



TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 3

Diversión NP-Completa

14 de octubre de 2024

María Delfina Cano Ros Langrehr
109338

mcano@fi.uba.ar

María Sol Orive
91351

morive@fi.uba.ar

Martín Alejandro Rosas
98535

mrosas@fi.uba.ar

Índice

1. Introducción	3
2. Análisis del problema	3
3. Demostración que el problema de Batalla Naval Individual está en NP	3
4. Demostración que el problema de Batalla Naval Individual es NP-Completo	5
4.0.1. Si hay 3P, hay BNI	7
4.0.2. Si hay BNI, hay 3P	8
5. Backtracking	8
6. Programación lineal entera	10
7. Algoritmo de Aproximación	13
7.1. Algoritmo	13
7.2. Complejidad	14
7.3. Análisis de la Aproximación	15
8. Conclusiones	18

1. Introducción

2. Análisis del problema

Para este trabajo práctico, continuamos con los hermanos, Sophia y Mateo, y en este caso juegan a la *Batalla Naval Individual*. Contamos con el siguiente problema: Dado un tablero de nm casilleros, y k barcos. Cada barco i tiene b_i de largo. Es decir, requiere de b_i casilleros para ser ubicado. Todos los barcos tienen 1 casillero de ancho. El tablero a su vez tiene un requisito de consumo tanto en sus filas como en sus columnas. Si en una fila indica un 3, significa que deben haber 3 casilleros de dicha fila siendo ocupados. No se puede ubicar dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente). Debemos ubicar todos los barcos de tal manera que se cumplan todos los requisitos.

3. Demostración que el problema de Batalla Naval Individual está en NP

Buscamos demostrar que el problema de decisión de la batalla naval individual es un problema NP-Completo, para poder realizar esto, debemos demostrar que se encuentra en NP. Para poder realizar esto debemos realizar un validador eficiente que demuestre si la solución es correcta en tiempo polinomial. Por lo tanto plantamos el siguiente problema de decisión:

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

Planteamos el siguiente validador eficiente donde para verificar que la solución es correcta debemos verificar: Planteamos el siguiente validador eficiente donde para verificar que la solución es correcta debemos verificar:

- Si la cantidad de casilleros ocupados de una fila comparada con la cantidad pedida por la restricción de esa fila, no son iguales, es incorrecta la solución, lo mismo sucede para la columna.
- Se verifica que la cantidad de barcos en la solución sea igual a la pedida.
- Se verifica que la longitud de cada uno de los barcos en la solución corresponda con alguno de la lista de barcos a insertar y que no se repitan.
- Ninguno de los barcos sea adyacentes entre sí

```
1 def hay_adyacentes(solucion, casilleros_visitado, x, y):
2     for i in [-1,0,1]:
3         for j in [-1,0,1]:
4             xi = x + i
5             yi = y + j
6
7             if xi != 0 or yi != 0:
8                 if 0 <= xi and xi < len(solucion) and 0 <= yi and yi < len(solucion
9                     [0]):
10                     if solucion[xi][yi] == 1 and not casilleros_visitado[xi][yi]:
11                         return True
12
13     return False
14
15 def es_barco_valido(x, y, solucion, casilleros_visitados, long_barco, es_horizontal
16     ):
```

```
16     if long_barco == 1:
17         return True
18     elif es_horizontal:
19         for i in range(long_barco):
20             if solucion[x][y+i] == 0 or casilleros_visitados[x][y+i] or y + i >=
len(solucion[0]):
21                 return False
22             if hay_adyacente(solucion, casilleros_visitados, x, y+i):
23                 return False
24             casilleros_visitados[x][y+i] = True
25     else:
26         for i in range(long_barco):
27             if solucion[x+i][y] == 0 or casilleros_visitados[x+i][y] or x + i >=
len(solucion):
28                 return False
29             if hay_adyacente(solucion, casilleros_visitados, x+i, y):
30                 return False
31             casilleros_visitados[x+i][y] = True
32
33     return True
34
35
36 def validador(matriz, barcos, restricciones_fil, restricciones_col, solucion):
37     if len(matriz) != len(solucion) or len(matriz[0]) != len(solucion[0]):
38         return False
39
40     fils = len(solucion)
41     cols = len(solucion[0])
42
43     for i in range(fils):
44         if sum(solucion[i]) != restricciones_fil[i]: # la cantidad de casilleros
ocupados por la fila k es distinta a la especificada en la restriccion
45             return False
46
47     for j in range(cols):
48         casilleros_ocupados_cols = sum(solucion[i][j] for i in range(fils))
49         if casilleros_ocupados_cols != restricciones_col[j]:
50             return False
51
52
53     casilleros_visitado = [[False] * cols for _ in range(fils)]
54     barcos_no_visitados = barcos[:] #verifico que todos los barcos solicitados
estén en la solucion
55
56     for i in range(fils):
57         for j in range(cols):
58             if solucion[i][j] == 1 and not casilleros_visitado[i][j]:
59                 casilleros_visitado[i][j] = True
60                 es_horizontal = False
61                 long_barco = 1
62
63                 if j+1 < cols and solucion[i][j+1] == 1:
64                     while j + long_barco < cols and solucion[i][j+long_barco] == 1:
65                         #calculo la longitud del barco actual
66                         long_barco += 1
67                     elif i+1 < fils and solucion[i+1][j] == 1:
68                         es_horizontal = True
69                         while i + long_barco < fils and solucion[i+long_barco][j] == 1:
70                             long_barco += 1
71
72                 if long_barco not in barcos_no_visitados or not es_barco_valido(i,j
, solucion, casilleros_visitado, long_barco, es_horizontal):
73                     return False
74
75                 barcos_no_visitados.remove(long_barco)
76
77     if len(barcos_no_visitados) != 0:
78         return False
79
80     return True
```

Por lo tanto, las complejidades de las funciones son:

- *hay_adyacentes*: ambas iteraciones son de rango igual a 3, entonces, tenemos $i * j = 3 * 3 = 9$, como los valores del rango de la iteración no varían, son constantes, $O(1)$. El resto de las operaciones son $O(1)$, es decir, la complejidad total de la función es $O(1)$.
- *es_barco_valido*: la primera iteración recorre la longitud del barco en caso de que este sea horizontal, y luego realiza operaciones $O(1)$ para validar que no sea inválido y llama a la función *hay_adyacente* que también es $O(1)$. Luego en el caso de que el barco no sea horizontal, también recorre la longitud del barco y realiza las mismas operaciones $O(1)$.

Por lo tanto, la complejidad de la función es $O(L)$ siendo L la longitud del barco.

En el peor de los casos, la longitud del barco es igual al largo de las columnas o filas, en ese caso podría ser: $O(m)$ o $O(n)$.

- *validador*:
 - La primera iteración recorre por completo la cantidad de filas: $O(m)$, siendo m la cantidad de filas
 - La segunda recorre las filas y las columnas para calcular la cantidad de casilleros ocupados en las columnas, por lo tanto, $O(m * n)$, siendo n la cantidad de columnas
 - Luego, se iteran las dimensiones de la matriz, $O(n * m)$ y dentro de esa iteración se calcula, por cada barco que se encuentra se calcula su longitud y se llama a la función *es_barco_valido* que tiene una complejidad de $O(L)$. Por lo tanto este paso es $O(L)$.
 \Rightarrow La complejidad de la función *validador* es $O(n * m) + O(n * m) + O(n * m * L) = O(n * m * L)$.

Como la complejidad del validador es polinomial, por lo tanto, se demuestra que el problema se encuentra en NP.

4. Demostración que el problema de Batalla Naval Individual es NP-Completo

En primer lugar para demostrar que el problema de *Batalla Naval Individual* es un problema NP-Completo debemos buscar un problema más sencillo a este y poder realizar la reducción donde utilizando la caja negra del problema de decisión de BNI se resuelva el problema más simple. Por lo tanto, realizando una cantidad de pasos polinomial y una cantidad polinomial de llamados a la caja negra que devuelve true o false si se pueden ubicar los barcos en la matriz donde ninguno sea adyacente entre sí y se cumplan todas las restricciones podremos realizar la reducción y de esta forma demostrar que el problema se encuentra en NP-C. Si BNI se puede resolver en tiempo polinomial, entonces el problema más simple, también.

Planteamos el problema de *3-Partition en código unario* donde tenemos:

"Dado un conjunto de números, S , cada uno expresado en código unario, debe decidir si puede cumplirse que los números pueden dividirse en 3 subconjuntos disjuntos, tal que la suma de todos los subconjuntos sea la misma. El Problema de 3-Partición en código unario es NP-Completo"

Por lo tanto, el problema 3P cuenta con las siguientes características:

1. S : conjunto de números expresados en código unario
2. ¿Se puede dividir el conjunto de números S en 3 subconjuntos disjuntos tal que la suma de todos los subconjuntos sea la misma?
3. El problema de 3-Partición en código unario es NP-Completo

Por lo tanto planteamos que:

$$3P \leq_p BNI \quad (1)$$

El problema de *3-Partition* es a lo sumo tan difícil que el problema de *Batalla Naval Individual*, si BNI se puede resolver en tiempo polinomial, entonces 3P también.

Ahora para comprender como realizaremos la reducción, planteamos el siguiente ejemplo:

$$S = \{s_1, s_2, s_3, s_4, s_5, s_6\} = \{111, 11, 1, 1, 1, 1\} \quad (2)$$

Dado este ejemplo, podemos visualizar el resultado donde se separa S en los siguientes subconjuntos disjuntos

$$Resultado = [S_1 = [s_1] = [111], S_2 = [s_2, s_3] = [11, 1], S_3 = [s_4, s_5, s_6] = [1, 1, 1]] \quad (3)$$

donde la suma de cada S_i es igual a 3.

Al querer realizar la reducción formulamos una posible reducción de la siguiente manera:

- El conjunto de números S serían los barcos que se deben ubicar en la matriz generada donde al ser un número unario cada uno, este sería la dimensión que debe cumplir el barco. Donde el valor de s_i es la longitud del barco b_i .
- Como debemos separar al conjunto S en 3 subconjuntos disjuntos, conformaremos la matriz del problema de los barcos con 5 filas donde la 1ª, 3ª y 5ª tendrán la restricción de $suma(S)/3$ y así poder separar en los tres subconjuntos y si se pueden ubicar los elementos de S en las tres filas, entonces si se puede separar en 3 subconjuntos. Además, para que también se pueda cumplir la condición de adyacencia, ubicamos en las filas 2 y 4, la restricción/demanda de barcos será 0 así no se ubican barcos en filas y se respetan los 3 posibles subconjuntos a armar.

Ahora para las columnas de la matriz crearemos $sum(A)$ columnas donde cada una de ellas tendrá como valor de restricción 1 así se respeta el problema en su formato unario. Por lo tanto, tendremos una matriz de dimensiones $5 * suma(S)$, las restricciones para las filas serán: $[suma(S)/3, 0, suma(S)/3, 0, suma(S)/3]$, y las restricciones de la columna serán de valor 1 para todas las columnas

Continuando con el ejemplo, así se vería representado el problema BP utilizando la reducción mencionada,

$$A = [111, 11, 1, 1, 1, 1]$$

1	1	1								$suma(A)/3$
										0
				1	1			1		$suma(A)/3$
										0
			1			1			1	$suma(A)/3$
1	1	1	1	1	1	1	1	1	1	

Figura 1: Primer ejemplo de reducción

Se puede visualizar que tenemos las 5 filas, donde se obtienen los 3 subconjuntos de forma efectiva y 9 columnas que es igual a la suma del conjunto S , y se cumplen las restricciones de las columnas.

Formalizando la reducción tenemos que,

- Los valores en formato unario del conjunto S del problema 3P, pasan a ser los barcos a ubicar en el problema BNI, donde el valor s_i es la longitud del barco b_i .
- La matriz del problema BNI tendrá dimensiones $5 * suma(S)$.
- Las restricciones de las filas serán $[suma(S)/3, 0, suma(S)/3, 0, suma(S)/3]$.
- Las restricciones de las columnas serán todas de valor 1.

Es decir se puede resolver un problema de 3-Partition de valores unarios utilizando la caja negra que resuelve el BNI.

A continuación demostraremos que la reducción es correcta, para esto demostraremos la siguiente afirmación:

Hay solución en *3-Partition en formato unario* \longleftrightarrow Hay solución para el problema de *Batalla Naval Individual*.

Demostramos ambas implicancias y que no existen falsos positivos ni negativos para la reducción:

4.0.1. Si hay 3P, hay BNI

Supongamos que hay una solución válida donde se puede partir el conjunto S en tres subconjuntos disjuntos donde:

$$suma(S_1) = suma(S_2) = suma(S_3) = suma(S)/3 \quad (4)$$

En el problema de BNI tenemos:

- Cada uno de los s_i es representado como el barco b_i de longitud s_i .
- Los barcos serán ubicados en una matriz de dimensión $5 * Suma(S)$
- Los barcos que corresponden a cada subconjunto S_i se colocan en las filas 1, 3 y 5, respectivamente donde las restricciones de las filas son $suma(S)/3$ lo cual corresponde con la suma que debe dar cada uno de los subconjuntos
- Las filas 2 y 4 tendrán como restricción 0 así podemos asegurar que ningún barco se ubique en esas filas y este por fuera de alguno de los subconjuntos
- Las columnas tienen restricción 1 lo cual asegura que no se ubique más de un barco en una columna y se respeten las adyacencias entre estos.

Si se puede dividir el conjunto S en 3 subconjuntos disjuntos, se puede asignar los barcos filas correspondientes sin que se excedan las restricciones de estas y la condición de adyacencia verticales se cumple porque las filas 2 y 4 están vacías y la adyacencia horizontal también se cumple al tener restricciones para las columnas de valor 1.

En el caso de que no se pudiesen separar los valores en tres subconjuntos, eso replicaría en que no se puede ubicar todos los barcos en la matriz y que se respeten las restricciones de las filas, columnas y/o adyacencia.

Se demuestra si hay solución en *2-Partition*, hay solución en PMA.

4.0.2. Si hay BNI, hay 3P

Ahora supongamos que existe una solución válida para el problema BNI donde la matriz tiene dimensión $5 * \text{suma}(S)$ donde las restricciones de las filas son $[\text{suma}(B)/3, 0, \text{suma}(B)/3, 0, \text{suma}(B)/3]$ y las de columnas tienen valor 1. Como todos los barcos están ubicados en las filas 1, 3 y 5 por sus restricciones y las filas 2 y 4 también se encuentran vacías. Cada subconjunto S_1, S_2 y S_3 estará formado por los barcos ubicados en las filas 1, 3 y 5, respectivamente.

Si suponemos que no se pueden ubicar los barcos de forma que se respeten las restricciones o las condiciones de adyacencias, pero si tenemos que los subconjuntos S_1, S_2 y S_3 tienen sumas iguales, esto generaría un absurdo ya que la solución proporcionada por 3P en un principio era incorrecta, lo que genera el absurdo ya que partimos de esa hipótesis. Es decir que necesariamente, si hay solución en BNI entonces hay solución en 3P.

En conclusión, se pudo demostrar que no existen falsos positivos ni negativos en la reducción de 3-Partition a BNI. También, transformar el problema 3-Partition a PMA utiliza una cantidad polinomial de pasos, ya que mantenemos el mismo vector de valores $S = B$, generamos la matriz con las dimensiones $5 * \text{suma}(S)$ y por últimos las restricciones son dos listas donde una tiene siempre el mismo tamaño $[\text{suma}(S)/3, 0, \text{suma}(S)/3, 0, \text{suma}(S)/3]$ y las columnas siempre tienen valor 1.

Se concluye que la reducción es polinomial. Finalmente, El problema de *3-Partition* es a lo sumo tan difícil que el problema de *Batalla Naval Individual*, es decir **Batalla Naval Individual** es NP-Completo.

5. Backtracking

Implementamos un algoritmo que resuelva el problema y encuentre el óptimo utilizando *Backtracking*. El siguiente código es nuestro algoritmo

```
1 def hay_adyacentes(solucion, x, y):
2     for i in [-1, 0, 1]:
3         for j in [-1, 0, 1]:
4             xi = x + i
5             yi = y + j
6             if (i != 0 or j != 0) and 0 <= xi < len(solucion) and 0 <= yi < len(
solucion[0]):
7                 if solucion[xi][yi] == 1:
8                     return True
9
10    return False
11
12
13 def se_puede_ubicar(matriz, fil, col, largo, orientacion, restricciones_fil,
restricciones_col):
14     n, m = len(matriz), len(matriz[0])
15     if (orientacion == "horizontal" and col + largo > m) or (orientacion == "
vertical" and fil + largo > n):
16         return False
17
18     if hay_adyacentes(matriz, fil, col):
19         return False
20
21     if orientacion == "horizontal":
22         if sum(matriz[fil]) + largo > restricciones_fil[fil]:
23             return False
24         for i in range(largo):
25             if sum(matriz[r][col + i] for r in range(n)) + 1 > restricciones_col[
col + i]:
26                 return False
27     elif orientacion == "vertical":
28         if sum(matriz[r][col] for r in range(n)) + largo > restricciones_col[col]:
29             return False
30         for i in range(largo):
31             if sum(matriz[fil + i]) + 1 > restricciones_fil[fil + i]:
```



```
32         return False
33
34     return True
35
36
37 def actualizar_posicion_matriz(matriz, i, j, largo, orientacion, value):
38     for k in range(largo):
39         fil, col = (i, j + k) if orientacion == "horizontal" else (i + k, j)
40         matriz[fil][col] = value
41
42
43 def calcular_demanda(matriz, restricciones_fil, restricciones_col):
44     n, m = len(matriz), len(matriz[0])
45     suma_filas = sum(
46         [min(sum(matriz[i]), restricciones_fil[i]) for i in range(n)])
47     suma_columnas = sum(
48         [min(sum(matriz[i][j] for i in range(n)), restricciones_col[j]) for j in
49         range(m)])
50     total_filas = sum(restricciones_fil)
51     total_cols = sum(restricciones_col)
52     demanda_llena = suma_columnas + suma_filas
53     demanda_total = total_filas + total_cols
54     return demanda_llena, demanda_total
55
56 def batalla_naual(matriz, barcos, restricciones_fil, restricciones_col):
57     barcos = sorted(barcos, reverse=True) # ordeno para mejorar la complejidad
58     posiciones = [None] * len(barcos)
59     mejor_solucion = matriz
60     return batalla_naual_bk(matriz, barcos, restricciones_fil, restricciones_col,
61     0, posiciones, mejor_solucion, 0, posiciones)
62
63 def batalla_naual_bk(matriz, barcos, restricciones_fil, restricciones_col, index,
64     posiciones, mejor_solucion, mejor_demanda, mejor_posiciones):
65     demanda_parcial, _ = calcular_demanda(
66         matriz, restricciones_fil, restricciones_col)
67     if index == len(barcos):
68         if demanda_parcial > mejor_demanda:
69             return demanda_parcial, [col[:] for col in matriz], posiciones[:]
70             return mejor_demanda, mejor_solucion, mejor_posiciones
71
72     if index > len(barcos):
73         return mejor_demanda, mejor_solucion, mejor_posiciones
74
75     if demanda_parcial + sum(barcos[i] for i in range(index, len(barcos))) <
76     mejor_demanda:
77         return mejor_demanda, mejor_solucion, mejor_posiciones
78
79     largo_barco = barcos[index]
80     for i in range(len(matriz)):
81         for j in range(len(matriz[0])):
82             if matriz[i][j] == 1:
83                 continue
84                 for orientacion in ["horizontal", "vertical"]:
85                     if se_puede_ubicar(matriz, i, j, largo_barco, orientacion,
86                     restricciones_fil, restricciones_col):
87                         posiciones[index] = (
88                             (i, j), (i + largo_barco - 1, j) if orientacion == "
89                             vertical" else (i, j + largo_barco - 1))
90
91                         actualizar_posicion_matriz(
92                             matriz, i, j, largo_barco, orientacion, 1)
93
94                         posible_demanda, posible_solucion, posible_posiciones =
95                         batalla_naual_bk(
96                             matriz, barcos, restricciones_fil, restricciones_col, index
97                             + 1, posiciones, mejor_solucion, mejor_demanda, mejor_posiciones)
98                         if posible_demanda > mejor_demanda:
99                             mejor_demanda = posible_demanda
```

```

94         mejor_solucion = posible_solucion
95         mejor_posiciones = posible_posiciones
96
97         actualizar_posicion_matriz(
98             matriz, i, j, largo_barco, orientacion, 0)
99         posiciones[index] = None
100
101     posible_demanda2, posible_solucion2, posible_posiciones2 = batalla_naval_bk(
102         matriz, barcos, restricciones_fil, restricciones_col, index + 1, posiciones
103         , mejor_solucion, mejor_demanda, mejor_posiciones)
104     if posible_demanda2 > mejor_demanda:
105         mejor_demanda = posible_demanda2
106         mejor_solucion = posible_solucion2[:]
107         mejor_posiciones = posible_posiciones2[:]
108     return mejor_demanda, mejor_solucion, mejor_posiciones

```

En esta implementación lo que se busca es poder aplicar la mayor cantidad de podas y de esta forma obtener un algoritmo, que no solo que obtenga el óptimo, sino que sea lo más eficiente posible. Decidimos tener un óptimo global y uno parcial que se va modificando a medida que se encuentran los barcos y en el caso de que la demanda del parcial es mejor que de la mejor solución y se iteraron todos los barcos, se devuelve eso y se convierte en la nueva mejor solución. Además, decidimos distintas podas y optimizaciones:

- Se ordenan los barcos para mejorar la complejidad y los tiempos de ejecución
- Si el index es mayor a la cantidad de barcos y la demanda de la matriz parcial es menor que la mejor opción, se poda
- Si la suma de la demanda parcial más los barcos que le faltan, no alcanzan todavía para llegar a la mejor solución encontrada, se poda
- No se inserta ningún barco a menos que se sepa que en esa posición ni sus adyacentes, los lugares estén desocupados. Así no se debe volver para atrás innecesariamente.

Continuando, calculamos tiempo de medición de ejemplos para el algoritmo, validar su corrección y optimicidad.

Archivo	Demanda cumplida	Demanda total	Tiempo Backtracking (s)	Es óptimo
3_3_2.txt	4	11	0.000418	Sí
5_5_6.txt	12	18	0.043368	Sí
8_7_10.txt	26	53	0.004268	Sí
10_3_3.txt	6	14	0.000469	Sí
15_10_15.txt	40	67	0.010741	Sí
12_12_21.txt	40	58	0.148294	Sí

Cuadro 1: Comparación de tiempos Backtracking

6. Programación lineal entera

Para el modelo de programación lineal entera que definimos, nuestra función objetivo se trata de minimizar la demanda incumplida. A la vez, la demanda incumplida será la resta entre la demanda pedida en cada fila/columna y la cantidad de celdas ocupadas de esa fila/columna.

Para esto definimos las siguientes variables de decisión (binarias):

barco en celda : Indica si un determinado barco (ship) con la orientación 'o' está colocado con su celda inicial en la posición (i, j).

barco no colocado: Indica si un determinado barco no está colocado

Las restricciones que se tuvieron en cuenta son:

- Demanda de filas y columnas.
- Cada barco debe estar colocado una sola vez o no colocado. Es decir, cada barco tendrá la suma de 1 entre la suma de las variables definidas.
- No superposición de barcos: En cada celda, puede haber como máximo un barco.
- Asegurar que no haya barcos adyacentes (chequeando las todas las celdas alrededor)

```
1 def resolve_batalla_naval(nro_filas, nro_columnas, ships_len, rows, cols,
2   posibles_posiciones):
3     ships = list(range(len(ships_len))) # Barcos enumerados
4     orientations = ['H', 'V'] # Horizontal (H) o Vertical (V)
5
6     barco_en_celda = LpVariable.dicts(
7         "barco_en_celda",
8         [(ship, o, i, j)
9          for ship in ships
10         for o in orientations
11         for (i, j) in posibles_posiciones[(ship, o)]],
12         0, 1, LpBinary
13     )
14     barco_no_colocado = LpVariable.dicts("barco_no_colocado", ships, 0, 1, LpBinary)
15
16     prob = LpProblem("LaBatallaNaval", LpMinimize)
17     prob += (
18         sum(rows[i] - lpSum(
19             lpSum(
20                 barco_en_celda[(ship, o, i_start, j_start)]
21                 for ship in ships
22                 for o in orientations
23                 for (i_start, j_start) in posibles_posiciones[(ship, o)]
24                 if (o == 'H' and i_start == i and j_start <= j < j_start +
25                 ships_len[ship]) or
26                 (o == 'V' and j_start == j and i_start <= i < i_start +
27                 ships_len[ship])
28             )
29         ) for j in range(nro_columnas)
30         ) for i in range(nro_filas)) +
31         sum(cols[j] - lpSum(
32             lpSum(
33                 barco_en_celda[(ship, o, i_start, j_start)]
34                 for ship in ships
35                 for o in orientations
36                 for (i_start, j_start) in posibles_posiciones[(ship, o)]
37                 if (o == 'H' and i_start == i and j_start <= j < j_start +
38                 ships_len[ship]) or
39                 (o == 'V' and j_start == j and i_start <= i < i_start +
40                 ships_len[ship])
41             )
42         ) for i in range(nro_filas)
43         ) for j in range(nro_columnas)),
44         "Demanda incumplida"
45     )
46
47     # Restriccion 1: Cada barco se coloca exactamente una vez o no se coloca
48     for ship in ships:
49         prob += lpSum(barco_en_celda[(ship, o, i, j)] for o in orientations for (i,
50         j) in posibles_posiciones[(ship, o)]) + barco_no_colocado[ship] == 1
51
52     # Restriccion 2: No superposicion de barcos
53     for i in range(nro_filas):
54         for j in range(nro_columnas):
55             prob += lpSum(
56                 barco_en_celda[(ship, o, i_start, j_start)]
57                 for ship in ships
58                 for o in orientations
```

```

53         for (i_start, j_start) in posibles_posiciones[(ship, o)]
54             if (o == 'H' and i_start == i and j_start <= j < j_start +
ships_len[ship]) or
55                 (o == 'V' and j_start == j and i_start <= i < i_start +
ships_len[ship])
56                 ) <= 1
57
58 # Restriccion 3: No adyacencia entre barcos
59 for ship in ships:
60     ship_len = ships_len[ship]
61     for o in orientations:
62         for (i, j) in posibles_posiciones[(ship, o)]:
63             covered_cells = [(i, j + offset) for offset in range(ship_len)] if
o == 'H' else [(i + offset, j) for offset in range(ship_len)]
64             adjacent_cells = calcular_celdas_adyacentes(nro_filas, nro_columnas
, covered_cells)
65             for (i_adj, j_adj) in adjacent_cells:
66                 prob += lpSum(
67                     barco_en_celda[(ship_adj, o_adj, i_start, j_start)]
68                     for ship_adj in ships
69                     for o_adj in orientations
70                     for (i_start, j_start) in posibles_posiciones[(ship_adj,
o_adj)]
71                     if (o_adj == 'H' and i_start == i_adj and j_start <= j_adj
< j_start + ships_len[ship_adj]) or
72                         (o_adj == 'V' and j_start == j_adj and i_start <= i_adj
< i_start + ships_len[ship_adj])
73                     ) + barco_en_celda[(ship, o, i, j)] <= 1
74
75 # Restriccion 4: Demanda maxima por fila
76 for i in range(nro_filas):
77     prob += lpSum(
78         lpSum(
79             barco_en_celda[(ship, o, i_start, j_start)]
80             for ship in ships
81             for o in orientations
82             for (i_start, j_start) in posibles_posiciones[(ship, o)]
83             if (o == 'H' and i_start == i and j_start <= j < j_start +
ships_len[ship]) or
84                 (o == 'V' and j_start == j and i_start <= i < i_start +
ships_len[ship])
85             )
86         for j in range(nro_columnas)
87     ) <= rows[i]
88
89 # Restriccion 5: Demanda maxima por columna
90 for j in range(nro_columnas):
91     prob += lpSum(
92         lpSum(
93             barco_en_celda[(ship, o, i_start, j_start)]
94             for ship in ships
95             for o in orientations
96             for (i_start, j_start) in posibles_posiciones[(ship, o)]
97             if (o == 'H' and i_start == i and j_start <= j < j_start +
ships_len[ship]) or
98                 (o == 'V' and j_start == j and i_start <= i < i_start +
ships_len[ship])
99             )
100         for i in range(nro_filas)
101     ) <= cols[j]
102
103 # Resolver el modelo
104 prob.solve()
105 return prob, barco_en_celda

```

En el caso de la programación lineal entera, el tiempo es mucho mayor que los tiempos que tenemos en backtracking. Esto tiene que ver con la cantidad de variables necesarias para poder modelar el problema. De hecho, hemos utilizado una función auxiliar *posibles posiciones* para poder pre-calcular las posibles posiciones que podrían tener los barcos ya limitado por el tamaño

del tablero para mantener los tiempos dentro de un límite razonable y poder armar este informe.

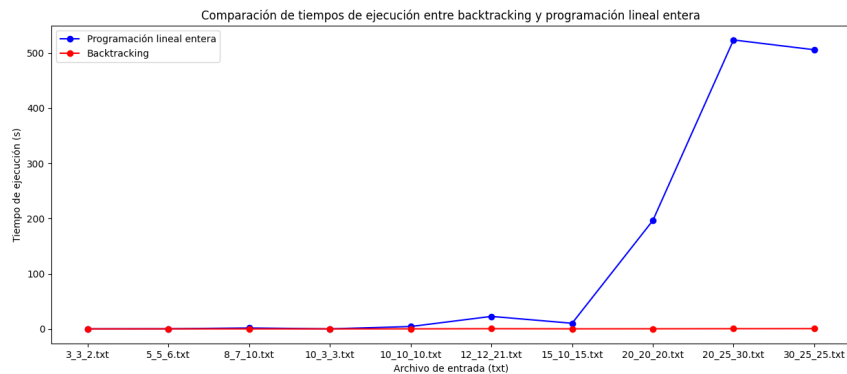


Figura 2: Comparación de tiempos con backtracking

7. Algoritmo de Aproximación

7.1. Algoritmo

Planteamos el siguiente algoritmo cumpliendo con los requisitos del enunciado donde se debe 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. Debemos aplicar esto hasta que no queden más barcos o no haya más demandas a cumplir.

```

1 def hay_adyacentes(solucion, x, y):
2     for i in [-1, 0, 1]:
3         for j in [-1, 0, 1]:
4             xi = x + i
5             yi = y + j
6             if (i != 0 or j != 0) and 0 <= xi < len(solucion) and 0 <= yi < len(
solucion[0]):
7                 if solucion[xi][yi] == 1:
8                     return True
9
10    return False
11
12
13 def es_posicion_valida(matriz, barco, x, y, es_horizontal, restricciones_fils,
restricciones_cols):
14     if es_horizontal:
15         # si no entra en la fila, no se puede agregar
16         if y + barco > len(matriz[0]):
17             return False
18
19         # si el barco ocupa mas de lo pedido en la restriccion de la fila, no se
puede agregar
20         if barco > restricciones_fils[x]:
21             return False
22
23         for i in range(barco):
24             if matriz[x][y + i] == 1 or hay_adyacentes(matriz, x, y + i) or
restricciones_cols[y + i] <= 0:
25                 return False
26         return True
27     else:
28         if x + barco > len(matriz):
29             return False
30
31         if barco > restricciones_cols[y]:
32             return False

```

```
33
34     for i in range(barco):
35         if matriz[x+i][y] == 1 or hay_adyacentes(matriz, x+i, y) or
restricciones_fils[x+i] <= 0:
36             return False
37     return True
38
39
40 def buscar_posicion(matriz, barco, restricciones_fils, restricciones_cols):
41     if max(restricciones_fils) > max(restricciones_cols):
42         for i, demanda in enumerate(restricciones_fils):
43             if demanda >= barco:
44                 for j in range(len(matriz[0])):
45                     if es_posicion_valida(matriz, barco, i, j, True,
restricciones_fils, restricciones_cols):
46                         return i, j, True
47                     # si no lo pude insertar dependiendo de la demanda de la fila,
lo salteo
48             else:
49                 for i, demanda in enumerate(restricciones_cols):
50                     if demanda >= barco:
51                         for j in range(len(matriz)):
52                             if es_posicion_valida(matriz, barco, j, i, False,
restricciones_fils, restricciones_cols):
53                                 return j, i, False
54
55     print("No se puede insertar")
56     return -1, -1, False
57
58
59 def colocar_barco_y_ocupar_casilleros(matriz, barco, x, y, es_horizontal,
restricciones_fils, restricciones_cols):
60     if barco == 1:
61         matriz[x][y] = 1
62     elif es_horizontal:
63         for i in range(barco):
64             matriz[x][y + i] = 1
65             restricciones_fils[x] -= 1
66             restricciones_cols[y+i] -= 1
67     else:
68         for i in range(barco):
69             matriz[x + i][y] = 1
70             restricciones_fils[x+i] -= 1
71             restricciones_cols[y] -= 1
72
73
74 # asumo que recibo una matriz llena de 0s
75 def aproximacion(matriz, barcos, restricciones_fils, restricciones_cols):
76     barcos.sort(reverse=True)
77
78     for barco in barcos: # empiezo con los de mayor tamaño
79         x, y, es_horizontal = buscar_posicion(
80             matriz, barco, restricciones_fils, restricciones_cols)
81         if x != -1:
82             colocar_barco_y_ocupar_casilleros(
83                 matriz, barco, x, y, es_horizontal, restricciones_fils,
restricciones_cols)
84
85     return matriz
```

7.2. Complejidad

Ahora, analicemos la complejidad del algoritmo implementado para la aproximación:

- *hay_adyacentes*: al igual que en el validador eficiente, esta función tiene una complejidad constante ya que ambas iteraciones son de rango igual a 3 y al no variar, son constantes, $O(1)$. El resto de las operaciones son $O(1)$, es decir, la complejidad total de la función es

$O(1)$.

- *es_posicion_valida*: esta función es muy similar a la del validador eficiente del punto 1, donde se verifica si las posición x e y de la matriz está libre y se puede empezar a colocar el barco a partir de ahí. Se verifica que las posiciones adyacentes no estén ocupadas por otro, que se respeten las restricciones y no se pase del valor pedido. Al iterar únicamente el largo del barco y dentro de la iteración se realizan operaciones $O(1)$ y el llamado a la función *hay_adyacentes*, la complejidad de la función es $O(l)$, donde l es largo del barco actual.
- *buscar_posicion*: se busca la restricción de mayor valor entre las columnas y filas, si esta entre las filas, se itera el vector de restricciones de filas, para encontrar la demanda y una vez encontrada la posición, se iteran las columnas para validar si es posible ubicar el barco de forma horizontal en la matriz, utilizando la función *es_posicion_valida*. Por lo tanto la complejidad de la función es $O(m * n * l)$, siendo m la cantidad de filas, n la cantidad de columnas y l el largo del barco actual, esto proviene de la complejidad *es_posicion_valida*.
- *colocar_barco_y_ocupar_casilleros*: en este caso de la complejidad de la función también es $O(L)$ ya que se recorre el largo del barco actual también y el resto de operaciones son $O(1)$.
- *aproximacion*: se ordenan los barcos por largo de mayor a menor, por lo tanto esto es $O(B \log(B))$, siendo B el tamaño del vector de barcos y luego se itera el vector de los barcos y se llama la función *buscar_posicion*, que tiene una complejidad $O(m * n * b)$ y a *colocar_barco_y_ocupar_casilleros*, $O(l)$. Finalmente, la complejidad es $O(B \log(B)) + O(B) * (O(m * n * l) + O(l)) \rightarrow O(B * m * n * l)$.

Por lo tanto, la complejidad final de la función de aproximación es $O(B * m * n * l)$.

7.3. Análisis de la Aproximación

Tras haber calculado las mediciones de tiempo del algoritmo por backtracking y mediciones con el algoritmo de aproximación, se puede ver una gran diferencia entre ambas.

Se observa que la solución exacta en todos los casos tardó más en ejecutarse pero dio el mismo resultado correcto que la aproximación, excepto en los casos mas grandes. Respecto a los tiempos, en ejemplos pequeños ambos dan buenos números, claro hay una tendencia clara en el caso exacto donde sería menos eficiente escalarlo. En cambio, el algoritmo de aproximación se comporta de una manera que aparenta ser mucho más escalable.

También se calculó a relación de aproximación $r(A)$. Para esto, realizamos dichas medidas. La relacion $r(A)$ se define como “la proporción entre la solución óptima $Z(I)$ y la solución aproximada $A(I)$, la cual proporciona una medida de cuanto se puede desviar la solución aproximada de la optima en el peor de los casos. Esto sirve para evaluar la efectividad del algoritmo de aproximación por su capacidad para encontrar soluciones que sean cercanas a la optima. Como resultado, vemos que se mantiene en 1 excepto en cantidades altas como para el caso de (20, 25, 30) y (30 25 25) lo cual indicar que la aproximación es confiable solo en ejemplos no muy grandes y no la podríamos escalar mucho.

Archivo	Tiempo Aproximación	A(I)	Tiempo Backtracking	z(I)	r(A)
3_3_2.txt	0.0000283	6	0.0001459	6	1.000000
5_5_6.txt	0.0001308	12	0.0866703	12	1.000000
8_7_10.txt	0.0001566	26	0.0080757	26	1.000000
10_3_3.txt	0.0001778	6	0.0008003	6	1.000000
10_10_10.txt	0.0002043	40	0.0635857	38	1.052631
12_12_21.txt	0.0012989	40	0.3379466	40	1.000000
15_10_15.txt	0.0004355	40	0.1211435	40	1.000000
20_25_30.txt	0.0275385	112	0.3327887	172	0.651162
30_25_25.txt	0.0133488	114	0.4561674	150	0.760000

Cuadro 2: Comparación de tiempos entre Backtracking y Aproximación

Para una mejor visualización, armamos un gráfico comparando los tiempos de ejecución y la optimalidad

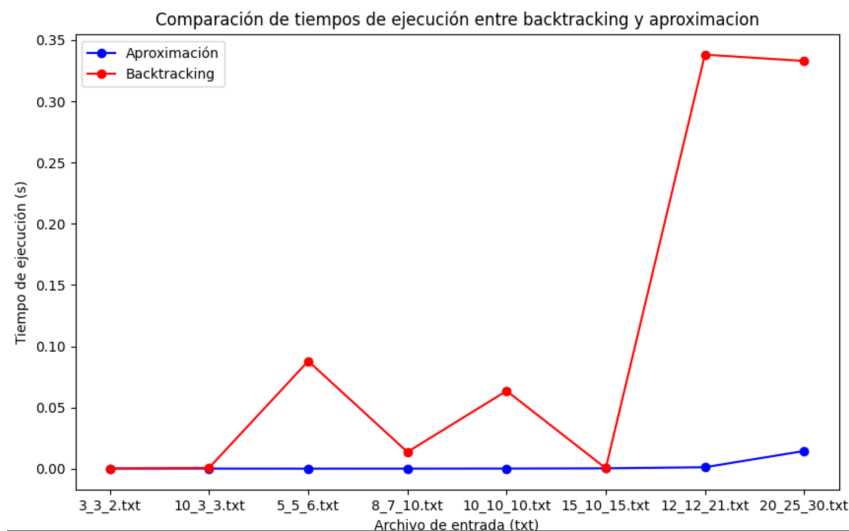


Figura 3: Comparación de tiempos entre backtracking y aproximación

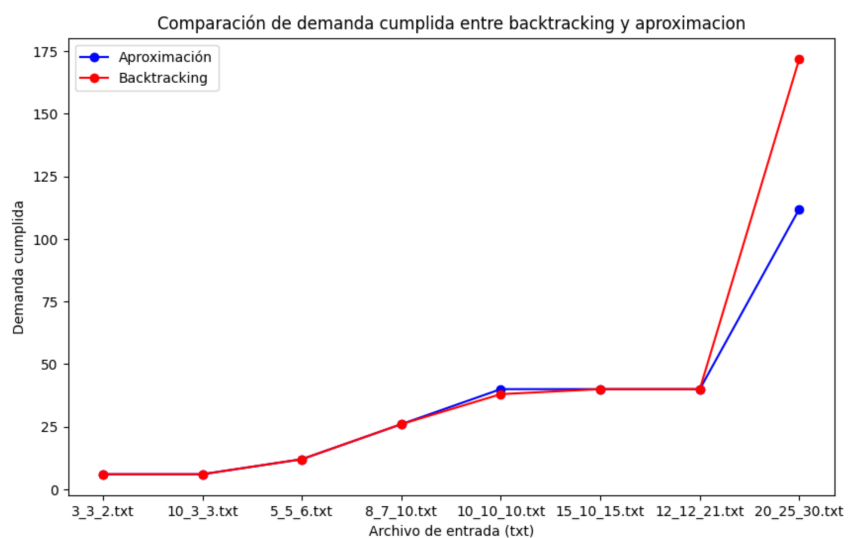


Figura 4: Comparación de cumplimiento de demanda

Como se puede observar el algoritmo de backtracking demora un poco más en la ejecución pero obtiene la solución óptima donde si se ve se obtienen las soluciones muy parecidas pero el algoritmo de backtracking es válida.

8. Conclusiones

Finalmente, a lo largo del trabajo, se abordaron diferentes enfoques para resolver el problema, desde su versión de problema de decisión para validar si el problema se encuentra en NP y NP-completo hasta la implementación y comparación de algoritmos exactos y aproximados.

En primer lugar, realizamos la demostración que el problema pertenece a NP porque se puede realizar un validador en tiempo polinómico. Y luego, mediante una reducción apropiada, también probamos que es NP-Completo. Por otra parte, implementamos un algoritmo utilizando la técnica *backtracking* donde podemos obtener soluciones óptimas al problema, pero su complejidad crece exponencialmente con el tamaño de la matriz y el número de barcos a ubicar.

Continuando implementamos otro algoritmo donde obtenemos la solución óptima utilizando programación lineal. Finalmente realizamos un algoritmo de aproximación el cuál es rápido pero no eficiente ya que la solución proporcionada no es óptima. Para finalizar, realizamos mediciones para comparar la optimalidad y tiempos de ejecución de los diferentes algoritmos.