Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Diversión NP-Completa

22 de noviembre de 2024

Victoria Avalos	108434
Santiago Tissoni	103856
Facundo Monpelat	92716

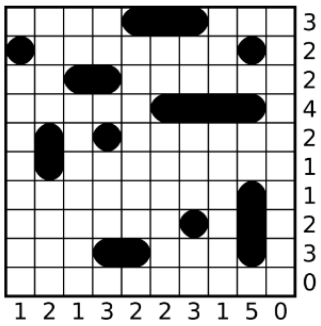
1. Introducción

En teoría de la complejidad computacional, un problema NP-completo es aquel que cumple con las siguientes características: pertenece a la clase NP, y a la vez cualquier otro problema que pertenezca a NP puede ser reducido polinomialmente a él.

En este trabajo práctico abordamos uno de estos problemas: la disposición de barcos en un tablero con restricciones específicas de ocupación y separación. Demostraremos que el problema es NP-completo mediante una reducción desde un problema conocido, lo que nos permitirá confirmar su dificultad en términos de complejidad computacional. Luego, implementaremos y compararemos diferentes técnicas para resolver el problema: backtracking como enfoque exhaustivo, programación lineal para modelarlo como un problema de optimización, y realizaremos una aproximación que busque soluciones eficientes sin necesariamente ser óptimas.

1.1. Descripción del Problema

El problema se define de la siguiente manera: tenemos un tablero de $n \times m$ casilleros y una cantidad de barcos k. Cada barco i tiene un largo fijo b_i, es decir, requiere b_i casilleros consecutivos para ser ubicado, en una única fila o columna. A su vez, cada fila y columna del tablero tiene un requisito de ocupación, que indica el número exacto de casilleros que deben estar ocupados. Adicionalmente, ningún barco puede estar adyacente a otro, ni en filas, ni en columnas, ni en diagonal. La solución debe cumplir con todas las restricciones establecidas, haciendo de este problema un reto complejo en términos de computación.



1.2. Objetivos

Los objetivos del presente trabajo práctico son los siguientes:

- Demostrar que el Problema de la Batalla Naval, en su versión de decisión, se encuentra en NP.
- Demostrar que el Problema de la Batalla Naval en su versión de decisión es, en efecto, un problema NP-Completo.
- Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema en la versión de optimización. Tomar mediciones de tiempos del mismo.
- Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.
- Utilizar un algoritmo de aproximación para resolver el problema, analizar su complejidad y analizar cuán buena aproximación es.

Resolución

2. ¿Por qué se trata de un problema NP-completo?

Dado un problema X del cual se quiere demostrar que es NP-completo, se debe cumplir que:

$$X \in \text{NP-Completo} \quad \text{si y solo si:}$$

I. $X \in \text{NP}$

II. $\forall Y \in \text{NP},\ Y \leq_p X$

A continuación, procederemos a demostrar estos ítems para probar que el problema pertenece a NP-completo. Para ello se utilizará la versión de decisión del problema:

Dado un tablero de $n \times m$ casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo), una lista de restricciones para las filas (donde la restricción j corresponde a la cantidad de casilleros a ser ocupados en la fila j) y una lista de restricciones para las columnas (símil filas, pero para columnas), ¿es posible definir una ubicación de dichos barcos de tal forma que se cumplan con las demandas de cada fila y columna, y las restricciones de ubicación?

2.1. ¿Pertenece a NP?

Un problema pertenece a NP si existe un certificador eficiente; siendo un certificador una función que dado el problema y una posible solución, determina si la propuesta es efectivamente una solución. En otras palabras, son aquellos problemas que se pueden validar en **tiempo polinomial**.

A lo largo del presente trabajo práctico, decidimos representar a los barcos de la siguiente forma: el tablero es una matriz donde cada barco se representa con el número de la posición que ocupa en el vector de barcos (comenzando por 1). Es decir, si los barcos son [3,1,2] el barco de longitud 3 se verá representado en el tablero como tres números 3 seguidos (horizontal o verticalmente, el segundo barco se verá como un número 2, y el tercer barco se verá como tres números 3.

Con esto en mente, implementamos el siguiente validador para demostrar que el problema pertenece a NP:

```
from collections import Counter

def validar_tablero(tablero, barcos, restricciones_filas, restricciones_columnas):

    n = len(tablero)
    m = len(tablero[0])

    for i, fila in enumerate(tablero):
        if sum(1 for x in fila if x != 0) != restricciones_filas[i]:
            return False

    for j in range(m):
        columna = [tablero[i][j] for i in range(n)]
        if sum(1 for x in columna if x != 0) != restricciones_columnas[j]:
            return False

    barco_contador = Counter()
    visitado = [[False] * m for _ in range(n)]

    for i in range(n):
        for j in range(m):
            if tablero[i][j] != 0 and not visitado[i][j]:
                valor = tablero[i][j]
                stack = [(i, j)]
                tama o_barco = 0
                while stack:
                    x, y = stack.pop()
```

```
                    if x < 0 or x >= n or y < 0 or y >= m or visitado[x][y] or
    tablero[x][y] != valor:
                        continue
                    visitado[x][y] = True
                    tama o_barco += 1
                    stack.append((x + 1, y))
                    stack.append((x - 1, y))
                    stack.append((x, y + 1))
                    stack.append((x, y - 1))
                barco_contador[valor] += 1
                if tama o_barco != barcos[valor - 1]:
                    return False

    for i, tama o in enumerate(barcos, start=1):
        if barco_contador[i] != 1:
            return False

    return True
```

El validador primero revisa que se cumplan las restricciones tanto de filas como de columnas sumando las posiciones ocupadas, caso en el cual devuelve True, y False de no cumplirse alguna demanda.

Luego, identifica y cuenta cada barco usando una búsqueda iterativa para medir su tamaño y verificar que coincida con el especificado. Para conseguir esto primero recorre cada casilla del tablero. Si encuentra una casilla con un barco y que aún no ha sido visitada, inicia una búsqueda en profundidad para encontrar todas las casillas conectadas a ese barco. Mientras explora el barco, cuenta cuántas casillas lo componen y verifica si este tamaño coincide con el tamaño esperado de ese tipo de barco. Si no coincide, devuelve False. Finalmente, verifica que todos los barcos dados en la lista se hayan usado exactamente una vez en el tablero.

Con respecto a la complejidad, primero se recorren todas las filas y se suman sus valores en las columnas, lo cual resulta ser de complejidad $\mathcal{O}(n \times m)$. Luego se recorren de igual forma las columnas y se suman los valores en las filas, lo cual vuelve a ser $\mathcal{O}(n \times m)$. Después se realiza un DFS que recorre todos los elementos del tablero, gracias a que se lleva registro de las casillas recorridas con la matriz *visitado*. Esto es $\mathcal{O}(n \times m)$.

En conclusión, la complejidad total del validador resulta ser $\mathcal{O}(n \times m) + \mathcal{O}(n \times m) + \mathcal{O}(n \times m) = \mathcal{O}(n \times m)$.

Debido a que la complejidad del validador es polinomial, queda demostrado que el problema pertenece a NP.

2.2. Reducción

El siguiente y último paso para demostrar que el problema de los barcos pertenece a NP-completo es reducir un problema NP-completo al problema de la Batalla Naval. Para ello elegimos el problema de Bin Packing en código unario: Dados un conjunto S de números, cada uno expresado en código unario, una cantidad de bins B, expresada en código unario, y la capacidad de cada bin C, expresada en código unario; donde la suma del conjunto de números es igual a C*B, debe decidir si puede cumplirse que los números pueden dividirse en B bins, subconjuntos disjuntos, tal que la suma de elementos de cada bin es exactamente igual a la capacidad C.

Para reducir este problema al de la Batalla Naval, vamos a realizar la siguiente transformación: crearemos un tablero de C filas y B columnas. Cada columna representará a uno de los recipientes (bins), y la restricción de cada una será la capacidad de los bins, es decir C. Para las filas, cada una de ellas tendrá como restricción la cantidad de bins, es decir B.

Los barcos representarán a los números del conjunto S de Bin Packing, y se pondrán de forma vertical en una columna. Los mismos no pueden superponerse entre sí.

Entonces, la relación con el problema de Bin Packing se da de la siguiente manera:

- Colocar los barcos verticalmente en las columnas corresponde a asignar elementos del con-

junto S a bins.

- Cumplir las restricciones de las columnas (sumar exactamente C) corresponde a cumplir con la capacidad de cada bin en el Bin Packing.
- Cumplir las restricciones de las filas (que cada fila esté completamente ocupada) asegura que todos los elementos del conjunto S se utilicen.

Planteamos la siguiente doble implicancia, siendo X el vector de barcos:

$$\exists \text{ Sol}(S) \text{ Bin Packing} \iff \exists \text{ Sol}(X) \text{ Batalla Naval}$$

Demostración de ambas implicancias por método directo:

1. $\exists$ Sol(S) Bin Packing $\Rightarrow \exists$ Sol(X) Batalla Naval

   Supongamos que existe una solución para una instancia del problema de Bin Packing. Esto significa que existe una partición del conjunto $S = \{s_1, s_2, \ldots, s_k\}$ en B subconjuntos disjuntos $S_1, S_2, \ldots, S_B$, tal que:

   $$\forall i \in \{1, \ldots, B\}, \quad \sum_{s \in S_i} s = C,$$

   donde C es la capacidad de cada bin, y la suma total satisface:

   $$\sum_{i=1}^{B} \sum_{s \in S_i} s = C \cdot B.$$

   En la reducción al problema de Batalla Naval:

   - Cada bin S_i se traduce en una columna del tablero, con una restricción de ocupación de C casillas.
   - Cada elemento $s_j \in S$ se convierte en un barco de tamaño s_j, que debe colocarse verticalmente en una de las columnas.

   Como en la solución de Bin Packing cada bin está lleno exactamente hasta su capacidad C, todas las columnas del tablero estarán ocupadas con exactamente C casillas, cumpliendo así las restricciones de las columnas. Además, dado que cada fila del tablero tiene como restricción la cantidad de bins B, y las columnas están completamente ocupadas sin dejar huecos, las restricciones de las filas también se cumplen:

   $$\forall j \in \{1, \ldots, C\}, \quad \sum_{i=1}^{B} \text{ocupación}(i, j) = B.$$

   Finalmente, como en la solución de Bin Packing se utilizan todos los elementos de S, todos los barcos se colocan en el tablero sin dejar ninguno fuera. Por lo tanto, existe una solución para el problema de Batalla Naval.

2. $\exists$ Sol(S) Bin Packing $\Leftarrow \exists$ Sol(X) Batalla Naval

   Supongamos ahora que existe una solución para una instancia del problema de Batalla Naval. Esto implica que:

   - Todas las restricciones de las columnas se cumplen, es decir:

     $$\forall i \in \{1, \ldots, B\}, \quad \text{ocupación de la columna } i = C.$$

- Todas las restricciones de las filas también se satisfacen:

$$\forall j \in \{1, \ldots, C\}, \quad \sum_{i=1}^{B} \text{ocupación}(i, j) = B.$$

En este contexto, cada columna del tablero corresponde a un bin en el problema de Bin Packing, y cada barco representa un elemento del conjunto S. Dado que cada columna está completamente ocupada con barcos cuya longitud total es C, esto asegura que cada bin contiene elementos cuya suma es exactamente C. Además, como todas las filas están completamente ocupadas y no quedan barcos sin ubicar, podemos concluir que todos los elementos de S han sido utilizados, cumpliendo la condición:

$$\sum_{i=1}^{B} \sum_{s \in S_i} s = C \cdot B.$$

Además, como los barcos están completamente dentro de las columnas y no se superponen, la partición de S entre los bins es válida. Por lo tanto, una solución para Batalla Naval implica una solución para Bin Packing.

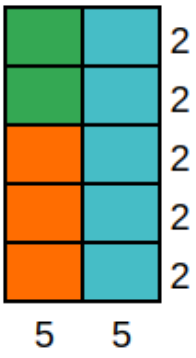
Ejemplo

Se cuenta con la siguiente instancia del problema de Bin Packing: se poseen 2 bins (B = 2) de capacidad 5 (C = 5), y el siguiente conjunto de números S = 5,3,2.

Nótese que una posible solución sería [5] [3,2].

La transformación del problema queda de la siguiente forma:

Y la solución resulta:



Justificación de que Bin Packing es NP-completo

Para que la demostración anterior tenga validez, justificaremos brevemente por qué Bin Packing es efectivamente NP-completo, utilizando un problema visto en clase: 2-Partition.

El problema de **2-Partition** consiste en determinar si, dado un conjunto de números $S = \{s_1, s_2, \ldots, s_k\}$, existe una partición en dos subconjuntos disjuntos S_1 y S_2 tal que:

$$\sum_{x \in S_1} x = \sum_{x \in S_2} x = \frac{\sum_{x \in S}}{2}.$$

Dada una instancia del problema 2-Partition, construimos una instancia del problema Bin Packing de la siguiente forma:

- El conjunto $S = \{s_1, s_2, \ldots, s_k\}$ será el conjunto de elementos a empacar.
- El número de recipientes (*bins*) será $B = 2$.
- La capacidad de cada recipiente será $C = \frac{\sum_{x \in S}}{2}$, que corresponde a la mitad de la suma total de S.

La relación entre los problemas se da de la siguiente forma: Si existe una partición válida para el problema Partition, los elementos de S pueden dividirse en dos subconjuntos S_1 y S_2 tales que:

$$\sum_{x \in S_1} = \sum_{x \in S_2} = C.$$

Esto significa que los elementos pueden distribuirse en dos recipientes de capacidad C, cumpliendo así las restricciones del problema Bin Packing.

Si existe una solución para la instancia de Bin Packing con $B = 2$ y $C = \frac{\sum_{x \in S}}{2}$, esto implica que los elementos de S se pueden empacar en dos recipientes de capacidad C. Por lo tanto, existe una partición de S en dos subconjuntos S_1 y S_2 cuya suma es C, resolviendo el problema Partition.

La transformación de una instancia de Partition a Bin Packing es claramente polinomial, ya que solo requiere calcular C (una suma y una división). Como Partition es un problema NP-completo, y hemos demostrado una reducción polinomial, concluimos que Bin Packing también es NP-completo.

3. Algoritmo por Backtracking

A continuación se muestra el código de solución por backtracking del problema.

```
# Funci n para verificar restricciones de colocaci n de un barco
def can_place(board, row, col, length, is_horizontal, row_demand, col_demand):
    is_vertical = not is_horizontal

    # Verificar que no salga del tablero
    if is_horizontal:
        if col + length > len(board[0]):
            return False

    if is_vertical:
        if row + length > len(board):
            return False

    # Verficiar las demandas
    if is_horizontal:
        if row_demand[row] - length < 0:
            return False

```

```
        for i in range(length):
            if col_demand[col + i] - 1 < 0:
                return False

        if col + length > len(board[0]):
            return False

    if is_vertical:
        for i in range(length):
            if row_demand[row + i] - 1 < 0:
                return False

        if col_demand[col] - length < 0:
            return False

        if row + length > len(board):
            return False

    # Verificar que no haya barcos adyacentes ni superpuestos
    for i in range(length):
        r, c = (row, col + i) if is_horizontal else (row + i, col)
        # Verificar que no haya barcos adyacentes en el borde del barco
        if i == 0 and is_horizontal:
            for dr, dc in ((0, -1), (-1, 0), (1, 0), (-1, -1), (1, -1)):
                if (
                    r + dr >= 0
                    and c + dc >= 0
                    and r + dr < len(board)
                    and c + dc < len(board[0])
                    and board[r + dr][c + dc] != 0
                ):
                    return False

        if i == 0 and is_vertical:
            for dr, dc in ((-1, 0), (-1, -1), (-1, 1), (0, -1), (0, 1)):
                if (
                    r + dr >= 0
                    and c + dc >= 0
                    and r + dr < len(board)
                    and c + dc < len(board[0])
                    and board[r + dr][c + dc] != 0
                ):
                    return False

        if i == length - 1 and is_horizontal:
            for dr, dc in ((0, 1), (-1, 1), (1, 1), (-1, 0), (1, 0)):
                if (
                    r + dr >= 0
                    and c + dc >= 0
                    and r + dr < len(board)
                    and c + dc < len(board[0])
                    and board[r + dr][c + dc] != 0
                ):
                    return False

        if i == length - 1 and is_vertical:
            for dr, dc in ((1, 0), (1, -1), (1, 1), (0, -1), (0, 1)):
                if (
                    r + dr >= 0
                    and c + dc >= 0
                    and r + dr < len(board)
                    and c + dc < len(board[0])
                    and board[r + dr][c + dc] != 0
                ):
                    return False

        # Verificar que la celda esten vacias
        if board[r][c] != 0:
            return False

```

```
89          # Verificar que no haya barcos adyacentes
90          if is_horizontal:
91              # Hay un barco adyacente en la fila de arriba
92              if r - 1 <= 0 and board[r - 1][c] != 0:
93                  return False
94              # Hay un barco adyacente en la fila de abajo
95              if r + 1 < len(board) and board[r + 1][c] != 0:
96                  return False
97
98          if is_vertical:
99              # Hay un barco adyacente en la columna de la izquierda
100             if c - 1 >= 0 and board[r][c - 1] != 0:
101                 return False
102             # Hay un barco adyacente en la columna de la derecha
103             if c + 1 < len(board[0]) and board[r][c + 1] != 0:
104                 return False
105
106     return True
107
108
109 # Funci n para actualizar demandas y tablero
110 def place_ship(
111     board,
112     row,
113     col,
114     barco,
115     is_horizontal,
116     row_demand,
117     col_demand,
118 ):
119     index, length = barco
120     for i in range(length):
121         r, c = (row, col + i) if is_horizontal else (row + i, col)
122         board[r][c] += index
123
124     if is_horizontal:
125         row_demand[r] -= length
126         for i in range(length):
127             col_demand[col + i] -= 1
128
129     if not is_horizontal:
130         col_demand[c] -= length
131         for i in range(length):
132             row_demand[row + i] -= 1
133
134
135 def unplace_ship(board, row, col, barco, is_horizontal, row_demand, col_demand):
136     index, length = barco
137     for i in range(length):
138         r, c = (row, col + i) if is_horizontal else (row + i, col)
139         board[r][c] -= index
140
141     if is_horizontal:
142         row_demand[r] += length
143         for i in range(length):
144             col_demand[col + i] += 1
145
146     if not is_horizontal:
147         col_demand[c] += length
148         for i in range(length):
149             row_demand[row + i] += 1
150
151
152 # Algoritmo de backtracking para ubicar barcos
153 def backtracking(board, barcos, row_demand, col_demand, mejor_solucion):
154     demanda_insatisfecha = sum(d for d in row_demand) + sum(d for d in col_demand)
155
156     # Actualizar mejor soluci n
157     if demanda_insatisfecha < mejor_solucion[1]:
158         mejor_solucion[1] = demanda_insatisfecha
```

```
        mejor_solucion[0] = [[cell for cell in row] for row in board]

    # Caso base
    if not barcos:
        return

    # backtrack
    if (
        demanda_insatisfecha - sum(barco[1] * 2 for barco in barcos)
        >= mejor_solucion[1]
    ):
        return

    # Caso: No colocar el barco actual
    backtracking(
        board,
        barcos[1:],
        row_demand,
        col_demand,
        mejor_solucion,
    )

    # Caso: Intentar colocar el barco actual en el tablero
    barco = barcos[0]
    for r in range(len(board)):
        if row_demand[r] == 0:
            continue
        for c in range(len(board[0])):
            if col_demand[c] == 0:
                continue
            for is_horizontal in (True, False):
                if can_place(
                    board, r, c, barco[1], is_horizontal, row_demand, col_demand
                ):
                    place_ship(
                        board,
                        r,
                        c,
                        barco,
                        is_horizontal,
                        row_demand,
                        col_demand,
                    )
                    backtracking(
                        board,
                        barcos[1:],
                        row_demand,
                        col_demand,
                        mejor_solucion,
                    )
                    unplace_ship(
                        board,
                        r,
                        c,
                        barco,
                        is_horizontal,
                        row_demand,
                        col_demand,
                    )


def problema(board, barcos, row_demand, col_demand, mejor_solucion):
    barcos.sort(key=lambda x: x[1], reverse=True)  # Ordenar por tama o del barco
    backtracking(board, barcos, row_demand, col_demand, mejor_solucion)
```

El problema presentado consiste en ubicar barcos en un tablero cumpliendo ciertas restricciones de demandas de filas y columnas, evitando conflictos en la disposición (como adyacencias entre barcos) y minimizando la demanda no satisfecha. Este tipo de problema pertenece a la categoría de optimización combinatoria y es adecuado para abordarse con backtracking, dado que implica

explorar exhaustivamente todas las posibles configuraciones de los barcos en el tablero, descartando las que no cumplen las restricciones.

3.1. Estructura del algoritmo

Verificación de restricciones: comprueba si un barco puede ser colocado en una posición dada del tablero. Las restricciones verificadas incluyen:

- Que el barco no exceda los límites del tablero.
- Que no viole las demandas de las filas o columnas.
- Que no entre en contacto con otro barco, ya sea en celdas adyacentes (en cualquier dirección, incluidas diagonales) o superpuestas.

Colocación y descolocación de los barcos: estas funciones actualizan el tablero y las demandas de las filas y columnas cuando se coloca o retira un barco. Esto permite retroceder al estado previo de manera eficiente, una característica esencial del backtracking que lo diferencia de un algoritmo por fuerza bruta.

Backtracking: explora todas las configuraciones posibles para colocar los barcos, considerando dos casos:

1. No colocar el barco actual: este caso permite omitir barcos cuando es más beneficioso no colocarlos para minimizar la demanda insatisfecha.
2. Intentar colocar el barco actual: se prueban todas las posiciones posibles en el tablero y en ambas orientaciones permitidas (horizontal y vertical).

Se utiliza una variable global para mantener la mejor configuración encontrada hasta el momento.

Criterios de poda: primero, los barcos se ordenan por tamaño en orden descendente para intentar primero aquellos que afectan más significativamente las demandas. Ademas, si la demanda insatisfecha potencialmente reducible con los barcos disponibles no es suficiente para alcanzar una nueva solución optima, no se continua por ese camino de posibles soluciones.

3.2. Justificación del uso de backtracking

El algoritmo utiliza backtracking porque:

- **Búsqueda exhaustiva con poda**: backtracking explora todas las configuraciones posibles del problema. Se evita explorar configuraciones que ya se sabe que no cumplirán las restricciones o no mejorarán la solución actual.
- **Reversión de estados**: permite colocar y descolocar barcos, restaurando las demandas de filas y columnas, facilitando la exploración de configuraciones alternativas.
- **Optimización**: aunque no se garantiza un rendimiento lineal (el problema es NP-Completo), el uso de técnicas como poda y ordenamiento de barcos mejora significativamente el rendimiento en comparación con una búsqueda completamente ciega.

3.3. Complejidad

En el peor de los casos debemos explorar todas las posibiles configuraciones, aunque esto en la practica no sucede ya que hemos implementado podas tipicas de un algoritmo de backtracking para reducir las dimesiones del espacio de busqueda.

Cada barco puede ser colocado o no colocado, y puede orientarse en dos direcciones (horizontal o vertical). Si asumimos un tablero de $n \times m$, el número de posibles posiciones para cada barco es:

$2 \cdot (n \cdot m)$. Por lo tanto, para k barcos, el número máximo de configuraciones posibles (sin contar restricciones) es: $(2 \cdot (n \cdot m))^k$

Por otro lado, por cada posible posicion de un barco, debemos verificar las restricciones (por ejemplo las adyacencias y las demandas)

1. Verificar las demandas de filas y columnas es O(1), ya que es una operación de comparación de dos numeros.
2. Verificar las adyacencias es $O(b_i)$, siendonosla b_i la longitud del barco i, ya que hay que recorrer cada posicion que ocuparia el barco para verificar si estan disponibles para colocar (tanto que no esten ocupadas como que no haya elementos adyacentes). En el peor de los casos esta complejidad es la longitud del barco mas largo b_{max}

De esta forma, concluimos que la **complejidad del algoritmo es** $O((2 \cdot (n \cdot m))^k \cdot b_{max})$, esto es un algoritmo con una complejdad supererior a polinomial: **complejidad exponencial**, como era esperado ya que de antemano sabiamos que el problema es NP-Completo, por lo que no tiene una solución en tiempo polinomial.

4. Algoritmo por Programación Lineal

Colocamos los barcos de longitudes predeterminadas en el tablero $n \times m$ de forma que se cumplan las siguientes restricciones:

1. Los barcos deben ser colocados horizontal o vertical en posiciones consecutivas.
2. Las casillas ocupadas deben cumplir con las demandas de cada fila y columna.
3. No se permite superposición de barcos ni adyacencia directa (ortogonal o diagonal).

El objetivo es minimizar las demandas insatisfechas de filas y columnas.

4.1. Modelo

El modelo de Programación Lineal incluye:

4.1.1. Variables de decisión

- $barcos\_h[i, j, k] \in \{0, 1\}$: Indica si el barco k se coloca horizontalmente con inicio en la posición (i, j).
- $barcos\_v[i, j, k] \in \{0, 1\}$: Indica si el barco k se coloca verticalmente con inicio en la posición (i, j).
- $d\_filas[i] \geq 0$: Demanda insatisfecha en la fila i.
- $d\_columnas[j] \geq 0$: Demanda insatisfecha en la columna j.

4.1.2. Parámetros

- b_k: Longitud del barco k.
- $demandas\_filas[i]$: Demanda de casillas ocupadas en la fila i.
- $demandas\_columnas[j]$: Demanda de casillas ocupadas en la columna j.

4.1.3. Función objetivo

Maximizar el cumplimiento total de las demandas, equivalente a minimizar la demanda insatisfecha:

$$\text{mín (Demanda insatisfecha total)} = \sum_{i=1}^{n} d\_filas[i] + \sum_{j=1}^{m} d\_columnas[j].$$

4.1.4. Restricciones

R1: Colocación única: Cada casilla del tablero (i, j) puede estar ocupada por a lo sumo un barco:

$$\sum_{k} barcos\_h[i, j, k] + \sum_{k} barcos\_v[i, j, k] \leq 1 \quad \forall i, j.$$

R2: Demanda de filas: La suma de casillas ocupadas en cada fila debe cumplir (parcialmente o completamente) con la demanda de esa fila:

$$\sum_{j=1}^{m} \sum_{k} \left(barcos\_h[i, j, k] \cdot b_k \right) + \sum_{j=1}^{m} \sum_{k} barcos\_v[i, j, k] \leq demandas\_filas[i] + d\_filas[i].$$

R3: Demanda de columnas: Análoga a la restricción anterior para columnas:

$$\sum_{i=1}^{n} \sum_{k} \left(barcos\_v[i, j, k] \cdot b_k \right) + \sum_{i=1}^{n} \sum_{k} barcos\_h[i, j, k] \leq demandas\_columnas[j] + d\_columnas[j].$$

R4: Consecutividad: Si un barco k se coloca horizontalmente, sus casillas $(i, j), (i, j + 1), \ldots, (i, j + b_k - 1)$ deben estar ocupadas, y de forma análoga para colocación vertical:

$$barcos\_h[i, j, k] \Rightarrow \text{Ocupar las siguientes } b_k \text{ casillas.}$$

$$barcos\_v[i, j, k] \Rightarrow \text{Ocupar las siguientes } b_k \text{ filas.}$$

R5: Adyacencia: Las casillas adyacentes (incluyendo diagonales) no deben estar ocupadas por otros barcos:

$$barcos\_h[i, j, k] + barcos\_v[i', j', k'] \leq 1 \quad \forall\, (i', j') \in \text{adyacentes a } (i, j).$$

4.2. Implementación

La solución se implementó en Python utilizando la biblioteca `PuLP`. Los pasos principales fueron:

1. Definir las variables de decisión y parámetros.
2. Construir las restricciones del modelo.
3. Resolver el problema con el método `problema.solve()`.
4. Validar la solución obtenida verificando el cumplimiento de las restricciones.

4.2.1. Código Principal

El código se estructuró de la siguiente manera:

- Definición del tablero como una matriz de dimensiones $n \times m$.
- Implementación de restricciones específicas utilizando bucles anidados para cada casilla (i, j) y cada barco k.
- Extracción del resultado en un formato interpretable, con el tablero final y el cálculo de demandas cumplidas.

4.2.2. Resultados obtenidos

- **Demanda cumplida:** Se logró ocupar X casillas de las demandas totales en filas y columnas.
- **Tablero generado:** Se obtuvo una representación visual del tablero indicando las posiciones exactas de los barcos.

5. Aproximación

Utilizamos el siguiente algoritmo de aproximación: Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.

```
def aproximar(row_demand, col_demand, ships, board):
    ships.sort(key=lambda x: x[1], reverse=True)
    demanda_cumplida = 0
    ship_indices = list(range(len(ships)))

    while ship_indices:
        max_row_demand = max(row_demand)
        max_col_demand = max(col_demand)

        if max_row_demand == 0 and max_col_demand == 0:
            break

        if max_row_demand >= max_col_demand:
            max_demand_index = row_demand.index(max_row_demand)
            placed_ship = False

            for i in ship_indices:
                index, ship_size = ships[i]
                if ship_size <= max_row_demand:
                    # Intentar colocar el barco horizontalmente en la fila
                    for j in range(len(board[0])):
                        if can_place(board, max_demand_index, j, ship_size, True, row_demand, col_demand):
                            place_ship(board, max_demand_index, j, ships[i], True, row_demand, col_demand)
                            demanda_cumplida += (ship_size * 2)
                            ship_indices.remove(i)
                            placed_ship = True
                            break
                    if placed_ship:
                        break

            if not placed_ship:  # Si no se coloco ningun barco, salir del bucle
                break

        else:
            max_demand_index = col_demand.index(max_col_demand)
            placed_ship = False

            # Intentar colocar los barcos en la columna con la demanda m xima
            for i in ship_indices:
                index, ship_size = ships[i]
                if ship_size <= max_col_demand:
                    # Intentar colocar el barco verticalmente en la columna
                    for j in range(len(board)):
                        if can_place(board, j, max_demand_index, ship_size, False, row_demand, col_demand):
                            place_ship(board, j, max_demand_index, ships[i], False, row_demand, col_demand)
                            demanda_cumplida += (ship_size * 2)
                            ship_indices.remove(i)
                            placed_ship = True
                            break
```

```
                    if placed_ship:
                        break

            if not placed_ship:  # Si no se coloco ningun barco, salir del bucle
                break

    return board, demanda_cumplida
```

Primero se ordenan de mayor a menor tamaño para intentar colocar primero los más grandes. Luego se pasa al ciclo principal: mientras haya barcos disponibles para colocar, el algoritmo busca la demanda máxima, comparando la mayor demanda entre filas y columnas. Dependiendo de si la mayor demanda está en las filas o columnas, el algoritmo procede a intentar colocar un barco en esa dirección. Además, si no quedan demandas por cumplir se sale del bucle.

Si la mayor demanda está en una fila, se identifica su índice (max_demand_index). Se recorre la lista de barcos y se intenta colocar uno cuyo tamaño no exceda la demanda de la fila. Para colocar un barco, se exploran todas las posiciones posibles en esa fila usando la función can_place, que verifica si es posible ubicar el barco en ese lugar. Para ello corrobora que se cumplan las demandas, que no se salga del rango del tablero, que las celdas estén vacías, y que no haya barcos adyacentes ni superpuestos. Si se encuentra un espacio válido, se coloca el barco con la función $place_ship$, que también actualiza el tablero, la demanda y remueve el índice del barco de la lista. Estas dos últimas funciones son las mismas que se utilizaron para el algoritmo de backtracking. Si la mayor demanda está en una columna, se realiza un proceso similar al de las filas, pero ahora se busca colocar un barco verticalmente en la columna de mayor demanda.

Si ningún barco se puede colocar para satisfacer la demanda máxima actual (en filas o columnas), el bucle principal termina. Al final, el algoritmo retorna el estado actualizado del tablero y el total de demanda cumplida, que corresponde a la suma de los tamaños de los barcos colocados exitosamente.

5.1. Complejidad

La complejidad de este algoritmo greedy en términos del tamaño del tablero $n \times m$ y la cantidad de barcos b es la siguiente:

- Ordenamiento de los barcos: $O(b \log b)$ donde b es la cantidad de barcos.
- Bucle principal: El bucle principal itera mientras haya barcos disponibles. En cada iteración:
  - Se encuentra la demanda máxima de fila o columna en $O(n + m)$, donde n y m son las dimensiones del tablero.
  - Se recorre la lista de barcos $O(b)$ para intentar colocar un barco. Esto involucra:
    - Un bucle que recorre las posiciones posibles de la fila o columna, lo cual toma $O(m)$ en el caso de la colocación horizontal o $O(n)$ en el caso de la colocación vertical.

El número total de operaciones en el peor caso es $O(b^2 \cdot (n + m))$, ya que:

- El bucle principal recorre los barcos b veces.
- Dentro de cada iteración, se recorre la lista de barcos nuevamente (b) y se verifica la colocación en las posiciones posibles, lo que toma $O(n)$ o $O(m)$.

Por lo tanto, la complejidad total es:

$$O(b^2 \cdot (n + m))$$

5.2. Cuán buena aproximación es

Sea I una instancia cualquiera del problema de La Batalla Naval, y $z(I)$ una solución óptima para dicha instancia. Sea $A(I)$ la solución aproximada. Se define que:

$$\frac{A(I)}{z(I)} \leq r(A)$$

para todas las instancias posibles.

Al calcular el valor de r(A) con diferentes test cases se obtiene lo siguiente:

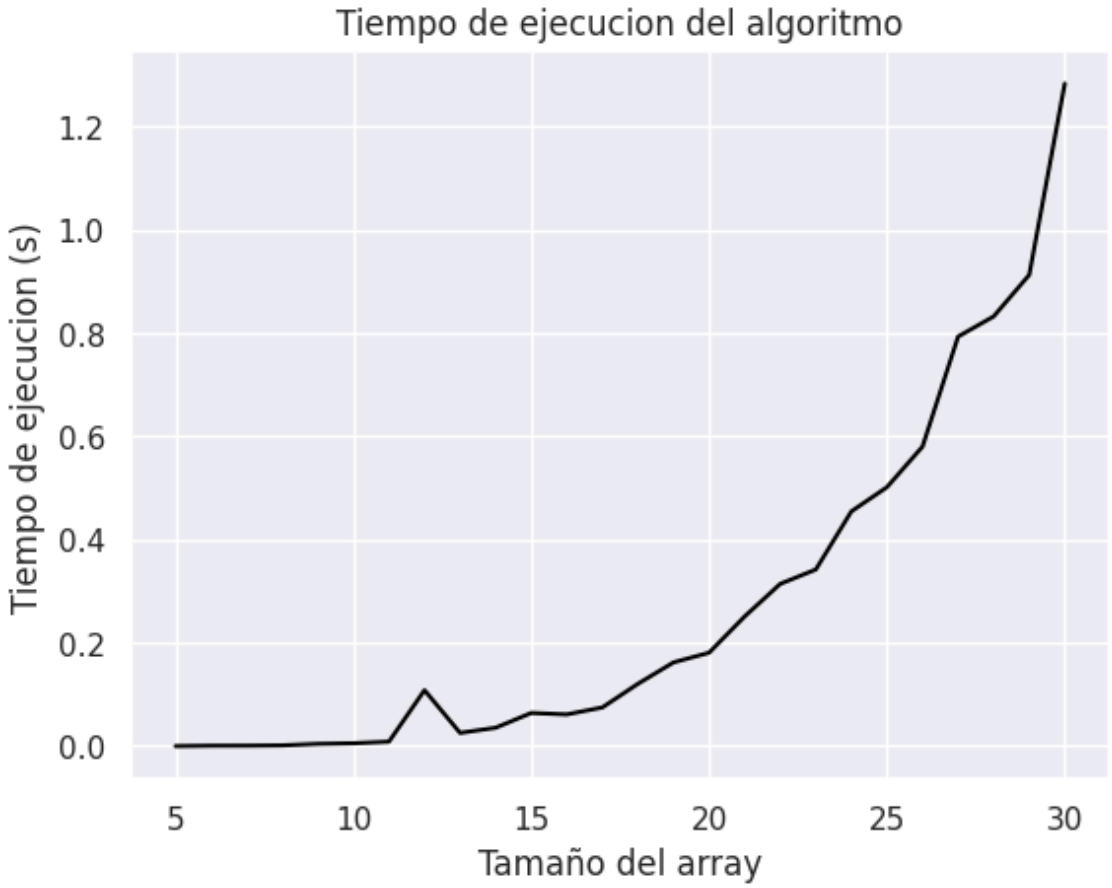
```
El factor para 3_3_2.txt es: 0.5
El factor para 5_5_6.txt es: 1.0
El factor para 8_7_10.txt es: 0.75
El factor para 10_3_3.txt es: 1.0
El factor para 10_10_10.txt es: 0.8947368421052632
El factor para 12_12_21.txt es: 0.6666666666666666
El factor para 15_10_15.txt es: 0.76
```

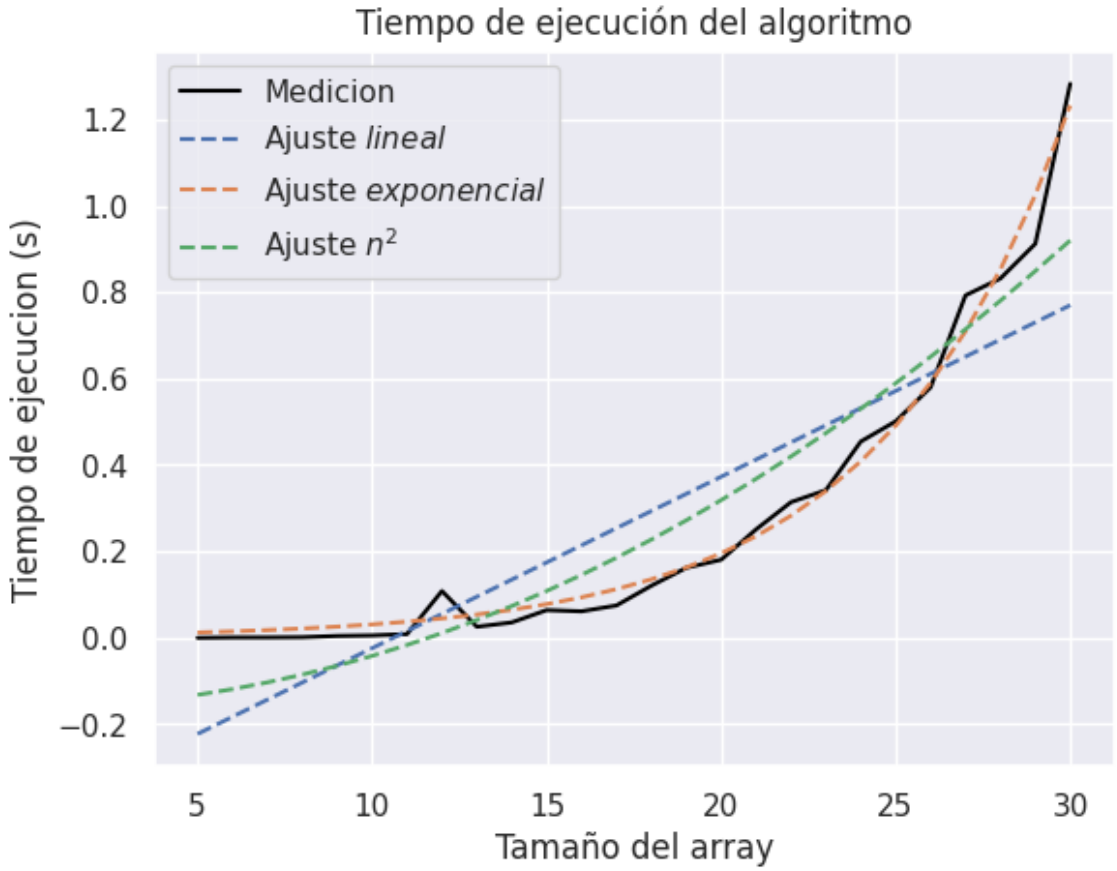
Se puede apreciar que el factor de aproximación se encuentra entre 0.5 y 1.

6. Mediciones

En este apartado del informe realizaremos mediciones para los diferentes algoritmos abordados. Las mismas se realizaron en base a crear diferentes tamaños de tableros, con vectores de barcos. El tamaño de dicho vector es proporcional a las medidas del tablero (un tercio).

6.1. Backtracking

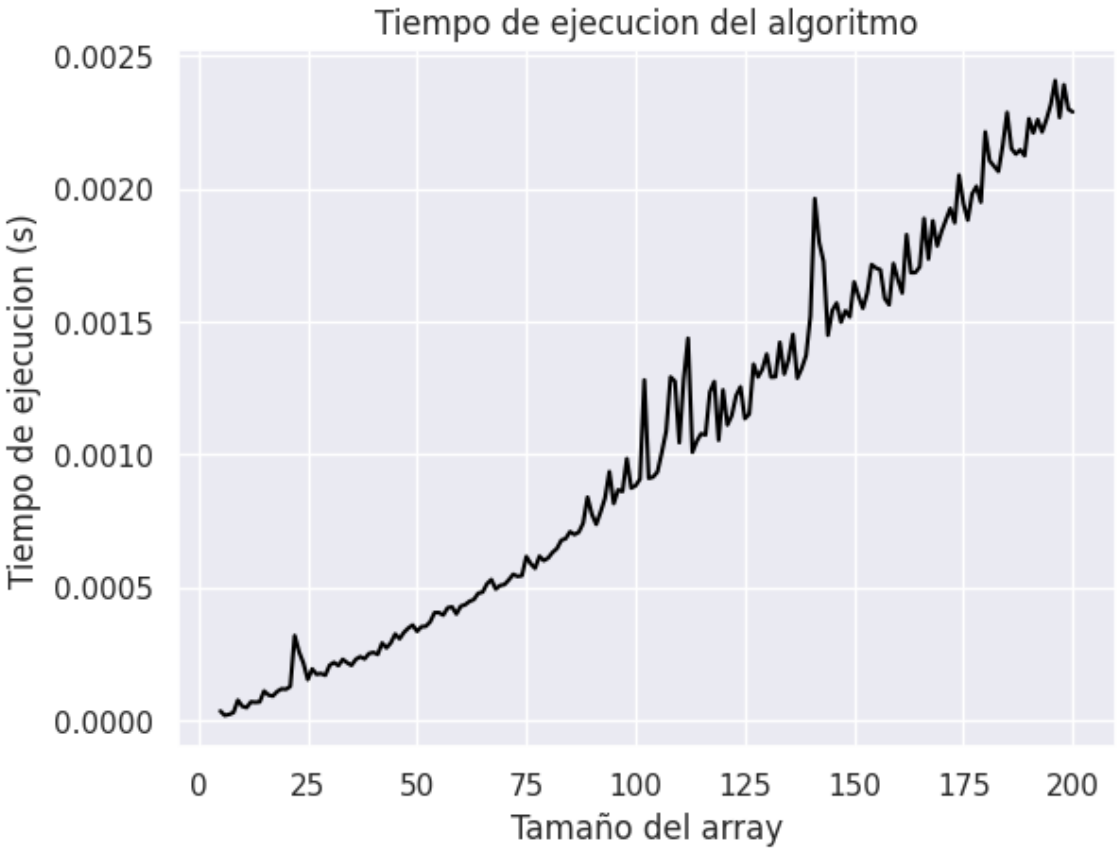


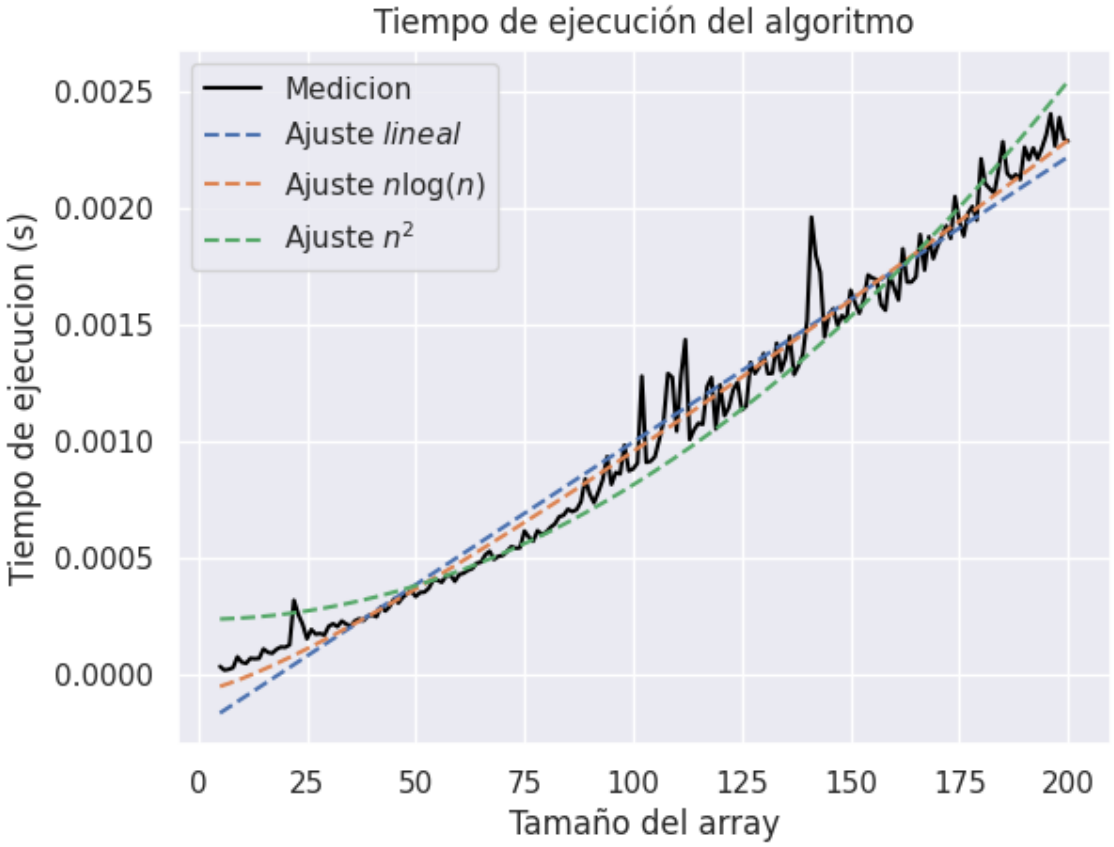


Como se puede apreciar, nuestro algoritmo de backtracking presenta una complejidad exponencial con respecto al tamaño del problema.

6.2. Programación Lineal

6.3. Aproximación





Se aprecia que el algoritmo de aproximación se ajusta mejor a n**2, lo cual coincide con la complejidad planteada anteriormente.

7. Conclusiones

Conclusiones

El análisis realizado permite identificar fortalezas y limitaciones en las diferentes estrategias aplicadas para resolver el problema. Por un lado, los enfoques exactos, como el backtracking y la programación lineal, garantizan la obtención de soluciones óptimas, pero muestran diferencias significativas en términos de rendimiento. El backtracking, aunque eficiente para tamaños pequeños y medianos, enfrenta dificultades de escalabilidad debido a la naturaleza exponencial del espacio de búsqueda. En cambio, la programación lineal, aunque robusta frente a restricciones complejas, presenta tiempos de ejecución elevados en instancias grandes, lo que restringe su practicidad en escenarios de gran escala, debido a la gran cantidad de variables que fueron necesarias de utilizar.

Por otro lado, el algoritmo de aproximación se destaca por su velocidad, lo que lo convierte en una alternativa atractiva para casos donde se prioriza el tiempo de respuesta sobre la precisión de la solución. Sin embargo, su incapacidad para garantizar siempre el óptimo resalta una compensación inherente entre precisión y eficiencia.

En conjunto, los resultados obtenidos enfatizan la importancia de seleccionar el enfoque adecuado en función de los requisitos del problema y las limitaciones computacionales. Los métodos exactos son ideales para instancias donde la precisión es prioritaria, mientras que los métodos aproximados ofrecen soluciones rápidas y razonablemente buenas en contextos de tiempo crítico o recursos limitados.