Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3: Diversión NP-Completa

16 de enero de 2025

Santiago Nahuel Ruiz Sugliani, 106768

Marcos Bianchi Fernandez, 108921

Índice

1. Introducción

Dado que para Sophia y Mateo se tornó aburrido del juego de las monedas y con el correr de los años comenzaron tomarle el gusto a los juegos individuales, ahora Sophia encuentra entretenido el juego de ”La Batalla Naval Individual”.

En este mismo contamos con un tablero de n x m y una cantidad k de barcos de largos que varían pero no necesariamente son todos distintos. Dicho tablero tiene varias restricciones y debemos ubicar los barcos de modo que las cumplamos todas, ellas son:

- Cada fila y cada columna tiene una demanda de consumo a cumplir con los largos de los barcos ubicados en ellas.
- Dos barcos no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente.

En el presente informe brindaremos una serie de demostraciones y algoritmos que giran en torno a los siguientes puntos:

1. ¿Se encuentra el Problema de la Batalla Naval en NP?
2. ¿Es el Problema de la Batalla Naval un problema NP - Completo?
3. Algoritmo de Backtracking que resuelve el problema de optimización del Problema de la Batalla Naval.
4. Modelo de programación lineal (y su implementación) que resuelve el Problema de la Batalla Naval de forma óptima.
5. Análisis de un algoritmo de aproximación.
6. Análisis de un algoritmo greedy.

2. Demostración de que es un problema NP-Completo

2.1. Demostración de que está en NP

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

El validador eficiente de $BatallaNaval$ es el siguiente:

```python
from collections import Counter


def conseguir_barcos(tablero):
    """
    La complejidad del algoritmo es O(n * m)
    """
    # O(n * m)
    tablero = tablero.copy()
    n, m = len(tablero), len(tablero[0])

    def en_rango(casillero):
        i, j = casillero
        return 0 <= i and 0 <= j and i < n and j < m

    def siguientes(i, j):
        yield i, j + 1
        yield i + 1, j

    def tomar_barco(i, j):
        if tablero[i][j] == 0:
            return 0

        largo = 0
        tablero[i][j] = 0
        for i, j in filter(en_rango, siguientes(i, j)):
            largo += tomar_barco(i, j)

        return largo + 1

    # O(n * m)
    barcos = []
    for i in range(n):
        for j in range(m):
            if tablero[i][j] == 1:
                largo = tomar_barco(i, j)
                barcos.append(largo)

    return barcos


def validador_batalla_naval(filas, columnas, b, solucion):
    """
    La complejidad del validador es O(n * m + k)
    """
    n, m = len(filas), len(columnas)

    # Validar que el tablero soluci n tiene la misma
    # cantidad de filas que el original.
    if len(solucion) != n:
        return False

    # O(n)
    #
    # Validar que el tablero soluci n tiene la misma
    # cantidad de columnas que el original en cada fila.
```

```
    for i in range(n):
        if m != len(solucion[i]):
            return False

    # O(n * m)
    #
    # Validar que se cumplen las restricciones por fila.
    for i in range(n):
        if sum(solucion[i]) != filas[i]:
            return False

    # O(n * m)
    #
    # Validar que se cumplen las restricciones por columna.
    for j in range(m):
        if sum(solucion[i][j] for i in range(n)) != columnas[j]:
            return False

    def en_rango(i, j):
        return 0 <= i and 0 <= j and i < n and j < m

    def giros(i, j):
        yield (i, j - 1), (i - 1, j)
        yield (i - 1, j), (i, j + 1)
        yield (i, j + 1), (i + 1, j)
        yield (i + 1, j), (i, j - 1)

    def hay_giro(i, j):
        for (Pi, Pj), (Qi, Qj) in giros(i, j):
            if not en_rango(Pi, Pj) or not en_rango(Qi, Qj):
                continue

            if solucion[Pi][Pj] + solucion[Qi][Qj] == 2:
                return True

        return False

    def hay_diagonal(i, j):
        for Di in [i - 1, i + 1]:
            for Dj in [j - 1, j + 1]:
                if en_rango(Di, Dj) and solucion[Di][Dj] == 1:
                    return True

        return False

    # O(n * m)
    #
    # Validar que no haya adyacencias.
    for i in range(n):
        for j in range(m):
            if solucion[i][j] == 1:
                if hay_giro(i, j) or hay_diagonal(i, j):
                    return False

    # O(n * m)
    barcos = conseguir_barcos(solucion)

    # Validar que no hay m s barcos en la soluci n
    # que en el arreglo de barcos.
    if len(b) < len(barcos):
        return False

    # O(k)
    #
    # Validar que hay a lo sumo las m sma cantidad
    # de barcos para cada largo como en el arreglo
    # de barcos.
    largos = Counter(b)
    for l, q in Counter(barcos).items():
        if l not in largos:
```

```
            return False

        if largos[l] < q:
            return False

    return True
```

Para este validador, obtuvimos una complejidad de $O(n \cdot m + k)$ donde n y m son la cantidad de filas y columnas respectivamente, y k el tamaño del arreglo b de barcos, al ser una complejidad polinómica, entonces demostramos que $BatallaNaval \in$ NP.

El validador chequea que el tablero solución tenga la misma dimensionalidad que el tablero original $O(n)$. Chequear que las restricciones se cumplen exactamente $O(n \cdot m)$. Para las adyacencias entre barcos se fija que no existan giros o diagonales, esto es una operación de tiempo constante, aplicado a cada uno de los casilleros toma tiempo $O(n \cdot m)$. Luego consigue los barcos mediante un algoritmo que toma tiempo $O(n \cdot m)$ que utiliza backtracking donde chequea casillero por casillero y por cada barco encontrado lo quita el tablero y lo guarda en un arreglo. Luego chequea que los barcos utilizados no hayan sido más que los originalmente planeados. Por último, valida que por cada largo de barco original haya a lo sumo esa cantidad $O(k)$.

2.2. Reducción

Para demostrar que $BatallaNaval \in$ NP-Completo, vamos a utilizar el problema NP-Completo 3-$Partition$ con números unarios. Este problema tiene una fuerte correlación con $BatallaNaval$ porque ambos se basan en insertar elementos en un tablero/conjunto con algún tipo de restricción.

El problema de decisión de 3-$Partition$ con números unarios es: Dado un multiset S que contiene números enteros unarios, ¿És posible asignar los elementos de S a 3 subconjuntos disjuntos tal que la suma de los valores de todos los subconjuntos sea la mísma?

La reducción a realizar es la de 3-$Partition \leq_p BatallaNaval$. Como sabemos que 3-$Partition$ $\in$ NP-Completo entonces vamos a demostrar que $BatallaNaval$ es por lo menos tán difícil que 3-$Partition$, como ya demostramos que $BatallaNaval \in$ NP, entonces quedaría demostrado que $BatallaNaval \in$ NP-Completo.

2.2.1. Transformación

Proponemos la siguiente transformación:

Teniendo en cuenta que el problema de 3-$Partition$ tiene como entrada el multiset $S = \{a_1, a_2, \ldots, a_k\}$ y que la entrada a $BatallaNaval$ es un arreglo de restricciones por fila, otro por columnas y un multiset de largos de barcos.

Podemos procesar S para generar el multiset de barcos b, de forma que los barcos a asignar sean los valores dentro del multiset original. Las restricciones por fila son tal que por cada a_i creamos una restricción de valor 0 y a_i filas de valor 1 de forma contigua. Las restricciones por columna son siempre $[p, 0, p, 0, p]$ tal que $p = \frac{1}{3}\sum_{x \in S} x$. Por la naturaleza del problema de 3-$Partition$, suponemos que p es entero pues $\sum_{x \in S} x$ debe ser múltiplo de 3, de no serlo sabemos de antemano que no hay solución a 3-$Partition$ y no tiene sentido seguir con el planteo.

Al final de la transformación obtenemos un multiset de largos de barcos, un arreglo de restricciones por filas de largo $k + \sum_{x \in S} x$ y un arreglo de restricciones por columnas de largo 6. Es importante destacar que esta transformación se logra en tiempo polinomial puesto que podemos formar b en $O(k)$, $filas$ en $O(k \cdot \sum_{x \in S} len(x))$ y $columnas$ en $O(1)$.

Un ejemplo es: $S = \{1, 11, 111\}$, obtenemos $b = \{1, 2, 3\}$, $filas = [0, 1, 0, 1, 1, 0, 1, 1, 1]$, $columnas = [0, 2, 0, 2, 0, 2]$.

Al utilizar valores unarios, la complejidad de la tranformación crece de forma lineal con la cantidad de 1s en los números. En otro caso la tranformación sería pseudo-polinomial y no sería válida para demostrar que 3-$Partition \leq_p BatallaNaval$.

Por como formulamos la transformación, garantizamos que todos los barcos se posicionen de forma vertical en la segunda, cuarta o última columna. Sabemos que siempre va a haber lugar para todos y que las restricciones por fila existen solo para que ningún barco comparta fila con otro.

Lo que determina si el problema es resoluble es si se cumplen las restricciones por columna, más especificamente las columnas 1, 3 y 5. Estas columnas representan los tres subconjuntos de partición y cada barco tiene lugar para estar en solo una de ellas gracias a que las restricciones por fila tienen un máximo de 1.

2.2.2. Doble Implicancia

$$3\text{-}Partition \implies BatallaNaval$$

Sabiendo que 3-$Partition$ es resoluble para alguna instancia del problema, un cierto S, entonces sabemos que existen tres subconjuntos S_1, S_2 y S_3 tal que todos ellos suman el mismo valor. Esto quiere decir que en nuestro modelo hay una partición en tres subconjuntos de b, b_1, b_2 y b_3 tal que $\sum_{b \in b_1} b = \sum_{b \in b_2} b = \sum_{b \in b_3} b$.

Teniendo en cuenta el paralelismo entre los valores de S y los barcos, sabemos que $\sum_{x \in S_1} x = \sum_{x \in S_2} x = \sum_{x \in S_3} x = \frac{1}{3} \sum_{x \in S} x = p$, que es el mísmo valor que toman las restricciones de columnas 1, 3 y 5.

Sabiendo que existe 1 posicion para cada uno de los k barcos de forma que no son adyacentes y cumplen todas las restricciones de filas, podemos garantizar que podemos posicionar algunos barcos en la segunda columna, otros en la cuarta y los que quedan en la última de forma que la suma en dichas columnas sea p. De esta forma se cumplen las todas las restricciones del problema, entonces se cumple $BatallaNaval$.

$$BatallaNaval \implies 3\text{-}Partition$$

Sabiendo que se cumplen todas las restricciones del tablero y no hay barcos adyacentes en nuestro tablero modelo. Existen posiciones para los k barcos tal que la suma de los largos de las columnas 1, 3 y 5 es la mísma, esta restricción tiene el valor de $p = \frac{1}{3} \sum_{x \in S}^{x}$, entonces existe una 3-partición de barcos b_1, b_2 y b_3 tal que $\sum_{b \in b_1} b = \sum_{b \in b_2} b = \sum_{b \in b_3} b = p$. Como nosotros modelamos a los elementos de S como los barcos, entonces se dá 3-$Partition$ para el multiset original S.

De esta forma queda demostrado que $BatallaNaval \in$ NP-Completo.

3. Algoritmo de Backtracking

3.1. Análisis

Con este algoritmo solucionamos de forma óptima el problema en su forma de optimización. Donde buscamos minimizar la cantidad de restricciones incumplidas utilizando a lo sumo todos los barcos que nos son disponibles.

Este algoritmo va a probar muchas de las configuraciones de barcos y se va a quedar con la que cumpla la mayor cantidad de restricciones posibles.

Primero ordena los barcos de forma descendente, para acotar las busquedas tempranas y mejorar las podas que describimos más abajo.

Por cada uno de los barcos va a recorrer el tablero posición por posición encontrando los posibles candidatos a posicionar dicho barco. Por cada uno de ellos va a insertarlo y probar con el siguiente, todo esto considerando que tal vez alguno de los barcos anteriores podría estar afectando negativamente futuras posiciones. Por lo que va a volver y reintentar en distintas configuraciones para cada uno de los barcos.

3.2. Análisis de complejidad

Como precomputo va a calcular el total de restricciones por fila y columna, copiar las restricciones para no modificar las originales y ordenar el arreglo de barcos, todo esto se logra en tiempo $O(n + m + k \cdot logk)$.

Luego va a crear un tablero vacio $O(n \cdot m)$ y obtener una solución aproximada utilizando el algoritmo de aproximación de John Jellicoe, cuya complejidad es $O(n \cdot m + (n+m) \cdot (log(n+m) + k \cdot (max(n,m) \cdot l - l^2)) + k \cdot logk)$ donde l es el promedio de largos de los barcos.

Ahora sí, arranca la parte exponencial del algoritmo, donde va a intentar colocar cada uno de los barcos en todas sus posibles posiciones, estamos hablando de una operación de tiempo $O(2^k \cdot n \cdot m \cdot l)$ puesto que a lo sumo las posibles posiciones son $n \cdot m$ y chequear si es posible insertar un barco en cada una de ellas e insertarlo toma tiempo $O(l)$.

En conclusión este algoritmo toma tiempo $O(2^k \cdot n \cdot m \cdot l)$ donde l es el promedio de largos de los barcos. Los demás términos los podemos desechar gracias a que este contiene todas las variables y crece mucho más rápio que los demás.

3.3. Podas

Implementamos 2 podas que ayudan a acelerar la terminación del algoritmo.

Por un lado a la hora de insertar un barco primero chequea que sea posible, por lo que si no lo fuera esta llamada no se hace y se considera posicionar el barco en otro casillero.

Por el otro lado en cada invocación calculamos la suma de barcos que faltan considerar. Si la suma de restricciones cumplidas hasta el momento y barcos restantes es menor o igual al valor óptimo actual, entonces esta rama se desecha, pues, es imposible conseguir un mejor resultado que el que ya tiene.

Esta segunda poda es todavía mejor puesto que con anterioridad ordenamos los barcos de forma descendente. Si quedara siempre un barco gigante por considerar entonces no sería muy buena poda, puesto que siempre quedaría la chance de considerar este barco que potencialmente cumpliría muchas restricciones.

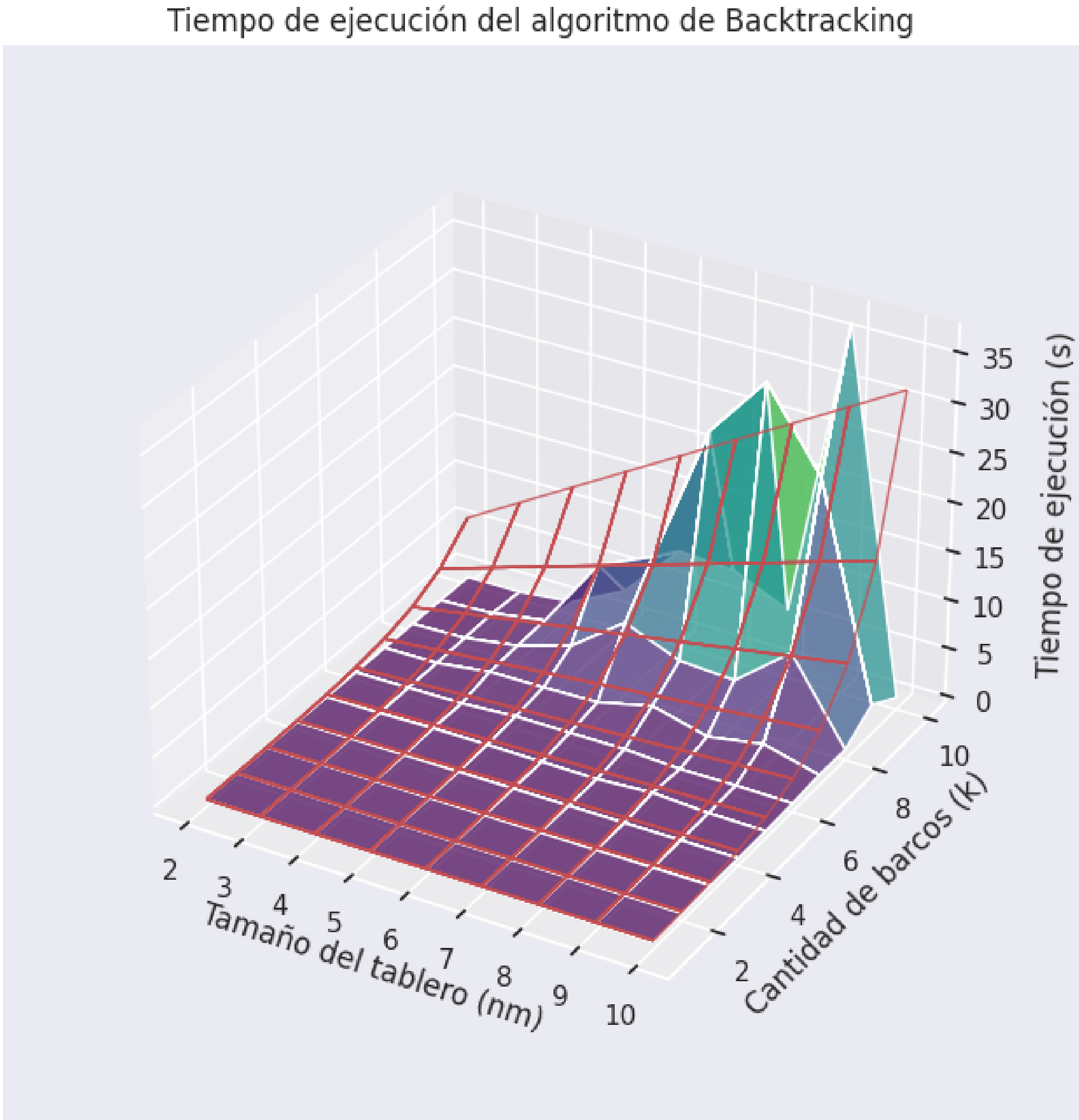
En vez de calcular esta suma en cada incovación, podríamos haber calculado las sumas de sufijos con anterioridad, osea, la suma de los largos de los barcos a la derecha del actual. Por lo que acceder a este dato se consigue en tiempo constante en vez de lineal en la cantiad de barcos.

3.4. Implementación

```python
from copy import deepcopy
from codigo.aprox import batalla_naval as batalla_naval_aprox


def batalla_naval(filas, columnas, b):
    """
    La complejidad del algoritmo es O(2^k * n * m * l)
    """
    # O(n + m + klogk)
    n, m = len(filas), len(columnas)
    total = sum(filas) + sum(columnas)
    filas, columnas = filas.copy(), columnas.copy()
    b = sorted(b, reverse=True)

    def en_rango(i, j):
        return 0 <= i and 0 <= j and i < n and j < m

    def insertar_barco_horizontal(i, j, tablero, l):
        """
        La complejidad es O(l)
        """
        if j + l > m:
            return False

        if l > filas[i]:
            return False

        # O(l)
        if any(columnas[j + k] == 0 for k in range(l)):
            return False

        # O(l)
        for Ci in range(i - 1, i + 2):
            for Cj in range(j - 1, j + l + 1):
                if en_rango(Ci, Cj) and tablero[Ci][Cj] == 1:
                    return False

        # O(l)
        filas[i] -= l
        for k in range(l):
            tablero[i][j + k] = 1
            columnas[j + k] -= 1

        return True

    def insertar_barco_vertical(i, j, tablero, l):
        """
        La complejidad es O(l)
        """
        if i + l > n:
            return False

        if l > columnas[j]:
            return False

        # O(l)
        if any(filas[i + k] == 0 for k in range(l)):
            return False

        # O(l)
        for Ci in range(i - 1, i + l + 1):
            for Cj in range(j - 1, j + 2):
                if en_rango(Ci, Cj) and tablero[Ci][Cj] == 1:
                    return False

        # O(l)
        columnas[j] -= l
        for k in range(l):
```

```python
            tablero[i + k][j] = 1
            filas[i + k] -= 1

        return True

    def quitar_barco_horizontal(i, j, tablero, l):
        """
        La complejidad es O(l)
        """
        # O(l)
        filas[i] += l
        for p in range(l):
            tablero[i][j + p] = 0
            columnas[j + p] += 1

    def quitar_barco_vertical(i, j, tablero, l):
        """
        La complejidad es O(l)
        """
        # O(l)
        columnas[j] += l
        for p in range(l):
            tablero[i + p][j] = 0
            filas[i + p] += 1

    def bt(r, opt, opt_acc, parcial, acc):
        """
        La complejidad es O(2^k * n * m * l)
        """
        if r == len(b):
            # O(n * m)
            return (deepcopy(parcial), acc) if acc > opt_acc else (opt, opt_acc)

        # O(k)
        if acc + 2 * sum(b[r:]) <= opt_acc:
            return opt, opt_acc

        # O(n * m * l)
        l = b[r]
        for i in range(n):
            for j in range(m):
                if insertar_barco_horizontal(i, j, parcial, l):
                    opt, opt_acc = bt(r + 1, opt, opt_acc, parcial, acc + 2 * l)
                    if opt_acc == total:
                        return opt, opt_acc

                    quitar_barco_horizontal(i, j, parcial, l)

                if insertar_barco_vertical(i, j, parcial, l):
                    opt, opt_acc = bt(r + 1, opt, opt_acc, parcial, acc + 2 * l)
                    if opt_acc == total:
                        return opt, opt_acc

                    quitar_barco_vertical(i, j, parcial, l)

        return bt(r + 1, opt, opt_acc, parcial, acc)

    # O(2^k * n * m * l)
    vacio = [[0] * m for _ in range(n)]
    aprox, cumplido = batalla_naval_aprox(filas, columnas, b)
    return bt(0, aprox, cumplido, vacio, 0)
```

3.5. Mediciones



Como se puede observar, la complejidad temporal del algoritmo crece muy rapidamente, especialmente a medida que incrementa la cantidad de barcos, tomando en cuenta que su complejidad es $O(2^k \cdot n \cdot m \cdot l)$ esto tiene mucho sentido.

4. Modelo de Programación Entera

4.1. Análisis

Con este algoritmo solucionamos otra vez el místmo problema de optimización pero esta vez usando Programación Entera.

Comencemos estableciendo las variables de nuestro problema:

- $Y_{i,j}$: Por cada casillero del tablero tendremos una variable de tipo binario donde los valores 1 y 0 representarán si la misma está siendo ocupada por parte de un barco o no respectivamente.
- $V_{i,j,l}$: Por cada posición del tablero donde podamos ubicar el inicio de un barco de un cierto largo de forma vertical existe una variable binaria que representa si un barco será o no tomado parte de la solución.
- $H_{i,j,l}$: Por cada posición del tablero donde podamos ubicar el inicio de un barco de un cierto largo de forma horizontal existe una variable binaria que representa si un barco será o no tomado parte de la solución.
- *Observación*: Para los barcos de largo 1, solo creamos una de las variables, pues insertarlo de forma vertical u horizontal es lo místmo.

Tomemos la siguiente notación:

- $filas$: Las restricciones por filas.
- $columnas$: Las restricciones por columnas.
- $largos$: Un diccionario que guarda la cantidad de barcos disponibles de un cierto largo.
- $ady_o(i, j, l)$: Un arreglo de posiciones adyacentes a un barco posicionado en la posición (i, j), de largo l y orientación o.

Ahora avancemos con las restricciones que va a imponer nuestro sistema de ecuaciones e inecuaciones lineales sobre los valores de las variables descritas anteriormente:

- Recordemos que cada fila tiene una demanda la cual no podemos exceder, por lo tanto:

$$\forall i \in [0, n-1], \quad \sum_{j=0}^{m-1} Y_{i,j} \leq filas[i]$$

- Análogamente al ítem anterior pero ahora sobre las columnas, cada columna tiene una demanda la cual no podemos exceder, por lo tanto:

$$\forall j \in [0, m-1], \quad \sum_{i=0}^{n-1} Y_{i,j} \leq columnas[j]$$

- La cantidad de barcos de un largo l ubicados no puede ser mayor a la cantidad de barcos disponible de dicho largo:

$$\forall l \in largos, \quad \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} V_{i,j,l} + \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} H_{i,j,l} \leq largos[l]$$

- Para cada una de las variables de barcos posibles, sus posiciones adyacentes deben encontrarse vacías.

$$\forall (i, j, l) \in [0, n-1] \times [0, m-1] \times largos$$
$$V_{i,j,l} \cdot (l + |ady_v(i, j, l)|) \leq \sum_{k=0}^{l-1} Y_{i+k,j} + \sum_{(i',j') \in ady_v(i,j,l)} (1 - Y_{i',j'})$$
$$H_{i,j,l} \cdot (l + |ady_h(i, j, l)|) \leq \sum_{k=0}^{l-1} Y_{i,j+k} + \sum_{(i',j') \in ady_h(i,j,l)} (1 - Y_{i',j'})$$

Ahora resta proponer una función objetivo a maximizar o minimizar para luego aplicar un algoritmo que resuelva el modelo, la función objetivo para este caso será:

$$MAX \quad \left(2 \cdot \sum_{l \in largos} l \cdot \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (V_{i,j,l} + H_{i,j,l}) \right)$$

Notar que el problema pide minimizar la cantidad de restricciones instatisfechas, nosotros estamos maximizando la cantidad de barcos multiplicado por su largo. Multiplicando por 2 logramos que la función objetivo valga la cantidad de restricciones satisfechas, pues, la suma de los barcos multiplicado por su largo, será la suma de los largos de los barcos solución, que es igual a la cantidad de restricción satisfecha dividido 2. Entonces maximizar esto es equivalente a minimizar la negación.

Finalmente devolvemos la solución óptima observando para qué variables $V_{i,j,l}$ y $H_{i,j,l}$ se ubica el principio de un barco y en función de si la variable representa una distribución vertical u horizontal de un largo l, se completan las posiciones correspondientes, también devolvemos la demanda total cumplida (en filas y en columnas).

La cantidad de variables y restricciones crece rapidamente con la cantidad de restricciones de filas, columnas y los distintos largos disponibles. La cantidad de barcos no modifica la cantidad de variables ni restricciones.

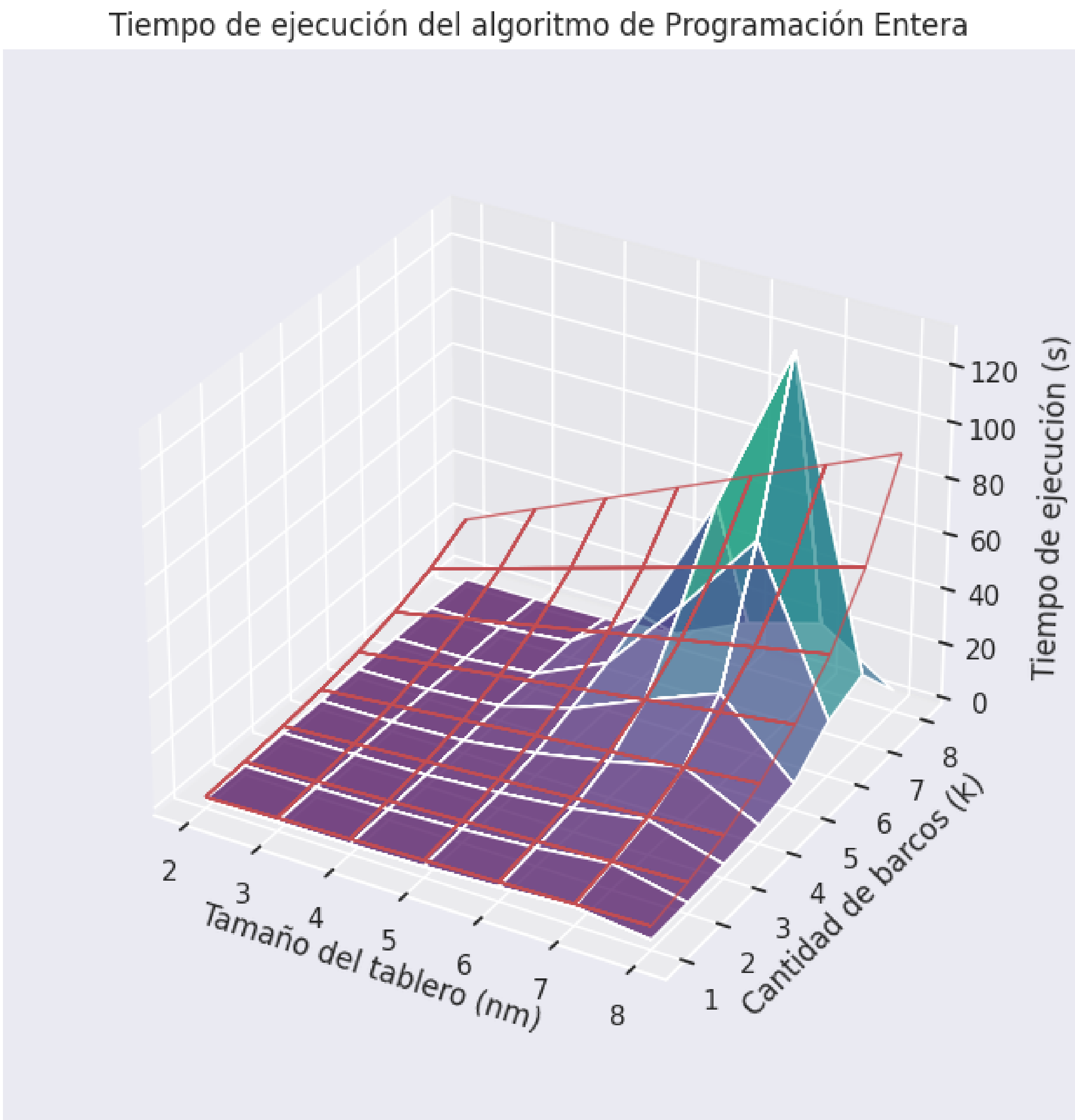
4.2. Implementación

```
from pulp import LpProblem, LpMaximize, LpVariable, lpSum, value, PULP_CBC_CMD
from collections import Counter


def batalla_naval(filas, columnas, b):
    n, m = len(filas), len(columnas)
    largos = Counter(b)  # O(k)

    def en_rango(casillero):
        i, j = casillero
        return 0 <= i and 0 <= j and i < n and j < m

    def adyacentes_hor(i, j, l):
        for k in range(-1, l + 1):
            yield i - 1, j + k
            yield i + 1, j + k

        yield i, j - 1
        yield i, j + l

    def adyacentes_ver(i, j, l):
        for k in range(-1, l + 1):
            yield i + k, j - 1
            yield i + k, j + 1

        yield i - 1, j
```

```
            yield i + l, j

    def entra_horizontal(i, j, l):
        """
        La complejidad es O(1)
        """
        if l > filas[i]:
            return False

        if j + l > m:
            return False

        # O(1)
        if any(columnas[j + k] == 0 for k in range(l)):
            return False

        return True

    def entra_vertical(i, j, l):
        """
        La complejidad es O(1)
        """
        if l > columnas[j]:
            return False

        if i + l > n:
            return False

        # O(1)
        if any(filas[i + k] == 0 for k in range(l)):
            return False

        return True

    prob = LpProblem("Batalla_Naval", LpMaximize)

    # O(n * m)
    ys = [[LpVariable(f"{i}_{j}", cat="Binary")
           for j in range(m)] for i in range(n)]

    # O(n * m * l)
    bs = {l: [] for l in largos}

    for i in range(n):
        for j in range(m):
            for l in largos:
                if entra_vertical(i, j, l):
                    v = LpVariable(f"V_{i}_{j}_{l}", cat="Binary")
                    ady = list(filter(en_rango, adyacentes_ver(i, j, l)))
                    sum_ocu = lpSum(ys[i + k][j] for k in range(l))
                    sum_ady = lpSum(1 - ys[Ai][Aj] for Ai, Aj in ady)
                    prob += v * (l + len(ady)) <= sum_ocu + sum_ady
                    bs[l].append(v)

                # No repetir los de largo 1.
                if l > 1 and entra_horizontal(i, j, l):
                    h = LpVariable(f"H_{i}_{j}_{l}", cat="Binary")
                    ady = list(filter(en_rango, adyacentes_hor(i, j, l)))
                    sum_ocu = lpSum(ys[i][j + k] for k in range(l))
                    sum_ady = lpSum(1 - ys[Ai][Aj] for Ai, Aj in ady)
                    prob += h * (l + len(ady)) <= sum_ocu + sum_ady
                    bs[l].append(h)

    # O(1)
    for l, barcos in bs.items():
        prob += lpSum(barcos) <= largos[l]

    # O(n)
    for i in range(n):
        prob += lpSum(ys[i]) <= filas[i]
```

```python

    # O(m)
    for j in range(m):
        prob += lpSum(ys[i][j] for i in range(n)) <= columnas[j]

    prob += 2 * lpSum(l * lpSum(bs[l]) for l in largos)
    prob.solve(PULP_CBC_CMD(msg=False))

    # O(n * m)
    tablero = [[0] * m for _ in range(n)]

    cumplido = 0  # O(n * m * l)
    for l in filter(lambda l: largos[l] > 0, bs):
        for barco in filter(lambda b: bool(value(b)), bs[l]):
            largos[l] -= 1

            orientacion, i, j, l = barco.name.split("_")
            i, j, l = int(i), int(j), int(l)
            if orientacion == "V":
                for i in range(i, i + l):
                    tablero[i][j] = 1

            elif orientacion == "H":
                for j in range(j, j + l):
                    tablero[i][j] = 1

            cumplido += 2 * l

    return tablero, cumplido
```

4.3. Mediciones



Nuevamente, podemos observar como el tiempo crece rapidamente con la cantidad de barcos, este algoritmo tiende a tardar más que el de Backtracking.

5. Algoritmo de aproximación

5.1. Propuesta de John Jellicoe

Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.

5.2. Análisis de complejidad

Primero creamos el tablero y copiamos las filas y columnas, esto tiene una complejidad $O(n \cdot m)$. Luego creamos el arreglo de restricciones y lo hacemos heap $O(n+m)$. Tomamos los distintos largos de los barcos y los ordenamos $O(k \cdot log(k))$. Dentro del loop principal, vamos a por cada restricción hacer un *heappop* $O(log(n+m))$ e intentar insertar los barcos de mayor tamaño primero dentro de la fila/columna $O(k)$, la complejidad de chequear si entra e insertarlo es $O(max(n,m) \cdot l - l^2)$ puesto que si el largo del barco fuera de 1 habrían n posiciones en una columna y m posiciones en una fila, tomando un largo arbitrario, obtenemos que existen $n-l+1$ formas de ubicar un barco de largo l en una columna y $m-l+1$ en una fila. Como la operación de inserción puede ser en una fila o columna, nos quedamos con la más grande, osea $max(n,m)$. Seguido a esto, chequear cada una de las $max(n,m)-l+1$ posiciones toma tiempo $O(l)$. Esta operación se realiza por cada barco, por lo que en total tiene una complejidad de $O(k \cdot (max(n,m) \cdot l - l^2))$. En caso de que el barco insertado no consuma toda la restricción, se vuelve a insertar la restricción actualizada dentro del heap $O(log(n+m))$.

La complejidad del algoritmo es $O(n \cdot m + (n+m) \cdot (log(n+m) + k \cdot (max(n,m) \cdot l - l^2)) + k \cdot logk)$ donde l es el valor promedio de largos de los barcos.

5.3. Análisis de aproximación

Repasemos los pasos del algoritmo para estimar su cota de aproximación.

1. Buscar la fila/columna de mayor demanda.
2. Probar con todos los barcos disponibles hasta insertar alguno de forma descendiente.
3. Volver a 1 hasta que no queden más barcos o nos hayamos quedado sin restricciones.

Donde más pierde este algoritmo es cuando estando en la "mejor"fila/columna para poner un barco, terminamos poniendo el barco más chico posible. Pues, el algoritmo cree que en esta fila/columna es donde va a tener mayor ganancia. Con "mejor"nos referimos a la fila/columna con mayor demanda.

Entonces, imaginemos un escenario donde siempre tomemos el barco más chico para llenar la fila/columna de mayor demanda disponible. La peor manera de llenar dicha fila/columna es con barcos de largo 1, de esta forma tendríamos $\frac{l}{2}$ casilleros ocupados en dicha fila/columna donde su valor es de l, pues las demandas cumplidas son por filas y por columnas, entonces cuentan doble.

De esta forma no solo llenamos esta fila/columna de la peor manera, sino que ocupamos las filas/columnas adyacentes y ahora no es posible insertar ningún barco en ellas. Notar que la demanda de dichas filas/columnas adyacentes tiene que ser necesariamente menor o igual a la que llenamos anteriormente, sino hubieran sido elegidas antes por el algoritmo.

Pero y si no solo inhabilitara a los casilleros adyacentes, sino que todavía peor, inhabilitara a toda la fila/columna en la que está posisionado. Esto lo podemos lograr en ciertas configuraciones de tableros y barcos. Por ejemplo la siguente:

- filas = [8, 1, 8, 1, 8]

- columnas = [100, 3, 3, 3, 3, 3, 3, 3, 3]
- b = [8, 8, 8, 1, 1, 1]

Con un poco de imaginación podemos ver como entran de forma horizontal los 3 barcos de largo 8 en las filas impares logrando un valor óptimo de 48.

Nuestro algoritmo va a primero encontrarse con la primer columna, pues tiene restricción de mayor demanda. Va a iterar los barcos de mayor a menor intentando insertarlos en vertical, como el tablero tiene una altura menor a 8, va a insertar un barco de largo 1 en la posición $(0, 0)$ y bajará su demanda a 99. La misma columna es todavía la de mayor demanda, por lo que se vuelve a intentar con otro barco de largo 1, se termina insertando en $(0, 2)$. Vuelve a pasar con el último barco de 1 que se inserta en $(0, 5)$.

Una vez que no pudimos vovler a insertar un barco en esta restricción, se da por terminada, no podemos mejorar su demanda incumplida. Entonces la proxima restricción será alguna de las de 8, pero, ya no tenemos barcos que entren de ninguna forma en ningún casillero del tablero. Por lo que termina el algoritmo.

Logramos un valor de demanda cumplida de 6. Realizando el calculo de relación de aproximación obtenemos la cota inferior de $\frac{z(I)}{A(I)} = \frac{48}{6} = 8 \leq r(I)$. Por lo que demostramos que este algoritmo es por lo menos una 8-*aproximación*.

Poniendonos un poco más formales, podemos probar lo mísmo de una forma más general. Tomemos el mísmo ejemplo, pero ahora generalizemoslo.

- filas = $[m, 1]\cdot \frac{n}{2}$
- columnas = $[c_1, c_2, ..., c_m] : c_1 - \frac{n}{2} \geq c_i \forall i \in [2, m]$
- b = $[m, 1] \cdot \frac{n}{2}$

En este caso vamos a tomar a $n = min(n, m)$ y $m = max(n, m)$, si esto no se cumpliera, podemos rotar el tablero, puseto que son casos análogos, también vamos a querer garantizar $n < m$.

De esta forma vamos a poder colocar $\frac{n}{2}$ barcos de largo 1. Tomando el ejemplo anterior, colocamos en la mitad de las filas un barco, entonces, en este caso también. De esta forma conseguimos un puntaje de $\frac{n}{2}$. La solución óptima, seria llenando el tablero con los barcos de largo m, puesto que tenemos $\frac{n}{2}$, esta es necesariamente la mejor forma de llenarlo para esta configuración de barcos.

Así es como llegamos a ubicar a $\frac{n}{2}$ barcos de largo 1, esto cumple una demanda de n. El óptimo es de $\frac{n}{2}$ barcos de largo m, con un valor de $n \cdot m$. Entonces tomando $\frac{z(I)}{A(I)} = \frac{n \cdot m}{n} = m \leq r(I)$. Obtuvimos una mejor cota, ahora de m, por lo que este algoritmo es por lo menos una m-*aproximación*, recordemos que $m = max(n, m)$, por lo que esto significa que podemos estar tán alejados del valor óptimo tantas veces como $A(I) * max(n, m)$.

5.4. Cota empírica

Corrimos juegos con tableros cuadrados donde utilizamos rangos para cada uno de los parametros de $[0, 150]$ de tamaño del tablero y cantidad de barcos. Por lo que se completaron 22500 juegos distintos obteniendo la aproximación ya sabiendo el óptimo por como armamos los juegos. Obtuvimos una cota de $r = 3$.

5.5. Implementación

```python
from heapq import heappush, heappop, heapify
from collections import Counter


def batalla_naval(filas, columnas, b):
```

```python
    """
    La complejidad del algoritmo es O(n * m + (n + m) * (log(n + m) + k * (max(n, m
    ) * l - l^2)) + klogk)
    """
    # O(n * m)
    n, m = len(filas), len(columnas)
    tablero = [[0] * m for _ in range(n)]
    filas = filas.copy()
    columnas = columnas.copy()

    def en_rango(i, j):
        return 0 <= i and 0 <= j and i < n and j < m

    def insertar_barco_horizontal(i, j, l):
        """
        La complejidad es O(l)
        """
        if j + l > m:
            return False

        if l > filas[i]:
            return False

        # O(l)
        if any(columnas[j + k] == 0 for k in range(l)):
            return False

        # O(l)
        for Ci in range(i - 1, i + 2):
            for Cj in range(j - 1, j + l + 1):
                if en_rango(Ci, Cj) and tablero[Ci][Cj] == 1:
                    return False

        # O(l)
        filas[i] -= l
        for k in range(l):
            tablero[i][j + k] = 1
            columnas[j + k] -= 1

        return True

    def insertar_barco_vertical(i, j, l):
        """
        La complejidad es O(l)
        """
        if i + l > n:
            return False

        if l > columnas[j]:
            return False

        # O(l)
        if any(tablero[i + k][j] == 1 for k in range(l)):
            return False

        # O(l)
        if any(filas[i + k] == 0 for k in range(l)):
            return False

        # O(l)
        for Ci in range(i - 1, i + l + 1):
            for Cj in range(j - 1, j + 2):
                if en_rango(Ci, Cj) and tablero[Ci][Cj] == 1:
                    return False

        # O(l)
        columnas[j] -= l
        for k in range(l):
            tablero[i + k][j] = 1
            filas[i + k] -= 1
```

```python

        return True

    def intentar_por_fila(l, i):
        """
        La complejidad es O(m * l - l^2)
        """
        # O(m * l - l^2)
        for j in range(m - l + 1):
            if insertar_barco_horizontal(i, j, l):
                return True

        return False

    def intentar_por_columna(l, j):
        """
        La complejidad es O(n * l - l^2)
        """
        # O(n * l - l^2)
        for i in range(n - l + 1):
            if insertar_barco_vertical(i, j, l):
                return True

        return False

    # O(n + m)
    fil = [(-f, 0, i, intentar_por_fila) for i, f in enumerate(filas)]
    col = [(-c, 1, j, intentar_por_columna) for j, c in enumerate(columnas)]
    restricciones = fil + col
    heapify(restricciones)

    # O(klogk)
    largos = Counter(b)
    b = sorted(largos, reverse=True)

    # O((n + m) * (log(n + m) + k * (max(n, m) * l - l^2)))
    cumplido = 0
    while restricciones:
        res = heappop(restricciones)
        v, t, ij, f = -res[0], res[1], res[2], res[3]

        for l in filter(lambda l: l in largos, b):
            if f(l, ij):
                cumplido += 2 * l
                if v - l > 0:
                    heappush(restricciones, (l - v, t, ij, f))
                if largos[l] == 1:
                    del largos[l]
                else:
                    largos[l] -= 1
                break

    return tablero, cumplido
```

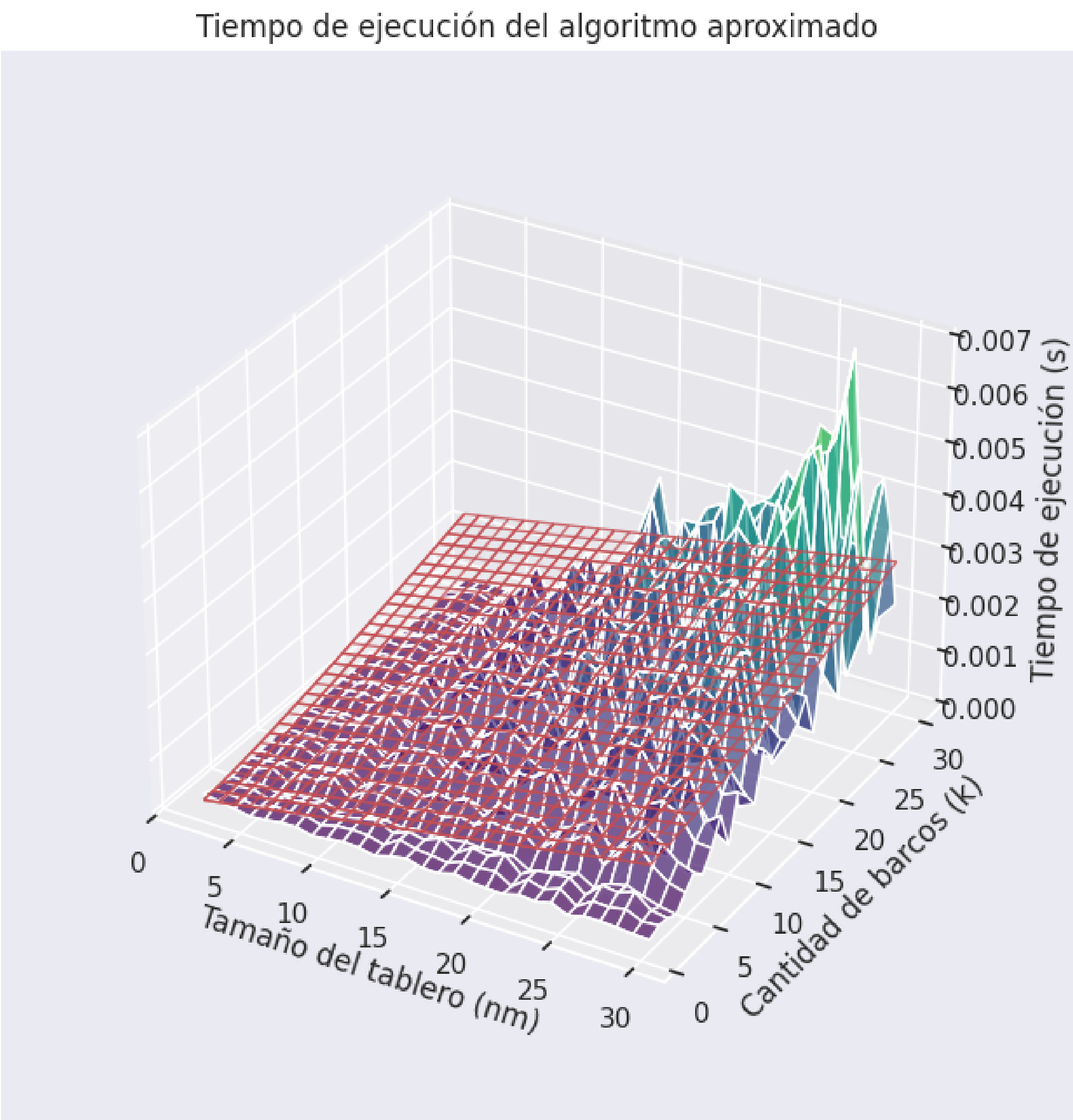
5.6. Comparación con Backtracking

Creamos una tabla utilizando los ejemplos proveidos por la cátedra para comparar la solución óptima obtenida por Backtracking y la obtenida por el algoritmo aproximado.

Dimensión Tablero	Cantidad Barcos	Backtracking	Aproximación	Relación
3×3	2	4	4	1,000
5×5	6	12	12	1,000
8×7	10	26	26	1,000
10×3	3	6	6	1,000
10×10	10	40	38	1,053
12×12	21	46	40	1,15
15×10	15	40	40	1,000
20×20	20	104	90	1,156
20×25	30	172	152	1,132
30×25	25	202	146	1,384

Cuadro 1: Relación de Aproximación entre el Algoritmo y la solución por Backtracking

5.7. Mediciones



Tomando en cuenta la complejidad que obtuvimos, mirando el gráfico tiene sentido que no vaya a escalar tan rápido el tiempo de ejecución como los otros algoritmos que siempre obtienen la solución óptima al problema.

6. Algoritmo Greedy

A continuación presentamos otro algoritmo de aproximación más sencillo que el anterior.

Este algoritmo va a buscar en cada iteración el barco más largo disponible e intentará insertarlo en algún lugar disponible del tablero. Teniendo en cuenta que los barcos más largos son los que van a satisfacer más restricciones, entonces busca en cada iteración satisfacer la mayor cantidad de restricciones utilizando la menor cantidad de barcos. En caso de que un barco no entre, entonces va a desecharlo.

6.1. Análisis de complejidad

Dado que iterativamente tomaremos un barco de largo l comenzando con el de mayor largo y terminando con el de menor largo, primero ordenamos el arreglo de largos b por lo tanto hasta este punto vamos a tener una complejidad de $O(k \cdot log(k))$ con k siendo la cantidad barcos en el arreglo.

Posteriormente para cada uno de estos largos vamos a buscar la primera ubicación posible tal que la misma respete tanto las demandas de la/s fila/s como la/s columna/s que ocupe y además no tenga barcos en ninguna de las posiciones adyacentes. Dado que para cada largo l recorreremos hasta $n \cdot m$ posiciones (con n la cantidad de filas y m la cantidad de columnas), a eso se le suma la complejidad de intentar insertar el barco cada vez que nos movemos en la matriz.

Vamos con este último punto, la complejidad que implica intentar insertar un barco tanto horizontal como verticalmente, primero intentaremos insertarlo de forma horizontal y en caso de que no sea posible lo intentaremos verticalmente.

Primero observemos la complejidad de `insertar_barco_horizontal`:

1. Corroboramos que el largo del barco no supere la demanda de la fila lo cual es $O(1)$.
2. Corroboramos que el largo del barco más la posición de la columna en la cual nos encontramos no exceda la cantidad de columnas, nuevamente $O(1)$.
3. Corroboramos que las columnas de la fila donde estamos intentando ubicar el barco aún tengan demanda $O(l)$.
4. Corroboramos que las posiciones adyacentes a la ubicación donde estamos intentando ubicar el barco se encuentren vacías $O(l)$.
5. Una vez que nos aseguramos de poder ubicar el barco en esta fila , actualizamos la demanda de la fila $O(1)$, actualizamos la demanda de cada columna afectada $O(l)$ y marcamos en el tablero la ubicación seleccionada.

Observaremos el caso de `insertar_barco_vertical` es análogo:

1. Corroboramos que el largo del barco no supere la demanda de la columna lo cual es $O(1)$.
2. Corroboramos que el largo del barco más la posición de la fila en la cual nos encontramos no exceda la cantidad de filas, nuevamente $O(1)$.
3. Corroboramos que las filas de la columna donde estamos intentando ubicar el barco aún tengan demanda $O(l)$.
4. Corroboramos que las posiciones adyacentes a la ubicación donde estamos intentando ubicar el barco se encuentren vacías $O(l)$.
5. Una vez que nos aseguramos de poder ubicar el barco en esta columna, actualizamos la demanda de la misma $O(1)$, actualizamos la demanda de cada fila afectada $O(l)$ y marcamos en el tablero la ubicación seleccionada.

Finalmente podemos observar que la complejidad del algoritmo es $O(n \cdot m \cdot k \cdot l + k \cdot log(k))$ donde n es la cantidad de filas, m la de columnas, k la de barcos y l es el largo promedio de los barcos. Teniendo en cuenta como se calcula el promedio, podemos concluir con que la complejidad es $O(n \cdot m \cdot \sum_{l \in b} l + k \cdot log(k))$

6.2. Implementación

```
def batalla_naval(filas, columnas, b):
    """
    La complejidad del algoritmo es O(n * m * k * l + klogk)
    """
    # O(n + m + n * m)
    n, m = len(filas), len(columnas)
    tablero = [[0] * m for _ in range(n)]
    filas, columnas = filas.copy(), columnas.copy()

    def en_rango(i, j):
        return 0 <= i and 0 <= j and i < n and j < m

    def insertar_barco_horizontal(i, j, l):
        """
        La complejidad es O(l)
        """
        if j + l > m:
            return False

        if l > filas[i]:
            return False

        # O(l)
        if any(columnas[j + k] == 0 for k in range(l)):
            return False

        # O(l)
        for Ci in range(i - 1, i + 2):
            for Cj in range(j - 1, j + l + 1):
                if en_rango(Ci, Cj) and tablero[Ci][Cj] == 1:
                    return False

        # O(l)
        filas[i] -= l
        for k in range(l):
            tablero[i][j + k] = 1
            columnas[j + k] -= 1

        return True

    def insertar_barco_vertical(i, j, l):
        """
        La complejidad es O(l)
        """
        if i + l > n:
            return False

        if l > columnas[j]:
            return False

        # O(l)
        if any(filas[i + k] == 0 for k in range(l)):
            return False

        # O(l)
        for Ci in range(i - 1, i + l + 1):
            for Cj in range(j - 1, j + 2):
                if en_rango(Ci, Cj) and tablero[Ci][Cj] == 1:
                    return False

        # O(l)
```

```
            columnas[j] -= l
            for k in range(l):
                tablero[i + k][j] = 1
                filas[i + k] -= 1

            return True

    def insertar_barco(l):
        """
        La complejidad es O(n * m * l)
        """
        # O(n * m * l)
        for i in range(n):
            for j in range(m):
                if insertar_barco_horizontal(i, j, l):
                    return True

                if insertar_barco_vertical(i, j, l):
                    return True

        return False

    # O(klogk)
    largos = {}
    for l in sorted(b):
        largos[l] = largos.get(l, 0) + 1

    # O (n * m * k * l)
    demanda_cumplida = 0
    while largos:
        l, veces = largos.popitem()

        for _ in range(veces):
            if insertar_barco(l):
                demanda_cumplida += 2 * l
            else:
                break

    return tablero, demanda_cumplida
```

6.3. Comparación con Backtracking

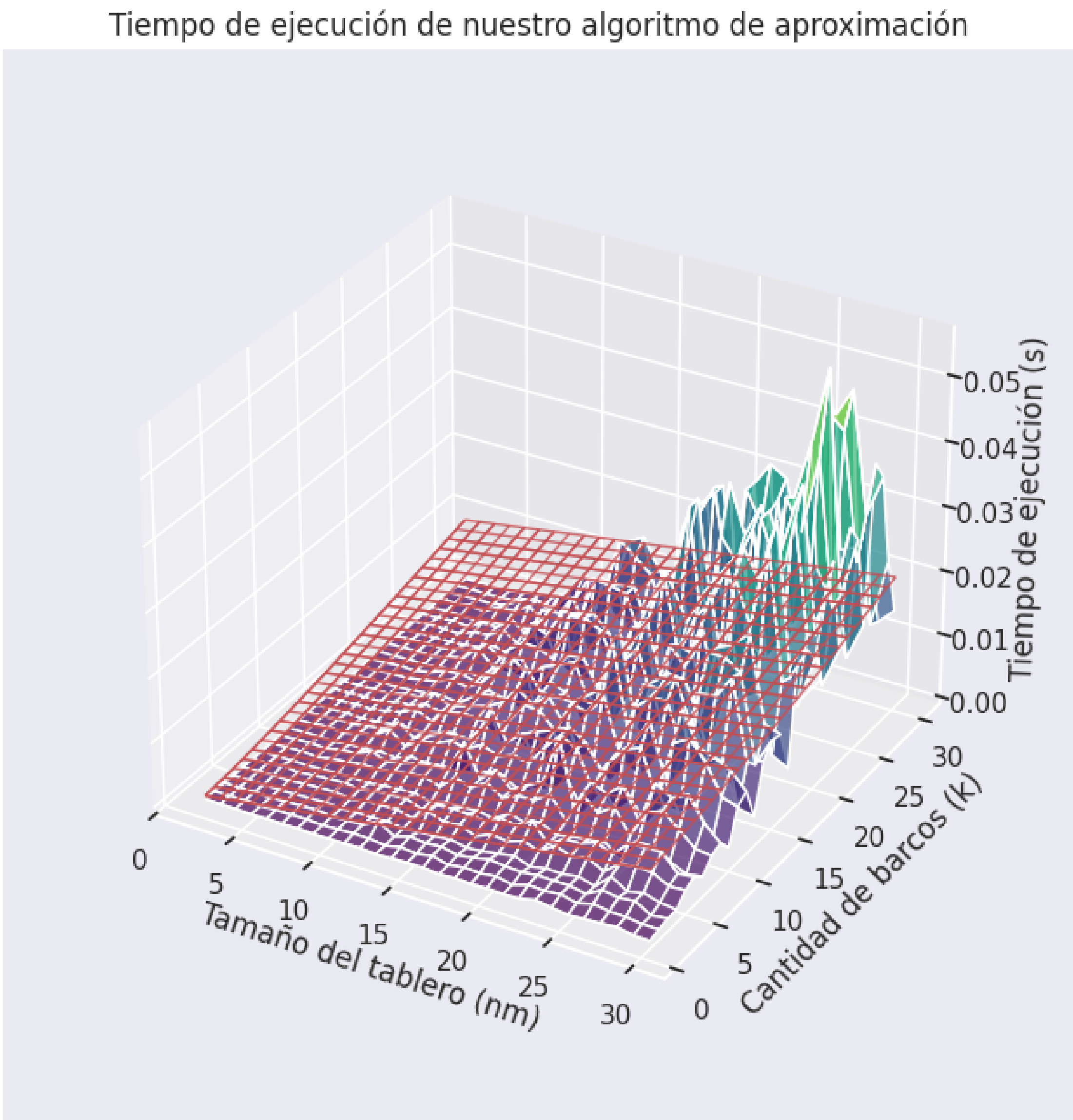
Creamos una tabla utilizando los ejemplos proveidos por la cátedra para comparar la solución óptima obtenida por Backtracking y la obtenida por nuestro algoritmo Greedy.

Dimensión Tablero	Cantidad Barcos	Backtracking	Nuestra aproximación	Relación
3×3	2	4	4	1,000
5×5	6	12	12	1,000
8×7	10	26	26	1,000
10×3	3	6	6	1,000
10×10	10	40	36	0,900
12×12	21	46	40	0,870
15×10	15	40	40	1,000
20×20	20	104	96	0,923
20×25	30	172	172	1,000
30×25	25	202	150	0,743

Cuadro 2: Relación de Aproximación entre el Algoritmo de John Jellicoe y nuestra aproximación

Nos sorprendió que para estos ejemplos, nuestra aproximación dió mejores resultados en general que la aproximación de John Jellicoe.

6.4. Mediciones



Para nuestra aproximación calculamos una complejidad de $O(n \cdot m \cdot k \cdot l + k \cdot logk)$, parece seguir una tendencia parecida a la otra aproximación.

7. Mediciones

Se realizaron mediciones de los distintos algoritmos utilizando un generador de tableros para demostrar empíricamente sus respectivas complejidades teóricas. Se tomoaron 50 muestras separadas uniformemente en cada uno de ellos.

Todos los gráficos muestran una superficie que mejor se ajusta a los datos conseguidos luego de aplicar el método de cuadrados mínimos.

El código utilizado se encuentra dentro de la carpete *codigo* en el archivo jupyter *mediciones.ipynb*. Modificamos la función *time_algorithm* otorgada por la cátedra para acomodar 2 variables en este caso. Hicimos variar la cantidad de barcos y el tamaño del tablero, en todos los casos se usaron tableros cuadrados.

El archivo está dividido en secciones para cada uno de las distintas técnicas de diseño, en caso de querer medir el algoritmo con alguno de ellos en particular, se puede correr su celda pasando el tamaño del tablero y una cantidad de barcos. $graficar$ es la función que recibe el algoritmo, un tope de tamaño y un tope de barcos a utilizar.

También implementamos el cálculo de cuadrados mínimos como fué recomendado en el enunciado del trabajo práctico.

Para todo esto utilizamos módulos de python como: *matplotlib.pyplot*, *seaborn*, *scipy* y *numpy*.

8. Conclusiones

Para resolver el problema planteado se propusieron varios algoritmos, uno por Backtracking, otro por Programación Entera y otro par Greedy. Se realizó una demostración de que $BatallaNaval$ es un problema NP-Completo utilizando 3-$Partition$ en su versión unaria. Se realizaron mediciones para constatar sus complejidades temporales de forma empírica. Para la aproximación se calculo de forma teórica una cota de aproximación y otra de forma empírica, obteniendo un $r = 3$.