Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Reducciones, Backtracking, Programacion lineal, Aproximaciones

22 de noviembre de 2024

Valentín Schneider 107964
Ian von der Heyde 107638
Juan Martín de la Cruz 109588

Índice

1. Introducción

Los hermanos siguieron creciendo. Mateo también aprendió sobre programación dinámica, y cada uno aplicaba la lógica sabiendo que el otro también lo hacía. El juego de las monedas se tornó aburrido en cuánto notaron que siempre ganaba quien empezara, o según la suerte. Los años pasaron, llegó la adolescencia y empezaron a tener gustos diferentes. En general, jugaban a juegos individuales. En particular, Sophia estaba muy enganchada con un juego inventado en Argentina por Jaime Poniachik (uno de los fundadores de Ediciones de Mente) en 1982: La Batalla Naval Individual.

En dicho juego, tenemos un tablero de $n \times m$ 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. Ni más, ni menos. No podemos poner 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.

1.1. Consigna

Dado un tablero de $n \times m$ casilleros y una lista de k barcos (donde el barco i tiene longitud b_i), se cuenta con una lista de restricciones para las filas y otra para las columnas. En cada fila j, la restricción indica la cantidad de casilleros a ser ocupados en esa fila, y lo mismo para cada columna ℓ.

El problema consiste en determinar si es posible ubicar los barcos de tal manera que se cumplan las demandas de cada fila y columna, y las restricciones de ubicación.

1. Demostrar que el Problema de La Batalla Naval se encuentra en NP

Para demostrar que el Problema de La Batalla Naval se encuentra en NP, debemos mostrar que, dado un tablero de $n \times m$ casilleros, una lista de barcos de longitud b_i y restricciones de ocupación para filas y columnas, es posible verificar en tiempo polinómico si una disposición dada de barcos cumple con todas las restricciones.

2. Demostrar que el Problema de La Batalla Naval es NP-Completo

Para demostrar que el Problema de La Batalla Naval es NP-Completo, debemos realizar una reducción desde un problema conocido como NP-Completo, y así demostrar que el Problema de La Batalla Naval también es NP-Completo. Si se utiliza un problema no visto en clase, será necesario agregar una breve demostración de la NP-completitud del problema base.

3. Algoritmo de Backtracking para la Solución Óptima

Escribir un algoritmo que, utilizando **backtracking**, encuentre la solución óptima para la versión de optimización del problema. En esta versión, dado un tablero de $n \times m$ casilleros, una lista de k barcos (donde el barco i tiene longitud b_i), y listas de restricciones de ocupación para filas y columnas, se busca minimizar la cantidad de demanda incumplida.

Las consideraciones son las siguientes:

- No es obligatorio usar todos los barcos.
- Para cada fila o columna, si la demanda de casilleros ocupados es mayor que la ocupación real, la diferencia se cuenta como demanda incumplida.
- No se permite exceder la demanda de ocupación en ninguna fila ni columna.

Es necesario generar conjuntos de datos para verificar la corrección del algoritmo, además de medir tiempos de ejecución.

4. Modelo de Programación Lineal para la Solución Óptima

Desarrollar un modelo de programación lineal para resolver el problema de forma óptima. Se debe ejecutar el modelo para los mismos conjuntos de datos utilizados en el algoritmo de backtracking, medir tiempos de ejecución y compararlos con los tiempos obtenidos en el algoritmo de backtracking.

5. Algoritmo de Aproximación Propuesto por John Jellicoe

Se implementa un algoritmo de aproximación, propuesto por John Jellicoe, con el siguiente procedimiento:

1. Identificar la fila o columna con mayor demanda.
2. Colocar el barco de mayor longitud que aún no se haya utilizado en esa fila o columna, en una posición válida.
3. Si el barco de mayor longitud es demasiado largo para la demanda de esa fila o columna, omitirlo y continuar con el siguiente.
4. Repetir los pasos anteriores hasta que no queden barcos disponibles o no haya más demandas a cumplir.

Este algoritmo se considera como una aproximación para el problema de La Batalla Naval.

Analizar la complejidad del algoritmo y cuán buena es la aproximación. Para ello, sea I una instancia cualquiera del problema de La Batalla Naval, y sea $z(I)$ una solución óptima para dicha instancia. Definimos $A(I)$ como la solución aproximada, y se espera que $A(I)/z(I) \leq r(A)$ para todas las instancias posibles. Calcular $r(A)$ para este algoritmo, demostrando que la cota calculada es correcta.

Realizar mediciones comparativas entre el algoritmo exacto y la aproximación, tanto para conjuntos de datos manejables por el algoritmo exacto como para volúmenes de datos donde el algoritmo exacto ya no es viable, con el fin de verificar empíricamente la cota calculada.

6. (Opcional) Implementar una Alternativa de Aproximación

Como ejercicio opcional, implementar una aproximación alternativa (o un algoritmo greedy) que pueda ser de interés para resolver el problema. Comparar los resultados obtenidos con los de la aproximación de Jellicoe, y analizar su complejidad.

7. Conclusiones

Añadir cualquier conclusión relevante sobre el rendimiento, eficiencia y precisión de los algoritmos y aproximaciones, así como sobre la dificultad de resolución del problema en diferentes configuraciones de entrada.

2. Análisis del problema

Repasemos la consigna: Tenemos un tablero de $n \times m$ casilleros. En este tablero deben colocarse k barcos, cada uno de una longitud especificada. Además, se imponen restricciones sobre la cantidad de casilleros ocupados en cada fila y cada columna. El objetivo es determinar si existe una disposición de los barcos que cumpla con todas las restricciones dadas.

2.1. Variables y Notación

- n: Número de filas del tablero.
- m: Número de columnas del tablero.
- k: Número de barcos.
- b_i: Longitud del barco i, para $i = 1, 2, \ldots, k$.
- f_j: Restricción de la fila j, que indica la cantidad de casilleros que deben ser ocupados en esa fila, para $j = 1, 2, \ldots, n$.
- c_ℓ: Restricción de la columna ℓ, que indica la cantidad de casilleros que deben ser ocupados en esa columna, para $\ell = 1, 2, \ldots, m$.

2.2. Restricciones

Las restricciones que deben cumplirse para una solución válida son las siguientes:

R1 - Restricciones de posición de los barcos: Cada barco debe colocarse en una única fila o columna y ocupar exactamente b_i casilleros contiguos.

R2 - Restricciones de filas: Para cada fila j, el número total de casilleros ocupados debe ser igual a la restricción f_j.

R3 - Restricciones de columnas: Para cada columna ℓ, el número total de casilleros ocupados debe ser igual a la restricción c_ℓ.

R4 - Restricción de no superposición: Dos barcos no pueden ocupar el mismo casillero.

R5 - Restricción de no adyacencia: Ningún barco puede ser adyacente a otro barco, es decir, ningún casillero ocupado por un barco puede estar directamente al lado (horizontal, vertical o diagonal) de un casillero ocupado por otro barco.

2.3. Ejemplo del juego

Consideremos un ejemplo simple para ilustrar el problema:

Datos de entrada

- Tamaño del tablero: $n = 3$ filas, $m = 4$ columnas.
- Número de barcos: $k = 2$.
- Longitudes de los barcos:
$$b_1 = 2, \quad b_2 = 3$$
- Restricciones de las filas:
$$f_1 = 3, \quad f_2 = 1, \quad f_3 = 2$$
- Restricciones de las columnas:
$$c_1 = 1, \quad c_2 = 2, \quad c_3 = 2, \quad c_4 = 1$$

Posible solución

Una disposición posible para los barcos que cumple con todas las restricciones es la siguiente:

–	1	2	2	1
3	B	B	B	X
1	X	X	X	X
2	·	X	B	B

- En la primera fila, se coloca el barco b_2 de longitud 3, ocupando los primeros tres casilleros.
- En la tercera fila, el barco b_1 de longitud 2 ocupa dos casilleros.
- Esta disposición cumple con las restricciones de filas y columnas.
- Marcamos con una X aquellos casilleros que no pueden ser ocupados por algún otro barco dada la disposición actual de barcos.

Entonces, el problema consiste en encontrar una disposición de los barcos que cumpla con todas las restricciones de filas, columnas, adyacencias y no superposición. A lo largo de este documento analizaremos diferentes enfoques para su resolución

3. Batalla Naval es un problema NP

3.1. ¿Qué es un problema NP?

Un problema pertenece a la clase *NP* (nondeterministic polynomial time) si y solo si puede ser resuelto en tiempo polinómico por una máquina de Turing no determinista. En términos prácticos, esto significa que, dada una solución candidata para el problema, es posible verificar en tiempo polinómico si dicha solución es correcta.

Formalmente, un problema de decisión L pertenece a NP si existe una función verificadora $V(x, y)$ tal que:

- Para toda entrada $x \in L$, existe un certificado y (llamado prueba o testigo) tal que $V(x, y)$ devuelve *true*.
- La función verificadora $V(x, y)$ opera en tiempo polinómico con respecto al tamaño de la entrada x.

Para demostrar que el problema de la Batalla Naval está en NP, debemos verificar si cumple con las condiciones que definen a la clase de complejidad NP:

1. **¿Es una versión de decisión del problema?** La versión de decisión del problema de la Batalla Naval pregunta si existe una disposición de barcos en un tablero de tamaño $n \times m$ que cumpla con ciertas restricciones. Esto hace que el problema sea una versión de decisión, ya que se responde con ***si*** o ***no***.
2. **¿Puede verificarse una solución en tiempo polinómico?** Para que un problema esté en NP, debe ser posible verificar una solución candidata en tiempo polinómico en función del tamaño de la entrada.

3.2. Código de Validación

El siguiente código corresponde al validador que, dado un conjunto de posiciones para los barcos, verifica si cumplen las restricciones del problema.

```
from auxiliares import es_contiguo, configurar_posiciones_barcos, POSICIONES

def validador(restricciones_filas, restricciones_columnas, posiciones_barcos,
    barcos, demanda_cumplida, demanda_total):
    # demanda cumplida
    if sum(restricciones_filas) + sum(restricciones_columnas) != demanda_cumplida:
        return False

    # demanda total
    if demanda_cumplida != demanda_total:
        return False
```

```
    n = len(restricciones_filas)
    m = len(restricciones_columnas)
    tablero_solucion = [[None] * m for _ in range(n)]

    # cantidad de barcos
    if len(posiciones_barcos) != len(barcos):
        return False

    # contiguidad de barcos
    for set_barco in posiciones_barcos:
        if set_barco is None:
            continue
        if len(set_barco) > 1:
            if not es_contiguo(set_barco):
                return False

    # cantidad de posiciones que ocupa cada barco
    for i, set_barco in enumerate(posiciones_barcos):
        if set_barco is not None:
            if len(set_barco) != barcos[i]:
                return False

    for i, set_barco in enumerate(posiciones_barcos):
        if set_barco is not None:
            for (fila, col) in set_barco:
                tablero_solucion[fila][col] = i

    # restricciones filas
    for i in range(n):
        ocupados_en_fila = sum(1 for j in range(m) if tablero_solucion[i][j] is not None)
        if ocupados_en_fila != restricciones_filas[i]:
            return False

    # restricciones columnas
    for j in range(m):
        ocupados_en_col = sum(1 for i in range(n) if tablero_solucion[i][j] is not None)
        if ocupados_en_col != restricciones_columnas[j]:
            return False

    # restricciones adyacencias
    for i, set_barco in enumerate(posiciones_barcos):
        for (fila, col) in set_barco:
            for dx, dy in POSICIONES:
                ni, nj = fila + dx, col + dy
                if 0 <= ni < n and 0 <= nj < m:
                    if tablero_solucion[ni][nj] is not None and tablero_solucion[ni][nj] != i:
                        return False

    return True
```

Por su parte, se importa lo siguiente:

```
IZQ_ARRIBA = (-1, -1)
DER_ARRIBA = (-1, 1)
ARRIBA = (-1, 0)
ABAJO = (1, 0)
IZQ = (0, -1)
DER = (0, 1)
IZQ_ABAJO = (1, -1)
DER_ABAJO = (1, 1)
POSICIONES = [IZQ_ARRIBA, DER_ARRIBA, ARRIBA, ABAJO, IZQ, DER, IZQ_ABAJO, DER_ABAJO]

def es_contiguo(set_barco):
    posiciones = list(set_barco)
    visitados = set()
    pila = [posiciones[0]]
```

```
15
16      while pila:
17          (fila, col) = pila.pop()
18          if (fila, col) not in visitados:
19              visitados.add((fila, col))
20
21              # Revisamos las posiciones adyacentes
22              for dx, dy in [ARRIBA, IZQ, DER, ABAJO]:
23                  adyacente = (fila + dx, col + dy)
24                  if adyacente in set_barco and adyacente not in visitados:
25                      pila.append(adyacente)
26
27      return len(visitados) == len(set_barco)
```

3.3. Análisis de Complejidad

El análisis de la complejidad del algoritmo `validador` se desglosa en las siguientes etapas:

1. Verificación de la Demanda Cumplida y Total

```
1       # demanda cumplida
2       if sum(restricciones_filas) + sum(restricciones_columnas) != demanda_cumplida:
3           return False
4
5       # demanda total
6       if demanda_cumplida != demanda_total:
7           return False
```

Para calcular las sumas de f_j y c_ℓ, la complejidad es:

$$O(n) + O(m) = O(n + m).$$

2. Creación de `tablero_solucion`

```
1       n = len(restricciones_filas)
2       m = len(restricciones_columnas)
3       tablero_solucion = [[None] * m for _ in range(n)]
```

La creación de una matriz de dimensiones $n \times m$ tiene una complejidad de:

$$O(n \cdot m).$$

3. Verificación de Cantidad de Barcos

```
1       # cantidad de barcos
2       if len(posiciones_barcos) != len(barcos):
3           return False
```

Comparar las longitudes es una operación constante:

$$O(1).$$

4. Verificación de Contiguidad de Barcos

```
1       # contiguidad de barcos
2       for set_barco in posiciones_barcos:
3           if set_barco is None:
4               continue
5           if len(set_barco) > 1:
6               if not es_contiguo(set_barco):
```

Para cada barco en `posiciones_barcos`, el código llama a la función `es_contiguo`, cuya complejidad depende de la cantidad de posiciones que involucran a cada barco y es razonable asumir que es $O(b)$, donde b es la longitud del barco. Por lo tanto, nos queda que:

$$O\left(\sum_{i=1}^{k} b_i\right).$$

5. Verificación de Posiciones Ocupadas por Cada Barco

```
    for i, set_barco in enumerate(posiciones_barcos):
        if set_barco is not None:
            if len(set_barco) != barcos[i]:
                return False
```

Comparar las longitudes de los barcos para cada uno de los k barcos toma:

$$O(k).$$

6. Llenado del Tablero con las Posiciones de los Barcos

```
    for i, set_barco in enumerate(posiciones_barcos):
        if set_barco is not None:
            for (fila, col) in set_barco:
                tablero_solucion[fila][col] = i
```

Para cada barco, recorremos todas sus posiciones. Si la longitud promedio de los barcos es $\bar{b}$, el costo total es:

$$O\left(\sum_{i=1}^{k} b_i\right).$$

7. Verificación de Restricciones de Filas

```
    # restricciones filas
    for i in range(n):
        ocupados_en_fila = sum(1 for j in range(m) if tablero_solucion[i][j] is not
     None)
        if ocupados_en_fila != restricciones_filas[i]:
            return False
```

Para cada fila, iteramos sobre las m columnas. Esto da un costo total de:

$$O(n \cdot m).$$

8. Verificación de Restricciones de Columnas

```
    # restricciones columnas
    for j in range(m):
        ocupados_en_col = sum(1 for i in range(n) if tablero_solucion[i][j] is not
    None)
        if ocupados_en_col != restricciones_columnas[j]:
            return False
```

Para cada columna, iteramos sobre las n filas. Esto da un costo total de:

$$O(n \cdot m).$$

9. Verificación de Restricciones de Adyacencias

```
    # restricciones adyacencias
    for i, set_barco in enumerate(posiciones_barcos):
        for (fila, col) in set_barco:
            for dx, dy in POSICIONES:
                ni, nj = fila + dx, col + dy
                if 0 <= ni < n and 0 <= nj < m:
                    if tablero_solucion[ni][nj] is not None and tablero_solucion[ni][nj] != i:
                        return False
```

Para cada posición ocupada por un barco, verificamos las posiciones adyacentes. Si cada barco tiene longitud b_i, y el total de posiciones ocupadas por los k barcos es $\sum_{i=1}^{k} b_i$, entonces el costo total es:

$$O\left(\sum_{i=1}^{k} b_i\right).$$

Complejidad Total

Sumando todas las secciones, la complejidad total es:

$$O(n \cdot m) + O(k) + O\left(\sum_{i=1}^{k} b_i\right).$$

En el caso peor, todas las celdas del tablero están ocupadas por barcos, es decir:

$$\sum_{i=1}^{k} b_i \leq n \cdot m.$$

Por lo tanto, la complejidad total queda acotada por:

$$O(n \cdot m).$$

3.4. Si, pertenece a NP

Dado que cada paso del validador se realiza en tiempo polinómico en función del tamaño del tablero y del número de barcos y sus longitudes, precisamente en $O(n \cdot m + k \cdot b)$, este algoritmo permite verificar si una disposición de barcos propuesta cumple con las restricciones del problema. Por lo tanto, el Problema de la Batalla Naval pertenece a la clase NP, ya que es posible verificar en tiempo polinómico una solución candidata utilizando el validador proporcionado.

4. La Batalla Naval como un problema NP-Completo

4.1. ¿Qué es un problema de NP-completo?

Un problema de decisión se clasifica como *NP-completo* si cumple con las siguientes dos condiciones:

1. **Pertenece a NP:** El problema puede verificarse en tiempo polinómico dado un certificado válido.
2. **Es NP-duro:** Todo problema en NP puede ser reducido a este problema en tiempo polinómico.

La condición de ser *NP-duro* se establece mediante reducciones polinómicas. Dado un problema A y un problema B, decimos que A se reduce a B en tiempo polinómico (denotado como $A \leq_p B$)

si existe una función computable en tiempo polinómico que transforma cualquier instancia de A en una instancia de B tal que las respuestas sean equivalentes. Para probar que un problema es *NP-completo*, es común usar un problema ya conocido como *NP-completo* y demostrar la reducción polinómica hacia el problema objetivo.

Para demostrar que el problema de la Batalla Naval, en su versión de decisión, es NP-completo, procederemos con los siguientes pasos.

Primero, hemos establecido que el problema de la Batalla Naval pertenece a NP, ya que, dada una configuración de barcos en un tablero, podemos verificar en tiempo polinómico si esta disposición cumple con las restricciones de filas, columnas y de no adyacencia entre los barcos.

A continuación, para demostrar que el problema es NP-completo, buscamos un problema que ya sabemos que es NP-completo y reducimos dicho problema al problema de la Batalla Naval. Formalmente, debemos mostrar que existe una reducción polinómica desde un problema NP-completo hacia el problema de la Batalla Naval, es decir, que:

$$\text{NP-completo} \leq_p \text{Batalla Naval}$$

Si logramos realizar esta reducción, entonces, por transitividad, podemos concluir que el problema de la Batalla Naval es NP-completo.

4.2. Problema de Bin Packing

El Problema de Bin Packing en código unario enuncia:

- Un conjunto de números $S = \{s_1, s_2, \ldots, s_n\}$, cada uno expresado en código unario.
- Una cantidad de bins B, expresada en código unario.
- La capacidad de cada bin C, expresada en código unario.

La condición es que la suma total de los elementos en S sea igual a $C \times B$. Debemos decidir si es posible dividir el conjunto S en exactamente B subconjuntos disjuntos, donde la suma de elementos en cada subconjunto (o 'bin') sea igual a C.

Este problema es **NP-Completo**, ya que la versión del problema en la que los valores están en código unario no reduce su complejidad; simplemente amplía el tamaño de la entrada debido a la representación, pero la dificultad de encontrar una solución persiste.

4.2.1. Ejemplo de Bin Packing

Sea $S = \{3, 3, 3, 3\}$, con $B = 2$ y $C = 6$, expresados en código unario:

- $S = \{111, 111, 111, 111\}$
- $B = 11$
- $C = 111111$

Queremos dividir el conjunto S en $B = 2$ bins de capacidad $C = 6$ cada uno.

La suma total de los elementos en S es:

$$\sum_{i=1}^{4} s_i = 3 + 3 + 3 + 3 = 12$$

y $C \times B = 6 \times 2 = 12$, por lo que la condición de suma total se cumple.

Una posible partición es:

$$\{111, 111\} \text{ y } \{111, 111\}$$

donde cada subconjunto tiene una suma igual a $C = 6$, por lo que existe una solución.

Pregunta de Bin Packing: ¿Es posible distribuir los elementos $a_1, a_2, \ldots, a_n$ en B subconjuntos (contenedores) de modo que la suma de los elementos en cada contenedor sea exactamente C?

4.3. Bin Packing pertenece a NP-completo

El problema de *Bin Packing*, en su versión de decisión, pertenece a la clase NP-completo. Para demostrarlo, debemos mostrar que:

1. El problema está en NP.
2. Existe una reducción polinómica desde un problema NP-completo conocido hacia *Bin Packing*.

Primero, notamos que *Bin Packing* está en NP porque, dada una solución (una asignación de elementos a contenedores), podemos verificar en tiempo polinómico si cumple con las restricciones del problema, es decir, que el peso total de los elementos en cada contenedor no exceda la capacidad máxima y que todos los elementos estén asignados exactamente a un contenedor.

Ahora, para probar que *Bin Packing* es NP-completo, reducimos el problema *2-Partition*, que es un problema NP-completo conocido, al problema *Bin Packing*.

4.3.1. Reducción desde 2-Partition a Bin Packing

El problema *2-Partition* se define como sigue: dado un conjunto de enteros positivos $S = \{a_1, a_2, \ldots, a_n\}$, determinar si es posible particionarlo en dos subconjuntos disjuntos S_1 y S_2 tal que la suma de los elementos en S_1 sea igual a la suma de los elementos en S_2.

Sea la suma total de los elementos de S igual a $T = \sum_{i=1}^{n} a_i$. Si T es impar, no existe una partición válida, y el problema retorna `No`. Si T es par, buscamos particionar S en subconjuntos donde cada uno tenga una suma igual a $T/2$.

Para reducir *2-Partition* a *Bin Packing*, construimos una instancia de *Bin Packing* de la siguiente manera:

1. Dado el conjunto $S = \{a_1, a_2, \ldots, a_n\}$, asignamos cada entero a_i como el peso de un objeto.
2. Establecemos la capacidad de los contenedores como $T/2$
3. Determinamos si es posible empaquetar los elementos de S en exactamente 2 contenedores, de modo que la suma de los pesos en cada contenedor no exceda $T/2$.

La solución del problema *Bin Packing* responde directamente al problema *2-Partition*:

- Si los elementos pueden ser empaquetados en 2 contenedores con capacidad $T/2$, entonces existe una partición válida de S en dos subconjuntos con suma igual.
- En caso contrario, no existe una partición válida.

4.4. Reducción de Bin Packing a Batalla Naval

Para demostrar que el problema de *Batalla Naval* es NP-completo, reducimos el problema de *Bin Packing* a una instancia de *Batalla Naval*. La variante de *Bin Packing* utilizada en esta reducción consiste en decidir si es posible particionar un conjunto de n números en B subconjuntos (contenedores o *bins*) de modo que la suma de los números en cada subconjunto sea exactamente C.

4.4.1. Ejemplo de Bin Packing

Dada la instancia de entrada:

- Conjunto $\{1, 2, 3\}$,
- Número de *bins* $B = 2$,
- Capacidad de cada *bin* $C = 3$,

una posible solución consiste en particionar el conjunto en los subconjuntos:

$$\text{Subconjunto 1: } \{1, 2\}, \quad \text{Subconjunto 2: } \{3\}.$$

4.4.2. Construcción de la instancia de Batalla Naval

Para reducir esta instancia de *Bin Packing* a una instancia de *Batalla Naval*, construimos un tablero con las siguientes dimensiones:

- *Ancho:* $B \times 2$,
- *Altura:* $S + n$,

donde n es el número de elementos del conjunto que queremos particionar, y S es la suma total de los elementos del conjunto.

Restricciones de las columnas

Cada par de columnas en el tablero representa un *bin*. Para cada par:

- Una columna tendrá un valor objetivo de 0,
- La otra columna tendrá un valor objetivo de C.

El objetivo es que el número de piezas de barcos en estas columnas sume exactamente a los valores objetivos indicados.

Restricciones de las filas

Para cada número a_i del conjunto que queremos particionar:

- Insertamos una fila separadora con un valor objetivo de 0.
- Agregamos a_i filas consecutivas con un valor objetivo de 1.

El valor objetivo de cada fila indica cuántas piezas de barcos deben estar presentes en esa fila. Para las filas asociadas a un número a_i, requerimos que haya exactamente una pieza de barco en cada fila, lo que garantiza que cada número sea cubierto por un barco de longitud a_i.

Cobertura de columnas y filas

En un par de columnas asociado a un *bin*, podemos:

- Cubrir las filas correspondientes a los números del conjunto con piezas de barcos, o
- Dejarlas vacías.

Sin embargo, la suma total de piezas de barcos en cada columna debe ser exactamente igual al valor objetivo de esa columna, es decir, 0 o C.

4.4.3. Ejemplo Resuelto

Consideremos la instancia de *Bin Packing* con:

- Conjunto $\{1, 2, 3\}$,
- $B = 2$,
- $C = 3$.

El tablero correspondiente tendrá:

- *Ancho:* $B \times 2 = 4$,
- *Altura:* $S + n = 3 + \sum_{i=1}^{n} a_i = 3 + (1 + 2 + 3) = 9$.

Las filas y columnas se construyen siguiendo las reglas descritas anteriormente. Por ejemplo:

1. Para cada par de columnas en el tablero, una columna tendrá un valor objetivo 0 y la siguiente tendrá un valor objetivo de $C = 3$
2. Para cada número a_i del conjunto que queremos particionar, insertamos una fila separadora con un valor objetivo de 0 y luego agregamos a_i filas consecutivas con un valor objetivo de 1.

Fila	**0**	**3**	**0**	**3**
0				
1		X		
0				
1		X		
0		X		
1				X
1				X
1				X

Las casillas sombreadas en el tablero resultante representan las piezas de los barcos, que cumplen con las restricciones del problema de *Bin Packing* al satisfacer los valores objetivos de cada columna.

Esta construcción se realiza en tiempo polinómico con respecto al tamaño de la entrada. Además, cualquier solución válida de *Batalla Naval* en el tablero construido corresponde directamente a una solución válida de la instancia de *Bin Packing*, y viceversa. Por lo tanto, hemos demostrado que *Bin Packing* se reduce polinómicamente a *Batalla Naval*.

4.5. Demostración de NP-Completitud

A continuación, establecemos que una solución al problema de *Bin Packing* transformado es una solución válida para el problema de la Batalla Naval si y solo si cumple con todas las restricciones de ocupación y no adyacencia de este último. Formalizamos esta equivalencia en dos direcciones, demostrando que:

4.5.1. Si existe una solución para Bin Packing, entonces existe una solución para Batalla Naval.

Supongamos que existe una partición del conjunto $\{a_1, a_2, \ldots, a_n\}$ en B subconjuntos tal que la suma de los elementos de cada subconjunto es exactamente C. Es decir, para cada bin k, se cumple que:

$$\sum_{a_i \in \text{bin } k} a_i = C$$

A partir de esta partición, construimos un tablero de Batalla Naval como se describe en la reducción. En este tablero, cada número a_i es representado por un barco de longitud a_i, y cada par de columnas en el tablero representa un bin.

La construcción del tablero garantiza que:

- Las columnas objetivo C tienen exactamente C piezas de barco.
- Las columnas objetivo 0 no contienen ninguna pieza de barco.

De esta manera, se cumple que la suma de las piezas de barco en cada columna de objetivo C es exactamente C, y las columnas objetivo 0 permanecen vacías.

Por lo tanto, si existe una solución para Bin Packing, entonces existe una solución para Batalla Naval en el tablero construido.

4.5.2. Si existe una solución para Batalla Naval, entonces existe una solución para Bin Packing.

Supongamos que existe una solución válida para Batalla Naval en el tablero construido, donde las piezas de barco están colocadas de manera que:

- Las piezas de barco en las columnas con objetivo C suman exactamente C.
- Las columnas con objetivo 0 no contienen ninguna pieza de barco.

De acuerdo con la construcción del tablero, cada barco representa un número a_i del conjunto original. Las columnas en las que se encuentran los barcos indican a qué bin pertenece cada número a_i.

Por lo tanto, la colocación de los barcos en el tablero representa una partición válida del conjunto de números $\{a_1, a_2, \ldots, a_n\}$ en B subconjuntos, cada uno con una suma exacta de C.

En consecuencia, si existe una solución válida para Batalla Naval en el tablero, entonces existe una solución válida para Bin Packing.

Hemos demostrado que existe una correspondencia biunívoca entre las soluciones de Bin Packing y Batalla Naval en la construcción dada. Es decir, hay una solución para Bin Packing si y solo si hay una solución para Batalla Naval. Dado que la construcción del tablero se realiza en tiempo polinómico, hemos completado la reducción.

4.6. Si, es NP-Completo

Ya hemos demostrado que el problema de la Batalla Naval se encuentra en NP. Con esta información, sabiendo que el problema de *Bin Packing* es NP-Completo, y dado que podemos convertir cualquier instancia de *Bin Packing* en una instancia de Batalla Naval en tiempo polinómico, nuestro problema de la Batalla Naval hereda esta complejidad por transitividad.

Por lo tanto, la Batalla Naval es NP-Completo debido a la reducción planteada con *Bin Packing* y a su cumplimiento de las propiedades de los problemas en NP.

5. Solución por Backtracking

5.1. ¿Qué es Backtracking?

Backtracking es una técnica algorítmica utilizada para resolver problemas de búsqueda o toma de decisiones en los que se exploran todas las posibles soluciones de manera sistemática. Este enfoque sigue un paradigma de prueba y error en el que, a medida que se construye una solución

parcial, se evalúa si esta puede conducir a una solución válida. Si en algún punto se determina que la solución parcial no es válida, el algoritmo retrocede (*backtrack*) y deshace las decisiones tomadas, explorando otras posibilidades.

El proceso de *backtracking* se puede describir de manera formal como una búsqueda en un árbol implícito donde:

- Cada nodo representa un estado parcial de la solución.
- Las ramas representan las decisiones posibles que se pueden tomar en ese estado.
- Las hojas corresponden a soluciones completas (válidas o no) del problema.

El algoritmo verifica cada solución potencial siguiendo estos pasos:

1. Construye una solución parcial inicial.
2. Si la solución parcial cumple con las restricciones del problema:
   - Si es una solución completa, la devuelve.
   - En caso contrario, intenta expandirla con nuevas decisiones.
3. Si la solución parcial no cumple con las restricciones, retrocede y explora otras opciones.

Para nuestro problema, el enfoque de backtracking para resolverlo consiste en explorar todas las posibles configuraciones válidas de los barcos en el tablero, verificando en cada paso si se cumplen las restricciones impuestas. Este método garantiza encontrar la solución óptima, pero puede ser computacionalmente costoso debido al tamaño del espacio de búsqueda, por lo que también incluye técnicas para reducir el espacio de búsqueda a través de ciertas condiciones y estrategias de poda.

5.2. Código del algoritmo

El siguiente código corresponde al algoritmo de backtraking que encuentra la solucion optima al problema.

```
class Batalla_Naval:
    tablero = None
    mejor_tablero = None
    vacio = "-"
    maxima_demanda_cumplida = 0
    n = 0
    m = 0

    def __init__(self, restricciones_filas, restricciones_columnas, barcos):
        self.n = len(restricciones_filas)
        self.m = len(restricciones_columnas)
        self.restricciones_filas = restricciones_filas[:]
        self.restricciones_columnas = restricciones_columnas[:]
        self.demanda_restante_filas = restricciones_filas[:]
        self.demanda_restante_columnas = restricciones_columnas[:]
        self.barcos = sorted(barcos, reverse=True)
        self.tablero = [[self.vacio for _ in range(self.m)] for _ in range(self.n)]
        self.mejor_tablero = [[self.vacio for _ in range(self.m)] for _ in range(
    self.n)]
        self.bloques_disponibles = {(i, j) for i in range(self.n) for j in range(
    self.m)}
        self.backtracking(0, 0)

    def backtracking(self, indice, demanda_cumplida_actual):

        if demanda_cumplida_actual + sum(self.barcos[indice:]) * 2 <= self.
    maxima_demanda_cumplida:
            return

```

```
27          if indice == len(self.barcos):
28              if demanda_cumplida_actual > self.maxima_demanda_cumplida:
29                  self.maxima_demanda_cumplida = demanda_cumplida_actual
30                  self.mejor_tablero = [fila[:] for fila in self.tablero]
31              return
32
33          largo = self.barcos[indice]
34          bloques_vacios = self.get_bloques_disponibles()
35
36          for fila, col in bloques_vacios:
37              for orientacion in (0, 1):
38                  if self.entra_en_el_tablero(fila, col, largo, orientacion) and not
    self.exede_demanda(fila, col, largo, orientacion):
39                      if not self.tiene_adyacentes(fila, col, largo, orientacion):
40                          self.colocar_barco(fila, col, orientacion, largo, str(
    indice))
41                          demanda_cumplida = self.contar_demanda_cumplida_de_barco(
    largo)
42                          self.backtracking(indice + 1, demanda_cumplida_actual +
    demanda_cumplida)
43                          self.quitar_barco(fila, col, largo, orientacion)
44
45          if demanda_cumplida_actual + sum(self.barcos[indice:]) * 2 > self.
    maxima_demanda_cumplida:
46              self.backtracking(indice + 1, demanda_cumplida_actual)
47
48
49      def entra_en_el_tablero(self, fila, col, largo, orientacion):
50          try:
51              if orientacion == 0:  # Horizontal
52                  return all(self.tablero[fila][j] == self.vacio for j in range(col,
    col + largo))
53              else:  # Vertical
54                  return all(self.tablero[i][col] == self.vacio for i in range(fila,
    fila + largo))
55          except IndexError:
56              return False
57
58      def exede_demanda(self, fila, col, largo, orientacion):
59          if orientacion == 0:  # Horizontal
60              if self.demanda_restante_filas[fila] < largo:
61                  return True
62              if any(self.demanda_restante_columnas[j] < 1 for j in range(col, col +
    largo)):
63                  return True
64          else:  # Vertical
65              if self.demanda_restante_columnas[col] < largo:
66                  return True
67              if any(self.demanda_restante_filas[i] < 1 for i in range(fila, fila +
    largo)):
68                  return True
69          return False
70
71      def contar_demanda_cumplida_de_barco(self, largo):
72          return largo*2
73
74      def tiene_adyacentes(self, fila, col, largo, orientacion):
75          if orientacion == 0:  # Horizontal
76              for j in range(col - 1, col + largo + 1):
77                  if not (0 <= j < self.m):
78                      continue
79                  for i in [fila - 1, fila, fila + 1]:
80                      if 0 <= i < self.n and self.tablero[i][j] != self.vacio:
81                          return True
82          else:  # Vertical
83              for i in range(fila - 1, fila + largo + 1):
84                  if not (0 <= i < self.n):
85                      continue
86                  for j in [col - 1, col, col + 1]:
87                      if 0 <= j < self.m and self.tablero[i][j] != self.vacio:
```

```
                        return True
        return False

    def colocar_barco(self, fila, col, orientacion, largo, ship_indice):
        if orientacion == 0:  # Horizontal
            for j in range(col, col + largo):
                self.tablero[fila][j] = ship_indice
                self.demanda_restante_columnas[j] -= 1
            self.demanda_restante_filas[fila] -= largo
        else:  # Vertical
            for i in range(fila, fila + largo):
                self.tablero[i][col] = ship_indice
                self.demanda_restante_filas[i] -= 1
            self.demanda_restante_columnas[col] -= largo


    def quitar_barco(self, fila, col, largo, orientacion):
        if orientacion == 0:  # Horizontal
            for j in range(col, col + largo):
                self.tablero[fila][j] = self.vacio
                self.demanda_restante_columnas[j] += 1
            self.demanda_restante_filas[fila] += largo
        else:  # Vertical
            for i in range(fila, fila + largo):
                self.tablero[i][col] = self.vacio
                self.demanda_restante_filas[i] += 1
            self.demanda_restante_columnas[col] += largo

    def get_bloques_disponibles(self):
        bloques_vacios = []
        for i in range(self.n):
            for j in range(self.m):
                if self.tablero[i][j] == self.vacio and not self.tiene_adyacentes(i
, j, 1, 0):
                    bloques_vacios.append((i, j))
        return bloques_vacios
```

El proceso se detalla a continuación:

1. **Ordenamiento**: El algoritmo comienza ordenando (por unica vez) la lista de barcos de mayor a menor longitud. Esta decisión busca minimizar el número de configuraciones inválidas, al colocar primero los barcos más grandes que son los que imponen mayores restricciones en su ubicación. Esta desicion es fundamental para intentar llegar lo antes posible a una solucion "buena"(en cuanto a cumplimiento de demanda) y así poder podar lo antes posible las soluciones no tan "buenas".

```
        self.barcos = sorted(barcos, reverse=True)
```

2. **Condición de poda inicial:** Si la demanda cumplida más la máxima posible a partir de los barcos restantes no supera la mejor solución encontrada, se detiene la exploración de esa rama. Esto garantiza que no se desperdicien recursos explorando configuraciones que no pueden mejorar la solución óptima.

```
        if demanda_cumplida_actual + sum(self.barcos[indice:]) * 2 <= self.
    maxima_demanda_cumplida:
            return
```

3. **Colocación de barcos:** Para cada barco, el algoritmo evalúa todas las posiciones disponibles en el tablero, tanto en orientación horizontal como vertical. Antes de colocar un barco, verifica:
   - Que no se salga del tablero.
   - Que no exceda la demanda de las filas y columnas correspondientes.
   - Que no viole la restricción de no adyacencia con otros barcos.

```
        for fila, col in bloques_vacios:
            for orientacion in (0, 1):
                if self.entra_en_el_tablero(fila, col, largo, orientacion) and not self.exede_demanda(fila, col, largo, orientacion):
                    if not self.tiene_adyacentes(fila, col, largo, orientacion):
                        self.colocar_barco(fila, col, orientacion, largo, str(indice))
```

4. **Actualización de demandas:** Al colocar un barco, se actualizan las demandas restantes de filas y columnas. Esto facilita la evaluación de las siguientes decisiones y permite deshacer cambios de forma eficiente durante la búsqueda.

```
    def colocar_barco(self, fila, col, orientacion, largo, ship_indice):
        if orientacion == 0:  # Horizontal
            for j in range(col, col + largo):
                self.tablero[fila][j] = ship_indice
                self.demanda_restante_columnas[j] -= 1
            self.demanda_restante_filas[fila] -= largo
        else:  # Vertical
            for i in range(fila, fila + largo):
                self.tablero[i][col] = ship_indice
                self.demanda_restante_filas[i] -= 1
            self.demanda_restante_columnas[col] -= largo
```

5. **Recursión y retroceso:** Tras colocar un barco, el algoritmo llama recursivamente a sí mismo para intentar posicionar el siguiente. Si no se encuentra una solución válida, se retrocede (*backtracking*) deshaciendo la colocación del barco y restaurando las demandas.

```
    def quitar_barco(self, fila, col, largo, orientacion):
        if orientacion == 0:  # Horizontal
            for j in range(col, col + largo):
                self.tablero[fila][j] = self.vacio
                self.demanda_restante_columnas[j] += 1
            self.demanda_restante_filas[fila] += largo
        else:  # Vertical
            for i in range(fila, fila + largo):
                self.tablero[i][col] = self.vacio
                self.demanda_restante_filas[i] += 1
            self.demanda_restante_columnas[col] += largo
```

6. **Evaluación de solución:** Al alcanzar el caso base (todos los barcos evaluados), se compara la demanda cumplida con la mejor encontrada hasta el momento. Si es mayor, se actualiza la solución óptima.

```
        if indice == len(self.barcos):
            if demanda_cumplida_actual > self.maxima_demanda_cumplida:
                self.maxima_demanda_cumplida = demanda_cumplida_actual
                self.mejor_tablero = [fila[:] for fila in self.tablero]
            return
```

5.3. Decisiones, Podas y Observaciones

En esta parte se describen las decisiones implementadas para optimizar el algoritmo de backtracking, las podas realizadas para reducir el espacio de búsqueda y las observaciones relevantes obtenidas durante el desarrollo. También se discuten algunas estrategias evaluadas que no resultaron efectivas en términos de mejora de la complejidad.

- **Ordenamiento inicial de barcos (de mayor a menor longitud):** Se decidió ordenar los barcos de mayor a menor longitud antes de iniciar la búsqueda. Como se aclaró previamente esta decisión busca minimizar el número de configuraciones inválidas sin ocupar espacio adicional en memoria.

- **Poda por demanda máxima actual:** Esta poda verifica si la suma de la demanda cumplida actual más la máxima demanda posible a partir de los barcos restantes no puede superar la mejor solución encontrada hasta el momento. Esto asegura que no se exploren ramas del árbol de búsqueda que no puedan producir soluciones mejores, reduciendo significativamente el tiempo de ejecución en entradas grandes.
- **Demandas restantes de filas y columnas:** Para evitar recalcular las demandas restantes de filas y columnas en cada paso, se decidió almacenarlas y actualizarlas únicamente al colocar o quitar un barco. Esta optimización mostró ser particularmente efectiva en la mayoría de los casos, reduciendo tanto el tiempo de ejecución como la complejidad de implementación teniendo un costo espacial pequeño de $O(n+m)$, donde n es el número de filas y m el número de columnas.
- **Observación sobre la contribución constante de los barcos:** Se observó que al colocar un barco, este siempre contribuye con una cantidad constante de demanda equivalente a su longitud multiplicada por dos. Esta propiedad simplifica los cálculos necesarios para actualizar las restricciones de filas y columnas, evitando operaciones redundantes.

```
    def contar_demanda_cumplida_de_barco(self, largo):
        return largo*2
```

- **Restricción de no adyacencia:** La validación de no adyacencia se realiza antes de colocar un barco, reduciendo la cantidad de configuraciones inválidas exploradas.

```
    def tiene_adyacentes(self, fila, col, largo, orientacion):
        if orientacion == 0:  # Horizontal
            for j in range(col - 1, col + largo + 1):
                if not (0 <= j < self.m):
                    continue
                for i in [fila - 1, fila, fila + 1]:
                    if 0 <= i < self.n and self.tablero[i][j] != self.vacio:
                        return True
        else:  # Vertical
            for i in range(fila - 1, fila + largo + 1):
                if not (0 <= i < self.n):
                    continue
                for j in [col - 1, col, col + 1]:
                    if 0 <= j < self.m and self.tablero[i][j] != self.vacio:
                        return True
        return False
```

Estrategias Evaluadas y Descartadas

- **Estructura para almacenar bloques disponibles:** Inicialmente, se intentó mantener una estructura adicional para registrar los bloques disponibles en el tablero. La idea era evitar calcular los bloques disponibles desde cero en cada paso, actualizando esta estructura únicamente al colocar o quitar un barco (de forma similar a como actualizamos las demandas restantes). Aunque parecía prometedor en teoría, las pruebas mostraron que no se lograba una mejora significativa en la complejidad temporal. Por lo tanto, se optó por calcular los bloques disponibles en función del estado actual del tablero, manteniendo un modelo más simple.
- **Ordenamiento de bloques disponibles por demanda:** Otra estrategia evaluada fue ordenar los bloques disponibles de mayor a menor demanda antes de intentar colocar barcos, con el objetivo de llenar primero los bloques más demandados. Sin embargo, el costo adicional de ordenar los bloques en cada paso superó los beneficios obtenidos en términos de eficiencia. Por este motivo, esta estrategia fue descartada.

Observaciones Finales

Durante el desarrollo del algoritmo, se observó que algunas podas y estrategias mejoraban la complejidad en casos específicos, pero empeoraban el rendimiento en otras entradas. Por esta

razón, se mantuvieron únicamente aquellas podas y optimizaciones que demostraron ser efectivas en la mayoría de los casos, mientras que las estrategias menos consistentes fueron descartadas. Este enfoque equilibrado permitió alcanzar un algoritmo eficiente y robusto frente a diferentes configuraciones de entrada.

5.4. Complejidad del Algoritmo

La complejidad de este algoritmo de backtracking depende de varios factores, entre los que destacan el número de barcos k, el tamaño del tablero $n \times m$, y las restricciones de demanda. A continuación se realiza un análisis detallado de los factores que afectan la complejidad del algoritmo.

Número de barcos (k): El algoritmo de backtracking recursivo itera sobre los barcos, tratando de colocarlos en el tablero. En el peor caso, el algoritmo realiza $k!$ iteraciones si no se aplica poda (optimización). Sin embargo, debido a las restricciones y las condiciones de poda, el número de iteraciones puede reducirse.

Posibilidades para cada barco :

- **Posiciones disponibles**: Para cada barco, puede colocarse en cualquier posición dentro del tablero $n \times m$. La cantidad total de posiciones disponibles para un barco es proporcional a $n \times m$, ya que el algoritmo recorre todos los bloques vacíos en el tablero para encontrar las posiciones válidas.
- **Orientaciones**: Cada barco tiene dos orientaciones posibles: horizontal y vertical. Esto multiplica las opciones disponibles por 2 para cada barco.

Poda en el backtracking: El algoritmo aplica poda en cada nivel de la recursión, evitando explorar ramas del árbol de búsqueda que no puedan cumplir con las restricciones de demanda. Sin embargo, en el peor caso, la poda no cambia la complejidad asintótica del algoritmo, aunque mejora el rendimiento en la práctica.

Recursividad: El algoritmo realiza una búsqueda exhaustiva de todas las combinaciones posibles de colocación de los barcos. El número de combinaciones está determinado por el número de posiciones y orientaciones disponibles para cada barco.

Combinaciones posibles :

- Para cada barco, hay $n \times m$ posiciones posibles.
- Cada barco tiene 2 orientaciones posibles.
- Hay k barcos que deben colocarse en el tablero.

Por lo tanto, el número total de combinaciones posibles de colocación de los barcos es aproximadamente:

$$O\left((n \times m)^k \cdot 2^k\right)$$

Es decir, en el peor caso, el algoritmo podría recorrer todas las combinaciones posibles de colocación de los k barcos, considerando todas las posiciones y orientaciones. Esto da como resultado una complejidad exponencial en función de n, m y k.

```
1    def backtracking(self, indice, demanda_cumplida_actual):
2
3        if demanda_cumplida_actual + sum(self.barcos[indice:]) * 2 <= self.
     maxima_demanda_cumplida:
4            return
5
6        if indice == len(self.barcos):
7            if demanda_cumplida_actual > self.maxima_demanda_cumplida:
8                self.maxima_demanda_cumplida = demanda_cumplida_actual
```

```
                self.mejor_tablero = [fila[:] for fila in self.tablero]
            return

        largo = self.barcos[indice]
        bloques_vacios = self.get_bloques_disponibles()

        for fila, col in bloques_vacios:
            for orientacion in (0, 1):
                if self.entra_en_el_tablero(fila, col, largo, orientacion) and not self.exede_demanda(fila, col, largo, orientacion):
                    if not self.tiene_adyacentes(fila, col, largo, orientacion):
                        self.colocar_barco(fila, col, orientacion, largo, str(indice))
                        demanda_cumplida = self.contar_demanda_cumplida_de_barco(largo)
                        self.backtracking(indice + 1, demanda_cumplida_actual + demanda_cumplida)
                        self.quitar_barco(fila, col, largo, orientacion)

        if demanda_cumplida_actual + sum(self.barcos[indice:]) * 2 > self.maxima_demanda_cumplida:
            self.backtracking(indice + 1, demanda_cumplida_actual)
```

Factores adicionales : Las funciones de validación como `entra_en_el_tablero`, `exede_demanda`, `tiene_adyacentes` y la actualización de los contadores de demanda en cada colocación de barcos tienen un costo adicional en cada iteración, pero este costo está relacionado principalmente con el tamaño del barco y la posición en el tablero. Estos cálculos no afectan la complejidad principal, pero contribuyen a la constante multiplicativa.

Conclusión: La complejidad asintótica del algoritmo es exponencial en el peor caso y está determinada por el número de combinaciones posibles de colocación de los barcos. La complejidad es aproximadamente:

$$O\left((n \times m)^k \cdot 2^k\right)$$

Esto implica que el algoritmo tiene una alta complejidad, y su rendimiento dependerá en gran medida de las restricciones y del tamaño del tablero.

5.5. Resultados del Algoritmo

El algoritmo fue probado en instancias pequeñas y medianas del problema, obteniendo soluciones óptimas en tiempos razonables. Para instancias más grandes, donde el espacio de búsqueda se vuelve intratable, se observaron tiempos de ejecución crecientes.

Hemos notado que el algoritmo de backtracking es una herramienta poderosa para encontrar soluciones exactas al problema de La Batalla Naval. Sin embargo, su complejidad inherente lo hace poco práctico para instancias grandes. En estos casos, es necesario recurrir a aproximaciones o métodos heurísticos para obtener soluciones en tiempos aceptables.

6. Solución por Programación Lineal

Dado el mismo planteamiento del enunciado de Backtracking, indaguemos cómo podemos resolver la Batalla Naval mediante programación lineal. Para abordar esta técnica, primero definamos brevemente qué es la programación lineal.

6.1. ¿Qué es la Programación Lineal?

La **programación lineal** es una técnica de optimización matemática que busca maximizar o minimizar una función objetivo, sujeta a un conjunto de restricciones lineales. Las variables involucradas están restringidas por inecuaciones o ecuaciones lineales, y las soluciones que satisfacen

estas restricciones constituyen la *región factible* del problema. El objetivo es encontrar el punto óptimo dentro de esta región, es decir, el valor que maximiza o minimiza la función objetivo.

7. Solución por Aproximación

7.1. Introducción

Como indica John Jellicoe, el algoritmo es el siguiente:

1. Ir a la fila o columna con mayor demanda.
2. Ubicar el barco de mayor longitud en dicha fila o columna en algún lugar válido.
3. Si el barco de mayor longitud es más largo que la demanda correspondiente, simplemente saltarlo y seguir con el siguiente barco.
4. Repetir este proceso hasta que no queden más barcos o no haya más demandas que cumplir.

La decisión de priorizar la fila o columna con mayor demanda asegura que se aborden primero las restricciones más exigentes, lo cual es fundamental para maximizar la eficiencia del algoritmo greedy. Esto reduce las posibilidades de que las celdas de alta demanda queden bloqueadas sin ser utilizadas.

7.2. Motivación:

La Batalla Naval es un problema clásico de optimización en el que se busca ubicar barcos en un tablero de forma que se maximice la demanda satisfecha. Sin embargo, debido a la complejidad del problema, los enfoques exactos no siempre son prácticos en escenarios con grandes cantidades de barcos y restricciones. El algoritmo de aproximación basado en una estrategia greedy presenta una solución eficiente aunque no necesariamente óptima, permitiendo un balance entre tiempo de ejecución y calidad de la solución.

7.3. Código del Algoritmo

El siguiente código corresponde al algoritmo de aproximación. Las funciones para verificar si es posible colocar un barco en una posición son las mismas que se utilizan para el algoritmo de backtracking.

```
def aproximacion(tablero, restricciones_filas, restricciones_columnas, barcos):

    restricciones = transoformar_restricciones(restricciones_filas,
    restricciones_columnas)

    n = len(restricciones_filas)
    m = len(restricciones_columnas)

    restricciones.sort(key=lambda x: x[1], reverse=True)
    barcos.sort(reverse=True)
    barco_actual = 0
    demanda_restante_filas = restricciones_filas[:]
    demanda_restante_columnas = restricciones_columnas[:]
    pude_poner = False

    for restriccion in restricciones:
        idx, demanda, es_fila = restriccion
        pude_poner = False
        if not barcos:
            break
        largo_barco = barcos.pop(0)
        # Ir a fila/columna de mayor demanda,
```

```
        while not pude_poner:
            if es_fila:
                idx_fila = idx
                for j in range(m):
                    # y ubicar el barco de mayor longitud en dicha fila/columna en alg n lugar v lido.
                    # Si el barco de mayor longitud es m s largo que dicha demanda, simplemente saltearlo y seguir con el siguiente.

                    if entra_en_el_tablero(tablero, idx_fila, j, largo_barco, 0) \
                        and not exede_demanda(idx_fila, j, largo_barco, 0, demanda_restante_filas, demanda_restante_columnas) \
                        and not tiene_adyacentes(tablero, idx_fila, j, largo_barco, 0, n, m):
                        tablero, demanda_restante_filas, demanda_restante_columnas = colocar_barco(tablero, idx_fila, j, 0, largo_barco, barco_actual, demanda_restante_filas, demanda_restante_columnas)
                        barco_actual += 1
                        pude_poner = True
                        break
            else:
                idx_col = idx
                for i in range(n):

                    if entra_en_el_tablero(tablero, i, idx_col, largo_barco, 1) \
                        and not exede_demanda(i, idx_col, largo_barco, 1, demanda_restante_filas, demanda_restante_columnas) \
                        and not tiene_adyacentes(tablero, i, idx_col, largo_barco, 1, n, m):
                        tablero, demanda_restante_filas, demanda_restante_columnas = colocar_barco(tablero, i, idx_col, 1, largo_barco, barco_actual, demanda_restante_filas, demanda_restante_columnas)
                        barco_actual += 1
                        pude_poner = True
                        break
            if not barcos:
                break

            largo_barco = barcos.pop(0)
```

7.4. Análisis de complejidad

1. Inicialización

```
    restricciones = transoformar_restricciones(restricciones_filas, restricciones_columnas)

    n = len(restricciones_filas)
    m = len(restricciones_columnas)

    restricciones.sort(key=lambda x: x[1], reverse=True)
    barcos.sort(reverse=True)
```

- La complejidad de (`transoformar_restricciones`) es $O(n+m)$, suponiendo n filas y m columnas.
- Ordenar las restricciones tiene una complejidad de $O((n+m)\log(n+m))$.
- Ordenar la lista de barcos tiene una complejidad de $O(k \log k)$, donde k es el número de barcos.

Por lo tanto, la complejidad de esta sección es:

$$O(n+m+(n+m)\log(n+m)+k\log k).$$

2. Iteración Principal Sobre Restricciones

El bucle principal recorre todas las restricciones:

```
    for restriccion in restricciones:
        idx, demanda, es_fila = restriccion
        pude_poner = False
        if not barcos:
            break
        largo_barco = barcos.pop(0)
```

Este bucle tiene $O(n+m)$ iteraciones, ya que n filas y m columnas se combinan en la lista de restricciones. Ademas, quitar el primer barco de la lista tiene una complejidad de $O(1)$, y esto ocurre como máximo k veces en todas las iteraciones.

3. Recorrer los barcos

```
        while not pude_poner:
            if es_fila:
                idx_fila = idx
                for j in range(m):
                    # y ubicar el barco de mayor longitud en dicha fila/columna en alg n lugar v lido.
                    # Si el barco de mayor longitud es m s largo que dicha demanda, simplemente saltearlo y seguir con el siguiente.

                    if entra_en_el_tablero(tablero, idx_fila, j, largo_barco, 0) \
                        and not exede_demanda(idx_fila, j, largo_barco, 0, demanda_restante_filas, demanda_restante_columnas) \
                        and not tiene_adyacentes(tablero, idx_fila, j, largo_barco, 0, n, m):
                        tablero, demanda_restante_filas, demanda_restante_columnas = colocar_barco(tablero, idx_fila, j, 0, largo_barco, barco_actual, demanda_restante_filas, demanda_restante_columnas)
                        barco_actual += 1
                        pude_poner = True
                        break
```

El proceso principal para ubicar el barco incluye un bucle que itera sobre filas o columnas:

La operación más costosa es iterar sobre m columnas o n filas.

Dentro de los bucles, tenemos las operaciones:

- `entra_en_el_tablero()`
- `exede_demanda()`
- `tiene_adyacentes()`
- `colocar_barco()`

Cada una de estas verificaciones cuesta $O(1)$, pero el bucle puede recorrer todas las celdas de una fila (m) o columna (n) antes de encontrar una ubicación válida. En el peor caso:

$$O(m) \quad \text{para filas, o} \quad O(n) \quad \text{para columnas.}$$

4. Complejidad General

Tenemos entonces:

- Inicializacion
$$O(n+m+(n+m)\log(n+m)+k\log k).$$
- Iterar por restricciones
$$O(n+m)$$

- Iterar por fila o columna hasta poder poner un barco

$$k \cdot \text{máx}(n, m)$$

El bucle externo itera sobre $O(n + m)$ restricciones, y para cada una intenta colocar barcos. Cada barco tiene un costo de $O(m)$ o $O(n)$. En el peor caso, se intentan colocar todos los barcos (k).

Esto da una complejidad de:

$$O((n + m) \cdot k \cdot \text{máx}(n, m)).$$

Entonces tenemos:

$$\begin{aligned} &O(n + m + (n + m)\log(n + m) + k \log k) \quad \text{(inicialización)} \\ &\quad + O((n + m) \cdot k \cdot \text{máx}(n, m)) \quad \text{(bucle principal)}. \end{aligned}$$

El término dominante es la fase principal, por lo que la complejidad general es:

$$O((n + m) \cdot k \cdot \text{máx}(n, m)).$$

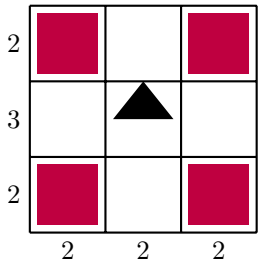
Conclusión: A pesar de que la complejidad de este algoritmo es mayor en escenarios con muchas restricciones y barcos, su comportamiento práctico sigue siendo adecuado en situaciones reales, ya que la estrategia greedy permite obtener soluciones razonablemente buenas con un tiempo de ejecución que sigue siendo aceptable para tableros de tamaño moderado.

7.5. Cálculo teórico de $r(A)$

En esta sección analizaremos el impacto teórico de la estrategia codiciosa (greedy) en el peor caso, denotado como $r(A)$, considerando el máximo desperdicio posible al priorizar las demandas y alinear los barcos. Este análisis se enfoca en la relación entre la demanda cumplida por el algoritmo de aproximación y la solución óptima en diferentes configuraciones de barcos y demandas.

- **Caso con barcos de tamaño 1:**

  En este primer caso, se considera una cuadrícula 3×3 donde la demanda en las filas y columnas es variable. Colocamos el barco (negro) de tamaño 1 en la posición central de la cuadrícula para ilustrar el peor caso, y los barcos bordó en las esquinas para ilustrar el caso óptimo.

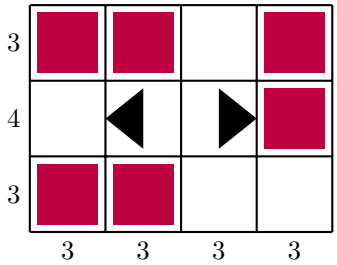


  En este ejemplo, al colocar uno de los barcos en la posición central, observamos que se bloquean otras ubicaciones posibles, lo que genera un desperdicio de espacio. Como resultado, en el peor caso, la demanda cumplida es 2, mientras que en el mejor caso se puede cumplir una demanda de 8. Por lo tanto, el valor de $r(A)$ es:

$$r(A) = \frac{\text{Demanda cumplida}}{\text{Demanda óptima}} = \frac{2}{8} = \frac{1}{4}.$$

- **Caso con barcos de tamaño 2:**

  Ahora consideraremos el caso con barcos de tamaño 2. Utilizaremos una cuadrícula 4×3 con demandas en las filas y columnas. Colocamos los barcos de tamaño 2 en posiciones estratégicas y analizamos el resultado.

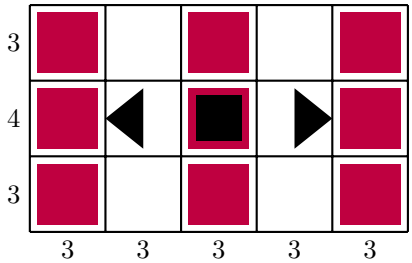


  En este escenario, el peor caso ocurre cuando uno de los barcos se coloca en una posición que bloquea otras áreas, cumpliendo solo con una demanda de 4. Mientras que en la solución óptima, se pueden cumplir 12 unidades de demanda. Por lo tanto, el valor de $r(A)$ es:

$$r(A) = \frac{4}{12} = \frac{1}{3}.$$

- **Caso con barcos de tamaño 3:**

  A continuación, exploramos un caso con barcos de tamaño 3 en una cuadrícula 5×3. En este caso, colocamos los barcos de tamaño 3 y analizamos el impacto de las decisiones del algoritmo de aproximación.



  En este caso, el peor escenario implica que se cumple una demanda de 6 unidades, mientras que en el mejor escenario la demanda cumplida es de 18. Por lo tanto, el valor de $r(A)$ es nuevamente:

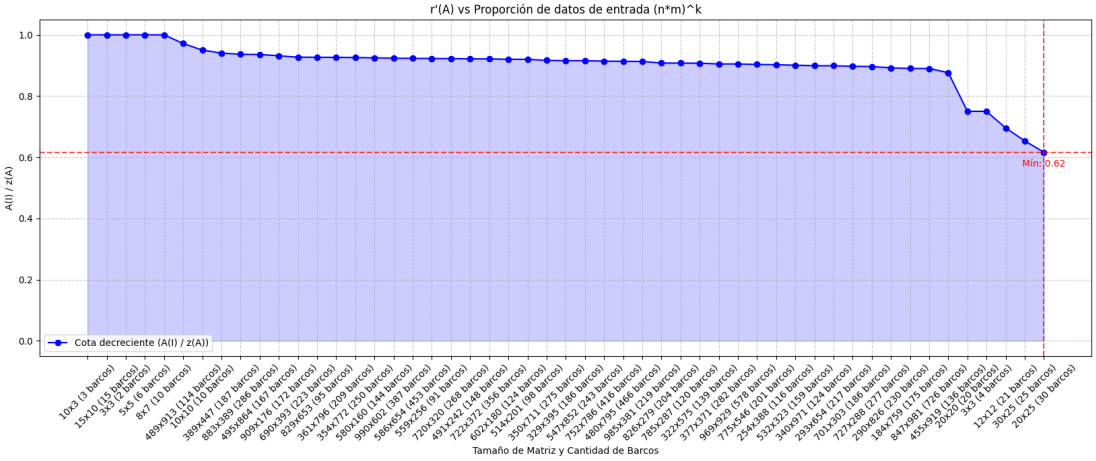
$$r(A) = \frac{6}{18} = \frac{1}{3}.$$

Conclusión:

A partir de los análisis realizados, se observa que, aunque el valor de $r(A)$ puede variar dependiendo de la configuración específica del problema, en la mayoría de los casos, este valor se estabiliza en torno a $\frac{1}{3}$ y el peor caso ocurre al utilizar barcos de tamaño 1, lo que da como resultado un valor de $r(A) = \frac{1}{4}$. Esto sugiere que el algoritmo greedy utilizado tiene una aproximación constante, logrando un factor de aproximación de 4 en el peor caso al problema de la Batalla Naval (como mucho la demanda obtenida valdrá $\frac{1}{4}$ de la demanda óptima).

7.6. Mediciones empiricas para $r(A)$.

A continuacion, realizamos mediciones del algoritmo de aproximacion y las comparamos con el valor optimo. En el grafico se visualizan distintos r(A) obtenidos.



Como se observa, el menor valor conseguido es 0.62, esto concuerda con el r(A) calculado teoricamente, que era como minimo 0.25. Es decir:

$$r(A)_{\text{teórico}} \leq r(A)_{\text{empírico}}$$

Los resultados obtenidos en el gráfico superan notablemente la cota teórica esperada, manteniendo valores cercanos a 1 para la relación $\frac{\text{A(I)}}{\text{z(I)}}$ en la mayoría de los casos. Esto indica que el algoritmo greedy utilizado no solo es eficiente, sino que también proporciona soluciones de alta calidad en la práctica, con un rendimiento promedio que excede las expectativas teóricas. Cabe destacar que incluso en los casos más desfavorables, el valor mínimo registrado fue de 0.62, lo cual sigue siendo un resultado competitivo.

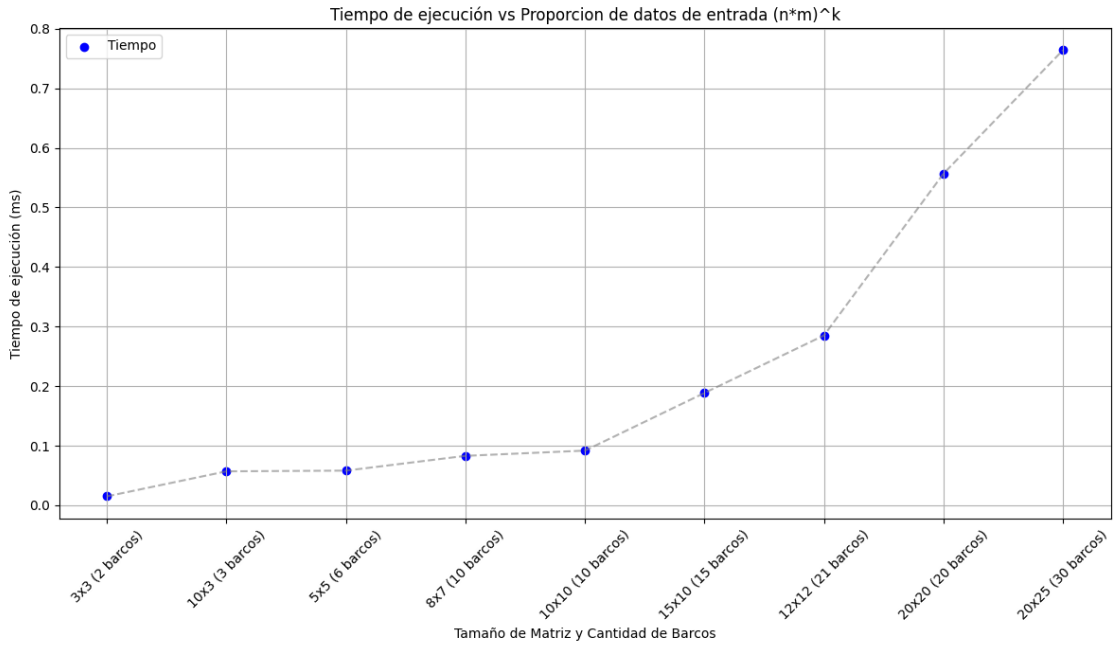
7.7. Conclusión

Este análisis muestra que el algoritmo de aproximación basado en una estrategia greedy es una herramienta útil cuando se busca una solución rápida en problemas de Batalla Naval con un número elevado de restricciones. Aunque no garantiza una solución óptima, proporciona un factor de aproximación razonable, lo que lo hace adecuado en situaciones donde el tiempo de ejecución es crucial.

8. Mediciones

8.1. Análisis del Tiempo de Ejecución del Algoritmo de Backtracking

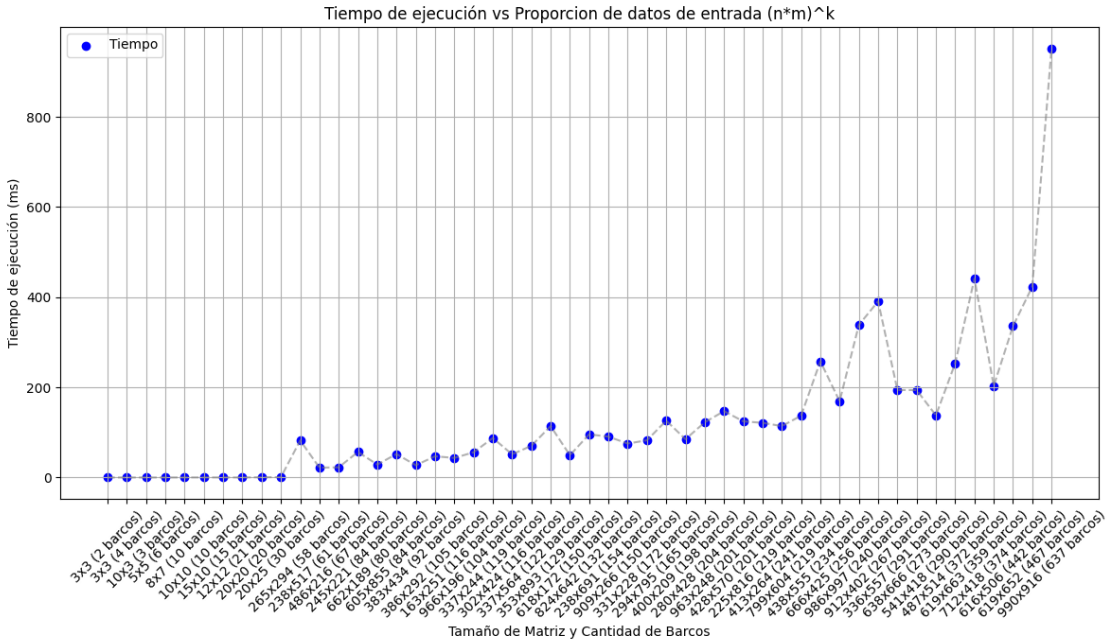
El siguiente gráfico muestra el tiempo de ejecución del algoritmo de backtracking en función del tamaño de entrada, definido por el tamaño de la matriz ($n \times m$) y la cantidad de barcos (k). A continuación, se presenta un análisis detallado:



8.1.1. Observaciones

- **Relación entre el tamaño de la entrada y el tiempo de ejecución:** A medida que aumenta el tamaño de la matriz y la cantidad de barcos, el tiempo de ejecución crece de manera no lineal, lo cual es característico de algoritmos de backtracking debido al crecimiento exponencial del espacio de búsqueda.
- **Comportamiento inicial:** Para tamaños pequeños de matriz y pocos barcos (e.g., 3×3 con 2 barcos, 5×5 con 6 barcos), los tiempos de ejecución son insignificantes (cercanos a 0 ms). Esto es esperado ya que el espacio de búsqueda es limitado.
- **Crecimiento acelerado:** A partir de tamaños de matriz intermedios (10×10 con 10 barcos), el tiempo de ejecución comienza a incrementarse de forma más evidente. Esto refleja la dificultad del problema al aumentar las restricciones y combinaciones posibles de ubicación de barcos.
- **Tendencia exponencial:** Para los tamaños más grandes (20×20 con 20-30 barcos), el tiempo de ejecución aumenta significativamente, lo cual es consistente con la complejidad del backtracking en problemas de optimización con múltiples restricciones.
- **Proporción del tiempo de ejecución:** La gráfica sigue una curva suavemente creciente, que podría aproximarse a un modelo polinómico o exponencial. Esto puede verificarse ajustando los datos a una función matemática para estimar la complejidad en términos de $O(f(n, m, k))$.
- **Escalabilidad limitada:** El algoritmo de backtracking es eficiente para entradas pequeñas o moderadas, pero su desempeño disminuye drásticamente con problemas de mayor tamaño debido a su complejidad inherente.

8.2. Análisis del Tiempo de Ejecución del Algoritmo de Aproximacion



8.2.1. Observaciones

Crecimiento del tiempo de ejecución: Para tamaños pequeños del tablero y una cantidad limitada de barcos, el tiempo de ejecución es prácticamente constante y cercano a cero. Esto indica que el algoritmo de aproximación maneja de manera eficiente instancias pequeñas del problema.

Escalabilidad: A medida que el tamaño del tablero y la cantidad de barcos aumentan, se observa un ligero incremento en el tiempo de ejecución, aunque este incremento no es tan abrupto como en algoritmos más complejos como el de backtracking. Esto confirma que el enfoque de aproximación es más escalable en comparación con métodos exactos.

Utilidad del algoritmo: Aunque el algoritmo de aproximación no garantiza soluciones óptimas, su rapidez lo hace ideal para escenarios donde la precisión no es crítica y el tamaño de los datos puede ser considerable.

Comportamiento lineal en pequeñas entradas: Para tamaños de entrada pequeños, el tiempo de ejecución se comporta de manera casi constante, lo que refuerza su eficiencia en problemas de baja complejidad.

9. Conclusiones

En este trabajo práctico implementamos el algoritmo de Batalla Naval utilizando tres enfoques: backtracking, programación lineal y un algoritmo de aproximación.

Pudimos observar que el algoritmo de aproximación, si bien no garantiza encontrar la solución óptima, es considerablemente más rápido que los otros dos llegando a resultados más que aceptables. Esto resulta útil en casos donde no se requiere un alto grado de precisión en la solución pero sí velocidad, o incluso en casos en donde se manejan volúmenes de datos inmanejables para el algoritmo exacto.

Por otro lado, los algoritmos basados en backtracking y programación lineal presentan un crecimiento exponencial en su tiempo de ejecución conforme aumenta la complejidad del problema. En particular, el algoritmo de backtracking muestra un crecimiento abrupto en su curva de tiempo

de ejecución a partir de un tablero de 10×10 con 10 barcos, como se ve en el grafico de mediciones.