.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Diversión NP-Completa

21 de noviembre de 2024

Alejo Fábregas	Camilo Fábregas	Juan Cruz Hernández
106160	103740	105711

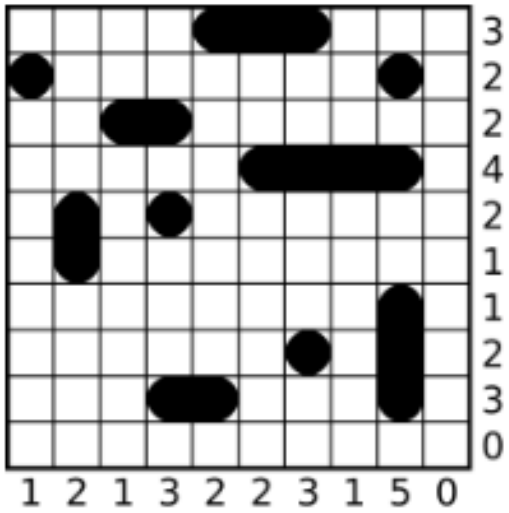
Diversión NP-Completa

1. Introducción

En el presente trabajo práctico analizaremos el problema de La Batalla Naval Individual, una versión para resolver en papel y en forma individual del tradicional juego de tablero, que fue inventada por el argentino Jaime Poniachik en 1982.

Este juego tiene un tablero de n filas por m columnas, y una cierta cantidad k de barcos. Los barcos tienen un largo bi, y siempre tienen un ancho de 1 casillero. Además, hay ciertas restricciones que hay que cumplir a la hora de colocar los barcos. Cada fila y cada columna tiene una demanda de espacios que deben ser ocupados por los barcos. Asimismo, los barcos no pueden ser adyacentes: tiene que haber un espacio entre un barco y otro (incluso de forma diagonal).

A continuación, adjuntamos un ejemplo:



Se tratará este problema en sus versiones de decisión y de optimización:

- Decisión: dada una instancia del problema, si se puede definir una ubicación para los barcos que cumpla todas las restricciones de esa instancia.
- Optimización: dada una instancia del problema, definir una ubicación para los barcos que minimice la demanda incumplida.

Primero vamos a evaluar la dificultad del problema, viendo si es un problema NP y luego si es NP-Completo, haciendo las demostraciones correspondientes.

Más adelante resolveremos el problema con distintas metodologías: en particular Backtracking y Programación Lineal, que deberían darnos una solución óptima. Tomaremos mediciones de tiempos de ejecución y las compararemos.

Por último, utilizaremos un algoritmo de aproximación propuesto en el enunciado para resolver el problema, y analizaremos qué tanto se desvía de la solución óptima, es decir, cuán buena es la aproximación en cuestión.

2. Demostración problema es NP (ej 1)

Un problema pertenece al conjunto NP si cuenta con un verificador eficiente. Es decir, dada un candidato a solución, se puede verificar si es correcto o no en tiempo polinómico.

Una solución candidata al problema de la Batalla Naval tendría los siguientes elementos:

- Una matriz n x m donde cada casillero puede ser 0 o 1, indicando si está ocupado por un barco o no.
- 2 arrays (uno para las filas y otro para las columnas) en donde se indique el requisito de consumo de las filas y columnas.
- El listado de barcos.

Dada esa información, la solución es valida si:

- No hay barcos adyacentes (ni en diagonal)
- Se satisfacen todos los requisitos de consumo (ni más ni menos).
- Todos los barcos fueron ubicados.

Ahora hay que analizar la complejidad de verificar que se cumplan las reglas:

1. **Verificar los requisitos de consumo**: por cada fila y columna hay que contar la cantidad de vértices ocupados y ver que coincida con el requisito.

   Eso son, para las filas, n veces recorrer arreglos de largo m y, para las columnas, m veces recorrer arreglos de largo n.

   Complejidad: $\mathcal{O}(n \cdot m) = \mathcal{O}(n \cdot m)$.

2. **Verificar que no haya barcos adyacentes**: se traduce a que no haya ningún casillero ocupado que tenga otro casillero ocupado como adyacente.

   Esto consiste en iterar cada casillero y verificar hasta 8 vecinos como máximo. Verificar es $\mathcal{O}(1)$.

   Complejidad: $\mathcal{O}(n \cdot m \cdot 8) = \mathcal{O}(n \cdot m)$.

3. **Verificar que todos los barcos fueron usados**: al no haber adyacentes, esto consiste en recorrer de izquierda a derecha y de arriba a abajo a la matriz. Cuando encuentro un casillero ocupado, lo seteo como desocupado y avanzo todo lo que pueda por los adyacentes, también marcandolos como desocupados. Me guardo la longitud del barco en una lista.

   Sigo recorriendo la matriz y repitiendo el proceso hasta llegar al final. El costo se reduce a recorrer toda la matriz una sola vez y recorrer los casilleros de los barcos, lo cual es acotable por $\mathcal{O}(n \cdot m)$, ya que si todos los casilleros pudieran ser ocupados la complejidad sería $\mathcal{O}(2 \cdot n \cdot m) = \mathcal{O}(n \cdot m)$. Dada la restricción de adyacencia, el peor caso es mejor que esto.

   Una vez finalizado, verifico que la lista resultante sea igual a la de los barcos a ubicar. Es necesario ordenar las listas para compararlas por lo que la complejidad es $\mathcal{O}(k \cdot log(k))$.

   Complejidad: $\mathcal{O}(n \cdot m + k \cdot log(k))$.

Los 3 pasos son polinomiales, por lo que la complejidad resultante también es polinomial.

Se encontró un verificador eficiente para el problema de la Batalla Naval, por lo que queda demostrado que pertenece a NP.

3. Demostración problema es NP-Completo (ej 2)

Para demostrar que un problema X es NP-Completo, se puede intentar reducir otro problema NP-Completo al problema X. Si esto es posible, X es NP-Completo, ya que un problema NP-Completo solo puede ser reducido a otro problema NP-Completo.

3.1. Problema a reducir: Bin Packing (versión unaria)

El objetivo es demostrar que Bin Packing Unario $\leq_p$ Batalla Naval.

El problema consiste en ocupar contenedores de tamaño fijo con elementos de tamaño variable, buscando minimizar la cantidad de contenedores utilizados.

En la versión unaria, el tamaño del elemento se codifica como n veces el 1 repetido.

La versión de Bin Packing como **problema de decisión** consiste en si se puede o no ubicar los elementos en los k bins.

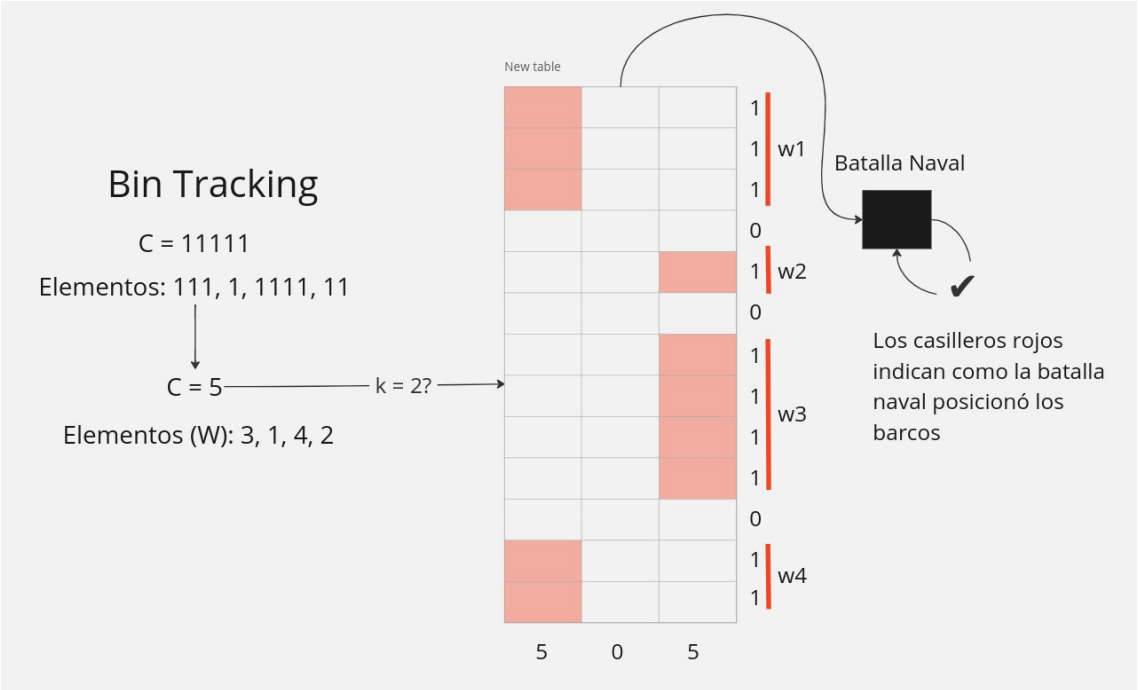
Entrada:

- **W**: lista de tamaños en representación unaria.
- **C**: capacidad fija para todos los contenedores.
- **k**: cantidad de bins en los que se quiere ubicar los elementos.

3.2. Reducción Bin Packing Unario $\leq_p$ Batalla Naval

Se busca resolver el problema de Bin Packing Unario, utilizando al problema de Batalla Naval (ambos en su versión de problema de decisión) como caja negra y con una cantidad polinómica de llamados.

Para esto es necesario transformar la entrada de Bin Tracking a una que el problema de la Batalla Naval pueda resolver. En la imágen a continuación se muestra de forma gráfica esta transformación:



La transformación consiste en lo siguiente:

1. Los items W se van a representar como barcos de largo w_i.
2. La capacidad de los bins se representa como el requisito de consumo de las columnas y por cada bin se crea una columna con requisito C.

   Dado que los barcos no se pueden ubicar adyacentemente, se crean las columnas alternando el requisito de consumo entre 0 y C.
3. Para las filas, lo que se hace es crear (w_1 + w_2 + ... + (len(W) - 1)) filas. Por cada item de largo w_i, se crean w_i filas adyacentes con requisito de consumo 1. Para respetar la no adyacencia es necesario agregar espacios en blanco (len(W) - 1 espacios).

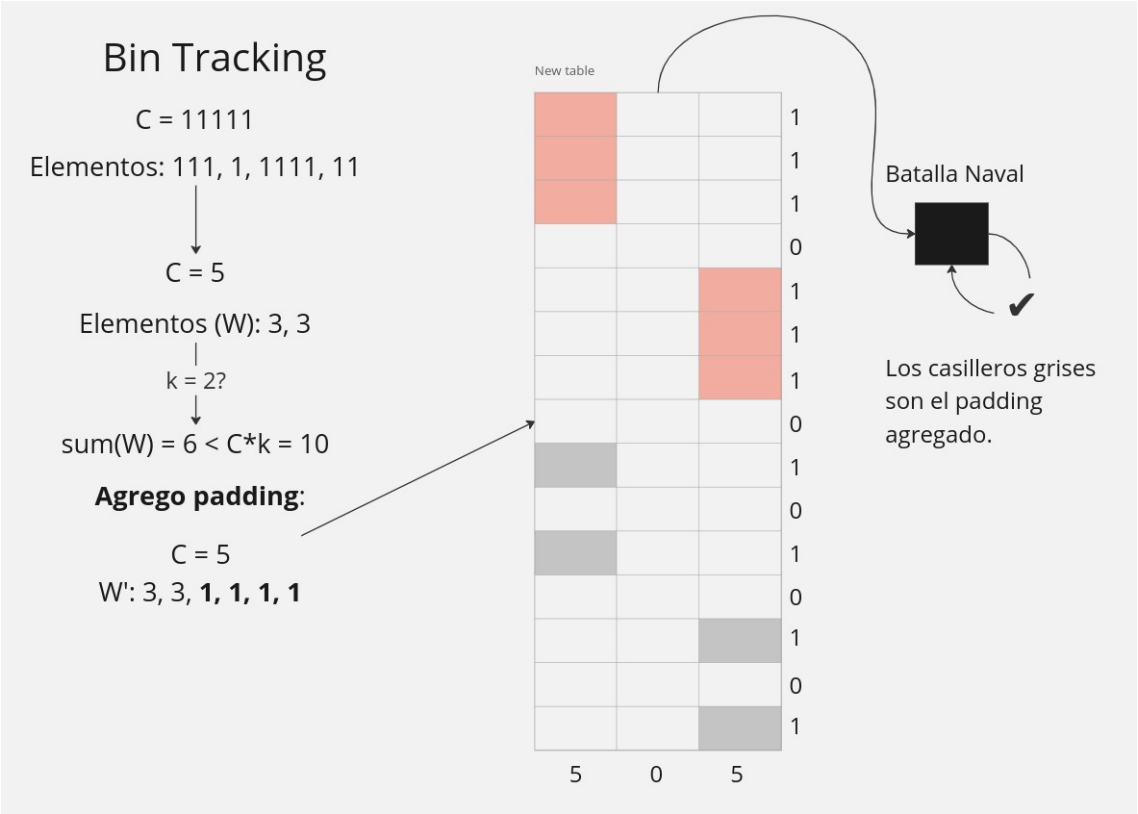
Esta reducción sigue siendo insuficiente, ya que solo funciona cuando la solución al problema de bin tracking es óptima (todos los elementos se pueden ubicar ocupando el 100 % de todos los bins). Esto no funciona en casos como el siguiente: para W = [3, 3] y C = 5, va a retornar false siempre, cuando debería devolver true para k ¿= 2. Esto porque nunca va a satisfacer el requisito de consumo de 5 de las columnas.

La solución consiste en agregar un "padding"de barcos de largo 1 al listado de elementos para que se alcance una solución óptima (la cantidad de bins necesarios no se va a ver afectada, solo estamos rellenando el espacio libre para satisfacer el requisito de las columnas y que funcione la caja negra de la batalla naval).

Para que el bin tracking tenga solución óptima, es necesario que $\sum w_i = C \cdot K$, donde K es la cantidad de bins y que ningún elemento exceda por si solo la capacidad C.

Por lo tanto la reducción completa tiene un paso más:

4. Agregar tantos barcos de longitud 1 como sea necesario para lograr que $\sum w_i = C \cdot K$.



Entonces, la resolución del problema de Bin Tracking consistió de armar el tablero, lo cual se

hace en tiempo polinómico (determinar las filas, columnas y agregar el padding si es necesario) e ir probando con valores de k hasta que la caja negra retorne true, también polinómico.

Al existir una reducción polinómica de un problema NP-Completo (Bin Tracking) a el problema de la Batalla Naval, queda demostrado que este también es un problema NP-Completo.

4. Resolución por Backtracking (ej 3)

En este apartado se propone un algoritmo por Backtracking que permite resolver la versión de optimización del problema de La Batalla Naval Individual, obteniendo la solución óptima.

El algoritmo prueba colocando o no a todos los barcos en todas las posiciones, explorando todas las posibilidades. Por cada barco, primero prueba no colocarlo y seguir con los siguientes barcos, y luego prueba colocándolo en todas las posiciones (siempre que se pueda, sea por las adyacencias o las demandas). Los barcos se ordenan del más largo al más corto para ir descartando opciones inválidas de antemano (por las restricciones).

El caso base es que no hayan más barcos para colocar, por lo que termina el algoritmo.

La poda del árbol de posibilidades se realiza cuando la demanda incumplida de la solución parcial (suma de demandas de columnas y de filas) es menor que la mínima demanda incumplida que se podría obtener si se colocan todos los barcos restantes. En ese caso no tiene sentido continuar explorando ya que incluso en el mejor caso, si se colocaran todos los barcos restantes, no se puede mejorar la solución actual.

Recordamos que al ser un problema NP-Completo, como ya fue demostrado, este algoritmo es exponencial.

A continuación podemos ver el algoritmo. Más adelante se harán mediciones de tiempo para comparar con otras metodologías.

```python
# Ejecuta el algoritmo de backtracking para la batalla naval individual, creando el tablero con las demandas de filas y columnas.
def batalla_naval(largo_barcos, demandas_fil, demandas_col):
    n = len(demandas_fil)
    m = len(demandas_col)
    tablero = [[0] * m for _ in range(n)]
    largo_barcos.sort(reverse=True)
    solucion = [tablero, float("inf")]

    batalla_naval_bt(tablero, largo_barcos, demandas_fil, demandas_col, solucion)

    for fila in solucion[TABLERO]:
        print(fila)

    demanda_total = sum(demandas_fil) + sum(demandas_col)
    demanda_cumplida = demanda_total - solucion[DEMANDA_INCUMPLIDA]

    return (demanda_cumplida, demanda_total)


# Algoritmo de backtracking para resolver el problema de optimizacion de la batalla naval individual.
def batalla_naval_bt(
    tablero, largo_barcos, demandas_fil, demandas_col, solucion_parcial
):
    demanda_incumplida = sum(demandas_fil) + sum(demandas_col)
    # Actualizar mejor solucion
    if demanda_incumplida < solucion_parcial[DEMANDA_INCUMPLIDA]:
        solucion_parcial[TABLERO] = [list(fila) for fila in tablero]
        solucion_parcial[DEMANDA_INCUMPLIDA] = demanda_incumplida
    # Caso base
    if not largo_barcos:
        return
    # Poda
    if solucion_parcial[DEMANDA_INCUMPLIDA] <= demanda_incumplida - sum(
        barco * 2 for barco in largo_barcos
    ):
        return

    # Pruebo sin colocar un barco
    batalla_naval_bt(
        tablero, largo_barcos[1:], demandas_fil, demandas_col, solucion_parcial
    )
    # Pruebo colocando un barco
```

```
    for i in range(len(tablero)):
        if demandas_fil[i] > 0:  # Procesar solo si hay demanda en la fila
            for j in range(len(tablero[0])):
                if demandas_col[j] > 0:  # Procesar solo si hay demanda en la columna
                    procesar_barco(
                        tablero,
                        largo_barcos,
                        i,
                        j,
                        demandas_fil,
                        demandas_col,
                        solucion_parcial,
                        True,
                    )
                    procesar_barco(
                        tablero,
                        largo_barcos,
                        i,
                        j,
                        demandas_fil,
                        demandas_col,
                        solucion_parcial,
                        False,
                    )


# Prueba de colocar el barco. Si se puede, sigue probando alternativas, hasta terminar y quitar el barco.
def procesar_barco(
    tablero,
    largo_barcos,
    i,
    j,
    demandas_fil,
    demandas_col,
    solucion_parcial,
    es_horizontal,
):
    if intentar_ubicar_barco(
        tablero, largo_barcos[0], i, j, es_horizontal, demandas_fil, demandas_col
    ):
        ubicar_barco(
            tablero, largo_barcos[0], i, j, demandas_fil, demandas_col, es_horizontal
        )
        batalla_naval_bt(
            tablero, largo_barcos[1:], demandas_fil, demandas_col, solucion_parcial
        )
        quitar_barco(
            tablero, largo_barcos[0], i, j, demandas_fil, demandas_col, es_horizontal
        )


# Intenta ubicar el barco, horizontalmente en la fila i_fil, o verticalmente en la columna i_col (segun el caso).
# Chequea que no se salga del tablero, que los casilleros a tomar no esten siendo utilizados por otro barco y que no tenga barcos adyacentes.
def intentar_ubicar_barco(
    tablero, largo_barco, i_fil, i_col, es_horizontal, demandas_fil, demandas_col
):
    n, m = len(tablero), len(tablero[0])

    # Verificar que el barco no salga del tablero
    if es_horizontal:
        if i_col + largo_barco > m:
            return False
    else:  # Es vertical
        if i_fil + largo_barco > n:
```

```
            return False
    # Verificar que no se violen las demandas de fila y columna
    if es_horizontal:
        if demandas_fil[i_fil] < largo_barco:
            return False
        for i in range(largo_barco):
            if demandas_col[i_col + i] < 1:
                return False
    else:  # Es vertical
        if demandas_col[i_col] < largo_barco:
            return False
        for i in range(largo_barco):
            if demandas_fil[i_fil + i] < 1:
                return False
    # Verificar que no haya barcos adyacentes o superpuestos
    for i in range(largo_barco):
        fil, col = (i_fil, i_col + i) if es_horizontal else (i_fil + i, i_col)
        # Revisa que el casillero no este ocupado por otro barco.
        if tablero[fil][col] != 0:
            return False
        # Revisa adyacencias (por fila, columna y diagonales)
        for dx, dy in [
            (-1, -1),
            (-1, 0),
            (-1, 1),
            (0, -1),
            (0, 1),
            (1, -1),
            (1, 0),
            (1, 1),
        ]:
            adj_fil, adj_col = fil + dx, col + dy
            if 0 <= adj_fil < n and 0 <= adj_col < m and tablero[adj_fil][adj_col]
    != 0:
                return False
    # Verificar extremos del barco
    if es_horizontal:
        if i_col > 0 and tablero[i_fil][i_col - 1] != 0:
            return False
        if i_col + largo_barco < m and tablero[i_fil][i_col + largo_barco] != 0:
            return False
    else:  # Es vertical
        if i_fil > 0 and tablero[i_fil - 1][i_col] != 0:
            return False
        if i_fil + largo_barco < n and tablero[i_fil + largo_barco][i_col] != 0:
            return False

    return True


# Ubica al barco en el tablero, de forma horizontal o vertical.
def ubicar_barco(
    tablero, largo_barco, i_fil, i_col, demandas_fil, demandas_col, es_horizontal
):
    if es_horizontal:
        for i in range(i_col, i_col + largo_barco):
            tablero[i_fil][i] = 1
        demandas_fil[i_fil] -= largo_barco
        for i in range(i_col, i_col + largo_barco):
            demandas_col[i] -= 1
    else:
        for i in range(i_fil, i_fil + largo_barco):
            tablero[i][i_col] = 1
        demandas_col[i_col] -= largo_barco
        for i in range(i_fil, i_fil + largo_barco):
            demandas_fil[i] -= 1


# Quita al barco del tablero.
def quitar_barco(
```

```
    tablero, largo_barco, i_fil, i_col, demandas_fil, demandas_col, es_horizontal
):
    if es_horizontal:
        for i in range(i_col, i_col + largo_barco):
            tablero[i_fil][i] = 0
        demandas_fil[i_fil] += largo_barco
        for i in range(i_col, i_col + largo_barco):
            demandas_col[i] += 1
    else:
        for i in range(i_fil, i_fil + largo_barco):
            tablero[i][i_col] = 0
        demandas_col[i_col] += largo_barco
        for i in range(i_fil, i_fil + largo_barco):
            demandas_fil[i] += 1
```

5. Resolución por Programación Lineal (ej 4)

El objetivo es desarrollar un modelo de programación lineal que resuelva el problema de la batalla naval de forma óptima, es decir, posicionando todos los barcos y cumpliendo con todos los requisitos de las filas y columnas.

Defino algunas variables a utilizar en las restricciones:

- **n**: cantidad de filas.
- **m**: cantidad de columnas.
- **x**: cantidad de barcos.
- B_x: largo del barco x.

5.1. Restricciones

5.1.1. Posición inicial del barco

Defino $s_{x,i,j}$ como una variable booleana que determina si el barco x comienza en la posición (i, j). Un barco solo puede empezar en un único casillero, por lo que $s_{x,i,j}$ debe cumplir lo siguiente:

$$\sum_{i=0}^{n-B_x} \sum_{j=0}^{m-B_x} s_{x,i,j} = 1 \quad \forall x$$

La razón por la que se resta B_x a la cantidad de filas y columnas, es porque los barcos solo van a poder çrecer", partiendo desde la posición inicial, hacia la derecha o hacia abajo, por lo que hay casilleros en los que un barco de longitud Bx nunca podría empezar.

5.1.2. El barco se encuentra posicionado

Defino $b_{x,i,j}$ como una variable booleana que indica si el casillero (i, j) está ocupado por el barco x.

Para todo barco x se debe cumplir que ocupa una cantidad B_x de casilleros:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} b_{x,i,j} = B_x - 1 \quad \forall x$$

Obs: el -1 es porque el casillero inicial del barco se modela con la variable $s_{x,i,j}$.

5.1.3. Posicionamiento correcto del barco

Defino h_x como una variable booleana que si vale:

- **1**: el barco x está posicionado horizontalmente.
- **0**: el barco x está posicionado verticalmente.

Luego, para cada barco se debe cumplir lo siguiente:

Posicionamiento Horizontal:

$$s_{x,i,j} + \sum_{col=j+1}^{j+B_x-1} b_{x,i,col} \geq B_x \cdot h_x - M \cdot (1 - s_{x,i,j})$$

Posicionamiento Vertical:

$$s_{x,i,j} + \sum_{fil=i+1}^{i+B_x-1} b_{x,fil,j} \geq B_x \cdot (1 - h_x) - M \cdot (1 - s_{x,i,j})$$

La suma de los siguientes B_x casilleros, partiendo desde el inicial, tiene que dar B_x. La variable h_x se usa para que esta suma sea requerida solo horizontal o verticalmente. Para el caso opuesto, la suma tiene que ser mayor o igual a 0, lo cual va a ser valido siempre.

Además, el análisis de la suma solo nos interesa si estamos evaluando el casillero donde inicia el barco (es decir, $s_{x,i,j}$ es igual a 1). Para todo el resto de casos, no nos interesa evaluarla. Para esto se usa el Big M. Si estamos evaluando el casillero inicial, el Big M se anula, pero si no estamos evaluandolo, el Big M nos garantiza que esas inecuaciones sean siempre válidas.

Entonces, si $s_{x,i,j}$ es igual a 1, nos quedan las siguientes inecuaciones:

Posicionamiento Horizontal:

$$s_{x,i,j} + \sum_{col=j+1}^{j+B_x-1} b_{x,i,col} \geq B_x \cdot h_x$$

Posicionamiento Vertical:

$$s_{x,i,j} + \sum_{fil=i+1}^{i+B_x-1} b_{x,fil,j} \geq B_x \cdot (1 - h_x)$$

Esto, junto a la restricción 3.1.1 y 3.1.2 nos asegura que los únicos casilleros ocupados por el barco son los $B_x - 1$ horizontales o verticales respecto del casillero inicial.

5.1.4. Restricciones de casillero

Defino $c_{i,j}$ como una variable booleana que indica que el casillero (i,j) se encuentra ocupado.

El casillero solo puede ser ocupado por un barco (no se permite la superposición, por lo que se debe cumplir la siguente inecuación:

$$c_{i,j} = \sum_{x=0}^{k-1} (s_{x,i,j} + b_{x,i,j}) \leq 1$$

5.1.5. Requisito consumo de filas y columnas

: Sean F_i y C_j el consumo de la fila i y la columna j respectivamente:

$$\sum_{j=0}^{m-1} c_{i,j} = F_i \quad \forall i \in \{0, 1, \ldots, n-1\}$$

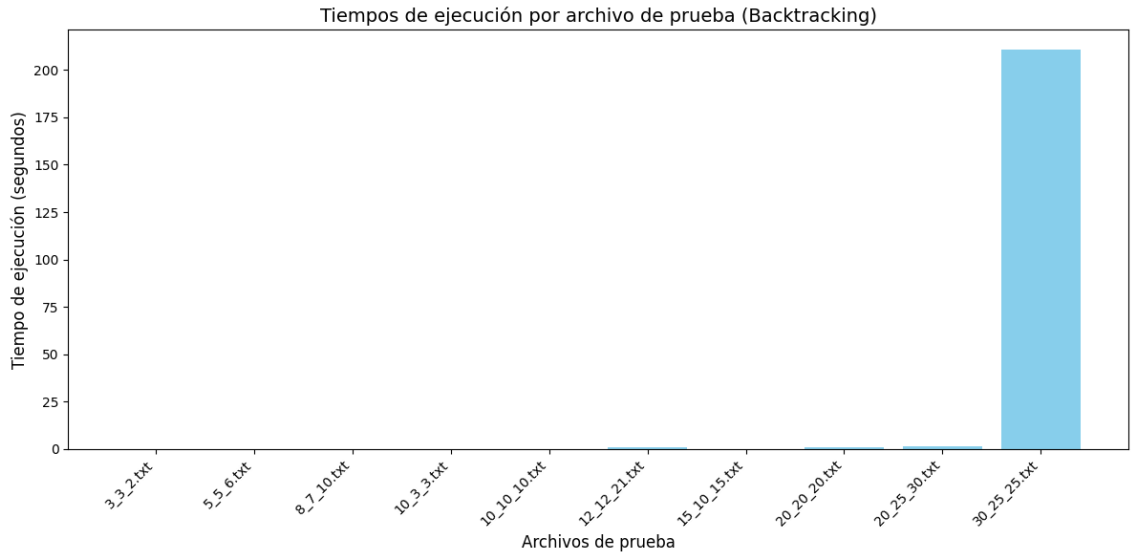
$$\sum_{i=0}^{n-1} c_{i,j} = C_j \quad \forall j \in \{0, 1, \ldots, m-1\}$$

5.1.6. No-adyacencia de los barcos

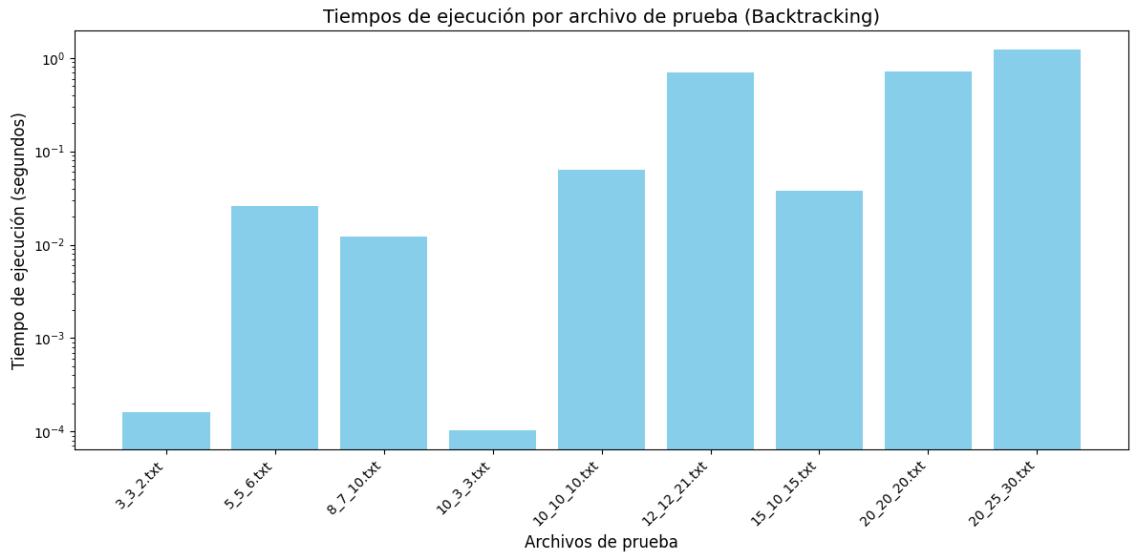
Para todo casillero (i,j) se debe cumplir que, de estar ocupado por un barco, sus adyacentes (si los hay), pertenecen al mismo barco y no a otro.

6. Tiempos de ejecución (BT vs PL)

6.1. Backtracking



Si no tenemos en cuenta el tiempo de 30_25_25.txt, podémos ver que el tiempo de ejecución es bastante similar entre las diferentes pruebas:



La razón por la que se dispara tanto el tiempo de la prueba 30_25_25.txt es que no solo el tablero es grande (30 x 25), sino que además las restricciones son de requisitos muy altos y con muchos barcos, por lo que las posibles combinaciones válidas a probar son muy elevadas (y las podas son menos frecuentes). Esto en contraste con un test como 20_25_30.txt donde, si bien el tablero es grande, las restricciones son más chicas por lo que las podas son más frecuentes.

6.2. Programación Lineal

7. Algoritmo de aproximación (ej 5)

7.1. Solución

En esta sección se propone un algoritmo de aproximación Greedy para resolver el problema de La Batalla Naval Individual. Tal como se propone en el enunciado, el mismo consiste de tomar las columnas o filas con mayor demanda, e intentar colocar los barcos del más largo al más corto, mientras entren en esas demandas.

Por su naturaleza, sabemos que este algoritmo no nos dará la solución óptima a este problema. Sin embargo, puede darnos una aproximación que quizás consideremos buena si tenemos en cuenta que el tiempo de ejecución de un algoritmo greedy es mucho menor al de un algoritmo que sí entregue la solución óptima, como backtracking o programación lineal.

El algoritmo propuesto es el siguiente:

```
# Ubica los barcos en el tablero, eligiendo la fila/columna mas demandada y eligiendo el barco mas grande posible que cubra esa demanda.
# Recibe la lista de longitud de barcos, y las listas de demandas de filas y columnas.
def batalla_naval(largo_barcos, demandas_fil, demandas_col):
    n = len(demandas_fil)
    m = len(demandas_col)
    tablero = [[0] * m for _ in range(n)]
    largo_barcos.sort(reverse=True)

    for barco in largo_barcos:
        ubicado = False

        while not ubicado:
            max_demanda_fil = max((d, i) for i, d in enumerate(demandas_fil))
            max_demanda_col = max((d, j) for j, d in enumerate(demandas_col))

            # El barco es mas grande que la demanda
            if max_demanda_fil[0] < barco and max_demanda_col[0] < barco:
                break

            if max_demanda_fil[0] >= max_demanda_col[0]:
                # Intenta ubicar por fila
                i_fil = max_demanda_fil[1]
                for i_col in range(m):
                    #if intentar_ubicar_barco(tablero, m, n, barco, i_fil, i_col, True):
                    if intentar_ubicar_barco(tablero, barco, i_fil, i_col, True, demandas_fil, demandas_col):
                        ubicar_barco(tablero, barco, i_fil, i_col, demandas_fil, demandas_col, True)
                        ubicado = True
                        break
            else:
                # Intenta ubicar por columna
                i_col = max_demanda_col[1]
                for i_fil in range(n):
                    #if intentar_ubicar_barco(tablero, m, n, barco, i_fil, i_col, False):
                    if intentar_ubicar_barco(tablero, barco, i_fil, i_col, False, demandas_fil, demandas_col):
                        ubicar_barco(tablero, barco, i_fil, i_col, demandas_fil, demandas_col, False)
                        ubicado = True
                        break

            # No se puede ubicar al barco
            if not ubicado:
                break

    return tablero, demandas_fil, demandas_col

# Intenta ubicar el barco, horizontalmente en la fila i_fil, o verticalmente en la
```

```
    columna i_col (segun el caso).
# Chequea que no se salga del tablero, que los casilleros a tomar no esten siendo utilizados por otro barco y que no tenga barcos adyacentes.
def intentar_ubicar_barco(
    tablero, largo_barco, i_fil, i_col, es_horizontal, demandas_fil, demandas_col
):
    n, m = len(tablero), len(tablero[0])

    # Verificar que el barco no salga del tablero
    if es_horizontal:
        if i_col + largo_barco > m:
            return False
    else:  # Es vertical
        if i_fil + largo_barco > n:
            return False
    # Verificar que no se violen las demandas de fila y columna
    if es_horizontal:
        if demandas_fil[i_fil] < largo_barco:
            return False
        for i in range(largo_barco):
            if demandas_col[i_col + i] < 1:
                return False
    else:  # Es vertical
        if demandas_col[i_col] < largo_barco:
            return False
        for i in range(largo_barco):
            if demandas_fil[i_fil + i] < 1:
                return False
    # Verificar que no haya barcos adyacentes o superpuestos
    for i in range(largo_barco):
        fil, col = (i_fil, i_col + i) if es_horizontal else (i_fil + i, i_col)
        # Revisa que el casillero no este ocupado por otro barco.
        if tablero[fil][col] != 0:
            return False
        # Revisa adyacencias (por fila, columna y diagonales)
        for dx, dy in [
            (-1, -1),
            (-1, 0),
            (-1, 1),
            (0, -1),
            (0, 1),
            (1, -1),
            (1, 0),
            (1, 1),
        ]:
            adj_fil, adj_col = fil + dx, col + dy
            if 0 <= adj_fil < n and 0 <= adj_col < m and tablero[adj_fil][adj_col] != 0:
                return False
    # Verificar extremos del barco
    if es_horizontal:
        if i_col > 0 and tablero[i_fil][i_col - 1] != 0:
            return False
        if i_col + largo_barco < m and tablero[i_fil][i_col + largo_barco] != 0:
            return False
    else:  # Es vertical
        if i_fil > 0 and tablero[i_fil - 1][i_col] != 0:
            return False
        if i_fil + largo_barco < n and tablero[i_fil + largo_barco][i_col] != 0:
            return False

    return True


# Ubica al barco en el tablero, de forma horizontal o vertical.
def ubicar_barco(tablero, largo_barco, i_fil, i_col, demandas_fil, demandas_col, es_horizontal):
    if es_horizontal:
        for i in range(i_col, i_col + largo_barco):
            tablero[i_fil][i] = 1
```

```
112         demandas_fil[i_fil] -= largo_barco
113         for i in range(i_col, i_col + largo_barco):
114             demandas_col[i] -= 1
115     else:
116         for i in range(i_fil, i_fil + largo_barco):
117             tablero[i][i_col] = 1
118         demandas_col[i_col] -= largo_barco
119         for i in range(i_fil, i_fil + largo_barco):
120          demandas_fil[i] -= 1
```

7.2. Análisis de complejidad

El análisis de complejidad para el algoritmo implementado se puede expresar en función de las variables del problema: las dimensiones del tablero n y m, y la cantidad de barcos b:

- Ordenamiento del vector de barcos (para tenerlos de mayor a menor largo): $\mathcal{O}(b \cdot log(b))$.
- Para cada barco tendremos una iteración donde:
  - Se obtiene la mayor demanda de fila ($\mathcal{O}(n)$) o de columna ($\mathcal{O}(m)$), dando una complejidad de $\mathcal{O}(n+m)$. Como es para cada barco, el costo total es de $\mathcal{O}(b \cdot (n+m))$.
  - Se intenta ubicar al barco en el tablero (vertical u horizontalmente), que en el peor caso recorrería toda la fila o columna candidata ($\mathcal{O}(n+m)$). Como cada barco puede tener un largo, llamémoslo l, el costo es de $\mathcal{O}(l \cdot (n+m))$. Y de nuevo, como esto se hace para cada barco, el costo total de intentar ubicar un barco en el tablero es de $\mathcal{O}(b \cdot l \cdot (n+m))$.
- Hay costos que pueden ser ignorados como el de buscar adyacencias ($\mathcal{O}(1)$), y el de ubicar al barco ($\mathcal{O}(l)$).

Por lo tanto, podemos concluir que la complejidad del algoritmo de aproximación implementado es de $\mathcal{O}(b \cdot l \cdot (n+m))$, donde b es el largo del vector de barcos, n y m las dimensiones del tablero, y l la longitud de cada barco.

7.3. Análisis de la aproximación propuesta

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 $\frac{A(I)}{z(I)} \leq r(A)$ como una cota para todas las instancias posibles.

A continuación, vamos a calcular el valor de $r(A)$ para los distintos sets de datos propuestos por la cátedra (que corresponden a soluciones óptimas).

- 3_3_2.txt
  - OPT - Demanda cumplida: $z(I) = 4$
  - APROX - Demanda cumplida: $A(I) = 4$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 5_5_6.txt
  - OPT - Demanda cumplida: $z(I) = 12$
  - APROX - Demanda cumplida: $A(I) = 12$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 8_7_10.txt
  - OPT - Demanda cumplida: $z(I) = 26$

- APROX - Demanda cumplida: $A(I) = 24$
- Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,9230769230769231$

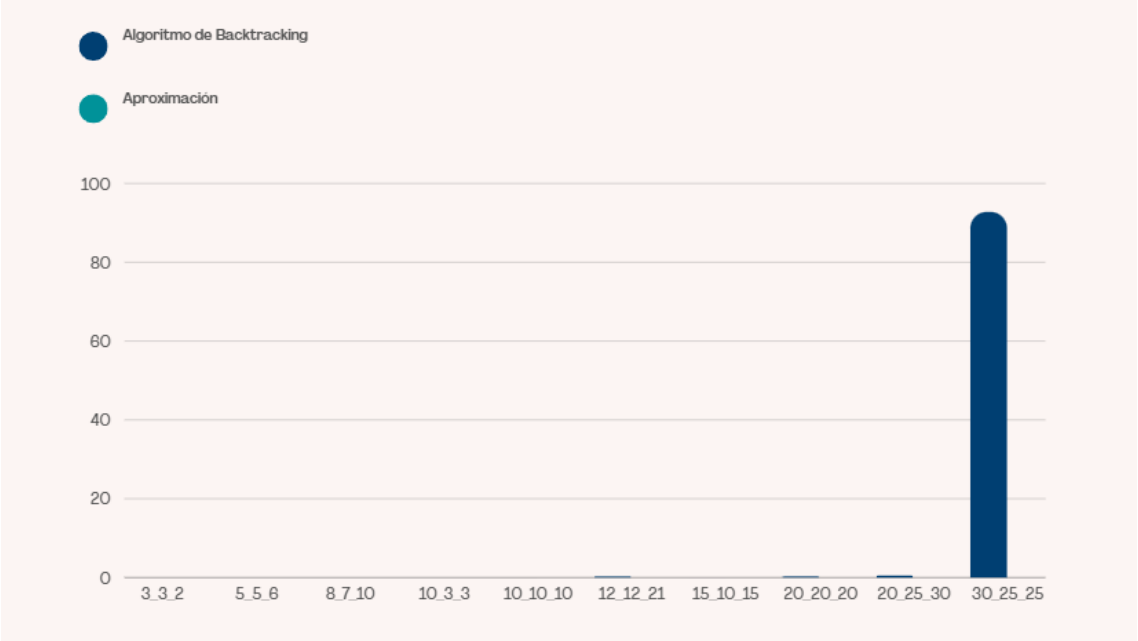
- 10_3_3.txt
  - OPT - Demanda cumplida: $z(I) = 6$
  - APROX - Demanda cumplida: $A(I) = 6$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 10_10_10.txt
  - OPT - Demanda cumplida: $z(I) = 40$
  - APROX - Demanda cumplida: $A(I) = 36$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,9$
- 12_12_21.txt
  - OPT - Demanda cumplida: $z(I) = 46$
  - APROX - Demanda cumplida: $A(I) = 20$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,4347826086956522$
- 15_10_15.txt
  - OPT - Demanda cumplida: $z(I) = 40$
  - APROX - Demanda cumplida: $A(I) = 30$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,75$
- 20_20_20.txt
  - OPT - Demanda cumplida: $z(I) = 104$
  - APROX - Demanda cumplida: $A(I) = 80$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,7692307692307692$
- 20_25_30.txt
  - OPT - Demanda cumplida: $z(I) = 172$
  - APROX - Demanda cumplida: $A(I) = 112$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,6511627906976744$
- 30_25_25.txt
  - OPT - Demanda cumplida: $z(I) = 202$
  - APROX - Demanda cumplida: $A(I) = 94$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,4653465346534653$

Podemos ver que, para estos sets de datos, la cota $r(A)$ adecuada es la más chica, que corresponde al caso del archivo 12_12_21.txt. Por lo tanto, $r(A) = 0,4347826086956522$.

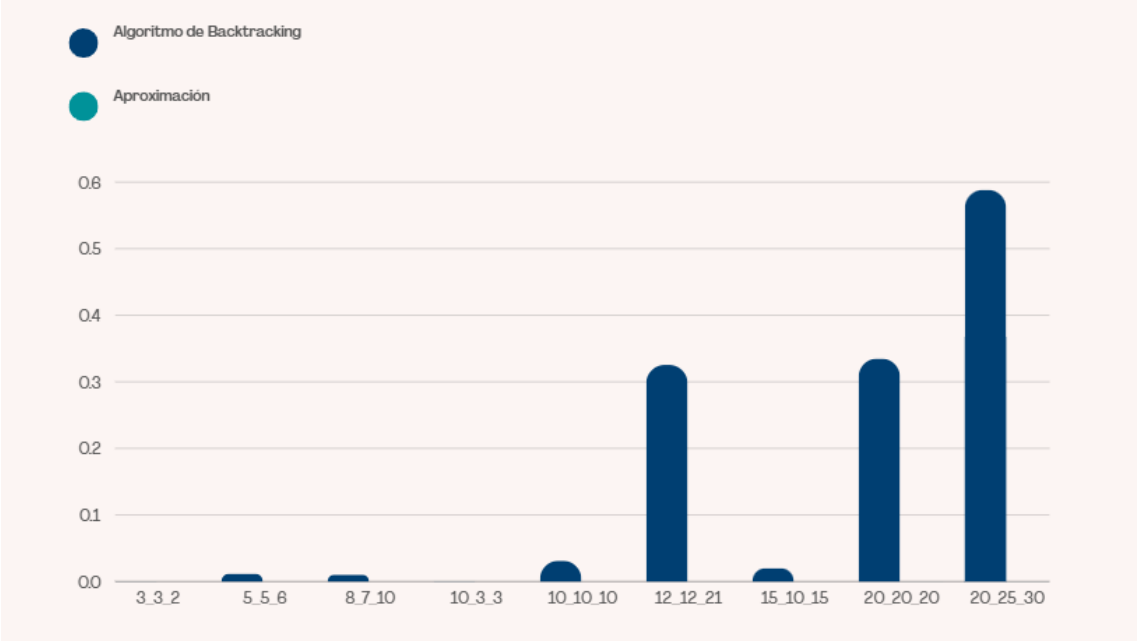
7.4. Mediciones de tiempos

A continuación vamos a comparar las mediciones de tiempos de este algoritmo de aproximación con el algoritmo de backtracking que vimos anteriormente, principalmente para ver si se justifica en tiempo calcular una solución aproximada que no es óptima.

Estos son los resultados para todos los casos de prueba:



Aquí exceptuamos el caso 30_25_25.txt para ver mejor los resultados:



Podemos observar que los tiempos de la aproximación son tan pequeños que ni siquiera llegan a distinguirse en el gráfico. Los adjuntamos en la siguiente tabla:

Prueba	Backtracking	Aproximación
3_3_2	0.00040	0.00001
5_5_6	0.01183	0.00003
8_7_10	0.01030	0.00006
10_3_3	0.00063	0.00003
10_10_10	0.03127	0.00008
12_12_21	0.32543	0.00013
15_10_15	0.02009	0.00009
20_20_20	0.33491	0.00027
20_25_30	0.58847	0.00032
30_25_25	92.78526	0.00036

Queda en claro que la aproximación es extremadamente más rápida que el algoritmo de backtracking. Vemos que esta tendencia se acentúa todavía más cuando la dificultad de la prueba incrementa, teniendo una diferencia de varios órdenes de magnitud.

8. Aproximación alternativa (ej 6)

8.1. Solución

En este apartado vamos a explorar una aproximación distinta a la que se propone en el enunciado del trabajo práctico. En este caso, se va a aplicar una función que calcule el impacto de colocar un barco en una posición determinada, penalizándolo si el impacto es muy grande.

Lo que se busca es colocar a cada barco en la posición que minimice su penalización, lo que equivale a minimizar los espacios residuales para evitar que queden celdas sin usar en ubicaciones que puedan ser útiles para barcos futuros. Se podría pensar que es algo similar a lo que se hacía en el algortimo de scheduling de charlas: primero se colocaban las charlas que terminaban primero para dejar espacio para las demás, mientras que ahora colocamos los barcos de forma que queden menos espacios residuales y los demás barcos tengan más posibilidades para ubicarse.

El algoritmo de esta solución es el siguiente:

```
# Ubica los barcos en el tablero, de forma tal que se minimice el impacto o penalizacion de colocarlos en cada posicion.
# Recibe la lista de longitud de barcos, y las listas de demandas de filas y columnas.
def batalla_naval(largo_barcos, demandas_fil, demandas_col):
    n = len(demandas_fil)
    m = len(demandas_col)
    tablero = [[0] * m for _ in range(n)]
    largo_barcos.sort(reverse=True)  # Ordenar barcos de mayor a menor longitud

    for barco in largo_barcos:
        mejor_penalizacion = float('inf')
        mejor_posicion = None

        # Evaluar todas las posibles ubicaciones
        for i_fil in range(n):
            for i_col in range(m):
                # Evaluar penalizacion para posicion horizontal
                if intentar_ubicar_barco(tablero, barco, i_fil, i_col, True, demandas_fil, demandas_col):
                    penalizacion = calcular_penalizacion_residual(tablero, barco, i_fil, i_col, True)
                    if penalizacion < mejor_penalizacion:
                        mejor_penalizacion = penalizacion
                        mejor_posicion = (i_fil, i_col, True)

                # Evaluar penalizacion para posicion vertical
                if intentar_ubicar_barco(tablero, barco, i_fil, i_col, False, demandas_fil, demandas_col):
                    penalizacion = calcular_penalizacion_residual(tablero, barco, i_fil, i_col, False)
                    if penalizacion < mejor_penalizacion:
                        mejor_penalizacion = penalizacion
                        mejor_posicion = (i_fil, i_col, False)

        # Ubicar el barco en la mejor posicion encontrada
        if mejor_posicion:
            i_fil, i_col, es_horizontal = mejor_posicion
            ubicar_barco(tablero, barco, i_fil, i_col, demandas_fil, demandas_col, es_horizontal)

    return tablero, demandas_fil, demandas_col

# Calcula una penalizacion o el impacto de colocar un barco en una posicion particular.
def calcular_penalizacion_residual(tablero, largo_barco, i_fil, i_col, es_horizontal):
    n, m = len(tablero), len(tablero[0])
    penalizacion = 0

    # Calcular penalizacion antes del barco
```

```
43      if es_horizontal:
44          penalizacion += sum(1 for j in range(max(0, i_col - 1), i_col) if tablero[
        i_fil][j] == 0)
45      else:
46          penalizacion += sum(1 for i in range(max(0, i_fil - 1), i_fil) if tablero[i
        ][i_col] == 0)
47
48      # Calcular penalizacion despues del barco
49      if es_horizontal:
50          penalizacion += sum(1 for j in range(i_col + largo_barco, min(m, i_col +
        largo_barco + 1)) if tablero[i_fil][j] == 0)
51      else:
52          penalizacion += sum(1 for i in range(i_fil + largo_barco, min(n, i_fil +
        largo_barco + 1)) if tablero[i][i_col] == 0)
53
54      return penalizacion
```

8.2. Análisis de complejidad

El análisis de complejidad para esta aproximación es muy similar al que hicimos anteriormente. Se puede expresar en función de las variables del problema: las dimensiones del tablero n y m, y la cantidad de barcos b:

- Ordenamiento del vector de barcos (para tenerlos de mayor a menor largo): $\mathcal{O}(b \cdot log(b))$.
- Para cada barco tendremos una iteración donde:
  - Se evalúan todas las posibilidades para colocar un barco en el tablero, calculando sus penalizaciones para cada caso. Esto se realiza para todas las filas y columnas, y se hace dos veces por las orientaciones vertical y horizontal, y el costo de calcular las penalizaciones es proporcional a la longitud del barco, por lo que tiene un costo total de $\mathcal{O}(2 \cdot l \cdot n \cdot m) = \mathcal{O}(l \cdot n \cdot m)$. Como es para cada barco, el costo total es de $\mathcal{O}(b \cdot l \cdot n \cdot m)$.
- Hay costos que pueden ser ignorados como el de buscar adyacencias ($\mathcal{O}(1)$), y el de ubicar al barco ($\mathcal{O}(l)$).

Por lo tanto, podemos concluir que la complejidad de este algoritmo de aproximación alternativo implementado es de $\mathcal{O}(b \cdot l \cdot n \cdot m)$, donde b es el largo del vector de barcos, n y m las dimensiones del tablero, y l la longitud de cada barco.

8.3. Análisis de la aproximación alternativa propuesta

Volveremos a calcular el valor de $r(A)$ para los distintos sets de datos propuestos por la cátedra (que corresponden a soluciones óptimas) para esta nueva aproximación.

- 3_3_2.txt
  - OPT - Demanda cumplida: $z(I) = 4$
  - APROX - Demanda cumplida: $A(I) = 4$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 5_5_6.txt
  - OPT - Demanda cumplida: $z(I) = 12$
  - APROX - Demanda cumplida: $A(I) = 12$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 8_7_10.txt

- OPT - Demanda cumplida: $z(I) = 26$
- APROX - Demanda cumplida: $A(I) = 26$
- Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$

- 10_3_3.txt
  - OPT - Demanda cumplida: $z(I) = 6$
  - APROX - Demanda cumplida: $A(I) = 6$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 10_10_10.txt
  - OPT - Demanda cumplida: $z(I) = 40$
  - APROX - Demanda cumplida: $A(I) = 34$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,85$
- 12_12_21.txt
  - OPT - Demanda cumplida: $z(I) = 46$
  - APROX - Demanda cumplida: $A(I) = 40$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,8695652173913043$
- 15_10_15.txt
  - OPT - Demanda cumplida: $z(I) = 40$
  - APROX - Demanda cumplida: $A(I) = 40$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 20_20_20.txt
  - OPT - Demanda cumplida: $z(I) = 104$
  - APROX - Demanda cumplida: $A(I) = 94$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,9038461538461538$
- 20_25_30.txt
  - OPT - Demanda cumplida: $z(I) = 172$
  - APROX - Demanda cumplida: $A(I) = 172$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 1$
- 30_25_25.txt
  - OPT - Demanda cumplida: $z(I) = 202$
  - APROX - Demanda cumplida: $A(I) = 150$
  - Ratio: $r(A) = \frac{A(I)}{z(I)} = 0,7425742574257426$

Podemos ver que, para estos sets de datos, la cota $r(A)$ adecuada es la más chica, que corresponde al caso del archivo 30_25_25.txt. Por lo tanto, $r(A) = 0,7425742574257426$.

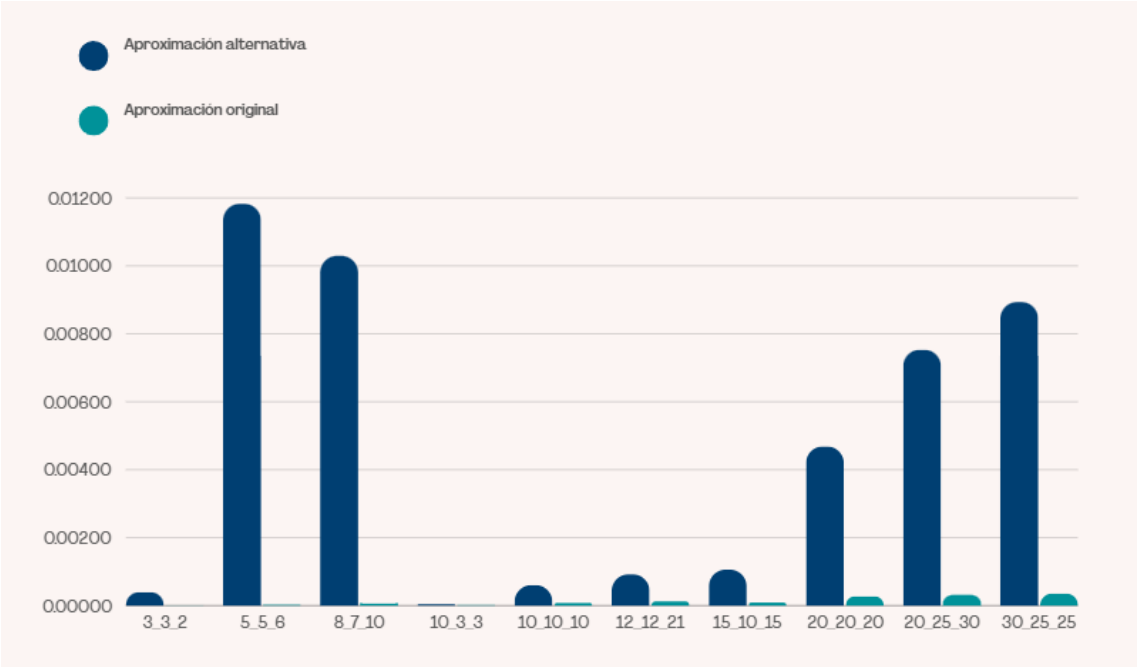
Además, se observa que esta aproximación se acerca mucho más a la solución óptima que la que vimos antes, exceptuando el caso de 10_10_10.txt. Incluso se llega a obtener la solución óptima en instancias difíciles como en 20_25_30.txt, lo cual es sorprendente. Queda en claro la importancia de la regla de aproximación que se utilice.

8.4. Mediciones de tiempos

A continuación vamos a comparar las mediciones de tiempos de este algoritmo de aproximación con el algoritmo de backtracking que vimos anteriormente, principalmente para ver si se justifica en tiempo calcular una solución aproximada que no es óptima.

Ahora vamos a comparar las mediciones de tiempos de este nuevo algoritmo de aproximación con el algoritmo de aproximación original que vimos anteriormente, para ver si la mejora en optimalidad se corresponde con un aumento en los tiempos de ejecución.

Estos son los resultados para todos los casos de prueba:



Se puede ver que los tiempos de ejecución de este nuevo algoritmo son mayores, lo cual tiene sentido por su mayor complejidad temporal y mejores resultados obtenidos. Los podemos ver en detalle en la siguiente tabla:

	Aproximación alternativa	Aproximación original
3_3_2	0.00040	0.00001
5_5_6	0.01183	0.00003
8_7_10	0.01030	0.00006
10_3_3	0.00004	0.00003
10_10_10	0.00060	0.00008
12_12_21	0.00092	0.00013
15_10_15	0.00106	0.00009
20_20_20	0.00469	0.00027
20_25_30	0.00753	0.00032
30_25_25	0.00894	0.00036

Las diferencias de tiempo son considerables, pero no son tan importantes en comparación al algoritmo de backtracking. En este caso, ambos algoritmos de aproximación tienen tiempos de ejecución muy pequeños.

9. Conclusiones

Como conclusión a este trabajo práctico, pudimos observar las ventajas y desventajas de distintas estrategias para resolver un problema NP-Completo como La Batalla Naval Individual. Primero abordamos metodologías como Backtracking y Programación Lineal que nos ayudaron a resolver el problema de forma óptima, pero que a su vez traen tiempos de ejecución muy altos a medida que aumenta la dificultad de la instancia del problema.

Luego fuimos por el camino de la aproximación, que no suele dar la solución óptima, pero permite obtener una solución válida aún cuando la dificultad del problema es muy alta. Esto puede ser muy útil en aquellos casos en los que una ejecución de Backtracking o Programación Lineal es exageradamente larga, y es allí cuando una aproximación, sobre todo si es buena, funciona muy bien como alternativa.