.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico Parte 3

9 de febrero de 2025

Nombre	Padrón
Maximo Giovanettoni	110303
Franco Bossi	111122
Luca Saluzzi	108088
Agustin Conti	107800
Gabriel Peralta	101767

1. Introducción

En la parte 3 del Trabajo Práctico se aborda la resolución de un juego de mesa de un único jugador conocido como la Batalla Naval Individual. En este juego tenemos un tablero de n $\times$ m casilleros, y k barcos. Cada barco tiene un largo, el cuál requiere la misma cantidad de 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.

En el siguiente informe se dan demostraciones de que el problema está en np, y de que además es np-completo. También se detalla una resolución usando la técnica de backtraking. Por último se muestra un algoritmos de aproximación propuesto por el almirante John Jellicoe.

Ejemplo de Resolución

2. Demostraciones de NP y NP-Completo

2.1. El problema de la batalla naval está en NP

Para demostrar que el problema de la batalla naval está en NP se debe proponer un validador que verifique, en tiempo polinomial, que un arreglo de barcos es solución del problema.

Se propone el siguiente validador:

Se requieren las listas de demandas por filas y por columnas, y las posiciones de todos los barcos de una solución dada. Por cada barco de la solución, se verifica que este en una posicion vertical u horizontal y que pueda ser ingresado en el tablero en esa posición cumpliendo todas las restricciones del juego, como por ejemplo, que no esté adyacente a otro barco. Así, todos los barcos deben entrar en el tablero o no será una solución válida.

```
# Solucion es una lista de tuplas (fila_ini, columna_ini, fila_fin, columna_fin) con las posiciones de cada barco
def validador_naval(demandas_filas, demandas_columnas, solucion):
    n = len(demandas_filas)
    m = len(demandas_columnas)
    acumulados_fila = [0] * len(demandas_filas)
    acumulados_columna = [0] * len(demandas_columnas)
    tablero = [[0] * m for _ in range(n)]
    for barco in solucion:  # O(k)
        if (barco[0] != barco[2] and barco[1] != barco[3]) or (barco[0] == barco[2] and barco[3] < barco[1] or
                                                               barco[1] == barco[3] and barco[2] < barco[0]):
            return False

        # O(n x m)
        if barco[0] == barco[2] and not ubicar_barco_horizontal(barco[3] - barco[1] + 1, tablero, demandas_filas,
                                                                demandas_columnas, acumulados_fila, acumulados_columna):
            return False

        # O(n x m)
        if barco[1] == barco[3] and not ubicar_barco_vertical(barco[2] - barco[0] + 1, tablero, demandas_filas,
                                                              demandas_columnas, acumulados_fila, acumulados_columna):
            return False
    return True
```

Se muestra como se ubican los barcos horizontalmente. La ubicacion de forma vertical es análoga.

```
    # Verifica que el barco a ubicar no sea adyacente a otro ya ubicado
def adyacente_horizontal(barco, tablero, fila, pos_ini, n, m):
    if pos_ini > 0 and (tablero[fila][pos_ini - 1] == 1 or (fila > 0 and tablero[fila - 1][pos_ini - 1] == 1) or
                        (fila < n - 1 and tablero[fila + 1][pos_ini - 1] == 1)):
        return False
    if pos_ini + barco < m and (tablero[fila][pos_ini + barco] == 1 or
                                (fila > 0 and tablero[fila - 1][pos_ini + 1] == 1) or
                                (fila < n - 1 and tablero[fila + 1][pos_ini + 1] == 1)):
        return False

    for i in range(pos_ini, pos_ini + barco): # O(L) con L el largo del barco actual
        if (fila > 0 and tablero[fila - 1][i] == 1) or (fila < n - 1 and tablero[fila + 1][i] == 1):
```

```
            return False

    return True
```

```
    # Ubica un barco de forma horizontal en el tablero
def ubicar_barco_horizontal(barco, tablero, demandas_filas, demandas_columnas, acum_fila, acum_columna):
    n = len(demandas_filas)
    m = len(demandas_columnas)
    for i in range(n): # O(m x n)
        if acum_fila[i] + barco > demandas_filas[i]:
            continue

        pos_primer_cero = -1
        for j in range(m): # O(m)
            if tablero[i][j] == 1 or (acum_columna[j] + 1 > demandas_columnas[j]):
                pos_primer_cero = -1
                continue
            if pos_primer_cero == -1:
                pos_primer_cero = j
            if j - pos_primer_cero == barco - 1:
                if not adyacente_horizontal(barco, tablero, i, pos_primer_cero, n, m):
                    pos_primer_cero = -1
                    continue
                asignar_barco_horizontal(barco, tablero, i, pos_primer_cero, acum_columna)
                acum_fila[i] += barco
                return True

    return False
```

La complejidad del algoritmo propuesto es $\mathcal{O}(k * n * m)$, con k la cantidad de barcos en la solución. Por lo tanto el problema está en NP.

2.2. El problema de la batalla naval es NP-Completo

Para demostrar que el problema de la batalla naval es np-completo se debe poder reducir bin packing (que es np-completo) al problema de la batalla naval.

Bin packing - problema de decisión: dado un arreglo de números y B bins, cada uno de capacidad C, con la suma de los números igual a $B * C$; se debe poder decidir si los números pueden dividirse en B bins de tal forma que la sumatoria de los números en cada bin sea igual a C.

Reducción propuesta: Se puede formar 1 tablero de (2BC)x2B (2BC filas, 2B columnas), la demanda de cada fila será de 1, y la demanda de cada columna será de C y 0 intercaladamente (C la primera columna, 0 la segunda, C la tercera, etc.) de forma tal que la demanda total entre todas las columnas sea de $B * C$. La demanda total sería de 3BC. Con esto se puede llamar al problema de decisión de la batalla naval, con el arreglo de números visto como barcos, y verificar que la demanda total cumplida sea de $2 * B * C$.

La idea es que, con los números vistos como barcos, cada barco sólo se pueda colocar de forma vertical y cumpla con la demanda C de cada columna.

Entonces existe una solución de Bin Packing para el arreglo de números y B bins de capacidad C si y solo si la demanda total cumplida para el tablero equivalente y el arreglo de números visto como barcos en el problema de la batalla naval es de $2 * B * C$.

Demostración:

- Si con el arreglo de números se puede llenar B bins de capacidad C entonces la sumatoria total de los numeros es de $B * C$. Si vemos los números como un arreglo de barcos, la demanda que se podrá satisfacer es de $2 * B * C$ si entran todos los barcos. Con esto, solo queda ver si entran todos los barcos; como la demanda de cada fila es 1, los barcos solo pueden entrar de forma vertical y sin solaparse entre las filas. Tenemos $2 * B * C$ filas por lo que todos los

barcos pueden entrar en una sola columna respetando la unidad de espacio entre cada barco. Se tienen $2 * B$ columnas, si se excede la demanda C de cada columna se puede ubicar en la columna siguiente que no tenga demanda 0. Asi, la demanda total cumplida será de $2 * B * C$.

- Si la demanda total cumplida es de $2 * B * C$ quiere decir que se tuvo que satisfacer la demanda de todas las columnas con demanda C, como los barcos solo se pueden ubicar en vertical inmediatamente se cumple que la suma del largo de los barcos en las B columnas es igual C.

Esta transformación es polinomial. Por lo tanto el problema de la batalla naval es NP-Completo

3. Explicación del algoritmo de Backtracking

Nosotros hemos decidido en este caso en no recorrer la matriz completa sino en antes poner en dos diccionarios (de filas y columnas) de los lugares posibles los cuales el barco seleccionado podria entrar en base a su valor siendo que no se salga del tablero y que no sea mayor a la demanda de la fila o columna. Luego, en base a estos diccionarios, se va a ir probando todas las combinaciones posibles de barcos en el tablero y se va a ir guardando la mejor solución encontrada.

Se hizo en esta funcion:

```
    def buscar_pos_disponibles_para_barcos(i, j, columnas, filas,
    dicc_posiciones_columnas, dicc_posiciones_filas, largo, max_barco):
    if filas[i] == 0 or columnas[j] == 0:
        return dicc_posiciones_columnas, dicc_posiciones_filas
    columnas_cumple = False
    filas_cumple = False
    if filas[i] > 0:
        dicc_posiciones_filas[1].add((i,j))
        filas_cumple = True
    if columnas[j] > 0:
        dicc_posiciones_columnas[1].add((i,j))
        columnas_cumple = True
    for largo in range(2, max_barco+1):
        if not filas_cumple and not columnas_cumple:
            break
        if columnas_cumple and (i, j-1) in dicc_posiciones_filas[largo-1] and filas
    [i] >= largo:
            dicc_posiciones_filas[largo].add((i,j))
        else:
            columnas_cumple = False
        if filas_cumple and (i-1, j) in dicc_posiciones_columnas[largo-1] and
    columnas[j] >= largo:
                dicc_posiciones_columnas[largo].add((i,j))
        else:
            filas_cumple = False
    return dicc_posiciones_columnas, dicc_posiciones_filas
```

Ademas, se han considerando unas podas para que el algoritmo sea mas eficiente y para que no se recorran todas las posibles combinaciones de barcos en el tablero. Siendo esto una de las bases del Backtracking Estas podas son las siguientes:

3.1. Primera poda

Si el indice a recorrer ya recorrio todos los barcos o si la suma de las demandas de las filas y columnas es 0 o si la demanda de la fila o columna es 0 o si el barco que se quiere poner en el tablero es mayor a la demanda de la fila o columna, entonces se retorna la mejor solución encontrada hasta el momento.

```
    if (idx >= len(barcos) or suma_columnas == 0 or suma_filas == 0 or
    barcos[-1][0] > suma_columnas or barcos[-1][0] > suma_filas):

    if suma_columnas + suma_filas < mejor_solucion[1]:
```

```
        mejor_solucion[0] = copy.deepcopy(matriz)
        mejor_solucion[1] = suma_columnas + suma_filas
        mejor_solucion[2] = copy.deepcopy(posiciones_barcos)
    return mejor_solucion
```

3.2. Segunda poda

Si el estado actual ya fue visitado, entonces se retorna esa solucion

```
    if estado_actual in visitados:
        return visitados[estado_actual]
```

3.3. Tercera poda

Si no hay posiciones disponibles para el barco que se quiere poner en el tablero, se avanza al siguiente barco.

```
    if not posiciones_por_fila[barco] and not posiciones_por_columna[barco]:
    return backtrack(
        matriz, barcos, idx + 1, demanda_filas, demanda_columnas,
        suma_filas, suma_columnas, mejor_solucion, posiciones_por_fila,
        posiciones_por_columna, demanda_restante - barco,
        visitados, posiciones_barcos
    )
```

3.4. Cuarta poda

Si la suma de las demandas de las filas y columnas menos el doble de la demanda restante es mayor o igual a la mejor solución encontrada hasta el momento, se corta la rama.

```
    if (suma_columnas + suma_filas) - (2 * demanda_restante) >= mejor_solucion[1]:
        return mejor_solucion
```

3.5. Quinta poda

Si cumple con toda la demanda, se retorna esa solucion.

```
    if mejor_solucion[1] == 0:
        return mejor_solucion
```

4. Algoritmo de John Jellicoe

El almirante J.J. nos propone un algoritmo de aproximacion para El Problema de la Batalla Naval. Buscamos la fila/columna con mayor demanda y ubicamos el barco de mayor longitud ahi. De no poder ubicarlo, lo salteamos y probamos con el siguiente hasta que no queden más barcos o no haya más demandas a cumplir.

4.1. Algoritmo completo

```
def algoritmo_JJ(n, m, largo_barcos, filas_restricciones, col_restricciones):
    tablero = np.zeros((n, m), dtype=int)

    barcos = sorted(largo_barcos, reverse=True) # O(k log k)

    while barcos:
        filas_demandas = [(i, filas_restricciones[i]) for i in range(n) if
    filas_restricciones[i] > 0] # O(n+m)
```

```
        col_demandas = [(j, col_restricciones[j]) for j in range(m) if col_restricciones[j] > 0] # O(n+m)

        if not filas_demandas and not col_demandas:
            break

        barco_colocado = False  # para verificar si el barco se coloc  en esta iteraci n

        for barco in barcos:
            # Intentar colocar el barco en una fila
            if filas_demandas and (not col_demandas or max(filas_demandas, key=lambda x: x[1])[1] >= max(col_demandas, key=lambda x: x[1])[1]):
                fila_i = max(filas_demandas, key=lambda x: x[1])[0]
                if filas_restricciones[fila_i] >= barco:  # Verificar si la fila puede acomodar el barco
                    for columna_inicio in range(m):
                        if all(0 <= columna_inicio + i < m and col_restricciones[columna_inicio + i] > 0 for i in range(barco)) and puede_poner_barco(tablero, n, m, fila_i, columna_inicio, barco, True):
                            colocar_barco(tablero, fila_i, columna_inicio, barco, True)
                            filas_restricciones[fila_i] -= barco
                            for i in range(barco):
                                col_restricciones[columna_inicio + i] -= 1
                            barcos.remove(barco)
                            barco_colocado = True
                            break
                if barco_colocado:
                    break


            # Intentar colocar el barco en una columna
            if col_demandas and not barco_colocado:
                columna_i = max(col_demandas, key=lambda x: x[1])[0]
                if col_restricciones[columna_i] >= barco:  # Verificar si la columna puede acomodar el barco
                    for fila_inicio in range(n):
                        if all(0 <= fila_inicio + i < n and filas_restricciones[fila_inicio + i] > 0 for i in range(barco)) and puede_poner_barco(tablero, n, m, fila_inicio, columna_i, barco, False):
                            colocar_barco(tablero, fila_inicio, columna_i, barco, False)
                            col_restricciones[columna_i] -= barco
                            for i in range(barco):
                                filas_restricciones[fila_inicio + i] -= 1
                            barcos.remove(barco)
                            barco_colocado = True
                            break
                if barco_colocado:
                    break


        # Si no se pudo colocar ning n barco en esta iteraci n, salimos del bucle
        if not barco_colocado:
            break

    demanda_restante = sum(filas_restricciones) + sum(col_restricciones)
    return tablero, demanda_restante
```

4.2. Complejidad del Algoritmo JJ

Primero ordenamos la lista de barcos, lo que es $O(klogk)$ siendo k la cantidad de barcos.

```
barcos = sorted(largo_barcos, reverse=True)
```

Extraer filas_demandas y col_demandas es $O(n+m)$.

```
filas_demandas = [(i, filas_restricciones[i]) for i in range(n) if
    filas_restricciones[i] > 0]
col_demandas = [(j, col_restricciones[j]) for j in range(m) if col_restricciones[j]
     > 0]
```

La función puede_poner_barco itera sobre el largo l de un barco $O(l)$, y verifica los adyacentes en cada punto $O(l \cdot 8)$. Por lo que la complejidad de puede_poner_barco es $O(l)$.

```
def puede_poner_barco(tablero, n, m, x, y, largo, es_horizontal):
    if es_horizontal:
        if y + largo > m:
            return False
        for i in range(largo): # revisar adyacentes si es horizontal
            if tablero[x, y + i] == 1 or \
                (x > 0 and tablero[x - 1, y + i] == 1) or \
                (x < n - 1 and tablero[x + 1, y + i] == 1) or \
                (x > 0 and y + i > 0 and tablero[x - 1, y + i - 1] == 1) or \
                (x > 0 and y + i < m - 1 and tablero[x - 1, y + i + 1] == 1) or \
                (x < n - 1 and y + i > 0 and tablero[x + 1, y + i - 1] == 1) or \
                (x < n - 1 and y + i < m - 1 and tablero[x + 1, y + i + 1] == 1):
                return False
        if y > 0 and tablero[x, y - 1] == 1:
            return False
        if y + largo < m and tablero[x, y + largo] == 1:
            return False
    else:
        if x + largo > n:
            return False
        for i in range(largo): # revisar adyacentes si no es horizontal
            if tablero[x + i, y] == 1 or \
                (y > 0 and tablero[x + i, y - 1] == 1) or \
                (y < m - 1 and tablero[x + i, y + 1] == 1) or \
                (x + i > 0 and y > 0 and tablero[x + i - 1, y - 1] == 1) or \
                (x + i > 0 and y < m - 1 and tablero[x + i - 1, y + 1] == 1) or \
                (x + i < n - 1 and y > 0 and tablero[x + i + 1, y - 1] == 1) or \
                (x + i < n - 1 and y < m - 1 and tablero[x + i + 1, y + 1] == 1):
                return False
        if x > 0 and tablero[x - 1, y] == 1:
            return False
        if x + largo < n and tablero[x + largo, y] == 1:
            return False
    return True
```

La función colocar_barco es $O(l)$ ya que itera sobre el largo del barco.

```
def colocar_barco(tablero, x, y, largo, es_horizontal):
    if es_horizontal:
        for i in range(largo):
            tablero[x, y + i] = 1
    else:
        for i in range(largo):
            tablero[x + i, y] = 1
```

El peor caso para un barco individual seria verificar la todas las posiciones del tablero $O(n \cdot m)$ y el barco $O(l)$, lo que seria $O(n \cdot m \cdot l)$. La complejidad total del peor caso del ciclo while seria $O(n \cdot m \cdot L)$. Con $L = \sum l_i$.

Sumando todos los componentes juntos:

- **Ordenamiento de los barcos:** $O(k \log k)$.
- **Colocación de barcos:** $O(n \cdot m \cdot L)$.
- **Procesamiento de `fil_demandas` y `col_demandas` en cada iteración:** $O(k \cdot (n + m))$.

El término dominante es el paso de colocación de barcos, por lo que la complejidad general del algoritmo es:

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

donde:

- n: Número de filas.
- m: Número de columnas.
- L: Longitud total de todos los barcos.

5. Optimalidad del Algoritmo

Cota de Aproximación

La idea para argumentar una cota de aproximación es "comparar" lo que hace el algoritmo en cada paso con lo que debería hacer la solución óptima. Una idea de análisis podría ser la siguiente:

Demanda máxima y elección del barco: Sea D la demanda (restante) de la fila (o columna) con mayor necesidad en un paso dado. El algoritmo selecciona el barco más largo que pueda colocarse en esa fila/columna sin exceder la demanda, llamémoslo de longitud L, con $L \leq D$.

Comparación con lo óptimo: La solución óptima, para cubrir esa misma fila (o columna) de demanda D, deberá "sumar" longitudes de barcos que lleguen a D.

Observación "a grosso modo": Si todos los barcos disponibles fueran muy pequeños (digamos, menores que $\frac{D}{2}$), para alcanzar una cobertura de D se necesitarían al menos tres de ellos (o, en el mejor de los casos, dos que se aproximen a $\frac{D}{2}$ cada uno). En cambio, si existe algún barco de tamaño mayor o igual a $\frac{D}{2}$, el algoritmo lo encontrará (pues ordena a partir del mayor barco que encaje en D).

Es decir, bajo una suposición razonable, se puede argumentar que en la fila (o columna) de máxima demanda, el barco que coloca el algoritmo tiene tamaño al menos $\frac{D}{2}$.

Acumulación del "desempeño" en cada paso: Supongamos que en la solución óptima se logra cubrir en esa fila una cantidad total D. El algoritmo, al colocar un barco de tamaño al menos $\frac{D}{2}$, "cubre" al menos la mitad de lo que la solución óptima logra en esa fila.

Si este razonamiento se aplica (de forma "local") en cada elección, se puede "acumular" que, en el peor de los casos, la cobertura (o, de forma equivalente, la reducción en la demanda incumplida) que obtiene el algoritmo es, en cada paso, al menos la mitad de la cobertura óptima en ese bloque o fila/columna.

Conclusión de la cota: Si en cada "decisión" local se garantiza que se cubre al menos la mitad de lo que se podría cubrir de manera óptima, se puede "sumar" el argumento y concluir que, en el peor caso, la demanda cumplida de la solución del algoritmo $A(I)$ es a lo sumo la mitad que la cumplida por el óptimo $z(I)$, es decir,

$$\frac{A(I)}{z(I)} \geq 1/2.$$

5.1. Comparación Empírica

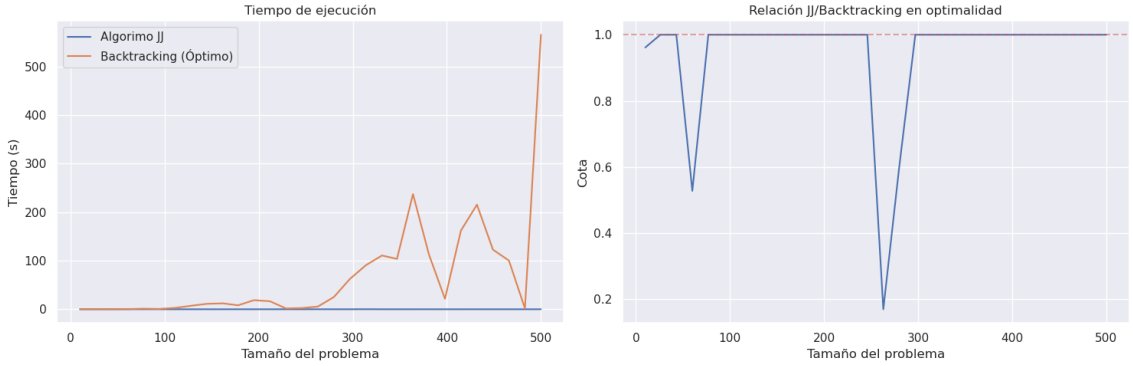
Ya que este algoritmo es una aproximación, nos gustaría saber que tan bueno es, que tanto se acerca al resultado optimo que se podría obtener con algoritmos mas costosos (como por ejemplo, backtracking).

```
.Tiempo de ejecuci n de la matriz 3 x 3 = 0.0:
Demanda cumplida JJ: 4 | Demanda cumplida BT: 4 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 5 x 5 = 0.0010008811950683594:
Demanda cumplida JJ: 12 | Demanda cumplida BT: 12 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 8 x 7 = 0.0010035445404052734:
Demanda cumplida JJ: 26 | Demanda cumplida BT: 26 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 10 x 10 = 0.0030014514923095703:
Demanda cumplida JJ: 38 | Demanda cumplida BT: 40 | Cota: 0.95

.Tiempo de ejecuci n de la matriz 10 x 3 = 0.0:
Demanda cumplida JJ: 6 | Demanda cumplida BT: 6 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 12 x 12 = 0.006998300552368164:
Demanda cumplida JJ: 40 | Demanda cumplida BT: 46 | Cota: 0.8695652173913043

```

```
.Tiempo de ejecuci n de la matriz 15 x 10 = 0.002000570297241211:
Demanda cumplida JJ: 40 | Demanda cumplida BT: 40 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 20 x 20 = 0.015009641647338867:
Demanda cumplida JJ: 104 | Demanda cumplida BT: 104 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 20 x 25 = 0.008996248245239258:
Demanda cumplida JJ: 172 | Demanda cumplida BT: 172 | Cota: 1.0

.Tiempo de ejecuci n de la matriz 30 x 25 = 30.547932624816895:
Demanda cumplida JJ: 170 | Demanda cumplida BT: 202 | Cota: 0.8415841584158416
```

Comparamos los casos de prueba provistos y el algoritmo JJ resultó ser bastante optimo, con una cota media de 0.97 entre los 10 casos. Para estos casos particulares con tableros relativamente pequeños este algoritmo aproximado es útil ya que tarda mucho menos, con resultados prácticamente instantáneos. Además, raramente no llega a la solución optima y en caso de no hacerlo la diferencia de demanda cumplida no es tan grande, con un máximo de 0.84 para la matriz 30 x 25 en la que tardo 30 segundos menos en llegar a esa aproximación.

Aparte creamos una función de casos de prueba aleatorios que genera diferentes tableros con demandas y largos de barcos randomizados. Luego, mediante una Jupyter Notebook corrimos varias iteraciones para generar entonces los siguientes gráficos.



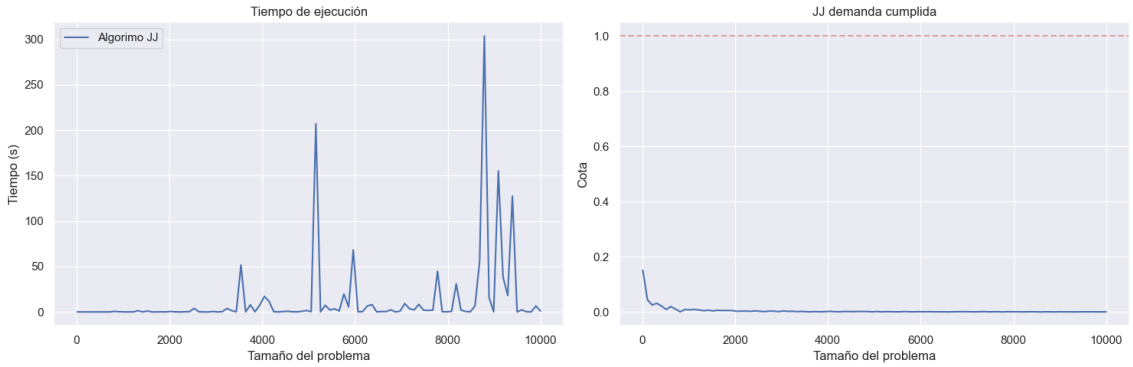
Éste gráfico se generó a partir de 50 iteraciones de tablero de nxn con n entre 10 y 500 con una variabilidad alta (con esto nos referimos a que un tablero podía llegar a ser de 10x3, no necesariamente siempre cuadrado).

Tal y como se puede observar en el segundo gráfico, la menor cota fue de 0.17 para un tablero de 263x165 con barcos [154, 152, 151, 142, 128, 99, 95, 90, 88, 71]. Esta es una cota muy baja, pero debemos observar el promedio de cotas para confirmar que este sea un caso aislado y no usual. Calculado la cota promedio nos da un valor de 0.942.

Por otro lado, en el primer gráfico, se puede observar el tiempo de ejecución de el algoritmo de aproximación (prácticamente lineal) comparado con el de backtracking que llega a tardar mas de 500 segundos (8 minutos).

Tomando estos datos en cuenta podríamos concluir que es un algoritmo de aproximación bastante bueno. con una fiabilidad y grado de aproximación alto.

Finalmente, vamos a calcular la demanda cumplida para un tablero muy grande (inalcanzable para una resolución exacta de backtracking). El mecanismo que usaremos para verificar la optimalidad de la demanda cumplida sobre la demanda total ($\frac{demanda\ cumplida}{demanda\ total}$), ya que no tenemos forma de acceder a la cota para un caso como este por la imposibilidad de calcular con el algoritmo exacto.



En las iteraciones anteriores (las mostradas en el gráfico pasado) la demanda cumplida sobre total promedio nos dio 0.14 (muy similar a la obtenida por BT, que fue 0.16). Ahora calculando 100 tableros de 10.000 filas obtuvimos una demanda comparable a la anterior con tableros menores a 500 pero que rápidamente cae en picado para números mas grandes con una media de 0.0051 entre los 100 tableros.

6. Conclusiones

Demostramos que el problema de la batalla de naval está en NP y que es NP-Completo. Además mostramos dos formas de resolver el problema; mediante backtracking y el algoritmo de aproximación de John Jellicoe. Con este último método se consiguen tiempos sustancialmente mejores y una precisión bastante buena. Dependerá del problema a resolver y de la precisión de se necesite que algoritmo usar.