Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Diversión NP-Completa

16 de diciembre de 2024

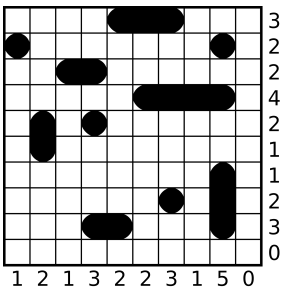
Juan Manuel Dalmau Rennella
109555

Gian Keberlein
109585

1. Enunciado

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 NxM 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×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.

   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. Para esto, recomendamos ver ya sea los problemas 3-Partition o Bin-Packing, ambos en su versión unaria. Si bien sería tentador utilizar 2-Partition, esta reducción no sería correcta. En caso de querer saber más al respecto, consultarnos :-)

2. 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×m casilleros, y una lista de k barcos (donde el barco i tiene bi 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 tene 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.

3. 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.

4. 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.

5. **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).

6. Agregar cualquier conclusión que parezca relevante.

Resolución

2. Batalla Naval Individual está en NP

Para demostrar que el problema de batalla naval está en NP hay que implementarle un verificador eficiente. El mismo debe poder confirmar que las reglas del juego se cumplen y que las demandas estén completamente satisfechas. El mismo debe poder devolver una respuesta certera en tiempo polinomial. Para esto se propone la siguiente implementación:

```
def verificador_batalla_naval(solucion, long_barcos, restricciones_col, restricciones_fil):
    if len(long_barcos) > len(solucion)*len(solucion[0]):
        return False
    aux_r_col = [0]*(len(restricciones_col)) # O(col)
    aux_r_fil = [0]*(len(restricciones_fil)) # O(fil)
    barcos_visitados = set()
    tamanos = []
    for n_fil in range(len(solucion)): # O(fil)
        for n_col in range(len(solucion[0])): # O(col)
            if not solucion[n_fil][n_col]: # Si no hay barco paso de largo
                continue
            if (n_fil,n_col) in barcos_visitados: # Si ya vi este barco antes paso de largo
                continue
            # Tengo que encontrar el resto del barco
            tam_barco_valido = descubrir_barco(solucion, barcos_visitados, aux_r_col, aux_r_fil, n_fil, n_col) # O(long_barco_i + cant_barcos)
            if not tam_barco_valido:
                return False
            tamanos.append(tam_barco_valido)

    if aux_r_col != restricciones_col or aux_r_fil != restricciones_fil: # O(col + fil)
        return False

    barcos = {}
    for b in long_barcos: # O(barcos)
        barcos[b] = 1 if b not in barcos else barcos[b] + 1

    for t in tamanos: # O(barcos)
        if t not in barcos:
            return False
        if barcos[t] == 1:
            del barcos[t]
        barcos[t] -= 1

    if len(barcos) != 0:
        return False

    return True

def descubrir_barco(solucion, barcos_visitados, aux_r_col, aux_r_fil, n_fil, n_col):
    ult_columna, ult_fila = len(solucion[0])-1, len(solucion)-1
    tam = 0
    horizontal = n_fil == ult_fila or not solucion[n_fil+1][n_col]

    # Ciclo para encontrar tamano, guardarlos en visitados, y actualizar restricciones de columnas
    while True:
        tam += 1
        barcos_visitados.add((n_fil,n_col))
        aux_r_col[n_col] += 1
        aux_r_fil[n_fil] += 1

        if horizontal:
            if (n_fil > 0 and solucion[n_fil-1][n_col] == 1) or (n_fil < ult_fila and solucion[n_fil+1][n_col]):
```

```
53                  # Si hay un barco arriba o abajo la solucion es invalida
54                  return 0
55              if n_col == ult_columna or not solucion[n_fil][n_col+1]: # Si a su
    derecha no hay un barco termino
56                  break
57              n_col += 1
58          else:
59              if (n_col > 0 and solucion[n_fil][n_col-1] == 1) or (n_col <
    ult_columna and solucion[n_fil][n_col+1] == 1):
60                  # Si hay un barco a alguno de sus lados la solucion es invalida
61                  return 0
62              if n_fil == ult_fila or not solucion[n_fil+1][n_col]: # Si abajo no hay
     un barco termino
63                  break
64              n_fil += 1
65
66      # Falta ver si se toca con otro barco en los extremos
67      if horizontal:
68          if n_col < ult_columna and (solucion[n_fil][n_col+1] or solucion[min(n_fil
    +1,ult_fila)][n_col+1] or solucion[max(n_fil-1,0)][n_col+1]):
69              return 0
70          if n_col - tam >= 0 and (solucion[n_fil][n_col-tam] or solucion[min(n_fil
    +1,ult_fila)][n_col-tam] or solucion[max(n_fil-1,0)][n_col-tam]):
71              return 0
72      else:
73          if n_fil < ult_fila and (solucion[n_fil+1][n_col] or solucion[n_fil+1][min(
    ult_columna,n_col+1)] or solucion[n_fil+1][max(0,n_col-1)]):
74              return 0
75          if n_fil - tam >= 0 and (solucion[n_fil-tam][n_col] or solucion[n_fil-tam][
    min(ult_columna,n_col+1)] or solucion[n_fil+1][max(0,n_col-1)]):
76              return 0
77
78      return tam
```

Esta implementación recorre como mucho 2 veces las celdas del tablero (Las veces que se encuentra con un barco) y luego itera los barcos encontrados para ver si están todos. Esto da una complejidad polinómica de $O(NxM + B)$, con lo cual queda demostrado que el problema de la batalla naval individual está en NP.

3. Batalla Naval Individual es NP-Completo

Para demostrar que BNI es un problema NP-Completo se debe reducir un problema de esta categoría a él. Para esto se usará el problema de $3 - Partition(Unario)$ en el cual se busca dividir un conjunto de valores enteros en exactamente 3 partes iguales sin dividir los elementos en sí. Para esto hay que transformar los datos que recibiría el problema de 3P en tiempo polinomial para que lo pueda resolver BNI.

3.1. Transformación

- Se toma la lista de valores unarios V de suma total S del 3-Partition
- Se crea una matriz M de 3 filas y $\#V$ columnas mas una columna por cada medio o coma
  - Es decir, si los valores originales eran [3,3,4] serian 3+3+4 columnas más 2 por cada coma.
- Las columnas que representan los valores tienen demanda 1, las que representan las comas tienen demanda 0, y las filas tienen demanda $\frac{S}{3}$
- Los valores V_i se representan con barcos de ese tamaño
- Si es posible que la batalla naval resuelva esto es que hay forma de dividir todo en 3 partes

Esta transformación es polinomial ya que se utiliza la versión unaria del 3P, por lo que queda en $O(V)$.

3.2. Demostración

- **Si hay un 3-Partition válido para un set V de elementos, entonces hay solución para para el juego de batalla naval construido**:

  Si hay 3-Partition entonces hay forma de dividir los elementos de V en partes iguales de valor $\frac{S}{3}$, por lo tanto se cumplirían las demandas de valor S de las filas. El 3P le da cada elemento a una sola de las partes, por lo tanto también se cumplirían las demandas de las columnas. Que haya una columna con demanda 0 por cada espacio entre valores hace que se cumplan las restricciones de adyacencia. Como todas las restricciones se cumplen, hay solución para esta batalla naval.

- **Si hay solución para la batalla naval construida, hay 3-Partition válido para los valores de V:**

  Si hay solución para la batalla naval, significa que los barcos están dispuestos de forma tal que en cada fila hay $\frac{S}{3}$ casilleros ocupados por barcos y donde como mucho hay un casillero ocupado por columna, significando que no hay superposición de barcos en las columnas. Cada espacio o set de columnas entre 2 columnas nulas (Demanda 0) representa un valor de V, por lo tanto esta solución implica que hay forma de dividir a todos los barcos/valores en 3 partes iguales sin darle el mismo barco/valor a 2 filas/partes distintas, por lo que significa que hay solución para el 3-Partition.

4. Batalla Naval por Backtracking

Para abordar la versión de optimización del problema de Batalla Naval, se diseñó un algoritmo de Backtracking que busca minimizar la demanda incumplida en el tablero. Este enfoque permite explorar exhaustivamente todas las configuraciones posibles de barcos en el tablero, retrocediendo en los casos en los que no se pueda avanzar hacia una solución válida.

4.1. Características del Algoritmo

- **Demanda como criterio de poda**: El algoritmo evalúa constantemente la demanda incumplida en filas y columnas, y detiene la búsqueda si esta supera a la mejor solución conocida hasta el momento.
- **Selección inteligente de posiciones**: Se priorizan las posiciones con mayor demanda insatisfecha al decidir dónde colocar un barco, ordenando las posibles ubicaciones según su impacto en la demanda total.
- **Validación estricta**: Se verifican restricciones como adyacencias prohibidas y el cumplimiento de la demanda antes de intentar colocar un barco.
- **Retroceso**: Al alcanzar un callejón sin salida, se retrocede, eliminando el último barco colocado y restaurando las demandas originales.

4.2. Funcionamiento

El algoritmo inicia con un tablero vacío y la lista de barcos ordenada por tamaño, desde el mayor al menor, para priorizar su colocación. En cada iteración:

- Se selecciona el barco más grande y se identifican las posiciones viables basadas en la demanda y las restricciones de ubicación.

- Si se encuentra una posición válida, se coloca el barco y se ajustan las demandas. Luego, se realiza una llamada recursiva para continuar la búsqueda con los barcos restantes.
- Si no se puede avanzar, se retrocede eliminando el barco y restaurando el estado previo del tablero y las demandas.

4.3. Podas Implementadas

- **Poda por barco no colocable**: Si un barco no puede colocarse en ninguna posición válida en el estado actual, se pasa al siguiente barco.

4.4. Ventajas y Limitaciones

El algoritmo es capaz de encontrar soluciones óptimas, pero su complejidad crece exponencialmente con el tamaño del tablero y la cantidad de barcos, haciéndolo impráctico para instancias muy grandes. No obstante, el uso de estrategias de poda y heurísticas reduce significativamente el espacio de búsqueda, haciéndolo viable para instancias moderadas.

4.5. Implementación

```python
import time

def batalla_naval_bt(n: int, m: int, row_demand: list[int], col_demand: list[int], ships: list, board=None, best_solution=None):
    """
    Resuelve el problema de batalla naval usando backtracking con podas.
    Retorna el mejor tablero encontrado y su demanda incumplida.
    """
    # Inicializar mejor solucion
    if best_solution is None:
        best_solution = {
            'board': None,
            'unmet': float('inf')
        }

    # Inicializar tablero
    if board is None:
        board = [[0] * m for _ in range(n)]
        ships.sort()

    # Calcular demanda incumplida actual
    current_unmet = sum(max(0, d) for d in row_demand) + sum(max(0, d) for d in col_demand)

    # Si encontramos una mejor solucion, actualizarla
    if current_unmet < best_solution['unmet']:
        best_solution['unmet'] = current_unmet
        best_solution['board'] = [row[:] for row in board]

    # Si no quedan barcos o ya cumplimos toda la demanda, retornar
    if not ships or current_unmet == 0:
        return best_solution['board'], best_solution['unmet']

    original_ships = ships.copy()
    while ships:
        # Seleccionar el barco mas grande restante
        ship = ships.pop()
        can_place = False

        # Encontrar fila/columna con mayor demanda
        max_row_idx = max(range(n), key=lambda i: row_demand[i])
        max_col_idx = max(range(m), key=lambda j: col_demand[j])

        # Probar todas las posiciones posibles, priorizando la mayor demanda
```

```
        positions = []

        # Agregar posiciones horizontales
        if row_demand[max_row_idx] > 0:
            positions.extend((max_row_idx, j, "horizontal")
                             for j in range(m - ship + 1))

        # Agregar posiciones verticales
        if col_demand[max_col_idx] > 0:
            positions.extend((i, max_col_idx, "vertical")
                             for i in range(n - ship + 1))

        # Ordenar posiciones por demanda total afectada
        def position_score(pos):
            i, j, direction = pos
            if direction == "horizontal":
                return row_demand[i] + sum(col_demand[j+k] for k in range(ship))
            else:
                return col_demand[j] + sum(row_demand[i+k] for k in range(ship))

        positions.sort(key=position_score, reverse=True)

        # Probar cada posicion
        for i, j, direction in positions:
            if se_puede_colocar(board, i, j, ship, direction, row_demand,
    col_demand):
                can_place = True

                # Colocar barco
                colocar_barco(board, i, j, ship, direction, row_demand, col_demand)

                # Recursion
                batalla_naval_bt(n, m, row_demand, col_demand, ships, board,
    best_solution)

                # Retroceso
                quitar_barco(board, i, j, ship, direction, row_demand, col_demand)

        # Si se pudo colocar el barco, no probar con los siguientes barcos. Salir
    del ciclo.
        if can_place:
            ships.append(ship)
            break

    # Restaurar lista de barcos
    ships.clear()
    ships.extend(original_ships)

    return best_solution['board'], best_solution['unmet']

# =========== Utilidades de Barcos =========== #

def se_puede_colocar(board, i, j, ship, direction, row_demand, col_demand):
    """Verifica si es valido colocar un barco en la posicion dada."""

    if direction == "horizontal":
        # Verificar si la fila tiene suficiente demanda
        if row_demand[i] < ship:
            return False

        # Verificar si el barco cabe en la fila y no hay conflictos con la demanda
    de columnas
        for k in range(ship):
            if board[i][j + k] != 0 or col_demand[j + k] <= 0:
                return False

        # Verificar adyacencias (incluyendo diagonales)
        return all(board[x][y] == 0 for x, y in obtener_adyacentes(board, i, j,
    ship, "horizontal"))

```

```
108     elif direction == "vertical":
109         # Verificar si la columna tiene suficiente demanda
110         if col_demand[j] < ship:
111             return False
112 
113         # Verificar si el barco cabe en la columna y no hay conflictos con la
    demanda de filas
114         for k in range(ship):
115             if board[i + k][j] != 0 or row_demand[i + k] <= 0:
116                 return False
117 
118         # Verificar adyacencias (incluyendo diagonales)
119         return all(board[x][y] == 0 for x, y in obtener_adyacentes(board, i, j,
    ship, "vertical"))
120 
121     return False
122 
123 def colocar_barco(board, i, j, ship, direction, row_demand, col_demand):
124     """Coloca un barco en el tablero, actualizando las demandas."""
125 
126     if direction == "horizontal":
127         for k in range(ship):
128             board[i][j + k] = 1
129             row_demand[i] -= 1
130             col_demand[j + k] -= 1
131 
132     elif direction == "vertical":
133         for k in range(ship):
134             board[i + k][j] = 1
135             row_demand[i + k] -= 1
136             col_demand[j] -= 1
137 
138 
139 def quitar_barco(board, i, j, ship, direction, row_demand, col_demand):
140     """Elimina un barco del tablero, restaurando las demandas."""
141 
142     if direction == "horizontal":
143         for k in range(ship):
144             board[i][j + k] = 0
145             row_demand[i] += 1
146             col_demand[j + k] += 1
147 
148     elif direction == "vertical":
149         for k in range(ship):
150             board[i + k][j] = 0
151             row_demand[i + k] += 1
152             col_demand[j] += 1
153 
154 # ========== Utilidades de Adyacencias ========== #
155 
156 def obtener_adyacentes(board, i, j, ship, direction):
157     """Obtiene las posiciones vecinas para verificar restricciones."""
158     neighbors = []
159 
160     if direction == "horizontal":
161         # Fila superior e inferior para todo el barco
162         neighbors += [(i - 1, j + dy) for dy in range(-1, ship + 1)]
163         neighbors += [(i + 1, j + dy) for dy in range(-1, ship + 1)]
164 
165         # Extremos izquierdo y derecho en la misma fila
166         neighbors.append((i, j - 1))
167         neighbors.append((i, j + ship))
168 
169     elif direction == "vertical":
170         # Columna izquierda y derecha para todo el barco
171         neighbors += [(i + dx, j - 1) for dx in range(-1, ship + 1)]
172         neighbors += [(i + dx, j + 1) for dx in range(-1, ship + 1)]
173 
174         # Extremos superior e inferior en la misma columna
175         neighbors.append((i - 1, j))
```

```
176         neighbors.append((i + ship, j))
177
178     return [(x, y) for x, y in neighbors if 0 <= x < len(board) and 0 <= y < len(
        board[0])]
179
180 # =========== Ejemplo de Uso =========== #
181
182 if __name__ == "__main__":
183     # Ejemplo 10 10 10
184     n, m = 10,10
185     row_demand = [3,2,2,4,2,1,1,2,3,0]
186     col_demand = [1,2,1,3,2,2,3,1,5,0]
187     ships = [4,3,3,2,2,2,1,1,1,1]
188
189     total_demand = sum(row_demand) + sum(col_demand)
190
191     start_time = time.time()
192     board, unmet = batalla_naval_bt(n, m, row_demand, col_demand, ships)
193     elapsed_time = time.time() - start_time
194
195     print("Tiempo de ejecucion:", elapsed_time)
196     print("Demanda incumplida minima:", unmet)
197     print("Demanda cumplida:", total_demand - unmet)
198     print("Demanda total:", total_demand)
199     for row in board:
200         print(row)
```

5. Batalla Naval Lineal

Para poder resolver el problema de la batalla naval con programación lineal hay que crear un modelo matemático junto con una función matemática a maximizar o minimizar cuya solución se pueda interpretar como solución del problema original. Esta es nuestra propuesta:

5.1. Variables y Constantes

- $C_{i,j,k}$: *Binaria*, Indica si el barco k está ocupando la celda de fila i y columna j
- $A2_{i,j,k}$: *Binario*, Indica si el casillero i, j del barco k tiene exactamente 2 adyacentes que son parte del barco
- $L1_k$: *Binaria*, Indica si el barco k tiene longitud 1
- B_k : *Binaria*, Indica si se está usando el barco k en la solución final
- O_k : *Binaria*, Indica la orientación del barco k (1 para vertical y 0 para horizontal)
- Dcc_j : *Entera*, Indica la demanda cumplida de la columna j
- Dcf_i : *Entera*, Indica la demanda cumplida de la fila i
- **Cte:** Df_i : *Entera*, Indica la demanda total de la fila i
- **Cte:** L_k : *Entera*, Indica la longitud del barco k
- **Cte:** Dc_j : *Entera*, Indica la demanda total de la columna j
- **Cte:** F : *Entera*, Indica la cantidad de filas
- **Cte:** C : *Entera*, Indica la cantidad de columnas
- **Cte:** B : *Entera*, Indica la cantidad de barcos

5.2. Restricciones

El barco debe ocupar su longitud pero solo si se esta usando

- $\sum_{i=1}^{F} \sum_{j=1}^{C} C_{i,j,k} = L_k * B_k : \forall k \epsilon [1, B]$

Los casilleros tienen que respetar su orientación si se están usando

- $C_{i,j,k} > C_{i-1,j,k} + C_{i+1,j,k} - M(1 - C_{i,j,k}) - M(O_k) : \forall k \epsilon [1, B], \forall i \epsilon [1, F], \forall j \epsilon [1, C]$
- $C_{i,j,k} > C_{i,j+1,k} + C_{i,j-1,k} - M(1 - C_{i,j,k}) - M(O_k) : \forall k \epsilon [1, B], \forall i \epsilon [1, F], \forall j \epsilon [1, C]$

Hay que forzar que los casilleros se queden juntos. Para eso se usa una restricción para chequear que tenga un adyacente pero que se obvie cuando la longitud es 1. Para eso usamos $L1$ y forzamos que sea 1 cuando la longitud del barco k es 1 y 0 si no lo es.

- $L_k > 1 - M(L1_k) : \forall k \epsilon [1, B]$
- $L_k \leq 1 + M(1 - L1_k) : \forall k \epsilon [1, B]$
- $C_{i+1,j,k} + C_{i-1,j,k} + C_{i,j+1,k} + C_{i,j-1,k} \geq 1 - M(L1_K) - M(1 - C_{i,j,k}) : \forall k \epsilon [1, B], \forall i \epsilon [1, F], \forall j \epsilon [1, C]$

Esto solo fuerza que tengan un adyacente pero no que se queden todos juntos. Para eso forzamos que $A2$ sea 1 cuando ese casillero se use y tenga los 2 adyacentes y 0 si no los tiene o no se usa. Luego tiene que haber como mínimo $L_k - 2$ casilleros con $A2$ en 1 (Todos menos los extremos).

- $C_{i+1,j,k} + C_{i-1,j,k} + C_{i,j+1,k} + C_{i,j-1,k} + C_{i,j,k} > 2 - M(1 - A2_{i,j,k}) : \forall k \epsilon [1, B], \forall i \epsilon [1, F], \forall j \epsilon [1, C]$
- $C_{i+1,j,k} + C_{i-1,j,k} + C_{i,j+1,k} + C_{i,j-1,k} + C_{i,j,k} \leq 2 + M(A2_{i,j,k}) : \forall k \epsilon [1, B], \forall i \epsilon [1, F], \forall j \epsilon [1, C]$
- $\sum_{i=1}^{F} \sum_{j=1}^{C} A2_{i,j,k} \geq L_K - 2 : \forall k \epsilon [1, B]$

Ahora no tengo que permitir que se ponga otro barco en los adyacentes a los casilleros ocupados si este barco se está usando para el juego.

- $\sum_{k=1,k \neq K}^{B} \sum_{Ady(I,J)+(I,J)} C_{i,j,k} \leq 0 + M(1 - C_{I,J,K}) : \forall K \epsilon [1, B], \forall I \epsilon [1, F], \forall J \epsilon [1, C]$

Ahora vemos cuánta demanda hay cumplida

- $Dcf_i = \sum_{k=0}^{B} \sum_{j=1}^{C} C_{i,j,k} : \forall i \epsilon [1, F]$
- $Dcc_J = \sum_{k=0}^{B} \sum_{i=1}^{F} C_{i,J,k} : \forall j \epsilon [1, C]$

Y le ponemos el tope

- $Dcf_i \leq Df_i : \forall i \epsilon [1, F]$
- $Dcc_j \leq Dc_j : \forall j \epsilon [1, C]$

5.3. Función a maximizar

Queremos minimizar la demanda incumplida, lo cual es lo mismo a maximizar la demanda cumplida con el 'tope' en la demanda máxima de la respectiva fila/columna.

- $\sum_{j=1}^{C} Dcc_j + \sum_{i=1}^{F} Dcf_i$

5.4. Obtención de la solución

Por cada par fila-columna i, j se iteran los k barcos para ver si alguno tiene a $C_{i,j,k}$ en 1. Si Alguno lo tiene significa que se uso en la solución final y se lo agrega al tablero solución.

5.5. Implementación

La implementación en Python utilizando la librería Pulp es la siguiente:

```python
import pulp
from pulp import LpAffineExpression as Sum

def adyacentes(matriz, fila, columna, direcciones):

    elementos = []

    for dr, dc in direcciones:
        nr, nc = fila + dr, columna + dc

        if 0 <= nr < len(matriz) and 0 <= nc < len(matriz[0]):
            elementos.append(matriz[nr][nc])

    return elementos

def batalla_naval_lineal(demandas_col: list, demandas_fil: list, long_barcos: list)
    :

    problem = pulp.LpProblem("bn", pulp.LpMaximize)

    casilleros = []
    for b in range(len(long_barcos)):
        barco = []
        for f in range(len(demandas_fil)):
            fila = []
            for c in range(len(demandas_col)):
                variables = {
                    "C": pulp.LpVariable("c"+str((f,c,b)),cat="Binary"),
                    "A2": pulp.LpVariable("a2"+str((f,c,b)),cat="Binary"),
                }
                fila.append(variables)
            barco.append(fila)
        casilleros.append(barco)

    barcos = []
    for b in range(len(long_barcos)):
        variables = {
            "L1": pulp.LpVariable("l1"+str(b),cat="Binary"),
            "B": pulp.LpVariable("b"+str(b),cat="Binary"),
            "O": pulp.LpVariable("o"+str(b),cat="Binary")
        }
        barcos.append(variables)

    variables_por_fila = [[] for _ in range(len(demandas_fil))]
    variables_por_columna = [[] for _ in range(len(demandas_col))]

    M = len(demandas_col)*len(demandas_fil)*max(long_barcos)

    # Ahora a definir las restricciones
    for i_barco in range(len(long_barcos)): # Por cada barco k...
        barco_actual = barcos[i_barco]

        # Fuerzo L1
        problem += long_barcos[i_barco] >= 2 - M * barco_actual["L1"]
        problem += long_barcos[i_barco] <= 1 + M * (1 - barco_actual["L1"])

        tablero_k = casilleros[i_barco]
        todos_los_casilleros = []
        for fil in range(len(tablero_k)):

```

```
            todos_los_casilleros += tablero_k[fil]
            for col in range(len(tablero_k[fil])): # Para cada casillero...

                actual = tablero_k[fil][col]

                # Lo meto a los arreglos correspondientes para despues
                variables_por_columna[col].append(actual)
                variables_por_fila[fil].append(actual)

                # Le saco los adyacentes
                horizontales = pulp.lpSum([a["C"] for a in adyacentes(tablero_k, fil, col, [(0,1),(0,-1)])])
                verticales = pulp.lpSum([a["C"] for a in adyacentes(tablero_k, fil, col, [(1,0),(-1,0)])])

                # Restricciones de orientacion
                problem += actual["C"] >= 1 + verticales - M * (1 - actual["C"]) - M * barco_actual["O"]
                problem += actual["C"] >= 1 + horizontales - M * (1 - actual["C"]) - M * (1 - barco_actual["O"])

                # De ser usados, deben tener un adyacente (A menos que sea un barco de tamano 1)
                problem += verticales + horizontales >= 1 - M * barco_actual["L1"] - M * (1 - actual["C"])

                # Fuerzo A2
                problem += actual["C"] + verticales + horizontales >= 3 - M * (1 - actual["A2"])
                problem += actual["C"] + verticales + horizontales <= 2 + M * actual["A2"]

                # Falta ver que en los otros tablero de barco, nadie se ponga cerca de los casilleros que se usan en este tablero
                ady = []
                for j_barco in range(len(long_barcos)):
                    if j_barco == i_barco: continue
                    ady += adyacentes(casilleros[j_barco], fil, col, [(0, 0), (0, 1), (1, 0), (1, 1),(1,-1)])

                problem += Sum([(a["C"],1) for a in ady]) <= 0 + M * (1 - actual["C"])

        # Solo tiene que haber Lk casilleros ocupados
        problem += Sum([(c["C"],1) for c in todos_los_casilleros]) == long_barcos[i_barco] * barco_actual["B"]

        # Debe haber Lk - 2 casilleros con A2 en 1
        problem += Sum([(c["A2"],1) for c in todos_los_casilleros]) >= (long_barcos[i_barco] - 2) * barco_actual["B"]

    # Solo faltan las demandas
    demandas_cumplidas_col = [pulp.LpVariable("dcc"+str(c)) for c in range(len(demandas_col))]
    demandas_cumplidas_fil = [pulp.LpVariable("dcf"+str(f)) for f in range(len(demandas_fil))]

    for i in range(len(demandas_cumplidas_col)):
        problem += demandas_cumplidas_col[i] == Sum([(c["C"],1) for c in variables_por_columna[i]])
        problem += demandas_cumplidas_col[i] <= demandas_col[i]

    for i in range(len(demandas_cumplidas_fil)):
        problem += demandas_cumplidas_fil[i] == Sum([(f["C"],1) for f in variables_por_fila[i]])
        problem += demandas_cumplidas_fil[i] <= demandas_fil[i]

    # Y maximizo las demandas cumplidas
    problem += Sum([(dem,1) for dem in demandas_cumplidas_col] + [(dem,1) for dem in demandas_cumplidas_fil])
```

```
112
113     # Hago que se resuelva el problema
114     problem.solve()
115
116     # Formo la matriz solucion
117     solucion = [[0 for c in range(len(demandas_col))] for f in range(len(
        demandas_fil))]
118     for b in range(len(casilleros)):
119         for f in range(len(casilleros[b])):
120             for c in range(len(casilleros[b][f])):
121                 if int(pulp.value(casilleros[b][f][c]["C"])) == 1:
122                     solucion[f][c] = 1
123
124     return solucion
```

6. Aproximación de John Jellicoe

El almirante nos propone una forma distinta de resolver el problema en menos tiempo pero sacrificando optimicidad. Su implementación se vería así:

```
1  def meter_barco_en_col(tablero, barco, col, demandas_col, demandas_fil): # O(Filas)
2      "Mete el barco en la columna provista si encuentra un lugar valido"
3      i = -1
4      contador = 0
5      for f in range(len(tablero)): # O(F)
6          if tablero[f][col] + tablero[f][max(0,col-1)] + tablero[f][min(col+1,len(
       demandas_col)-1)] == 0:
7              if demandas_fil[f] == 0:
8                  i = f
9                  contador = 0
10             contador += 1
11         else:
12             contador = 0
13             i = f+1
14         if contador == barco + 2 or (i == 0 and contador == barco+1) or (f == len(
       tablero)-1 and contador == barco + 1):
15             for f in range(barco): # O(b)
16                 tablero[i+1][col] = 1
17                 demandas_fil[i+1] -= 1
18                 i += 1
19             demandas_col[col] -= barco
20             return True
21     return False
22
23 def meter_barco_en_fil(tablero, barco, fil, demandas_col, demandas_fil): # O(
       Columnas)
24     "Mete el barco en la fila provista si encuentra un lugar valido"
25
26     i = -1
27     contador = 0
28     for c in range(len(tablero[0])):
29         if tablero[fil][c] + tablero[max(fil-1,0)][c] + tablero[min(fil+1, len(
       demandas_fil)-1)][c] == 0 and demandas_col[c] > 0:
30             contador += 1
31         else:
32             contador = 0
33             i = c+1
34         if contador == barco + 2 or (i == -1 and contador == barco+1) or (c == len(
       tablero[0])-1 and contador == barco + 1):
35             for c in range(barco):
36                 tablero[fil][i+1] = 1
37                 demandas_col[i+1] -= 1
38                 i += 1
39             demandas_fil[fil] -= barco
40             return True
41     return False
42
```

```python
def aproximacion_john_jellicoe(demandas_col: list,demandas_fil: list,long_barcos: list, modified = False):
    long_barcos.sort(reverse=True) # O(B*log(B))
    tablero = [[0 for c in range(len(demandas_col))] for f in range(len(demandas_fil))] # O(F*C)

    filas, columnas = list(range(len(demandas_fil))), list(range(len(demandas_col))) # O(F + C)

    while filas or columnas: # O(Min(F+C, B))
        es_col = True
        max_dem, index = 0, 0
        for f in filas: # O(Filas)
            if demandas_fil[f] > max_dem:
                max_dem, index = demandas_fil[f], f
                es_col = False
        for c in columnas: # O(Columnas)
            if demandas_col[c] > max_dem:
                max_dem, index = demandas_col[c], c
                es_col = True
        if max_dem == 0:
            return tablero

        entro = False
        for barco in long_barcos: # O(Barcos)
            if barco > max_dem:
                continue

            # O( Max(F,C) )
            entro = meter_barco_en_col(tablero, barco, index, demandas_col, demandas_fil) if es_col else meter_barco_en_fil(tablero, barco, index, demandas_col, demandas_fil)

            # O(Barcos)
            if entro:
                long_barcos.remove(barco)
                if len(long_barcos) == 0:
                    return tablero
                break

        # O(Max(F,C))
        if not entro:
            if not modified: return tablero
            if es_col:
                columnas.remove(index)
            else:
                filas.remove(index)

    return tablero
```

6.1. Análisis de Complejidad

- Ordenar los barcos es $O(B * LogB)$
- Crear la matriz y arreglos es $O(F * C + F + C)$
- El while corta luego de que se acaben los barcos o las columnas y filas donde ponerlos por lo que el pero caso es que los vea todos $O(B + F + C)$
    - Buscar la máxima demanda es $O(F + C)$
    - El for itera los barcos: $O(B)$
        - Poner el barco cuesta F o C dependiendo el caso por lo que la cota superior es el máximo de estos: $O(Max(F, C))$
        - Luego quitar el barco de las opciones es $O(B)$

Con esto la complejidad total queda en:

$$O((B+F+C)*(F+C+B*(Max(F,C)+B)+F*C+F+C+BLog(B)$$

Pero la cantidad máxima de barcos que se pueden poner es $\frac{F*C}{4}$, por lo que $B = F*C$ en complejidad y reemplazando queda:

$$O((FC+F+C)*(F+C+FC*(Max(F,C)+FC))+FC+F+C+FCLog(FC)$$

Cuando se tiende a infinito $FC > F+C$ por lo cual se vuelven insignificantes. Luego de una simplificación la complejidad queda en:

$$O((FC)^3+FC+FC*Log(FC)$$

Con el mismo argumento, al tender a infinito $(FC)^3 > FCLog(FC) > FC$ con lo cual la complejidad del algoritmo nos queda finalmente polinomial con cota superior en:

$$O((FC)^3)$$

6.2. ¿Cuánto se aproxima?

Al colocar un pedazo de barco, se están bloqueando todos los casilleros a su alrededor, que tal vez hubieran formado parte de la solución óptima. Si se tratara de un barco de tamaño 1, se estarían bloqueando como mucho 6 casilleros usables, pero para calcular la aproximación debemos tender a valores más grandes, por lo que para un tamaño de barco N, se estarán bloqueando 2N bloques (N a cada lado) que podrían haber formado parte de la solución óptima, más los 6 de los extremos que se vuelven insignificantes cuando N es muy grande. **Por lo que el algoritmo estaría dando una 2-Aproximación a la mejor solución**.

Se puede ver mejor en el siguiente ejemplo en el que se disponía de 2 barcos de tamaño 6.

	6	7	6
0			
2		B	
2		B	
2		B	
2		B	
2		B	
2		B	

Cuadro 1: Solución del algoritmo aproximado

En este ejemplo, el algoritmo aproximado eligió la columna con mayor demanda (7) y posicionó unos de sus barcos más grandes, cuando en realidad la solución óptima era colocar uno a cada lado, duplicando la demanda cumplida. Se puede ver la solución óptima a continuación:

	6	7	6
0			
2	B		B
2	B		B
2	B		B
2	B		B
2	B		B
2	B		B

Cuadro 2: Solución del algoritmo óptimo

6.3. Pruebas de Aproximación

Para probar prácticamente la aproximación teórica, se tomaron las soluciones óptimas proporcionadas en los archivos de prueba de la cátedra y se compararon con los provistos por el algoritmo aproximado para calcular el $r(A)$, luego se hizo un promedio de estos r individuales.

Con esta prueba se obtuvo un $r(A) = 1{,}6$ lo cual tiene sentido con nuestra estimación de que este algoritmo es una **2-Aproximación**

6.4. Prueba de Volumen

Para demostrar la utilidad de esta aproximación se realizaron unas pruebas de volumen en las que se generaron arreglos randomizados de gran tamaño que ya serían inmanejables para los algoritmos óptimos vistos previamente. Los mismos se generaron a partir de una semilla para que sean reproducibles. Este es el código y a continuación los resultados:

```
def prueba_volumen_random(seed, n_col, n_fil, n_bar):
    rd.seed(seed)

    demandas_col = [rd.randint(0,n_fil) for _ in range(n_col)]
    demandas_fil = [rd.randint(0,n_col) for _ in range(n_fil)]
    long_barcos = [rd.randint(0,max(n_fil,n_col)) for _ in range(n_bar)]

    solucion_aprox = aproximacion_john_jellicoe(demandas_col,demandas_fil,
    long_barcos)

    demanda_cubierta = sum([sum(fila) for fila in solucion_aprox])*2
    demanda_total = sum(demandas_col) + sum(demandas_fil)
    print(f"El algoritmo aproximado pudo cubrir {demanda_cubierta} de {
    demanda_total} de demanda total")
```

Se ejecutó este código con una matriz de 100x100 y 50 barcos, y en otro caso con una matriz de 1000x1000 con 200 barcos. Ambos ya son números inmanejables para los algoritmos óptimos, por lo que es de utilidad tener este algoritmo de aproximación para tener una idea de por donde va la solución óptima en tiempo polinomial.

- Para el caso de 100x100x50, el algoritmo aproximado pudo cubrir 1464 de 8612 de demanda total
- Para el caso de 1000x1000x200, el algoritmo aproximado pudo cubrir 26118 de 985009 de demanda total

Los cálculos anteriores nos podrían confirmar que esta solución podría ser como mucho la mitad de buena que la solución óptima de ambos casos.

7. Nuevo algoritmo de Aproximación

Nuestra propuesta para un nuevo algoritmo, más preciso, de aproximación es una versión modificada del algoritmo de John Jellicoe. En la misma, en vez de terminar el programa cuando no se puede colocar un barco en la columna o fila con mayor demanda, se la saca de las opciones, y se busca la siguiente demanda mas grande hasta que no entre ningún barco más o se acaben los barcos.

El mismo tiene la misma complejidad ya que solo agrega una operación en tiempo lineal dentro del while que se ve opacada por la complejidad cuadrática del for. Para ejecutarlo se le debe pasar un parámetro extra 'True' a la función 'aproximacion_john_jellicoe'.

Teóricamente, este algoritmo también es una **2-Aproximación** pero al ejecutarlo con los mismos tests, se obtuvo un $r(A) = 1{,}3$ que es mejor que el $r(A) = 1{,}6$ del algoritmo original.

Al pasarlo por las mismas pruebas de volumen se obtuvieron estos resultados:

- Para el caso de 100x100x50, el algoritmo modificado pudo cubrir 1680 de 8612 de demanda total
- Para el caso de 1000x1000x200, el algoritmo modificado pudo cubrir 26118 de 985009 de demanda total

Es curioso notar que en el caso más pequeño de obtuvo una mejoría palpable mientras que en el caso más grande obtuvo el mismo resultado.

8. Mediciones De Tiempo

Con el objetivo de evaluar la eficiencia de nuestros algoritmos y corroborar las complejidades temporales teóricas realizamos unas pruebas de tiempo con casos de prueba randomizados variando el tamaño del tablero y la cantidad de barcos con el siguiente código. El mismo fija un tamaño M llamado *max_dim* (Por máxima dimensión) y luego varia el tamaño del tablero con una cantidad fija M de barcos, para probar la sensibilidad de los algoritmos a la cantidad de demandas, y luego fija el tablero en un tamaño de MxM variando la cantidad de barcos, para corroborar su sensibilidad a estos.

8.1. Implementación

```
# Funcion para medir el tiempo de ejecucion de un algoritmo
def medir_tiempo(algoritmo, nombre, demandas_col, demandas_fil, long_barcos):
    inicio = time.time()

    if nombre == 'Backtracking':
        resultado, _ = algoritmo(len(demandas_col), len(demandas_fil), demandas_col
    , demandas_fil, long_barcos)
    else:
        resultado = algoritmo(demandas_col, demandas_fil, long_barcos)

    duracion = time.time() - inicio
    return duracion, resultado

def aprox_mod(demandas_col, demandas_fil, long_barcos):
    return aproximacion_john_jellicoe(demandas_col, demandas_fil, long_barcos, True
    )

def medir_y_graficar_tiempos():
    algoritmos = [
        (aproximacion_john_jellicoe, "Aproximacion"),
        (aprox_mod, "Aproximacion_mod"),
        (batalla_naval_bt, "Backtracking"),
        (batalla_naval_lineal, "Lineal")
    ]
    max_dim = 150
    demandas_col = [rd.randint(0, max_dim) for _ in range(max_dim)]
    demandas_fil = [rd.randint(0, max_dim) for _ in range(max_dim)]
    long_barcos = [rd.randint(1, max_dim) for _ in range(max_dim)]

    print(demandas_col)
    print(demandas_fil)
    print(long_barcos)

    resultados_tamanos = {alg[1]: [] for alg in algoritmos}
    resultados_barcos = {alg[1]: [] for alg in algoritmos}

    # Variar columnas
    for cols in range(1, max_dim + 1):

        for algoritmo, nombre in algoritmos:
            print(f"Probando {nombre} con {cols} columnas")
            duracion, _ = medir_tiempo(algoritmo, nombre, demandas_col[0:cols].copy
    (), demandas_fil[0:cols].copy(), long_barcos.copy())
```

```
            resultados_tamanos[nombre].append(duracion)

    # Variar barcos
    for barcos in range(1, max_dim + 1):

        for algoritmo, nombre in algoritmos:
            print(f"Probando {nombre} con {barcos} barcos")
            duracion, _ = medir_tiempo(algoritmo, nombre, demandas_col.copy(),
    demandas_fil.copy(), long_barcos[0:barcos].copy())
            resultados_barcos[nombre].append(duracion)

    # Graficar resultados
    x = list(range(1, max_dim + 1))

    # Grafico 1: Tiempo vs Tamanos
    plt.figure(figsize=(10, 6))
    for nombre, tiempos in resultados_tamanos.items():
        plt.plot(x, tiempos, label=nombre)
        plt.title("Tiempo vs Cantidad de Columnas")
        plt.xlabel("Cantidad de Columnas")
        plt.ylabel("Tiempo (s)")
        plt.legend()
        plt.grid()
        plt.savefig(f"graficos/{nombre}/tiempo_vs_tam.png")
        plt.close()

    # Grafico 2: Tiempo vs Barcos
    plt.figure(figsize=(10, 6))
    for nombre, tiempos in resultados_barcos.items():
        plt.plot(x, tiempos, label=nombre)
        plt.title("Tiempo vs Cantidad de Barcos")
        plt.xlabel("Cantidad de Barcos")
        plt.ylabel("Tiempo (s)")
        plt.legend()
        plt.grid()
        plt.savefig(f"graficos/{nombre}/tiempo_vs_barcos.png")
        plt.close()


# Llamar a la funcion
medir_y_graficar_tiempos()
```

8.2. Gráficos resultantes

A continuación presentamos los resultados de ejecutar dicho programa con diferentes parámetros acorde a las capacidades de nuestras máquinas y eficiencia de los algoritmos.

8.2.1. Backtracking

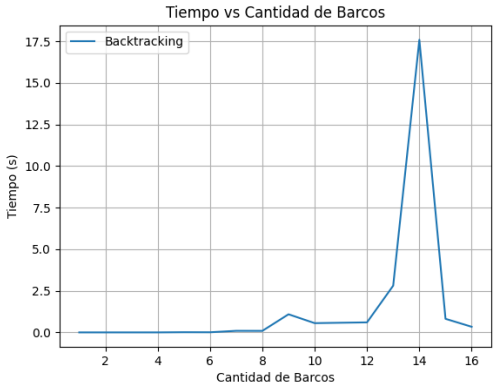


Figura 1: Tiempos vs Cantidad de barcos con Backtracking

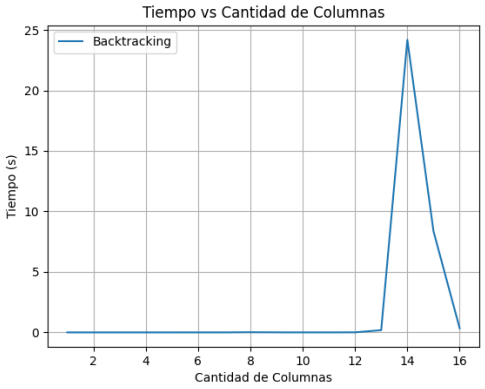


Figura 2: Tiempos vs Tamaño del tablero con Backtracking

Nuestro algoritmo que utiliza Backtracking pudo alcanzar los escenarios de $M = 16$. Al intentar nuestro programa de pruebas con un M mayor, no pudieron terminar la ejecución.

8.2.2. Programación Lineal

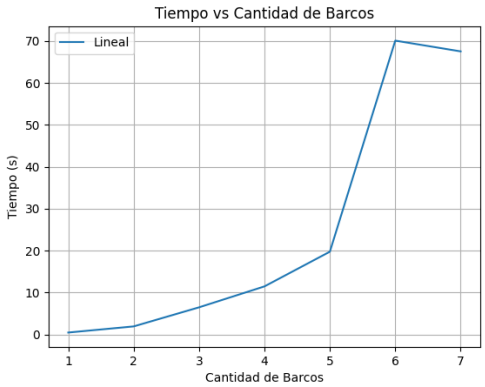


Figura 3: Tiempos vs Cantidad de barcos con Programación lineal

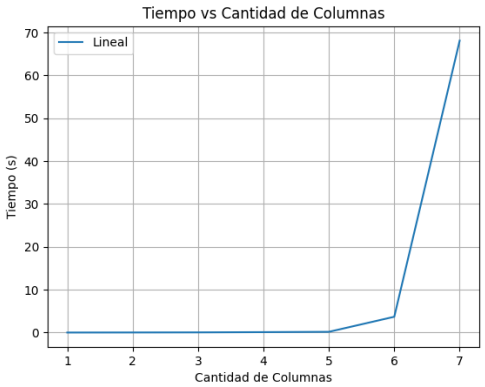


Figura 4: Tiempos vs Tamaño del tablero con Programación lineal

Similar al algoritmo de backtracking, el algoritmo que usa programación lineal aguantó hasta un $M = 7$, al ejecutarlo con cantidades mas grandes, la ejecución no pudo terminar. En comparación, esta versión es mucho menos eficiente.

8.2.3. Aproximación de John Jellicoe

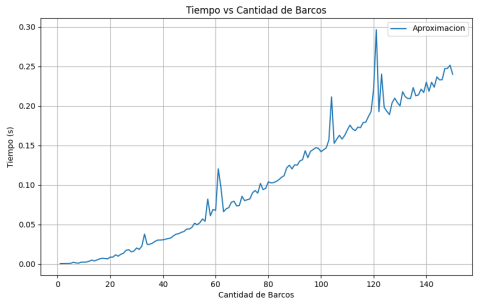


Figura 5: Tiempos vs Cantidad de barcos con el algoritmo de Aproximación

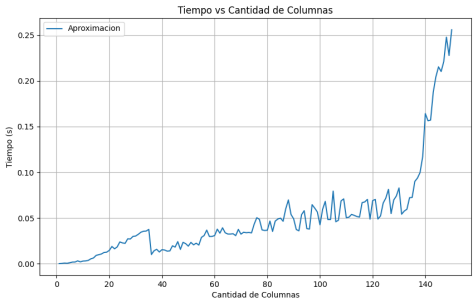


Figura 6: Tiempos vs Tamaño del tablero con el algoritmo de Aproximación

El algoritmo de aproximación logra su objetivo de poder lograr muchas mejores velocidades a costa de optimicidad, le tomamos los tiempos hasta casos de prueba con 150 elementos, el algoritmo aguanta cantidades mas grandes pero a fines de demostración perdía sentido hacerlo ya que queda claro que es mucho mas rápido que sus contrapartes óptimas.

8.2.4. Aproximación de John Jellicoe modificada

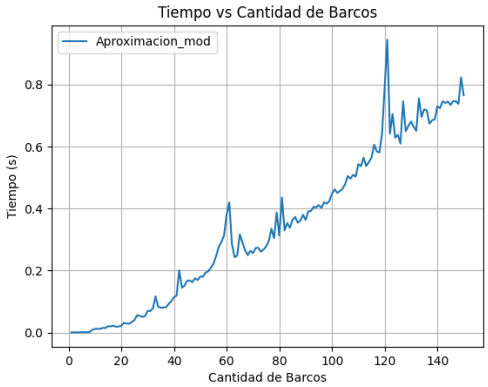


Figura 7: Tiempos vs Cantidad de barcos con el algoritmo de Aproximación modificado

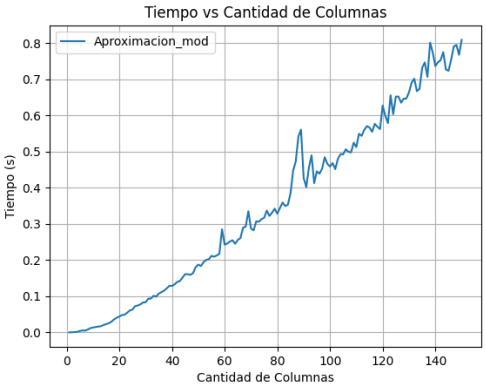


Figura 8: Tiempos vs Tamaño del tablero con el algoritmo de Aproximación modificado

Igual que con su versión original, el algoritmo de John Jellicoe que modificamos también logra tiempos mucho mejores que sus contrapartes óptimas, pero en busca de un resultado más cercano al óptimo, sacrifica un poco de esta eficiencia. Este sacrificio de velocidad se puede notar al ver que en los casos de $M = 150$ hay una diferencia de medio segundo entre ambos.

9. Conclusión

El problema de la Batalla Naval Individual es un ejemplo representativo de la complejidad inherente de los problemas NP-Completos. A través de este trabajo, se implementaron y analizaron diferentes enfoques para resolver tanto la versión de decisión como la versión de optimización

del problema, utilizando técnicas exactas como Backtracking y Programación Lineal, así como algoritmos de aproximación.

El algoritmo de Backtracking demostró ser una herramienta poderosa para encontrar soluciones óptimas, especialmente en instancias de tamaño moderado. La incorporación de podas y heurísticas permitió reducir el espacio de búsqueda de manera significativa, optimizando los tiempos de ejecución. Sin embargo, como era de esperarse, su escalabilidad es limitada debido a su crecimiento exponencial en problemas de mayor tamaño.

Por otro lado, el modelo de Programación Lineal ofreció una manera formal de abordar el problema, permitiendo obtener soluciones óptimas en tiempos razonables para tamaños similares, mientras que los algoritmos aproximados presentaron una alternativa eficiente para escenarios más grandes, sacrificando precisión a cambio de tiempos de ejecución polinomiales. En particular, se observó que los algoritmos aproximados ofrecen resultados aceptables en la práctica, con un coeficiente de aproximación consistente en pruebas empíricas.

En general, el análisis de los diferentes enfoques confirma la importancia de adaptar la técnica de resolución al contexto del problema. Para casos pequeños o donde la optimalidad es crítica, Backtracking y Programación Lineal son las opciones preferidas. Sin embargo, para problemas de gran escala o en escenarios donde el tiempo de respuesta es un factor determinante, los algoritmos aproximados son herramientas esenciales para obtener resultados útiles.