Trabajo Práctico 3
Diversión NP-Completa

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Nombre	Padrón
Denise Dall'Acqua	108645
Martín Alejo Polese	106808
Nicolás Agustín Riedel	102130

Índice

Consigna

Para los primeros dos puntos considerar la versión de decisión del problema de La Batalla Naval: 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?

- Demostrar que el Problema de la Batalla Naval se encuentra en NP.
- Demostrar que el Problema de la Batalla Naval es, en efecto, un problema NP-Completo. Si se hace una reducción involucrando un problema no visto en clase, agregar una (al menos resumida) demostración que dicho problema es NP-Completo. Para esto, recomendamos ver ya sea los problemas 3-Partition o Bin-Packing, ambos en su versión unaria. Si bien sería tentador utilizar 2-Partition, esta reducción no sería correcta. En caso de querer saber más al respecto, consultarnos :-)
- Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema (valga la redundancia) en la versión de optimización: 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 las demandas de las n filas y una lista de las m demandas de las columnas, dar la asignación de posiciones de los barcos de tal forma que se reduzca al mínimo la cantidad de demanda incumplida. Pueden no utilizarse todos los barcos. Si simplemente no se cumple que una columna que debería tener 3 casilleros ocupados tiene 1, entonces contará como 2 de demanda incumplida. Por el contrario, no está permitido exceder la cantidad demandada. Generar sets de datos para corroborar su correctitud, así como tomar mediciones de tiempos.
- (opcional) Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Ejecutarlo para los mismos sets de datos para corroborar su correctitud. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.
- John Jellicoe (almirante de la Royal Navy durante la batalla de Jutlandia) nos propone el siguiente algoritmo de aproximación: Ir a la 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.

  Este algoritmo sirve como una aproximación para resolver el problema de La Batalla Naval. Implementar dicho algoritmo, analizar su complejidad y analizar cuán buena aproximación es. Para esto, considerar lo siguiente: Sea I una instancia cualquiera del problema de La Batalla Naval, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles. Calcular $r(A)$ para el algoritmo dado, demostrando que la cota está bien calculada. Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar dicha relación. Realizar también mediciones que contemplen volúmenes de datos ya inmanejables para el algoritmo exacto, a fin de corroborar empíricamente la cota calculada anteriormente.
- (opcional) Implementar alguna otra aproximación (o algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto anterior. Indicar y justificar su complejidad. No es obligatorio hacer este punto para aprobar el trabajo práctico (pero sí resta puntos no hacerlo).
- Agregar cualquier conclusión que parezca relevante.

Resolución

1. Demostración: NP

Para que un problema se encuentre en NP, se debe poder encontrar un validador que valide si la solución es correcta, y lo haga en tiempo polinomial.

Nuestro problema, está dado por:

- Una lista con las demandas para cada fila
- Una lista con las demandas para cada columna
- Una lista de k barcos (donde el barco s tiene b_s de largo)

La solución, está dada por una Matriz de tamaño $n * m$, donde para cada casillero ij:

- Si no hay barco, entonces $Matriz[i][j] = None$
- Si hay un barco, entonces $Matriz[i][j] = s$

Nota: Siendo s el índice de dicho barco

El validador entonces verifica lo siguiente:

- Que los barcos no sean adyacentes
- Que las demandas de las filas se cumplan de manera exacta
- Que las demandas de las columnas se cumplan de manera exacta
- Que se coloquen todos los barcos

1.1. Que los barcos no sean adyacentes

Para verificar que los barcos no sean adyacentes, basta con recorrer cada casillero de la matriz $n*m$ y en cada celda visitar las 8 celdas vecinas (un cuadrado). En cada una de las celdas vecinas que visito, veo si hay otro barco distinto, y si lo hay, la solución no es válida. Visitar las 8 celdas vecinas se hace en tiempo constante $O(1)$. Por lo tanto, verificar que no haya barcos adyacentes tiene un costo de $O(n * m)$

1.2. Que las demandas de las filas y columnas se cumplan de manera exacta

Para verificar que las demandas de las filas se cumplan, basta con recorrer cada casillero de la matriz $n * m$ y en cada casillero que haya un barco, restar 1 a la demanda de la fila y columna a la que pertenece. Finalmente, se recorre la lista con las demandas para cada fila y nos fijamos que sea todo igual a cero $O(n)$. Lo mismo con la lista de las demandas para cada columna $O(m)$. Esto nos da un costo total de $O(n * m)$

1.3. Que se coloquen todos los barcos

Para verificar que se colocaron todos los barcos, hay que recorrer cada casillero de la matriz $n * m$, y en el caso de que en el casillero haya un barco, se agrega a los barcos visitados. Luego se recorren todos los barcos $O(k)$ y nos fijamos que hayan sido todos colocados. Esto nos da un costo total de $O(n * m)$.

1.4. Codigo

```python
from main import *

# Devuelve true si no hay barcos adyacentes al barco dad, false en caso contrario
def validate_adjacency(grid, row, col, ship):
    # Veo si la posicion es adyacente a un barco
    for i in range(-1, 2):
        for j in range(-1, 2):
            if row + i >= 0 and row + i < len(grid) and col + j >= 0 and col + j < len(grid[0]):
                adjacent_location = grid[row + i][col + j]
                if adjacent_location != None and adjacent_location != ship:
                    return False
    return True

# Devuelve true si la solucion es valida, false en caso contrario
# La solucion debe ser una matriz con el problema resuelto
def validator(solution, row_demands, col_demands, ships):

    # Primero validamos adyacencias
    for i in range(len(row_demands)):
        for j in range(len(col_demands)):
            if solution[i][j] != None:
                ship = solution[i][j]
                valid = validate_adjacency(solution, i, j, ship)
                if valid == False:
                    return False

    # Validamos que las demandas se cumplan
    for i in range(len(row_demands)):
        for j in range(len(col_demands)):
            if solution[i][j] != None:
                row_demands[i] -= 1
                col_demands[j] -= 1

    for i in range(len(row_demands)):
        if row_demands[i] != 0:
            return False

    for j in range(len (col_demands)):
        if col_demands[j] != 0:
            return False

    # Validamos que se hayan colocado todos los barcos
    visited_ships = set()
    for i in range(len(row_demands)):
        for j in range(len(col_demands)):
            if solution[i][j] != None:
                visited_ships.add(solution[i][j])

    if len(visited_ships) != len(ships):
        return False

    # Si llego hasta aca, la solucion es valida
    return True
```

1.5. Conclusion

En conclusión, como pudimos validar una solución al problema en tiempo polinomial, podemos afirmar que el problema se encuentra en NP.

2. Demostración: NP-Completo

Reducción desde el problema 3-Partition

Vamos a demostrar que la batalla naval es NP-Completo. Pero, ¿Como se demuestra que un problema es NP-Completo? Tiene que cumplir dos condiciones:

- **Pertenencia a NP:** La verificación de una solución candidata es posible en tiempo polinomial.
- **NP-dificultad:** Se puede realizar una reducción polinomial desde cualquier problema NP-completo hacia este problema.

Ya en la sección anterior pudimos verificar con éxito que nuestro problema es un problema NP, ahora hay que demostrar que se puede realizar una reducción polinomial desde cualquier problema NP-Completo. Recordemos que todos los NP completos pueden ser reducidos despues cualquier problema NP completo.

En nuestro caso, utilizaremos el problema de *3-Partition*, que como fue demostrado anteriormente en clase, es un problema NP-Completo, para verificar que el problema de la batalla naval pertenece a NP-Completo. Es decir, se puede realizar la reduccion polinomial: **3-Partition** $\leq_p$ **PBN**.

Definamos un poco cómo se compone el problema *3-Partition*

Definición del problema 3-Partition

Dado un conjunto de $3m$ números positivos $A = \{a_1, a_2, \ldots, a_{3m}\}$, donde cada número está representado en notación unaria y la suma total de los números es $S = m \cdot B$, queremos particionar A en m subconjuntos disjuntos $S_1, S_2, \ldots, S_m$ tales que:

- Cada subconjunto S_i tiene exactamente 3 elementos.
- La suma de los elementos en cada subconjunto es exactamente B.

Reducción de 2-Partition a 3-Partition

Para usar **3-Partition**, antes debemos demostrar que es NP-Completo. Vamos a usar **2-Partition** para demostrarlo. Este problema como se demostro en clase, es NP-completo.

2-Partition

El problema de **2-Partition** consiste en dividir un conjunto de números en dos subconjuntos disjuntos con la misma suma.

3-Partition

El problema de **3-Partition** consiste en dividir un conjunto de números en tres subconjuntos disjuntos con la misma suma.

Reducción de 2-Partition a 3-Partition

Vamos a agarrar una instancia de **2-Partition** y la convertiremos en una instancia de **3-Partition**:

- **Suma total**: Calculamos la suma de los números en el conjunto S.
- Si la suma es impar, no podemos dividirlo en 2 partes iguales, así que para **3-Partition** también será no.
- Si la suma es par, agregamos un número t que sea la mitad de la suma total.
- Ahora tenemos un nuevo conjunto S', que es el conjunto original más el número t.

2-Partition y 3-Partition

- Si podemos dividir S en dos subconjuntos con la misma suma, entonces podemos dividir S' en tres subconjuntos con la misma suma (dos del conjunto original y uno con el número t).
- Si no podemos hacer la división en 2-Partition, tampoco lo vamos a poder hacer en 3-Partition.

Conclusión

Quedo demostrado entonces, que si podemos resolver 3-Partition, podemos resolver 2-Partition. Como sabemos que 2-Partition es NP-completo, esto demuestra que 3-Partition también es NP-completo.

Reducción de 3-Partition a La Batalla Naval

Dado una instancia del problema *3-Partition*, construiremos una instancia del problema *La Batalla Naval*.

Construcción del tablero:

- **Dimensiones del tablero:** Construimos un tablero con m filas (una fila para cada subconjunto S_i) y $3m$ columnas (una columna para cada elemento de A).
- **Restricciones de las filas:** Las restricciones de las filas (r_i) serán B, es decir, cada fila i debe contener exactamente B casillas ocupadas.
- **Restricciones de las columnas:** Las restricciones de las columnas (c_j) serán los elementos del conjunto $A = \{a_1, a_2, \ldots, a_{3m}\}$. Es decir, la columna j tendrá una restricción igual a a_j.
- **Barcos:** Introducimos $3m$ barcos, uno por cada elemento de A, donde el barco b_j tiene una longitud igual a a_j.

Ejemplo de construcción: Veamos con un ejemplo para poner en contexto lo que estamos diciendo:

Tenemos los siguientes datos del problema 3-Partition:

- $A = \{6, 5, 4, 5, 3, 7\}$,
- $B = 15$, porque por ejemplo si tengo los subconjuntos $S_1 = 6, 5, 4$ y $S_2 = 5, 3, 7$ la suma de los elementos de S_1 es 15 al igual que la suma de los elementos de S_2
- $m = 2$, porque $|A| = 6 = 3 \times 2 = 3 \times m$

Ajustando el Tablero

Para que cada fila pueda contener exactamente $r_i = 15$ casillas ocupadas, necesitamos que el número de columnas del tablero sea suficiente para permitir esa suma. Ajustamos las dimensiones del tablero:

- $n = 2$ filas (porque $m = 2$, el número de subconjuntos).
- $m = 15$ columnas (ya que cada fila debe contener hasta 15 casillas ocupadas).

Por lo tanto, el tablero tiene $2 \times 15 = 30$ casillas, lo cual coincide con la suma total de $\sum A = 30$.

Nueva Configuración

1. **Restricciones de Filas y Columnas:**
   - Cada fila i debe tener exactamente $r_i = B = 15$ casillas ocupadas.
   - Cada columna j debe contener exactamente $c_j = a_j$, con $A = \{6, 5, 4, 5, 3, 7\}$.
2. **Barcos:** Introducimos un barco para cada elemento en A:
   - Barco 1: longitud 6,
   - Barco 2: longitud 5,
   - Barco 3: longitud 4,
   - Barco 4: longitud 5,
   - Barco 5: longitud 3,
   - Barco 6: longitud 7.
3. **Restricciones Adicionales:**
   - Los barcos no pueden ser adyacentes entre sí (ni horizontal, ni vertical, ni diagonalmente).
   - Todos los barcos deben estar dentro del tablero, y la suma total de las celdas ocupadas debe ser $\sum A = 30$.

Solución del Problema de Batalla Naval

Dado el tablero con $n = 2$ filas y $m = 15$ columnas, encontramos la siguiente distribución:

- **Fila 1 (S_1):** Barcos $\{6, 5, 4\}$, que suman $6 + 5 + 4 = 15$.
- **Fila 2 (S_2):** Barcos $\{5, 3, 7\}$, que suman $5 + 3 + 7 = 15$.

Esto satisface las restricciones de las filas ($r_i = 15$) y las columnas ($c_j = a_j$).

Relación entre 3-Partition y Batalla Naval

1. Resolver el problema de *La Batalla Naval* (encontrar la disposición de los barcos en el tablero) equivale a resolver el problema de 3-Partition:
   - Cada fila representa un subconjunto de A.
   - La longitud de los barcos en cada fila corresponde a los elementos del subconjunto.
   - La suma de las longitudes de los barcos en cada fila debe ser igual a $B = 15$.
2. Si no existe una solución para el tablero naval, no existe una partición válida en el problema de 3-Partition.

Equivalencia de las instancias

- Si existe una solución para el problema *3-Partition*, entonces podemos construir una configuración válida del tablero para *La Batalla Naval*.
- Si existe una configuración válida para *La Batalla Naval*, entonces podemos construir una partición válida para el problema *3-Partition*.

Conclusión

Hemos demostrado que:

- *La Batalla Naval* pertenece a NP.
- Reducimos un problema NP-completo (*3-Partition*) a *La Batalla Naval* en tiempo polinomial.

Por lo tanto, *La Batalla Naval* es NP-completo.

3. Algoritmo de Backtracking

3.1. Codigo

```

# Verifica si podemos colocar un barco en un casillero
def verify_position(grid, row, col, rows, cols, demand):

    if row < 0 or row >= len(grid) or col < 0 or col >= len(grid[0]):
        return False

    if rows[row] == 0 or cols[col] == 0:
        return False

    if demand == 0:
        return

    if grid[row][col] != None:
        return False

    for i in range(-1, 2):
        for j in range(-1, 2):
            if row + i >= 0 and row + i < len(grid) and col + j >= 0 and col + j <
    len(grid[0]):
                if grid[row + i][col + j] != None:
                    return False

    return True

# Calcula la demanda cumplida para la grilla
def calculate_score(grid):
    total = 0
    for row in grid:
        for cell in row:
            if cell is not None:
                total += 2
    return total

# Calcula la demanda que se cumple si se colocasen todos los barcos a partir de un
    indice
def calculate_possible_max_ships(ships, current_index):
    score = 0
    available_ships = ships[current_index:]

    for ship in available_ships:
        score += ship
```

```python
    score *= 2

    return score

# Funcion principal. Dada una demanda de filas, columnas y barcos, maximiza la demanda cumplida (minimiza la
# demanda incumplida).
def ship_placement(rows, cols, ships):
    grid = [[None] * len(cols) for _ in range(len(rows))]
    best_solution_grid = [[None] * len(cols) for _ in range(len(rows))]
    total_amount = sum(rows) + sum(cols)
    ships.sort(reverse=True)

    # Llamamos al algoritmo de backtracking
    ship_placement_aux(rows[:], cols[:], ships, grid, best_solution_grid,  0, set())

    # Imprimimos resultados
    print("Gained ammount: ", calculate_score(best_solution_grid))
    print("Total ammount: ", total_amount)
    print_grid(best_solution_grid, rows, cols)

# Coloca el barco horizontalmente
def place_ship_horizontally(grid, row, col, ship_size, rows, cols):
    for k in range(ship_size):
        grid[row][col + k] = 1
        rows[row] -= 1
        cols[col + k] -= 1

# Coloca el barco verticalmente
def place_ship_vertically(grid, row, col, ship_size, rows, cols):
    for k in range(ship_size):
        grid[row + k][col] = 1
        rows[row + k] -= 1
        cols[col] -= 1

# Descoloca el barco horizontalmente
def unplace_ship_horizontally(grid, row, col, ship_size, rows, cols):
    for k in range(ship_size):
        grid[row][col + k] = None
        rows[row] += 1
        cols[col + k] += 1

# Descoloca el barco verticalmente
def unplace_ship_vertically(grid, row, col, ship_size, rows, cols):
    for k in range(ship_size):
        grid[row + k][col] = None
        rows[row + k] += 1
        cols[col] += 1

# Calcula la cantidad de espacios libres en la grilla
def available_places(grid):
    empty_places = 0

    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == None:
                empty_places += 1

    return empty_places
# Funcion de BT para maximizar la demanda cumplida
def ship_placement_aux(rows, cols, ships, grid, best_solution_grid, current_idx_ship, memo):

    # Vemos si ya nos encontramos con este mismo escenario, y de ser asi podamos
    state = (tuple(rows), tuple(cols), current_idx_ship)
    if state in memo:
        return
    memo.add(state)

```

```python
    # Calculamos las puntuaciones (demandas cumplidas)
    score_grid = calculate_score(grid)
    score_best = calculate_score(best_solution_grid)

    # Si encontramos una mejor solucion, la pisamos
    if score_grid > score_best:
        for i in range(len(rows)):
            for j in range(len(cols)):
                best_solution_grid[i][j] = grid[i][j]

    # Caso base
    if (current_idx_ship >= len(ships)):
        return

    # Poda 1 (la puntuacion que podria llegar a conseguir en el mejor de los casos no supera la mejor conseguida)
    if score_grid <= score_best:
        available_spaces = available_places(grid)
        max_possible_score_ships = calculate_possible_max_ships(ships, current_idx_ship)
        max_possible_score = score_grid + min(max_possible_score_ships, available_spaces * 2)
        if max_possible_score <= score_best:
            return

    # Poda 2 (el barco no entra para ninguna fila/columna por falta de demanda)
    maximo = max(max(rows), max(cols))
    while (ships[current_idx_ship] > maximo) and (current_idx_ship < len(ships)):
        current_idx_ship += 1

    # Con el barco actual, iteramos por la grilla intentando colocarlo
    ship_size = ships[current_idx_ship]
    for i in range(len(rows)):
        if rows[i] == 0:
            continue
        for j in range(len(cols)):
            if cols[j] == 0:
                continue
            can_place_horizontal = True
            can_place_vertical = True
            demand = 0

            # Vemos si se puede colocar horizontal
            if ship_size > rows[i]:
                can_place_horizontal = False
            else:
                for k in range(ship_size):
                    if (k == 0):
                        demand = rows[i]
                    if (not verify_position(grid, i, j + k, rows, cols, demand)):
                        can_place_horizontal = False
                        break
                    demand -= 1

            # Vemos si se puede colocar vertical
            if ship_size > cols[j]:
                can_place_vertical = False
            else:
                for k in range(ship_size):
                    if (k == 0):
                        demand = cols[j]
                    if (not verify_position(grid, i + k, j, rows, cols, demand)):
                        can_place_vertical = False
                        break
                    demand -= 1

            # Colocamos los barcos segun corresponda

            if (can_place_horizontal):
                place_ship_horizontally(grid, i, j, ship_size, rows, cols)
```

```
                ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
    current_idx_ship + 1, memo)
                unplace_ship_horizontally(grid, i, j, ship_size, rows, cols)

            if (can_place_vertical):
                place_ship_vertically(grid, i, j, ship_size, rows, cols)
                ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
    current_idx_ship + 1, memo)
                unplace_ship_vertically(grid, i, j, ship_size, rows, cols)

    # Caso en el que decidimos no colocar el barco
    ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
    current_idx_ship + 1, memo)
```

3.2. Análisis

Estamos probando todas las combinaciones posibles, al iterar por todos los barcos y por cada uno, iterar por cada casillero de la matriz. En cada posición del casillero, estamos intentando meter el barco, tanto vertical como horizontalmente, y también consideramos el caso de que no se ponga dicho barco. Esto nos termina dando una complejidad exponencial en cantidad de barcos, ya que como mencionamos al principio, estamos probando todas las combinaciones posibles, por lo tanto: $O(2^n)$.

Estamos realizando dos podas:

- Si el barco por el que estoy iterando no cabe por la capacidad máxima de la fila o la columna (que tenga mayor valor entre ambos máximos), entonces el barco no se puede meter en la solución actual, por lo que se pasa al siguiente barco.
- Sumamos los barcos que nos quedan por colocar (multiplicado por 2), y eso es como mucho lo máximo que puede mejorar nuestra solución. Si no supera la mejor solución ya obtenida, no tiene sentido seguir, por lo que se poda la rama.

4. Ejemplos de ejecución

A continuación, se ilustrarán los resultados de los ejemplos provistos por la cátedra.

```
Demanda cumplida: 4
Demanda total: 11
Demanda de filas: [3, 1, 2]
Demanda de columnas: [3, 2, 0]
----------------------
 1  -  -
 -  -  -
 1  -  -
----------------------


Demanda cumplida: 12
Demanda total: 18
Demanda de filas: [3, 3, 0, 1, 1]
Demmand de columnas: [3, 1, 0, 3, 3]
----------------------
 1  1  -  1  -
 -  -  -  1  -
 -  -  -  -  -
 1  -  -  -  -
 1  -  -  -  -
----------------------


Demanda cumplida: 26
Demanda total: 53
Demanda de filas: [1, 4, 4, 4, 3, 3, 4, 4]
Demanda de columnas: [6, 5, 3, 0, 6, 3, 3]
```

```
----------------------
 1  -  -  -  -  -  -
 1  -  1  -  1  1  -
 1  -  1  -  -  -  -
 -  -  -  -  1  1  -
 1  1  -  -  -  -  -
 -  -  -  -  1  -  1
 -  -  -  -  -  -  -
 -  -  -  -  -  -  -
----------------------


Demanda cumplida:  6
Demanda total:  14
Demanda de filas:  [1, 0, 1, 0, 1, 0, 0, 1, 1, 1]
Demanda de columnas:  [1, 4, 3]
----------------------
 -  -  -
 -  -  -
 -  -  -
 -  -  -
 -  -  -
 -  -  -
 -  -  -
 -  1  -
 -  1  -
 -  1  -
----------------------


Demanda cumplida:  40
Demanda total:  40
Demanda de filas:  [3, 2, 2, 4, 2, 1, 1, 2, 3, 0]
Demanda de columnas:  [1, 2, 1, 3, 2, 2, 3, 1, 5, 0]
----------------------
 -  -  -  1  -  1  -  -  1  -
 -  -  -  1  -  -  -  -  1  -
 -  -  -  1  -  -  -  -  1  -
 1  1  -  -  -  -  1  -  1  -
 -  -  -  -  1  -  1  -  -  -
 -  -  -  -  -  -  1  -  -  -
 -  -  -  -  -  -  -  -  1  -
 -  1  1  -  -  -  -  -  -  -
 -  -  -  -  1  1  -  1  -  -
 -  -  -  -  -  -  -  -  -  -
----------------------


Demanda cumplida:  46
Demanda total:  58
Demanda de filas:  [3, 6, 1, 2, 3, 6, 5, 2, 0, 3, 0, 3]
Demanda de columnas:  [3, 0, 1, 1, 3, 1, 0, 3, 3, 4, 1, 4]
----------------------
 -  -  -  -  -  -  -  1  -  1  -  -
 -  -  1  1  1  1  -  1  -  1  -  -
 -  -  -  -  -  -  -  -  -  1  -  -
 -  -  -  -  -  -  -  -  -  1  -  1
 1  -  -  -  1  -  -  -  -  -  -  1
 1  -  -  -  1  -  -  -  1  -  -  1
 1  -  -  -  -  -  -  -  1  -  -  1
 -  -  -  -  -  -  -  -  1  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  1  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -
----------------------


Demanda cumplida:  40
Demanda total:  67
```

```
Demanda de filas:  [0, 3, 4, 1, 1, 4, 5, 0, 4, 5, 4, 2, 4, 3, 2]
Demanda de columnas:  [0, 0, 3, 4, 1, 4, 6, 5, 2, 0]
----------------------
 - - - - - - - - - -
 - - 1 1 - - 1 - - -
 - - - - - - 1 - - -
 - - - - - - 1 - - -
 - - - - - - 1 - - -
 - - 1 1 - - 1 - 1 -
 - - - - - - 1 - 1 -
 - - - - - - - - - -
 - - 1 - 1 - - 1 - -
 - - - - - - - 1 - -
 - - - 1 - - - 1 - -
 - - - - - - - 1 - -
 - - - - - - - 1 - -
 - - - - - - - - - -
 - - - - - - - - - -
----------------------


Demanda cumplida:  104
Demanda total:  120
Demanda de filas:  [5, 0, 0, 6, 2, 1, 6, 3, 3, 1, 2, 4, 5, 5, 2, 5, 4, 0, 4, 5]
Demanda de columnas  [0, 5, 5, 0, 6, 2, 2, 6, 2, 1, 3, 1, 2, 3, 1, 4, 5, 2, 1, 6]
----------------------
 - - - - 1 1 1 1 1 - - - - - - - - - - -
 - - - - - - - - - - - - - - - - - - - -
 - - - - - - - - - - - - - - - - - - - -
 - - - - 1 1 1 1 1 1 - - - - - - - - - -
 - - 1 - - - - - - - - - - - - - - - - 1
 - - - - - - - - - - - - - - - - - - - 1
 - - - - - - - - - - - 1 1 1 1 1 - - - 1
 - - 1 - - - - 1 - - - - - - - - - - - 1
 - - 1 - - - - - - - - - 1 - - - - - - 1
 - - - - - - - - - - - - - - - - - - - 1
 - 1 - - 1 - - - - - - - - - - - - - - -
 - 1 - - 1 - - 1 - - 1 - - - - - - - - -
 - 1 - - 1 - - 1 - - 1 - - - - - 1 - - -
 - 1 - - 1 - - 1 - - 1 - - - - - 1 - - -
 - 1 - - - - - - - - - - - - - - 1 - - -
 - - - - - - - - - - - - - 1 - - 1 - 1 -
 - - 1 - - - - - - - - - - - - - 1 - - -
 - - - - - - - - - - - - - - - - - - - -
 - - 1 - - - - - - - - - - 1 - - - - - -
 - - - - - - - - - - - - - - - - - - - -
----------------------


Demanda cumplida:  172
Demanda total:  247
Demanda de filas:  [1, 2, 5, 10, 11, 0, 11, 11, 3, 9, 9, 3, 9, 6, 1, 8, 3, 11, 6, 7]
Demanda de columnas:  [5, 4, 5, 2, 10, 1, 0, 8, 7, 6, 0, 5, 4, 8, 4, 7, 4, 0, 8, 5, 6, 2, 4, 9, 7]
----------------------
 1 - - - - - - - - - - - - - - - - - - - - - - - -
 1 - 1 - - - - - - - - - - - - - - - - - - - - - -
 1 - 1 - - - - - 1 - - - - - - 1 1 - - - - - - - -
 1 - 1 - - - - - 1 - - - - - - - - - 1 1 1 1 1 1 1
 1 - 1 - - 1 - - 1 - - 1 1 1 1 1 1 - - - - - - - -
 - - - - - - - - - - - - - - - - - - - - - - - - -
 - - 1 - 1 - - 1 - 1 - 1 1 1 1 1 - - - - - - - - -
 - - - - 1 - - 1 - 1 - - - - - - - - 1 1 1 1 1 - -
 - - - - 1 - - 1 - 1 - - - - - - - - - - - - - - -
 - - - - 1 - - 1 - 1 - 1 1 1 1 1 - - - - - - - - -
 - - - - 1 - - 1 - 1 - - - - - - - - 1 1 1 - 1 1 1
 - - - - 1 - - 1 - 1 - - - - - - - - - - - - - - -
 - - - - 1 - - 1 - - - 1 1 1 1 - - - - - - - - - -
 - - - - 1 - - 1 - - - - - - - - - - - - - - - - -
```

```
 -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
---------------------


Demanda cumplida:  202
Demanda total:  360
Demanda de filas:  [3, 11, 11, 1, 2, 5, 4, 10, 5, 2, 12, 6, 12, 7, 0, 2, 0, 8, 10,
    11, 6, 10, 0, 11, 5, 8, 6, 9, 8, 0]
Demanda de columnas:  [3, 12, 1, 5, 14, 15, 6, 11, 2, 10, 12, 10, 6, 2, 7, 1, 5,
    11, 5, 10, 7, 11, 4, 0, 5]
---------------------
 1  -  -  -  1  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  1  1  1  1  1  1  1  -  -
 -  -  -  -  1  -  -  -  -  1  1  1  1  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  1  -  -  -  -  -  -  -  -  1  1  1  1  1  1  -  -  -
 -  1  -  -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  1  1  1  1  1  1  -  -  -
 -  1  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  1  -  -  -  -  -  -  -  1  1  1  1  1  1  1  1  1  1  1  -  -  -  -  -
 -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  1  1  1  1  1  1  -  -  -
 -  -  -  1  1  1  1  1  1  1  1  1  1  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  1  1  1  1  1  1  1  1  1  1  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 1  1  1  1  1  1  1  1  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
---------------------
```

5. Algoritmo de aproximación

5.1. Descripción del algoritmo

A continuación se presenta un algoritmo de aproximación para el problema de la batalla naval.

```python
def find_index_of_max(list):

    i = 0
    max = list[0]

    for row, idx in list:
        if row > max:
            max = row
            i = idx
        i += 1

    return i

```

```python
def verify_position(grid, row, col, rows, cols):

    if row < 0 or row >= len(grid) or col < 0 or col >= len(grid[0]):
        return False

    if rows[row] == 0 or cols[col] == 0:
        return False

    # Vemos si la posicion esta ocupada
    if grid[row][col] != None:
        return False
    # Vemos si la posicion es adyacente a una nave
    for i in range(-1, 2):
        for j in range(-1, 2):
            if row + i >= 0 and row + i < len(grid) and col + j >= 0 and col + j < len(grid[0]):
                if grid[row + i][col + j] != None:
                    return False
    return True

def try_to_put_ship_horizontally_in_row(ship, grid, idx_f, rows, cols):

    row_to_put_ship = grid[idx_f]
    for idx_c in range(len(row_to_put_ship)):

        can_place = True
        for k in range(ship):
            if not verify_position(grid, idx_f, idx_c + k, rows, cols): # Veo si puedo poner el barco en esa celda
                can_place = False
                break

        if can_place:
            for k in range(ship):
                grid[idx_f][idx_c + k] = ship
                rows[idx_f] -= 1
                cols[idx_c + k] -= 1

def aproximation(rows, cols, ships):
    grid = [[None] * len(cols) for _ in range(len(rows))]
    ships.sort(reverse=True)

    for ship in ships:
        indice_fila_max = rows.index(max(rows))
        indice_columna_max = cols.index(max(cols))

        if rows[indice_fila_max] >= cols[indice_columna_max]:

            for j in range(len(cols)):
                if verify_position(grid, indice_fila_max, j, rows, cols):

                    if (cols[j] >= ship) and (indice_fila_max + ship <= len(rows)):
                        can_place = True
                        for k in range(ship):
                            if not verify_position(grid, indice_fila_max, j + k, rows, cols):
                                can_place = False
                                break
                        if can_place:
                            for k in range(ship):
                                grid[indice_fila_max][j + k] = 1 # Se coloca el barco
                                rows[indice_fila_max] -= 1
                                cols[j + k] -= 1
                            break
        else:
          for i in range(len(rows)):
                if verify_position(grid, i, indice_columna_max, rows, cols):

```

```
                    if (rows[i] >= ship) and (indice_columna_max + ship <= len(cols)):
                        can_place = True
                        for k in range(ship):
                            if not verify_position(grid, i + k, indice_columna_max, rows, cols):
                                can_place = False
                                break
                        if can_place:
                            for k in range(ship):
                                grid[i + k][indice_columna_max] =  1 # Se coloca el barco
                                rows[i + k] -= 1
                                cols[indice_columna_max] -= 1
                            break

    return grid
```

5.2. Análisis del algoritmo

El algoritmo de aproximación para el problema de la batalla naval tiene una complejidad de $O((n+m) \cdot b \cdot k)$, donde n es el número de filas, m es el número de columnas del tablero, b es la longitud del barco y k es la cantidad de barcos que hay. Ya que el algoritmo recorre los barcos, compara cual tiene más demanda, si la columna o la fila, que luego define si poner el barco horizontal o vertical.

Cuadro 1: Resultados de demandas

Demanda Total	Demanda Aproximada - A(I)	Demanda Óptima - Z(I)	Relación $r(A)$
11	2	4	0.5000
18	12	12	1.0000
53	18	26	0.6923
40	20	40	0.5000
58	18	46	0.3913
67	18	40	0.4500
120	38	104	0.3653
247	62	172	0.3604
360	88	202	0.4356

Como podemos ver, no es el mejor algoritmo para acercarse a la solución óptima, que sería maximizar la demanda cumplida. Sin embargo, en algunos casos llega como por ejemplo en el segundo ejemplo ilustrado anteriormente, o se acerca bastante como el tercer ejemplo.

La cota a la que llegamos empiricamente, para ver la relación entre el resultado óptimo, es $r(A) = 0{,}3604$.

Por lo tanto, en el peor de los casos, obtuvimos una aproximación de $3/10$ a la solución óptima.

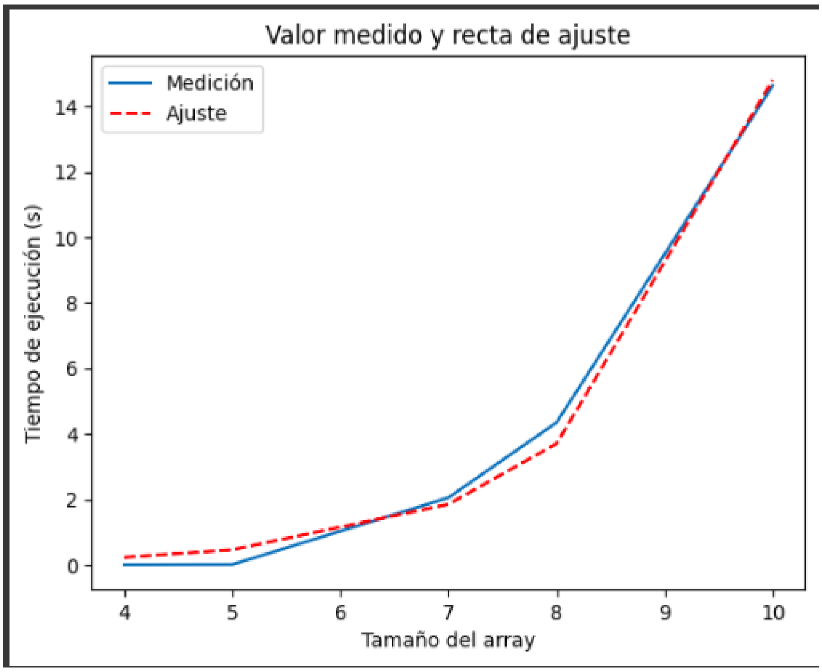
6. Medición empírica

Para comprobar empíricamente la complejidad $\mathbf{O}(2^n)$ del algoritmo, se decidió ejecutar el mismo con distintos tamaños de entrada y medir el tiempo de ejecución. Se generaron muestras de tamaño n, las cuales varían desde 10 hasta 1000.

Para cada muestra se registró el tiempo de ejecución, obteniendo el siguiente gráfico:

A simple vista se puede observar un crecimiento *exponencial*. Para confirmar esto, vamos a ajustar los datos a una recta mediante cuadrados mínimos. Esto lo realizamos con *Python* y la función *optimize.curve_fit* de la biblioteca *scipy*.

Obtenemos que el gráfico se puede ajustar a la curva $y = 1{,}45e^{-2} \times 2^x$, con un error cuadrático medio de $1{,}485e^{-1}$. Por lo tanto, podemos verificar lo que ya vimos en la sección 3, que el orden es exponencial $\mathbf{O}(2^n)$.



7. Conclusiones

En este trabajo práctico, vimos dos versiones del problema de la batalla naval. Por un lado, tenemos un problema NP-Completo, donde intentamos colocar los barcos de forma tal que se cumplan las demandas de todas las filas y columnas, siempre respetando las restricciones de adyacencias. Como demostramos en las secciones anteriores, el problema es de tipo NP y lo pudimos demostrar realizando una reducción polinomial de otro problema NP-Completo como es 3-Partition.

Luego vimos otra variante del problema, en la cual minimizamos la demanda incumplida utilizando un algoritmo de Backtracking.

Por último, implementamos el algoritmo que nos propone John Jellicoe, el cual no nos lleva a la solución óptima, pero nos permite aproximarnos con una cota inferior de $0,3604$, lo cual es aproximádamente un 33 % de la solución óptima.