$ y $m = |E|$).

Sin embargo, cuando $q > 1$, el tamaño del frente de Pareto puede crecer exponencialmente con la profundidad de la solución o, de manera equivalente, con el número de nodos en el grafo. Hansen (1979) [4] demostró que, incluso en el caso bicriterio ($q = 2$), el número de soluciones no dominadas puede crecer exponencialmente en el peor caso. Por tanto, el problema no puede resolverse de manera eficiente en grafos grandes sin recurrir a técnicas específicas de poda y heurísticas.

La complejidad del MOSP también se ha estudiado mediante reducciones a problemas conocidos. Serafini (1986) [5] demostró que el problema de caminos más cortos con múltiples criterios puede transformarse en un problema de la mochila binaria, el cual fue identificado como NP-completo por Garey y Johnson (1979) [6]. Esta reducción establece

que el MOSP es NP-Hard, ya que implica explorar todas las combinaciones posibles de caminos para identificar aquellos que son no dominados.

En grafos con costos no negativos, la ausencia de ciclos negativos garantiza que el número de caminos eficientes es finito. Sin embargo, la cantidad de estos caminos puede ser exponencial en función del número de nodos y objetivos, lo que implica que el tiempo requerido para enumerarlos también crece exponencialmente en el peor caso.

Otro factor clave que afecta la complejidad del problema es el número de objetivos q. En espacios de estado con costos enteros acotados, Mandow y Pérez de la Cruz (2009) [8] demostraron que el número de soluciones no dominadas crece de forma polinómica con la profundidad de la solución, pero sigue siendo exponencial con respecto a q. Esto significa que, aunque el problema puede manejarse en casos con pocos objetivos, su complejidad aumenta rápidamente cuando se introducen más criterios.

En particular, para $q = 2$, las verificaciones de dominancia entre rutas pueden realizarse en tiempo constante, y muchas optimizaciones específicas, como algoritmos de poda, son aplicables. Sin embargo, para $q > 2$, estas optimizaciones pierden efectividad, y el problema se vuelve intratable en términos prácticos para grafos grandes.

Martins y Santos (1999) [7] establecieron que el problema MOSP es finito si no existen ciclos negativos en el grafo para ningún criterio. En otras palabras, si para cualquier ciclo $P_{\text{ciclo}} \in G$ y cualquier objetivo k, se cumple que $c^k(P_{\text{ciclo}}) > 0$, entonces el número de soluciones no dominadas es finito. Además, demostraron que la ausencia de ciclos negativos es una condición necesaria y suficiente para garantizar tanto la finitud como el acotamiento del problema.

La complejidad del MOSP depende de los algoritmos utilizados para resolverlo. Para casos generales, los algoritmos exactos como el T-MDA tienen una complejidad espacial y temporal que puede describirse como sigue:

$$\text{Complejidad Espacial: } O(m \cdot N_{\max}),$$

$$\text{Complejidad Temporal: } O(d \cdot N_{\max} \cdot (n \log n + N_{\max} \cdot m)),$$

donde $N_{\max}$ es el número máximo de soluciones no dominadas en un nodo, d es el número de objetivos, n es el número de nodos y m es el número de aristas.

En el caso bicriterio ($q = 2$), estas complejidades se reducen significativamente debido a la menor cantidad de verificaciones necesarias para comprobar dominancia.

Resumiendo, el problema de caminos más cortos multiobjetivo es un problema NP-completo caracterizado por un crecimiento exponencial en el número de soluciones no dominadas en el peor caso. Este crecimiento depende del tamaño del grafo, la profundidad de las soluciones y el número de objetivos considerados. Aunque los avances en algoritmos exactos y heurísticos han permitido manejar casos específicos de manera más eficiente, el MOSP sigue siendo intratable para grafos grandes con múltiples objetivos, lo que refuerza la necesidad de enfoques aproximados o heurísticos en aplicaciones prácticas.

B. Problema de Camino Más Corto Multiobjetivo de Uno-a-Uno

El Problema de Camino Más Corto Multiobjetivo (MOSP, por sus siglas en inglés) busca identificar rutas óptimas en un grafo dirigido $G = (V, A)$, donde V representa el conjunto de nodos y A el conjunto de arcos. Cada arco $a \in A$ está asociado con un vector

de costos multidimensional $c_a \in \mathbb{R}^d_{\geq 0}$, donde d es el número de criterios que se desea optimizar simultáneamente. Los costos de un camino p se definen como la suma de los costos de sus arcos, es decir:

$$c(p) := \sum_{a \in p} c_a.$$

Para un conjunto de caminos P, los costos asociados $c(P)$ se representan como el conjunto de todos los costos de los caminos en P:

$$c(P) := \{c(p) \mid p \in P\} \subseteq \mathbb{R}^d_{\geq 0}.$$

En el contexto multiobjetivo, un camino se evalúa con base en su dominancia y eficiencia. Dados dos caminos p y q entre dos nodos v y w en el grafo G, se dice que p domina a q si $c_i(p) \leq c_i(q)$ para todo $i \in \{1, \ldots, d\}$ y existe al menos un $j \in \{1, \ldots, d\}$ tal que $c_j(p) < c_j(q)$. Un camino p se llama eficiente si no existe ningún otro camino que lo domine; de lo contrario, se considera un camino dominado. El vector de costos $c(p)$ de un camino eficiente se denomina no dominado, mientras que el de un camino dominado también es considerado dominado.

El objetivo del problema es encontrar un conjunto completo mínimo de caminos eficientes entre un nodo fuente s y un nodo objetivo t. Este conjunto completo debe incluir al menos un camino s-t eficiente para cada vector de costos no dominado en $c(P_{st})$, donde P_{st} denota el conjunto de todos los caminos eficientes s-t en G. Dicho de otro modo, para cada vector de costos no dominado, los algoritmos que resuelven este problema deben devolver exactamente un camino eficiente que lo genere.

Formalmente, una instancia del Problema de Camino Más Corto Multiobjetivo de Uno-a-Uno se define como (G, s, t, c), donde $G = (V, A)$ es el grafo dirigido, $s \in V$ es el nodo fuente, $t \in V$ es el nodo objetivo y $c_a \in \mathbb{R}^d_{\geq 0}$ es el vector de costos asociado a cada arco $a \in A$.

El resultado del problema es un conjunto de caminos que representa una cobertura completa y mínima del frente de Pareto, asegurando que todas las soluciones no dominadas sean representadas por al menos un camino eficiente entre s y t.

Heurísticas y Cotas de Dominancia

En el contexto del Problema de Caminos Más Cortos Multiobjetivo de Uno-a-Uno (MOSP), se generalizan varios conceptos utilizados en el caso de un solo criterio para manejar múltiples dimensiones de costos. Consideremos una instancia d-dimensional del problema MOSP representada como $I = (G, s, t, c)$, donde G es un grafo dirigido, s y t son los nodos fuente y objetivo, respectivamente, y c son los costos multidimensionales asociados a las aristas.

Una heurística multiobjetivo $h : V \to \mathbb{R}^d_{\geq 0}$ se denomina admisible si cumple que el valor estimado de los costos desde cualquier nodo v hasta el nodo objetivo t no excede el costo real de un camino eficiente entre esos nodos. Formalmente, se dice que h es admisible si:

$$h(v) \leq c(p), \quad \forall v \in V, \, p \text{ camino eficiente } v\text{-}t\text{-camino}.$$

Adicionalmente, la heurística h es monótona si cumple dos condiciones clave: $h(t) = 0$, y para cualquier arco $(u, v) \in A$, se satisface:

$$h(u) \leq c_{uv} + h(v).$$

La monotonía garantiza que la estimación heurística respete las restricciones de costo a lo largo de las aristas, lo que permite preservar propiedades importantes como la optimalidad en los algoritmos que utilizan estas heurísticas. Es importante notar que una heurística monótona es siempre admisible.

Para simplificar el análisis de los costos en caminos multiobjetivo, se introducen los costos reducidos de un arco (u, v), definidos como:

$$\tilde{c}_{uv} := c_{uv} + h(v) - h(u).$$

Estos costos reducidos permiten ajustar los valores de los costos originales incorporando la información heurística, lo que facilita la identificación de caminos eficientes. Para un camino p que conecta s con v, los costos reducidos acumulados se expresan como:

$$\sum_{a \in p} \tilde{c}(a) = c(p) + h(v) - h(s).$$

Dado que todos los caminos considerados comienzan en el nodo fuente s, el término constante $h(s)$ puede ignorarse, resultando en:

$$\tilde{c}(p) := c(p) + h(v).$$

El uso de heurísticas y costos reducidos permite derivar propiedades útiles. Por ejemplo, si h es una heurística monótona, se garantiza que $\tilde{c}_a \geq 0$ para cualquier arco $a \in A$. Además, para un camino simple p que comienza en s, se cumple que los costos reducidos son no decrecientes a medida que se avanza en el camino:

$$\tilde{c}(p_{su}) \leq \tilde{c}(p_{sv}),$$

donde u precede a v en el camino.

Un concepto relacionado es el de las cotas de dominancia. Una cota de dominancia es un vector $\delta \in \mathbb{R}^d$ que actúa como un umbral, asegurando que ningún camino eficiente s-t esté dominado por δ. Este concepto es crucial en los algoritmos MOSP, ya que permite identificar y descartar caminos que no pueden extenderse para producir soluciones eficientes. Al comienzo de un algoritmo, cuando la información sobre los caminos eficientes es limitada, las cotas de dominancia sirven para guiar el proceso de búsqueda de manera más efectiva.

En los métodos descritos, se asume que tanto una heurística como una cota de dominancia forman parte de los datos de entrada del algoritmo. En etapas de preprocesamiento, estas herramientas se calculan para cada instancia del problema MOSP, facilitando su resolución en etapas posteriores.

C. El Algoritmo de Dijkstra Multiobjetivo Dirigido (T-MDA)

El algoritmo T-MDA[1] es una versión adaptada del Algoritmo de Dijkstra Multiobjetivo con fijación de etiquetas. Este algoritmo está diseñado específicamente para resolver el Problema de Caminos Más Cortos Multiobjetivo de Uno-a-Uno (MOSP). Una de las principales características del T-MDA es que evita almacenar caminos eficientes s-v para nodos intermedios $v \in V \setminus \{s, t\}$, si se puede demostrar que esos caminos no pueden extenderse para convertirse en caminos eficientes s-t. Esto lo diferencia de otros algoritmos

similares y lo hace más eficiente en términos prácticos. Aunque el algoritmo aumenta la complejidad espacial, este compromiso no afecta la cota teórica del tiempo de ejecución, logrando una mejora significativa en el rendimiento.

La entrada del T-MDA consiste en una instancia d-dimensional del problema MOSP representada como (G, s, t, c), una heurística h y una cota de dominancia δ_t. Los caminos explorados por el algoritmo se gestionan mediante una cola de prioridad Q, que está ordenada de manera lexicográfica con respecto a los costos reducidos $\tilde{c}$. La ordenación lexicográfica se define considerando que un vector x es menor que y si el primer componente j en el que difieren satisface $x_j < y_j$. Esta propiedad se aplica tanto a vectores de costos como a caminos.

El objetivo del T-MDA es encontrar un conjunto completo mínimo de caminos eficientes s-t. Para un nodo v, si existen dos caminos eficientes s-v, denotados por p y q, con los mismos costos reducidos $\tilde{c}(p) = \tilde{c}(q)$, pero uno de ellos ya fue almacenado, el otro puede descartarse. Esto es posible gracias a una relación binaria reflexiva denominada operador de dominancia o equivalencia, que permite verificar si un camino domina o es equivalente a otro.

El T-MDA clasifica los caminos en dos categorías principales. Los caminos permanentes son aquellos que han sido identificados como eficientes y que se almacenan en el conjunto P_{sv} para un nodo v. Por otro lado, los caminos explorados son aquellos que aún no han sido clasificados como permanentes, pero que no han sido descartados, ya que podrían extenderse para formar un camino eficiente s-t. Es importante destacar que los conjuntos de caminos explorados y permanentes son disjuntos en todo momento durante la ejecución del algoritmo.

El algoritmo utiliza una cola de prioridad Q para almacenar como máximo un único camino explorado por cada nodo v, seleccionado como el más prometedor según el criterio lexicográfico. Los caminos explorados que no están en la cola Q se almacenan en listas auxiliares denominadas NQP_{uv}, organizadas según el último arco del camino. Estas listas

permiten gestionar caminos que podrían volver a entrar en Q más adelante.

Algorithm 5: Algoritmo de Dijkstra Multiobjetivo A* (T-MDA)

Input: Instancia MOSP $I = (G, s, t, c)$, heurística h para I, cota de dominancia δ_t para I.
Output: Conjunto completo mínimo P_{st} de caminos eficientes s-t.
Inicializar la cola de prioridad $Q \leftarrow \emptyset$, ordenada según $\tilde{c}$;
$\forall (u, v) \in A$, listas de caminos explorados $\text{NQP}_{uv} \leftarrow \emptyset$;
$\forall v \in V$, listas de caminos eficientes $P_{sv} \leftarrow \emptyset$;
Insertar el camino inicial vacío p_{init} en Q;
while $Q \neq \emptyset$ **do**
    Extraer el camino p de Q con menor $\tilde{c}$;
    $v \leftarrow \text{head}(p)$ (el último nodo de p);
    $P_{sv} \leftarrow P_{sv} \cup \{p\}$;
    Buscar un nuevo camino de cola p^v_{new} para v usando `nextQueuePath`;
    if $p^v_{new} \neq NULL$ **then**
        Insertar p^v_{new} en Q;
    if $v = t$ **then**
        continue;
    for $w \in \delta^+(v)$ **do**
        Propagar p hacia w usando `propagate`;
return P_t;

Propagación en el Algoritmo T-MDA

La propagación es una etapa clave en el Algoritmo de Dijkstra Multiobjetivo Dirigido (T-MDA), donde un camino extraído de la cola de prioridad Q se expande hacia los nodos vecinos a través de las aristas salientes. Este proceso permite generar nuevos caminos que podrían formar parte del conjunto de soluciones eficientes o ser descartados según las condiciones de dominancia.

Dado un camino s-v, p, extraído de Q, y un nodo vecino w, la propagación genera un nuevo camino p^w_{new}, que resulta de extender p por la arista (v, w). Este nuevo camino tiene costos acumulados dados por:

$$\tilde{c}(p^w_{\text{new}}) = \tilde{c}(p) + \tilde{c}_{vw},$$

donde $\tilde{c}_{vw}$ representa los costos reducidos de la arista (v, w).

El camino p^w_{new} se evalúa utilizando una serie de condiciones de dominancia. Si cualquiera de estas condiciones se cumple, el camino es descartado inmediatamente:

- Si $\tilde{c}(p^w_{\text{new}})$ está dominado por algún vector de costos en $c(P_{st})$, el conjunto de costos de todos los caminos eficientes s-t encontrados hasta el momento.
- Si $\tilde{c}(p^w_{\text{new}})$ está dominado por la cota de dominancia δ_t, que actúa como un umbral para identificar caminos irrelevantes.
- Si $\tilde{c}(p^w_{\text{new}})$ está dominado por algún vector de costos en $c(P_{sw})$, el conjunto de costos de los caminos eficientes s-w.

En el caso de que p^w_{new} no sea dominado, se considera un nuevo camino explorado y se clasifica según las siguientes posibilidades:

- Si ya existe un camino s-w en la cola Q cuyo costo reducido $\tilde{c}$ es lexicográficamente menor que $\tilde{c}(p^w_{\text{new}})$, el nuevo camino se almacena en la lista NQP_{vw}, que contiene caminos explorados para w a través de v.

- Si ya existe un camino s-w en Q cuyo costo reducido $\tilde{c}$ es lexicográficamente mayor que $\tilde{c}(p^w_{\text{new}})$, el nuevo camino reemplaza al existente mediante una operación `decreaseKey`, y el camino reemplazado se almacena en NQP_{vw}.

- Si no existe ningún camino s-w en Q, el nuevo camino se inserta directamente en Q.

La propagación es fundamental para garantizar que solo los caminos más prometedores permanezcan en la cola de prioridad Q. Este enfoque permite minimizar la cantidad de caminos explorados que deben procesarse, mejorando la eficiencia del algoritmo.

Por último, es importante destacar que, una vez completada la propagación para un camino p, este se clasifica como permanente, moviéndolo al conjunto P_{sv}. De esta manera, el T-MDA mantiene una separación clara entre caminos explorados y caminos permanentes, asegurando que cada nodo tenga al menos un camino eficiente representado en P.

El procedimiento `propagate` devuelve la cola de prioridad Q posiblemente actualizada.

Algorithm 6: Procedimiento `propagate`

Input: Camino s-v, p; nodo $w \in \delta^+(v)$; cola de prioridad Q; caminos permanentes P; listas de caminos explorados NQP; cota de dominancia δ_t.
Output: Cola de prioridad Q actualizada.

$p^w_{\text{new}} \leftarrow p \circ (v, w)$;
if $c(P_{st}) \prec_D \tilde{c}(p^w_{new})$ ***or*** $\delta_t \prec_D \tilde{c}(p^w_{new})$ **then**
  return Q;
if $c(P_{sw}) \not\prec_D c(p^w_{new})$ **then**
  if $\neg Q.contains(w)$ **then**
    Q.insert(p^w_{new});
  else
    $q \leftarrow Q$.getPath(w);
    if $\tilde{c}(p^w_{new}) \prec_{lex} \tilde{c}(q)$ **then**
      Q.decreaseKey(w, p^w_{new});
      $(x, w) \leftarrow$ lastArc(q);
      NQP_{xw}.prepend(q) `// Agregar` q `al inicio de` NQP_{xw}.
    else
      NQP_{vw}.append(p^w_{new}) `// Agregar` p^w_{new} `al final de` NQP_{vw}.
return Q;

Un invariante clave en el T-MDA es que, en cualquier momento durante la ejecución, Q contiene como máximo un camino de cola para v, para cualquier nodo $v \in V$. Por lo tanto, después de extraer p de Q en el algoritmo principal, no hay ningún camino s-v en Q. El procedimiento `nextQueuePath` tiene como objetivo encontrar un nuevo camino de cola para v.

La búsqueda se realiza entre los caminos en las listas NQP_{uv} de caminos explorados s-v a través de u, con $u \in \delta^-(v)$. Estas listas están ordenadas en orden creciente léxico con respecto a $\tilde{c}$ (ver Lema 11). Entonces, comenzando con el primer camino en cada lista, se eliminan los caminos de NQP_{uv} hasta encontrar el primer camino en la lista que cumpla con las condiciones de un camino de cola o hasta vaciar la lista.

De esta forma, `nextQueuePath` identifica como máximo un candidato s-v a través de cada nodo predecesor u. El camino p^v_{new} devuelto por la búsqueda es el más pequeño léxicamente entre estos candidatos, si existe alguno. Luego, p^v_{new} se convierte en el camino de cola para v y se inserta en Q.

Algorithm 7: Procedimiento `nextQueuePath`

Input: Camino s-v, p; listas de caminos explorados NQP; conjunto de nodos $\delta^-(v)$; caminos permanentes P; cota de dominancia δ_t.
Output: Nuevo camino de cola para v, si existe.

```
p^v_new ← NULL // Asumimos c̃(p^v_new) = [∞]^d.
for u ∈ δ⁻(v) do
    for p_v ∈ NQP_uv do
        if c(P_st) ≺_D c̃(p_v) or δ_t ≺_D c̃(p_v) then
            NQP_uv.remove(p_v);
            continue;
        if ¬c(P_sv) ≺_D c(p_v) then
            if c̃(p_v) ≺_lex c̃(p^v_new) then
                p^v_new ← p_v;
            break;
return p^v_new;
```

Corrección del Algoritmo T-MDA

La corrección del Algoritmo T-MDA se basa en una serie de propiedades fundamentales que garantizan que el conjunto de caminos encontrados al final del proceso es un conjunto completo mínimo de caminos eficientes s-t. Este análisis se estructura en tres pasos principales: primero, se prueba que los elementos de las listas P_{sv} están ordenados en orden léxico creciente con respecto a los costos originales c y los costos reducidos $\tilde{c}$. Segundo, se demuestra que cada camino extraído de la cola de prioridad Q es un camino eficiente. Finalmente, se establece que el conjunto P_{st} contiene exactamente un camino eficiente para cada vector de costos no dominado c_t, completando así la corrección del algoritmo.

Para garantizar estas propiedades, comenzamos con la siguiente observación clave: para cualquier nodo $v \in V$, el orden léxico entre dos caminos s-v, p y q, es consistente tanto con los costos originales c como con los costos reducidos $\tilde{c}$. Formalmente:

$$\tilde{c}(p) \prec_{\text{lex}} \tilde{c}(q) \iff c(p) + h(v) \prec_{\text{lex}} c(q) + h(v) \iff c(p) \prec_{\text{lex}} c(q).$$

Esto implica que el orden en que los caminos son procesados en Q y almacenados en P_{sv} respeta este criterio de consistencia.

Propiedades de Orden y Propagación

Mientras las listas NQP de caminos explorados permanezcan ordenadas en orden léxico no decreciente con respecto a $\tilde{c}$, los caminos se extraen de Q en ese mismo orden. Esta propiedad asegura que los caminos procesados en cada iteración del algoritmo son siempre los más prometedores, de acuerdo con el criterio lexicográfico.

Las listas NQP también permanecen ordenadas incluso cuando se agregan nuevos caminos explorados. Esto es crucial, ya que las operaciones realizadas por las funciones `propagate` y `nextQueuePath` dependen de que las listas NQP mantengan esta estructura.

Eficiencia de los Caminos Extraídos

Cualquier camino s-v, p, extraído de Q es un camino eficiente. Supongamos, por contradicción, que p no es eficiente y existe un camino p_0 que lo domina. En tal caso, p_0 tendría que haber sido extraído antes que p debido a su orden léxico menor, lo que haría que p fuera descartado por las verificaciones de dominancia en las funciones del algoritmo. Esta contradicción demuestra que solo los caminos eficientes son extraídos de Q.

Conjunto Completo Mínimo de Caminos Eficientes

Al final del T-MDA, el conjunto P_{st} es un conjunto completo mínimo de caminos eficientes s-t. Esto significa que P_{st} contiene exactamente un camino para cada vector de costo no dominado c_t. La demostración se basa en que el algoritmo asegura que: 1. Cada camino en P_{st} es eficiente. 2. No hay vectores de costo no dominados que no estén representados en P_{st}. 3. Los caminos con el mismo vector de costos no dominado no se incluyen más de una vez en P_{st}.

Para lograr esto, el algoritmo utiliza las verificaciones de dominancia en las líneas clave de las funciones `propagate` y `nextQueuePath`. Estas verificaciones garantizan que solo se procesen los caminos que pueden contribuir al frente de Pareto final. Además, las operaciones de inserción y extracción en Q aseguran que los caminos se procesen en el orden adecuado para mantener la corrección del algoritmo.

La corrección del T-MDA se deriva de las propiedades de orden y propagación, junto con las verificaciones de dominancia implementadas en cada paso del algoritmo. Estas características garantizan que el conjunto P_{st} obtenido al final del algoritmo es un conjunto completo mínimo de caminos eficientes. Por lo tanto, P_{st} contiene exactamente un camino eficiente para cada vector de costo no dominado c_t.

Complejidad de las Búsquedas de Nuevos Caminos de Cola

El análisis de la complejidad del algoritmo T-MDA incluye tanto el manejo eficiente de la memoria como el impacto en el tiempo de ejecución. Consideremos un grafo dirigido G con n nodos y m arcos. Durante la ejecución del T-MDA, el número total de caminos extraídos se denota como N, mientras que $N_{\max}$ representa la máxima cantidad de caminos permanentes almacenados en cualquier lista P_{sv} al finalizar el algoritmo.

Para optimizar el almacenamiento de caminos, se utiliza una codificación basada en etiquetas (*labels*). Cada camino p que conecta s con v puede representarse como una tupla que almacena el nodo final v y una referencia al subcamino s-u, donde (u, v) es el último arco del camino. Esta representación garantiza que el espacio requerido para almacenar

un camino es $O(1)$. Además, los caminos completos pueden reconstruirse recursivamente en tiempo $O(n)$ tras la finalización del algoritmo, dado que el T-MDA solo genera caminos simples.

En la formulación original del MDA, no se utilizan listas NQP para almacenar caminos explorados. Cuando un nuevo camino explorado p^w_{new} hacia el nodo w no puede reemplazar al camino de cola actual en Q, se descarta temporalmente. Sin embargo, este camino se reconsidera en una etapa posterior mediante una subrutina llamada `nextCandidatePath`, que reconstruye caminos explorados expandiendo los caminos permanentes de los predecesores de w. Para agilizar este proceso, el MDA emplea índices llamados $\text{lastProcessedPath}_{uv}$, que indican el punto inicial de las búsquedas en las listas P_{su} para cada arco (u, v). Estos índices requieren $O(m)$ de memoria.

El T-MDA mejora este enfoque mediante las listas NQP, que almacenan las expansiones de los caminos permanentes s-v hacia un nodo w a lo largo de cada arco (v, w). Estas listas permiten evitar la repetición innecesaria de expansiones. Sin embargo, en el peor de los casos, las listas NQP pueden almacenar todas las expansiones de los caminos en P_{sv}, lo que implica un costo espacial adicional de $O(mN_{\max})$. Este aumento en el consumo de memoria es el precio a pagar por la mejora en el rendimiento práctico del algoritmo.

Impacto en la Complejidad Temporal

A pesar del incremento en el uso de memoria, el empleo de listas NQP no afecta las cotas de tiempo de ejecución del algoritmo original. En el MDA, las expansiones de caminos explorados requieren sumar los costos de los arcos, pero este esfuerzo es independiente del uso de listas NQP. Además, como estas listas se mantienen ordenadas automáticamente al insertar nuevos caminos en el inicio o final, no se requiere esfuerzo adicional para ordenarlas, lo cual está garantizado por la estructura del algoritmo.

El tiempo de ejecución del T-MDA depende del número de objetivos d. Para problemas biobjetivo ($d = 2$), el tiempo de ejecución es $O(N \log n + N_{\max} m)$. En problemas con más de dos objetivos ($d > 2$), la cota de tiempo es $O(dN_{\max}(n \log n + N_{\max} m))$. Estas cotas son consistentes con las del algoritmo MDA original, como se demuestra en trabajos previos. Sin embargo, el uso de listas NQP reduce significativamente el número de operaciones redundantes, mejorando el rendimiento práctico sin cambiar la complejidad teórica.

Finalmente, es importante destacar que, debido a la naturaleza del problema MOSP de Uno-a-Uno, no es posible diseñar un algoritmo con tiempo de ejecución polinómico en relación con el tamaño de la entrada y la salida. Esto se debe a que el número de soluciones no dominadas puede crecer exponencialmente con el tamaño del grafo. Sin embargo, las mejoras del T-MDA, incluyendo las listas NQP, ofrecen un compromiso eficiente entre el uso de memoria y el rendimiento práctico, haciendo del algoritmo una herramienta poderosa para problemas de optimización multiobjetivo.

Obtención de Heurísticas y Cotas de Dominancia

El algoritmo T-MDA requiere como entrada una heurística h y una cota de dominancia δ_t, las cuales desempeñan un papel fundamental en la optimización del proceso de búsqueda. Estas herramientas permiten ordenar los caminos en la cola de prioridad y aplicar técnicas de poda que descartan caminos irrelevantes antes de que sean procesados.

Cálculo de Puntos Ideales. Para una instancia d-dimensional del problema MOSP de Uno-a-Uno (G, s, t, c), se define el punto ideal de un conjunto $X \subseteq \mathbb{R}^d$ como:

$$x^\star := \left(\min_{x \in X} x_1, \ldots, \min_{x \in X} x_d \right),$$

donde $x^\star$ representa una cota inferior para los valores de los vectores no dominados en $X^\star$. Este concepto es esencial para construir heurísticas efectivas que guíen la búsqueda hacia el nodo objetivo t.

Para calcular los puntos ideales $c^\star_{v,t}$ de los conjuntos de costos $c(P_{vt})$, se ejecutan consultas léxicas de Dijkstra en un grafo invertido de G con raíz en t. Una consulta léxica utiliza un orden de preprocesamiento ξ, que es una permutación de $(1, \ldots, d)$, para procesar los caminos explorados en orden léxico no decreciente según el componente principal ξ_1. Al ejecutar d consultas léxicas, una para cada orden en un conjunto de preprocesamiento Θ, se obtiene el árbol de caminos más cortos invertido T_ξ y, para cada nodo v, un camino $T_\xi(v)$ con respecto al componente principal.

Heurística a Partir de Puntos Ideales. Los puntos ideales calculados en las consultas léxicas permiten definir una heurística admisible y monótona como:

$$h(v) := c^\star_{v,t}, \quad \forall v \in V.$$

Dado que $h(v)$ proporciona una estimación de los costos más bajos posibles para llegar al nodo t, se cumple que $h(v) \leq c(p)$ para cualquier camino eficiente v-t. Esto garantiza que h sea una heurística válida.

Construcción de la Cota de Dominancia. Además de la heurística, el T-MDA utiliza una cota de dominancia δ_t para descartar caminos explorados que no puedan extenderse hacia t para producir un camino eficiente. Esta cota se define como:

$$\delta_t := \left(\max_{\xi \in \Theta} c_1(T_\xi(s)) + \varepsilon, \ldots, \max_{\xi \in \Theta} c_d(T_\xi(s)) + \varepsilon \right), \quad \varepsilon > 0.$$

El término ε se introduce para garantizar que δ_t no esté dominada por ningún vector de costos de caminos eficientes. Por construcción, δ_t asegura que no existe un camino eficiente s-t, p, que sea dominado por δ_t.

Justificación Teórica. La validez de δ_t como cota de dominancia se demuestra considerando que para cualquier $i \in \{1, \ldots, d\}$:

$$c_i(p) > \delta_{t,i} = \max_{\xi \in \Theta} c_i(T_\xi(s)).$$

Esto implica que si un camino p es dominado por δ_t, entonces p no puede ser eficiente, ya que sus costos exceden los máximos permitidos definidos por los puntos ideales obtenidos en las consultas léxicas.

Tanto la heurística h como la cota de dominancia δ_t se utilizan para optimizar el T-MDA. La heurística guía la búsqueda hacia el nodo objetivo t al ordenar los caminos en la cola de prioridad Q en función de los costos reducidos $\tilde{c}$. Por otro lado, δ_t permite aplicar técnicas de poda, como la *poda por conjunto eficiente objetivo*, que descarta caminos

p cuando $c(P_{st}) \prec_D \tilde{c}(p)$, y la *poda por cota de dominancia*, que descarta p si $\delta_t \prec_D \tilde{c}(p)$. Estas estrategias contribuyen significativamente a reducir el espacio de búsqueda y mejorar el rendimiento computacional del algoritmo.

El cálculo de heurísticas y cotas de dominancia en el T-MDA es una etapa de preprocesamiento clave que asegura la eficiencia del algoritmo. Las consultas léxicas de Dijkstra no solo proporcionan estimaciones precisas de los costos ideales, sino que también facilitan la construcción de límites superiores que evitan el procesamiento de caminos irrelevantes. Esto refuerza la capacidad del T-MDA para resolver de manera efectiva problemas MOSP de Uno-a-Uno en contextos multiobjetivo complejos.

Demostración de Correctitud del Algoritmo T-MDA

Para demostrar la correctitud del algoritmo *Time-based Multiobjective Dijkstra Algorithm* (T-MDA), se debe establecer que el algoritmo encuentra todas las soluciones óptimas de Pareto en el grafo, que no devuelve caminos dominados y que el procedimiento termina en un número finito de pasos.

Sea un grafo dirigido $G = (V, E)$, donde cada arista $e_{ij} \in E$ tiene un vector de pesos $w_{ij} = (t_{ij}, d_{ij}, s_{ij})$ que representan el tiempo de viaje, la distancia y la seguridad respectivamente. El objetivo es determinar el conjunto de caminos $P(O, D)$ que son óptimos en el sentido de Pareto, desde un nodo origen O hasta un nodo destino D, de manera que no exista otro camino que sea estrictamente mejor en todos los criterios.

El algoritmo T-MDA se basa en la exploración de caminos utilizando una extensión del algoritmo de Dijkstra en la que se almacena un conjunto de soluciones no dominadas para cada nodo. En cada iteración, el algoritmo expande los caminos de manera lexicográfica y descarta aquellos que son dominados por otros previamente almacenados. Es necesario demostrar que este proceso garantiza que al finalizar la ejecución, el conjunto de soluciones encontradas es exactamente el frente de Pareto de caminos entre O y D.

Para demostrar la optimalidad de la solución generada por T-MDA, se prueba por inducción sobre la cantidad de iteraciones del algoritmo. Inicialmente, el conjunto de caminos hacia el nodo origen O contiene únicamente el camino trivial con costo cero en todos los criterios. En cada paso del algoritmo, se selecciona el camino no explorado con el menor tiempo acumulado y se expande hacia sus vecinos, generando nuevos caminos que son evaluados y almacenados solo si no son dominados por caminos previamente registrados en el nodo de destino. Si un camino nuevo domina a otro en el conjunto de soluciones de un nodo, este último se elimina, garantizando que en cada nodo solo se almacenan caminos eficientes en el sentido de Pareto.

El algoritmo garantiza que ningún camino óptimo es descartado erróneamente. Supóngase, en búsqueda de contradicción, que existe un camino óptimo de Pareto P^* de O a D que no es registrado en el conjunto de soluciones del nodo D. Dado que T-MDA explora todos los caminos posibles hasta alcanzar el nodo destino y solo descarta aquellos que son estrictamente dominados, la única posibilidad de que P^* no sea considerado es que haya sido incorrectamente marcado como dominado por otro camino P en algún nodo intermedio. Sin embargo, la relación de dominancia utilizada en T-MDA es transitiva y estricta en al menos un criterio, lo que implica que si P^* es efectivamente un camino de Pareto, no puede ser dominado por otro camino en el conjunto, lo que contradice la hipótesis inicial. Esto demuestra que T-MDA encuentra todas las soluciones de Pareto.

La terminación del algoritmo se garantiza porque el conjunto de soluciones en cada nodo está acotado por la cantidad de caminos de Pareto posibles en un grafo. Aunque este

número puede ser exponencial en el peor caso, sigue siendo finito para un grafo con un número finito de nodos y aristas. Además, la estructura de datos utilizada para almacenar las soluciones permite una actualización eficiente de los caminos en cada nodo sin generar ciclos infinitos.

Por lo tanto, se concluye que el algoritmo T-MDA es correcto, ya que encuentra todas las soluciones óptimas de Pareto, no devuelve caminos dominados y finaliza en un número finito de pasos.

Referencias

[1] Pedro Maristany de las Casas, Luitgard Kraus, Antonio Sedeno-Noda, Ralf Borndörfer. Targeted multiobjective Dijkstra Algorithm. journal Networks, volume 82, number 3, pages 277 – 298, 2023

[2] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1, 269271.

[3] Hart, P., Nilsson, N., Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100107.

[4] Hansen, P. (1979). Bicriterion path problems.

[5] Serafini, P. (1986). Some considerations about computational complexity for multiobjective combinatorial problems.

[6] Garey, M. R., Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman.

[7] Martins, E. Q. V., and Santos, J. L. (1999). The labeling algorithm for the multiobjective shortest path problem.

[8] Mandow, L., Pérez de la Cruz, J. L. (2009). Multiobjective A* search with bounded costs.

[9] E. W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik, vol. 1, no. 1, pp. 269–271, 1959.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100–107, 1968.

[11] K. Deb et al., *A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, 2002.

[12] Y. Chen et al., *Multi-Objective Path Planning Using Deep Reinforcement Learning*, IEEE Transactions on Intelligent Transportation Systems, 2022.

[13] J. Schulman et al., *Proximal Policy Optimization Algorithms*, arXiv preprint arXiv:1707.06347, 2017.

[14] H. Zhang et al., *Graph Neural Networks for Multi-Objective Routing Problems*, NeurIPS, 2023.

[15] J. Wang et al., *Convex Optimization for Multi-Objective Traffic Routing*, Transportation Science, 2022.

[16] Google, *Google Maps Navigation: Intelligent Routing Based on Real-time Traffic Data*, Technical Report, 2010.

[17] Amazon Logistics, *Optimized Delivery Networks Using AI and Graph Algorithms*, White Paper, 2020.

[18] UPS, *ORION: The Future of Logistics Optimization*, UPS Research and Development Report, 2016.

[19] Uber Engineering, *How Uber Optimizes Routes Using Graph Algorithms and Real-Time Data*, Uber Tech Blog, 2019.

[20] J. Moy, *OSPF Version 2*, RFC 2328, Internet Engineering Task Force (IETF), 2008.

[21] Facebook Engineering, *Optimizing Global Infrastructure Using Graph Theory*, Facebook AI Research, 2021.

[22] S. Ahmadi, G. Tack, D. Harabor, and P. Kilby. Bi-Objective Search with Bi-Directional A*. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of Leibniz International Proceedings in Informatics (LIPIcs), pages 3:13:15, Dagstuhl, Germany, 2021. Schloss Dagstuhl Leibniz-Zentrum für Informatik.

[23] F. K. Bökler. *Output-sensitive complexity of multiobjective combinatorial optimization with an application to the multiobjective shortest path problem*. PhD thesis, Technische Universität Dortmund, 2018.

[24] N. Cabrera, A. L. Medaglia, L. Lozano, and D. Duque. An exact bidirectional pulse algorithm for the constrained shortest path. *Networks*, 76(2):128146, jun 2020.

[25] C. Demetrescu, A. Goldberg, and D. Johnson. 9th DIMACS Implementation Challenge - Shortest Paths. `http://www.diag.uniroma1.it/~challenge9/`. Accessed: 2021-12-15.

[26] E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Operations Research*, 27(1):161186, feb 1979.

[27] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269271, dec 1959.

[28] D. Duque, L. Lozano, and A. L. Medaglia. An exact method for the biobjective shortest path problem for large-scale road networks. *European Journal of Operational Research*, 242(3):788797, 2015.

[29] M. Ehrgott. *Multicriteria Optimization*. Springer-Verlag, 2005.

[30] L. Mandow and J. L. Pérez-de-la Cruz. A new approach to multiobjective A* search. In *IJCAI*, 2005.

[31] L. Mandow and J. L. Pérez-de-la Cruz. Multiobjective A* search with consistent heuristics. *Journal of the ACM*, 57(5):125, jun 2010.

[32] P. Maristany de las Casas, A. Sedeño-Noda, and R. Borndörfer. An improved multi-objective shortest path algorithm. *Computers & Operations Research*, page 105424, jun 2021.

[33] E. Q. V. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236245, may 1984.

[34] F. J. Pulido, L. Mandow, and J. L. Pérez-de-la Cruz. Dimensionality reduction in multiobjective shortest path search. *Computers and Operations Research*, 64:6070, dec 2015.

[35] L. Pulido, F. J. Mandow, and J. L. Pérez-de-la Cruz. Multiobjective shortest path problems with lexicographic goal-based preferences. *European Journal of Operational Research*, 239(1):89101, nov 2014.

[36] A. Raith and M. Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):12991331, apr 2009.

[37] A. Raith, M. Schmidt, A. Schöbel, and L. Thom. Extensions of labeling algorithms for multi-objective uncertain shortest path problems. *Networks*, 72(1):84127, mar 2018.

[38] A. Sedeño-Noda and A. Raith. A Dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem. *Computers and Operations Research*, 57:8394, may 2015.

[39] A. Sedeño-Noda and M. Colebrook. A biobjective Dijkstra algorithm. *European Journal of Operational Research*, 276(1):106118, jul 2019.

[40] A. J. V. Skriver and K. A. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27(6):507524, may 2000.

[41] C. H. Ulloa, W. Yeoh, J. A. Baier, H. Zhang, L. Suazo, and S. Koenig. A simple and fast bi-objective search algorithm. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 143151. AAAI Press, 2020.