

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. Código	5
1.5. Conclusion	5
2. Demostración: NP-Completo	6
3. Algoritmo de Backtracking	9
3.1. Código	9
3.2. Análisis	12
4. Ejemplos de ejecución	13
5. Algoritmo de aproximación	16
5.1. Descripción del algoritmo	16
5.2. Análisis del algoritmo	17
6. Medición empírica	18
7. Conclusiones	19

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 saltarlo 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. Código

```
1 from main import *
2
3 # Devuelve true si no hay barcos adyacentes al barco dad, false en caso contrario
4 def validate_adjacency(grid, row, col, ship):
5     # Veo si la posicion es adyacente a un barco
6     for i in range(-1, 2):
7         for j in range(-1, 2):
8             if row + i >= 0 and row + i < len(grid) and col + j >= 0 and col + j <
len(grid[0]):
9                 adjacent_location = grid[row + i][col + j]
10                if adjacent_location != None and adjacent_location != ship:
11                    return False
12            return True
13
14 # Devuelve true si la solucion es valida, false en caso contrario
15 # La solucion debe ser una matriz con el problema resuelto
16 def validator(solution, row_demands, col_demands, ships):
17
18     # Primero validamos adyacencias
19     for i in range(len(row_demands)):
20         for j in range(len(col_demands)):
21             if solution[i][j] != None:
22                 ship = solution[i][j]
23                 valid = validate_adjacency(solution, i, j, ship)
24                 if valid == False:
25                     return False
26
27     # Validamos que las demandas se cumplan
28     for i in range(len(row_demands)):
29         for j in range(len(col_demands)):
30             if solution[i][j] != None:
31                 row_demands[i] -= 1
32                 col_demands[j] -= 1
33
34     for i in range(len(row_demands)):
35         if row_demands[i] != 0:
36             return False
37
38     for j in range(len(col_demands)):
39         if col_demands[j] != 0:
40             return False
41
42     # Validamos que se hayan colocado todos los barcos
43     visited_ships = set()
44     for i in range(len(row_demands)):
45         for j in range(len(col_demands)):
46             if solution[i][j] != None:
47                 visited_ships.add(solution[i][j])
48
49     if len(visited_ships) != len(ships):
50         return False
51
52     # Si llego hasta aca, la solucion es valida
53     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 $S = \{a_1, a_2, \dots, a_n\}$, 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 $S_i = m$
- $\text{sum}(S) = \text{sum}(S_1) + \text{sum}(S_2) + \text{sum}(S_3) = 3m$

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 $A = \{a_1, a_2, \dots, a_n\}$ 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) $B = \{b_1, b_2, \dots, b_n\}$ 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 $S = \{a_1, a_2, \dots, a_n\}$, 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)
- $sum(S_1) = sum(S_2) = sum(S_3)$
- $S_1 \cup S_2 \cup S_3 = S$
- $S_1 \cap S_2 \cap S_3 = \emptyset$

Dado que verificar la representacion unaria, comprobar que los subconjuntos son disjuntos, calcular la suma de los elementos de cada subconjunto, verificar que la sumas de cada subconjunto son iguales, se puede hacer en tiempo polinomial, decimos que 3-Partition esta en NP.

Reducción

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

Consideramos el conjunto $S = \{a_1, a_2, \dots, a_n\}$ 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-Partition, 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 el 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:** 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 $A = \{a_1, a_2, \dots, a_{3m}\}$. 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. Código

```
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]):
6         return False
7
8     if rows[row] == 0 or cols[col] == 0:
9         return False
10
11     if demand == 0:
12         return
13
14     if grid[row][col] != None:
15         return False
16
17     for i in range(-1, 2):
18         for j in range(-1, 2):
19             if row + i >= 0 and row + i < len(grid) and col + j >= 0 and col + j <
len(grid[0]):
20                 if grid[row + i][col + j] != None:
21                     return False
22
23     return True
24
25 # Calcula la demanda cumplida para la grilla
26 def calculate_score(grid):
27     total = 0
28     for row in grid:
29         for cell in row:
30             if cell is not None:
31                 total += 2
32     return total
33
34 # Calcula la demanda que se cumple si se colocasen todos los barcos a partir de un
indice
35 def calculate_possible_max_ships(ships, current_index):
36     score = 0
37     available_ships = ships[current_index:]
38
39     for ship in available_ships:
40         score += ship
41     score *= 2
42
43     return score
44
45 # Funcion principal. Dada una demanda de filas, columnas y barcos, maximiza la
demanda cumplida (minimiza la
46 # demanda incumplida).
47 def ship_placement(rows, cols, ships):
48     grid = [[None] * len(cols) for _ in range(len(rows))]
49     best_solution_grid = [[None] * len(cols) for _ in range(len(rows))]
50     total_amount = sum(rows) + sum(cols)
51     ships.sort(reverse=True)
52
53     # Llamamos al algoritmo de backtracking
54     ship_placement_aux(rows[:], cols[:], ships, grid, best_solution_grid, 0, set()
)
55
56     # Imprimimos resultados
57     print("Gained ammount: ", calculate_score(best_solution_grid))
58     print("Total ammount: ", total_amount)
59     print_grid(best_solution_grid, rows, cols)
60
61 # Coloca el barco horizontalmente
62 def place_ship_horizontally(grid, row, col, ship_size, rows, cols):
63     for k in range(ship_size):
64         grid[row][col + k] = 1
65         rows[row] -= 1
66         cols[col + k] -= 1
```

```
67
68 # Coloca el barco verticalmente
69 def place_ship_vertically(grid, row, col, ship_size, rows, cols):
70     for k in range(ship_size):
71         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
100 # Funcion de BT para maximizar la demanda cumplida
101 def ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
102     current_idx_ship, memo):
103
104     # Vemos si ya nos encontramos con este mismo escenario, y de ser asi podamos
105     state = (tuple(rows), tuple(cols), current_idx_ship)
106     if state in memo:
107         return
108     memo.add(state)
109
110     # Calculamos las puntuaciones (demandas cumplidas)
111     score_grid = calculate_score(grid)
112     score_best = calculate_score(best_solution_grid)
113
114     # Si encontramos una mejor solucion, la pisamos
115     if score_grid > score_best:
116         for i in range(len(rows)):
117             for j in range(len(cols)):
118                 best_solution_grid[i][j] = grid[i][j]
119
120     # Caso base
121     if (current_idx_ship >= len(ships)):
122         return
123
124     # Poda 1 (la puntuacion que podria llegar a conseguir en el mejor de los casos
125     # no supera la mejor conseguida)
126     if score_grid <= score_best:
127         available_spaces = available_places(grid)
128         max_possible_score_ships = calculate_possible_max_ships(ships,
129             current_idx_ship)
130         max_possible_score = score_grid + min(max_possible_score_ships,
131             available_spaces * 2)
132         if max_possible_score <= score_best:
133             return
134
135     # Poda 2 (el barco no entra para ninguna fila/columna por falta de demanda)
136     maximo = max(max(rows), max(cols))
137     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,
176                 current_idx_ship + 1, memo)
177                 unplace_ship_horizontally(grid, i, j, ship_size, rows, cols)
178
179             if (can_place_vertical):
180                 place_ship_vertically(grid, i, j, ship_size, rows, cols)
181                 ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
182                 current_idx_ship + 1, memo)
183                 unplace_ship_vertically(grid, i, j, ship_size, rows, cols)
184
185             # Caso en el que decidimos no colocar el barco
186             ship_placement_aux(rows, cols, ships, grid, best_solution_grid,
187             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.

```
1 Demanda cumplida: 4
2 Demanda total: 11
3 Demanda de filas: [3, 1, 2]
4 Demanda de columnas: [3, 2, 0]
5 -----
6 1 - -
7 - - -
8 1 - -
9 -----
10
11
12 Demanda cumplida: 12
13 Demanda total: 18
14 Demanda de filas: [3, 3, 0, 1, 1]
15 Demanda de columnas: [3, 1, 0, 3, 3]
16 -----
17 1 1 - 1 -
18 - - - 1 -
19 - - - - -
20 1 - - - -
21 1 - - - -
22 -----
23
24
25 Demanda cumplida: 26
26 Demanda total: 53
27 Demanda de filas: [1, 4, 4, 4, 3, 3, 4, 4]
28 Demanda de columnas: [6, 5, 3, 0, 6, 3, 3]
29 -----
30 1 - - - - -
31 1 - 1 - 1 1 -
32 1 - 1 - - - -
33 - - - - 1 1 -
34 1 1 - - - - -
35 - - - - 1 - 1
36 - - - - - - -
37 - - - - - - -
38 -----
39
40
41 Demanda cumplida: 6
42 Demanda total: 14
43 Demanda de filas: [1, 0, 1, 0, 1, 0, 0, 1, 1, 1]
44 Demanda de columnas: [1, 4, 3]
45 -----
46 - - -
47 - - -
48 - - -
49 - - -
50 - - -
51 - - -
52 - - -
53 - 1 -
54 - 1 -
55 - 1 -
56 -----
57
```

```
58
59 Demanda cumplida: 40
60 Demanda total: 40
61 Demanda de filas: [3, 2, 2, 4, 2, 1, 1, 2, 3, 0]
62 Demanda de columnas: [1, 2, 1, 3, 2, 2, 3, 1, 5, 0]
63 -----
64 - - - 1 - 1 - - 1 -
65 - - - 1 - - - - 1 -
66 - - - 1 - - - - 1 -
67 1 1 - - - - 1 - 1 -
68 - - - - 1 - 1 - - -
69 - - - - - - 1 - - -
70 - - - - - - - - 1 -
71 - 1 1 - - - - - - -
72 - - - - 1 1 - 1 - -
73 - - - - - - - - - -
74 -----
75
76
77 Demanda cumplida: 46
78 Demanda total: 58
79 Demanda de filas: [3, 6, 1, 2, 3, 6, 5, 2, 0, 3, 0, 3]
80 Demanda de columnas: [3, 0, 1, 1, 3, 1, 0, 3, 3, 4, 1, 4]
81 -----
82 - - - - - - 1 - 1 - -
83 - - 1 1 1 1 - 1 - 1 - -
84 - - - - - - - - 1 - -
85 - - - - - - - - 1 - 1
86 1 - - - 1 - - - - - 1
87 1 - - - 1 - - - 1 - - 1
88 1 - - - - - - - 1 - - 1
89 - - - - - - - - 1 - - -
90 - - - - - - - - - - - -
91 - - - - - - - 1 - - - -
92 - - - - - - - - - - - -
93 - - - - - - - - - - - -
94 -----
95
96
97 Demanda cumplida: 40
98 Demanda total: 67
99 Demanda de filas: [0, 3, 4, 1, 1, 4, 5, 0, 4, 5, 4, 2, 4, 3, 2]
100 Demanda de columnas: [0, 0, 3, 4, 1, 4, 6, 5, 2, 0]
101 -----
102 - - - - - - - - - -
103 - - 1 1 - - 1 - - - -
104 - - - - - - 1 - - - -
105 - - - - - - 1 - - - -
106 - - - - - - 1 - - - -
107 - - 1 1 - - 1 - 1 - -
108 - - - - - - 1 - 1 - -
109 - - - - - - - - - - -
110 - - 1 - 1 - - 1 - - -
111 - - - - - - 1 - - - -
112 - - - 1 - - - 1 - - -
113 - - - - - - 1 - - - -
114 - - - - - - 1 - - - -
115 - - - - - - - - - - -
116 - - - - - - - - - - -
117 -----
118
119
120 Demanda cumplida: 104
121 Demanda total: 120
122 Demanda de filas: [5, 0, 0, 6, 2, 1, 6, 3, 3, 1, 2, 4, 5, 5, 2, 5, 4, 0, 4, 5]
123 Demanda de columnas: [0, 5, 5, 0, 6, 2, 2, 6, 2, 1, 3, 1, 2, 3, 1, 4, 5, 2, 1, 6]
124 -----
125 - - - - 1 1 1 1 1 - - - - - - - - - -
126 - - - - - - - - - - - - - - - - - -
127 - - - - - - - - - - - - - - - - - -
```

```
128 - - - - 1 1 1 1 1 1 - - - - - - - - -
129 - - 1 - - - - - - - - - - - - - 1
130 - - - - - - - - - - - - - - - 1
131 - - - - - - - - - - 1 1 1 1 1 - - - 1
132 - - 1 - - - - 1 - - - - - - - - 1
133 - - 1 - - - - - - - - - 1 - - - 1
134 - - - - - - - - - - - - - - - 1
135 - 1 - - 1 - - - - - - - - - - -
136 - 1 - - 1 - - 1 - - 1 - - - - - -
137 - 1 - - 1 - - 1 - - 1 - - - 1 - -
138 - 1 - - 1 - - 1 - - 1 - - - 1 - -
139 - 1 - - - - - - - - - - - 1 - -
140 - - - - - - - - - - - 1 - - 1 -
141 - - 1 - - - - - - - - - 1 - -
142 - - - - - - - - - - - - - - -
143 - - 1 - - - - - - - - - 1 - - -
144 - - - - - - - - - - - - - - -
145 -----
146
147
148 Demanda cumplida: 172
149 Demanda total: 247
150 Demanda de filas: [1, 2, 5, 10, 11, 0, 11, 11, 3, 9, 9, 3, 9, 6, 1, 8, 3, 11, 6,
151 7]
151 Demanda de columnas: [5, 4, 5, 2, 10, 1, 0, 8, 7, 6, 0, 5, 4, 8, 4, 7, 4, 0, 8, 5,
152 6, 2, 4, 9, 7]
152 -----
153 1 - - - - - - - - - - - - - - - - -
154 1 - 1 - - - - - - - - - - - - - - -
155 1 - 1 - - - - - 1 - - - - - 1 1 - - -
156 1 - 1 - - - - - 1 - - - - - - - 1 1 1 1 1 1
157 1 - 1 - - 1 - - 1 - - 1 1 1 1 1 1 - - -
158 - - - - - - - - - - - - - - - - -
159 - - 1 - 1 - - 1 - 1 - 1 1 1 1 1 - - -
160 - - - - 1 - - 1 - 1 - - - - - 1 1 1 1 -
161 - - - - 1 - - 1 - 1 - - - - - - - -
162 - - - - 1 - - 1 - 1 - 1 1 1 1 - - -
163 - - - - 1 - - 1 - 1 - - - - - 1 1 1 - 1 1
164 - - - - 1 - - 1 - 1 - - - - - - - -
165 - - - - 1 - - 1 - - - 1 1 1 1 - - -
166 - - - - 1 - - 1 - - - - - - - - -
167 - - - - 1 - - - - - - - - - - -
168 - 1 - - 1 - - - 1 - - - - - - - -
169 - 1 - - - - - 1 - - - - - - - -
170 - 1 - - - - - 1 - - - - - - - -
171 - 1 - - - - - 1 - - - - - - - -
172 - - - - - - - - - - - - - - - -
173 -----
174
175
176 Demanda cumplida: 202
177 Demanda total: 360
178 Demanda de filas: [3, 11, 11, 1, 2, 5, 4, 10, 5, 2, 12, 6, 12, 7, 0, 2, 0, 8, 10,
179 11, 6, 10, 0, 11, 5, 8, 6, 9, 8, 0]
179 Demanda de columnas: [3, 12, 1, 5, 14, 15, 6, 11, 2, 10, 12, 10, 6, 2, 7, 1, 5,
180 11, 5, 10, 7, 11, 4, 0, 5]
180 -----
181 1 - - - 1 - 1 - - - - - - - - - - -
182 - - - - 1 - - - - - - - - - 1 1 1 1 1 1 -
183 - - - - 1 - - - - 1 1 1 1 - - - - - -
184 - - - - 1 - - - - - - - - - - - - -
185 - 1 - - 1 - - - - - - - - - - - - -
186 - 1 - - 1 - - 1 - - - - - - - - - -
187 - 1 - - 1 - - 1 - - - - - - - - - -
188 - 1 - - 1 - - 1 - - - - - 1 1 1 1 1 1 -
189 - 1 - - 1 - - 1 - - - - - - - - - -
190 - 1 - - 1 - - - - - - - - - - - - -
191 - 1 - - 1 - - - - - - - - 1 1 1 1 1 1 -
192 - 1 - - 1 - - - - - - - - - - - - -
193 - 1 - - - - - - - 1 1 1 1 1 1 1 1 1 -
```

211 -----

5.1. Descripción del algoritmo

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

```
40         if not verify_position(grid, idx_f, idx_c + k, rows, cols): # Veo si
puedo poner el barco en esa celda
```



```
41         can_place = False
42         break
43
44     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 aproximacion(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,
67 rows, cols):
68                             can_place = False
69                             break
70                         if can_place:
71                             for k in range(ship):
72                                 grid[indice_fila_max][j + k] = 1 # Se coloca el
73 barco
74                                 rows[indice_fila_max] -= 1
75                                 cols[j + k] -= 1
76                             break
77                     else:
78                         for i in range(len(rows)):
79                             if verify_position(grid, i, indice_columna_max, rows, cols):
80
81                                 if (rows[i] >= ship) and (indice_columna_max + ship <= len(cols
82 ))):
83                                     can_place = True
84                                     for k in range(ship):
85                                         if not verify_position(grid, i + k, indice_columna_max,
86 rows, cols):
87                                         can_place = False
88                                         break
89                                     if can_place:
90                                         for k in range(ship):
91                                             grid[i + k][indice_columna_max] = 1 # Se coloca el
92 barco
93                                             rows[i + k] -= 1
94                                             cols[indice_columna_max] -= 1
95                                         break
96
97     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

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

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.

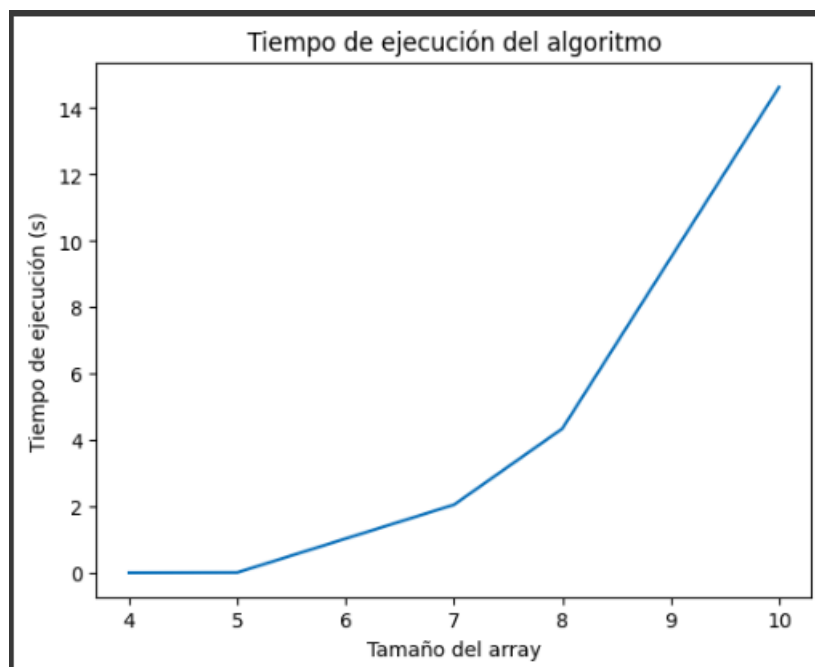
La cota a la que llegamos empíricamente, para ver la relación entre el resultado óptimo, es $r(A) = 0,3604$.

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 $O(2^n)$ 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.

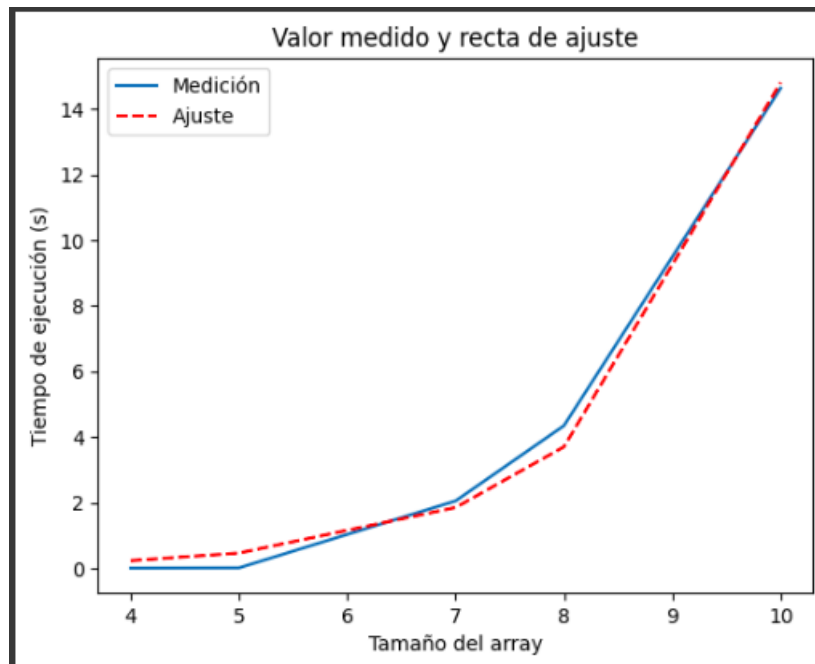
Para cada muestra se registró el tiempo de ejecución, obteniendo el siguiente gráfico:



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 $O(2^n)$.



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 aproximadamente un 33 % de la solución óptima.