.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Diversión NP-Completa

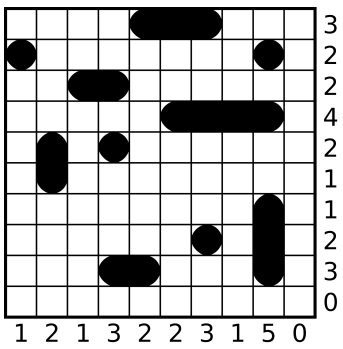
28 de noviembre de 2024

Jonathan Dominguez	Manuel Pato	Mateo Godoy Serrano
110057	110640	110912

1. Introducción

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

En dicho juego, tenemos un tablero de $n \times m$ casilleros, y k barcos. Cada barco i tiene b_i de largo. Es decir, requiere de b_i casilleros para ser ubicado. Todos los barcos tienen 1 casillero de ancho. El tablero a su vez tiene un requisito de consumo tanto en sus filas como en sus columnas. Si en una fila indica un 3, significa que deben haber 3 casilleros de dicha fila siendo ocupados. Ni más, ni menos. No podemos poner dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente). Debemos ubicar todos los barcos de tal manera que se cumplan todos los requisitos. A continuación mostramos un ejemplo de un juego resuelto:



1.1. Consigna

Para los primeros dos puntos considerar la versión de decisión del problema de La Batalla Naval: Dado un tablero de $n \times m$ 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?

1. Demostrar que el Problema de la Batalla Naval se encuentra en NP.
2. Demostrar que el Problema de la Batalla Naval es, en efecto, un problema NP-Completo. Si se hace una reducción involucrando un problema no visto en clase, agregar una (al menos resumida) demostración que dicho problema es NP-Completo.
3. Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema (valga la redundancia) en la versión de optimización: Dado un tablero de $n \times m$ casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo) una lista de las demandas de las n filas y una lista de las m demandas de las columnas, dar la asignación de posiciones de los barcos de tal forma que se reduzca al mínimo la cantidad de demanda incumplida. Pueden no utilizarse todos los barcos. Si simplemente no se cumple que una columna que debería tener 3 casilleros ocupados tiene 1, entonces contará como 2 de demanda incumplida. Por el contrario, no está permitido exceder la cantidad demandada. Generar sets de datos para corroborar su correctitud, así como tomar mediciones de tiempos.

4. Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Ejecutarlo para los mismos sets de datos para corroborar su correctitud. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.

5. John Jellicoe (almirante de la Royal Navy durante la batalla de Jutlandia) nos propone el siguiente algoritmo de aproximación: 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.

   Este algoritmo sirve como una aproximación para resolver el problema de La Batalla Naval. Implementar dicho algoritmo, analizar su complejidad y analizar cuán buena aproximación es. Para esto, considerar lo siguiente: Sea I una instancia cualquiera del problema de La Batalla Naval, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles. Calcular $r(A)$ para el algoritmo dado, demostrando que la cota está bien calculada. Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar dicha relación. Realizar también mediciones que contemplen volúmenes de datos ya inmanejables para el algoritmo exacto, a fin de corroborar empíricamente la cota calculada anteriormente.

6. **Opcional**: Implementar alguna otra aproximación (o algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto anterior. Indicar y justificar su complejidad. No es obligatorio hacer este punto para aprobar el trabajo práctico (pero si resta puntos no hacerlo).

7. Agregar cualquier conclusión que parezca relevante.

2. Análisis del Problema

2.1. Análisis de complejidad

La clase de complejidad NP es el conjunto de problemas de decisión cuyas soluciones pueden comprobarse en tiempo polinomial. Su nombre refiere a **Nondeterministic Polynomial Time**, y, más formalmente, se define que un problema de decisión X pertenece a NP si puede ser resuelto en tiempo polinómico por una máquina de Turing no determinista.

De todas formas, lo que nos interesa de esta clase es que, dada una propuesta de solución para un problema que está en NP, se puede verificar en tiempo polinómico si dicha solución para ese problema es válida o no.

Los problemas de NP que en este informe tendrán principal foco son los denominados **problemas NP-Completos**. Por definición, todo problema que esté en NP puede reducirse polinómicamente a un problema NP-Completo, y se dice que son *los problemas más difíciles dentro de NP*.

Otra clase de complejidad es la de **NP-Difícil**, de la que solo nos interesa la siguiente propiedad:

$$X \text{ es } NP\text{-difícil} \iff Y \leq_p X \quad \forall Y \in NP$$

Donde $\leq_p$ indica una reducción polinomial del problema Y al problema X. De esto se deduce que

$$X \text{ es } NP\text{-Completo} \iff X \in NP \wedge X \text{ es } NP\text{-Difícil}$$

El objetivo de la presente sección es dar una demostración de que el problema de decisión la Batalla Naval (a partir de ahora, BN), enunciado en la introducción del informe, es un problema NP-Completo. Siguiendo la última implicancia, para demostrar esto será necesario demostrar que el problema BN está en NP, y que, además, es NP-Difícil.

2.1.1. Primera demostración: BN está en NP

Para que un problema pertenezca a NP, una solución propuesta para dicho problema debe ser comprobable en tiempo polinomial. Esto es, si existe algún algoritmo A que verifique si una posible solución al problema de BN es válida, y si además se puede acotar la complejidad temporal del algoritmo A por un polinomio, entonces A se dice un certificador eficiente y BN pertenece a NP.

Entonces, proveeremos un certificador eficiente de BN. Para ello, es conveniente analizar las condiciones del problema:

1. Se requiere un tablero de tamaño $n \times m$, una cantidad k de barcos, cada uno con un tamaño b_i, una lista de n demandas para las filas una lista de m demandas para las columnas.
2. Dada una solución, debe respetar el tamaño del tablero, la cantidad de barcos y el tamaño de cada uno, y la cantidad, orden y magnitud de todas las demandas.
3. Dada una solución, debe respetar las restricciones de no adyacencia de barcos y de satisfacibilidad de demandas de manera exacta.
4. Dada una solución, no pueden quedar barcos sin distribuir en el tablero.

Con todas estas restricciones, podemos plantear un certificador eficiente que tenga complejidad temporal polinómica. Por ejemplo, considérese el siguiente validador.

```python
from ..Utilidades.utils_generales import AGUA, ORIENTACIONES, ADYACENTES

def verificar_solucion(tablero, demanda_filas, demanda_columnas, barcos): # O(n*m + b)
    """
    Verifica si la soluci n es correcta.
    - No hay superposiciones de barcos.
    - No hay barcos adyacentes (ni horizontal, vertical, ni diagonalmente).
    - Se cumple con la demanda de cada fila y columna.
    - Que est n todos los barcos colocados y respeten sus tama os.
    """
    filas = len(tablero)
    columnas = len(tablero[0])

    if not verificar_adyacencias(tablero, filas, columnas): # O(n*m)
        return False

    if not verificar_demanda(tablero, demanda_filas, demanda_columnas): # O(n*m)
        return False

    if not verificar_barcos(tablero, filas, columnas, barcos): # O(n*m + b)
        return False

    return True

def verificar_adyacencias(tablero, filas, columnas):
    # Verificar que no hay barcos adyacentes
    for f in range(filas):
        for c in range(columnas):
            if tablero[f][c] != AGUA:
                for df, dc in ADYACENTES:  # Revisar todas las posiciones adyacentes
                    ady_f, ady_c = f + df, c + dc
                    if (0 <= ady_f < filas and 0 <= ady_c < columnas and tablero[ady_f][ady_c] != AGUA):
                        if tablero[f][c] != tablero[ady_f][ady_c]:
                            return False
    return True

def verificar_demanda(tablero, demanda_filas, demanda_columnas):

    # Verificar que las demandas de filas se cumplen
    for i, demanda in enumerate(demanda_filas):
        if sum(1 for celda in tablero[i] if celda != AGUA) != demanda:
            return False

    # Verificar que las demandas de columnas se cumplen
    for j, demanda in enumerate(demanda_columnas):
        if sum(1 for fila in tablero if fila[j] != AGUA) != demanda:
            return False

    return True

def verificar_barcos(tablero, filas, columnas, barcos):
    # Identificar los barcos presentes en el tablero
    barcos_encontrados = {}
    visitado = [[False] * columnas for _ in range(filas)]

    def dfs(f, c, barco_id, direcciones):
        # Realiza una DFS para encontrar un barco completo.
        visitado[f][c] = True
        tama o = 1
        for df, dc in direcciones:
            nf, nc = f + df, c + dc
            if 0 <= nf < filas and 0 <= nc < columnas and not visitado[nf][nc] and tablero[nf][nc] == barco_id:
                tama o += dfs(nf, nc, barco_id, direcciones)
        return tama o

    for f in range(filas):
```

```
67          for c in range(columnas):
68              if tablero[f][c] != AGUA and not visitado[f][c]:
69                  barco_id = tablero[f][c]
70                  # Verificar si el barco es horizontal o vertical
71                  tama o_barco = dfs(f, c, barco_id, ORIENTACIONES)
72                  if barco_id in barcos_encontrados:
73                      return False  # Barco duplicado
74                  barcos_encontrados[barco_id] = tama o_barco
75
76      # Verificar que todos los barcos est n presentes y tienen el tama o correcto
77      barcos_esperados = {idx: barcos[idx] for idx in range(len(barcos))}
78      if barcos_encontrados != barcos_esperados:
79          return False  # Faltan barcos, hay barcos de m s o tama os incorrectos
```

2.1.2. Problema NP-Difícil

Para la siguiente demostración, requeriremos de un problema NP-Difícil, de modo que pueda ser reducido a BN. El problema que vamos a considerar es el de **Bin Packing**. El problema de decisión de Bin Packing (a partir de ahora BP) plantea lo siguiente:

$$\text{Sea } A = \{a_1, ..., a_n\} \text{ y } B \text{ y } C \text{ dos números tales que}$$

$$a_1 + ... + a_n = BC$$

Entonces, ¿Existen B conjuntos $A_1, ..., A_B$ disjuntos cuya unión sea A y cuya suma de elementos para cada uno sea igual a C?

Este es un problema, no solo NP-Difícil, sino NP-Completo, y esto puede demostrarse reduciendo **3-Partition**, que es NP-Completo tanto en forma unaria como en forma binaria, a Bin Packing. Si bien no es el objetivo de este informe demostrar esta reducción, es sencillo notar que es casi directa; basta con considerar un conjunto A y definir que la cantidad B de subsets es 3, mientras que la cantidad C es un tercio de la suma de los elementos de A.

Lo importante es que este problema es NP-Difícil, y buscaremos reducirlo a BN.

2.1.3. Segunda demostración: BN es NP-Difícil

Asumiremos, sin pérdida de generalidad, un conjunto de elementos A de enteros no negativos, lo que no cambia el hecho de que (como demostraremos) BN es un problema NP-Completo.

Por último, es preciso aclarar que tomaremos una representación **unaria** tanto de los elementos $a_1, ..., a_n$ como de los barcos en el problema de BN, y que esto tampoco afecta a la demostración porque el problema de BP es un problema NP-Completo en representación unaria.

Demostración

Sea X una instancia arbitraria de BP, determinada por el conjunto A de enteros no negativos. Buscamos definir un algoritmo P que transforme en tiempo polinómico a la instancia X en una instancia de BN. Luego, dado que existe un algoritmo F que soluciona el problema de decisión BN, si P existe y F se puede aplicar una cantidad polinómica de veces sobre la transformación $P(X)$ de modo que el problema de decisión de BN tenga solución si y solo si el problema de decisión de BP también la tiene, entonces eso probará que BP se puede reducir polinomialmente a BN y que, por lo tanto, todo problema en NP también puede hacerlo, concluyendo que $BN \in NP$.

Para el algoritmo de transformación P, y dada la instancia X de BP, la **Figura 1** muestra una disposición que permitirá comprender de manera sencilla la forma de la reducción.

Dado dicho tablero, se puede asociar cada elemento a_i a un barco de tamaño a_i en el mismo. Es imprescindible notar que cada columna no nula demanda exactamente C fracciones de barco, mientras que cada fila no nula requiere una única fracción. De este modo, podemos corresponder a cada columna de demanda C un subconjunto de elementos de A de manera que sean disjuntos, ya que no es posible asignar varios barcos del mismo tamaño sobre la misma fila y en distintas columnas.

Por último, es válido preguntarse si es posible distribuir los barcos de modo que puedan colocarse en espacios de mayor tamaño al que tienen. Sin embargo, no es difícil notar que este tipo de asignación produciría una de dos situaciones: o bien no se colocan todos los barcos, o bien se violan las demandas de filas o columnas.

Todas estas observaciones se traducen en lo siguiente: es posible interpretar una instancia de BP como una de BN, de modo que, si es posible separar al conjunto A en B subconjuntos disjuntos donde la suma de los elementos de cada uno sume la misma constante C, entonces es posible distribuir barcos sobre la instancia de BP asociada. Análogamente, si no es posible construir dichos B subconjuntos, no será posible distribuir los barcos sobre la instancia de BN.

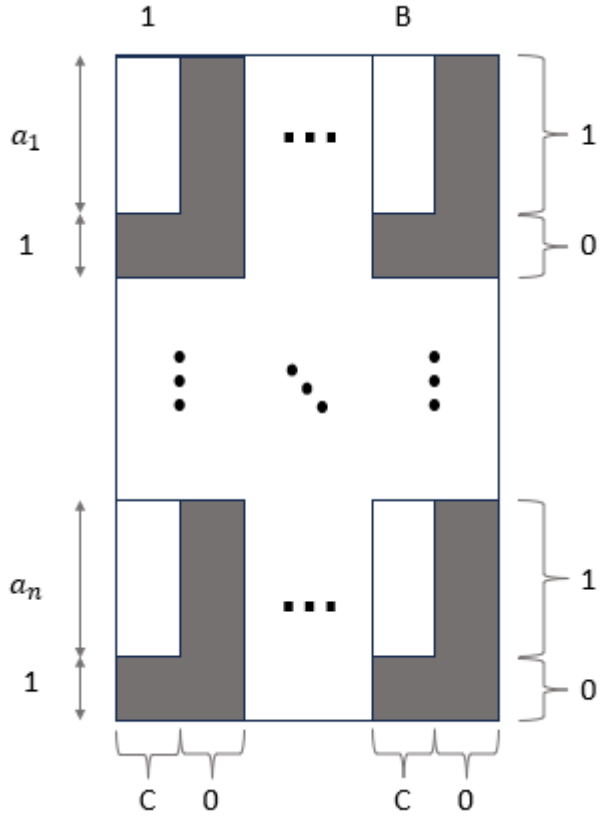


Figura 1: Tablero de la reducción

Esto prueba que BN es un problema NP-Difícil, ya que es posible reducir un problema NP-Difícil a BN. Entonces, como BN es un problema de NP, y además es NP-Difícil, es **NP-Completo**.

3. Análisis de algoritmos

3.1. Backtracking

Según la sección 3, se requiere la implementación de un algoritmo de **backtracking** para la solución del problema de optimización de Batalla Naval.

El algoritmo que propusimos presenta un flujo sencillo, con algunas podas que permiten reducir la cantidad de casos a considerar. Brevemente, el algoritmo itera sobre una lista de barcos y prueba colocar cada uno vertical y horizontalmente, y también no colocarlo, siempre que estas decisiones respeten las condiciones de adyacencia y demanda; con cada decisión tomada, se prueba cada uno de los árboles de posibilidades que no sean podados, registrando soluciones óptimas hasta encontrar la mejor de todas.

Las podas que consideramos son las siguientes:

1. El caso base, que es cuando iteramos todos los barcos, debemos decidir si la solución actual es mejor a la que ya teniamos. Dado el caso que fuera se retorna.
2. Otra que las podas es verificar si el máximo de las demandas tanto filas como columnas es menor al tamaño de nuestro barco actual, se poda llamando recursivamente con el siguiente barco. Dado que ese barco no iba entrar en ninguna de las filas o columnas.

3. Se sabe que, poner un barco hace que la demanda se reduzca en 2 * longitud del barco. Dado que si yo pongo un barco, afecto con su longitud tanto las demandas de filas como de columnas. Entonces si mi demanda incumplida actual menos la suma de los barcos que me resta * 2 es mayor o igual a la mejor demanda incumplida, se poda. Dado que con los barcos que me restan no puedo mejorar la demanda incumplida.

4. Dado un llamado recursivo para un barco k, si el siguiente barco (el $k + 1$) tiene la misma longitud que el barco k, entonces solo consideramos colocarlo a partir de la posición siguiente a la que se colocó el barco k.

5. No esta mal mencionar que también tenemos unas "mini"podas en los ciclos. Donde por ejemplo si mi demanda de filas en esa fila es cero, se puede pasar a la siguiente. Dado que no se puede poner ningún barco en esa fila. Como también otro caso donde si tanto la demanda de esa fila y de esa columna, ambas son menores al tamaño del barco, quiere decir que no podemos poner el barco en ninguna de esas

Finalmente, presentamos implementación del algoritmo de backtracking, incluyendo las funciones que consideramos troncales para su entendimiento, y descartando algunas triviales o que toquen demasiado la lógica de más bajo nivel de la implementación.

```python
def battleship(demanda_filas, demanda_columnas, barcos):
    estado = {"tablero": [[AGUA for _ in range(len(demanda_columnas))] for _ in range(len(demanda_filas))],
                    "demanda_filas": demanda_filas[:], "demanda_columnas": demanda_columnas[:],
                    "incumplimiento": float("inf")}
    mejor_tablero = {"tablero": None, "incumplimiento": float("inf")}
    barcos_ord = sorted(barcos, reverse=True)
    ultima_posicion = {"fila": 0, "columna": 0, "tama o": 0}
    battleship_bt(estado, barcos_ord, 0, mejor_tablero, ultima_posicion)
    return mejor_tablero["tablero"]

def battleship_bt(estado, barcos, idx_b, mejor_tablero, ultima_posicion):
    actualizar_demanda_incumplida(estado)

    # Caso base
    if idx_b >= len(barcos):
        if estado["incumplimiento"] < mejor_tablero["incumplimiento"]:
            mejor_tablero["tablero"] = copiar_tablero(estado["tablero"])
            mejor_tablero["incumplimiento"] = estado["incumplimiento"]
        return

    barco = barcos[idx_b]
    # Poda: si no hay espacio suficiente para este barco
    if barco > max(max(estado["demanda_filas"]), max(estado["demanda_columnas"])):
        battleship_bt(estado, barcos, idx_b + 1, mejor_tablero, ultima_posicion)
        return

    # Poda: si la soluci n actual ya es peor que la mejor encontrada
    demanda_restante = 2 * sum(barcos[idx_b:])
    if estado["incumplimiento"] - demanda_restante >= mejor_tablero["incumplimiento"]:
        return

    # Determinar desde d nde iniciar la b squeda
    inicio_fila, inicio_columna = 0, 0
    if ultima_posicion["tama o"] == barco:
        inicio_fila, inicio_columna = ultima_posicion["fila"], ultima_posicion["columna"]

    for fila in range(inicio_fila, len(estado["tablero"])):
        if estado["demanda_filas"][fila] == 0: continue
        for columna in range(inicio_columna if fila == inicio_fila else 0, len(estado["tablero"][0])):
            if estado["demanda_columnas"][columna] == 0: continue
            if estado["tablero"][fila][columna] != AGUA: continue
            if estado["demanda_filas"][fila] < barco and estado["demanda_columnas"][columna] < barco: continue
            for orientacion in ORIENTACIONES:
                if es_compatible(estado, fila, columna, orientacion, barco):
                    poner_barco(estado["tablero"], fila, columna, orientacion, barco, idx_b)
                    actualizar_demandas(fila, columna, estado, orientacion, barco, -1)

                    if orientacion == (0, 1):  # Horizontal
                        ultima_posicion["fila"] = fila
                        ultima_posicion["columna"] = columna + barco
                    elif orientacion == (1, 0):  # Vertical
                        ultima_posicion["fila"] = fila + barco
                        ultima_posicion["columna"] = columna

                    ultima_posicion["tama o"] = barco
                    battleship_bt(estado, barcos, idx_b + 1, mejor_tablero, ultima_posicion)

                    actualizar_demandas(fila, columna, estado, orientacion, barco, 1)
                    sacar_barco(estado["tablero"], fila, columna, orientacion, barco)
```

```
60
61      # Reinicio de  ltima  posici n si se pasa al siguiente barco
62      ultima_posicion["fila"], ultima_posicion["columna"], ultima_posicion["tama o"]
         = 0, 0, 0
63      battleship_bt(estado, barcos, idx_b + 1, mejor_tablero, ultima_posicion)
64
65  def es_compatible(estado, fila, columna, orientacion, barco):
66      filas, columnas = len(estado["tablero"]), len(estado["tablero"][0])
67
68      if not puede_entrar(fila, columna, estado["demanda_filas"], estado["
        demanda_columnas"], orientacion, barco):
69          return False
70      if not no_excede_demandas(fila, columna, orientacion, barco, estado["
        demanda_filas"], estado["demanda_columnas"]):
71          return False
72      if not se_puede_colocar(fila, columna, filas, columnas, orientacion, barco,
        estado["tablero"]):
73          return False
74
75      return True
```

3.2. Mediciones

La complejidad del algoritmo de backtracking, si bien es al menos exponencial, como se esperaría de un algoritmo del estilo, es difícil de analizar, ya que no depende exclusivamente del tamaño del tablero y de la cantidad de barcos, sino que depende de la demanda en cada fila y en cada columna, y de la longitud de cada barco. Por esto mismo, decidimos realizar una *medición parcialmente sesgada*, es decir, construimos pruebas donde se pusiera en evidencia el comportamiento exponencial del algoritmo (por ejemplo, los tests provistos por la cátedra causaban grandes variaciones en el comportamiento del algoritmo, pasando de resolver casos de tableros de 20×25 y 30 barcos, y tamaños inferiores, en apenas fracciones de segundos, a resolver casos de tableros de 30×25 y 25 barcos en, al menos, 10 segundos).

Los susodichos tests están sesgados en el sentido de que se predefinen la cantidad de filas, columnas, barcos, demandas y longitudes. Dada una instancia de test asociada al número entero i, se define un tablero de tamaño $i \times i$, donde cada demanda es igual a i, y se tienen $\frac{1}{2}i$ barcos de tamaño $\frac{1}{2}i$. Estos tests, además de ser *perfectamente óptimos para los algoritmos de aproximación analizados en este informe*, esto es, siempre coinciden con la satisfacción óptima de demanda (verificado empíricamente hasta un tamaño de tablero de 26×26), ponen en evidencia el comportamiento exponencial del algoritmo de backtracking. Para distribuciones más irregulares, el mismo puede tender a comportarse erráticamente.

Una vez aclarado esto, presentamos un grafico de las mediciones realizadas, donde se relaciona el área del tablero (asociado al parámetro $N = n \cdot m$) y la cantidad de barcos b con el tiempo de ejecución (motivo por el que este, y todos los ajustes siguientes, se muestran en gráficos tridimensionales representados a curvas), para una serie de instancias de tests asociados a los números enteros entre $i = 4$ e $i = 26$.

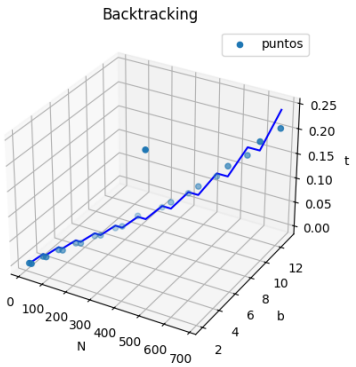


Figura 2: Gráfico del algoritmo de Backtracking

Además, los errores asociados a esta medición fueron los siguientes.

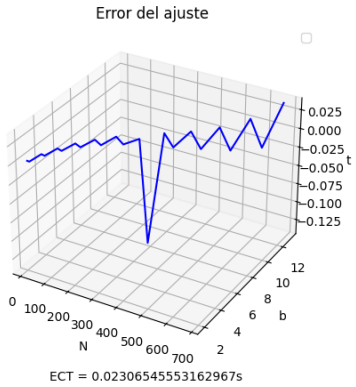


Figura 3: Errores del algoritmo de backtracking

Para esta medición, y para estos tests, la función con la que se ajustó la medición fue

$$f(N, b) = (1{,}44)^b$$

Es una función de dos variables, y fue deducida de forma empírica ajustando la función ajustadora cada vez. Asumir el crecimiento controlado de $N = n \cdot m$ permite establecer una base constante, aunque en la forma más general de un caso de testeo esta función dependería de todas las variables involucradas en el problema de la batalla naval, que incluyen los tamaños del tablero y la cantidad de barcos, pero también contemplan las demandas de cada fila y columna y la longitud de cada barco.

3.3. Algoritmo de aproximación (John Jellicoe)

Según el apartado 5, se propone un algoritmo de aproximación para la solución del problema de optimización de Batalla Naval que consta de aplicar los siguientes pasos:

1. Se identifica la fila o columna de mayor demanda.
2. Se prueba colocar el barco de mayor tamaño en dicha fila o columna en la primera posición disponible válida.
3. Si el barco de mayor tamaño no cabe en dicha fila o columna, se omite y se prueba con el siguiente.
4. Se repiten los pasos indicados, hasta que no haya barcos por colocar o no haya más demandas por suplir.

Si se sigue al pie de la letra estas indicaciones, es sencillo construir un caso donde **el algoritmo sea muy ineficiente**: basta con determinar una columna o fila con una demanda lo suficientemente grande como para que siempre sea la que se identifique como la de mayor demanda cada vez que se quiera colocar un barco.

Debido a esto, nos tomamos la libertad de añadir ciertas restricciones que deben satisfacerse, que, esperamos, se encuentren razonables:

1. Las demandas de cada fila/columna no deben superar la cantidad de casillas de dicha fila/-columna.
2. Dada una iteración asociada a un barco en particular, si se tienen múltiples filas/columnas que alcanzan la máxima demanda, entonces se prueba colocar el barco en cada una de ellas antes de descartar el barco.

Con estas consideraciones, se logró implementar el algoritmo de aproximación con una complejidad $O(W^3)$, donde $W = \max(n, m)$, con n y m la cantidad de filas y columnas, respectivamente. La complejidad no es dependiente de la cantidad de barcos (b) porque consideramos que la misma está acotada por W, luego de definir lo que denominamos densidad de barcos (es decir, $\frac{b}{W}$), y notar, que cuando se supera este umbral la mayoría de barcos quedan descartados.

En síntesis, la complejidad del algoritmo es $O(W^3)$, pero, acorde a la reducción del tiempo de ejecución, se produce una reducción en la demanda satisfecha. Analizamos la calidad de la aproximación al estudiar el cociente propuesto, que tiene la forma

$$\frac{A(i)}{Z(i)} \leq R(A)$$

Para esto, recogimos los resultados arrojados por los tests provistos por la cátedra (no utilizamos tests parcialmente sesgados, ya que, naturalmente, no son representativos de la calidad de la aproximación) y evaluamos dicho cociente. La suma de todas las demandas satisfechas por el algoritmo divido por toda la demanda satisfecha esperada arrojó un resultado de 0,3306, mientras que el máximo alcanzado por este cociente para un test en particular fue de 0,8, que, aún así, no es para nada representativo de la calidad del algoritmo.

Por último, esta es la implementación del algoritmo, incluyendo, nuevamente, solo las partes troncales del mismo.

```python
def maximos(arreglo): # O(W)
    maximo = arreglo[0]
    indices = []
    for i in range(len(arreglo)):
        if arreglo[i] == maximo:
            indices.append(i)
        elif arreglo[i] > maximo:
            maximo = arreglo[i]
            indices.clear()
            indices.append(i)
    return indices

def posicion(tablero, d_filas, d_cols, barco, demanda, longitud_demandas, indice):
    # O(W)
    lado_mas_largo = max(barco[0], barco[1])

    if demanda - lado_mas_largo < 0:
        return None

    # O(W)
    for i in range(longitud_demandas - lado_mas_largo): # O(W)
        pos_inicio = (indice, i) if barco[0] == 1 else (i, indice)

        no_tiene_adyacentes = adyacentes_libres(tablero, pos_inicio, barco[0], barco[1]) # O(W)
        no_excede_demandas = demandas_validas(d_filas, d_cols, pos_inicio, barco[0], barco[1]) # O(W)
        posicion_valida = no_tiene_adyacentes and no_excede_demandas

        if posicion_valida:
            return pos_inicio

    return None

def actualizar_tablero_y_demandas(tablero, d_filas, d_cols, pos_inicio, barco, indice_barco): # O(W)

    alto, ancho = barco

    # Aunque se usa un doble for, cuando uno de ellos tiene complejidad O(W), el otro siempre toma
    # una  nica  iteraci n, debido a que los barcos son de ancho 1, as  que en total toman O(W)
    for i in range(pos_inicio[0], pos_inicio[0] + alto): # O(W), O(1)
        for j in range(pos_inicio[1], pos_inicio[1] + ancho): # O(1), O(W)
            tablero[i][j] = indice_barco
            d_filas[i] -= 1
            d_cols[j] -= 1

def battleship_aproximacion(demandas_filas, demandas_columnas, barcos_originales):
    # O(W)

    # Los siguientes ser n los tama os asociados a cada entrada:
    #   1. Una matriz es de tama o n x m
    #   2. n es la cantidad de filas
    #   3. m es la cantidad de columnas
    #   4. b es la cantidad de barcos
    #   5. Usaremos que W = max(n, m)
    #
    #   Una consideraci n es que el barco de mayor tama o que el algoritmo principal considerar  es de W. Barcos de
    #   mayor tama o nunca alcanzar n el algoritmo principal.

    tablero = construir_tablero(len(demandas_filas), len(demandas_columnas)) # O(W)

    d_filas = demandas_filas.copy()
    d_cols = demandas_columnas.copy()

    barcos = sorted(list(zip(barcos_originales, range(len(barcos_originales)))),
```

```
    reverse = True, key = lambda b: b[0]) # O(b    log(b))
62
63      i = 0 # Para iterar sobre los barcos
64
65      # El bucle termina al suplir con todas las demandas de filas o columnas, o al
        haber revisado todos los barcos
66      # O(W      b)
67      while i < len(barcos) and queda_demanda(d_filas, d_cols): # O(b)
68          indice_barco = barcos[i][1]
69
70          # Ubico todos los m ximos. Esto es por si hay varias filas/columnas con la
         misma demanda
71          maximos_filas = maximos(d_filas) # O(W)
72          maximos_cols = maximos(d_cols) # O(W)
73
74          primer_maximo_filas = d_filas[maximos_filas[0]]
75          primer_maximo_cols = d_cols[maximos_cols[0]]
76
77          # Algunas 'podas'
78
79          # O(1)
80          if barcos[i][0] > primer_maximo_filas and barcos[i][0] > primer_maximo_cols
    :
81              i += 1
82              continue
83
84          # O(1)
85          if barcos[i][0] > max(len(tablero), len(tablero[0])):
86              i += 1
87              continue
88
89          barco_h = (1, barcos[i][0]) # Barco dispuesto horizontalmente, en una fila
90          barco_v = (barcos[i][0], 1) # Barco dispuesto verticalmente, en una columna
91
92          j = 0
93          colocado = False
94
95          if primer_maximo_filas >= primer_maximo_cols:
96              # O(W      b)
97              while j < len(maximos_filas) and not colocado: # O(b)
98                  colocado = intentar_colocar( # O(W  )
99                      tablero,
100                     d_filas,
101                     d_cols,
102                     indice_barco,
103                     barco_h,
104                     d_filas,
105                     maximos_filas[j]
106                 )
107                 j += 1
108             if not colocado and primer_maximo_filas == primer_maximo_cols:
109                 while j < len(maximos_cols) and not colocado: # O(W)
110                     colocado = intentar_colocar( # O(W  )
111                         tablero,
112                         d_filas,
113                         d_cols,
114                         indice_barco,
115                         barco_v,
116                         d_cols,
117                         maximos_cols[j]
118                     )
119                     j += 1
120         else:
121             # O(W      b)
122             while j < len(maximos_cols) and not colocado: # O(W)
123                 colocado = intentar_colocar( # O(W  )
124                     tablero,
125                     d_filas,
126                     d_cols,
127                     indice_barco,
```

```
128                     barco_v,
129                     d_cols,
130                     maximos_cols[j]
131                 )
132                 j += 1
133 
134         i += 1
135 
136     return tablero
```

3.4. Mediciones

Para las mediciones del tiempo de ejecución de este algoritmo se utilizaron (al igual que para el caso de backtracking) tests parcialmente sesgados. Sin embargo, es importante considerar que esto **no afecta el tiempo de ejecución del algoritmo**, esto es, el tiempo de ejecución es independiente de la calidad de la aproximación, y el hecho de que estos tests sean perfectamente óptimos para este algoritmo de aproximación no altera el tiempo que le toma encontrar una solución.

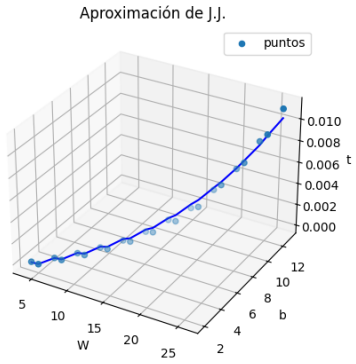


Figura 4: Gráfico de la aproximación de John Jellicoe

Además, estos son los errores asociados al ajuste.

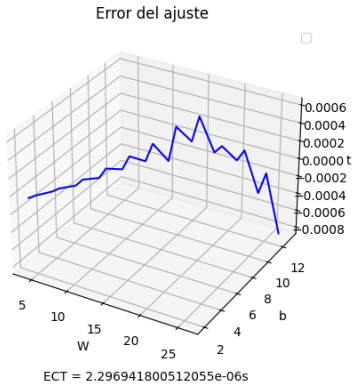


Figura 5: Errores del ajuste de John Jellicoe

La función utilizada para el ajuste fue

$$f(W, b) = W^3$$

que, como se mencionó, es idéntica a la complejidad del algoritmo.

3.5. Algoritmo Greedy

En el apartado 6, se nos pide implementar un algoritmo greedy (o aproximación), distinto del propuesto por John Jellicoe, y hacer una comparación de los resultados de ambos algoritmos.

Planteamos un algoritmo greedy que prioriza la colocación del barco de mayor longitud en el mejor espacio posible. Esto es, el algoritmo busca la fila o columna cuya demanda sea más cercana

a la longitud del barco y sea colocable. Es decir, selecciona la fila o columna que, tras colocar el barco, minimice la demanda sobrante.

Sean n y m, las dimensiones del tablero, representando la cantidad de filas y columnas respectivamente. Definimos $W = \text{máx}(n, m)$, y llamamos b el número de barcos.

En estos términos, la complejidad del algoritmo propuesto es $O(W^3)$. Esto contrasta con la complejidad del algoritmo propuesto por John Jellicoe, cuya complejidad, como fue previamente mencionado, también es de $O(W^3)$.

En el apartado 5 se pide analizar la calidad de la aproximación estudiando empíricamente el cociente entre la demanda satisfecha por el algoritmo de aproximación de John Jellicoe y el algoritmo de backtracking (que, por como está implementado, siempre es óptimo). Al hacer este análisis se llegó a que el promedio de este cociente es 0,3306; haciendo el mismo seguimiento para este algoritmo greedy de aproximación, usando los mismos tests provistos por la cátedra (no sesgados) que se usaron para evaluar el algoritmo de aproximación de John Jellicoe, obtuvimos que el promedio de demanda satisfecha (esto es, la suma de todas las demandas satisfechas sobre la suma de todas las demandas que se esperaban satisfacer) es 0,5694, mientras que el máximo del cociente fue de 0,9, que es considerablemente superior al resultado del algoritmo anterior.

Luego de evaluar esto, notamos que, sin un incremento significativo de la complejidad, logramos implementar un algoritmo de aproximación con resultados notablemente superiores.

Finalmente, esta es la implementación del algoritmo, incluyendo, nuevamente, solo las funciones troncales del mismo.

```python
def posicion(tablero, d_filas, d_cols, barco, demanda): # O(W W)
    indice = demanda[1]
    lado_mas_largo = max(barco[0], barco[1])
    longitud_maxima = len(tablero) if barco[0] == 1 else len(tablero[0])

    if demanda[0] < lado_mas_largo:
        return None

    for i in range(longitud_maxima - lado_mas_largo):
        pos_inicio = (indice, i) if barco[0] == 1 else (i, indice)

        no_tiene_adyacentes = adyacentes_libres(tablero, pos_inicio, barco[0], barco[1])
        no_excede_demandas = demandas_validas(d_filas, d_cols, pos_inicio, barco[0], barco[1])
        posicion_valida = no_tiene_adyacentes and no_excede_demandas

        if posicion_valida:
            return pos_inicio

    return None

def queda_demanda(d_filas, d_cols): # O(W)
    i = 0
    demanda_cumplida = True
    tama o_d_filas = len(d_filas)
    tama o_d_cols = len(d_cols)

    while i < tama o_d_filas and demanda_cumplida:
        demanda_cumplida = (d_filas[i] == 0)
        i += 1

    i = 0
    while i < tama o_d_cols and demanda_cumplida:
        demanda_cumplida = (d_cols[i] == 0)
        i += 1

    return not demanda_cumplida

def actualizar_tablero_y_demandas(tablero, d_filas_heap, d_filas, d_cols_heap, d_cols, pos_inicio, barco, indice_barco): # O(W W)

    filas_desactualizadas = []
    columnas_desactualizadas = []

    alto, ancho = barco

    for i in range(pos_inicio[0], pos_inicio[0] + alto):
        for j in range(pos_inicio[1], pos_inicio[1] + ancho):
            tablero[i][j] = indice_barco
            d_filas[i] -= 1
            d_cols[j] -= 1
            filas_desactualizadas.append(i)
            columnas_desactualizadas.append(j)

def battleship_greedy(demandas_filas, demandas_columnas, barcos_originales):
    tablero = construir_tablero(len(demandas_filas), len(demandas_columnas)) #O (W. W)

    d_filas = demandas_filas.copy() # O(W)
    d_cols = demandas_columnas.copy() # O(W)

    d_filas_heap = heapificar_demandas(d_filas) # O(W log(W))
    d_cols_heap = heapificar_demandas(d_cols)  # O(W log(W))

    barcos = sorted(list(zip(barcos_originales, range(len(barcos_originales)))), reverse=True, key=lambda b: b[0]) # O(W log(W))

    while barcos and queda_demanda(d_filas, d_cols):
        barco = barcos.pop(0)
```

```
66
67          mejor_demanda, mejor_posicion, mejor_orientacion = encontrar_mejor_demanda(
        # O(W.W.W)
68              tablero, d_filas, d_cols, barco, d_filas_heap, d_cols_heap
69          )
70
71          if mejor_demanda and mejor_posicion:
72              actualizar_tablero_y_demandas( # O(W W)
73                  tablero, d_filas_heap, d_filas, d_cols_heap, d_cols, mejor_posicion
    , mejor_orientacion, barco[1]
74              )
75              d_filas_heap = actualizar_heap(d_filas_heap, d_filas) # O(W log(W))
76              d_cols_heap = actualizar_heap(d_cols_heap, d_cols) # O(W log(W))
77
78      return tablero
```

3.6. Mediciones

Para el ajuste de este algoritmo se utilizaron tests parcialmente sesgados. Nuevamente, reiteramos que la calidad de la aproximación es independiente al tiempo de ejecución del algoritmo, con lo que esto no altera la calidad del ajuste.

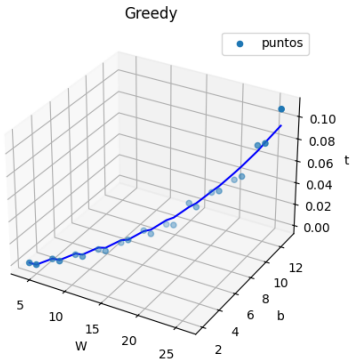


Figura 6: Gráfico del algoritmo Greedy

Además, los errores del ajuste son los siguientes.

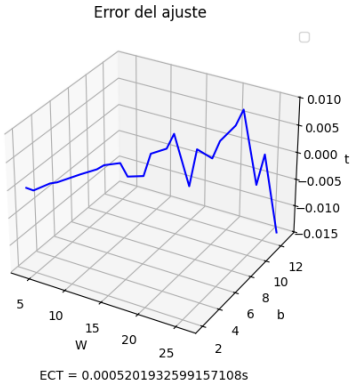


Figura 7: Gráfico del algoritmo Greedy

La función para ajustar estas mediciones fue

$$f(W, b) = W^3$$

que, como se mencionó, coincide con la complejidad del algoritmo.

4. Conclusiones

El desarrollo de este informe desveló dos observaciones que, consideramos, es esencial tener en cuenta.

Primeramente, la complejidad del algoritmo de backtracking tiene una expresión dependiente de todas las variables del problema de batalla naval (dimensiones del tablero, cantidad y longitud de barcos, magnitud de las demandas por cada fila y columna), lo que lo hace errático y de difícil

aproximación. Aún así, casos de testeo parcialmente sesgados ponen en evidencia que el algoritmo puede llegar (y hasta sobrepasar) a un tiempo de ejecución exponencial, como naturalmente se espera de un algoritmo de optimización para un problema NP-Completo como lo es el de Batalla Naval.

En segundo lugar, la aproximación greedy propuesta en este informe resultó de mejor calidad que la propuesta por John Jellicoe, lo que quedó demostrado empíricamente, y esta mejora está relacionada a la fineza de los criterios de selección de filas y columnas y colocación de barcos. Pese a esto, ambos son perfectamente óptimos para casos parcialmente sesgados.

Por último, los ajustes realizados por cuadrados mínimos, a diferencia de en análisis previos, se realizaron con una aproximación más propia del análisis numérico, sin afectar su veracidad y calidad. Este cambio de acercamiento se debe a la introducción de funciones de varias variables.