.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico: Juego de Hermanos

3 de diciembre de 2024

Integrante	Padrón	Mail
Mateo Lardiez	107992	mlardiez@fi.uba.ar
Tomás Vainstein	109043	tvainstein@fi.uba.ar
Julián Gorge	104286	jgorge@fi.uba.ar
Juan Cuevas	107963	jcuevas@fi.uba.ar
Víctor Zacarías	107080	vzacarias@fi.ubar.ar

1. Primera parte: Algoritmos greedy

1.1. Análisis del problema

El juego de las monedas es un juego multijugador por turnos muy simple que consiste en que dada una fila de monedas, cada jugador en su respectivo turno deberá escoger una moneda de los extremos de la fila, dando lugar al turno del siguiente jugador. Una vez que se hayan agotado las monedas, el jugador que sume mayor valor es el ganador del juego.

En particular, se están estudiando el juego para el caso en el que se tiene a Sophia y Mateo como únicos jugadores. El objetivo es el de generar un algoritmo greedy que obtenga una solución que se asegure de que Sophia siempre gane el juego, teniendo en cuenta que como Mateo es muy chico Sophia juega por él (hace trampa).

1.2. Algoritmo greedy

Dado un arreglo de monedas sin ordenar, con una cantidad de monedas que puede ser par o impar, el siguiente algoritmo greedy devuelve la solución óptima al problema de las monedas devolviendo como salida una lista de las monedas de Sophia (las mejores) y una lista de las monedas de Mateo (las peores).

Para la implementación del algoritmo se tuvieron en cuenta las restricciones de que el juego es un juego por turnos y de que solo se pueden agarrar monedas de los extremos. Nótese que Mateo siempre tendrá las peores monedas para sí mismo.

```
def play_game(coins_list):
    sophia_coins = []
    mateo_coins = []

    sophia_is_playing = True

    beginning = 0
    end = len(coins_list) - 1

    while beginning <= end:

        selected_coin = -1

        if sophia_is_playing:
            if coins_list[beginning] <= coins_list[end]:
                selected_coin = coins_list[end]
                end -=1
            else:
                selected_coin = coins_list[beginning]
                beginning +=1

            sophia_coins.append(selected_coin)
            sophia_is_playing = False
        else:
            if coins_list[beginning] >= coins_list[end]:
                selected_coin = coins_list[end]
                end -=1
            else:
                selected_coin = coins_list[beginning]
                beginning +=1

            mateo_coins.append(selected_coin)
            sophia_is_playing = True

    return sophia_coins, mateo_coins
```

1.3. Solución óptima

Por definición, en un algoritmo greedy las soluciones óptimas se construyen a partir de una secuencia de óptimos locales que paso a paso van llevando a la mejor solución a un problema. La lógica greedy es la de tomar la mejor decisión en cada paso de un problema, y que el conjunto de las mejores decisiones lleven al óptimo.

Esta definición podría confundirse con la de la programación dinámica. Pero la diferencia principal es que los algoritmos greedy no tienen memoria.

Claramente no todo problema solucionado de forma greedy tiene solución óptima, sin embargo, para el caso del juego de las monedas con las restricciones definidas anteriormente sí se alcanza el óptimo.

En primer lugar, Sophia busca maximizar las ganancia total al terminar el juego, en cada turno en el que le toque jugar va a tomar la moneda de mayor valor (óptimo local).

Por último, como Sophia juega por Mateo y se asegura de ganar, siempre va a escoger las monedas de menor valor para él, minimizándose así la ganancia de Mateo.

Esta combinación se asegura que en cada paso Sophia tomé la mejor moneda para sí misma, y que Mateo se lleve la peor moneda. Como resultado, Sophia tendrá las monedas de mayor valor el finalizar el juego, y habrá ganado el juego con la solución óptima (exceptuando el caso borde de una cantidad par de monedas del mismo valor, donde se produce un empate).

Respecto al análisis de la complejidad algorítmica. Se formulo la hipótesis teórica de que el algoritmo propuesto tiene una complejidad $O(n)$. En cada paso del problema se está realizando un problema de decisión entre dos monedas, lo cual se resuelve en tiempo $O(1)$, por otra parte, la lista se itera alternando sus extremos de afuera hacia adentro, al iterar sobre todos los elementos se tendría una complejidad $O(n)$ (la complejidad de decidir cuál moneda tomar es despreciable comparada a la complejidad de moverse por la lista de monedas).

1.4. Variabilidad y optimalidad

La variabilidad de los valores de las monedas no afectaría a los tiempos de ejecución del algoritmo plateando. El tiempo de ejecución del algoritmo no depende del valor de las monedas sino de la cantidad de monedas con las que Sophia y Mateo estén jugando (una moneda no es más que un valor entero almacenado en una lista, y en cada paso simplemente se busca la mayor moneda entre los dos extremos) .

Respecto a la optimalidad de la solución. La variabilidad de los valores de las monedas no afectaría a la solución ya que Sophia siempre maximiza su ganancia acumulada y minimiza la del Mateo (por más que varíen los valores de las monedas Sophia seguirá buscando lo mejor para sí misma).

El algoritmo greedy implementado es robusta ante diferentes distribuciones de valores.

1.5. Ejemplos de ejecución

A modo de ejemplos, se utilizaron los datasets provistos por la cátedra para testear la optimalidad del algoritmo implementado.

- 20.txt espera ganancia de 7165 para Sophia.
- 25.txt espera ganancia de 9135 para Sophia.
- 50.txt espera una ganancia de 17750 para Sophia.
- 100.txt espera una ganancia de 35009 para Sophia.
- 1000.txt espera una ganancia de 357814 para Sophia.

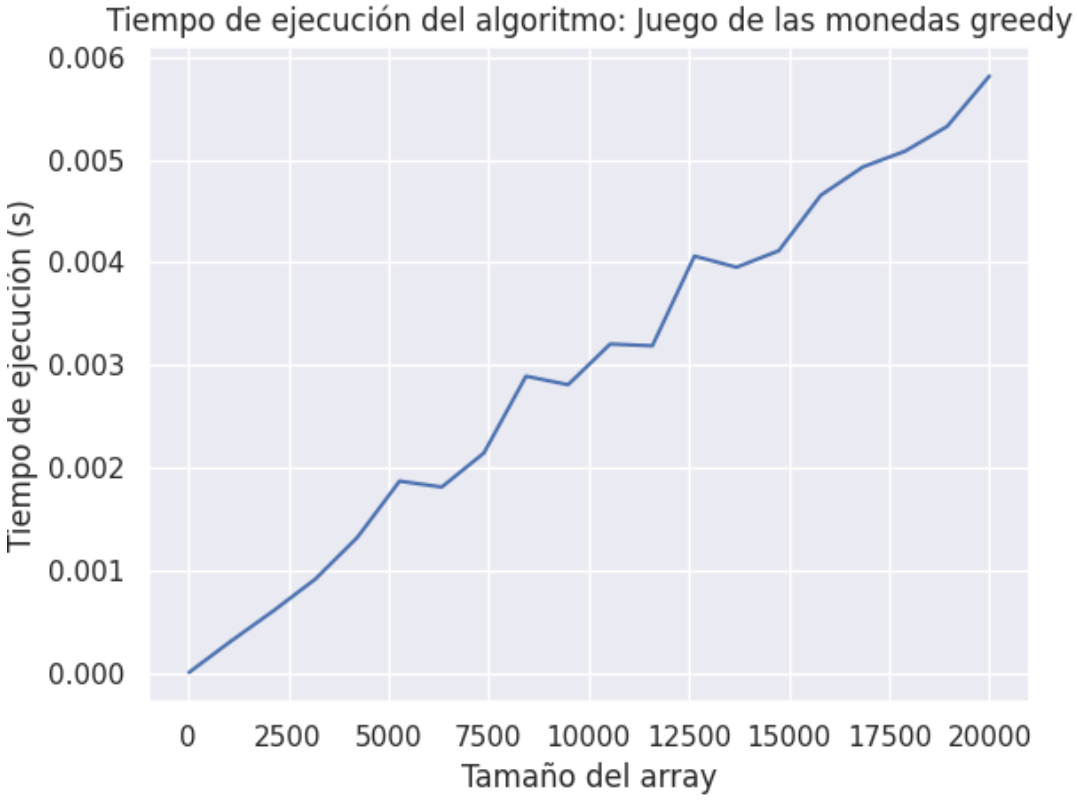
- 10000.txt espera una ganancia de 3550307 para Sophia.
- 20000.txt espera una ganancia de 7139357 para Sophia.

Al cumplir con la salida esperada de todos los ejemplos por se podría decir el algoritmo greedy implementado funciona correctamente.

Cabe aclarar que en los últimos 2 casos (10000.txt y 20000.txt) hay casos en que ambos extremos tienen el mismo valor, por lo que se decidió elegir que, en esta situación, si ambos valores de la primer moneda y la ultima son iguales, se tome la ultima moneda

1.6. Comprobación de la complejidad

Para corroborar empíricamente la complejidad teórica de $O(n)$, realizamos mediciones del tiempo de ejecución del algoritmo. Estos tiempos fueron luego graficados en función de n (el tamaño del problema) para observar la relación entre el tamaño de la entrada y el tiempo de ejecución.



Se puede apreciar que el gráfico corresponde con la hipótesis de la complejidad teórica.

1.7. Conclusiones y comentarios adicionales

Se utilizo un enfoque greedy para hallar la solución óptima al juego de las monedas entre Sophia y Mateo. Dadas las restricciones y a la implementación de la solución, Sophia siempre ganará el juego con el puntaje máximo y con complejidad $O(n)$.

Se comprobó que siempre se llega a la solución óptima esperada, utilizando diferentes datasets.

Respecto al proceso de implementación del algoritmo, inicialmente se consideró utilizar métodos de ordenamiento sobre las monedas antes de iniciar el juego, lo que hacía que la solución óptima

se resumiese en siempre sacar monedas del mismo lugar hasta terminar el juego. Esta idea fue descartada inmediatamente al corroborar que rompía por completa con la lógica y esencia del juego.

En términos de implementación, no se presentaron grandes dificultades, salvo en la creación de los gráficos de complejidad, que requirieron del uso de librerías de las cuales se desconocía su uso. Estos gráficos confirmaron la relación lineal entre el tiempo de ejecución y el número de monedas, alineándose con la complejidad teórica calculada.

2. Segunda parte: Mateo empieza a jugar

2.1. Análisis del problema

Al igual que en la sección anterior se está jugando al juego de las monedas entre Sophia y Mateo, pero con un particular cambio, ahora Mateo empieza a jugar con lógica greedy (es decir, Sophia ya no escoge las monedas por él), lo cual hace que Sophia recurra a un cambio en la forma en que juega.

En esta ocasión, Sophia empieza a usar programación dinámica obtener las mejores monedas del juego, y de ser posible ganar (sí, ahora es posible que Sophia pierda). Si bien Sophia puede perder, si lo hace, lo hará teniendo la mejor colección de monedas.

Para diseñar el algoritmo se tuvo en cuenta que como ahora Mateo juega con lógica greedy siempre va a tomar las mejores monedas de los extremos.

Para lograr resolver este problema se pensó lo siguiente:

Como las monedas siempre se agarran de los extremos de la colección de monedas. Se definieron a estos extremos como **i** y **j**. Los cuales van variando a la medida que se van sacando monedas de la colección.

Por otra parte, si arma una matriz de nxn (siendo n la cantidad de monedas del juego). Y se definen los índices de las filas como **i** y a los índices de las columnas como **j**, es posible obtener una matriz con los óptimos para los distintos valores que pueden tomar los extremos de la colección de monedas.

Los índices **i** y **j** indican en que posición de la colección de n monedas se encuentran los extremos actuales.

Esta matriz permitiría construir los óptimos para cada posible valor de **i** y de **j**, y así, basándose en una lógica inductiva y al uso de memoization, se podría obtener la solución óptima al juego de las monedas para Sophia.

Ejemplo de una matriz teniendo 5 monedas:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

El óptimo para el juego de las monedas se encontraría en la posición i=0, j=n-1. Estando las monedas almacenadas en una lista.

Esta descripción es solo un acercamiento inicial a la implementación del problema. Y a continuación se la describira en más detalle mediante la ecuación de recurrencia.

2.2. Ecuación de recurrencia

La siguiente ecuación de recurrencia es descrita basándose en lógica algorítmica de lenguajes de programación (en lo que se refiere a la forma de acceder a las monedas, a la variación de los extremos, y al acceso de la matriz de óptimos)

En cada paso se puede tomar una moneda de un extremo u otro, en base a cual moneda se tomé será necesario calcular un nuevo óptimo para los nuevos externos teniendo en cuenta que Mateo siempre elegirá las monedas de mayor valor entre los extremos.

$$\text{OPT}(i,j) = \text{máx}\left(\text{monedas}[i]+\text{mejor\_moneda\_siguiente}(i+1,j),\ \text{monedas}[j]+\text{mejor\_moneda\_siguiente}(i,j-1)\right)$$

$$\begin{aligned} &i = \text{inicio de la lista de monedas, varía según la moneda seleccionada} \\ &j = \text{fin de la lista de monedas, varía según la moneda seleccionada} \\ &\text{monedas}[i] = \text{valor de la moneda en la posición } i \\ &\text{monedas}[j] = \text{valor de la moneda en la posición } j \end{aligned} \tag{1}$$

$$\text{mejor\_moneda\_siguiente} = \begin{cases} 0, & \text{si } i+1 > j \text{ (no quedan monedas para Sofía)} \\ \text{OPT}(i+1, j), & \text{si monedas}[i] \geq \text{monedas}[j] \text{ (Mateo toma la izquierda)} \\ \text{OPT}(i, j-1), & \text{si monedas}[i] < \text{monedas}[j] \text{ (Mateo toma la derecha)} \end{cases}$$

2.3. Algoritmo planteado

El siguiente código muestra la implementación del algoritmo haciendo uso de la ecuación de recurrencia descrita anteriormente.

```
from libs import datasets_parser

def best_next_coin(i, j, gains_matrix, coins_list):
    if i+1 >j:
        return 0
    elif coins_list[i] >= coins_list[j]:
        return gains_matrix[i+1][j]
    else:
        return gains_matrix[i][j-1]


def get_gains_matrix(coins_list):

    coins_list_len = len(coins_list)

    gains_matrix = [[0 for _ in range(coins_list_len)] for _ in range(
    coins_list_len)]

    for i in range(coins_list_len-1, 0, -1):

        for j in range(0, coins_list_len):

            if  i > j:
                gains_matrix[i][j] = 0
            else:
                gains_matrix[i][j] = max(coins_list[i] + best_next_coin(i+1, j,
    gains_matrix, coins_list),
                                         coins_list[j] + best_next_coin(i, j-1,
    gains_matrix, coins_list))

    # La iteracion anterior llega hasta i=1 porque el cero del primer for (el que
    itera i) es excluyente. Por eso se hace este for aparte para abarcar el caso de
     la fila 0
    for j in range(0, coins_list_len):
        i = 0
        gains_matrix[i][j] = max(coins_list[i] + best_next_coin(i + 1, j,
    gains_matrix, coins_list),
                                 coins_list[j] + best_next_coin(i, j - 1, gains_matrix,
    coins_list))

    return gains_matrix
```

Este algoritmo utiliza una lógica Top Down para armar el espacio de soluciones requeridos para resolver el problema.

La matriz inicialmente tiene todos sus casilleros seteados en cero (0). La matriz se recorre de abajo hacia arriba y de izquierda a derecha, utilizando en cada paso la ecuación de recurrencia para obtener el óptimo local (notar que a diferencia de la solución greedy, acá la solución al problema sí tiene cierto nivel de memoria que usado para calcular los óptimos locales).

La función `get_gains_matriz()` se ocupa de rellenar la matriz como corresponde. Los for anidados permiten iterar sobre la matriz y calcular los óptimos para cada posición. La utilización de dos bloques de fors se debe particularidades de python en como genera los valores en un **range()**.

El primer bloque (el de los dos fors) itera desde la última fila a la segunda, y de izquierda a derecha. El segundo bloque itera sobre la primera fila, de izquierda a derecha.

La posición (0, n-1) contiene el óptimo buscado para esta versión del juego de las monedas.

Como comentario adicional, se menciona que en la matriz aquellas posiciones donde i ¿j tiene como valor óptimo cero (0) ya que no corresponden a un caso lógico del juego de las monedas.

Un ejemplo para una matriz de 4*4 es el siguiente:

13	14	15	16
0	10	11	12
0	0	7	8
0	0	0	4

Nuevamente, los valores de las monedas no afectan a los tiempos de ejecución del algoritmo (por el mismo motivo que el descrito en la sección de la solución greedy), como sí lo hacen la cantidad de monedas con las que se va a jugar. De hecho en esta ocasión el tiempo aumenta de forma cuadrática en base al tamaño del problema.

2.4. Ejemplos de ejecución

Como ejemplos de ejecución se utilizaron los datasets provistos por la cátedra para verificar el correcto funcionamiento del algoritmo.

- 5.txt espera ganancia de 1483 para Sophia.
- 10.txt espera ganancia de 2338 para Sophia.
- 20.txt espera ganancia de 5234 para Sophia.
- 25.txt espera ganancia de 7491 para Sophia.
- 50.txt espera una ganancia de 14976 para Sophia.
- 100.txt espera una ganancia de 28844 para Sophia.
- 1000.txt espera una ganancia de 1401590 para Sophia.
- 10000.txt espera una ganancia de 2869340 para Sophia.
- 20000.txt espera una ganancia de 34107537 para Sophia.

A priori, se puede decir que la solución implementada funciona correctamente y obtiene la mejor ganancia para Sophia.

2.5. Complejidad y medición

El bloque de los fors anidado tiene como complejidad algorítmica $O(N * (N - 1))$.

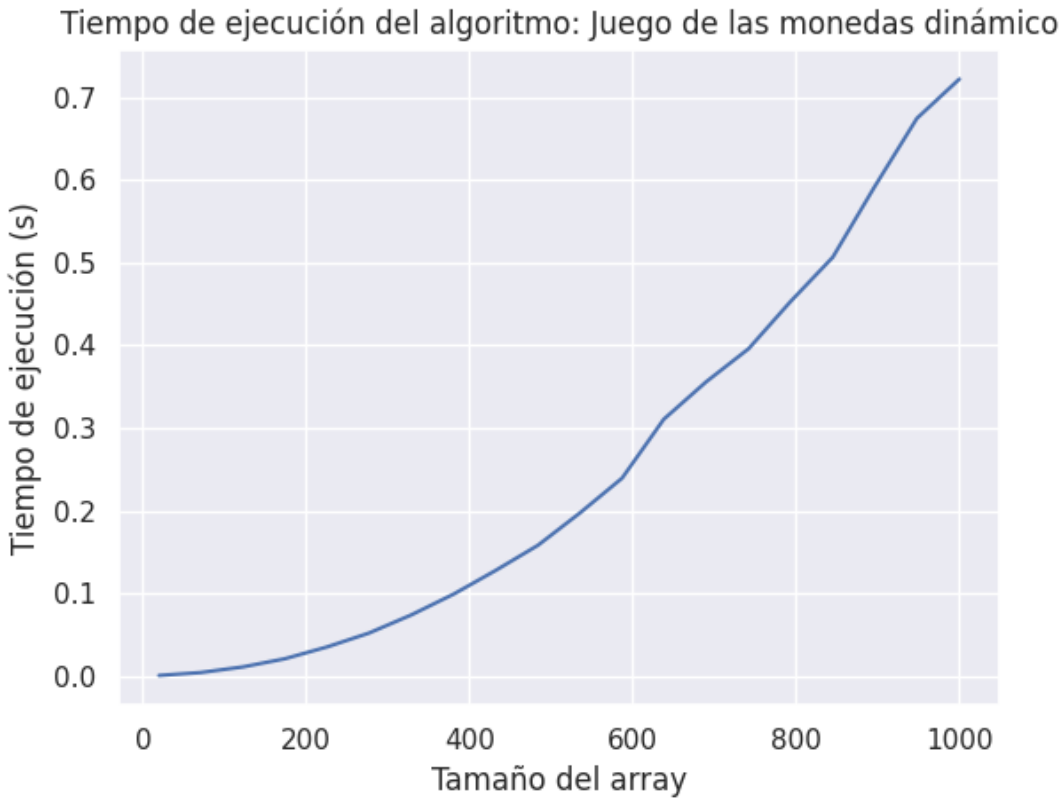
Por otro lado, el último for tiene como complejidad algorítmica $O(N)$

y el for de abajo tiene complejidad $O(N)$. Por lo que quedaría:

Por aproximación se simplifica a: $O(N^2) + O(N)$ y nuevamente se simplifica a $O(N^2)$

Por ende este algoritmo tiene complejidad: $O(N^2)$

A modo de validar la complejidad teórica se realizó el siguiente gráfico tiempo de ejecución VS tamaño del problema.



Lo cual es acorde con la hipótesis de la complejidad cuadrática.

2.6. Conclusiones y comentarios adicionales

La complejidad de esta sección fue mayor a la de la primera parte (complejidad de implementar una solución al problema). Fue necesario hacer una análisis exhaustivo del problema y de la forma en que se descomponía el problema en base a las monedas tomadas.

Inicialmente, se llevó la ecuación de recurrencia a un algoritmo mediante una lógica recursiva, que si bien era correcto lógicamente, en términos de eficiencia no lo era, ya que para tamaños muy grandes del problema se agotaba el stack asignado a la ejecución del algoritmo. Por lo tanto, se optó por una implementación iterativa.

Se comprobó la optimalidad de la implementación mediante la ejecución de los datasets provistos por la cátedra. Y se logró demostrar que la implementación de la solución es acorde con la complejidad algorítmica teórica.

3. Tercera parte: Cambios

3.1. Demostración de que el problema pertenece a NP

Para demostrar que el problema de la **Batalla Naval** pertenece a la clase **NP**, se debe probar que existe un *certificador* , es decir, una función que en tiempo polinomial pueda verificar que una salida del problema de la batalla naval es una solución válida.

3.1.1. Estructura del problema

El problema se define por los siguientes datos de entrada:

- **Tablero:** Una matriz $n \times m$ que representa el área donde se deben colocar los barcos.
- **Restricciones de filas:** Una lista de tamaño n que indica el número máximo de casillas que se pueden ocupar en cada fila.
- **Restricciones de columnas:** Una lista de tamaño m que indica el número máximo de casillas que se pueden ocupar en cada columna.
- **Barcos:** Una lista donde cada barco está definido por su tamaño. Los barcos deben colocarse sin superponerse ni estar adyacentes entre sí.

3.1.2. Proceso de verificación

El proceso de verificación consiste en recorrer el tablero propuesto y comprobar que cumple con todas las reglas del problema. Este proceso se detalla en los siguientes pasos:

1. **Validación de restricciones de filas y columnas:** Se recorre el tablero para contar el número de casillas ocupadas en cada fila y columna. Los valores obtenidos se comparan con las restricciones proporcionadas. **Complejidad:** $O(n \times m)$.
2. **Verificación de la validez de los barcos:** Durante el recorrido, si se encuentra una casilla ocupada, se verifica que forme parte de un barco válido. Esto incluye:
   - Comprobar que el barco tiene el tamaño especificado.
   - Asegurar que no hay superposición con otros barcos.
   - Verificar que no hay adyacencia con otros barcos (ni horizontal, ni vertical, ni diagonal).

   Estas validaciones locales tienen una complejidad de $O(1)$ por casilla.
3. **Comprobación de restricciones finales:** Al finalizar el recorrido, se asegura que las restricciones de las filas y columnas se cumplen exactamente. **Complejidad:** $O(n + m)$.

El pseudocódigo del proceso de verificación sería:

```
for i in range(n):
    for j in range(m):
        if tablero[i][j] == casilla_ocupada:
            casillas_por_fila[i] += 1
            casillas_por_columna[j] += 1

            if not cumple_restricciones_adyacencia(i, j):
                return False

if not cumple_restricciones_totales(casillas_por_fila, casillas_por_columna):
    return False

return True
```

3.1.3. Complejidad del algoritmo de verificación

La complejidad del proceso de verificación se descompone en los siguientes pasos:

- **Recorrer el tablero:** La operación de recorrer todas las casillas del tablero tiene una complejidad de $O(n \times m)$.
- **Validaciones locales:** Para cada casilla ocupada, se verifican la adyacencia y el tamaño del barco. Cada una de estas validaciones tiene una complejidad de $O(1)$ por casilla.
- **Verificación final de restricciones:** Al finalizar el recorrido, se comprueba que las restricciones de las filas y columnas se cumplen correctamente. Esta verificación tiene una complejidad de $O(n + m)$.

Por lo tanto, la complejidad total del proceso de verificación es:

$$O(n \times m) + O(n + m) + O(1) = O(n \times m) + O(n) + O(m) + O(1) = O(n \times m)$$

Dado que existe un certificador que puede verificar en tiempo polinómico, se concluye que el problema de la **Batalla Naval** pertenece a **NP**.

3.2. Demostración de que el problema es NP-Completo

Para demostrar que un problema pertenece a la clase de problemas NP-Completo, se deben cumplir con dos requisitos:

1. **Pertenencia a NP:** Mostrar que existe un certificador que pueda verificar una solución en tiempo polinómico. Esto fue demostrado en el punto anterior.
2. **Reducción desde un problema NP-Completo conocido:** Reducir un problema NP-Completo conocido al problema en cuestión, demostrando que si se puede resolver el segundo, también se puede resolver el primero.

3.2.1. Reducción desde un problema NP-Completo conocido

Para demostrar que el problema es NP-Completo, se reduce el problema **3-Partition**, que es NP-Completo, al problema de la Batalla Naval.

Descripción del problema 3-Partition: Dado un conjunto S de $3m$ números enteros positivos y un entero B, ¿es posible dividir S en m subconjuntos disjuntos de 3 elementos cada uno, tal que la suma de los elementos de cada subconjunto sea exactamente B?

Reducción: Dado una instancia del problema 3-Partition, creamos una instancia del problema de la Batalla Naval como sigue:

1. Generamos un tablero de dimensiones $n = m$ filas y m columnas.
2. La suma de cada subconjunto de 3 elementos en 3-Partition corresponde a un barco de tamaño B.
3. Las restricciones de filas y columnas aseguran que cada fila y cada columna contiene exactamente B casillas ocupadas, simulando las restricciones de suma del problema 3-Partition.
4. Garantizamos que los barcos no se superpongan ni se ubiquen adyacentes (ni horizontal, ni vertical, ni diagonalmente).

Si existe una solución válida para el problema de la Batalla Naval, esta corresponde a una partición válida en 3-Partition.

Complejidad de la reducción: La transformación de una instancia de 3-Partition al problema de la Batalla Naval se realiza en tiempo $O(n \times m)$, que es polinómico respecto al tamaño de S.

Demostramos que el problema de la Batalla Naval pertenece a NP, ya que una solución candidata puede verificarse en tiempo polinómico. Además, hemos presentado una reducción polinómica desde el problema 3-Partition, que es NP-Completo. Por lo tanto, concluimos que el problema de la Batalla Naval es NP-Completo.

3.3. Algoritmo Backtracking

El problema consiste en colocar una serie de barcos de longitudes específicas en un tablero de $n \times m$, cumpliendo con las restricciones de demanda de ocupación para cada fila y columna, mientras se minimiza la cantidad de demanda incumplida. La solución se implementa mediante un algoritmo de *backtracking* que explora todas las posibles configuraciones válidas para los barcos.

El algoritmo se divide en múltiples funciones que colaboran entre sí para construir la solución. A continuación, se detalla cada componente clave del algoritmo:

3.3.1. Generación de combinaciones de posiciones válidas

Para colocar un barco horizontalmente o verticalmente, se generan todas las combinaciones posibles que cumplan con las restricciones de ocupación de filas y columnas. Por ejemplo, la función `generate_rows_combinations` encuentra las posiciones horizontales válidas para un barco en el tablero.

```
def generate_rows_combinations(board, rows_restrictions, rows_occupation,
                               columns_restrictions, columns_occupation,
    current_ship_len):
    rows_combinations = []
    rows_amount = len(rows_restrictions)
    columns_amount = len(columns_restrictions)

    for i, row in enumerate(board):
        if (rows_occupation[i] + current_ship_len) > rows_restrictions[i]:
            continue

        adjacent_grids = 0
        for j in range(columns_amount):
            if (board[i][j] == 0) and (columns_occupation[j] < columns_restrictions
    [j]):
                adjacent_grids += 1
            else:
                adjacent_grids = 0

            if adjacent_grids == current_ship_len:
                ship_begin_position = j - current_ship_len + 1

                if ships_operations.is_valid_position(board, i, ship_begin_position
    , current_ship_len, HORIZONTAL_DIRECTION):
                    rows_combinations.append((i, ship_begin_position))
                adjacent_grids -= 1

    return rows_combinations
```

Esta función examina cada fila del tablero, verificando si es posible colocar un barco horizontalmente sin exceder las restricciones.

3.3.2. Intentos de colocación en direcciones horizontales y verticales

Una vez generadas las combinaciones, el algoritmo intenta colocar el barco en cada posición válida, evaluando el impacto en la solución global. Por ejemplo, la función `try_horizontal_combinations` intenta colocar el barco horizontalmente y actualiza el estado del tablero.

```python
def try_horizontal_combinations(
    ships, current_ship_index, board, rows_restrictions, columns_restrictions,
    rows_occupation, columns_occupation, local_solution, global_solution
):
    current_ship_len = ships[current_ship_index]
    ship_id = current_ship_index + 1  # IDs para distinguir a los barcos.
    best_board = copy.deepcopy(board)

    rows_combinations = generate_rows_combinations(
        board=board, rows_restrictions=rows_restrictions,
        rows_occupation=rows_occupation, columns_restrictions=columns_restrictions,
        columns_occupation=columns_occupation, current_ship_len=current_ship_len
    )

    for row, ship_begin_position in rows_combinations:
        can_put_ship = ships_operations.put_ship_on_row(
            game_board=board, current_row=row, current_column=ship_begin_position,
            ship_len=current_ship_len, ship_id=ship_id, rows_occupation=
    rows_occupation,
            columns_occupation=columns_occupation
        )
        if can_put_ship:
            local_solution_aux = local_solution + (ships[current_ship_index] * 2)

            board_aux, local_solution_aux = build_game_board_bt(
                ships=ships, current_ship_index=current_ship_index+1, game_board=
    board,
                rows_restrictions=rows_restrictions, columns_restrictions=
    columns_restrictions,
                rows_occupation=rows_occupation, columns_occupation=
    columns_occupation,
                local_solution=local_solution_aux, global_solution=global_solution
            )

            if local_solution_aux > global_solution:
                global_solution = local_solution_aux
                best_board = copy.deepcopy(board_aux)

            ships_operations.remove_ship_on_row(
                board, row, ship_begin_position, current_ship_len,
                rows_occupation, columns_occupation
            )

    return best_board, global_solution
```

La lógica es similar para las combinaciones verticales, utilizando una función separada `try_vertical_combinations`.

3.3.3. Función principal BT

La función `build_game_board_bt` se utiliza para resolver el problema de Batalla Naval utilizando un enfoque de backtracking. A continuación, se proporciona una explicación detallada de cada parte del código.

```python
def build_game_board_bt(ships, current_ship_index, game_board, rows_restrictions,
                        columns_restrictions, rows_occupation, columns_occupation,
                        local_solution, global_solution):

    best_game_board = copy.deepcopy(game_board)
```

La función `build_game_board_bt` se define con varios parámetros, entre ellos:

- `ships`: lista de barcos con sus longitudes.
- `current_ship_index`: índice del barco actual que se intenta colocar.
- `game_board`: tablero actual.

- `rows_restrictions`, `columns_restrictions`, `rows_occupation`, `columns_occupation`: restricciones y ocupación en filas y columnas.
- `local_solution`: solución parcial.
- `global_solution`: mejor solución global encontrada hasta ahora.

Al inicio, se hace una copia profunda del tablero `game_board` para trabajar con una copia sin modificar el original.

```
    if local_solution > global_solution:
        global_solution = local_solution

    if current_ship_index >= len(ships):
        return best_game_board, global_solution
```

Aquí, se verifica si la `local_solution` es mejor que la `global_solution`, y si es así, se actualiza la solución global. Además, si el índice del barco actual es mayor o igual al número total de barcos (`len(ships)`), significa que todos los barcos han sido colocados, por lo que se retorna la mejor solución.

```
    if local_solution + sum(ships[current_ship_index:]*2) <= global_solution:
        return best_game_board, global_solution
```

Este bloque evalúa si la `local_solution` más el "potencial"de los barcos restantes (el cálculo `sum(ships[current_ship_index:] * 2)`) no puede superar la solución global. Si esto es cierto, se retorna la mejor solución encontrada hasta ahora sin seguir intentando colocar más barcos.

```
best_game_board = copy.deepcopy(game_board)

game_board_aux, local_solution_aux = try_horizontal_combinations(
    ships=ships,
    current_ship_index=current_ship_index,
    board=game_board,
    rows_restrictions=rows_restrictions,
    columns_restrictions=columns_restrictions,
    rows_occupation=rows_occupation,
    columns_occupation=columns_occupation,
    local_solution=local_solution,
    global_solution=global_solution
)
```

Aquí se llama a la función `try_horizontal_combinations`, que intenta colocar el barco en combinaciones horizontales en el tablero. La función devuelve un tablero actualizado (`game_board_aux`) y una nueva solución parcial (`local_solution_aux`).

```
    if local_solution_aux > global_solution:
        global_solution = local_solution_aux
        best_game_board = copy.deepcopy(game_board_aux)
```

Si la solución obtenida con las combinaciones horizontales es mejor que la solución global, se actualiza `global_solution` y `best_game_board` con los valores obtenidos.

```
game_board_aux, local_solution_aux = try_vertical_combinations(
    ships=ships,
    current_ship_index=current_ship_index,
    board=game_board,
    rows_restrictions=rows_restrictions,
    columns_restrictions=columns_restrictions,
    rows_occupation=rows_occupation,
    columns_occupation=columns_occupation,
    local_solution=local_solution,
    global_solution=global_solution
)
```

De manera similar a las combinaciones horizontales, se llaman a las combinaciones verticales utilizando la función `try_vertical_combinations`. Esta función trata de colocar el barco de manera vertical en el tablero.

```
    if local_solution_aux > global_solution:
        global_solution = local_solution_aux
        best_game_board = copy.deepcopy(game_board_aux)
```

Si la solución obtenida con las combinaciones verticales es mejor que la global, se actualizan `global_solution` y `best_game_board`.

```
game_board_aux, local_solution_aux = build_game_board_bt(
    ships=ships,
    current_ship_index=current_ship_index+1,
    game_board=game_board,
    rows_restrictions=rows_restrictions,
    columns_restrictions=columns_restrictions,
    rows_occupation=rows_occupation,
    columns_occupation=columns_occupation,
    local_solution=local_solution,
    global_solution=global_solution
)
```

Si no se ha llegado a la solución final, se realiza una llamada recursiva a `build_game_board_bt`, pasando el siguiente barco (`current_ship_index + 1`). Esta recursión permite explorar todas las posibles configuraciones de los barcos en el tablero.

```
    if local_solution_aux > global_solution:
        global_solution = local_solution_aux
        best_game_board = copy.deepcopy(game_board_aux)
```

Tras la llamada recursiva, si la solución obtenida es mejor que la global, se actualizan `global_solution` y `best_game_board`.

```
    return best_game_board, global_solution
```

Finalmente, la función retorna el `best_game_board`, que contiene la mejor configuración encontrada, y la `global_solution`, que es la mejor solución global.

Construcción inicial del tablero Finalmente, la función `build_naval_battle_game_board` se encarga de iniciar el proceso, preparando las estructuras necesarias para el algoritmo de *backtracking*.

```
def build_naval_battle_game_board(rows_restrictions, columns_restrictions, ships ):
    game_board = [[0 for _ in range(len(columns_restrictions))] for _ in range(len(
    rows_restrictions))]
    rows_occupation = [0]*len(rows_restrictions)
    columns_occupation = [0]*len(columns_restrictions)
    ships.sort(reverse=True)

    return build_game_board_bt(ships,0, game_board, rows_restrictions,
    columns_restrictions,
                               rows_occupation, columns_occupation, 0, 0)
```

3.4. Mediciones

Para ver la efectividad del codigo procedemos a hacer algunas mediciones con los distintos datasets provistos por la catedra.

- El dataset `3_3_2.txt` se ejecuta en: 0,0001 segundos
- El dataset `5_5_6.txt` se ejecuta en: 0,0455 segundos
- El dataset `8_7_10.txt` se ejecuta en: 0,0045 segundos
- El dataset `10_3_3.txt` se ejecuta en: 0,0002 segundos
- El dataset `10_10_10.txt` se ejecuta en: 0,9951 segundos
- El dataset `12_12_21.txt` se ejecuta en: 1,9863 segundos
- El dataset `15_10_15.txt` se ejecuta en: 0,0109 segundos
- El dataset `20_20_20.txt` se ejecuta en: 1,1200 segundos
- El dataset `20_25_30.txt` se ejecuta en: 0,1037 segundos

Con estos tiempos podemos observar la efectivdad de nuestro codigo, ya que resuelve muy rapidamente el problema, siendo el dataset que mas se demora el `12_12_21.txt` con un tiempo de 1,9863 sgundos lo cual es muy poco

3.5. Algoritmo de aproximación

Implementamos un algoritmo de aproximación para resolver el problema de la Batalla Naval.

La complejidad del algoritmo implementado es $O(n.m.S)$, donde S es el número de barcos (ships).

Para analizar qué tan buena es la aproximación en cada instancia I del problema, calcularemos el factor de aproximación que cumple con

$$\frac{A(I)}{z(I)} \leq r(A)$$

para todas las instancias.

Comprobamos que $r(A) \leq r$, siendo $r = 1$ la constante que representa la cota de aproximación, en todas las instancias. Esto significa que el valor calculado de cada $r(A)$ oscila entre 0 y 1, debido a que si el valor es mayor que 1 significaría que el algoritmo de aproximación encontró mejores resultados que el algoritmo de backtracking.

Instancia	Óptimo	Aproximación	$r(A)$
3_3_2.txt	4	2	0,5
5_5_6.txt	12	8	0,67
8_7_10.txt	26	13	0,5
10_3_3.txt	6	6	1,0
10_10_10.txt	40	20	0,5
12_12_21.txt	46	19	0,41
15_10_15.txt	40	15	0,375
20_20_20.txt	104	42	0,40
20_25_30.txt	172	97	0,56
30_25_25.txt	202	89	0,44

Cuadro 1: Valores de r(A) para cada instancia del problema de Batalla Naval

Como podemos ver, para cada instancia I del problema, $r(A)$ varía entre los valores 0,375 y 1,0. Esto indica que el algoritmo de aproximación cumple con un rendimiento aceptable, pero no óptimo.

El peor caso de aroximación, es decir el valor mínimo encontrado, es el de la instancia de $15\_10\_15.txt$ con $r(A) = 0,375$.

Conclusiones

En esta parte del trabajo abordamos la resolución del problema de la Batalla Naval, un problema NP-completo, utilizando dos enfoques: un algoritmo exacto basado en *backtracking* y un algoritmo de aproximación. El *backtracking* demostró ser efectivo al encontrar soluciones óptimas, pero su escalabilidad se ve limitada en instancias más grandes debido a su alta complejidad. Los tiempos de ejecución obtenidos fueron razonables para los datasets analizados, siendo el más exigente el archivo `12_12_21.txt` con un tiempo de 1,9863 segundos.

El algoritmo de aproximación, por su parte, proporcionó soluciones más rápidas pero con menor precisión. Analizamos su rendimiento mediante el factor de aproximación $r(A)$, que osciló entre 0,375 y 1,0, cumpliendo con los límites establecidos. Aunque en algunos casos $r(A)$ mostró un rendimiento limitado, el método de aproximación se presenta como una alternativa válida cuando el tiempo es una restricción crítica.

En resumen, la comparación entre ambos enfoques resalta la necesidad de un balance entre precisión y eficiencia según el contexto. Este análisis demuestra cómo técnicas exactas y aproximadas pueden complementarse para abordar problemas complejos de manera efectiva.