# Reporte: Proyecto Semana 3 - Modelando un Mapa de Tráfico con Grafos

## Estructuras de Datos Avanzadas

## 1. Introducción

Este proyecto implementa un **mapa de tráfico urbano** utilizando **teoría de grafos**. El objetivo es modelar intersecciones (vértices) y calles (aristas) de una ciudad pequeña, comparando dos enfoques:

1. **Grafo No Dirigido**: Calles bidireccionales (se puede ir y volver)
2. **Grafo Dirigido**: Calles con dirección específica (un solo sentido)

### Tecnologías Utilizadas

- **C#**: Implementación de la estructura de datos Graph
- **Python**: Análisis y visualización de estadísticas
- **Listas de Adyacencia**: Representación eficiente para grafos dispersos

## 2. Diseño del Mapa

### 2.1 Vértices (Intersecciones)

| ID | Descripción |
|---|---|
| **A** | Centro Comercial |
| **B** | Zona Norte |
| **C** | Zona Sur |
| **D** | Este Industrial |
| **E** | Oeste Residencial |
| **F** | Zona Industrial |
| **G** | Hospital |
| **H** | Estadio |

### 2.2 Aristas (Calles)

**Calles Bidireccionales (↔)**

- A ↔ B: 2.0 km
- A ↔ C: 3.0 km
- B ↔ D: 1.0 km

- C ↔ E: 4.0 km
- D ↔ F: 5.0 km
- E ↔ F: 2.0 km
- G ↔ H: 6.0 km

**Calles Unidireccionales (→)**

- A → G: 1.0 km (al hospital)
- B → H: 3.0 km (al estadio)
- C → D: 2.0 km
- F → E: 4.0 km
- H → A: 5.0 km (vuelta al centro)

### 2.3 Diagrama Simplificado

```
    B(Norte) ↔ A(Centro) ↔ C(Sur)
    ↓ 1km         ↓ 1km         ↓ 2km
    D(Este)       G(Hospital)   E(Oeste)
    ↓ 5km                       ↑ 4km
    F(Industrial) ←←←←←←←←←←← (F→E: 4km)

    H(Estadio) ↔ G(Hospital) (6km)
    ↑
    └─── A(Centro) (H→A: 5km)
```

---

## 3. Implementación en C#

### 3.1 Características de la Clase Graph

- **Genérica**: Funciona con cualquier tipo de dato (string, int, etc.)
- **Flexible**: Soporta grafos dirigidos y no dirigidos
- **Eficiente**: Usa listas de adyacencia (O(n+m) espacio)
- **Robusta**: Manejo de errores y validaciones

### 3.2 Métodos Principales

```
public class Graph<T> where T : IComparable<T>
{
    // Operaciones básicas
    void AddVertex(T vertex)
    void AddEdge(T from, T to, double weight, bool isDirected)

    // Consultas
    bool HasEdge(T from, T to)
    int GetOutDegree(T vertex)
    int GetInDegree(T vertex)
    IEnumerable<(T neighbor, double weight)> GetNeighbors(T vertex)
```

```
    // Exportación
    void ExportToFile(string filename, bool includeWeights, bool
deduplicateUndirected)
    void PrintGraph()
}
```

### 3.3 Resultados de Ejecución

```
🌍 === Generando Mapa de Tráfico === 🌍

🛣️ Agregando calles bidireccionales...
☑️ Archivo 'edges_undirected.txt' exportado exitosamente.

🚦 Creando mapa completo con calles direccionales...
☑️ Archivo 'edges_directed.txt' exportado exitosamente.

🧪 === Pruebas de Funcionalidad ===
Grado de A (no dirigido): 2 (esperado: 2) ✓
¿Existe A↔B no dirigido? True (esperado: True) ✓
¿Existe B↔A no dirigido? True (esperado: True) ✓

Grado salida A (dirigido): 3
Grado entrada A (dirigido): 2
¿Existe A→G dirigido? True (esperado: True) ✓
¿Existe G→A dirigido? False (esperado: False) ✓
```

---

## 4. Análisis en Python

### 4.1 Estadísticas del Grafo No Dirigido

```
📊 Estadísticas generales:
   • Vértices: 8
   • Aristas: 14 (7 bidireccionales × 2)
   • Densidad: 0.250
   • Tipo: Disperso

🏆 Vértice más conectado: A (grado total: 4)
📏 Distancia total de calles: 34.0 km
```

### 4.2 Estadísticas del Grafo Dirigido

```
📊 Estadísticas generales:
   • Vértices: 8
   • Aristas: 19
```

```
    • Densidad: 0.339
    • Tipo: Disperso

🏆 Vértice más conectado: A (out: 3, in: 2)
📏 Distancia total de calles: 53.0 km
```

### 4.3 Detalles por Vértice (Grafo Dirigido)

| Vértice | Out-Degree | In-Degree | Vecinos |
|---|---|---|---|
| A | 3 | 2 | B(2.0km), C(3.0km), G(1.0km) |
| B | 3 | 2 | A(2.0km), D(1.0km), H(3.0km) |
| C | 3 | 1 | A(3.0km), D(2.0km), E(4.0km) |
| D | 2 | 2 | B(1.0km), F(5.0km) |
| E | 2 | 3 | C(4.0km), F(2.0km) |
| F | 3 | 2 | D(5.0km), E(4.0km), E(2.0km) |
| G | 1 | 1 | H(6.0km) |
| H | 2 | 2 | A(5.0km), G(6.0km) |

---

## 5. Comparación: Matriz vs Lista de Adyacencia

### 5.1 Análisis de Complejidad

| Operación | Matriz | Lista |
|---|---|---|
| **Memoria (sparse)** | $O(n^2)$ = 64 espacios | O(n+m) = 27 conexiones |
| **¿Existe arista (u,v)?** | O(1) | O(grado(u)) |
| **Agregar arista** | O(1) | O(1) |
| **Eliminar arista** | O(1) | O(grado(u)) |
| **Recorrer vecinos de u** | O(n) | O(grado(u)) |

### 5.2 Ahorro de Memoria

Para nuestro grafo con **8 vértices** y **19 aristas**:

- **Matriz**: $8^2$ = 64 espacios
- **Lista**: 19 conexiones
- **Ahorro**: ~70.3% de memoria

### 5.3 Justificación de la Elección

☑ **Elegimos Listas de Adyacencia porque:**

1. **Grafo Disperso**: Solo 19 aristas de 64 posibles (29.7% de densidad)
2. **Memoria Eficiente**: Ahorra ~70% de espacio
3. **Recorridos Frecuentes**: En algoritmos de búsqueda (BFS/DFS), necesitamos iterar vecinos
4. **Escalabilidad**: Si agregamos más vértices, la matriz crece cuadráticamente

❌ **La Matriz sería mejor si:**

- El grafo fuera denso (>50% de aristas)
- Necesitáramos consultas frecuentes de "¿existe arista (u,v)?"
- El número de vértices fuera muy pequeño

---

# 6. Análisis de Resultados

## 6.1 Observaciones Clave

1. **Vértice Central (A)**:
    - Mayor conectividad (grado total: 5)
    - Punto estratégico del mapa
    - Conexión directa al hospital (emergencias)
2. **Calles Bidireccionales**:
    - 7 de 12 aristas (58.3%)
    - Facilitan el tráfico fluido
    - Reducen congestión
3. **Calles Unidireccionales**:
    - 5 aristas específicas (41.7%)
    - Optimizan flujo de tráfico
    - Ejemplo: H→A (regreso al centro desde el estadio)

## 6.2 Densidad del Grafo

- **Densidad**: 0.339 (33.9%)
- **Clasificación**: Grafo **disperso**
- **Implicación**: Listas de adyacencia son óptimas

## 6.3 Distancia Total

- **No Dirigido**: 34.0 km (solo calles bidireccionales)
- **Dirigido**: 53.0 km (todas las conexiones)
- **Diferencia**: 19.0 km de calles unidireccionales

---

# 7. Reflexión Técnica

## 7.1 ¿Qué perderíamos si cambiamos de lista a matriz?

**Desventajas de usar Matriz:**

- **Memoria**: Desperdiciaríamos 70% del espacio (45 de 64 celdas vacías)
- **Escalabilidad**: Si agregamos 2 vértices más, la matriz crece de 64 a 100 (56% más)
- **Recorridos**: Tendríamos que revisar todas las 64 celdas para encontrar vecinos

**Ventajas (mínimas en este caso):**

- Consultas O(1) para "¿existe arista?" (pero no es nuestra operación principal)

### 7.2 ¿Cómo afecta la dirección o el peso a las decisiones de diseño?

**Dirección:**

- **No Dirigido**: Simplifica la lógica (simetría automática)
- **Dirigido**: Permite modelar calles de un solo sentido (más realista)
- **Mixto**: Nuestro enfoque combina ambos para máxima flexibilidad

**Peso:**

- Representa **distancia en km** (podría ser tiempo, costo, etc.)
- Permite algoritmos de camino más corto (Dijkstra, A*)
- Facilita análisis de optimización de rutas

### 7.3 ¿Qué cambios harías para soportar grafos no ponderados?

**Modificaciones mínimas:**

```
// Opción 1: Peso por defecto = 1.0
public void AddEdge(T from, T to, bool isDirected = true)
{
    AddEdge(from, to, 1.0, isDirected);
}

// Opción 2: Usar bool en lugar de double
private Dictionary<T, List<T>> adjacencyList; // Sin peso
```

**Ventajas:**

- Menos memoria (no almacenar pesos)
- Código más simple
- Suficiente para problemas de conectividad

---

## 8. Extensiones Futuras

### 8.1 Algoritmos a Implementar (Próximas Semanas)

1. **BFS (Breadth-First Search)**
    - Encontrar camino más corto (en número de aristas)
    - Detectar componentes conexas

2. **DFS (Depth-First Search)**
    - Detectar ciclos
    - Ordenamiento topológico
3. **Dijkstra**
    - Camino más corto ponderado
    - Navegación GPS
4. **A***
    - Búsqueda heurística
    - Optimización de rutas

### 8.2 Mejoras Propuestas

- ☐ Visualización gráfica del mapa (NetworkX en Python)
- ☐ Interfaz web interactiva
- ☐ Simulación de tráfico en tiempo real
- ☐ Integración con APIs de mapas reales

---

## 9. Conclusiones

### 9.1 Logros del Proyecto

☑ **Conceptuales:**

- Comprensión profunda de grafos dirigidos vs no dirigidos
- Dominio de listas de adyacencia
- Análisis de trade-offs (memoria vs tiempo)

☑ **Prácticos:**

- Implementación robusta en C# con genéricos
- Análisis estadístico en Python
- Integración entre lenguajes (C# → archivos → Python)

☑ **Profesionales:**

- Código limpio y documentado
- Manejo de errores
- Pruebas de funcionalidad

### 9.2 Lecciones Aprendidas

1. **La representación importa**: Listas de adyacencia ahorran 70% de memoria en grafos dispersos
2. **La flexibilidad es clave**: Soportar dirigidos y no dirigidos en la misma clase
3. **La integración multiplica**: C# para estructuras + Python para análisis = solución completa

### 9.3 Aplicaciones Reales

Este proyecto sienta las bases para:

- **Sistemas de navegación** (Google Maps, Waze)
- **Redes sociales** (conexiones entre usuarios)
- **Redes de computadoras** (routing de paquetes)
- **Logística** (optimización de rutas de entrega)

---

## 10. Archivos Entregables

### Estructura del Proyecto

```
Semana3_Grafos/
├── Program.cs                  # Implementación C# del grafo
├── Semana3_Grafos.csproj      # Archivo de proyecto
├── analyze_graph.py           # Análisis en Python
├── edges_undirected.txt       # Datos del grafo no dirigido
├── edges_directed.txt         # Datos del grafo dirigido
├── Reporte_Semana3.md         # Este reporte
└── README.md                  # Guía de uso
```

### Cómo Ejecutar

```
# 1. Generar archivos de datos con C#
cd Semana3_Grafos
dotnet run

# 2. Analizar con Python
python analyze_graph.py
```

---

**Fecha de Completación**: Diciembre 2025
**Curso**: Estructuras de Datos Avanzadas
**Semana**: 3 - Teoría de Grafos
**Estudiante**: [Tu Nombre]
**Profesor**: [Nombre del Profesor]

---

## Referencias

1. Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
3. Documentación oficial de C#: https://docs.microsoft.com/dotnet/csharp/
4. Documentación de Python: https://docs.python.org/3/