Proyecto Final de Diseño y Análisis de Algoritmos

Problema: "El Comerciante Holandés"

Claudia Hernández Pérez Joel Aparicio Tamayo

21 de diciembre de 2025

Resumen

Este documento presenta el análisis completo del problema *"El Comerciante Holandés"*. Se incluye la formalización matemática del problema, análisis de complejidad computacional, diseño de algoritmos por fuerza bruta y eficiente (con uso de metaheurística), y finalmente una comparación entre ambos enfoques.

1. Definición del problema

La prestigiosa Compañía Holandesa de las Indias Orientales, en su afán por dominar el comercio mundial, se enfrenta a un desafío monumental. Un capitán experimentado, al mando de una de sus valiosas flotas, debe emprender una expedición comercial que partirá de Ámsterdam y, tras recorrer los puertos más lucrativos del Viejo y Nuevo Continente, deberá regresar a su puerto de origen.

Los inversores de la Compañía han proporcionado un capital inicial considerable y han establecido un plazo máximo para la duración de la expedición. El capitán tiene la libertad de elegir qué puertos visitar y en qué orden, con la única condición de no visitar el mismo puerto dos veces en el mismo viaje (por cuestiones de acuerdos comerciales y evitar saturación del mercado).

En cada puerto, el capitán encontrará una lista de mercancías disponibles, con sus respectivos precios de compra y venta (que pueden variar significativamente). El capitán puede vender las mercancías que lleva a bordo y comprar nuevas. Sin embargo, debe ser astuto:

- La capacidad de carga de su barco es limitada, por lo que no puede llevar más de lo que su bodega permite.
- No es necesario vender todas las mercancías al llegar a un puerto; el capitán puede decidir retener parte de su cargamento si cree que podrá venderlo a un precio más alto en un puerto posterior.
- Debe asegurarse de que, después de cada operación de compra, le quede suficiente dinero para cubrir los salarios de la tripulación, los impuestos portuarios y las posibles reparaciones del barco hasta el siguiente destino.
- El tiempo es oro; la duración total del viaje, incluyendo el tiempo de navegación entre puertos, no debe exceder el plazo fijado por los inversores.

El objetivo del capitán es claro: planificar la ruta y las transacciones en cada puerto de tal manera que, al regresar a Amsterdam, el capital final de la expedición sea el máximo posible, superando con creces la inversión inicial.

2. Formalización del Problema

Antes de abordar cualquier solución algorítmica, es fundamental establecer una representación matemática precisa del problema. Esta sección presenta nuestra formalización del Comerciante Holandés, definiendo las estructuras de datos de entrada, las restricciones que deben cumplirse y la función objetivo a optimizar. Este modelo elimina las ambigüedades de la descripción narrativa original.

2.1. Definiciones Matemáticas

Sea:

- $G = (V, E)$: grafo completo de n puertos, donde $V = \{0, 1, \ldots, n-1\}$ (0 representa Ámsterdam, punto de inicio y fin).
- d_{ij}: tiempo de navegación entre puertos i y j. Se asume:
  - $d_{ii} = 0, \quad \forall i$
  - $d_{ij} = d_{ji}, \quad \forall i, j$
  - $d_{ij} \leq d_{ik} + d_{kj}, \quad \forall i, k, j$

- T_{max}: tiempo máximo total permitido para la expedición.
- C_{max}: capacidad máxima de carga del barco.
- K_0: capital inicial proporcionado por los inversores.
- K_{min}: capital mínimo que debe mantenerse tras cada transacción (para gastos operativos).

En cada puerto $i \in V$:

- $M_i = \{(w_{ik}, p_{ik}^{compra}, p_{ik}^{venta})\}_{k=1}^{m_i}$: conjunto de m_i mercancías disponibles, donde:
  - $w_{ik} \in \mathbb{R}$: peso de la mercancía k en puerto i
  - $p_{ik}^{compra} \in \mathbb{R}$: precio de compra de la mercancía k en puerto i
  - $p_{ik}^{venta} \in \mathbb{R}$: precio de venta de la mercancía k en puerto i
- **Importante**: No hay garantía de que $p_{ik}^{venta} > p_{jk}^{compra}\ \forall i \neq j$. Pueden haber mercancías con pérdida potencial, lo que obliga al capitán a tomar decisiones estratégicas. Sin embargo sí se garantiza que $p_{ik}^{venta} \leq p_{ik}^{compra}\ \forall i$ (los del puerto tampoco son tontos, no van a pagar más de lo que ofertan).

2.2. Estado del Sistema

El estado al llegar a un puerto i se define como una tupla:

$$S_i = (K_i, L_i, R_i, t_i)$$

donde:

- $K_i \in \mathbb{R}$: capital disponible al llegar al puerto i.
- $L_i \subseteq \{m = (i', k, w, p^{compra}) : i', k \in \mathbb{N}; w, p^{compra} \in \mathbb{R}\}$: conjunto de mercancías a bordo, donde cada mercancía está identificada por:
  - i': puerto donde fue comprada
  - k: índice de la mercancía en ese puerto
  - w: peso de la mercancía
  - p^{compra}: precio pagado por la mercancía
- $R_i \subseteq V$: conjunto de puertos ya visitados (ruta parcial).
- $t_i \in \mathbb{R}^+$: tiempo acumulado hasta llegar al puerto i.

2.3. Operaciones

Al llegar a un puerto i, el capitán realiza secuencialmente:

1. **Venta parcial**: Para cada mercancía $g = (i_g, k_g, w_g, p_g^{compra}) \in L_i$, el capitán decide si venderla en el puerto actual i obteniendo p_{i,k_g}^{venta} (precio de venta de esa mercancía en puerto i). Sea $M_i' \subseteq L_i$ el subconjunto de mercancías vendidas:

$$K_i \leftarrow K_i + \sum_{g \in M_i'} p_{i,k_g}^{venta}$$

   Las mercancías vendidas se eliminan de la carga: $L_i \leftarrow L_i \setminus M_i'$.

2. **Compra selectiva**: Considerando las mercancías disponibles en M_i, el capitán selecciona un subconjunto $C_i \subseteq M_i$ para comprar, sujeto a:

   - **Restricción financiera**:

$$K_i - \sum_{(w, p^{compra}, p^{venta}) \in C_i} p^{compra} \geq K_{min}$$

   - **Restricción de capacidad**:

$$\sum_{g \in L_i} w_g + \sum_{(w, p^{compra}, p^{venta}) \in C_i} w \leq C_{max}$$

   Las mercancías compradas se añaden a bordo:

$$\forall (w, p^{compra}, p^{venta}) \in C_i : \quad L_i \leftarrow (i, k, w, p^{compra})$$

   Luego se actualiza el capital:

$$K_i \leftarrow K_i - \sum_{(w, p^{compra}, p^{venta}) \in C_i} p^{compra}$$

3. **Selección de próximo destino**: Sea $j \in V \setminus (R_i \cup \{i\})$ un candidato a próximo puerto, j es seleccionable si:

$$t_i + d_{ij} + d_{j0} \leq T_{max}$$

   Si ningún j cumple esta condición, el capitán debe regresar a Ámsterdam ($j = 0$). Luego de seleccionar el próximo destino, se actualiza el recorrido actual:

$$R_i \leftarrow R_i \cup \{i\}$$

2.4. Función Objetivo

$$\text{máx}\, K_f \quad ,$$

donde K_f es el capital tras vender cualquier carga restante en Ámsterdam (a precios de Ámsterdam).

sujeto a:

1. **Ruta simple**: Sea $\pi = (v_0, v_1, \ldots, v_m, v_{m+1})$ la secuencia de puertos visitados, donde $v_0 = v_{m+1} = 0$ (Ámsterdam). Entonces:

   $$\forall i \in \{1, \ldots, m\} : v_i \in V \setminus \{0\} \quad \wedge \quad \forall i \neq j \in \{1, \ldots, m\} : v_i \neq v_j$$

   Es decir, ningún puerto no-inicial se repite en la ruta.

2. **Tiempo límite**: $t_f \leq T_{max}$, siendo t_f el tiempo de llegada a Ámsterdam al finalizar el viaje.

3. **Capital suficiente para operatividad**: $\forall i$, $K_i \geq K_{min}$.

4. **Capacidad**: $\forall i$, $\sum_{g \in L_i} w_g \leq C_{max}$.

3. Análisis de Complejidad Computacional

El problema es presentado originalmente como un problema de optimización, cuyo objetivo es hallar una ruta y una estrategia de transacciones que maximicen el capital final K_f. Para demostrar su clasificación en la jerarquía de complejidad, se define su versión de decisión asociada:

> *Dada la instancia del problema descrita en la Sección 2 y un umbral de capital $K_{objetivo} \in \mathbb{R}$, ¿existe una secuencia de puertos y transacciones tales que el tiempo total no exceda T_{max} y el capital al regresar a Ámsterdam sea al menos $K_{objetivo}$?*

A continuación, se demuestra que este problema es **NP-Completo**.

3.1. Pertenencia a la Clase NP

Sea un certificado de solución una secuencia de puertos $\pi = (v_0, v_1, \ldots, v_m, v_{m+1})$ y una secuencia de conjuntos de transacciones (compras C_i y ventas M_i') para cada puerto visitado. La verificación del certificado consiste en:

1. **Verificación de ruta:** Comprobar que π es una ruta simple (excepto por v_0 y v_{m+1}) y que el tiempo total de navegación cumple $\sum d_{v_i,v_{i+1}} \leq T_{max}$. Esta operación se realiza en $O(m)$ mediante un vector de presencia para los nodos.

2. **Verificación de carga:** En cada puerto v_i, se calcula el peso total sumando las mercancías retenidas en la bodega L_i después de vender y las nuevas adquisiciones $C_i \subseteq M_i$, verificando que $\sum_{g \in L_i} w_g + \sum_{C_i} w \leq C_{max}$. La complejidad es $O(m \cdot \max |L_i|)$, lo cual es polinomial respecto a la cantidad de mercancías.

3. **Verificación financiera:** Simular el flujo de caja K_i sumando ingresos por ventas M_i' y restando costos de compra C_i. Se verifica que en todo momento $K_i \geq K_{min}$ y que al regresar a Ámsterdam el capital final $K_f \geq K_{objetivo}$.

Como todas las operaciones de verificación se realizan en tiempo polinomial respecto al tamaño de la entrada, el problema **pertenece a NP**.

3.2. NP-Completitud mediante Reducción de TSP

Para demostrar la NP-Completitud, se reduce una instancia del problema de decisión de **TSP** (NP-Completo) a una instancia del problema.

> ***Definición de TSP:*** *Dado un grafo $G_{TSP} = (V', E', \omega)$ y una distancia límite L, ¿existe un tour que visite todos los nodos de V' tal que la distancia recorrida sea a lo sumo L?*
>
> $\omega(e_{ij})$: distancia entre nodos i y j.

3.2.1. Construcción de la reducción:

Dada una instancia de **TSP**, se construye una instancia del problema de la siguiente forma:

- **Grafo y Tiempos:** $V = V'$, $d_{ij} = peso(E'_{ij})$, y se fija $T_{max} = L$.
- **Incentivo a la Visita:** En cada puerto $i \in V \setminus \{0\}$, se define una única mercancía $m_i \in M_i$ con $w = 0$ y $p_{compra} = 0$. Su precio de venta es 0 en todo puerto, excepto en Ámsterdam (v_0), donde $p_{venta} = 1$.
- **Condiciones Iniciales:** $K_0 = 0$, $K_{min} = 0$, $C_{max} = \infty$.
- **Meta de Ganancia:** Se define la meta de capital final como $K_{objetivo} = |V| - 1$.

3.2.2. Demostración de equivalencia:

- ($\Rightarrow$) Si **TSP** tiene un tour de longitud de a lo sumo L, el capitán puede seguir dicho tour visitando cada puerto una vez, recoger las $|V| - 1$ mercancías y volver a Ámsterdam dentro del tiempo T_{max}. Al venderlas, obtiene $K_f = |V| - 1$, satisfaciendo el problema.
- ($\Leftarrow$) Si el problema tiene solución con capital de al menos K, el capitán necesariamente visitó todos los puertos para recolectar el capital (dada la ausencia de otros beneficios). Como lo hizo en un tiempo menor de T_{max}, el recorrido representa un tour válido para **TSP**.

La reducción es ejecutable en tiempo polinomial y dado que **TSP** se reduce al problema, concluimos que es **NP-Completo**.

4. Diseño de soluciones algorítmicas

Para resolver el problema, se han diseñado dos soluciones por fuerza bruta y una solución eficiente que combina un algoritmo voraz (*greedy*) con metaheurística (*simulate annealing*). Todas tienen las mismas entradas, emplean las mismas estructuras de datos y producen una salida.

4.1. Entradas

- n: número de puertos.
- $d[i][j]$: matriz de distancias entre puertos (d_{ij}).
- $T_{\text{máx}}$: tiempo máximo permitido para viajar.
- $C_{\text{máx}}$: capacidad máxima de carga del barco.
- K_0: capital inicial disponible.
- $K_{\text{mín}}$: capital mínimo requerido en todo momento.
- M_i: conjunto de ítems disponibles en el puerto i, cada uno con:
  - w_{ik}: peso del ítem k en el puerto i.
  - p_{ik}^{buy}: precio de compra del ítem.
  - p_{ik}^{sell}: precio de venta del ítem.

4.2. Salida

- Ganancia máxima obtenida tras visitar los puertos, comprar y vender ítems, respetando las restricciones de tiempo y capacidad.

4.3. Estructuras de datos utilizadas

- **Item**: representa un ítem disponible en un puerto. Contiene:
  - w: peso del ítem (w_{ik}).
  - *buy_price*: precio de compra (p_{ik}^{buy}).
  - *sell_price*: precio de venta (p_{ik}^{sell}).
- **Merchandise**: representa una mercancía que ya ha sido comprada y está a bordo. Contiene:
  - i: puerto en el que fue adquirida.
  - k: índice del ítem en ese puerto.
  - w: peso de la mercancía.
  - *buy_price*: precio de compra pagado.
- **itemsByPort**: matriz de dimensión $n \times m$ que contiene los ítems disponibles en cada puerto.
  - n: número de puertos.
  - m: número de ítems por puerto (se asume igual en todos los puertos por comodidad).
  - La ausencia de un ítem se representa como un objeto `Item` con:
    - peso $w = \infty$,
    - precio de compra $p^{buy} = \infty$,
    - precio de venta $p^{sell} = -\infty$.
- **State**: representa el estado completo del comerciante en un momento dado de la simulación. Contiene:
  - *current_port*: puerto actual en el que se encuentra.
  - *capital*: capital disponible en ese instante.
  - *time*: tiempo acumulado de viaje y operaciones.
  - *used_capacity*: capacidad de carga utilizada en el barco.

- *route*: lista de puertos visitados en orden.
- *inventory*: lista de mercancías (`Merchandise`) que están actualmente a bordo.

4.4. Descripción de las soluciones

El Algoritmo 1 es una de las implementaciones de fuerza bruta, que se describe a continuación.

Descripción del flujo del algoritmo. El algoritmo comienza en la función **Solve**, que inicializa el estado de los puertos visitados y llama a la función **Buy** para explorar las decisiones iniciales de compra.

La función **Buy** evalúa, en cada puerto, si es posible adquirir una mercancía dadas las restricciones de capital mínimo y capacidad de carga. Si la compra es viable, la mercancía se añade temporalmente a la carga y se realiza una llamada recursiva para continuar explorando. También se contempla la opción de no comprar y avanzar al siguiente ítem. En todo momento se compara la ganancia obtenida por comprar con la ganancia de viajar directamente (**Travel**).

La función **Sell** se encarga de explorar las posibles ventas de las mercancías que están a bordo en un puerto dado. Para cada mercancía se considera la opción de venderla o mantenerla, actualizando el capital y la capacidad, y llamando recursivamente a sí misma para evaluar el resto de mercancías. El resultado es la mejor ganancia entre vender y no vender.

La función **Travel** evalúa la posibilidad de desplazarse a otros puertos no visitados, siempre que el tiempo restante lo permita. Marca el puerto como visitado, llama a **Sell** para explorar las transacciones en ese puerto, y luego desmarca el puerto para permitir otras rutas. El valor devuelto es la máxima ganancia alcanzable considerando todas las rutas posibles.

En conjunto, el flujo del algoritmo es una búsqueda exhaustiva recursiva que combina decisiones de compra, venta y viaje, manteniendo en todo momento las restricciones de tiempo, capacidad y capital mínimo.

Correctitud. El algoritmo es correcto porque explora todas las combinaciones posibles de decisiones (comprar, vender, viajar o no hacerlo) y siempre mantiene las restricciones globales. Al devolver el máximo entre las ganancias obtenidas en cada rama de decisión, garantiza que el resultado final corresponde a la mejor estrategia posible. La recursividad asegura que no se omite ninguna ruta ni combinación de transacciones.

Complejidad Temporal El algoritmo explora todas las posibles rutas que comienzan en Ámsterdam y regresan dentro del tiempo máximo permitido. La clave está en que no basta con elegir un subconjunto de puertos: también importa el **orden en que se visitan**. Por eso, lo que se explora son **variaciones sin repetición**: todas las secuencias posibles de longitud k tomadas de n puertos. El número de secuencias de longitud k es:

$$P(n,k) = \frac{n!}{(n-k)!}.$$

Dado que el algoritmo permite rutas de cualquier longitud k (desde 0 hasta n), el total de rutas posibles es:

$$\sum_{k=0}^{n} P(n,k).$$

Esta sumatoria es del orden de $n!$ (tiende a $e \cdot n!$). En general:

- Exploración de rutas: $\Theta(n!)$, por el orden de visita de los puertos.
- Decisiones de compra/venta: $\Theta(2^m)$ por puerto, ya que cada ítem puede comprarse/venderse o no.
- Complejidad Total:
$$O(n! \cdot 2^m).$$

lo que lo hace impracticable para instancias grandes. Sin embargo, es útil como referencia exacta para instancias pequeñas y como base para comparar heurísticas.

El Algoritmo 2 es otra implementación de fuerza bruta, que se describe a continuación.

Descripción del flujo del algoritmo. El algoritmo comienza en la función **Solve**, que prepara las estructuras iniciales y llama a la función interna **_Solve**. Esta función recursiva construye rutas posibles desde Ámsterdam, explorando todos los puertos alcanzables dentro del tiempo máximo $T_{\text{máx}}$. Cada vez que se regresa al puerto inicial, se evalúa la ruta completa mediante la función **_ProfitByRoute**.

La función **_ProfitByRoute** calcula la ganancia máxima posible siguiendo la secuencia de puertos de la ruta. En cada puerto se consideran todas las combinaciones de ventas (**_Sell**) de las mercancías a bordo y todas las combinaciones de compras (**_Buy**) de las mercancías disponibles en el puerto.

Después de aplicar las decisiones de compra y venta, se actualizan las cargas y se continúa recursivamente hacia el siguiente puerto de la ruta. La función **_SellAll** se utiliza para liquidar todas las mercancías al final del viaje, y **_CalculateWeight** asegura que las restricciones de capacidad se respeten.

En conjunto, el flujo del algoritmo es una búsqueda exhaustiva que combina la exploración de rutas posibles con la evaluación de todas las combinaciones de compra y venta en cada puerto, manteniendo las restricciones de tiempo, capacidad y capital mínimo.

Correctitud. El algoritmo es correcto porque explora todas las rutas factibles que comienzan y terminan en Ámsterdam dentro del tiempo máximo, y para cada ruta considera exhaustivamente todas las combinaciones de compra y venta de mercancías. Al devolver siempre el máximo entre las ganancias obtenidas en cada rama de decisión, garantiza que el resultado final corresponde a la mejor estrategia posible. La recursividad asegura que no se omite ninguna ruta ni ninguna combinación de transacciones.

Complejidad Temporal. El algoritmo explora todas las posibles rutas de longitud variable que comienzan en Ámsterdam y regresan dentro del tiempo máximo permitido. No basta con elegir un subconjunto de puertos: también importa el **orden en que se visitan**. Por eso, lo que se explora son **variaciones sin repetición**: todas las secuencias posibles de longitud k tomadas de n puertos. El número de secuencias de longitud k es:

$$P(n,k) = \frac{n!}{(n-k)!}.$$

Dado que el algoritmo permite rutas de cualquier longitud k (desde 0 hasta n), el total de rutas posibles es:

$$\sum_{k=0}^{n} P(n,k).$$

Esta sumatoria es del orden de $n!$ (tiende a $e \cdot n!$).

En cuanto a las decisiones de compra y venta, en cada puerto se exploran todos los subconjuntos de mercancías a bordo para vender ($2^{|L|}$ posibilidades) y todos los subconjuntos de mercancías disponibles para comprar (2^m posibilidades). En el peor caso, $|L| \leq m$, por lo que el espacio de decisiones por puerto es $O(4^m)$.

- Exploración de rutas: $\Theta(n!)$, por el orden de visita de los puertos.

- Decisiones de compra/venta: $O(4^m)$ por puerto, considerando todas las combinaciones de venta y compra.
- Complejidad Total:

$$O(n! \cdot 4^m).$$

Esto lo hace impracticable para instancias grandes, pero útil como referencia exacta para instancias pequeñas y como base para comparar heurísticas.

El Algoritmo 3 es una implementación aproximada que combina una solución inicial *greedy* con el método de *Simulated Annealing*, y se describe a continuación.

Descripción del flujo del algoritmo. El algoritmo comienza en la función **Solve**, que genera una solución inicial mediante la función **GreedyInitialSolution**. Esta solución se construye recorriendo los puertos de manera heurística, vendiendo y comprando mercancías según convenga, y asegurando que se pueda regresar a Ámsterdam dentro del tiempo máximo permitido.

La función **GreedyInitialSolution** utiliza tres funciones auxiliares:

- **SellInCurrentPort**: vende las mercancías en el puerto actual si el precio de venta es mayor que el de compra.
- **BuyInCurrentPort**: selecciona mercancías prometedoras en el puerto actual usando una heurística tipo mochila 0/1, priorizando la relación ganancia/peso.
- **SelectNextPortGreedy**: elige el próximo puerto a visitar en función de una puntuación heurística que combina distancia y cantidad de ítems disponibles.

Al finalizar la ruta *greedy*, la función **SellAllInAmsterdam** liquida todo el inventario en el puerto inicial para calcular el capital final de la solución inicial.

Una vez obtenida la solución inicial, la función **SimulatedAnnealing** se encarga de mejorarla. Este procedimiento mantiene un estado actual y un estado óptimo encontrado hasta el momento. En cada iteración:

- Se genera una ruta vecina mediante **GenerateNeighbor**, que modifica la ruta actual aplicando operaciones como intercambio, inserción, eliminación o inversión de segmentos.

- La ruta vecina se evalúa con **SimulateRoute**, que simula el recorrido completo aplicando las funciones de compra y venta en cada puerto y verificando las restricciones de capacidad y capital.
- El capital final de la ruta simulada se calcula con **EvaluateState**, que vende todo el inventario en Ámsterdam.
- El criterio de aceptación de **SimulatedAnnealing** permite aceptar soluciones peores con cierta probabilidad dependiente de la temperatura, lo que ayuda a escapar de óptimos locales.

El proceso de enfriamiento reduce gradualmente la temperatura, haciendo que el algoritmo se vuelva más estricto en la aceptación de soluciones. Al finalizar las iteraciones, se devuelve el mejor estado encontrado.

En conjunto, el flujo del algoritmo combina una construcción inicial *greedy* con una búsqueda aproximada basada en *Simulated Annealing*, explorando rutas vecinas y evaluando las decisiones de compra y venta de mercancías para aproximarse a una solución de alta calidad.

5. Análisis empírico

Con el objetivo de evaluar el desempeño del algoritmo eficiente frente al enfoque de fuerza bruta, se realizaron una serie de experimentos variando tanto el número de ítems como el número de puertos en las instancias del problema. El análisis completo se encuentra en el directorio `../src/validation`.

5.1. Calidad de la solución

Al comparar la calidad de la solución obtenida por el algoritmo eficiente frente a la solución óptima de fuerza bruta tanto para variaciones de puertos (ver Figura 1) como variaciones de items (ver Figura 2) la solución eficiente siempre se mantiene sobre el 70 % de la solución óptima.

5.2. Tiempos de ejecución

El algoritmo de fuerza bruta presenta tiempos prácticamente nulos para instancias muy pequeñas, pero crecen rápidamente conforme aumenta el tamaño. Mientras que el algoritmo eficiente mantiene tiempos muy bajos y estables en todas las instancias. Este comportamiento se observa en las comparaciones de ambos algoritmos, en las Figura 3 y Figura 4

5.3. Tamaño máximo de instancia para fuerza bruta

El algoritmo de fuerza bruta puede resolver en tiempos razonables (menos de un minuto ver Figura 5) instancias de 3 puertos y hasta 10 items. A partir de 6 puertos con 4 items, los tiempos crecen de forma exponencial (comportamiento esperado por la complejidad) y superan los 60 segundos (ver Figura 6).

5.4. Comportamiento del algoritmo eficiente

Para las instancias del problema donde la solución óptima es no viajar o hacer ventas en puertos inmediatos a los puertos donde se realizó la compra el algoritmo eficiente devuelve una solución cercana al óptimo. Sin embargo, para aquellas instancias donde la mejor estrategia para una mercancía es venderla en un puerto muy distante a donde se compró (con respecto a la cantidad de puertos que los separan en la ruta), funciona peor porque se basa en no retener mercancías con ganancia conocida.

En instancias pequeñas casi siempre alcanza el óptimo, dado que el espacio de soluciones es pequeño y potencialmente lo explora en casi su totalidad.

Anexos

Algorithm 1 Fuerza Bruta. Joel Aaparicio Tamayo

```
1: Function Solve(n, d, T_máx, C_máx, K_0, K_mín, itemsByPort)
2: Initialize portsVisited ← list of False
3: portsVisited[0] ← True
4: m ← number of items at port 0
5: return máx(K_0, Buy(n, d, T_máx, C_máx, K_0, K_mín, m,
6:         itemsByPort, 0, [ ], portsVisited, 0))
7:
8: Function Sell(n, d, T_máx, C_máx, K_0, K_mín, size,
9:         itemsByPort, port, itemsOnBoard, portsVisited, j)
10: purchaseGain ← Buy(...)
11: if j ≥ length of itemsOnBoard then
12:    return purchaseGain
13: end if
14: if itemsOnBoard not empty then
15:    Remove item m at position j
16:    sellPrice ← itemsByPort[port][m.k].sellPrice
17:    sellGain ← Sell(...) with updated capital and capacity
18:    Reinsert item m
19:    sellGain ← máx(sellGain, Sell(..., j + 1))
20: end if
21: return máx(purchaseGain, sellGain)
22:
23: Function Buy(n, d, T_máx, C_máx, K_0, K_mín, size,
24:         itemsByPort, port, itemsOnBoard, portsVisited, j)
25: travelGain ← K_0
26: if K_mín > travelGain then
27:    return −1
28: end if
29: travelGain ← Travel(...)
30: if j ≥ size then
31:    return travelGain
32: end if
33: if K_mín ≤ funds and capacityLeft ≥ 0 then
34:    Append to itemsOnBoard: Merchandise(i = port, k = j,
35:          w = currentItem.w, buyPrice = currentItem.buyPrice)
36:    purchase ← Buy(..., j + 1) with updated capital and capacity
```

```
    Remove last item from itemsOnBoard
end if
purchase ← max(purchase, Buy(..., j + 1)) with current capital and capacity
return máx(purchase, travelGain)

Function Travel(n, d, T_máx, C_máx, K_0, K_mín, size,
        itemsByPort, port, itemsOnBoard, portsVisited)
finalGain ← K_0 + ∑ selling prices at initial port
maxGain ← −1
for each port i do
    if not visited and enough time then
        Mark port i as visited
        nextGain ← Sell(...) at port i
        maxGain ← máx(maxGain, nextGain)
        Unmark port i
    end if
end for
return maxGain
```

Algorithm 2 Fuerza Bruta. Claudia Hernández Pérez

```
Function Solve(n, d, T_máx, C_máx, K_0, K_mín, itemsByPort)
return _Solve(n, d, T_máx, C_máx, K_0, K_mín, itemsByPort,
        0, K_0, [False for i = 1..n], [ ] , [[ ] for i = 1..n])

Function _Solve(n, d, T_máx, C_máx, K_0, K_mín, itemsByPort,
        port_i, maxProfit, visited, route, L_i)
if port_i = 0 and visited[0] = True then
    Append 0 to route
    maxProfit ← máx(maxProfit,
            _ProfitByRoute(C_máx, K_0, K_mín, itemsByPort,
                route, L_i, 0, maxProfit))
    Remove last element from route
    return maxProfit
end if
Append port_i to route
for each port j in 0..n − 1 do
    if port_i ≠ port_j and not visited[j] and
```

```
18:            T_máx − (d[port_i][j] + d[j][0]) ≥ 0 then
19:        visited[j] ← True
20:        maxProfit ← máx(maxProfit,
21:                _Solve(n, d, T_máx − d[port_i][j], C_máx, K_0,
22:                        K_mín, itemsByPort, j, maxProfit, visited, route, L_i))
23:        visited[j] ← False
24:    end if
25: end for
26: Remove last element from route
27: return maxProfit
28:
29: Function _ProfitByRoute(C_máx, K_0, K_mín, itemsByPort,
30:         route, L_i, i, maxProfit)
31: if K_0 < 0 then
32:    return −∞
33: end if
34: port ← route[i]
35: if i = |route| − 1 then
36:    return K_0+ _SellAll(L_i[port], itemsByPort[port])
37: end if
38: for each sell in _Sell(L_i[port], itemsByPort[port], [ ], 0, [ ]) do
39:    rest ← L_i[port] \ sell[0]
40:    sold ← sell[0]
41:    capacitySold ← _CalculateWeight(sold)
42:    for each buy in _Buy(port, itemsByPort[port], [ ], 0, [ ],
43:            C_máx + capacitySold, K_0 + sell[1] − K_mín) do
44:        capacityBought ← _CalculateWeight(buy[0])
45:        L_i[route[i + 1]] ← rest + buy[0]
46:        maxProfit ← máx(maxProfit,
47:                _ProfitByRoute(C_máx + capacitySold − capacityBought,
48:                        K_0 + sell[1] − K_mín − buy[1], K_mín,
49:                        itemsByPort, route, L_i, i + 1, maxProfit))
50:    end for
51: end for
52: return maxProfit
53:
54: Function _Sell(goods, itemsAtPort, goodsSold, i, result)
55: if i = |goods| then
56:    sold ← ∑_{j∈goodsSold} itemsAtPort[j.k].sellPrice
57:    Append (goodsSold, sold) to result
58:    return result
```

```
end if
merchandise ← goods[i]
Append merchandise to goodsSold
_Sell(goods, itemsAtPort, goodsSold, i + 1, result)
Remove last element from goodsSold
_Sell(goods, itemsAtPort, goodsSold, i + 1, result)
return result

Function _SellAll(goods, itemsAtPort)
return ∑_{j∈goods} itemsAtPort[j.k].sellPrice

Function _Buy(port, itemsAtPort, goodsSold, i, result, C_máx, K)
if i = |itemsAtPort| then
    cost ← ∑_{j∈goodsSold} j.buyPrice
    Append (goodsSold, cost) to result
    return result
end if
merchandise ← Merchandise(port, i, itemsAtPort[i].w,
        itemsAtPort[i].buyPrice)
if C_máx − merchandise.w ≥ 0 and K − merchandise.buyPrice ≥ 0 then
    Append merchandise to goodsSold
    _Buy(port, itemsAtPort, goodsSold, i + 1, result,
            C_máx − merchandise.w, K − merchandise.buyPrice)
    Remove last element from goodsSold
end if
_Buy(port, itemsAtPort, goodsSold, i + 1, result, C_máx, K)
return result

Function _CalculateWeight(goods)
return ∑_{item∈goods} item.w
```

Algorithm 3 Solución Eficiente

```
Function Solve(n, d, T_máx, C_máx, K_0, K_mín, itemsByPort)
initialState ← GreedyInitialSolution()
if |initialState.route| = 1 then
    return initialState.capital
end if
bestState ← SimulatedAnnealing(initialState, maxIter = 2000)
```

```
finalCapital ← EvaluateState(bestState, itemsByPort)
return máx(finalCapital, K_0)

Function GreedyInitialSolution()
state ← State(current_port = 0, capital = K_0, time = 0,
        usedCapacity = 0, route = [0], inventory = [ ])
unvisited ← {1..n − 1}
while unvisited ≠ ∅ do
    state ← SellInCurrentPort(state, itemsByPort)
    nextPort ← SelectNextPortGreedy(state, unvisited, d, T_máx)
    if nextPort = None then
        break
    end if
    state ← BuyInCurrentPort(state, itemsByPort, nextPort)
    if state.capital < K_mín then
        break
    end if
    state.capital ← state.capital − K_mín
    state.time ← state.time + d[state.current_port][nextPort]
    state.current_port ← nextPort
    Append nextPort to state.route
    Remove nextPort from unvisited
end while
if state.current_port ≠ 0 then
    state.time ← state.time + d[state.current_port][0]
    state.capital ← state.capital − K_mín
    state.current_port ← 0
    Append 0 to state.route
    state ← SellAllInAmsterdam(state, itemsByPort)
end if
return state

Function SellInCurrentPort(state, itemsByPort)
newState ← state.copy()
newInventory ← [ ]
for each merch in state.inventory do
    (i, k, w, buyPrice) ← merch
    if i ≠ state.current_port then
        sellPrice ← itemsByPort[state.current_port][k].sellPrice
        if sellPrice > buyPrice then
            newState.capital ← newState.capital + sellPrice
```

```
      else
        Append merch to newInventory
      end if
    else
      Append merch to newInventory
    end if
  end for
  newState.inventory ← newInventory
  return newState

  Function BuyInCurrentPort(state, itemsByPort, nextPort)
  newState ← state.copy()
  availableCapacity ← C_máx − state.usedCapacity
  availableCapital ← state.capital − K_mín
  candidates ← [ ]
  for each item k in itemsByPort[state.current_port] do
    if item.w ≤ availableCapacity and item.buyPrice ≤ availableCapital
    then
      profit ← itemsByPort[nextPort][k].sellPrice − item.buyPrice
      if profit > 0 then
        Append (k, item.w, item.buyPrice, profit) to candidates
      end if
    end if
  end for
  Sort candidates by profit/weight descending
  for each (k, w, buyPrice, profit) in candidates do
    if availableCapacity ≥ w and availableCapital ≥ buyPrice then
      newState.capital ← newState.capital − buyPrice
      newState.usedCapacity ← newState.usedCapacity + w
      Append Merchandise(port, k, w, buyPrice) to newState.inventory
      availableCapacity ← availableCapacity − w
      availableCapital ← availableCapital − buyPrice
    end if
  end for
  return newState

  Function SelectNextPortGreedy(state, unvisited, d, T_máx)
  bestPort ← None
  bestScore ← −∞
  current ← state.current_port
  for each port in unvisited do
```

```
    travelTime ← d[current][port]
    timeBack ← d[port][0]
    totalTime ← state.time + travelTime + timeBack
    if totalTime ≤ T_máx then
      distanceScore ← 1/(travelTime + 10^−6)
      itemsCount ← |itemsByPort[port]|
      itemScore ← mín(itemsCount/10, 1)
      score ← distanceScore + itemScore
      if score > bestScore then
        bestScore ← score
        bestPort ← port
      end if
    end if
  end for
  return bestPort

  Function SellAllInAmsterdam(state, itemsByPort)
  newState ← state.copy()
  for each merch in state.inventory do
    (i, k, w, buyPrice) ← merch
    sellPrice ← itemsByPort[0][k].sellPrice
    newState.capital ← newState.capital + sellPrice
  end for
  newState.inventory ← [ ]
  return newState

  Function EvaluateState(state, itemsByPort)
  newState ← SellAllInAmsterdam(state, itemsByPort)
  return newState.capital

  Function SimulateRoute(route, d, itemsByPort)
  if route = [ ] or route[0] ≠ 0 then
    return None
  end if
  state ← State(current_port = 0, capital = K_0, time = 0,
          usedCapacity = 0, route = [0], inventory = [ ])
  for i = 1 to |route| − 1 do
    state ← SellInCurrentPort(state, itemsByPort)
    nextPort ← route[i]
    travelTime ← d[state.current_port][nextPort]
    if state.time + travelTime > T_máx then
```

```
        return None
    end if
    state ← BuyInCurrentPort(state, itemsByPort, nextPort)
    if state.capital < 0 or state.usedCapacity > C_máx then
        return None
    end if
    state.time ← state.time + travelTime
    state.current_port ← nextPort
    state.capital ← state.capital − K_mín
    Append nextPort to state.route
end for
state ← SellAllInAmsterdam(state, itemsByPort)
return state

Function GenerateNeighbor(state)
route ← state.route.copy()
if |route| = 3 then
    available ← {1..n − 1} \ route
    if available ≠ ∅ then
        insertPos ← random position in [1, |route| − 2]
        insertPort ← random choice from available
        Insert insertPort at insertPos in route
    end if
    return route
end if
operation ← random choice in {swap, insert, remove, reverse}
if operation = swap then
    Swap two random ports in route (not Amsterdam)
else if operation = insert then
    Insert a random unvisited port into route (excluding first and last positions)
else if operation = remove then
    Remove a random port from route (not Amsterdam)
else if operation = reverse then
    Reverse a random segment of route (excluding first and last positions)
end if
return route

Function SimulatedAnnealing(initialState, maxIter)
currentState ← initialState
bestState ← initialState.copy()
```

$currentValue \leftarrow$ EvaluateState($currentState, itemsByPort$)
$bestValue \leftarrow currentValue$
$temperature \leftarrow 1000$
$coolingRate \leftarrow 0{,}995$
for $iter = 1$ to $maxIter$ **do**
  $newRoute \leftarrow$ GenerateNeighbor($currentState$)
  $newState \leftarrow$ SimulateRoute($newRoute, d, itemsByPort$)
  if $newState = None$ **then**
    continue
  end if
  $newValue \leftarrow$ EvaluateState($newState, itemsByPort$)
  $\Delta \leftarrow newValue - currentValue$
  if $\Delta > 0$ **or** random() $< e^{\Delta / temperature}$ **then**
    $currentState \leftarrow newState$
    $currentValue \leftarrow newValue$
    if $currentValue > bestValue$ **then**
      $bestState \leftarrow currentState.copy()$
      $bestValue \leftarrow currentValue$
    end if
  end if
  $temperature \leftarrow temperature \cdot coolingRate$
end for
return $bestState$

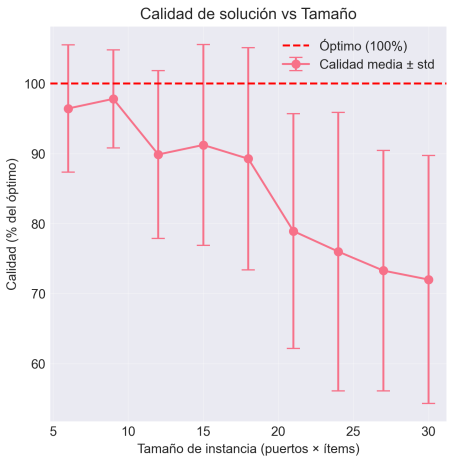

Figura 1: Calidad con puertos fijos

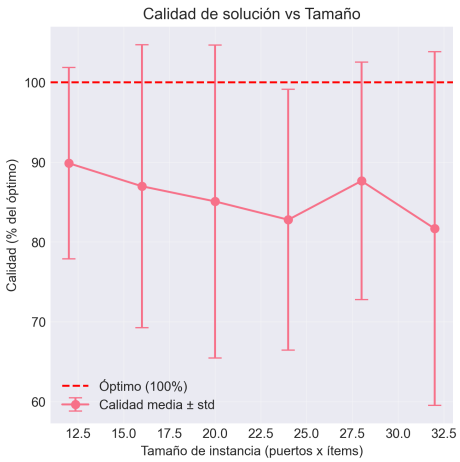

Figura 2: Calidad con items fijos

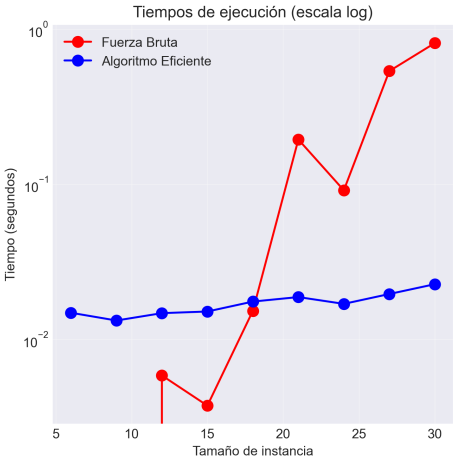

Figura 3: Tiempos con puertos fijos

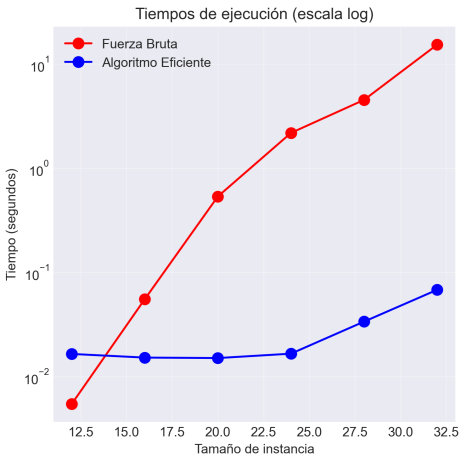

Figura 4: Tiempo con items fijos

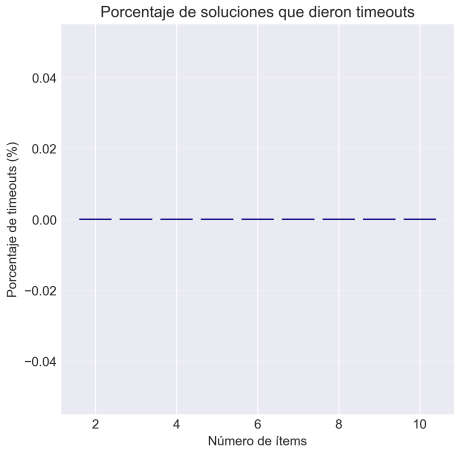

Figura 5: Timeouts con puertos fijos

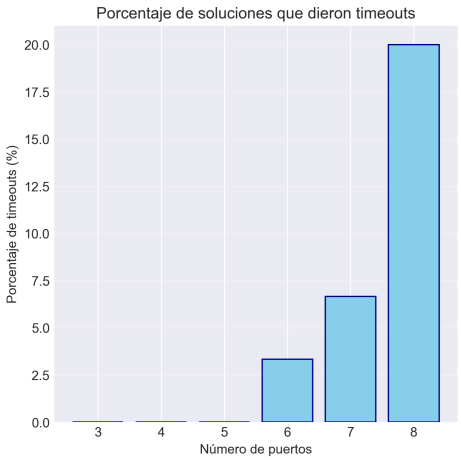

Figura 6: Timeouts con items fijos