.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico Integrador

12 de febrero de 2025

Marcos Tomás Weng Xu
109153

Leandro Brizuela
109842

José Rafael Patty Morales
109843

Matías Rea
99770

Lucas Araujo
109867

Índice

1. Reducción

1.1. Regla de decisión

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 (similar a las 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?

1.2. Demostración NP

Para mostrar que el problema está en NP, necesitamos probar que dado un verificador, podemos verificar en tiempo polinómico si una solución es una solución válida o no. El verificador representa una solución candidata al problema.

Para esto debemos asegurarnos de varias cosas:

- Cada casilla ocupada por un barco no debe tener ninguna casilla vecina que contenga otro barco (horizontal, vertical o diagonal) ocupada por otro barco.
- Satisfacción de demanda para todas las filas y columnas.
- No violación de la restricción de tamaño de los barcos y su posición.
- No utilización de un barco más de una vez.

Dado entonces un verificador que tiene los siguientes valores de entrada:

- El tablero de tamaño $n \times m$.
- Las restricciones para las filas y columnas.
- La lista de barcos con sus tamaños b_i.
- La solución a verificar que contiene la ubicación y casillas ocupadas por el barco.

Cada casilla ocupada por un barco no debe tener ninguna casilla vecina (horizontal, vertical o diagonal) que esté ocupada por otro barco. El algoritmo debe recorrer cada casilla ocupada por un barco verificando que esté dentro del tablero y que sus casillas vecinas no estén ocupadas por otro barco. Este algoritmo tiene una complejidad de $O(k \times b_{\max})$, donde $b_{\max}$ es la longitud del barco más largo y k la cantidad de barcos.

También debemos recorrer toda la matriz para ir registrando las demandas cumplidas. Luego, verificamos que cada una de las demandas cumplidas sea menor a la demanda requerida en cada fila y columna. Este algoritmo tiene una complejidad de $O(n \times m)$, ya que recorremos el tablero completo.

Por último, debemos verificar que cada barco i utilizado tiene efectivamente el tamaño b_i. La complejidad de este último es $O(k)$, ya que la verificación de casillas ocupadas se puede realizar en tiempo constante por barco.

En conclusión, hemos demostrado que la verificación de una solución candidata al problema se puede realizar en tiempo polinómico. Primero, se verifica que las casillas ocupadas por los barcos no tengan ninguna casilla vecina ocupada por otro barco, lo que se realiza en una complejidad de $O(k \times b_{\max})$, donde k es la cantidad de barcos y $b_{\max}$ es la longitud del barco más largo. Luego, se recorre el tablero completo para registrar las demandas cumplidas y verificar que estas sean menores a las demandas requeridas para cada fila y columna, con una complejidad de $O(n \times m)$. Finalmente, se asegura que cada barco tiene el tamaño correcto, lo cual se verifica en tiempo $O(k)$, ya que la validación de las casillas ocupadas se realiza en tiempo constante por barco.

En conjunto, la complejidad total del verificador es:

$$O(k \times b_{\max}) + O(n \times m) + O(k) = O(k \times b_{\max} + n \times m)$$

Este tiempo es polinómico en función de los parámetros de entrada, lo que demuestra que la verificación de la solución se realiza en tiempo polinómico y, por lo tanto, el problema pertenece a la clase NP.

A continuación se mostrara el verificador planteado:

```python
def verificar_solucion(tablero, restricciones_filas, restricciones_columnas, barcos):
    n = len(tablero)
    m = len(tablero[0])

    for i in range(n):
        for j in range(m):
            if tablero[i][j] == 1:
                for di in [-1, 0, 1]:
                    for dj in [-1, 0, 1]:
                        if di == 0 and dj == 0:
                            continue
                        ni, nj = i + di, j + dj
                        if 0 <= ni < n and 0 <= nj < m:
                            if tablero[ni][nj] == 1:
                                return False

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

    for barco in barcos:
        b_i, coordenadas = barco
        casillas_ocupadas = 0
        for (x, y) in coordenadas:
            if tablero[x][y] == 1:
                casillas_ocupadas += 1
        if casillas_ocupadas != b_i:
            return False

    return True
```

1.3. Demostración NP-Completo

Para hacer esta demostración, vamos a reducir un problema NP-Completo conocido al problema de la Batalla Naval. En este caso, vamos a usar la reducción desde el problema 3-Partition, que es un problema NP-Completo.

Si logramos reducir 3-Partition a Batalla Naval eso quiere decir que Batalla Naval es por lo menos igual de difícil que 3-Partition, por ende Batalla Naval sería NP-Completo.

para lograr la reducción primero debemos comprender en qué consiste el problema de 3-Partition, especialmente lo veremos en notación unaria.

El problema de 3-Partition en notación unaria es una variante del problema clásico de 3-Partition, donde los números del conjunto S y el valor objetivo B se representan en notación unaria. Esto implica que cada número se codifica como una cadena de unos cuya longitud corresponde al valor del número. El problema dice

Dado:

- Un conjunto $S = \{a_1, a_2, \ldots, a_{3m}\}$, con $3m$ números representados en notación unaria.

- Un entero B, también en notación unaria.

El objetivo es determinar si es posible dividir S en exactamente m subconjuntos $S_1, S_2, \ldots, S_m$ tales que:

- La suma de los elementos en cada subconjunto sea exactamente B.
- Cada subconjunto contenga exactamente tres elementos.

Para resolver 3-Partition Unario con Batalla Naval, la idea es transformar la entrada del problema de 3-Partition unario en una instancia del problema de Batalla Naval de forma que una solución válida para Batalla Naval también sea una solución válida para 3-Partition unario. Para eso necesitamos:

- Un tablero de tamaño $5 \times 3B$, siendo B el valor objetivo.
- Crear un conjunto de $3m$ barcos, donde cada barco representa un número del conjunto S en 3-Partition, siendo $3m$ la cantidad de elementos de S. La longitud de cada barco corresponde al valor del número en S (en unario). Por ejemplo, si un número $a_i = 3$, el barco correspondiente tendrá longitud 3.
- Demandas de filas y columnas: asignamos demandas B a las filas 1, 3 y 5; luego asignamos demanda 0 a todas las demás filas y 1 a las columnas.

Si la instancia de Batalla Naval con las restricciones dadas encuentra una solución válida, esto implica que los elementos de S pueden dividirse en subconjuntos disjuntos que sumen exactamente B, resolviendo así el problema de 3-Partition Unario.

Este enfoque transforma el problema de 3-Partition Unario en una instancia del problema de Batalla Naval, y el cumplimiento de las demandas en las filas y columnas garantiza que la solución obtenida corresponda a una partición válida en 3-Partition. Por lo tanto, podemos decir que finalmente Batalla Naval es un problema NP-Completo.

2. Backtracking

2.1. Objetivo

Utilizamos un algoritmo backtracking que explora todas las posibles distribuciones de los k barcos disponibles a poner en un tablero n x m para cumplir con la mayor demanda posible.

2.2. ¿Qué es un algoritmo de backtracking?

Es una técnica de programación de resolución de problemas que implica encontrar una solución de forma incremental probando diferentes opciones. Esta técnica utiliza condiciones de corte conocidas como podas, las cuales nos permiten descartar aquellas soluciones parciales que no nos llevan a una solución válida.

A continuación se mostrara la función de backtraking:

```
def ubicar_barcos_backtracking(tablero, barcos, demandas_filas, demandas_columnas,
    mejor_solucion, mejor_cumplida, demanda_actual, casillas_disponibles):

    if demanda_actual > mejor_cumplida[0]:
        mejor_cumplida[0] = demanda_actual
        mejor_solucion[:] = [fila[:] for fila in tablero]

    if not barcos:
        return
```

```
9
10      contribucion_barcos = sum(barco * 2 for barco in barcos)
11      contribucion_casillas = calcular_demanda_obtenible(tablero, demandas_filas,
        demandas_columnas)
12
13      contribucion_maxima_restante = min(contribucion_barcos, contribucion_casillas)
14
15      if demanda_actual + contribucion_maxima_restante <= mejor_cumplida[0]:
16          return
17
18      for fila, col in casillas_disponibles:
19          for horizontal in [True, False]:
20              if not horizontal and barcos[0] == 1:
21                  continue
22
23              if tablero[fila][col] != 0:
24                  continue
25
26              if es_posicion_valida(tablero, fila, col, barcos[0], horizontal,
        demandas_columnas, demandas_filas):
27                  barco = barcos.pop(0)
28                  colocar_barco(tablero, fila, col, barco, horizontal)
29
30                  nuevas_casillas = actualizar_casillas_disponibles(
        casillas_disponibles, fila, col, barco, horizontal, tablero)
31                  nueva_demanda = demanda_actual + barco * 2
32
33                  ubicar_barcos_backtracking(tablero, barcos, demandas_filas,
        demandas_columnas, mejor_solucion, mejor_cumplida, nueva_demanda,
        nuevas_casillas)
34
35                  deshacer_barco(tablero, fila, col, barco, horizontal)
36                  barcos.insert(0, barco)
37
38      ubicar_barcos_backtracking(tablero, barcos[1:], demandas_filas,
        demandas_columnas, mejor_solucion, mejor_cumplida, demanda_actual,
        casillas_disponibles)
```

2.3. Podas y optimizaciones

Antes de buscar la solución óptima, ordenamos el arraglo de barcos de mayor a menor tamaño, para así filtrar aquellos barcos que no entran en el tablero. Además, son los mas restrictivos por lo que es conveniente colocarlos primero.

Se verifica si la máxima contribución posible es mejor que el óptimo actual, en caso contrario no tiene sentido seguir explorando esa rama del árbol recursivo. La *máxima contribución posible* se calcula como el mínimo entre la contribución obtenible usando todos los barcos restantes y todas las casillas restantes.

```
1       if demanda_actual + contribucion_maxima_restante <= mejor_cumplida[0]:
2           return
```

Previamente a colocar un barco en el tablero, se verifica que este se encuentre en una posición válida, donde el barco pueda ingresar en su totalidad sin excederse de las demandas y sin ser adyacentes a otros barcos.

```
1     if es_posicion_valida(tablero, fila, col, barcos[0], horizontal,
        demandas_columnas, demandas_filas):
```

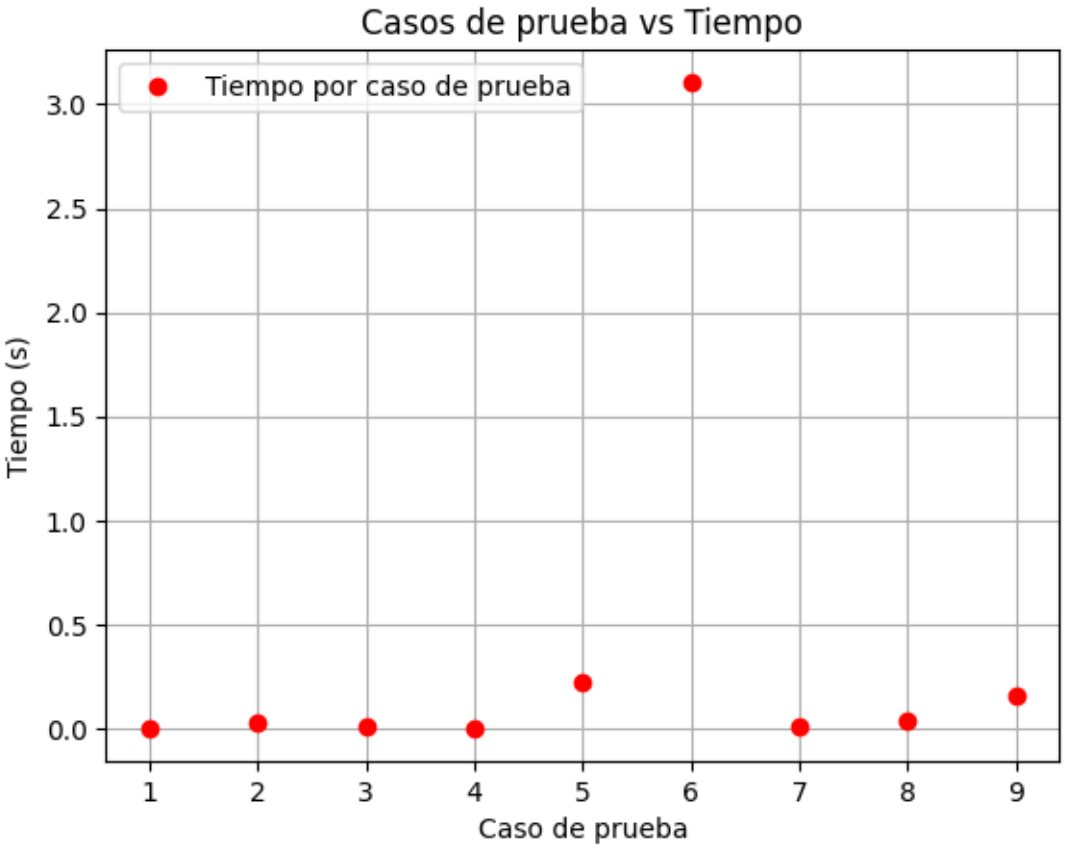
Antes de poner otro barco en el tablero, actualizamos las casillas disponibles, es decir, aquellas casillas las cuales estan ocupadas por un barco o son adyacentes a alguno se descartan, en consecuencia esto nos permite optimizar la cantidad de ramas disponibles a recorrer.

```
1   nuevas_casilla = actualizar_casillas_disponibles(casillas_disponibles, fila, col,
      barco, horizontal, tablero)
```

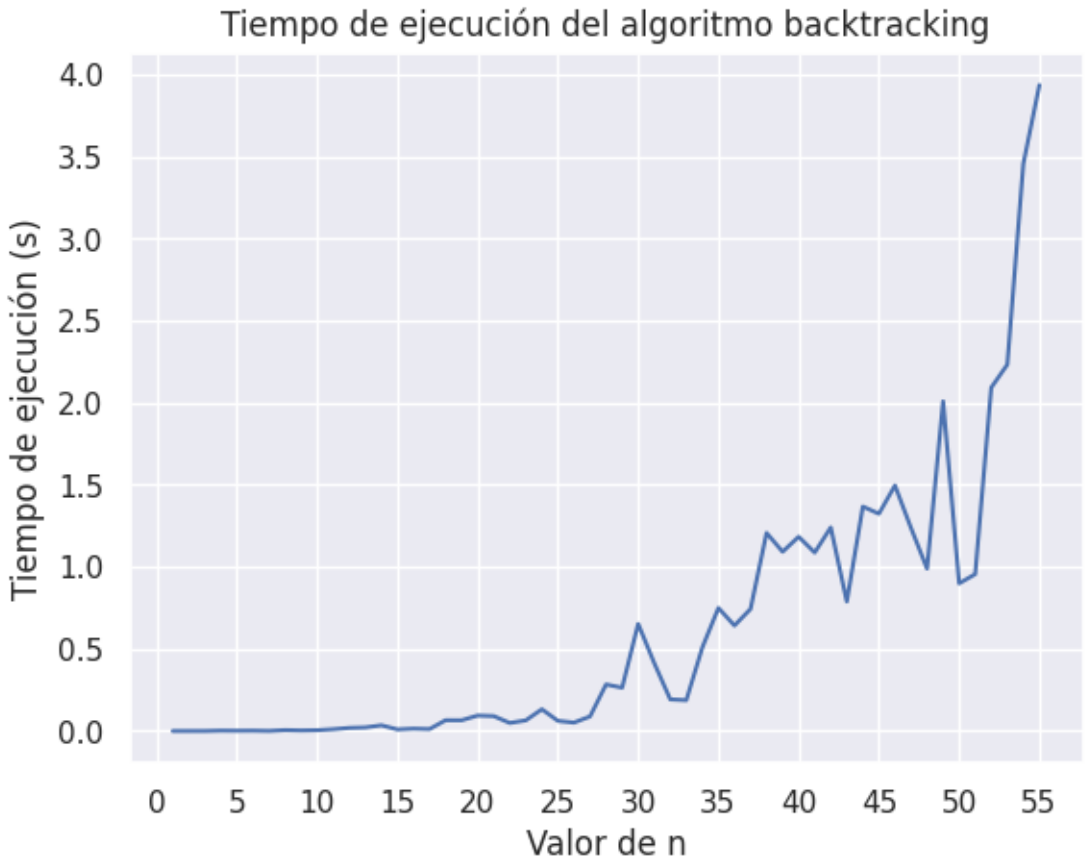
2.4. Mediciones

A continuación, se realizaron mediciones de tiempo con diferentes demandas, tamaño de tablero y cantiad de barcos con distintos tamaños. En en el eje x se muestran los 9 casos de prueba que fueron medidos, los cuales fueron:

Caso de prueba	Tamaño de tablero	Cantidad de barcos	Tiempo (s)
1	3 x 3	2	0.000250
2	5 x 5	6	0.030067
3	8 x 7	10	0.004545
4	10 x 3	3	0.000507
5	10 x 10	10	0.217328
6	12 x 12	21	3.106919
7	15 x 10	15	0.007811
8	20 x 20	20	0.040491
9	20 x 25	30	0.155105



A continuación, se muestra un gráfico con los tiempos de ejecución dependiendo del tamaño del tablero, para n = i , se toma un tablero de tamaño i x i generado aleatoriamente. los barcos utilizados para cada tablero también son generados aleatoriamente. el tiempo de ejecución para estos casos aleatorios se pueden visualizar en el siguiente gráfico:



Con los datos visualizados podemos llegar a la conclusión de que el algoritmo reduce el espacio de búsqueda mediante podas, lo cual ayuda a ralentizar el crecimiento del tiempo de ejecución del problema para tableros de tamaño pequeño y medio.

2.5. Conclusión

Una de las claves al realizar este código, es la busqueda de podas ya que las mismas optimizan el tiempo de ejecucion para encontrar una solución óptima. Por otro lado, al estar corriendo diferentes pruebas, pudimos notar que dar una respuesta a cuánto tardaría en resolver el problema, es algo bastante complejo ya que, el mismo depende de varias variables:

- Cantidad de barcos.
- Largo de cada barco
- Tamaño de la matriz.
- Demanda de filas como de columnas.

Al modificar ciertas variables, el tiempo de ejecución del algoritmo de backtracking puede variar significativamente. Por ello, la optimización de las podas resulta crucial para minimizar el espacio de búsqueda y acelerar la obtención de una solución.

LLega un punto en el que el algoritmo, debido a su complejidad exponencial, a ser mas grande el problema, mas exhautiva va a ser la busqueda. Por ejemplo, cuando probamos con un caso de tablero $30x25$ con 25 barcos, este tardó alrededor de 20 minutos (Se excluyó de las mediciones debido al tiempo excesivamente largo en comparación con el resto).

3. Algoritmo greedy de aproximación para el problema de La Batalla Naval

3.1. Análisis del problema

En esta sección se analiza el algoritmo de aproximación propuesto por el almirante John Jellicoe, diseñado para resolver el problema de distribución en La Batalla Naval. Este algoritmo sigue un enfoque greedy, donde en cada paso selecciona la fila o columna con mayor demanda restante y ubica el barco de mayor longitud que pueda satisfacer dicha demanda en una posición válida. Si el barco no se puede colocar, este es ignorado, y el proceso continúa con el siguiente barco.

Este enfoque se basa en la idea de maximizar la utilidad inmediata en cada iteración, sin considerar las implicaciones futuras, lo que lo convierte en una solución aproximada pero computacionalmente menos costoso. El algoritmo finaliza cuando no quedan más barcos o todas las demandas han sido satisfechas.

El objetivo es determinar cuán optima es esta aproximación en el peor de los casos y validar su aplicabilidad para instancias grandes donde las soluciones óptimas con backtracking son inviables.

Para hacer esto realizaremos experimentos en grandes volúmenes de datos para hallar la cota.

3.2. Implementación del Algoritmo Greedy

Primero empezamos haciendo el código que propone el almirante John Jellicoe

```
def aprox_sea_battle(row_demand, column_demand, ships):
    table = create_table(row_demand, column_demand)

    row_demand_remaining = row_demand.copy()
    column_demand_remaining = column_demand.copy()
    row_demands_aux = row_demand.copy()
    column_demands_aux = column_demand.copy()

    ships_remaining = ships_remaining = sorted(ships, reverse=True)

    #Se utilizan dos demandas aux para poder diferenciar las demandas en las que no
     entr  un barco
    #para sacarla de la proxima busqueda de max demanda

    (bigger_demand, index, is_column) = calculate_bigger_demand(row_demands_aux,
    column_demands_aux)

    while ships_remaining and bigger_demand:
        a_ship_is_placed = False
        for ship in ships_remaining:
            if ship_not_enter_in_demand(bigger_demand, ship):
                continue
            width_start = find_valid_position(table, row_demand_remaining,
    column_demand_remaining, index, is_column, ship)
            if position_not_valid(width_start):
                continue
            put_ship_in_table(table, row_demand_remaining, column_demand_remaining,
     index, width_start, is_column, ship)
            ships_remaining.remove(ship)
            a_ship_is_placed = True
            break
        if a_ship_is_placed:
            column_demands_aux = column_demand_remaining.copy()
            row_demands_aux = row_demand_remaining.copy()
        else:
            deny_demand(row_demands_aux, column_demands_aux, index, is_column)
        (bigger_demand, index, is_column) = calculate_bigger_demand(row_demands_aux
    , column_demands_aux)

    row_unfulfilled_demands = sum(row_demand_remaining)
    column_unfulfilled_demands = sum(column_demand_remaining)
```

```
37      return table, row_unfulfilled_demands, column_unfulfilled_demands
```

3.3. Análisis de Complejidad

Recorremos el código línea por línea para calcular la complejidad, ya sea temporal como espacial.Sea N la cantidad de demandas por fila y M por columna, y P la cantidad de barcos:

Complejidad temporal:

- Línea 2: se crea una matriz de NxM para todos las posiciones posibles. $O(n \cdot m)$.
- Línea 4 a 7: se crean copias de las demandas de filas y columnas, dos para cada una. $O(2 \cdot n) + O(2 \cdot m)$.
- Línea 9: se ordena el arreglo de barcos en orden descendente. $O(p)$.
- Línea 14: se encuentra la máxima demanda disponible entre filas y columnas. $O(n + m)$.
- Línea de 16-33:
  - Línea 16: se realiza un bucle mientras haya demandas y barcos $O(n + m + p)$.
  - Línea de 18-27:
    - Línea 18: se recorren los barcos disponibles.
    - Línea 19: se comprueba si el barco actual entra en la demanda en tiempo constante. $O(k)$.
    - Línea 21-23: se busca una posición valida para el barco. Depende si el ancho es en las filas o columnas. $O(max(n, m))$.
    - Línea 24: se agregar el barco al tablero y se actualizan las demandas. $O(max(n, m))+ O(p)$
    - Línea 25: se elimina el barco de los disponibles. $O(p)$.

    Entonces la complejidad total del bucle de barcos es:

    $$p \cdot (O(max(n, m)) + O(p)) = O(p \cdot max(n, m) + p^2)$$

  - Línea 28-33: se actualizan las demandas. En el peor de los casos se crearan copias. $O(n + m)$

    La complejidad del bucle while es:

    $$O((n + m + p) \cdot (p \cdot max(n, m) + p^2 + (n + m)))$$

- Línea 35: se realiza una suma de las demandas de fila. $O(n)$
- Línea 36: se realiza una suma de las demandas de fila. $O(m)$

Basándonos estrictamente en la notación O, y como los términos de mayor valor vuelven despreciables a los de menor, así como las constantes multiplicativas. Complejidad temporal final =

$$O((n + m + p) \cdot (p \cdot max(n, m) + p^2 + (n + m)) + n \cdot m)$$

3.4. Generación de Tableros Aleatorios

En el siguiente código se implementó un generador aleatorio de tableros que proporciona las restricciones por fila y columna, así como la solución óptima de manera anticipada.

```python
import random
import greedy_aprox as greedy

def agregar_barcos_aletoriamente(barcos, n):
    cantidad_barcos = 3 * n

    for _ in range(cantidad_barcos):
        barcos.append(random.randint(1, n))
    return barcos

def generar_matriz_con_barcos(n, barcos):
    matriz = [[0 for _ in range(n)] for _ in range(n)]
    restricciones_fila = [0 for _ in range(n)]
    restricciones_columna = [0 for _ in range(n)]
    barcos_colocados = []

    def casilla_esta_vacia(fila,columna):
        return matriz[fila][columna] == 0

    def es_posicion_valida(fila, columna):
        # verifica que la fila y columna estee dentro de los limites de la matriz
        return 0 <= fila < n and 0 <= columna < n

    def es_adyacente_vacio(fila, columna):
        for desplazamiento_fila in [-1, 0, 1]:
            for desplazamiento_columna in [-1, 0, 1]:
                if desplazamiento_fila == 0 and desplazamiento_columna == 0:
                    continue
                f, c = fila + desplazamiento_fila, columna + desplazamiento_columna
                if es_posicion_valida(f, c) and not casilla_esta_vacia(f,c):
                    return False
        return True

    def puede_colocar_barco(fila, columna, longitud, horizontal):
        if horizontal:
            if columna + longitud > n:
                return False
            for c in range(columna, columna + longitud):
                if matriz[fila][c] != 0 or not es_adyacente_vacio(fila, c):
                    return False
        else:
            if fila + longitud > n:
                return False
            for f in range(fila, fila + longitud):
                if matriz[f][columna] != 0 or not es_adyacente_vacio(f, columna):
                    return False
        return True

    def colocar_barco(fila, columna, longitud, horizontal):
        if horizontal:
            for c in range(columna, columna + longitud):
                matriz[fila][c] = 1
                restricciones_fila[fila] += 1
                restricciones_columna[c] += 1
        else:
            for f in range(fila, fila + longitud):
                matriz[f][columna] = 1
                restricciones_fila[f] += 1
                restricciones_columna[columna] += 1

    for longitud_barco in barcos:
        colocado = False
        intentos = 0
        #Intentamos 100 veces colocar el barco en una posicion aletoria
        while not colocado and intentos < 1000:
            fila = random.randint(0, n - 1)
            columna = random.randint(0, n - 1)
            horizontal = random.choice([True, False])
            if puede_colocar_barco(fila, columna, longitud_barco, horizontal):
                colocar_barco(fila, columna, longitud_barco, horizontal)
```

```
                barcos_colocados.append(longitud_barco)
                colocado = True
            intentos += 1

    return matriz, restricciones_fila, restricciones_columna, barcos_colocados
```

3.5. Evaluación Empírica de la Cota del Algoritmo Greedy en Tableros de Distintos Tamaños

Una vez que ya tenemos el algoritmo greedy y la generación de tableros aleatorios, ya podemos proceder a calcular la cota.

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

Sea I una instancia cualquiera del problema de La Batalla Naval, $z(I)$ representa la solución óptima para una instancia I, es decir, la cantidad máxima de demandas satisfechas en filas y columnas. Por otro lado, $A(I)$ es la solución aproximada obtenida mediante el algoritmo greedy. Finalmente, $r(A)$ es la cota del rendimiento del algoritmo, que cumple la siguiente relación:

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

Se hizo la generación de varios set de datos, con tableros NxN que van del N=1 al N=300. Estos tableros por como son generados se sabe la solución optima de antemano sin necesidad de utilizar el algoritmo backtracking. Agregamos mas barcos de los que se necesita para calcular la solución optima, para potenciar las fallas que comete el algoritmo greedy con el objetivo de buscar el peor caso.

3.6. Conclusión de la optimalidad del algoritmo greedy

Hemos ejecutado el algoritmo y generado un gráfico en el que el eje horizontal representa el tamaño del tablero (n), mientras que el eje vertical muestra las cotas obtenidas.

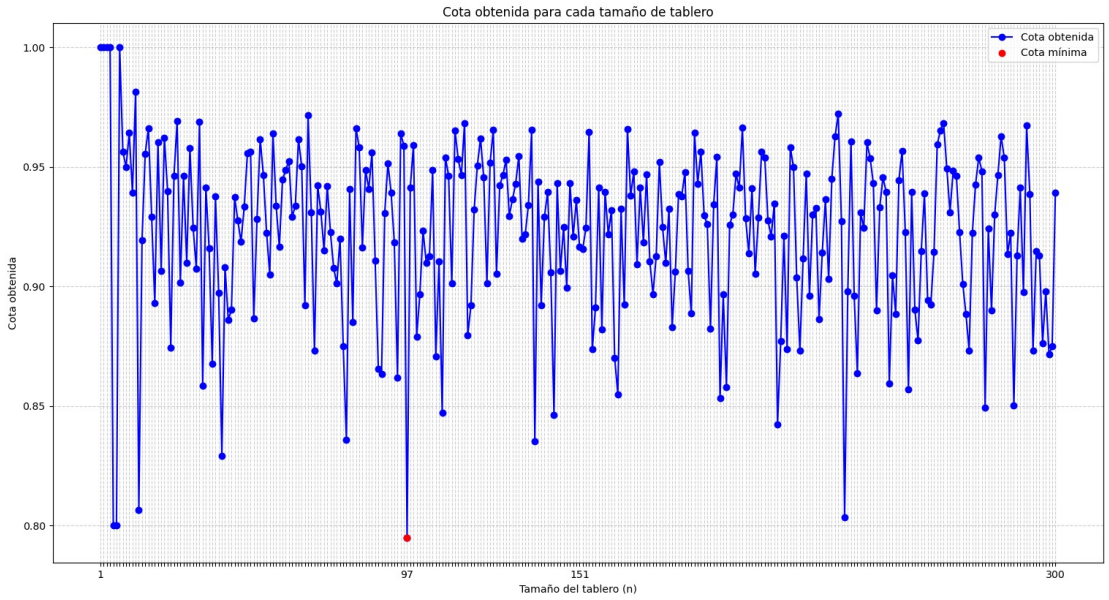


Figura 1: Cotas obtenidas y la cota mínima para distintos tamaños de tablero.

Al observar el gráfico, se pueden identificar picos muy irregulares. Inicialmente, esperábamos que, a medida que el tamaño del tablero aumentara, las cotas fueran progresivamente peores, imaginando un comportamiento similar al de una función logarítmica.

Sin embargo, los resultados obtenidos revelan una dinámica diferente. Consideramos que estas irregularidades en el gráfico podrían deberse a la dependencia de la aleatoriedad tanto en la generación del tablero como en la ejecución del algoritmo. En tableros más pequeños, hemos observado cotas mínimas, probablemente porque el espacio de soluciones es más reducido y las decisiones greedy tienen un impacto proporcionalmente mayor. Además, en tableros pequeños, la falta de opciones para distribuir los barcos amplifica el impacto de las decisiones subóptimas.

Por otro lado, en tableros más grandes, los errores al posicionar un barco de manera no óptima tienden a ser menos significativos, ya que el mayor espacio disponible ofrece más alternativas para compensar estas decisiones y alcanzar un mejor resultado general.

La cota que encontramos fue de 0.79481 entonces nos queda:

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

$$0{,}79481 \leq r(A)$$

Con esto podemos decir que el algoritmo greedy no puede ser peor que el optimo de demandas multiplicado 0.79481. El factor 0.79481 me garantiza que el algoritmo greedy cubrirá al menos el 79 % de esas demandas.

Por ejemplo dado un tablero aleatorio donde mi optimo de demandas por fila y por columna es de 100 entonces la aproximación greedy asegura cubrir por lo menos mas de 79 demandas.

Estos resultados muestra que tan viable es el algoritmo greedy como una solución aproximada para el problema de la Batalla Naval, asegurando un rendimiento aceptable en la mayoría de los casos.

3.7. Anexo: código completo de aproximación

```
VALID = 0
NOT_VALID = 1
SHIP = 2

COLUMN = 1
ROW = 0

def aprox_sea_battle(row_demand, column_demand, ships):
    table = create_table(row_demand, column_demand)
    row_demand_remaining = row_demand.copy()
    column_demand_remaining = column_demand.copy()
    ships_remaining = ships_remaining = sorted(ships, reverse=True)

    row_demands_aux = row_demand.copy()
    column_demands_aux = column_demand.copy()
    (bigger_demand, index, is_column) = calculate_bigger_demand(row_demands_aux,
    column_demands_aux)

    while ships_remaining and bigger_demand:
        a_ship_is_placed = False
        for ship in ships_remaining:
            if ship_not_enter_in_demand(bigger_demand, ship):
                continue
            width_start = find_valid_position(table, row_demand_remaining,
    column_demand_remaining, index, is_column, ship)
            if position_not_valid(width_start):
                continue
            put_ship_in_table(table, row_demand_remaining, column_demand_remaining,
     index, width_start, is_column, ship)
```

```
27              ships_remaining.remove(ship)
28              a_ship_is_placed = True
29              break
30          if a_ship_is_placed:
31              column_demands_aux = column_demand_remaining.copy()
32              row_demands_aux = row_demand_remaining.copy()
33          else:
34              deny_demand(row_demands_aux, column_demands_aux, index, is_column)
35          (bigger_demand, index, is_column) = calculate_bigger_demand(row_demands_aux
    , column_demands_aux)
36
37      row_unfulfilled_demands = sum(row_demand_remaining)
38      column_unfulfilled_demands = sum(column_demand_remaining)
39      return table, row_unfulfilled_demands, column_unfulfilled_demands
40
41
42  def create_table(row_demand, column_demand):
43      table = [ [VALID for i in range(len(column_demand))] for j in range(len(
    row_demand)) ]
44      return table
45
46
47  def calculate_bigger_demand(row_demand_remaining, column_demand_remaining):
48      index_big_row, big_elemt_row = get_bigger_element_info(row_demand_remaining)
49      index_big_column, big_elemt_column = get_bigger_element_info(
    column_demand_remaining)
50      return (big_elemt_row, index_big_row, ROW) if big_elemt_row >= big_elemt_column
         else (big_elemt_column,index_big_column, COLUMN)
51
52  def get_bigger_element_info(array):
53      max = array[0]
54      index = 0
55      for i in range(1, len(array)):
56          if array[i] > max:
57              max = array[i]
58              index = i
59      return index, max
60
61  def ship_not_enter_in_demand(demand, ship):
62      return ship > demand
63
64
65
66  def find_valid_position(table, row_demands, column_demands, lenght_index, is_column
    , ship):
67      width_demands = row_demands if is_column else column_demands
68      posible_contiguous_space = 0
69      index = 0
70      while index<len(width_demands) and posible_contiguous_space < ship:
71          position_status = table[index][lenght_index] if is_column else table[
    lenght_index][index]
72          if width_demands[index] >= 1 and position_status == VALID:
73              posible_contiguous_space += 1
74          else:
75              posible_contiguous_space = 0
76          index += 1
77      return index-posible_contiguous_space if posible_contiguous_space == ship else
    -1
78
79  def position_not_valid(width_start):
80      return width_start == -1
81
82
83
84  def put_ship_in_table(table, row_demands, column_demands, lenght_index, width_index
    , is_column, ship):
85      update_demands(row_demands, column_demands, lenght_index, width_index,
    is_column, ship)
86      update_table(ship, table, lenght_index, width_index, is_column)
87
```

```
88 def update_demands(row_demands, column_demands, lenght_index, width_index,
       is_column, ship):
89     notify_width_demands(row_demands, column_demands, width_index, is_column, ship)
90     notify_lenght_demands(row_demands, column_demands, lenght_index, is_column,
       ship)
91
92 def notify_width_demands(row_demands, column_demands, width_index, in_rows, ship):
93     for i in range(width_index, width_index+ship):
94         if in_rows:
95             row_demands[i] -= 1
96         else:
97             column_demands[i] -= 1
98
99 def notify_lenght_demands(row_demands, column_demands, lenght_index, is_column,
       ship):
100     if is_column:
101         column_demands[lenght_index] -= ship
102     else:
103         row_demands[lenght_index] -= ship
104
105
106 def update_table(ship, table, lenght_index, width_index, is_column):
107     for i in range( max(0, width_index), min(width_index+ship+1, len(table)) ):
108         for j in range( max(0, lenght_index-1), min(lenght_index+2, len(table[0]) )
       ):
109             if is_column:
110                 if (j == lenght_index) and (width_index <= i < width_index+ship):
111                     table[i][j] = SHIP
112                 else:
113                     table[i][j] = NOT_VALID
114             else:
115                 if (j == lenght_index) and (width_index <= i < width_index+ship):
116                     table[j][i] = SHIP
117                 else:
118                     table[j][i] = NOT_VALID
119
120
121 def deny_demand(row_demands, column_demands, index, is_column):
122     if is_column:
123         column_demands[index] = 0
124     else:
125         row_demands[index] = 0
```