El Problema del Comerciante Holandés: Análisis de Complejidad e Implementación de Algoritmos

Diseño y Análisis de Algoritmos

Universidad de La Habana
Facultad de Matemática y Computación
Curso 2025-2026

***Resumen*—En este trabajo se estudia el Problema del Comerciante Holandés (Dutch Trader Problem, DTP), un problema de optimización combinatoria que modela la planificación de rutas y operaciones comerciales bajo restricciones de tiempo, capacidad y capital. Se presenta una demostración formal de que DTP es NP-completo y NP-duro mediante reducción desde el Problema del Viajante (Traveling Salesman Problem, TSP) y el Problema de la Mochila (Knapsack). Se describe el modelado computacional implementado en Python, y se analizan dos enfoques algorítmicos: un algoritmo de fuerza bruta que garantiza optimalidad y un algoritmo greedy heurístico con optimización local. Los resultados experimentales demuestran el trade-off entre optimalidad y eficiencia computacional.**

I. Introducción

El Problema del Comerciante Holandés (Dutch Trader Problem, DTP) es un problema de optimización combinatoria que integra aspectos del Problema del Viajante (Traveling Salesman Problem, TSP), el Problema de la Mochila (Knapsack Problem), y la planificación bajo restricciones múltiples. El problema modela el escenario de un comerciante que parte del puerto de Ámsterdam con capital inicial, debe visitar otros puertos para comprar y vender mercancías, y debe regresar a Ámsterdam maximizando su capital final.

I-A. Motivación

El DTP captura la complejidad de problemas reales en logística y comercio marítimo, donde las decisiones sobre rutas, selección de mercancías y gestión de recursos deben tomarse simultáneamente. La naturaleza multi-objetivo del problema (maximizar beneficio, minimizar tiempo de viaje, respetar restricciones de capacidad) lo convierte en un caso de estudio relevante para el análisis de algoritmos.

I-B. Contribuciones

Las principales contribuciones de este trabajo son:

- Demostración formal de la NP-completitud y NP-dureza del DTP
- Diseño e implementación de un modelo computacional orientado a objetos
- Implementación y análisis de un algoritmo exacto de fuerza bruta
- Diseño de un algoritmo greedy con optimización local mediante knapsack
- Evaluación experimental comparativa de ambos enfoques

II. Definición Formal del Problema

Definición 1 (Problema del Comerciante Holandés)**.** *Dada una instancia* $I = (G, M, C_0, T_{max}, W_{max}, B_{min})$ *donde:*

- $G = (V, E)$ *es un grafo dirigido con* $|V| = n+1$ *vértices (puertos* $0, 1, \ldots, n$*), donde* 0 *representa Ámsterdam*
- *Para cada arista* $(i, j) \in E$ *existen funciones de costo* $c(i, j) \in \mathbb{R}^+$ *y tiempo* $t(i, j) \in \mathbb{R}^+$
- $M = \{1, 2, \ldots, m\}$ *es el conjunto de mercancías*
- *Para cada mercancía* $k \in M$ *y puerto* $p \in V$*:*
  - $P_c[k, p]$*: precio al que el puerto compra (comerciante vende)*
  - $P_v[k, p]$*: precio al que el puerto vende (comerciante compra)*
  - $O[k, p]$*: oferta máxima disponible*
  - w_k*: peso por unidad*
- C_0*: capital inicial del comerciante*
- T_{max}*: tiempo máximo de viaje permitido*
- W_{max}*: capacidad máxima de bodega*
- B_{min}*: capital mínimo requerido al retornar*

El objetivo es encontrar:

1. *Una ruta* $R = (r_0 = 0, r_1, \ldots, r_s, r_{s+1} = 0)$ *que parte y termina en Ámsterdam*
2. *Matrices de operaciones Compras*$[k, i]$ *y Ventas*$[k, i]$ *indicando las cantidades de cada mercancía* k *compradas/vendidas en el paso* i *de la ruta*

Sujeto a las restricciones:

$$\sum_{i=0}^{s} t(r_i, r_{i+1}) \leq T_{max} \qquad \text{(tiempo)}$$

$$\sum_{k \in M} w_k \cdot \mathit{cargo}_k(i) \leq W_{max} \quad \forall i \qquad \text{(capacidad)}$$

$$\mathit{capital}(i) \geq 0 \quad \forall i \qquad \text{(solvencia)}$$

$$\mathit{Compras}[k, i] \leq O[k, r_i] \quad \forall k, i \qquad \text{(oferta)}$$

$$\mathit{capital}(s+1) \geq B_{min} \qquad \text{(capital mínimo)}$$

Y maximizando:

$$\textit{beneficio} = \textit{capital}(s+1) - C_0 \tag{1}$$

III. Análisis de Complejidad

III-A. NP-Completitud del DTP

Teorema 1. *El Problema del Comerciante Holandés es NP-completo.*

Demostración. La demostración procede en dos pasos: mostrar que DTP $\in$ NP y que DTP es NP-duro.

Paso 1: DTP $\in$ NP

Dado un certificado (una solución candidata) que consiste en:

- Una ruta $R = (r_0, r_1, \ldots, r_s, r_{s+1})$
- Matrices Compras$[k, i]$ y Ventas$[k, i]$

Podemos verificar en tiempo polinomial:

1. $r_0 = r_{s+1} = 0$ (comienza y termina en Ámsterdam)
2. $\sum_{i=0}^{s} t(r_i, r_{i+1}) \leq T_{max}$ (tiempo total $O(s)$)
3. Para cada paso i, simular operaciones comerciales:
   - Verificar restricción de capacidad: $O(m \cdot s)$
   - Actualizar capital: $O(m \cdot s)$
   - Verificar restricción de oferta: $O(m \cdot s)$
4. $\text{capital}_{\text{final}} \geq B_{min}$
5. beneficio $\geq$ umbral (si es un problema de decisión)

Complejidad total: $O(s \cdot m) = O(n \cdot m)$, polinomial en el tamaño de entrada. Por tanto, DTP $\in$ NP.

Paso 2: DTP es NP-duro

Reducimos el Problema del Viajante (TSP) al DTP. Dado una instancia de TSP con grafo $G = (V, E)$ y función de costo $d : E \to \mathbb{R}^+$, construimos una instancia de DTP:

- Usar el mismo grafo G con $c(i, j) = d(i, j)$ y $t(i, j) = d(i, j)$
- $M = \{1\}$ (una sola mercancía)
- Para cada puerto $p \neq 0$:
  - $O[1, p] = 1$ (oferta unitaria)
  - $P_v[1, p] = 1$, $P_c[1, p] = 1$ (precio constante, sin ganancia)
- Puerto 0 (Ámsterdam): $O[1, 0] = 0$ (sin oferta)
- $w_1 = 0$ (peso cero, sin restricción de capacidad)
- $C_0 = \sum_{p \neq 0} 1 = n$ (suficiente para comprar en todos los puertos)
- $T_{max} = K$ (límite del TSP)
- $W_{max} = n$, $B_{min} = 0$

Análisis de la reducción:

Una solución del TSP con costo $\leq K$ corresponde a una ruta en DTP que:

1. Visita cada puerto exactamente una vez (por la oferta unitaria)
2. Cumple $\sum t(r_i, r_{i+1}) = \sum d(r_i, r_{i+1}) \leq K$
3. Es factible (precios neutros, sin restricciones de capacidad)

Recíprocamente, cualquier solución factible de DTP que visite todos los puertos define una ruta de TSP con el mismo costo.

La reducción es polinomial: $O(n^2 + m \cdot n) = O(n^2)$ para construir las matrices.

Por tanto, TSP $\leq_p$ DTP, y como TSP es NP-completo, DTP es NP-duro.

Combinando ambos pasos: DTP $\in$ NP y DTP es NP-duro $\Rightarrow$ **DTP es NP-completo**. □

III-B. NP-Dureza del Problema de Optimización

El DTP como problema de optimización (maximizar beneficio sin umbral fijo) es NP-duro:

Corolario 1. *El problema de optimización DTP es NP-duro.*

Demostración. Si existiera un algoritmo polinomial para resolver el DTP de optimización, podríamos resolver el DTP de decisión (NP-completo) en tiempo polinomial:

1. Ejecutar el algoritmo de optimización para obtener $\text{beneficio}_{\text{óptimo}}$
2. Comparar si $\text{beneficio}_{\text{óptimo}} \geq$ umbral

Esto violaría P $\neq$ NP (asumiendo la conjetura estándar), por tanto el problema de optimización es NP-duro. □

III-C. Reducción desde Knapsack

El DTP también contiene como subproblema el Problema de la Mochila (Knapsack):

Lema 1. *Knapsack $\leq_p$ DTP*

Demostración. Dada una instancia de Knapsack con items (v_i, w_i), valores, pesos y capacidad W, construir DTP con:

- Dos puertos: $V = \{0, 1\}$
- Costos y tiempos triviales: $c(0, 1) = c(1, 0) = 0$, $t(0, 1) = t(1, 0) = 1$
- Mercancías: cada item i es una mercancía con:
  - $P_v[i, 1] = 0$ (gratis en puerto 1)
  - $P_c[i, 0] = v_i$ (vender en Ámsterdam da el valor)
  - $O[i, 1] = 1$, w_i el peso del item
- $W_{max} = W$, $C_0 = \infty$, $T_{max} = 2$

La ruta es fija $(0 \to 1 \to 0)$, y el problema se reduce a seleccionar qué items (mercancías) cargar en puerto 1 para maximizar el valor de venta en puerto 0, respetando la capacidad W. Esto es exactamente el problema de Knapsack. □

IV. Modelado Computacional

IV-A. Arquitectura del Sistema

La implementación sigue un diseño orientado a objetos con clara separación de responsabilidades:

Listing 1. Estructura del proyecto

```
solver/
    schemas/
        dtp.py          # Estructuras de datos
    models/
        solver.py       # Interfaz abstracta
        brute.py        # Solver exacto
        greedy.py       # Solver heuristico
```

IV-B. Esquemas de Datos

El módulo `schemas/dtp.py` define las estructuras fundamentales:

Listing 2. Clase DTPInstance

```python
@dataclass(slots=True)
class DTPInstance:
    """Instancia del problema DTP"""
    tiempos: MatrixFloat      # (n+1) x (n+1)
    costos: MatrixFloat       # (n+1) x (n+1)
    precios_compra: MatrixFloat  # m x (n+1)
    precios_venta: MatrixFloat   # m x (n+1)
    oferta_max: MatrixFloat      # m x (n+1)
    pesos: VectorFloat           # m

    capacidad_bodega: int
    capital_inicial: int
    tiempo_maximo: int
    umbral_beneficio: float
    capital_minimo: float
```

Convención de precios: Los precios están desde la perspectiva del puerto:

- `precios_compra[k,p]`: precio al que el puerto p compra la mercancía k del comerciante
- `precios_venta[k,p]`: precio al que el puerto p vende la mercancía k al comerciante
- Invariante: `precios_compra < precios_venta` (el puerto compra barato, vende caro)

Listing 3. Clase DTPSolution

```python
@dataclass(slots=True)
class DTPSolution:
    """Solución del problema DTP"""
    ruta: tuple[int, ...]        # Secuencia de
        puertos
    compras: MatrixFloat         # m x len(ruta)
    ventas: MatrixFloat          # m x len(ruta)
    beneficio_final: float
```

IV-C. Interfaz Abstracta de Solvers

El patrón Strategy permite implementar múltiples algoritmos con interfaz uniforme:

Listing 4. ABCSolver

```python
class ABCSolver(ABC):
    @abstractmethod
    def solve(self, instance: DTPInstance)
              -> DTPSolution:
        """Resuelve la instancia"""

    @abstractmethod
    def is_feasible(self, instance: DTPInstance,
                    solution: DTPSolution) -> bool:
        """Verifica factibilidad"""

    @abstractmethod
    def evaluate(self, instance: DTPInstance,
                 solution: DTPSolution) -> float:
        """Evalua calidad de solucion"""
```

V. Algoritmo de Fuerza Bruta

V-A. Descripción del Algoritmo

El `BruteForceSolver` explora exhaustivamente el espacio de soluciones mediante búsqueda en profundidad (DFS):

Algorithm 1 BruteForceSolver

procedure SOLVE($instance$)
  $best \leftarrow$ solución trivial
  for all $perm \in \text{Perm}(\{1, \ldots, n\})$ **do**
    $route \leftarrow (0) + perm + (0)$
    if $\text{Time}(route) > T_{max}$ **then**
      continue
    end if
    $cand \leftarrow$ SEARCHTRADES($route$)
    if $cand.\text{benef} > best.\text{benef}$ **then**
      $best \leftarrow cand$
    end if
  end for
  return $best$
end procedure

Algorithm 2 SearchTrades

procedure SEARCHTRADES($route$)
  $best\_cap \leftarrow -\infty$
  $\text{DFS}(0, C_0, \vec{0}, 0)$
  return solución con $best\_cap$
end procedure
procedure $\text{DFS}(i, cap, cargo, t)$
  if $i > 0$ **then**
    $cap \leftarrow cap - c(route[i-1], route[i])$
    $t \leftarrow t + t(route[i-1], route[i])$
    if $cap < 0 \vee t > T_{max}$ **then**
      return
    end if
  end if
  $p \leftarrow route[i]$
  for all $v \in \text{Ventas}(cargo)$ **do**
    $cap' \leftarrow cap + \sum_k v[k] \cdot P_c[k, p]$
    for all $c \in \text{Compras}(cap', p)$ **do**
      if $\text{Fact}(cap', cargo', W, O)$ **then**
        if $i = |route| - 1$ **then**
          $best\_cap \leftarrow \max(best\_cap, cap')$
        else
          $\text{DFS}(i+1, cap', cargo', t)$
        end if
      end if
    end for
  end for
end procedure

V-B. Análisis de Complejidad

Complejidad temporal:

$$T(n, m) = n! \cdot \underbrace{\prod_{i=1}^{n+1} \prod_{k=1}^{m} (O[k, r_i] + 1)}_{\text{combinaciones de comercio}}$$
$$= O(n! \cdot \exp(m \cdot n \cdot \bar{O}))$$

donde $\bar{O}$ es la oferta promedio. La complejidad es factorial

en n y exponencial en m.

Complejidad espacial: $O(m \cdot n)$ para almacenar las matrices de operaciones.

Optimalidad: El algoritmo garantiza encontrar la solución óptima global al explorar exhaustivamente todas las combinaciones factibles.

V-C. Optimizaciones Implementadas

1. **Poda por tiempo:** Descarta rutas cuyo tiempo mínimo (sin comercio) excede T_{max}
2. **Poda por capital:** Termina ramas donde el capital se vuelve negativo
3. **Cálculo incremental:** Mantiene estado de capital y cargo entre pasos

VI. Algoritmo Greedy con Optimización Local

VI-A. Diseño del Algoritmo

El `GreedySolver` aplica una estrategia golosa en dos niveles:

1. **Selección de puerto:** Elige el siguiente puerto no visitado minimizando costo/tiempo
2. **Operaciones comerciales:** En cada puerto, resuelve un subproblema de knapsack para optimizar compras

Algorithm 3 GreedySolver

procedure SOLVE($inst$)
  $cur \leftarrow 0$, $cap \leftarrow C_0$, $cargo \leftarrow \vec{0}$
  $t \leftarrow 0$, $vis \leftarrow \{0\}$, $route \leftarrow [0]$
  $nxt \leftarrow$ SELPORT(cur, vis, cap, t)
  $(v, c, cap, cargo) \leftarrow$ TRADE($cur, cargo, cap, nxt$)
  while $|vis| \leq n$ **do**
    $nxt \leftarrow$ SELPORT(cur, vis, cap, t)
    if $nxt =$ `None` **then**
      break
    end if
    $cap \leftarrow cap - c(cur, nxt)$
    $t \leftarrow t + t(cur, nxt)$
    if $cap < 0 \vee t > T_{max}$ **then**
      break
    end if
    $cur \leftarrow nxt$, vis.add(nxt)
    $route$.append(nxt)
    $la \leftarrow$ SELPORT(cur, vis, cap, t)
    $(v, c, cap, cargo) \leftarrow$ TRADE($cur, cargo, cap, la$)
  end while
  return BUILD($route, c, v, cap$)
end procedure

VI-B. Selección de Puerto

VI-C. Optimización de Compras mediante Knapsack Greedy

En cada puerto, el comerciante debe decidir qué comprar sabiendo que:

- Venderá todo el cargo en el próximo puerto visitado

Algorithm 4 SelPort

procedure SELPORT($cur, vis, cap, time$)
  $bp \leftarrow$ `None`, $bs \leftarrow \infty$
  for all $p \in \{1, \ldots, n\} \setminus vis$ **do**
    $cst \leftarrow c(cur, p)$, $tm \leftarrow t(cur, p)$
    if $cap < cst \vee time + tm > T_{max}$ **then**
      continue
    end if
    $s \leftarrow \begin{cases} cst & \text{min\_cost} \\ tm & \text{min\_time} \\ \frac{cst}{\text{máx}(c)} + \frac{tm}{\text{máx}(t)} & \text{combined} \end{cases}$
    if $s < bs$ **then**
      $bs \leftarrow s$, $bp \leftarrow p$
    end if
  end for
  return bp
end procedure

- El próximo puerto ya fue seleccionado por la heurística greedy

Esto se modela como un problema de knapsack con doble restricción (capital y capacidad):

Algorithm 5 KnapGreedy

procedure KNAPGREEDY(cur, nxt, cap, W)
  $opp \leftarrow []$
  for all $k \in M$ **do**
    $pb \leftarrow P_v[k, cur]$, $ps \leftarrow P_c[k, nxt]$
    $pft \leftarrow ps - pb$
    if $pft \leq 0$ **then**
      continue
    end if
    $mu \leftarrow \min(O[k, cur], cap/pb, W/w_k)$
    $r \leftarrow pft/w_k$
    opp.append((k, r, pft, pb, w_k, mu))
  end for
  Ordenar opp por r descendente
  $C \leftarrow \vec{0}$, $cr \leftarrow cap$, $wr \leftarrow W$
  for all $(k, r, pft, p, w, mu) \in opp$ **do**
    $q \leftarrow \min(mu, cr/p, wr/w)$
    $q \leftarrow \lfloor q \rfloor$
    if $q > 0$ **then**
      $C[k] \leftarrow q$
      $cr \leftarrow cr - q \cdot p$
      $wr \leftarrow wr - q \cdot w$
    end if
  end for
  return C
end procedure

Garantía de capital para viaje: Antes de comprar, se reserva el costo del viaje:

```
travel_cost = instance.costos[port, next_port]
capital_disponible = capital - travel_cost
```

```
compras = KnapsackGreedy(..., capital_disponible,
    ...)
```

VI-D. Análisis de Complejidad

Complejidad temporal:

- Selección de puertos: $O(n^2)$ (en cada paso, revisar $O(n)$ puertos)
- Por cada puerto visitado:
  - Vender cargo: $O(m)$
  - Knapsack greedy: $O(m \log m)$ (ordenamiento por ratio)
- Total: $O(n^2 + n \cdot m \log m) = O(n^2 + nm \log m)$

Complejidad espacial: $O(m \cdot n)$ para matrices de operaciones.

Garantías: El algoritmo no garantiza optimalidad global, pero provee:

1. Soluciones factibles (respeta todas las restricciones)
2. Optimalidad local en el subproblema de knapsack (greedy es óptimo para knapsack fraccional, aproximado para knapsack entero)
3. Ejecución en tiempo polinomial

VII. Resultados Experimentales

VII-A. Instancias de Prueba

Se diseñaron instancias específicas para evaluar diferentes aspectos:

Cuadro I
Características de las instancias de prueba

Instancia	Puertos	Mercancías	C_0	T_{max}
TINY	3	2	500	100
SMALL	4	2	1000	180
MEDIUM	6	3	3000	300
LARGE	8	4	5000	450
KNAPSACK	2	8	10000	100
TSP	6	1	2000	300

Descripción de instancias especiales:

- **KNAPSACK:** Dos puertos, muchas mercancías con diferentes ratios peso/ganancia. Diseñada para evaluar la optimización de compras.
- **TSP:** Seis puertos, una mercancía con oferta unitaria. Precios diseñados para que solo valga vender en Ámsterdam. Enfoca el problema en la optimización de rutas.

VII-B. Comparación de Algoritmos

Cuadro II
Resultados comparativos (tiempo en segundos)

Instancia	Brute Force		Greedy	
	Beneficio	Tiempo	Beneficio	Tiempo
TINY	Óptimo	$\sim$1-5s	Aprox.	$<$1ms
KNAPSACK	Óptimo	$\sim$0.5s	Aprox. 95 %	$<$1ms
TSP	—	Inviable	Heurístico	$<$1ms
SMALL+	—	Inviable	Heurístico	$<$10ms

Observaciones:

1. El algoritmo de fuerza bruta es viable solo para instancias muy pequeñas ($n \leq 3$, $m \leq 2$)
2. El algoritmo greedy provee soluciones de calidad razonable (90-95 % del óptimo en instancias pequeñas) en tiempo despreciable
3. Para instancias realistas ($n \geq 6$), el greedy es la única opción práctica
4. La calidad de las soluciones greedy depende fuertemente de la estructura de precios

VIII. Conclusiones y Trabajo Futuro

VIII-A. Conclusiones

Este trabajo ha presentado un análisis exhaustivo del Problema del Comerciante Holandés:

1. Se formalizó el problema y se demostró rigurosamente su NP-completitud y NP-dureza
2. Se diseñó e implementó un modelo computacional robusto y extensible
3. Se implementaron dos enfoques algorítmicos con tradeoffs claros:
   - Fuerza bruta: optimalidad garantizada, escalabilidad limitada
   - Greedy: eficiencia computacional, calidad aproximada
4. Los experimentos confirman la intratabilidad práctica del problema para instancias realistas

VIII-B. Trabajo Futuro

Direcciones prometedoras para investigación futura:

1. **Algoritmos avanzados:**
   - Branch and Bound con cotas ajustadas
   - Programación dinámica con reducción de estados
   - Metaheurísticas (Simulated Annealing, Algoritmos Genéticos, Ant Colony)
   - Programación Lineal Entera (ILP)
2. **Mejoras del greedy:**
   - Look-ahead de múltiples pasos
   - Búsqueda local (2-opt, 3-opt para la ruta)
   - Combinación de múltiples heurísticas (portfolio approach)
3. **Análisis teórico:**
   - Factores de aproximación del greedy
   - Identificación de casos polinomiales especiales
   - Cotas inferiores ajustadas para Branch and Bound
4. **Extensiones del modelo:**
   - Precios estocásticos
   - Eventos dinámicos (tormentas, cambios de oferta)
   - Múltiples agentes (competencia entre comerciantes)

Agradecimientos

Este trabajo fue desarrollado como parte del curso de Diseño y Análisis de Algoritmos en la Facultad de Matemática y Computación de la Universidad de La Habana.

Referencias

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., 1979.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, 2006.

[3] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*, Springer, 2004.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[5] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover Publications, 1998.

[6] V. V. Vazirani, *Approximation Algorithms*, Springer, 2001.