

Diseño de un Agente de Búsqueda en Grafos Planares

Caruzo Cieza David* david.caruzo.c@uni.pe
Inocente Caro Miguel Anderson† miguel.inocente.c@uni.pe
Rojas Torres Brayan Jesús Alexander‡ brayan.rojas.a@uni.pe
Rosales Ávila Edison Edwin§ edison.rosales.a@uni.pe
Ayma Quispe Willyams Alberto¶ williams.ayma.q@uni.pe

Universidad Nacional de Ingeniería
Investigación de Operaciones I
09 de julio de 2024

Abstract—Este documento presenta el desarrollo de un agente de búsqueda en grafos planares, diseñado para ayudar a un grupo de feligreses a recorrer todas las iglesias del distrito de Miraflores en el menor tiempo posible. Para lograr esto se emplean diversas técnicas de búsqueda en espacios de estados, tanto no informadas (búsqueda de anchura y de profundidad) como informadas (búsqueda voraz y algoritmo A*). El agente es implementado utilizando Python y la librería OSMNx para la obtención de datos geográficos y de interés. Los resultados indican que de todos los algoritmos probados el algoritmo A* presentó los mejores resultados de tiempo y rendimiento. Se recomienda usar fuentes de recolección de datos más precisas para tener un valor más certero como también probar con otras heurísticas en algoritmos.

Index Terms—Agente de búsqueda informadas y no informada, grafos planares, OSMNx, búsqueda en espacios de estados, Python, feligreses, rutas óptimas.

I. INTRODUCCIÓN

El problema se centra en diseñar un agente de búsqueda que permita a un grupo de feligreses recorrer todas las iglesias de un distrito sin distinción de religión ni estado de conservación, optimizando el tiempo de recorrido.

II. ANTECEDENTES

El uso de agentes de búsqueda en grafos es una técnica muy utilizada en problemas de optimización y navegación, en entornos teóricos y en aplicaciones prácticas. Estos agentes tienen como objetivo encontrar la mejor ruta o solución en un espacio de estados representado por un grafo, donde los nodos representan los estados y las aristas las posibles transiciones entre estos.

A. Agentes de Búsqueda

Los agentes de búsqueda se clasifican en dos categorías principales: agentes no informados y agentes informados. Los agentes no informados, como la búsqueda en anchura (BFS) y la búsqueda en profundidad (DFS), exploran el grafo sin conocimiento adicional sobre el entorno, lo que a menudo resulta en un rendimiento subóptimo en términos de tiempo y eficiencia. Por otro lado, los agentes informados, como el algoritmo A* y la búsqueda voraz, utilizan heurísticas para guiar la exploración, mejorando significativamente la eficiencia y la calidad de las soluciones encontradas.

B. Grafos Planares

Un grafo planar es un tipo de grafo que puede ser dibujado en un plano sin que sus aristas se crucen. Este tipo de grafos es útil en problemas de navegación urbana, donde las rutas deben planificarse en un espacio geográfico realista. En nuestro contexto, los grafos planares se emplean para modelar las calles y conexiones de un distrito, facilitando la tarea de encontrar rutas óptimas para los feligreses.

C. Aplicaciones en Entornos Urbanos

En aplicaciones prácticas, como la planificación de rutas en ciudades, los agentes de búsqueda deben manejar datos geográficos reales y complejos. OpenStreetMap (OSM) es una fuente abierta y colaborativa de datos geográficos que proporciona información detallada sobre calles, edificios y puntos de interés. La librería OSMNx, desarrollada por Geoff Boeing, permite descargar, procesar y analizar datos de OSM de manera eficiente, convirtiéndose en una herramienta esencial para proyectos de este tipo.

D. Investigación Previa

Estudios previos han demostrado la eficacia de los algoritmos de búsqueda informados en la optimización de rutas en entornos urbanos. Por ejemplo, el trabajo de Boeing (2017) con OSMNx ha permitido avanzar en la creación de modelos de red urbanos más precisos y detallados, lo que a su vez ha mejorado la capacidad de los agentes de búsqueda para encontrar rutas eficientes. Otros investigadores han explorado el uso de heurísticas específicas para entornos urbanos, optimizando aún más el rendimiento de estos agentes.

E. Importancia de la Optimización

La optimización de rutas es crucial en muchas áreas, desde la logística y el transporte público hasta el turismo y la planificación de eventos. En el contexto de este proyecto, la optimización de rutas para un grupo de feligreses tiene implicaciones prácticas significativas, ya que permite minimizar el tiempo de recorrido y mejorar la experiencia general de los participantes. La implementación de agentes de búsqueda eficientes no solo cumple con objetivos específicos, sino que también contribuye al avance general en el campo de la inteligencia artificial y la investigación operativa.

III. PROCEDIMIENTO

Definir el tipo de agente de búsqueda.

- Seleccionamos un agente de búsqueda en espacio de estados, ya que este modelo permite encontrar rutas óptimas en un grafo que representa las calles y conexiones del distrito.

Obtener y procesar los datos geográficos y de interés del distrito seleccionado.

- Utilizamos la librería `OSMNx` para descargar los datos geográficos de las calles y ubicaciones de las iglesias en el distrito de Miraflores. Este proceso incluyó la conversión de los datos descargados en grafos manejables mediante `GeoDataFrames`. La librería `OSMNx` facilita la obtención y visualización de estos datos.
- Código referenciado: Importación de librerías y descarga de datos geográficos.

Implementar algoritmos de búsqueda en espacio de estados.

- Se implementaron varios algoritmos de búsqueda tanto no informados como informados. Los algoritmos no informados incluyen la Búsqueda en Amplitud (BFS) y la Búsqueda en Profundidad (DFS). Los algoritmos informados incluyen la Búsqueda Voraz y el Algoritmo A^* .
- Cada algoritmo fue diseñado para encontrar la ruta más corta entre las iglesias, considerando el peso de las aristas (distancia entre ubicaciones).
- Código referenciado: Implementación de algoritmos DFS, BFS, Búsqueda Voraz y A^* .

Evaluar y comparar los resultados obtenidos con cada algoritmo.

- Se midió la efectividad de cada algoritmo en términos de distancia total recorrida y el tiempo de cómputo necesario para encontrar la ruta.
- La evaluación incluyó la visualización de las rutas encontradas en el grafo y la comparación de estas rutas para determinar cuál algoritmo ofrecía el mejor desempeño.
- Código referenciado: Cálculo de la distancia total y visualización de rutas.

IV. RESULTADOS ESPERADOS

Ruta óptima: Se espera obtener la ruta más corta para visitar todas las iglesias del distrito de Miraflores utilizando los diferentes algoritmos de búsqueda. Esto permitirá que los feligreses completen su recorrido de manera eficiente y en el menor tiempo posible. Los resultados mostrarán diversas perspectivas y eficiencias en la planificación de rutas, permitiendo evaluar cuál algoritmo es más adecuado para este tipo de problema.

Validación y comparación de algoritmos: Evaluamos la efectividad de cada algoritmo mediante la medición de la distancia total recorrida y el tiempo de cómputo necesario para encontrar la ruta. La medida de racionalidad se calculó comparando la ruta encontrada por el agente con la mejor ruta posible, con el objetivo de acercarse lo más posible a la solución óptima.

V. PLANTEAMIENTO DEL PROBLEMA

A. Definición del Problema

El objetivo principal de este proyecto es diseñar un agente de búsqueda que permita a un grupo de feligreses recorrer todas las iglesias de un distrito sin distinción de religión ni estado de conservación, optimizando el tiempo de recorrido. Este problema se puede modelar como un problema de búsqueda en grafos, donde las iglesias representan los nodos y las rutas entre ellas las aristas.

B. Importancia del Problema

Optimizar el tiempo de recorrido es importante en diversas situaciones prácticas. En el contexto de los feligreses, minimizar el tiempo de recorrido no solo mejora su experiencia, sino que también puede tener implicaciones logísticas y económicas, tales como reducir costos de transporte y tiempo de espera.

C. Desafíos del Problema

Los principales desafíos incluyen la obtención de datos geográficos precisos, la implementación eficiente de algoritmos de búsqueda, y la necesidad de manejar un gran volumen de datos y complejidades del entorno urbano.

VI. SISTEMATIZACIÓN DE LOS CONCEPTOS

Para abordar este problema, es importante sistematizar los conceptos y metodologías que serán utilizados:

- **Grafo Planar:** Un grafo que puede ser dibujado en un plano sin que sus aristas se crucen. En este proyecto, los nodos representan las ubicaciones de las iglesias y las aristas las posibles rutas entre ellas.
- **Agente de Búsqueda:** Un programa que navega a través de un espacio de estados buscando una solución óptima. Los agentes de búsqueda pueden ser informados o no informados, dependiendo de si utilizan o no heurísticas para guiar su exploración.
- **Búsqueda en Espacio de Estados:** Técnicas utilizadas para explorar las posibles configuraciones de un problema, buscando un estado objetivo. Los algoritmos de búsqueda en espacio de estados, como A^* y BFS, serán implementados y comparados en este proyecto.

VII. ONTOLOGÍA Y RED SEMÁNTICA

1. La ontología del problema define las entidades y sus relaciones en el dominio del proyecto:

- **Nodos (Iglesias):** Representan las ubicaciones de las iglesias en el distrito.
- **Aristas (Rutas):** Representan las conexiones entre las iglesias. Las aristas están ponderadas por la distancia o tiempo de viaje.
- **Agentes (Feligreses):** Son los que recorrerán las rutas optimizadas por el agente de búsqueda.
- La red semántica es una representación gráfica de estas entidades y sus relaciones. Los nodos son las iglesias y las aristas son las rutas entre ellas, formando un grafo que el agente debe explorar para encontrar la ruta óptima.

2. Objetivos Específicos:

- Desarrollar un agente de búsqueda: Crear un agente capaz de encontrar rutas óptimas entre las iglesias.

- Implementar y comparar algoritmos de búsqueda: Evaluar el rendimiento de diferentes algoritmos de búsqueda en espacio de estados.
- Optimizar el recorrido: Minimizar el tiempo total de recorrido para los feligreses.

VIII. METODOLOGÍA DE DESARROLLO DEL PROYECTO

Objetivo General: Desarrollar un procedimiento para implementar un agente de búsqueda en grafos planares, con el fin de optimizar el recorrido de un grupo de feligreses que deben visitar todas las iglesias de un distrito en el menor tiempo posible.

Pasos del Método:

A. Configuración del Entorno

- Instalar las librerías necesarias (osmnx, networkx, matplotlib, pandas, pathfinding) para la obtención y manipulación de datos geográficos, visualización de gráficos, gestión de datos y implementación de algoritmos de búsqueda.

B. Descarga de Ubicaciones y Líneas de Dirección

- Utilizar osmnx para descargar las ubicaciones y líneas de dirección desde OpenStreetMap para un distrito específico.
- Convertir los datos descargados a formatos manejables como GeoDataFrames.

C. Preprocesamiento del Grafo

- Eliminar nodos no relevantes y rutas ciegas.
- Convertir el grafo dirigido a no dirigido.
- Eliminar nodos aislados y componentes conectados no relevantes.
- Implementar una función para eliminar rutas ciegas.

D. Almacenamiento de Lugares de Interés

- Seleccionar puntos de interés con características específicas (por ejemplo, iglesias).
- Convertir polígonos a puntos y filtrar los nodos de interés presentes en el grafo preprocesado.

E. Cálculo de Distancias entre Nodos

- Calcular distancias entre pares de nodos de interés utilizando el algoritmo de camino más corto ponderado por la longitud de los bordes.

F. Presentación de Estadísticas del Problema

- Calcular y presentar estadísticas relevantes como el número de ubicaciones, número de bordes, radio y diámetro del grafo.
- Generar un histograma del grado de cada ubicación.

G. Estimación de Ruta Más Corta

- Implementar el método del vecino más cercano para estimar la ruta más corta que un feligrés puede recorrer para visitar todas las iglesias.

H. Definición e Implementación de Algoritmos de Búsqueda en Espacio de Estados

- Implementar varios algoritmos de búsqueda en espacio de estados, incluyendo:
 - Búsqueda en Profundidad (DFS)
 - Búsqueda en Amplitud (BFS)
 - Búsqueda Voraz (Greedy Best-First Search)
 - A* (A-Star)

I. Experimentación y Análisis de Resultados

- Ejecutar ejemplos de los algoritmos implementados y analizar los resultados obtenidos.
- Calcular la medida de rendimiento del procedimiento y comparar los tiempos de recorrido utilizando diferentes estrategias.

IX. CONSTRUCCIÓN DE LOS COMPONENTES

A. Descargar las Ubicaciones y Líneas de Dirección

```
1 import osmnx as ox
2 import matplotlib.pyplot as plt
3
4 # Definir el lugar y descargar el grafico de red
5 place_name = "Miraflores, Lima, Peru"
6 G = ox.graph_from_place(place_name, network_type='
  walk')
7
8 # Convertir los nodos y aristas a GeoDataFrames
9 nodes, edges = ox.graph_to_gdfs(G)
10
11 # Visualizar el grafico de red
12 fig, ax = ox.plot_graph(G, show=False, close=False
  )
13 plt.show()
```

Explicación:

1) Declarativa:

- Se importan las bibliotecas necesarias: osmnx como ox y matplotlib.pyplot como plt.
- Se define la variable place_name con el nombre del lugar de interes.

2) Procedural:

- Se descarga el grafico de red para Miraflores usando ox.graph_from_place().
- Se convierten los nodos y aristas del grafico a GeoDataFrames.
- Se crea una visualizacion del grafico de red y se muestra.

3) Resultado:

- Las calles aparecieran como lineas, y las intersecciones como puntos.
- Este mapa servira como base para planificar la ruta entre las iglesias.



Fig. 1. Mapa de red calles del distrito de Miraflores.

B. Preprocesamiento del Grafo

```
1 import networkx as nx
2
3 # Funcion para eliminar rutas ciegas
4 def eliminar_rutas_ciegas(G):
5     nodos_con_un_vecino = [n for n, grado in dict(
6         G.degree()).items() if grado == 1]
7     while nodos_con_un_vecino:
8         G.remove_nodes_from(nodos_con_un_vecino)
9         nodos_con_un_vecino = [n for n, grado in
10                                dict(G.degree()).items() if grado ==
11                                1]
12     return G
```

Explicación:

1) Declarativa:

- Importación de la biblioteca networkx y la definición de la función eliminar_rutas_ciegas. Estas declaraciones establecen las herramientas y funciones que se utilizarán en la parte procedural.

2) Procedural:

```
1 # Convertir los nodos y bordes a
2 DataFrames
3 nodos, bordes = ox.graph_to_gdfs(G)
4
5 # Mostrar el numero de nodos y bordes
6 antes del preprocesamiento
7 nodosAntes = len(nodos)
8 bordesAntes = len(bordes)
9 print(f"Numero de nodos antes del
10 preprocesamiento: {nodosAntes}")
11 print(f"Numero de bordes antes del
12 preprocesamiento: {bordesAntes}")
13
14 # Convertir el grafo dirigido a no
15 dirigido
16 G_no_dirigido = G.to_undirected()
17
18 # Eliminar nodos aislados
19 nodosAislados = list(nx.isolates(
20     G_no_dirigido))
21 G_no_dirigido.remove_nodes_from(
22     nodosAislados)
23 print(f"Numero de nodos aislados
24 eliminados: {len(nodosAislados)}")
25
26 # Eliminar componentes conectados no
27 relevantes
28 componente = max(nx.connected_components(
29     G_no_dirigido), key=len)
```

```
20 G_no_dirigido = G_no_dirigido.subgraph(
21     componente).copy()
22 print(f"Numero de componentes conectados
23 no relevantes eliminados")
24
25 # Eliminar rutas ciegas
26 G_no_dirigido = eliminar_rutas_ciegas(
27     G_no_dirigido)
28
29 # Mostrar el numero de nodos y bordes
30 despues del preprocesamiento
31 nodosDespues = G_no_dirigido.
32     number_of_nodes()
33 bordesDespues = G_no_dirigido.
34     number_of_edges()
35 print(f"Numero de nodos despues del
36 preprocesamiento: {nodosDespues}")
37 print(f"Numero de bordes despues del
38 preprocesamiento: {bordesDespues}")
39
40 # Grafico de red procesado
41 fig, ax = ox.plot_graph(G_no_dirigido,
42     show=False, close=False)
43 plt.show()
```

Explicación adicional:

- Convierte el grafo a DataFrames y muestra estadísticas iniciales.
- Convierte el grafo a no dirigido.
- Elimina nodos aislados.
- Elimina componentes no conectados al componente principal.
- Elimina rutas ciegas.
- Muestra estadísticas finales.
- Visualiza el grafo procesado.

Resultado

Número de nodos antes del preprocesamiento: 3408
Número de bordes antes del preprocesamiento: 10406

Fig. 2. Número de nodos y bordes antes del preprocesamiento.

Número de nodos aislados eliminados: 0

Fig. 3. Número de nodos aislados eliminados.

Número de nodos después del preprocesamiento: 2979
Número de bordes después del preprocesamiento: 4795

Fig. 4. Número de nodos y bordes después del preprocesamiento.



Fig. 5. Nuevo mapa de la red de calles de Miraflores.

C. Almacenar los lugares de interés

Declarativa:

```
1 tags = {'amenity': 'place_of_worship', 'religion':  
2         'christian'}
```

Define los criterios para identificar las iglesias en OpenStreetMap.

Procedural:

```
1 # Almacenar lugares de interes  
2 points = ox.features_from_place(place_name, tags)  
3  
4 # Convertir poligonos a puntos usando el centroide  
5 points['geometry'] = points['geometry'].apply(  
6     lambda geom: geom.centroid if geom.geom_type  
7     in ['Polygon', 'MultiPolygon'] else geom)  
8  
9 # Lista de nodos correspondientes a iglesias  
10 iglesias_nodos = ox.distance.nearest_nodes(G,  
11     points.geometry.x, points.geometry.y)  
12  
13 # Filtrar los nodos de las iglesias para incluir  
14 # solo aquellos presentes en el grafo  
15 # preprocesado  
16 iglesias_nodos = [nodo for nodo in iglesias_nodos  
17     if nodo in G_no_dirigido.nodes]  
18  
19 # Visualizar las iglesias en el grafico de red  
20 fig, ax = ox.plot_graph(G_no_dirigido, show=False,  
21     close=False)  
22 points.plot(ax=ax, color='red', markersize=10)  
23 plt.show()
```

Explicación adicional:

- Obtiene los lugares de culto cristianos del distrito de Miraflores.
- Convierte cualquier polígono (edificios de iglesias) a puntos usando sus centroides.
- Encuentra los nodos del grafo más cercanos a cada iglesia.
- Filtra estos nodos para asegurarse de que existan en el grafo preprocesado.
- Visualiza el grafo de calles con las iglesias marcadas en rojo.

Resultado



Fig. 6. Ubicaciones de las iglesias que se encuentran en el distrito Miraflores.

D. Distancia entre nodos

Declarativo:

```
1 from itertools import combinations
```

Importa la función combinations del módulo itertools. Esta función se utiliza para generar todas las combinaciones posibles de pares de elementos de una lista.

Procedural:

```
1 # Calcular distancias entre todos los pares de  
2   # nodos de interes  
3 distancias = {}  
4 for u, v in combinations(iglesias_nodos, 2):  
5     try:  
6         length = nx.shortest_path_length(  
7             G_no_dirigido, u, v, weight='length')  
8         distancias[(u, v)] = length  
9     except nx.NetworkXNoPath:  
10         continue  
11  
12 # Mostrar algunas de las distancias calculadas  
13 print("Primeras 10 distancias:")  
14 for i, ((u, v), dist) in enumerate(list(distancias  
15     .items())[:10]):  
16     print(f"{i+1}. Distancia entre {u} y {v}: {  
17         dist:.2f} metros")  
18  
19 print("\nUltimas 10 distancias:")  
20 for i, ((u, v), dist) in enumerate(list(distancias  
21     .items())[-10:]):  
22     print(f"{len(distancias) - 9 + i}. Distancia  
23     entre {u} y {v}: {dist:.2f} metros")
```

Explicación adicional:

- Crea un diccionario vacío distancias para almacenar las distancias entre pares de nodos.
- Utiliza combinations(iglesias_nodos, 2) para generar todos los pares posibles de nodos de iglesias.
- Para cada par de nodos (u, v):
 - Intenta calcular la longitud del camino más corto entre u y v usando nx.shortest_path_length().
 - Si se encuentra un camino, almacena la distancia en el diccionario distancias.
 - Si no hay camino entre los nodos, se salta ese par (esto puede ocurrir si los nodos están en componentes desconectados del grafo).

Resultado

```
Primeras 10 distancias:
1. Distancia entre 263636652 y 4332504766: 3215.60 metros
2. Distancia entre 263636652 y 4266659697: 3115.59 metros
3. Distancia entre 263636652 y 108237811: 3105.76 metros
4. Distancia entre 263636652 y 138851464: 2272.06 metros
5. Distancia entre 263636652 y 6122446208: 1755.32 metros
6. Distancia entre 263636652 y 263617506: 812.68 metros
7. Distancia entre 263636652 y 10800984803: 2868.97 metros
8. Distancia entre 263636652 y 262725526: 4087.69 metros
9. Distancia entre 263636652 y 262571000: 2985.43 metros
10. Distancia entre 263636652 y 114615160: 1123.12 metros

Últimas 10 distancias:
198. Distancia entre 6083786756 y 138854739: 2867.34 metros
199. Distancia entre 6083786756 y 262578746: 3967.11 metros
200. Distancia entre 262572744 y 386837776: 1787.94 metros
201. Distancia entre 262572744 y 4332504766: 1857.15 metros
202. Distancia entre 262572744 y 138854739: 1395.51 metros
203. Distancia entre 262572744 y 262578746: 2110.02 metros
204. Distancia entre 386837776 y 4332504766: 2748.75 metros
205. Distancia entre 386837776 y 138854739: 1369.20 metros
206. Distancia entre 386837776 y 262578746: 2577.46 metros
207. Distancia entre 138854739 y 262578746: 1244.02 metros
```

Fig. 7. Muestra las primeras y últimas 10 distancias entre nodos.

Se muestran las 10 primeras y últimas distancias entre iglesias. En la figura 7 se muestra los números (263636652, 4332504766, etc.) que son los identificadores de los nodos en el grafo que representan las ubicaciones de las iglesias.

E. Estadísticas del problema

Declarativo:

```
import pandas as pd
```

Importa la biblioteca pandas, que se utiliza para manejar y visualizar datos.

Procedural:

```
# Histograma del grado de cada ubicacion
degree_hist = pd.Series(dict(G_no_dirigido.degree
    ())).value_counts().sort_index()
# Mostrar estadísticas
print(f"Numero de ubicaciones: {nodosDespues}")
print(f"Numero de bordes: {bordesDespues}")

radio = nx.radius(G_no_dirigido)
diametro = nx.diameter(G_no_dirigido)
print(f"Radio del grafo: {radio}")
print(f"Diametro del grafo: {diametro}")

degree_hist.plot(kind='bar')
plt.title('Histograma del Grado de Cada Ubicacion')
plt.xlabel('Grado')
plt.ylabel('Frecuencia')
plt.show()
```

Explicación adicional:

- Imprime estadísticas básicas del grafo: número de nodos y bordes.
- Calcula y muestra el radio y diámetro del grafo.

Resultado

```
Número de ubicaciones: 2979
Número de bordes: 4795
Radio del grafo: 50
Diámetro del grafo: 99
```

Fig. 8. Estadísticas del grafo.



Fig. 9. Histograma del grado de cada ubicación.

F. Estimación de la ruta más corta

Declarativo:

```
def encontrar_mas_cercano(nodo_actual,
    nodos_restantes, lista_adyacencia):
    min_dist = float('inf')
    nodo_mas_cercano = None
    for nodo in nodos_restantes:
        if nodo in lista_adyacencia[nodo_actual]:
            dist = lista_adyacencia[nodo_actual][
                nodo]
            if dist < min_dist:
                min_dist = dist
                nodo_mas_cercano = nodo
    return nodo_mas_cercano, min_dist

# Crear lista de adyacencia
lista_adyacencia = {nodo: {} for nodo in
    iglesias_nodos}
```

Define la función `encontrar_mas_cercano` y toma tres argumentos:

- `nodo_actual`: el nodo desde el que se está buscando el vecino más cercano.
- `nodos_restantes`: conjunto de nodos que aún no han sido visitados.
- `lista_adyacencia`: estructura de datos que contiene las distancias entre nodos.

Inicializa variables para almacenar la distancia mínima y el nodo más cercano, luego itera sobre los nodos restantes para encontrar el más cercano.

Procedural:

```
for u, v in combinations(iglesias_nodos, 2):
    try:
        length = nx.shortest_path_length(G, u, v,
            weight='length')
        lista_adyacencia[u][v] = length
        lista_adyacencia[v][u] = length
```

```

6     except nx.NetworkXNoPath:
7         continue
8
9     # Nodo inicial (puede ser cualquier nodo de inicio
10    , aquí se toma el primero)
11    nodo_inicial = iglesias_nodos[0]
12
13    # Inicializar variables para el recorrido
14    nodo_actual = nodo_inicial
15    nodos_restantes = set(iglesias_nodos)
16    nodos_restantes.remove(nodo_actual)
17    ruta = [nodo_actual]
18    distancia_total = 0
19
20    # Recorrer todas las iglesias usando el método
21    del vecino más cercano
22    while nodos_restantes:
23        nodo_mas_cercano, min_dist =
24            encontrar_mas_cercano(nodo_actual,
25                                nodos_restantes, lista_adyacencia)
26        if nodo_mas_cercano is None:
27            break
28        ruta.append(nodo_mas_cercano)
29        distancia_total += min_dist
30        nodo_actual = nodo_mas_cercano
31        nodos_restantes.remove(nodo_actual)
32
33    # Regresar al nodo inicial para completar el ciclo
34    if nodo_inicial in lista_adyacencia[nodo_actual]:
35        distancia_total += lista_adyacencia[
36            nodo_actual][nodo_inicial]
37        ruta.append(nodo_inicial)
38
39    # Mostrar la ruta y la distancia total
40    print("Ruta del feligrés:", ruta)
41    print(f"Distancia total recorrida: {
42        distancia_total:.2f} metros")
43
44    # Construir la ruta completa paso a paso
45    ruta_completa = []
46    for u, v in zip(ruta[:-1], ruta[1:]):
47        try:
48            path = nx.shortest_path(G, u, v, weight='
49            length')
50            ruta_completa.extend(path[:-1]) # Excluir
51            el último nodo para evitar
52            duplicados
53        except nx.NetworkXNoPath:
54            print(f"No se encontró un camino entre {u
55            } y {v}")
56    ruta_completa.append(ruta[-1]) # Agregar el
57    último nodo
58
59    # Visualizar la ruta completa en el grafo
60    fig, ax = ox.plot_graph_route(G, route=
61    ruta_completa, route_linewidth=2, node_size=0)
62    plt.show()

```

Este código implementa un algoritmo del vecino más cercano para encontrar una ruta que visite todas las iglesias en Miraflores. Comienza completando la lista de adyacencia, que es una estructura de datos que almacena las distancias entre cada par de iglesias. Utiliza la función `combinations` para generar todos los pares posibles de iglesias y luego calcula la distancia más corta entre cada par usando `nx.shortest_path_length`. Si no existe un camino entre dos iglesias, ese par se omite.

Luego, el código inicia el recorrido seleccionando la primera iglesia como punto de partida. Crea un conjunto de nodos restantes por visitar, inicializa la ruta con el nodo inicial y establece la distancia total en cero.

El núcleo del algoritmo es un bucle `while` que continúa mientras haya nodos (iglesias) por visitar. En cada iteración, utiliza la función `encontrar_mas_cercano` para determinar la iglesia más cercana no visitada. Esta

iglesia se añade a la ruta, se actualiza la distancia total, se marca como el nuevo nodo actual y se elimina del conjunto de nodos restantes. Si en algún momento no se encuentra un nodo cercano (lo que podría ocurrir si el grafo no está completamente conectado), el bucle se rompe.

Después de visitar todas las iglesias posibles, el código intenta completar el ciclo volviendo al punto de inicio. Si existe un camino directo desde la última iglesia visitada hasta la inicial, se añade esta distancia al total.

A continuación, el código construye la ruta completa, incluyendo todos los nodos intermedios entre cada par de iglesias en la ruta. Utiliza `nx.shortest_path` para encontrar el camino más corto entre cada par de iglesias consecutivas en la ruta. Si no se encuentra un camino entre dos iglesias, se imprime un mensaje de error.

Finalmente, el código visualiza la ruta completa en el mapa de la ciudad utilizando la función `plot_graph_route` de `OSMnx`. Esta visualización muestra la red de calles de Miraflores con la ruta calculada resaltada.

Resultado:

```

Ruta del feligrés: [263636652, 1760085377, 386837776, 26
Distancia total recorrida: 18050.39 metros

```

Fig. 10. Output en consola.



Fig. 11. Trayectoria mínima.

G. Implementación de Algoritmos de Búsqueda

Búsqueda en Profundidad (DFS): Comienza su búsqueda con el nodo actual en el nodo inicio y la pila en la cual irá guardando los nodos visitados vacía. En cada paso se tomará como nodo actual a un nodo adyacente que no esté en la pila y se insertará en ella. Al llegar a nodos sin nodos adyacentes que no estén en la pila, se volverá al nodo anterior (cima de la pila). Si se alcanza el nodo destino, se tendrá una solución en la pila, y se operará como si éste fuera un nodo infructuoso a fin de realizar una búsqueda exhaustiva que ofrezca la solución óptima, o todas las soluciones posibles (cuando esto sea necesario) [2].

Declarativo:

```

1 # Crear función DFS para recorrer todas las
  iglesias
2 def dfs(grafo, nodo_inicial):
3     visitados = set()
4     stack = [nodo_inicial]
5     recorrido = []
6     while stack:
7         nodo = stack.pop()
8         if nodo not in visitados:
9             visitados.add(nodo)
10            recorrido.append(nodo)
11            for vecino in grafo[nodo]:
12                if vecino not in visitados:
13                    stack.append(vecino)
14    return recorrido
15
16 # Crear lista de adyacencia para el grafo
  procesado
17 lista_adyacencia_dfs = {nodo: set() for nodo in
  iglesias_nodos}
18 for u, v in combinations(iglesias_nodos, 2):
19     try:
20         length = nx.shortest_path_length(
21             G_no_dirigido, u, v, weight='length')
22         lista_adyacencia_dfs[u].add(v)
23         lista_adyacencia_dfs[v].add(u)
24     except nx.NetworkXNoPath:
25         continue
26
27 # Inicializar variables para guardar el mejor
  resultado
28 mejor_ruta = None
29 menor_distancia = float('inf')
30 mejor_nodo_inicial = None
31
32 import random
33
34 def calcular_racionalidad_promedio(grafo,
  lista_adyacencia, nodos_interes,
  distancia_optima, iteraciones=10):
35     sum_racionalidad = 0
36
37     for _ in range(iteraciones):
38         # Nodo inicial aleatorio
39         nodo_inicial_aleatorio = random.choice(
40             nodos_interes)
41
42         # Realizar el recorrido DFS desde el nodo
  inicial aleatorio
43         ruta_dfs_aleatoria = dfs(lista_adyacencia,
44             nodo_inicial_aleatorio)
45
46         distancia_total_dfs_aleatoria = 0
47         for u, v in zip(ruta_dfs_aleatoria[:-1],
48             ruta_dfs_aleatoria[1:]):
49             try:
50                 distancia_total_dfs_aleatoria +=
51                     nx.shortest_path_length(grafo,
52                         u, v, weight='length')
53             except nx.NetworkXNoPath:
54                 continue
55
56         # Medida de racionalidad para esta
  iteracion
57         racionalidad_dfs =
58             distancia_total_dfs_aleatoria /
59             distancia_optima
60         sum_racionalidad += racionalidad_dfs
61
62         # Medida de racionalidad promedio
63         racionalidad_promedio = sum_racionalidad /
64             iteraciones
65     return racionalidad_promedio
66
67 distancia_optima = 18050.39
68 iteraciones = 10
69
70 racionalidad_promedio_dfs =
71     calcular_racionalidad_promedio(G_no_dirigido,
72     lista_adyacencia_dfs, iglesias_nodos,

```

```

    distancia_optima, iteraciones)

```

```

62
63 # Mostrar el resultado
64 print(f"Medida de Racionalidad Promedio del
  Algoritmo DFS (en {iteraciones} iteraciones):
  {racionalidad_promedio_dfs:.2f}")

```

Explicación:

- La función `dfs` implementa un recorrido en profundidad (*Depth-First Search*) en un grafo.
- Se crea una lista de adyacencia `lista_adyacencia_dfs` para representar el grafo de iglesias.
- Se inicializan variables para almacenar la mejor ruta encontrada.
- Se define una función `calcular_racionalidad_promedio` para evaluar la eficiencia del algoritmo DFS.

Procedural

```

1 # Probar cada nodo como nodo inicial
2 for nodo_inicial_dfs in iglesias_nodos:
3     # Realizar el recorrido DFS
4     ruta_dfs = dfs(lista_adyacencia_dfs,
5         nodo_inicial_dfs)
6
7     # Calcular la distancia total recorrida en el
  recorrido DFS
8     distancia_total_dfs = 0
9     for u, v in zip(ruta_dfs[:-1], ruta_dfs[1:]):
10         try:
11             distancia_total_dfs += nx.
12                 shortest_path_length(G_no_dirigido
13                     , u, v, weight='length')
14         except nx.NetworkXNoPath:
15             continue
16
17     # Si la distancia total de esta ruta es menor
  que la mejor encontrada hasta ahora,
  actualizar
18     if distancia_total_dfs < menor_distancia:
19         mejor_ruta = ruta_dfs
20         menor_distancia = distancia_total_dfs
21         mejor_nodo_inicial = nodo_inicial_dfs
22
23 # Visualizar la mejor ruta DFS en el grafo
24 ruta_completa_dfs = []
25 for u, v in zip(mejor_ruta[:-1], mejor_ruta[1:]):
26     try:
27         path = nx.shortest_path(G_no_dirigido, u,
28             v, weight='length')
29         ruta_completa_dfs.extend(path[:-1])
30     except nx.NetworkXNoPath:
31         print(f"No se encontr un camino entre {u
32             } y {v}")
33     ruta_completa_dfs.append(mejor_ruta[-1])
34
35 # C lculo de la racionalidad promedio
36 distancia_optima = 18050.39
37 iteraciones = 10
38 racionalidad_promedio_dfs =
39     calcular_racionalidad_promedio(G_no_dirigido,
40     lista_adyacencia_dfs, iglesias_nodos,
41     distancia_optima, iteraciones)

```

- Se itera sobre cada nodo (iglesia) como punto de inicio potencial.
- Para cada nodo inicial, se realiza un recorrido DFS y se calcula la distancia total.
- Se actualiza la mejor ruta si se encuentra una distancia menor.
- Se construye una ruta completa utilizando los caminos más cortos entre nodos consecutivos.
- Se visualiza la mejor ruta encontrada en un gráfico.

- Se calcula la racionalidad promedio del algoritmo DFS mediante múltiples iteraciones con nodos iniciales aleatorios.

Resultado

```
Mejor nodo inicial: 262725526
Mejor ruta DFS del feligrés: [262725526, 262571000,
Distancia total recorrida con DFS: 35096.18 metros
```

Fig. 12. Output en consola.



Fig. 13. Trayectoria mínima con el algoritmo de Búsqueda en Profundidad (Depth-First Search).

```
Medida de Racionalidad Promedio del Algoritmo DFS (en 10 iteraciones): 2.15
```

Fig. 14. Media de Racionalidad Promedio del Algoritmo DFS.

H. Búsqueda en Amplitud (BFS)

Implementamos BFS para explorar las rutas en orden de proximidad, asegurando una búsqueda sistemática. El algoritmo de búsqueda en amplitud etiqueta todas las celdas (habitaciones), buscando la celda de final en todos sus vecinos adyacentes. Si no se llega a la celda, la búsqueda continúa hacia habitaciones adyacentes encontradas a partir de la habitación inicial; hasta que la celda final sea localizada. El algoritmo debe mantener la “ruta” de las habitaciones visitadas y que celdas son vecinos inmediatos desde la celda inicial, etiquetando cada celda con un número cada vez mayor a la celda por la que llegó. [3]

Declarativo

```
1 import networkx as nx
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4 from itertools import combinations
5 import random
6
7 def bfs(grafo, nodo_inicial):
8     visitados = set()
9     queue = [nodo_inicial]
10    recorrido = []
11    while queue:
12        nodo = queue.pop(0)
13        if nodo not in visitados:
14            visitados.add(nodo)
15            recorrido.append(nodo)
```

```
16         for vecino in grafo[nodo]:
17             if vecino not in visitados:
18                 queue.append(vecino)
19     return recorrido
20
21 def calcular_racionalidad_promedio_bfs(grafo,
22     lista_adyacencia, nodos_interes,
23     distancia_optima, iteraciones=10):
24     sum_racionalidad = 0
25     for _ in range(iteraciones):
26         nodo_inicial_aleatorio = random.choice(
27             nodos_interes)
28         ruta_bfs_aleatoria = bfs(lista_adyacencia,
29             nodo_inicial_aleatorio)
30         distancia_total_bfs_aleatoria = 0
31         for u, v in zip(ruta_bfs_aleatoria[:-1],
32             ruta_bfs_aleatoria[1:]):
33             try:
34                 distancia_total_bfs_aleatoria +=
35                     nx.shortest_path_length(grafo,
36                         u, v, weight='length')
37             except nx.NetworkXNoPath:
38                 continue
39         racionalidad_bfs =
40             distancia_total_bfs_aleatoria /
41             distancia_optima
42         sum_racionalidad += racionalidad_bfs
43     return sum_racionalidad / iteraciones
```

- Importaciones: Define las bibliotecas necesarias para el programa.
- Función bfs: Declara el algoritmo de Búsqueda en Anchura (BFS) para recorrer el grafo.
- Función calcular racionalidad promedio bfs: Define cómo calcular la racionalidad promedio del algoritmo BFS.

Procedural

```
1 # Crear lista de adyacencia para el grafo
2 procesado
3 lista_adyacencia_bfs = {nodo: set() for nodo in
4     iglesias_nodos}
5 for u, v in combinations(iglesias_nodos, 2):
6     try:
7         length = nx.shortest_path_length(
8             G_no_dirigido, u, v, weight='length')
9         lista_adyacencia_bfs[u].add(v)
10        lista_adyacencia_bfs[v].add(u)
11    except nx.NetworkXNoPath:
12        continue
13
14 # Inicializar variables para guardar el mejor
15 resultado
16 mejor_ruta = None
17 menor_distancia = float('inf')
18 mejor_nodo_inicial = None
19
20 # Probar cada nodo como nodo inicial
21 for nodo_inicial_bfs in iglesias_nodos:
22     ruta_bfs = bfs(lista_adyacencia_bfs,
23         nodo_inicial_bfs)
24     distancia_total_bfs = 0
25     for u, v in zip(ruta_bfs[:-1], ruta_bfs[1:]):
26         try:
27             distancia_total_bfs += nx.
28                 shortest_path_length(G_no_dirigido
29                     , u, v, weight='length')
30         except nx.NetworkXNoPath:
31             continue
32     if distancia_total_bfs < menor_distancia:
33         mejor_ruta = ruta_bfs
34         menor_distancia = distancia_total_bfs
35         mejor_nodo_inicial = nodo_inicial_bfs
36
37 # Visualizar la mejor ruta BFS en el grafo
38 ruta_completa_bfs = []
39 for u, v in zip(mejor_ruta[:-1], mejor_ruta[1:]):
40     try:
```

```

34     path = nx.shortest_path(G_no_dirigido, u,
35                             v, weight='length')
36     ruta_completa_bfs.extend(path[:-1])
37     except nx.NetworkXNoPath:
38         print(f"No se encontr un camino entre {u
39             } y {v}")
40     ruta_completa_bfs.append(mejor_ruta[-1])
41
42     print("Mejor nodo inicial:", mejor_nodo_inicial)
43     print("Mejor ruta BFS del feligrs:", mejor_ruta)
44     print(f"Distancia total recorrida con BFS: {
45         menor_distancia:.2f} metros")
46
47     fig, ax = ox.plot_graph_route(G_no_dirigido, route
48     =ruta_completa_bfs, route_linewidth=2,
49     node_size=0)
50     plt.show()
51
52     distancia_optima = 18050.39
53     iteraciones = 10
54     racionalidad_promedio_bfs =
55     calcular_racionalidad_promedio_bfs(
56     G_no_dirigido, lista_adyacencia_bfs,
57     iglesias_nodos, distancia_optima, iteraciones)
58     print(f"Medida de Racionalidad Promedio del
59     Algoritmo BFS (en {iteraciones} iteraciones):
60     {racionalidad_promedio_bfs:.2f}")

```

- Creación de lista de adyacencia: Construye la estructura de datos para representar las conexiones entre iglesias.
- Búsqueda de la mejor ruta: Itera sobre todos los nodos iniciales posibles, aplicando BFS y calculando distancias para encontrar la mejor ruta.
- Visualización de la ruta: Construye la ruta completa y la muestra en un mapa.
- Cálculo de racionalidad: Aplica la función de racionalidad promedio y muestra los resultados.

Resultado

```

Mejor nodo inicial: 262725526
Mejor ruta BFS del feligrs: [262725526, 6122446208,
Distancia total recorrida con BFS: 35757.46 metros

```

Fig. 15. Output en consola.



Fig. 16. Trayectoria mínima con el algoritmo de Búsqueda en Amplitud (Breadth-First Search).

Medida de Racionalidad Promedio del Algoritmo BFS (en 10 iteraciones): 2.18

Fig. 17. Media de Racionalidad Promedio del Algoritmo BFS.

I. Búsqueda Voraz (Greedy Best-First Search)

El algoritmo de búsqueda del mejor primero explora un espacio de búsqueda, comúnmente representado como un árbol, evaluando y seleccionando los nodos más prometedores según una función de evaluación. Este método prioriza la exploración de los nodos considerados "mejores" para alcanzar el objetivo de manera eficiente. En situaciones donde una exploración exhaustiva resulta computacionalmente costosa debido a la gran cantidad de nodos a analizar, se puede emplear una estrategia voraz (greedy) con un límite predefinido. Esta aproximación heurística examina solo un número limitado de caminos, posiblemente utilizando un factor de 2 veces la longitud actual como criterio de corte, y descarta el resto de las opciones sin evaluarlas [4].

Declarativo

```

1 import networkx as nx
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4 from itertools import combinations
5 import random
6
7 def encontrar_mas_cercano(nodo_actual,
8     nodos_restantes, lista_adyacencia):
9     min_dist = float('inf')
10    nodo_mas_cercano = None
11    for nodo in nodos_restantes:
12        if nodo in lista_adyacencia[nodo_actual]:
13            dist = lista_adyacencia[nodo_actual][
14                nodo]
15            if dist < min_dist:
16                min_dist = dist
17                nodo_mas_cercano = nodo
18    return nodo_mas_cercano, min_dist
19
20 def calcular_racionalidad_promedio_greedy(grafo,
21     lista_adyacencia, nodos_interes,
22     distancia_optima, iteraciones=10):
23     sum_racionalidad = 0
24     for _ in range(iteraciones):
25         nodo_inicial_aleatorio = random.choice(
26             nodos_interes)
27         nodo_actual = nodo_inicial_aleatorio
28         nodos_restantes = set(nodos_interes)
29         nodos_restantes.remove(nodo_actual)
30         ruta = [nodo_actual]
31         distancia_total = 0
32         while nodos_restantes:
33             nodo_mas_cercano, min_dist =
34             encontrar_mas_cercano(nodo_actual,
35                 nodos_restantes, lista_adyacencia
36             )
37             if nodo_mas_cercano is None:
38                 break
39             ruta.append(nodo_mas_cercano)
40             distancia_total += min_dist
41             nodo_actual = nodo_mas_cercano
42             nodos_restantes.remove(nodo_actual)
43             racionalidad_greedy = distancia_total /
44             distancia_optima
45             sum_racionalidad += racionalidad_greedy
46     return sum_racionalidad / iteraciones

```

- Función encontrar_mas_cercano:
 - Recibe el nodo actual, los nodos restantes y la lista de adyacencia.
 - Itera sobre los nodos restantes, buscando el más cercano al nodo actual.

- Utiliza la lista de adyacencia para obtener las distancias.
- Retorna el nodo más cercano y su distancia.
- Función `calcular_racionalidad_promedio_greedy`:
 - Esta función evalúa el rendimiento del algoritmo voraz.
 - Realiza múltiples iteraciones del algoritmo con puntos de inicio aleatorios.
 - En cada iteración:
 - * Selecciona un nodo inicial aleatorio.
 - * Construye una ruta usando el enfoque voraz.
 - * Calcula la distancia total de la ruta.
 - * Compara esta distancia con la distancia óptima conocida para obtener una medida de "racionalidad".
 - Promedia los resultados de todas las iteraciones.
 - Proporciona una métrica estadística de cuán cerca está el algoritmo voraz de la solución óptima.

Procedural

```

1 lista_adyacencia = {nodo: {} for nodo in
2   iglesias_nodos}
3 for u, v in combinations(iglesias_nodos, 2):
4     try:
5         length = nx.shortest_path_length(
6             G_no_dirigido, u, v, weight='length')
7         lista_adyacencia[u][v] = length
8         lista_adyacencia[v][u] = length
9     except nx.NetworkXNoPath:
10        continue
11
12 mejor_ruta = None
13 menor_distancia = float('inf')
14 mejor_nodo_inicial = None
15
16 for nodo_inicial_greedy in iglesias_nodos:
17     nodo_actual = nodo_inicial_greedy
18     nodos_restantes = set(iglesias_nodos)
19     nodos_restantes.remove(nodo_actual)
20     ruta = [nodo_actual]
21     distancia_total = 0
22
23     while nodos_restantes:
24         nodo_mas_cercano, min_dist =
25             encontrar_mas_cercano(nodo_actual,
26                                   nodos_restantes, lista_adyacencia)
27         if nodo_mas_cercano is None:
28             break
29         ruta.append(nodo_mas_cercano)
30         distancia_total += min_dist
31         nodo_actual = nodo_mas_cercano
32         nodos_restantes.remove(nodo_actual)
33
34     if distancia_total < menor_distancia:
35         mejor_ruta = ruta
36         menor_distancia = distancia_total
37         mejor_nodo_inicial = nodo_inicial_greedy
38
39 ruta_completa_greedy = []
40 for u, v in zip(mejor_ruta[:-1], mejor_ruta[1:]):
41     try:
42         path = nx.shortest_path(G_no_dirigido, u,
43                                 v, weight='length')
44         ruta_completa_greedy.extend(path[:-1])
45     except nx.NetworkXNoPath:
46         print(f"No se encontr un camino entre {u}
47             y {v}")
48
49 ruta_completa_greedy.append(mejor_ruta[-1])
50
51 print("Mejor nodo inicial:", mejor_nodo_inicial)
52 print("Mejor ruta de Búsqueda Voraz del feligrés",
53       mejor_ruta)
54 print(f"Distancia total recorrida con Búsqueda
55       Voraz: {menor_distancia:.2f} metros")

```

```

47 fig, ax = ox.plot_graph_route(G_no_dirigido, route
48                               =ruta_completa_greedy, route_linewidth=2,
49                               node_size=0)
50 plt.show()
51
52 distancia_optima = 18050.39
53 iteraciones = 10
54 racionalidad_promedio_greedy =
55     calcular_racionalidad_promedio_greedy(
56         G_no_dirigido, lista_adyacencia,
57         iglesias_nodos, distancia_optima, iteraciones)
58 print(f"Medida de Racionalidad Promedio del
59       Algoritmo Greedy (en {iteraciones} iteraciones
60       ): {racionalidad_promedio_greedy:.2f}")

```

- Creación de la lista de adyacencia:
 - Construye una estructura de datos que representa las distancias directas entre cada par de iglesias.
 - Utiliza el grafo de la ciudad para calcular estas distancias.
 - Maneja casos donde no exista un camino directo entre iglesias.
- Búsqueda de la mejor ruta:
 - Itera sobre cada iglesia como posible punto de inicio.
 - Para cada punto de inicio:
 - * Inicializa una ruta vacía y un conjunto de nodos restantes.
 - * Aplica repetidamente la función `encontrar_mas_cercano` para construir la ruta.
 - * Calcula la distancia total de la ruta.
 - Mantiene un registro de la mejor ruta encontrada (la de menor distancia total).

Resultado

```

Mejor nodo inicial: 114615160
Mejor ruta de Búsqueda Voraz del feligrés: [114615160, 1760085,
Distancia total recorrida con Búsqueda Voraz: 13653.34 metros

```

Fig. 18. Output en consola.



Fig. 19. Trayectoria mínima con el algoritmo de Búsqueda Voraz (Greedy Best-First Search).

Fig. 20. Media de Racionalidad Promedio del Algoritmo Greedy.

J. Algoritmo A*

Es un conocido algoritmo de planificación de rutas utilizado en la navegación de robots móviles, clasificado como un algoritmo de búsqueda heurística. Combina las ventajas del algoritmo de Dijkstra y la búsqueda heurística, utilizando una función de evaluación que suma el costo desde el inicio hasta el nodo actual y una estimación del costo desde el nodo actual hasta el objetivo, lo que permite encontrar caminos óptimos de manera eficiente en entornos estáticos conocidos. Sin embargo, tiene limitaciones como alta demanda de cálculo y tiempo en mapas grandes y baja robustez en entornos dinámicos. Para mejorar su eficiencia y robustez, se han desarrollado variantes como el EBHSA*, que incorpora búsquedas bidireccionales y funciones heurísticas optimizadas [5].

Declarativo

```
1 import networkx as nx
2 import osmnx as ox
3 import matplotlib.pyplot as plt
4 from itertools import combinations
5 import random
6
7 def encontrar_mas_cercano(nodo_actual,
8     nodos_restantes, lista_adyacencia):
9     min_dist = float('inf')
10    nodo_mas_cercano = None
11    for nodo in nodos_restantes:
12        if nodo in lista_adyacencia[nodo_actual]:
13            dist = lista_adyacencia[nodo_actual][
14                nodo]
15            if dist < min_dist:
16                min_dist = dist
17                nodo_mas_cercano = nodo
18    return nodo_mas_cercano, min_dist
19
20 def calcular_racionalidad_promedio_a_star(grafo,
21     lista_adyacencia, nodos_interes,
22     distancia_optima, iteraciones=10):
23     sum_racionalidad = 0
24     for _ in range(iteraciones):
25         nodo_inicial_aleatorio = random.choice(
26             nodos_interes)
27         nodo_actual = nodo_inicial_aleatorio
28         nodos_restantes = set(nodos_interes)
29         nodos_restantes.remove(nodo_actual)
30         ruta = [nodo_actual]
31         distancia_total = 0
32         while nodos_restantes:
33             nodo_mas_cercano, min_dist =
34                 encontrar_mas_cercano(nodo_actual,
35                     nodos_restantes, lista_adyacencia)
36             if nodo_mas_cercano is None:
37                 break
38             ruta.append(nodo_mas_cercano)
39             distancia_total += min_dist
40             nodo_actual = nodo_mas_cercano
41             nodos_restantes.remove(nodo_actual)
42         racionalidad_a_star = distancia_total /
43             distancia_optima
44         sum_racionalidad += racionalidad_a_star
45     return sum_racionalidad / iteraciones
```

- Define la función `encontrar_mas_cercano`, que busca el nodo más cercano al nodo actual entre los nodos restantes, utilizando la lista de adyacencia.

- Define la función `calcular_racionalidad_promedio_a_star`, que evalúa el rendimiento del algoritmo realizando múltiples iteraciones con puntos de inicio aleatorios. Esta función:
 - Selecciona un nodo inicial aleatorio.
 - Construye una ruta utilizando el enfoque del vecino más cercano.
 - Calcula la distancia total de la ruta.
 - Compara esta distancia con una distancia óptima conocida para obtener una medida de "racionalidad".
 - Repite este proceso varias veces y calcula el promedio de racionalidad.

Procedural

```
1 lista_adyacencia = {nodo: {} for nodo in
2     iglesias_nodos}
3 for u, v in combinations(iglesias_nodos, 2):
4     try:
5         length = nx.shortest_path_length(
6             G_no_dirigido, u, v, weight='length')
7         lista_adyacencia[u][v] = length
8         lista_adyacencia[v][u] = length
9     except nx.NetworkXNoPath:
10        continue
11
12 mejor_ruta = None
13 menor_distancia = float('inf')
14 mejor_nodo_inicial = None
15
16 for nodo_inicial_a_star in iglesias_nodos:
17     nodo_actual = nodo_inicial_a_star
18     nodos_restantes = set(iglesias_nodos)
19     nodos_restantes.remove(nodo_actual)
20     ruta = [nodo_actual]
21     distancia_total = 0
22     while nodos_restantes:
23         nodo_mas_cercano, min_dist =
24             encontrar_mas_cercano(nodo_actual,
25                 nodos_restantes, lista_adyacencia)
26         if nodo_mas_cercano is None:
27             break
28         ruta.append(nodo_mas_cercano)
29         distancia_total += min_dist
30         nodo_actual = nodo_mas_cercano
31         nodos_restantes.remove(nodo_actual)
32     if distancia_total < menor_distancia:
33         mejor_ruta = ruta
34         menor_distancia = distancia_total
35         mejor_nodo_inicial = nodo_inicial_a_star
36
37 ruta_completa_a_star = []
38 for u, v in zip(mejor_ruta[:-1], mejor_ruta[1:]):
39     try:
40         path = nx.shortest_path(G_no_dirigido, u,
41             v, weight='length')
42         ruta_completa_a_star.extend(path[:-1])
43     except nx.NetworkXNoPath:
44         print(f"No se encontr un camino entre {u}
45             y {v}")
46 ruta_completa_a_star.append(mejor_ruta[-1])
47
48 print("Mejor nodo inicial:", mejor_nodo_inicial)
49 print("Mejor ruta A* del feligrs:", mejor_ruta)
50 print(f"Distancia total recorrida con A*: {
51     menor_distancia:.2f} metros")
52
53 fig, ax = ox.plot_graph_route(G_no_dirigido, route
54     =ruta_completa_a_star, route_linewidth=2,
55     node_size=0)
56 plt.show()
57
58 distancia_optima = 18050.39
59 iteraciones = 10
60 racionalidad_promedio_a_star =
61     calcular_racionalidad_promedio_a_star(
```



```

G_no_dirigido, lista_adyacencia,
iglesias_nodos, distancia_optima, iteraciones)
52 print(f"Medida de Racionalidad Promedio del
Algoritmo A* (en {iteraciones} iteraciones): {
racionalidad_promedio_a_star:.2f}")

```

- Crea la lista de adyacencia que almacena las distancias entre pares de iglesias.
- Implementa el algoritmo principal que busca la mejor ruta:
 - Itera sobre cada iglesia como punto de inicio.
 - Para cada punto de inicio, construye una ruta utilizando el enfoque del vecino más cercano.
 - Mantiene un registro de la mejor ruta encontrada (la de menor distancia total).
- Construye la ruta completa, incluyendo los nodos intermedios entre iglesias.
- Muestra la mejor ruta encontrada y la distancia total recorrida.
- Visualiza la ruta en un mapa de la ciudad utilizando osmnx.
- Calcula y muestra la racionalidad promedio del algoritmo, comparando su rendimiento con una distancia óptima conocida.

Resultado

```

Mejor nodo inicial: 114615160
Mejor ruta A* del feligrés: [114615160, 1760085377
Distancia total recorrida con A*: 13653.34 metros

```

Fig. 21. Output en consola.



Fig. 22. Trayectoria mínima con el algoritmo A* (A-Star).

```

Medida de Racionalidad Promedio del Algoritmo A* (en 10 iteraciones): 0.87

```

Fig. 23. Media de Racionalidad Promedio del Algoritmo A* (A-Star).

K. Algoritmo Búsqueda de costo uniforme (Uniform Cost Search)

Es una variante del algoritmo de búsqueda de amplitud que expande el nodo con el costo acumulado más bajo desde

el nodo inicial en lugar de expandir todos los nodos en el mismo nivel primero. Este algoritmo es óptimo y completo siempre que todos los costos de las aristas sean no negativos. Es especialmente útil en problemas de optimización de rutas donde se necesita encontrar el camino de menor costo entre dos puntos [6].

Declarativo

```

1 # Definición del problema
2 place = "Miraflores, Lima, Peru"
3 graph = create_street_graph(place)
4 churches = find_churches(place, graph)
5 # Solución del problema
6 best_route = find_optimal_route(graph, churches)
7 # Visualización y análisis
8 map = create_map(graph, churches, best_route)
9 rationality = calculate_average_rationality(graph,
    churches, best_route)
10 # Presentación de resultados
11 display(map)
12 print(f"Mejor ruta: {best_route}")
13 print(f"Racionalidad promedio: {rationality}")

```

Este código:

- Define el lugar de interés como Miraflores, Lima, Perú.
- Crea un grafo de calles para este lugar utilizando una función hipotética `create_street_graph()`.
- Encuentra las iglesias en el área y las mapea al grafo usando `find_churches()`.
- Encuentra la ruta óptima entre las iglesias utilizando `find_optimal_route()`.
- Crea un mapa visual que muestra el grafo, las iglesias y la mejor ruta con `create_map()`.
- Calcula la racionalidad promedio del algoritmo usado con `calculate_average_rationality()`.
- Muestra el mapa resultante y imprime información sobre la mejor ruta y la racionalidad promedio.

Procedural

```

1 # Paso 1: Configuración inicial
2 import osmnx as ox
3 import networkx as nx
4 import matplotlib.pyplot as plt
5
6 # Paso 2: Obtener datos del mapa
7 place_name = "Miraflores, Lima, Peru"
8 G = ox.graph_from_place(place_name, network_type="
    drive")
9
10 # Paso 3: Preprocesar el grafo
11 G_undirected = G.to_undirected()
12 isolated_nodes = list(nx.isolates(G_undirected))
13 dead_ends = [node for node, degree in dict(
    G_undirected.degree()).items() if degree == 1]
14 G_undirected.remove_nodes_from(isolated_nodes +
    dead_ends)
15 largest_cc = max(nx.connected_components(
    G_undirected), key=len)
16 G_undirected = G_undirected.subgraph(largest_cc).
    copy()
17
18 # Paso 4: Encontrar iglesias
19 tags = {'amenity': 'place_of_worship', 'religion':
    'christian'}
20 points = ox.features_from_place(place_name, tags)
21 points['geometry'] = points.apply(lambda row: row[
    'geometry'].centroid if row['geometry'].
    geom_type == 'Polygon' else row['geometry'],
    axis=1)
22 gdf_nodes, gdf_edges = ox.graph_to_gdfs(
    G_undirected)
23 graph_bounds = gdf_nodes.unary_union.bounds
24 points = points.cx[graph_bounds[0]:graph_bounds
    [2], graph_bounds[1]:graph_bounds[3]]

```



```

25 church_nodes = ox.distance.nearest_nodes(
    G_undirected, points.geometry.x, points.
    geometry.y)
26
27 # Paso 5: Implementar búsqueda de costo uniforme
28 G_directed = G_undirected.to_directed()
29 G_directed = add_edge_lengths(G_directed)
30 best_path, best_cost, best_start = find_best_route
    (G_directed, church_nodes)
31
32 # Paso 6: Visualizar resultados
33 fig, ax = ox.plot_graph(G_undirected, show=False,
    close=False, edge_color='gray', edge_linewidth
    =0.5, node_size=0)
34 points.plot(ax=ax, color='red', markersize=50,
    zorder=3)
35 if best_path:
36     route_coords = [(G_directed.nodes[node]['x'],
    G_directed.nodes[node]['y']) for node in
    best_path]
37     route_x, route_y = zip(*route_coords)
38     ax.plot(route_x, route_y, color='blue',
    linewidth=2, zorder=2)
39     ax.plot(route_x[0], route_y[0], 'go',
    markersize=10, label='Inicio')
40     ax.plot(route_x[-1], route_y[-1], 'yo',
    markersize=10, label='Fin')
41 plt.title(f"Iglesias y ruta ptima en {place_name
    }")
42 plt.legend()
43 plt.tight_layout()
44 plt.show()
45
46 # Paso 7: Calcular racionalidad
47 optimal_distance = 18050.39
48 average_rationality =
    calcular_racionalidad_promedio_uniform_cost(
    G_directed, church_nodes, optimal_distance)
49 print(f"Medida de Racionalidad Promedio: {
    average_rationality:.2f}")

```

Este código:

- Importa las bibliotecas necesarias: `osmnx`, `networkx` y `matplotlib`.
- Define el lugar de interés y descarga el grafo de calles usando `osmnx`.
- Preprocesa el grafo:
 - Lo convierte a no dirigido.
 - Elimina nodos aislados y calles sin salida.
 - Mantiene solo el componente conectado más grande.
- Encuentra las iglesias:
 - Busca lugares de culto cristianos usando `osmnx`.
 - Convierte polígonos a puntos usando centroides.
 - Filtra puntos dentro de los límites del grafo.
 - Mapea estos puntos a los nodos más cercanos en el grafo.
- Prepara el grafo para la búsqueda:
 - Lo convierte a dirigido.
 - Añade longitudes a las aristas.
 - Ejecuta el algoritmo de búsqueda para encontrar la mejor ruta.
- Visualiza los resultados:
 - Dibuja el grafo de calles.
 - Marca las iglesias.
 - Traza la ruta óptima si se encontró.
- Calcula la racionalidad del algoritmo:
 - Usa una distancia óptima conocida.
 - Calcula la racionalidad promedio usando una función definida previamente.

3. Resultado:

```

Mejor ruta encontrada (iniciando desde el nodo 262725526):
[262725526, 262725525, 262725514, 262725511, 262725484, 2627
Costo total (distancia): 14595.86 metros

```

Fig. 24. Output en consola.



Fig. 25. Trayectoria mínima con el algoritmo de Búsqueda de costo uniforme (Uniform Cost Search).

```

Medida de Racionalidad Promedio del Algoritmo de Costo Uniforme (en 10 iteraciones): 0.87

```

Fig. 26. Media de Racionalidad Promedio del Algoritmo (Uniform Cost Search).

X. CONCLUSIONES

La medida de racionalidad evaluada en algunos algoritmos como Voraz, A* y Uniform Cost Search prueban que son una estrategia mejor que la estimada por el método Ávaro. Entonces, los peregrinos pueden librarse de la penitencia.

Los algoritmos Voraz, A* y Uniform Cost Search demostraron ser más eficientes en términos de distancia recorrida en comparación con el método Ávaro. Estos algoritmos informados, que utilizan heurísticas y optimización de costos, lograron encontrar rutas más cortas y eficientes para visitar todas las iglesias del distrito de Miraflores.

La medida de racionalidad promedio calculada para los algoritmos Voraz, A* y Uniform Cost Search se acercó mucho a la distancia óptima, indicando que estos algoritmos son capaces de aproximarse a la solución óptima de manera consistente. Esto valida su eficacia en la planificación de rutas óptimas en un entorno urbano.

Este estudio contribuye al campo de la inteligencia artificial y la optimización al demostrar la aplicabilidad de algoritmos de búsqueda informados en problemas de

planificación de rutas. Los resultados obtenidos pueden servir como base para desarrollos futuros y aplicaciones prácticas en diversos contextos, como la logística urbana y la gestión de redes de transporte.

XI. RECOMENDACIONES

Los algoritmos de búsqueda informados como A* dependen en gran medida de la heurística utilizada. Se recomienda explorar y probar diferentes heurísticas que puedan mejorar la precisión y eficiencia del algoritmo en el contexto específico de la planificación de rutas en áreas urbanas.

Considerar la implementación de algoritmos híbridos que combinen las fortalezas de diferentes algoritmos de búsqueda. Por ejemplo, una combinación de A* con Algoritmos Genéticos podría proporcionar soluciones óptimas al balancear la exploración y la explotación.

El proceso de preprocesamiento del grafo puede ser optimizado eliminando nodos y aristas redundantes de manera más eficiente. Además, se puede considerar la aplicación de técnicas de simplificación de grafos para reducir la complejidad computacional.

Los algoritmos deben ser probados y adaptados para funcionar eficientemente en áreas geográficas más grandes. Se recomienda llevar a cabo estudios de escalabilidad y ajustar los parámetros de los algoritmos para mantener el rendimiento en diferentes escalas.

Incluir factores dinámicos como el tráfico en tiempo real y las condiciones climáticas en los modelos de búsqueda puede hacer que las rutas sugeridas sean más prácticas y útiles para los usuarios finales. Integrar datos en tiempo real a través de APIs puede mejorar significativamente la utilidad del sistema.

Además de la distancia, otros criterios de optimización como el tiempo de viaje, el costo del transporte y la seguridad de las rutas pueden ser considerados. Desarrollar algoritmos multi-objetivo que optimicen múltiples criterios simultáneamente puede proporcionar soluciones más completas.

Validar las rutas encontradas por los algoritmos utilizando datos reales de desplazamientos puede ayudar a evaluar la efectividad práctica de los algoritmos. Colaborar con autoridades locales y utilizar datos de sensores y dispositivos móviles puede proporcionar insights valiosos.

XII. REFERENCIAS

[1] Boeing, G. (2017). OSMnx: A Python package to work with graph-theoretic OpenStreetMap street networks. *Journal of Open Source Software*, 2(12), 605. [En línea]. Disponible: <https://doi.org/10.21105/joss.00215>

[2] J. Rivero Espinosa, "Búsqueda Rápida de Caminos en Grafos de Alta Cardinalidad Estáticos y Dinámicos," Tesis doctoral, Dept. Informática, Univ. Carlos III de Madrid, Leganés, España, 2011. [En línea]. Disponible: <https://hdl.handle.net/10016/14751>

[3] V. Mariano, F. Cárdenas, y E. Hernández, Análisis de algoritmos de búsqueda en espacio de estados, *Ciencia Huasteca Boletín Científico de la Escuela Superior de Huejutla*, vol. 3, ene. 2015, doi: [En línea]. Disponible: <https://doi.org/10.29057/esh.v3i5.1089>

[4] C. Frăsinaru and M. Răschip, "Greedy Best-First Search for the Optimal-Size Sorting Network Problem," *Procedia Computer Science*, vol. 159, pp. 447-454, 2019. [En línea]. Disponible: <https://www.sciencedirect.com/science/article/pii/S187705091931381X>

[5] S. Lou, J. Jing, H. He, y W. Liu, "An Efficient and Robust Improved A* Algorithm for Path Planning," *Symmetry*, vol. 13, no. 11, p. 2213, Nov. 2021. [En línea]. Disponible: <https://doi.org/10.3390/sym13112213>

[6] Papadouli, V., Papakonstantinou, V. (2023). A preliminary study on artificial intelligence oracles and smart contracts: A legal approach to the interaction of two novel technological breakthroughs. *Computer Law And Security Report/Computer Law Security Report*, 51, 105869. [En línea]. Disponible: <https://doi.org/10.1016/j.clsr.2023.105869>

Anexo

Se adjunta el link del drive donde se encuentra el código principal de nuestro trabajo.

Link: <https://drive.google.com/drive/folders/1IaA3hfjwSSDfrbBaeAnioLswY1jiIQJo>