Búsqueda en Grafos: BFS y DFS aplicados al Metro CDMX

Inteligencia Artificial

Nombre: **Diego Coronado Perez**
Grupo: **10**

Facultad de Ingeniería – UNAM
Fecha de entrega: 20 de febrero de 2026

Índice

1. Introducción

El objetivo de este reporte es presentar la implementación y el análisis de dos algoritmos fundamentales de Inteligencia Artificial enfocados en la búsqueda no informada en grafos: la **Búsqueda en Anchura (BFS)** y la **Búsqueda en Profundidad (DFS)**. El dominio de aplicación seleccionado para este estudio es el sistema de transporte colectivo Metro de la Ciudad de México.

El problema central consiste en encontrar una ruta válida entre una estación de origen y una de destino. Para lograrlo, la red del Metro se abstrae y se modela como un grafo matemático. Esta representación permite evaluar el comportamiento de los algoritmos en un escenario real, analizando sus propiedades teóricas fundamentales como la completitud, la optimalidad y su complejidad computacional.

2. Modelado del Grafo

El Metro CDMX se representa formalmente como un grafo $G = (V, E)$ con las siguientes características:

- **Nodos (V):** Representan cada una de las estaciones del metro.
- **Aristas (E):** Representan las vías o conexiones directas físicas entre dos estaciones adyacentes.
- **No dirigido:** Se asume que los trenes viajan en ambos sentidos de la vía, por lo que las conexiones son bidireccionales.
- **No ponderado:** Para fines de esta implementación, se ignora la distancia física o el tiempo de traslado. Cada arista tiene un costo uniforme exacto de 1.

Este modelo permite medir la distancia entre estaciones estrictamente mediante el número de transiciones o "saltos". Para ilustrar esta abstracción, la Figura 1 muestra la topología de la red. Visualmente, cada punto de parada o transbordo en el mapa representa un **nodo** del grafo, mientras que los segmentos de vía físicos que unen dos estaciones consecutivas representan las **aristas**.

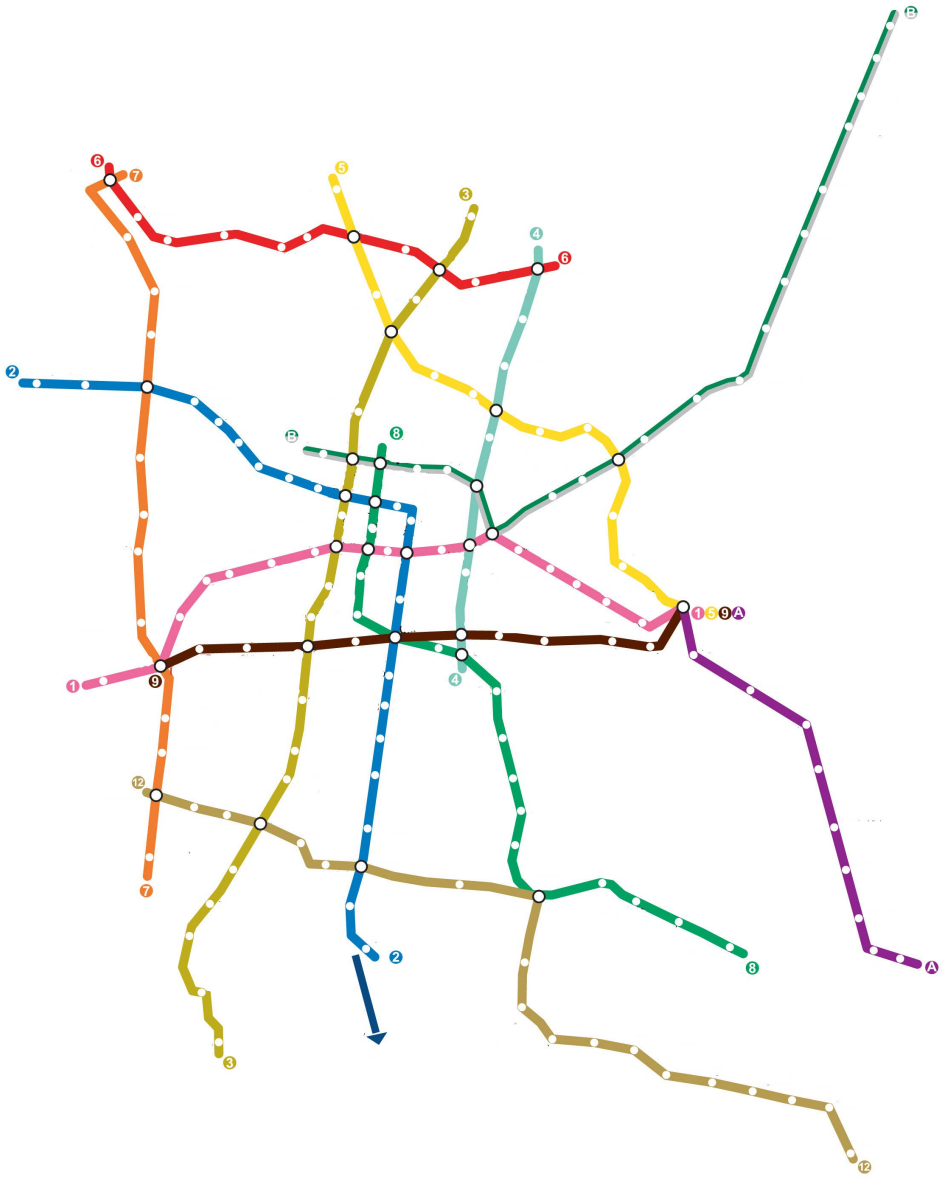

Figura 1: Representación visual de la red del Metro CDMX, donde las estaciones actúan como nodos y las vías como aristas.

Esta correspondencia visual se traduce de manera directa a nuestra implementación computacional en Python. En el código, la red se construye instanciando una clase base que gestiona esta conectividad. Cada estación (el nodo visual) se convierte en una llave dentro de un diccionario, y las conexiones directas que se observan en el mapa (las aristas) se agregan a una lista asociada a esa llave, tal como se muestra a continuación:

Listing 1: Fragmento del modelado de la clase Grafo

```
class MetroCDMX:
    def __init__(self):
        self.grafo = {}
        self._construir_grafo()

    def agregar_arista(self, estacion1, estacion2):
        """Agrega una arista bidireccional entre dos estaciones"""
        if estacion1 not in self.grafo:
            self.grafo[estacion1] = []
```

```
        if estacion2 not in self.grafo:
            self.grafo[estacion2] = []

        if estacion2 not in self.grafo[estacion1]:
            self.grafo[estacion1].append(estacion2)
        if estacion1 not in self.grafo[estacion2]:
            self.grafo[estacion2].append(estacion1)
```

3. Representación de Adyacencia

Para almacenar la estructura del grafo en memoria, se optó por utilizar una **lista de adyacencia**, implementada de forma nativa en Python mediante un diccionario (`dict`), donde las llaves son los nombres de las estaciones y los valores son listas (`list`) que contienen a sus estaciones vecinas.

3.1. Ventajas y Desventajas

- **Ventajas:** Es una estructura altamente eficiente en memoria, ocupando un espacio proporcional a $O(|V| + |E|)$. En grafos dispersos como el Metro (donde el número de conexiones por estación es pequeño, promediando de 2 a 4), evita el desperdicio masivo de memoria que generaría una matriz de adyacencia llena de ceros.
- **Desventajas:** Comprobar si existe una conexión específica entre dos estaciones cualesquiera toma tiempo lineal $O(k)$ respecto al número de vecinos de la estación, en lugar de tiempo constante $O(1)$ como en una matriz. No obstante, para los recorridos BFS y DFS, esto no es un problema, ya que los algoritmos iteran directamente sobre los vecinos existentes.

3.2. Construcción del grafo

El siguiente fragmento muestra cómo se puebla el modelo utilizando la estructura elegida:

Listing 2: Fragmento de construcción del grafo poblado con la Línea 1

```
linea1 = [
            "Observatorio", "Tacubaya", "Juanacatlan", "Chapultepec",
            "Sevilla", "Insurgentes", "Cuauhtemoc", "Balderas",
            "Salto del Agua", "Isabel la Catolica", "Pino Suarez",
            "Merced", "Candelaria", "San Lazaro", "Moctezuma",
            "Balbuena", "Boulevard Puerto Aereo", "Gomez Farias",
            "Zaragoza", "Pantitlan"
        ]

# En cada vuelta del ciclo, se toma la estacion actual y se une con la
    adyacente
for i in range(len(linea1) - 1):
    mi_grafo.agregar_arista(linea1[i], linea1[i + 1])
```

4. Algoritmos Implementados

4.1. Búsqueda en Anchura (BFS)

La búsqueda en anchura explora el grafo por niveles utilizando una estructura de datos tipo cola (FIFO: *First-In, First-Out*).

Listing 3: Implementación de la Búsqueda en Anchura (BFS)

```
from collections import deque

def bfs(self, origen, destino):
        """
        Busqueda en Anchura (BFS - Breadth-First Search)

        Retorna:
            - ruta: lista ordenada de estaciones desde origen hasta
               destino
            - saltos: numero de aristas (k-1 donde k es longitud de ruta
               )
            - nodos_expandidos: numero de nodos visitados durante la
               busqueda
        """
        if origen not in self.grafo:
            return None, 0, 0, f"Estacion origen '{origen}' no existe"
        if destino not in self.grafo:
            return None, 0, 0, f"Estacion destino '{destino}' no existe"

        if origen == destino:
            return [origen], 0, 1, "Origen y destino son iguales"

        # Estructuras de datos para BFS
        cola = deque([origen])
        visitados = {origen}
        padres = {origen: None}
        nodos_expandidos = 0

        encontrado = False

        while cola and not encontrado:
            actual = cola.popleft()
            nodos_expandidos += 1

            # Explorar vecinos
            for vecino in self.grafo[actual]:
                if vecino not in visitados:
                    visitados.add(vecino)
                    padres[vecino] = actual
                    cola.append(vecino)

                    if vecino == destino:
                        encontrado = True
                        break

        # Reconstruir ruta
        if destino not in padres:
            return None, 0, nodos_expandidos, "No existe ruta"

        ruta = []
        actual = destino
        while actual is not None:
            ruta.append(actual)
            actual = padres[actual]
        ruta.reverse()

        saltos = len(ruta) - 1
```

```
        return ruta, saltos, nodos_expandidos, "Exito"
```

El uso de una cola garantiza que BFS explore primero a todos los vecinos inmediatos antes de avanzar a los vecinos de segundo grado, lo que asegura encontrar el camino más corto en términos de aristas.

4.2. Búsqueda en Profundidad (DFS)

La búsqueda en profundidad (DFS) explora una rama del grafo tan profundo como sea posible antes de retroceder. Se apoya en una estructura de pila (LIFO: *Last-In, First-Out*).

Listing 4: Implementación de la Búsqueda en Profundidad (DFS)

```
def dfs(self, origen, destino):
        """
        Busqueda en Profundidad (DFS - Depth-First Search)

        Retorna:
            - ruta: lista ordenada de estaciones desde origen hasta
               destino
            - saltos: numero de aristas (k-1 donde k es longitud de ruta
               )
            - nodos_expandidos: numero de nodos visitados durante la
               busqueda
        """
        if origen not in self.grafo:
            return None, 0, 0, f"Estacion origen '{origen}' no existe"
        if destino not in self.grafo:
            return None, 0, 0, f"Estacion destino '{destino}' no existe"

        if origen == destino:
            return [origen], 0, 1, "Origen y destino son iguales"

        # Estructuras de datos para DFS
        pila = [origen]
        visitados = {origen}
        padres = {origen: None}
        nodos_expandidos = 0

        encontrado = False

        # Busqueda DFS
        while pila and not encontrado:
            actual = pila.pop()
            nodos_expandidos += 1

            # Explorar vecinos
            for vecino in self.grafo[actual]:
                if vecino not in visitados:
                    visitados.add(vecino)
                    padres[vecino] = actual
                    pila.append(vecino)

                    if vecino == destino:
                        encontrado = True
                        break

        # Reconstruir ruta
```

```
        if destino not in padres:
            return None, 0, nodos_expandidos, "No existe ruta"

        ruta = []
        actual = destino
        while actual is not None:
            ruta.append(actual)
            actual = padres[actual]
        ruta.reverse()

        saltos = len(ruta) - 1

        return ruta, saltos, nodos_expandidos, "Exito"
```

DFS no garantiza rutas óptimas, ya que su recorrido depende íntegramente del orden en el que se insertan y extraen los vecinos en la pila.

5. Propiedades Teóricas

5.1. Completitud

Un algoritmo de búsqueda es completo si garantiza encontrar una solución siempre que esta exista.

BFS es completo en el contexto del Metro CDMX debido a que el grafo que representa la red de transporte es finito. Al explorar el grafo por niveles, BFS asegura que todas las estaciones alcanzables desde el origen serán eventualmente visitadas, asumiendo que el grafo es conexo (o al menos que el destino está en la misma componente conexa que el origen).

DFS, por su naturaleza, puede no ser completo en árboles de búsqueda infinitos o si no se controla la presencia de ciclos infinitos. Sin embargo, dado que nuestra implementación mantiene un conjunto en memoria de nodos `visitados`, se evitan los ciclos. Al tratarse de un número finito de estaciones, DFS también es completo para este modelado del Metro.

5.2. Optimalidad

Un algoritmo es óptimo si garantiza encontrar la mejor solución posible (aquella con el menor costo de ruta).

BFS es óptimo en grafos no ponderados. Como explora radialmente en capas equivalentes a k saltos de distancia, la primera vez que BFS inserta el nodo `objetivo` en los visitados, lo hace a través del camino que requiere el menor número de saltos.

Por el contrario, **DFS** no es optimo. DFS puede encontrar una solución válida siguiendo una línea que da un rodeo innecesario por la ciudad antes de llegar a un destino que bien podría haber estado a solo dos saltos de distancia, simplemente porque esa fue la primera rama que la pila decidió explorar en profundidad.

6. Complejidad Computacional

La complejidad se evalúa considerando factores del árbol de búsqueda: b (factor de ramificación máximo), d (profundidad del nodo objetivo más superficial) y m (longitud máxima de cualquier camino en el espacio de estados). Asimismo, también se analiza bajo la notación clásica de grafos con $|V|$ (vértices) y $|E|$ (aristas).

6.1. Complejidad en Tiempo

- **BFS:** En el peor de los casos, recorre la totalidad del árbol de búsqueda expandiendo cada nivel, lo que resulta en un tiempo de $O(b^d)$. Visto desde la teoría de grafos, como utilizamos listas de adyacencia y un control de nodos visitados, se inspecciona cada vértice y cada arista una vez, logrando $O(|V| + |E|)$.

- **DFS:** Al explorar en profundidad, si la solución está a la derecha del árbol pero iniciamos por la izquierda, podemos llegar a recorrer todo el grafo, dándonos un tiempo de $O(b^m)$, o de igual manera $O(|V| + |E|)$ gracias al conjunto de `visitados`.

Ambos logran tiempos casi instantáneos en la práctica para la red del Metro CDMX (aproximadamente 195 estaciones).

6.2. Complejidad en Espacio

- **BFS:** El cuello de botella de BFS es la memoria. Debe mantener en la cola todos los nodos de la "frontera"del nivel actual para expandir el siguiente. Esto es $O(b^d)$. En notación de grafos requiere $O(|V|)$ para almacenar el diccionario de `padres` y los `visitados`.

- **DFS:** Su ventaja teórica es el ahorro de espacio. La pila de recursión (o pila iterativa) solo necesita almacenar un solo camino desde la raíz hasta la hoja actual, más los nodos hermanos no expandidos, ocupando $O(b \cdot m)$. Sin embargo, al guardar el conjunto completo de `visitados` para evitar ciclos en este grafo específico, terminamos con una complejidad espacial que también ronda los $O(|V|)$.

7. Resultados Experimentales

7.1. Métricas

Listing 5: Cálculo de saltos y nodos visitados

```
ruta = reconstruir_ruta(padres, inicio, objetivo)
saltos = len(ruta) - 1
nodos_visitados = len(visitados)
```

7.2. Casos de Prueba Obligatorios

En esta sección se presentan los resultados obtenidos al ejecutar los algoritmos BFS y DFS sobre el grafo del Metro CDMX, utilizando exactamente los mismos casos de prueba solicitados en la tarea. La información mostrada corresponde directamente a la salida en consola del programa desarrollado.

7.2.1. Caso 1: Observatorio → Ciudad Azteca

Listing 6: Salida en consola – Caso 1: Observatorio ->Ciudad Azteca

```
ALGORITMO: BFS (Busqueda en Anchura)
Origen: Observatorio
Destino: Ciudad Azteca
Estado: Exito

METRICAS:
- Longitud en aristas (saltos): 24
- Numero de estaciones en la ruta: 25
- Nodos expandidos/visitados: 162
- Costo total (peso uniforme): 24

ALGORITMO: DFS (BUsqueda en Profundidad)
Origen: Observatorio
Destino: Ciudad Azteca
Estado: Exito

METRICAS:
- Longitud en aristas (saltos): 26
- Numero de estaciones en la ruta: 27
- Nodos expandidos/visitados: 35
- Costo total (peso uniforme): 26
```

Análisis del Caso 1: BFS encontró la ruta más corta en términos de número de estaciones, mientras que DFS expandió un menor número de nodos debido a su exploración en profundidad.

7.2.2. Caso 2: Indios Verdes → Velódromo

Listing 7: Salida en consola – Caso 2: Indios Verdes ->Velódromo

```
ALGORITMO: BFS (Busqueda en Anchura)
Origen: Indios Verdes
Destino: Velodromo
Estado: Exito

METRICAS:
- Longitud en aristas (saltos): 13
- NUmero de estaciones en la ruta: 14
- Nodos expandidos/visitados: 79
- Costo total (peso uniforme): 13

ALGORITMO: DFS (Busqueda en Profundidad)
Origen: Indios Verdes
Destino: Velodromo
Estado: Exito

METRICAS:
- Longitud en aristas (saltos): 28
- Numero de estaciones en la ruta: 29
- Nodos expandidos/visitados: 40
- Costo total (peso uniforme): 28
```

Análisis del Caso 2: La diferencia entre BFS y DFS es más notable. BFS obtiene una ruta mucho más corta, mientras que DFS realiza un recorrido considerablemente más largo antes de alcanzar el destino.

7.2.3. Caso 3: UAM-I → El Rosario

Listing 8: Salida en consola – Caso 3: UAM-I ->El Rosario

```
ALGORITMO: BFS (Busqueda en Anchura)
Origen: UAM-I
Destino: El Rosario
Estado: Exito

METRICAS:
- Longitud en aristas (saltos): 23
- Numero de estaciones en la ruta: 24
- Nodos expandidos/visitados: 152
- Costo total (peso uniforme): 23

ALGORITMO: DFS (B squeda en Profundidad)
Origen: UAM-I
Destino: El Rosario
Estado: Exito

METRICAS:
- Longitud en aristas (saltos): 48
- Numero de estaciones en la ruta: 49
- Nodos expandidos/visitados: 100
- Costo total (peso uniforme): 48
```

Análisis del Caso 3: BFS vuelve a garantizar la solución óptima en términos de saltos, mientras que DFS recorre una gran parte de la red antes de llegar al destino.

Figura 2: Enter Caption

7.3. Resumen Comparativo Global

Cuadro 1: Resumen comparativo BFS vs DFS en todos los casos

Caso	Saltos BFS	Saltos DFS	Nodos BFS	Nodos DFS
Observatorio → Ciudad Azteca	24	26	162	35
Indios Verdes → Velódromo	13	28	79	40
UAM-I → El Rosario	23	48	152	100

7.4. Evidencia

Figura 3: BFS (OBSERVATORIO - CD. AZTECA)

Figura 4: DFS (OBSERVATORIO - CD. AZTECA)

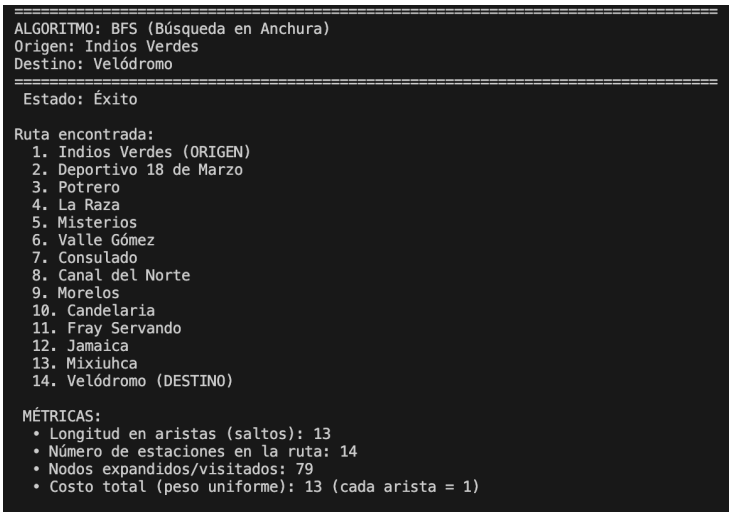

Figura 5: BFS (INDIOS VERDES - VELODROMO)

Figura 6: DFS (INDIOS VERDES - VELODROMO)

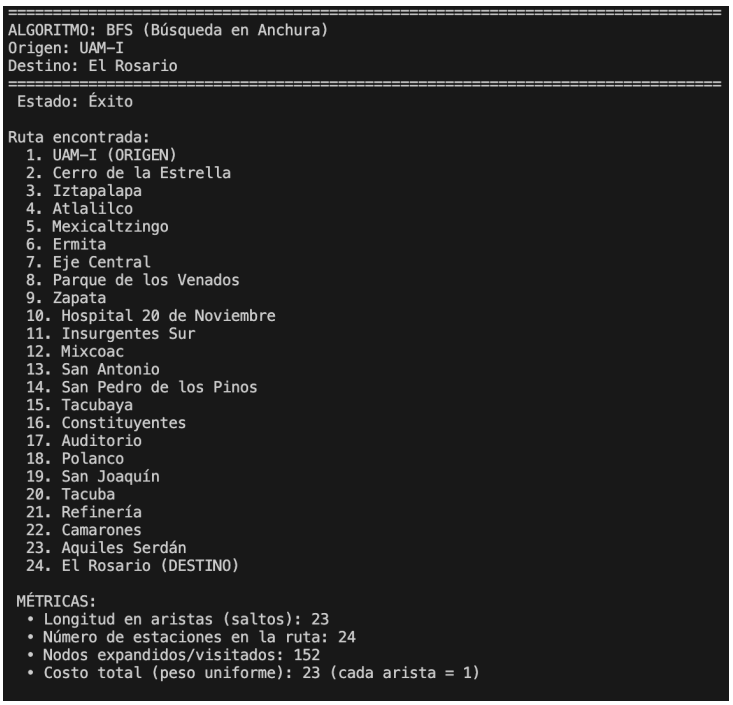

Figura 7: BFS (UAM-I - El Rosario)

Figura 8: DFS (UAM-I - El Rosario)

8. Conclusiones

El desarrollo de esta tarea permitió comprender de manera profunda cómo los algoritmos de búsqueda en grafos se comportan cuando se aplican a un problema real y de gran escala como la red del Metro de la Ciudad de México. La modelación del sistema de transporte como un grafo no dirigido y no ponderado resultó adecuada para analizar rutas en términos del número de estaciones recorridas.

Los resultados experimentales confirman que la **Búsqueda en Anchura (BFS)** es la estrategia más apropiada para problemas de navegación en grafos no ponderados, ya que garantiza la obtención de la ruta con el menor número de saltos. En todos los casos evaluados, BFS encontró consistentemente rutas más cortas, aunque a costa de expandir un mayor número de nodos, lo cual refleja un mayor consumo de memoria y tiempo de exploración.

Por otro lado, la **Búsqueda en Profundidad (DFS)** mostró un comportamiento menos predecible. Si bien suele expandir menos nodos y requiere menor memoria en promedio, las rutas obtenidas son altamente dependientes del orden de exploración y, en la mayoría de los casos, resultan considerablemente más largas que las encontradas por BFS. Esto evidencia que DFS no es adecuada para encontrar rutas óptimas en sistemas de transporte, aunque puede ser útil para tareas de exploración o análisis estructural del grafo.

Asimismo, el uso de una lista de adyacencia mediante diccionarios en Python demostró ser una representación eficiente y clara para grafos dispersos como el del Metro CDMX, facilitando tanto la implementación de los algoritmos como el análisis de sus propiedades teóricas. La correspondencia directa entre la salida en consola y los resultados presentados en el reporte refuerza la validez del análisis realizado.

Finalmente, la realizacion de esta tarea evidencia que la elección del algoritmo de búsqueda depende directamente del objetivo del problema. Mientras que BFS es ideal para minimizar el número de estaciones recorridas, una extensión natural de este trabajo sería incorporar pesos que modelen tiempos de traslado y penalizaciones por transbordo, permitiendo analizar escenarios más realistas y evaluar algoritmos de búsqueda más avanzados. En este sentido, la práctica sienta una base sólida para el estudio de técnicas de búsqueda informada aplicadas a problemas del mundo real.