.UBAfiuba
FACULTAD DE INGENIERÍA

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

1. Demostración: NP **4**
1.1. Que los barcos no sean adyacentes . . . 4
1.2. Que las demandas de las filas y columnas se cumplan de manera exacta . . . 4
1.3. Que se coloquen todos los barcos . . . 4
1.4. Codigo . . . 5
1.5. Conclusion . . . 5

2. Demostración: NP-Completo **6**
2.1. Veamos otro ejemplo: . . . 9
2.2. Hay 3-Partition → hay PBN: . . . 9
2.3. Hay PBN → hay 3-Partition: . . . 10

3. Algoritmo de Backtracking **10**
3.1. Codigo . . . 10
3.2. Análisis . . . 13

4. Ejemplos de ejecución **14**

5. Algoritmo de aproximación **17**
5.1. Descripción del algoritmo . . . 17
5.2. Análisis del algoritmo . . . 18

6. Medición empírica **19**

7. Conclusiones **20**

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 en su version unaria

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 demostraremos más abajo, 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*.

Nota: Por simplificacion, siempre que hablemos de 3-Partition, nos estamos refiriendo a la version unaria del problema.

Definición del problema 3-Partition

Dado un conjunto de enteros positivos $S = \{a_1, a_2, \ldots, a_n\}$, donde cada número está representado en notación unaria, decide si el conjunto puede partirse en 3 subconjuntos disjuntos S_1, S_2, S_3, tal que la suma de cada subconjunto sea la misma. Es decir:

- La suma de los elementos de cada subconjunto S_i = m
- sum(S) = $sum(S_1) + sum(S_2) + sum(S_3)$ = 3m

Veamos un ejemplo:

$S = \{1, 111, 11, 1, 1, 1\}$

Nota: Es la representacion unaria de $S = \{1, 3, 2, 1, 1, 1\}$

Una solución a esta instancia del problema de 3-Partition es la siguiente:

$S_1 = \{111\}$, $S_2 = \{11, 1\}$, $S_3 = \{1, 1, 1\}$

Vemos que es solución, ya que se cumple que la suma de los elementos en cada subconjunto $S_i = 3$.

Reducción de 2-Partition a 3-Partition

Definiciones

2-Partition: El problema consiste en dividir un conjunto de números $A = \{a_1, a_2, \ldots, a_n\}$ en dos subconjuntos disjuntos S_1 y S_2 tales que la suma de los elementos de ambos subconjuntos sea igual.

3-Partition: El problema consiste en dividir un conjunto de números (en representacion unaria) $B = \{b_1, b_2, \ldots, b_n\}$ en tres subconjuntos disjuntos S_1, S_2, S_3, tales que la suma de los elementos en cada subconjunto sea la misma.

Pertenencia a NP

Primero hay que demostrar que 3-Partition pertenece a NP, para ello, debemos poder encontrar un validador que valide una solución al problema de 3-Partition en tiempo polinomial. Dicho validador es el siguiente:

Dado $S = \{a_1, a_2, \ldots, a_n\}$, y tres subconjuntos S_1, S_2, y S_3, se valida que:

- S, S_1, S_2, S_3 estan en representacion unaria (lo estan si su representacion consiste en a_i veces el numero 1)
- $sum(S_1) = sum(S_2) = sum(S_3)$
- $S_1 \cup S_2 \cup S_3 = S$
- $S_1 \cap S_2 \cap S_3 = \emptyset$

Verificar la representación unaria, comprobar que los subconjuntos son disjuntos, calcular la suma de los elementos de cada subconjunto y verificar que las sumas son iguales, se pueden realizar en tiempo polinomial, puesto que para realizar todas estas operaciones, habría que recorrer elemento por elemento cada subconjunto.

Reducción

Para reducir polinomialmente una instancia de **2-Partition** a una instancia de **3-Partition**, hacemos lo siguiente:

Consideramos el conjunto $S = \{a_1, a_2, \ldots, a_n\}$ tal que $sum(S) = 2m$. Donde $m = sum(S_1) = sum(S_2)$.

Lo que debemos hacer, es calcular m, y agregarlo al conjunto S. Luego, realizamos la representacion unaria de los elementos de S, y resolvemos el problema usando la caja negra de 3-Partition. Si hay solución para 3-Partion, hay solución para 2-Partition.

Veamoslo con un ejemplo:

$S = \{3, 4, 3, 4\}$. Una solución al problema de 2-Partition es la siguiente: $S_1 = \{3, 4\}$, $S_2 = \{3, 4\}$. Pero vamos a resolverlo usando 3-Partition de la forma que mencionamos arriba, es decir, vamos a transformar esta instancia a una que pueda resolverse con 3-Partition.

Vemos que $m = sum(S)\ /\ 2 = (3 + 4 + 3 + 4)\ /\ 2 = 7$, entonces al agregar a m al conjunto S y representar todo de forma unaria, tenemos $S = \{111, 1111, 111, 1111, 1111111\}$. Y usando la caja negra de 3-Partition, vemos que hay solución. De hecho, es la siguiente:

$S_1 = \{111, 1111\}$, $S_2 = \{111, 1111\}$, $S_3 = \{1111111\}$.

Nota: Siempre vamos a encontrar que un tercer subconjunto va a ser exactamente m, y por lo tanto, sumar exactamente m tambien (que es lo que por supuesto, suman los otros 2 subconjuntos).

Si pudimos encontrar estos 3 subconjuntos, implica que hay solución para 2-Partition, ya que ahora en 3-Partition $sum(S) = 3m = 2m + m$, y esto solo sucede si $sum(S_1) = sum(S_2) = sum(S_3) = m$, que significa que en la instancia original de 2-Partition existe solución tal que $sum(S) = 2m$, siendo $sum(S_1) = m$, $sum(S_2) = m$.

Dado que encontrar m, agregar m al conjunto S y representar a los elementos de S en forma unaria, pueden hacerse en tiempo polinomial. Entonces pudimos hacer la reduccion **2-Partition** $\leq_p$ **3-Partition**. Quedando demostrado que 3-Partition pertenece a 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:**
  - Cantidad de filas: 5. Cada subconjunto S_i ocupa una fila, y se agrega una fila vacía después de cada subconjunto (excepto después del último).
  - Cantidad de columnas: $sum(S) = 3m$.
- **Restricciones de las filas:**
  - Las filas que contienen subconjuntos tienen una restricción igual a m.
  - Las filas vacías (entre subconjuntos) tienen una restricción igual a 0.
- **Restricciones de las columnas:** Las restricciones de las columnas van a ser 1 para todas. Ya que en cada columna va a haber exactamente una parte de un barco.
- **Barcos:** Los barcos van a ser los elementos a_i que pertenecen al conjunto S.

Ejemplos

A continuación desarrollaremos 2 ejemplos para visualizar la relación entre 3-Partition y la Batalla Naval:

Instancia del problema de 3-Partition:

$$S = \{1, 111, 11, 1, 1, 1\}$$

Pasamos a no unario:

$$S = \{1, 3, 2, 1, 1, 1\}$$

Calculamos la suma total y hallamos m:

$$\text{sum}(S) = 9 = 3m \rightarrow m = 3$$

Lo transformamos a un problema de batalla naval:

$$\text{Filas} = [3, 0, 3, 0, 3]$$
$$\text{Columnas} = [1, 1, 1, 1, 1, 1, 1, 1, 1]$$
$$\text{Barcos} = [1, 3, 2, 1, 1, 1]$$

Usamos la caja negra de la batalla naval. Veamos una posible solución:

1	1	1	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–
–	–	–	1	–	–	1	–	1
–	–	–	–	–	–	–	–	–
–	–	–	–	1	1	–	1	–

Vemos que se cumplen las restricciones de las filas, ya que la primer, tercera y quinta fila tienen 3 de demanda cumplida en cada una. También se cumplen las restricciones de las columnas, pues en cada una de ellas se coloca exactamente una parte de un barco. Además vemos que se están colocando todos los barcos (los 6 barcos), respetando las restricciones de adyacencia. Por lo tanto, dada esta configuración de demandas de filas, columnas y longitudes de barcos, vemos que existe una solución al problema de la batalla naval.

2.1. Veamos otro ejemplo:

Instancia del problema de 3-Partition:

$$S = \{111111, 11111, 1111, 11111, 111, 1111111\}$$

Pasamos a no unario:

$$S = \{6, 5, 4, 5, 3, 7\}$$

Calculamos la suma total y hallamos m:

$$\text{sum}(S) = 30 = 3m \rightarrow m = 10$$

Nota: Solución válida de 3-partition:

$$\{1111111, 111\}, \{111111, 1111\}, \{11111, 11111\}$$

Lo transformamos a un problema de la batalla naval:

$$\text{Filas} = [10, 0, 10, 0, 10]$$

$$\text{Columnas} = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]$$

$$\text{Barcos} = [6, 5, 4, 5, 3, 7]$$

Usamos la caja negra de la batalla naval. Veamos una posible solución:

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

Al igual que en el ejemplo anterior, vemos que se cumplen las restricciones de las filas, pues la primer, tercera y quinta fila tienen 10 de demanda cumplida cada una. También se cumplen las restricciones de las columnas, pues en cada columna se coloca exactamente una parte de un barco. Además vemos que se estan colocando todos los barcos (los 6 barcos), respetando las restricciones de adyacencia. Por lo tanto, dada esta configuración de demandas de filas, columnas y longitudes de barcos, vemos que existe una solución al problema de la batalla naval.

Relación con 3-Partition:

- Cada subconjunto S_i corresponde a una fila del tablero.
- Los elementos de cada subconjunto S_i son equivalentes a los barcos en la batalla naval, ocupando exactamente m celdas en esa fila.
- Si se puede resolver el problema de la batalla naval, tras configurar el tablero cumpliendo todas las restricciones mencionadas anteriormente, entonces existe una solución para el problema de 3-Partition.

2.2. Hay 3-Partition → hay PBN:

Supongamos que hay solución en 3-Partition, es decir, dado un conjunto S en representación unaria, pudimos encontrar S_1, S_2, S_3 subconjuntos disjuntos de S tal que $sum(S1) = sum(S2) = sum(S3) = m$.

Resulta entonces:

- Los elementos de cada subconjunto S_i van a estar en una fila de la matriz en la solución de nuestro problema de la batalla naval.
- Los barcos van a ser representados por los elementos de S_i, tal que su longitud van a ser igual al valor de dicho elemento.
- La cantidad de filas van a ser 5. Para respetar las restricciones sobre la adyacencia, hay que colocar una fila vacía tras colocar cada una de las filas que representan los subconjuntos, salvo la última. Luego la demanda de las filas que representan los subconjuntos va a ser igual a m, y las filas vacías van a tener de demanda 0.
- La cantidad de columnas van a ser $sum(S)$, y van a tener todas restricción 1, para asegurarnos de que en una misma columna solo se va a colocar una parte de un barco.

Como pudimos encontrar 3 subconjuntos del conjunto S que cumplen con las características del problema 3-Partition, resulta que, siempre vamos a poder encontrar una configuración válida para la batalla naval, en la cual cada subconjunto S_i representa una fila y $sum(S)$ representa la cantidad de columnas. Además, la demanda cumplida en cada fila con demanda mayor a 0 es igual a m, y la demanda cumplida en cada columna es igual a 1.

2.3. Hay PBN $\rightarrow$ hay 3-Partition:

- Tendremos una configuración en donde la tabla se compone de 5 filas, las cuales 3 tienen demanda mayor a 0 y 2 con demanda igual a 0, y $sum(S)$ columnas con demanda igual 1.
- Al colocar los barcos en dichas filas, resulta que estas se mapean a un subconjunto del problema de 3-Partition: Al tener 3 filas con demanda mayor a 0, tendremos 3 subconjuntos.
- La demanda cumplida es igual en todas las filas cuya demanda es mayor a 0.

Como pudimos encontrar una disposición de barcos que cumplan con la demanda pedida, resulta que hay solución en 3-Partition tal que cada fila no vacía va a ser un subconjunto, y los barcos colocados en dicha fila, van a ser los elementos que pertenecen al subconjunto en cuestión.

Conclusión

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

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

3. Algoritmo de Backtracking

3.1. Codigo

```
1
2 # Verifica si podemos colocar un barco en un casillero
3 def verify_position(grid, row, col, rows, cols, demand):
4
5     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
    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
```

```
72          rows[row + k] -= 1
73          cols[col] -= 1
74
75  # Descoloca el barco horizontalmente
76  def unplace_ship_horizontally(grid, row, col, ship_size, rows, cols):
77      for k in range(ship_size):
78          grid[row][col + k] = None
79          rows[row] += 1
80          cols[col + k] += 1
81
82  # Descoloca el barco verticalmente
83  def unplace_ship_vertically(grid, row, col, ship_size, rows, cols):
84      for k in range(ship_size):
85          grid[row + k][col] = None
86          rows[row + k] += 1
87          cols[col] += 1
88
89  # Calcula la cantidad de espacios libres en la grilla
90  def available_places(grid):
91      empty_places = 0
92
93      for i in range(len(grid)):
94          for j in range(len(grid[0])):
95              if grid[i][j] == None:
96                  empty_places += 1
97
98      return empty_places
99  # Funcion de BT para maximizar la demanda cumplida
100 def ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
    current_idx_ship, memo):
101
102     # Vemos si ya nos encontramos con este mismo escenario, y de ser asi podamos
103     state = (tuple(rows), tuple(cols), current_idx_ship)
104     if state in memo:
105         return
106     memo.add(state)
107
108     # Calculamos las puntuaciones (demandas cumplidas)
109     score_grid = calculate_score(grid)
110     score_best = calculate_score(best_solution_grid)
111
112     # Si encontramos una mejor solucion, la pisamos
113     if score_grid > score_best:
114         for i in range(len(rows)):
115             for j in range(len(cols)):
116                 best_solution_grid[i][j] = grid[i][j]
117
118     # Caso base
119     if (current_idx_ship >= len(ships)):
120         return
121
122     # Poda 1 (la puntuacion que podria llegar a conseguir en el mejor de los casos
    no supera la mejor conseguida)
123     if score_grid <= score_best:
124         available_spaces = available_places(grid)
125         max_possible_score_ships = calculate_possible_max_ships(ships,
    current_idx_ship)
126         max_possible_score = score_grid + min(max_possible_score_ships,
    available_spaces * 2)
127         if max_possible_score <= score_best:
128             return
129
130     # Poda 2 (el barco no entra para ninguna fila/columna por falta de demanda)
131     maximo = max(max(rows), max(cols))
132     while (ships[current_idx_ship] > maximo) and (current_idx_ship < len(ships)):
133         current_idx_ship += 1
134
135     # Con el barco actual, iteramos por la grilla intentando colocarlo
136     ship_size = ships[current_idx_ship]
137     for i in range(len(rows)):
```

```
138         if rows[i] == 0:
139             continue
140         for j in range(len(cols)):
141             if cols[j] == 0:
142                 continue
143             can_place_horizontal = True
144             can_place_vertical = True
145             demand = 0
146 
147             # Vemos si se puede colocar horizontal
148             if ship_size > rows[i]:
149                 can_place_horizontal = False
150             else:
151                 for k in range(ship_size):
152                     if (k == 0):
153                         demand = rows[i]
154                     if (not verify_position(grid, i, j + k, rows, cols, demand)):
155                         can_place_horizontal = False
156                         break
157                     demand -= 1
158 
159             # Vemos si se puede colocar vertical
160             if ship_size > cols[j]:
161                 can_place_vertical = False
162             else:
163                 for k in range(ship_size):
164                     if (k == 0):
165                         demand = cols[j]
166                     if (not verify_position(grid, i + k, j, rows, cols, demand)):
167                         can_place_vertical = False
168                         break
169                     demand -= 1
170 
171             # Colocamos los barcos segun corresponda
172 
173             if (can_place_horizontal):
174                 place_ship_horizontally(grid, i, j, ship_size, rows, cols)
175                 ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
    current_idx_ship + 1, memo)
176                 unplace_ship_horizontally(grid, i, j, ship_size, rows, cols)
177 
178             if (can_place_vertical):
179                 place_ship_vertically(grid, i, j, ship_size, rows, cols)
180                 ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
    current_idx_ship + 1, memo)
181                 unplace_ship_vertically(grid, i, j, ship_size, rows, cols)
182 
183     # Caso en el que decidimos no colocar el barco
184     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

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:
```

```
45              for k in range(ship):
46                  grid[idx_f][idx_c + k] = ship
47                  rows[idx_f] -= 1
48                  cols[idx_c + k] -= 1
49
50  def aproximation(rows, cols, ships):
51      grid = [[None] * len(cols) for _ in range(len(rows))]
52      ships.sort(reverse=True)
53
54      for ship in ships:
55          indice_fila_max = rows.index(max(rows))
56          indice_columna_max = cols.index(max(cols))
57
58          if rows[indice_fila_max] >= cols[indice_columna_max]:
59
60              for j in range(len(cols)):
61                  if verify_position(grid, indice_fila_max, j, rows, cols):
62
63                      if (cols[j] >= ship) and (indice_fila_max + ship <= len(rows)):
64                          can_place = True
65                          for k in range(ship):
66                              if not verify_position(grid, indice_fila_max, j + k,
    rows, cols):
67                                  can_place = False
68                                  break
69                          if can_place:
70                              for k in range(ship):
71                                  grid[indice_fila_max][j + k] = 1 # Se coloca el
    barco
72                                  rows[indice_fila_max] -= 1
73                                  cols[j + k] -= 1
74                              break
75          else:
76                for i in range(len(rows)):
77                  if verify_position(grid, i, indice_columna_max, rows, cols):
78
79                      if (rows[i] >= ship) and (indice_columna_max + ship <= len(cols
    )):
80                          can_place = True
81                          for k in range(ship):
82                              if not verify_position(grid, i + k, indice_columna_max,
     rows, cols):
83                                  can_place = False
84                                  break
85                          if can_place:
86                              for k in range(ship):
87                                  grid[i + k][indice_columna_max] =  1 # Se coloca el
     barco
88                                  rows[i + k] -= 1
89                                  cols[indice_columna_max] -= 1
90                              break
91
92      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.

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

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

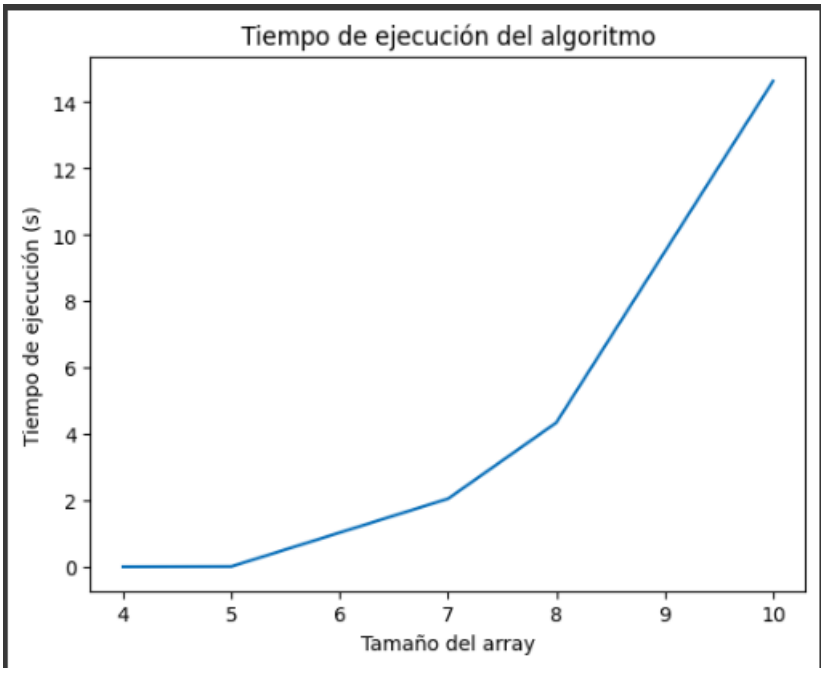
$r(A) = 0{,}3604$.

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

6. Medición empírica

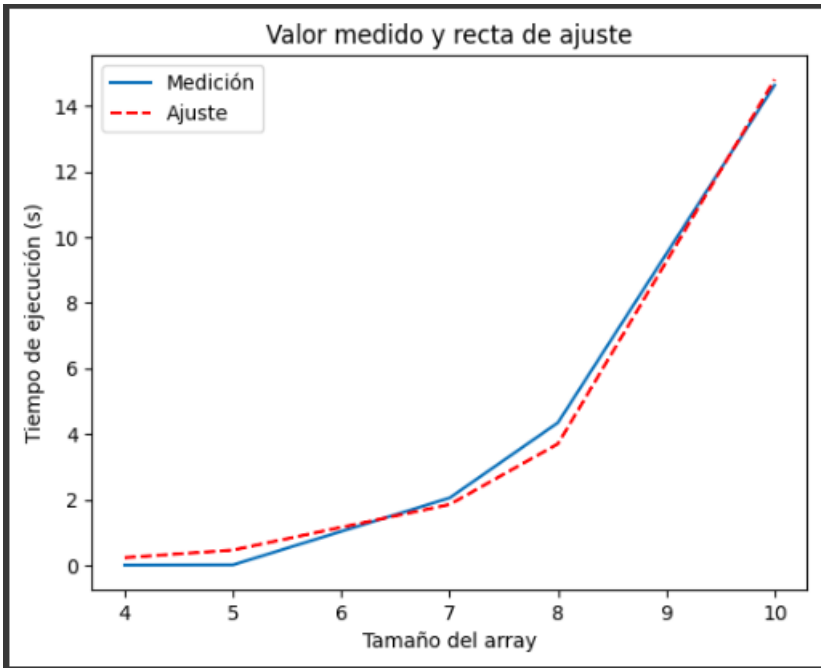
Para comprobar empíricamente la complejidad $\mathbf{O}(2^n)$ del algoritmo, se decidió ejecutar el mismo con distintos tamaños de entrada y medir el tiempo de ejecución. Se generaron muestras de tamaño n, las cuales varían desde 10 hasta 1000.

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.