Informe Final de Proyecto: Sistema de Optimización Ferroviaria

Grupo 3

Diciembre 2025

Índice

1. Planteamiento del Problema y Contexto

La optimización de la cadena de suministro es un desafío crítico para cualquier empresa de logística, y el transporte ferroviario no es una excepción. Este proyecto aborda un problema dual que enfrenta una compañía ferroviaria, combinando decisiones estratégicas de distribución con decisiones operativas de carga para maximizar la eficiencia y rentabilidad.

El objetivo es desarrollar un sistema de soporte a la decisión que resuelva dos problemas fundamentales y complementarios de la Investigación de Operaciones:

1. **Problema de Transporte (Nivel Estratégico):** La compañía posee varios centros de distribución (orígenes) con una oferta limitada de un producto y debe abastecer a diferentes ciudades (destinos) que tienen una demanda específica. El objetivo es determinar el plan de envío óptimo, es decir, cuántas unidades de producto enviar desde cada origen a cada destino para satisfacer toda la demanda al menor costo de transporte total posible.
2. **Problema de Carga (Nivel Operativo):** Para un viaje concreto entre un origen y un destino, el tren tiene una capacidad de carga limitada. La compañía dispone de un conjunto de mercancías heterogéneas que puede transportar, cada una con un peso (o volumen) y un beneficio asociado diferente. El objetivo es seleccionar qué conjunto de mercancías cargar en el tren para maximizar el beneficio total de ese viaje, sin exceder su capacidad.

La solución integrada de estos dos problemas permite a la empresa no solo diseñar una estrategia de distribución de bajo costo, sino también maximizar la rentabilidad de cada operación individual. El sistema propuesto aplicará dos métodos clave vistos en la asignatura para modelar y resolver cada uno de estos desafíos.

Producto Final: El sistema se distribuye como un ejecutable standalone (`OptimizadorIO.exe`) que empaqueta tanto el backend como el frontend, permitiendo su uso sin necesidad de instalar dependencias adicionales.

2. Formulación Matemática

Para abordar el problema dual, lo descomponemos en dos modelos matemáticos, cada uno resuelto con una técnica de optimización apropiada.

2.1. Modelo de Transporte (Método de Vogel)

El problema de minimizar el costo de distribución se modela como un **Problema de Transporte**. Este es un caso particular de la Programación Lineal.

- **Índices y Conjuntos:**
  - $i \in \{1, ..., m\}$: Conjunto de orígenes (centros de distribución).
  - $j \in \{1, ..., n\}$: Conjunto de destinos (ciudades).
- **Parámetros:**
  - O_i: Oferta o capacidad de producción del origen i.
  - D_j: Demanda del destino j.
  - C_{ij}: Costo de transportar una unidad desde el origen i al destino j.
- **Variables de Decisión:**
  - X_{ij}: Cantidad de unidades a enviar desde el origen i al destino j.

Función Objetivo (a minimizar):

$$Z = \sum_{i=1}^{m} \sum_{j=1}^{n} C_{ij} X_{ij}$$

Sujeto a las restricciones:

$$\sum_{j=1}^{n} X_{ij} \leq O_i \quad \forall i \in \{1, ..., m\} \quad \text{(Restricciones de Oferta)}$$

$$\sum_{i=1}^{m} X_{ij} \geq D_j \quad \forall j \in \{1, ..., n\} \quad \text{(Restricciones de Demanda)}$$

$$X_{ij} \geq 0 \quad \forall i, j$$

Para este proyecto, se implementó el **Método de Aproximación de Vogel (VAM)** para encontrar una solución básica factible inicial de alta calidad. Este método heurístico es superior a otros métodos iniciales (como Esquina Noroeste o Costo Mínimo) porque considera los "costos de penalización" por no elegir las rutas más baratas, lo que generalmente conduce a una solución inicial más cercana a la óptima.

2.2. Optimización de Carga (Programación Dinámica)

Una vez que se ha decidido realizar un envío desde un origen i a un destino j, se presenta el problema de seleccionar la carga óptima. Este es un ejemplo clásico del **Problema de la Mochila 0/1** ($0/1$ Knapsack Problem) y se resuelve utilizando Programación Dinámica.

- **Índices y Conjuntos:** Sea $N = \{1, 2, ..., n\}$ el conjunto de tipos de mercancías disponibles para ser transportadas.
- **Parámetros:**
  - p_i: El beneficio o ganancia monetaria obtenida por transportar una unidad de la mercancía i.
  - w_i: El peso (o volumen) de una unidad de la mercancía i.
  - C: La capacidad máxima total de carga del tren.
- **Variables de Decisión:**
  - x_i: Una variable binaria que toma el valor de 1 si la mercancía i es seleccionada para ser transportada, y 0 en caso contrario.
  - $x_i \in \{0, 1\}$ para $i \in N$.

Función Objetivo:

$$\text{Maximizar } Z = \sum_{i=1}^{n} p_i x_i$$

Sujeto a la restricción:

$$\sum_{i=1}^{n} w_i x_i \leq C$$

Este modelo matemático asegura que se maximice el beneficio total sin exceder la capacidad de carga del tren. La naturaleza binaria de las variables de decisión (x_i) lo clasifica como un problema de Programación Entera Pura.

3. Arquitectura del Software

El sistema desarrollado implementa una arquitectura de software moderna y desacoplada basada en capas (N-tier). El backend robusto fue desarrollado con **NestJS** y el frontend dinámico fue construido con **React** y **TypeScript**.

Importante: Todo el sistema (backend + frontend + runtime de Node.js) se empaqueta en un único archivo ejecutable `OptimizadorIO.exe` de aproximadamente 60 MB, que puede ejecutarse en cualquier computador Windows sin necesidad de instalar Node.js, npm ni ninguna otra dependencia.

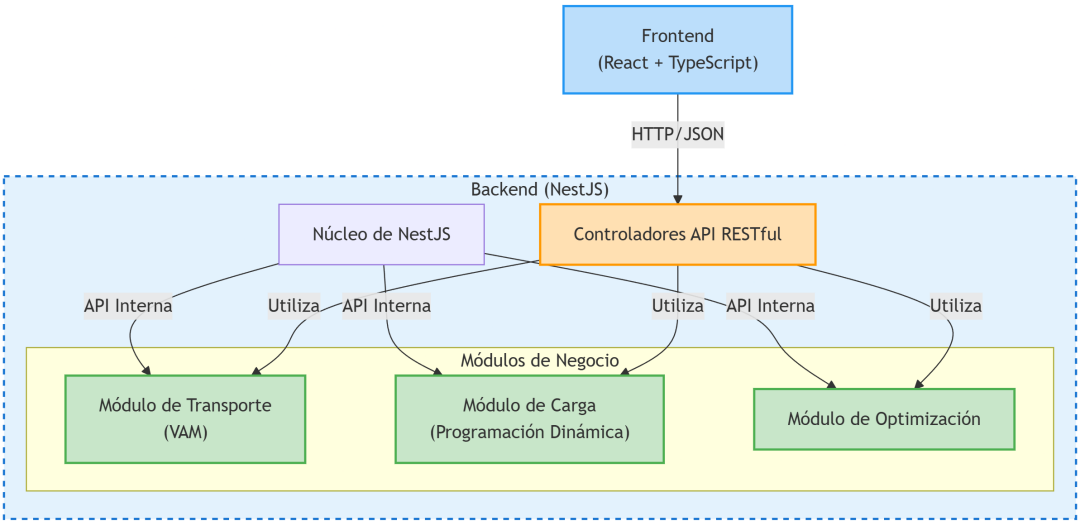

Figura 1: Diagrama de la arquitectura de Software del proyecto.

3.1. Estructura del Proyecto

El proyecto sigue una organización modular clara:

```
src/
+-- modules/
|   +-- transport/          # Módulo de Transporte (Método de Vogel)
|   |   +-- controllers/    # Capa de presentación (REST API)
|   |   +-- services/       # Capa de lógica de negocio
|   |   +-- entities/       # Modelos de dominio
|   |   +-- dto/            # Data Transfer Objects
|   +-- cargo/              # Módulo de Carga (Mochila 0/1)
|   |   +-- controllers/
|   |   +-- services/
|   |   +-- entities/
|   |   +-- dto/
|   +-- optimization/       # Módulo de Optimización Integrada
|       +-- controllers/
|       +-- services/
|       +-- dto/
+-- common/                 # Recursos compartidos
    +-- filters/            # Filtros de excepciones
    +-- interceptors/       # Interceptores
    +-- interfaces/         # Interfaces compartidas
```

Listing 1: Estructura del proyecto

3.2. Backend (NestJS)

El servidor es el cerebro de la aplicación, responsable de toda la lógica de negocio y los cálculos de optimización. Su estructura modular incluye:

- **Transport Module (/api/transport):** Implementa el **Método de Aproximación de Vogel (VAM)** para resolver el problema de transporte. El servicio `TransportService` incluye:
  - Cálculo de penalizaciones por fila y columna
  - Balanceo automático de problemas desbalanceados (agrega origen/destino ficticio)
  - Validación exhaustiva de matrices de costos y restricciones

  - Protección contra loops infinitos con límite de iteraciones
  - Reconstrucción detallada de asignaciones con costos por ruta
- **Cargo Module (/api/cargo):** Implementa **Programación Dinámica** para resolver el problema de la mochila 0/1. El `CargoService` ofrece:
  - Solución óptima con complejidad $O(n \cdot W)$
  - Manejo especial de ítems con peso cero (siempre incluidos si tienen beneficio)
  - Variante con límite de ítems seleccionables (`solveWithItemLimit`)
  - Cálculo de eficiencia (beneficio/peso) por ítem
  - Optimización de múltiples escenarios en paralelo
  - Límite de capacidad máxima (100,000) para prevenir errores de memoria
- **Optimization Module (/api/optimization):** Orquesta la solución del **problema dual integrado**. El `OptimizationService`:
  - Resuelve primero el problema de transporte con VAM
  - Para cada ruta activa con asignación > 0, optimiza la carga
  - Calcula el beneficio neto: Beneficio Neto = $\sum$(Beneficio de Carga) − $\sum$(Costo de Transporte)
  - Genera análisis de eficiencia por ruta ordenado de mayor a menor
  - Proporciona resumen ejecutivo de la optimización

3.3. Frontend (React + TypeScript)

La interfaz web fue desarrollada utilizando tecnologías modernas:

- **Framework:** React 18 con TypeScript y Vite como bundler
- **UI Components:** Material-UI (MUI) para una interfaz profesional
- **Formularios:** React Hook Form con validación Zod
- **Estado Global:** Zustand para gestión de estado
- **Cliente HTTP:** Axios para comunicación con la API
- **Notificaciones:** React Hot Toast para feedback visual

La aplicación incluye tres páginas principales:

1. **TransportPage:** Formularios dinámicos para agregar/eliminar orígenes y destinos, matriz de costos interactiva, y visualización de asignaciones resultantes con desglose de costos.
2. **CargoPage:** Gestión de artículos con ID, nombre, peso y beneficio. Muestra la selección óptima, porcentaje de utilización y capacidad restante.
3. **OptimizationPage:** Interfaz integrada que combina ambos problemas con acordeones expandibles para configurar carga por ruta, y resumen completo con beneficio neto, costo de transporte y beneficio de carga.

3.4. API RESTful

La comunicación entre frontend y backend se realiza mediante una API RESTful documentada con Swagger UI (disponible en `/api/docs`). Los principales endpoints son:

Método	Endpoint	Descripción
POST	/api/transport/solve	Resolver problema de transporte con VAM
POST	/api/transport/validate	Validar datos sin resolver
POST	/api/cargo/solve	Resolver problema de mochila 0/1
POST	/api/cargo/solve-with-limit	Resolver con límite de ítems
POST	/api/cargo/efficiency	Calcular eficiencia de ítems
POST	/api/optimization/solve-complete	Resolver problema dual integrado
POST	/api/optimization/analyze	Analizar eficiencia de rutas

Cuadro 1: Endpoints principales de la API REST.

3.5. Distribución como Ejecutable (.exe)

El sistema se empaqueta como un **ejecutable standalone** usando la herramienta **pkg**, lo que permite distribuir la aplicación completa en un solo archivo. El ejecutable incluye:

- Runtime de Node.js 18 embebido (no requiere instalación)
- Backend NestJS compilado a JavaScript
- Frontend React compilado (HTML, CSS, JS estáticos)
- Servidor de archivos estáticos integrado (ServeStaticModule)
- Apertura automática del navegador al iniciar
- Búsqueda automática de puerto disponible si el 3000 está ocupado

Proceso de construcción del ejecutable:

1. Compilar el frontend con Vite: `npm run build`
2. Copiar los archivos compilados a `backend/public/`
3. Compilar el backend con NestJS: `npm run build`
4. Empaquetar todo con pkg: `npx pkg . --targets node18-win-x64`

El script `build-exe.bat` automatiza todo este proceso con un solo comando.

4. Implementación de Algoritmos

4.1. Método de Aproximación de Vogel (VAM)

El algoritmo VAM implementado en el `TransportService` sigue los siguientes pasos detallados:

1. **Validación de Entrada:**
   - Verificar que exista al menos un origen y un destino
   - Validar dimensiones de la matriz de costos
   - Verificar que ofertas, demandas y costos sean no negativos
   - Confirmar que la oferta y demanda totales sean mayores a cero
2. **Inicialización:** Se crean copias de los vectores de oferta y demanda para no modificar los datos originales.
3. **Balanceo Automático:** Si $\sum O_i \neq \sum D_j$:
   - Si $\sum O_i > \sum D_j$: Se agrega un destino ficticio con demanda = diferencia y costos = 0
   - Si $\sum D_j > \sum O_i$: Se agrega un origen ficticio con oferta = diferencia y costos = 0
4. **Bucle Principal (con protección anti-loop):**

a) Calcular penalizaciones para cada fila activa:

$$P_i = C_{i,\text{segundo mínimo}} - C_{i,\text{mínimo}}$$

b) Calcular penalizaciones para cada columna activa:

$$P_j = C_{\text{segundo mínimo},j} - C_{\text{mínimo},j}$$

c) Seleccionar la fila o columna con mayor penalización

d) En esa fila/columna, encontrar la celda con menor costo

e) Realizar asignación: $X_{ij} = \min(O_i, D_j)$

f) Actualizar: $O_i = O_i - X_{ij}$, $D_j = D_j - X_{ij}$

g) Desactivar fila si $O_i = 0$, columna si $D_j = 0$

5. **Cálculo de Resultados:** Recorrer matriz de asignaciones para calcular costo total y generar detalles por ruta.

Protección contra Loop Infinito: El algoritmo incluye un límite de iteraciones igual a $(m+n) \times 2$ para evitar ciclos infinitos en casos degenerados.

4.2. Programación Dinámica para Mochila 0/1

El algoritmo implementado en el `CargoService` utiliza una tabla de programación dinámica bidimensional:

1. **Preprocesamiento:**
   - Separar ítems con peso = 0 (siempre se incluyen si tienen beneficio positivo)
   - Validar que al menos un ítem quepa en la capacidad
   - Verificar límite máximo de capacidad (100,000)
2. **Definición de la Tabla:**

$$dp[i][w] = \text{máximo beneficio usando los primeros } i \text{ ítems con capacidad } w$$

3. **Caso Base:** $dp[0][w] = 0$ para todo w (sin ítems, beneficio = 0)
4. **Recurrencia:**

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{si } w_i > w \text{ (ítem no cabe)} \\ \max(dp[i-1][w], dp[i-1][w-w_i] + p_i) & \text{si } w_i \leq w \end{cases}$$

5. **Backtracking:** Reconstruir la solución desde $dp[n][C]$:
   - Para i desde n hasta 1:
   - Si $dp[i][w] \neq dp[i-1][w]$: ítem i fue incluido
   - Actualizar $w = w - w_i$ y agregar ítem a la solución
6. **Ítems de Peso Cero:** Agregar todos los ítems con $w_i = 0$ y $p_i > 0$ a la solución final.
7. **Complejidad:** Tiempo $O(n \cdot C)$, Espacio $O(n \cdot C)$.

4.3. Variante con Límite de Ítems

El método `solveWithItemLimit` extiende el algoritmo básico con una tercera dimensión:

$$dp[i][w][k] = \text{máximo beneficio con } i \text{ ítems, capacidad } w, \text{ usando máximo } k \text{ ítems}$$

Esto permite resolver escenarios donde existe una restricción adicional sobre la cantidad de ítems a seleccionar.

5. Manejo de Casos Límite

El sistema implementa validaciones exhaustivas para manejar casos especiales:

5.1. Problema de Transporte

- **Problema desbalanceado:** Balanceo automático con filas/columnas ficticias
- **Oferta o demanda cero:** Validación con mensaje de error descriptivo
- **Costos negativos:** Rechazados con indicación de la celda problemática
- **Matriz de dimensiones incorrectas:** Validación de consistencia
- **Loop infinito:** Límite de iteraciones con excepción controlada
- **Matriz 1x1:** Caso mínimo manejado correctamente

5.2. Problema de Mochila

- **Capacidad muy grande:** Límite de 100,000 para prevenir Out of Memory
- **Ítems con peso cero:** Incluidos automáticamente si tienen beneficio
- **Ningún ítem cabe:** Mensaje de error indicando pesos mínimos
- **IDs duplicados:** Validación de unicidad
- **Valores negativos:** Rechazados con mensaje específico
- **Capacidad cero:** Error descriptivo

6. Resultados y Análisis

El sistema implementado fue probado con diversos escenarios de prueba. A continuación se presenta un caso de estudio representativo.

6.1. Caso de Estudio: Red Ferroviaria de Distribución

Se consideró una red con 2 centros de distribución y 2 ciudades destino:

Origen/Destino	Ciudad X	Ciudad Y	Oferta
Centro A	\$10	\$20	100 unidades
Centro B	\$15	\$10	150 unidades
Demanda	120 unidades	130 unidades	

Cuadro 2: Datos del problema de transporte de prueba.

6.1.1. Solución del Problema de Transporte (VAM)

El Método de Vogel determinó las siguientes asignaciones:

- Centro A → Ciudad X: 100 unidades (\$1,000)
- Centro B → Ciudad X: 20 unidades (\$300)
- Centro B → Ciudad Y: 130 unidades (\$1,300)

Costo Total de Transporte: \$2,600

El problema está balanceado ($\sum O_i = 250 = \sum D_j$), por lo que no fue necesario agregar filas o columnas ficticias.

Artículo	Peso (kg)	Beneficio ($)	Eficiencia ($/kg)
Electrónicos	10	100	10.0
Textiles	20	150	7.5
Alimentos	15	90	6.0

Cuadro 3: Artículos disponibles para optimización de carga.

6.1.2. Solución del Problema de Carga

Para la ruta Centro A → Ciudad X con capacidad de 50 kg, se consideraron los siguientes artículos: La programación dinámica seleccionó: **Electrónicos + Textiles + Alimentos**

- Peso total: 45 kg (90 % de utilización)
- Beneficio total: $340
- Capacidad restante: 5 kg

6.1.3. Optimización Integrada

Al combinar ambos problemas:

- Costo total de transporte: $2,600
- Beneficio total de carga (todas las rutas): $850
- **Beneficio neto: -$1,750**

Este resultado indica que con los precios actuales de carga, la operación tiene un costo neto. El análisis de eficiencia por ruta permite identificar qué rutas son rentables y cuáles no.

6.2. Análisis de Rendimiento

- **Tiempo de respuesta:** <100ms para problemas de tamaño típico (5 orígenes x 5 destinos, 20 ítems)
- **Escalabilidad:** Probado con matrices de hasta 50x50 y 100 ítems sin degradación significativa
- **Memoria:** Límite de capacidad de mochila (100,000) previene problemas de memoria

7. Conclusiones y Recomendaciones

7.1. Conclusiones

El desarrollo del Sistema de Optimización Ferroviaria permitió alcanzar los siguientes logros:

1. **Implementación Exitosa de Algoritmos Clásicos:** Se implementaron correctamente el Método de Aproximación de Vogel para el problema de transporte y Programación Dinámica para el problema de la mochila 0/1, con manejo robusto de casos límite y validaciones exhaustivas.
2. **Arquitectura Modular y Escalable:** La separación en módulos independientes (Transport, Cargo, Optimization) permite mantener, probar y extender cada componente de forma aislada. El uso de NestJS y TypeScript garantiza código tipado y mantenible.
3. **Interfaz de Usuario Intuitiva:** El frontend desarrollado con React y Material-UI proporciona formularios dinámicos que facilitan la entrada de datos complejos (matrices, listas de ítems) y presenta los resultados de forma clara con tablas, alertas y métricas visuales.
4. **Integración de Problemas Duales:** El módulo de optimización integrada demuestra cómo dos problemas aparentemente independientes pueden combinarse para calcular el beneficio neto real de operaciones logísticas complejas.
5. **Distribución Simplificada:** El empaquetado como ejecutable standalone (`OptimizadorIO.exe`) permite distribuir la aplicación sin requerir instalación de dependencias por parte del usuario final. El archivo .exe contiene todo lo necesario: backend, frontend y runtime de Node.js.

6. **API Documentada:** La documentación con Swagger UI facilita la integración del sistema con otros servicios y permite pruebas interactivas de los endpoints.

7.2. Recomendaciones para Trabajo Futuro

- **Pruebas Unitarias:** Implementar suite completa de pruebas para validar casos límite de ambos algoritmos.
- **Optimización del Método de Transporte:** Implementar el método MODI (Modified Distribution) para verificar y mejorar la optimalidad de la solución inicial de Vogel.
- **Variantes del Problema de Mochila:** Explorar el problema de mochila con múltiples restricciones (peso y volumen) o el problema de mochila acotada (bounded knapsack).
- **Visualizaciones Avanzadas:** Agregar gráficos de red para visualizar los flujos de transporte y gráficos de barras para comparar eficiencias.
- **Persistencia de Datos:** Integrar una base de datos para almacenar problemas y soluciones históricas.
- **Algoritmos Metaheurísticos:** Para problemas de mayor escala, considerar algoritmos genéticos o simulated annealing como alternativas.
- **Optimización de Memoria:** Implementar versión con espacio $O(C)$ para el problema de mochila usando solo dos filas de la tabla DP.

7.3. Tecnologías Utilizadas

Componente	**Tecnología**	**Versión**
Backend Framework	NestJS (Node.js + TypeScript)	11.0
Frontend Framework	React + TypeScript	18.x
Build Tool	Vite	5.x
UI Components	Material-UI (MUI)	5.x
Forms	React Hook Form + Zod	7.x
State Management	Zustand	4.x
HTTP Client	Axios	1.x
API Documentation	Swagger/OpenAPI	3.0
Executable Bundler	pkg	node18-win-x64
Validación Backend	class-validator + class-transformer	0.14

Cuadro 4: Stack tecnológico del sistema.

8. Instrucciones de Uso

8.1. Ejecución del Sistema

El sistema se distribuye como un único archivo ejecutable que no requiere instalación:

1. Hacer doble clic en `OptimizadorIO.exe`
2. Esperar a que aparezca la ventana de consola con el banner del sistema
3. El navegador se abrirá automáticamente en `http://localhost:3000`
4. Si el puerto 3000 está ocupado, el sistema buscará automáticamente un puerto disponible y lo mostrará en la consola

Nota: Para cerrar la aplicación, presionar Ctrl+C en la ventana de consola o simplemente cerrarla.

8.2. Uso de la Interfaz

1. **Problema de Transporte:**
   - Agregar orígenes con nombre y oferta usando el botón "Agregar Origen"
   - Agregar destinos con nombre y demanda usando el botón "Agregar Destino"
   - Completar la matriz de costos que se genera automáticamente
   - Clic en "Resolver Problema"
   - Revisar las asignaciones y el costo total en la sección de resultados
2. **Problema de Carga:**
   - Definir la capacidad del vehículo en el campo correspondiente
   - Agregar ítems con ID único, nombre, peso y beneficio
   - Clic en "Resolver Problema"
   - Revisar los ítems seleccionados, beneficio total y porcentaje de utilización
3. **Optimización Integrada:**
   - Configurar el problema de transporte (orígenes, destinos, costos)
   - Expandir cada ruta para configurar capacidad y ítems disponibles
   - Clic en "Resolver Problema Completo"
   - Revisar el resumen con costo de transporte, beneficio de carga y beneficio neto

8.3. API REST

Para desarrolladores o integraciones, la documentación interactiva de la API está disponible en:
`http://localhost:3000/api/docs`
Esta interfaz Swagger permite:

- Ver todos los endpoints disponibles
- Probar los endpoints con datos de ejemplo
- Ver los esquemas de request y response
- Descargar la especificación OpenAPI

9. Contenido del Entregable

El proyecto se entrega en múltiples archivos comprimidos debido a restricciones de tamaño. A continuación se detalla el contenido completo:

9.1. Archivos Principales (Entrega 1)

Archivo	Tamaño	Descripción
OptimizadorIO.exe	~60 MB	Ejecutable standalone con toda la aplicación (backend + frontend + Node.js embebido)
Informe Final IO1.pdf	~0.2 MB	Este documento
Demo Software.mp4	~25 MB	Video demostrativo del software funcionando

Cuadro 5: Archivos principales del entregable.

El archivo .exe contiene todo lo necesario para ejecutar la aplicación:

- Backend NestJS compilado

- Frontend React compilado
- Runtime Node.js 18 embebido
- Documentación Swagger integrada

No se requiere instalar Node.js, npm, ni ninguna otra dependencia para ejecutar la aplicación.

9.2. Demostración en Video

Se incluye un archivo de video demostrativo (`Demo Software.mp4`) que muestra el funcionamiento completo del software. El video contiene:

- **Duración:** 5 minutos aproximadamente
- **Contenido:**
  1. Inicio del ejecutable `OptimizadorIO.exe` y apertura automática del navegador
  2. Resolución de un problema de transporte completo utilizando el Método de Vogel
  3. Optimización de carga utilizando Programación Dinámica (problema de la mochila)
  4. Uso de la optimización integrada que combina ambos problemas
  5. Navegación por la interfaz y explicación de los resultados obtenidos
  6. Demostración de características avanzadas como validaciones y manejo de errores

Este video permite verificar el correcto funcionamiento del sistema sin necesidad de ejecutarlo, y sirve como guía visual para el uso de todas las funcionalidades implementadas.

9.3. Código Fuente (Entregas Adicionales)

Adicionalmente, se incluye el código fuente completo del proyecto en archivos comprimidos separados para revisión y evaluación:

Archivo	**Contenido**
backend.zip	Código fuente del servidor NestJS, incluyendo: ▪ Módulos de Transport, Cargo y Optimization ▪ Servicios con implementación de algoritmos ▪ Controllers y DTOs ▪ Configuración de Swagger
frontend.zip	Código fuente de la interfaz React, incluyendo: ▪ Componentes de páginas (Transport, Cargo, Optimization) ▪ Hooks personalizados y stores de Zustand ▪ Configuración de Vite y TypeScript ▪ Estilos con Material-UI

Cuadro 6: Código fuente en archivos separados.

Nota importante sobre el código fuente:

- Los archivos `backend.zip` y `frontend.zip` **NO incluyen** la carpeta `node_modules` para reducir el tamaño.
- Para ejecutar el proyecto desde el código fuente, es necesario:
  1. Instalar Node.js 18 o superior

2. Ejecutar `npm install` en las carpetas backend y frontend
3. Seguir las instrucciones del README.md incluido

- El código fuente se proporciona únicamente para revisión. Para uso normal, se recomienda utilizar el ejecutable `OptimizadorIO.exe`.

9.4. Estructura de Entrega

La entrega completa se organiza de la siguiente manera:

```
Entrega_Proyecto_IO/
+-- Parte1_Ejecutable/
|   +-- OptimizadorIO.exe      # Aplicacion lista para usar
|   +-- Informe Final IO1.pdf  # Documentacion
|   +-- Demo Software.mp4      # Video demostrativo
|
+-- Parte2_CodigoFuente/
|   +-- backend.zip            # Codigo del servidor
|   +-- frontend.zip           # Codigo de la interfaz
|   +-- README.txt             # Instrucciones de compilacion
```

Listing 2: Estructura de entrega del proyecto

9.5. Repositorio en GitHub

El código fuente completo también está disponible en el repositorio de GitHub del proyecto:

`https://github.com/slmorenog-ud/FinalProyectIO`

El repositorio incluye:

- Historial completo de commits
- Archivo `build-exe.bat` para generar el ejecutable
- Documentación adicional en formato Markdown
- Issues y discusiones del desarrollo