.UBAfiuba
FACULTAD DE INGENIERÍA

Trabajo Práctico 3
Diversión NP-Completa

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Nombre	Padrón
Denise Dall'Acqua	108645
Martín Alejo Polese	106808
Nicolás Agustín Riedel	102130

Índice

Consigna

Resolución

1. Demostración: NP

Para que un problema se encuentre en NP, se debe poder encontrar un validador que valide si la solución es correcta, y lo haga en tiempo polinomial.

Nuestro problema, está dado por:

- Una lista con las demandas para cada fila
- Una lista con las demandas para cada columna
- Una lista de k barcos (donde el barco s tiene b_s de largo)

La solución, está dada por una Matriz de tamaño n * m, donde para cada casillero ij:

- Si no hay barco, entonces Matriz[i][j] = None
- Si hay un barco, entonces Matriz[i][j] = s

Nota: Siendo s el índice de dicho barco

El validador entonces verifica lo siguiente:

- Que los barcos no sean adyacentes
- Que las demandas de las filas se cumplan de manera exacta
- Que las demandas de las columnas se cumplan de manera exacta
- Que se coloquen todos los barcos

1.1. Que los barcos no sean adyacentes

Para verificar que los barcos no sean adyacentes, basta con recorrer cada casillero de la matriz (n * m) y en cada celda visitar las 8 celdas vecinas (un cuadrado). En cada una de las celdas vecinas que visito, veo si hay otro barco distinto, y si lo hay, la solución no es válida. Visitar las 8 celdas vecinas se hace en tiempo constante O(1). Por lo tanto, verificar que no haya barcos adyacentes tiene un costo de O(n * m)

1.2. Que las demandas de las filas y columnas se cumplan de manera exacta

Para verificar que las demandas de las filas se cumplan, basta con recorrer cada casillero de la matriz (n * m) y en cada casillero que haya un barco, restar 1 a la demanda de la fila y columna a la que pertenece. Finalmente, se recorre la lista con las demandas para cada fila y nos fijamos que sea todo igual a cero O(n). Lo mismo con la lista de las demandas para cada columna O(m). Esto nos da un costo total de O(n * m)

1.3. Que se coloquen todos los barcos

Para verificar que se colocaron todos los barcos, hay que recorrer cada casillero de la matriz (n * m), y en el caso de que en el casillero haya un barco, se agrega a los barcos visitados. Luego se

recorren todos los barcos O(k) y nos fijamos que hayan sido todos colocados. Esto nos da un costo total de O(n * m).

En conclusion, como pudimos validar una solucion al problema en tiempo polinomial, podemos afirmar que el problema se encuentra en NP.

2. Demostración: NP-Completo

Reducción desde el problema 3-Partition

Vamos a demostrar que la batalla naval es NP completo. Pero, ¿Como se demuestra que un algortimo es NP completo? Tiene que cumplir dos puntos:

- **Pertenencia a NP:** La verificación de una solución candidata es posible en tiempo polinomial.
- **NP-dificultad:** Se puede realizar una reducción polinomial desde cualquier problema NP-completo hacia este problema.

Ya en la sección anterior pudimos verificar con éxito que nuestro problema es un problema NP, ahora hay que demostrar que se puede realizar una reducción polinomial desde cualquier problema NP-Completo. Recordemos que todos los NP completos pueden ser reducidos despues cualquier problema NP completo.

En nuestro caso, utilizamos el problema de *3-Partition* para verificar que la batalla naval puede ser resuelta en tiempo logarítmico.

Definamos un poco cómo se compone el problema *3-Partition*

Definición del problema 3-Partition

Dado un conjunto de $3m$ números positivos $A = \{a_1, a_2, \ldots, a_{3m}\}$, donde cada número está representado en notación unaria y la suma total de los números es $S = m \cdot B$, queremos particionar A en m subconjuntos disjuntos $S_1, S_2, \ldots, S_m$ tales que:

- Cada subconjunto S_i tiene exactamente 3 elementos.
- La suma de los elementos en cada subconjunto es exactamente B.

Reducción de 3-Partition a La Batalla Naval

Dado una instancia del problema *3-Partition*, construiremos una instancia del problema *La Batalla Naval*.

Construcción del tablero:

- **Dimensiones del tablero:** Construimos un tablero con m filas y $3m$ columnas. Las restricciones de las filas (r_i) serán B, es decir, cada fila i debe contener exactamente B casillas ocupadas.
- **Restricciones de las columnas:** Las restricciones de las columnas (c_j) serán los elementos del conjunto $A = \{a_1, a_2, \ldots, a_{3m}\}$. Es decir, la columna j tendrá una restricción igual a a_j.
- **Barcos:** Introducimos $3m$ barcos, uno por cada elemento de A, donde el barco b_j tiene una longitud igual a a_j.

vamos con un ejemplo para poner en contexto esto que estamos diciendo:

Tenemos los siguientes datos del problema 3-Partition:

- $A = \{6, 5, 4, 5, 3, 7\}$,
- $B = 15$, porque por ejemplo si tengo los subconjuntos $S_1 = 6, 5, 4$ y $S_2 = 5, 3, 7$ la suma de los elementos de S_1 es 15 al igual que la suma de los elementos de S_2
- $m = 2$. porque $|A| = 6 = 3{,}2 => 3.m$

Ajustando el Tablero

Para que cada fila pueda contener exactamente $r_i = 15$ casillas ocupadas, necesitamos que el número de columnas del tablero sea suficiente para permitir esa suma. Ajustamos las dimensiones del tablero:

- $n = 2$ filas (porque $m = 2$, el número de subconjuntos).
- $m = 15$ columnas (ya que cada fila debe contener hasta 15 casillas ocupadas).

Por lo tanto, el tablero tiene $2 \times 15 = 30$ casillas, lo cual coincide con la suma total de $\sum A = 30$.

Nueva Configuración

1. **Restricciones de Filas y Columnas:**
   - Cada fila i debe tener exactamente $r_i = B = 15$ casillas ocupadas.
   - Cada columna j debe contener exactamente $c_j = a_j$, con $A = \{6, 5, 4, 5, 3, 7\}$.
2. **Barcos:** Introducimos un barco para cada elemento en A:
   - Barco 1: longitud 6,
   - Barco 2: longitud 5,
   - Barco 3: longitud 4,
   - Barco 4: longitud 5,
   - Barco 5: longitud 3,
   - Barco 6: longitud 7.
3. **Restricciones Adicionales:**
   - Los barcos no pueden ser adyacentes entre sí (ni horizontal, ni vertical, ni diagonalmente).
   - Todos los barcos deben estar dentro del tablero, y la suma total de las celdas ocupadas debe ser $\sum A = 30$.

Solución del Problema de Batalla Naval

Dado el tablero con $n = 2$ filas y $m = 15$ columnas, encontramos la siguiente distribución:

- **Fila 1 (S_1):** Barcos $\{6, 5, 4\}$, que suman $6 + 5 + 4 = 15$.
- **Fila 2 (S_2):** Barcos $\{5, 3, 7\}$, que suman $5 + 3 + 7 = 15$.

Esto satisface las restricciones de las filas ($r_i = 15$) y las columnas ($c_j = a_j$).

Relación entre 3-Partition y Batalla Naval

1. Resolver el problema de *La Batalla Naval* (encontrar la disposición de los barcos en el tablero) equivale a resolver el problema de 3-Partition:
   - Cada fila representa un subconjunto de A.
   - La longitud de los barcos en cada fila corresponde a los elementos del subconjunto.
   - La suma de las longitudes de los barcos en cada fila debe ser igual a $B = 15$.
2. Si no existe una solución para el tablero naval, no existe una partición válida en el problema de 3-Partition.

Equivalencia de las instancias

- Si existe una solución para el problema *3-Partition*, entonces podemos construir una configuración válida del tablero para *La Batalla Naval*.
- Si existe una configuración válida para *La Batalla Naval*, entonces podemos construir una partición válida para el problema *3-Partition*.

Conclusión

Hemos demostrado que:

- *La Batalla Naval* pertenece a NP.
- Reducimos un problema NP-completo (*3-Partition*) a *La Batalla Naval* en tiempo polinomial.

Por lo tanto, *La Batalla Naval* es NP-completo.

3. Algoritmo de Backtracking

4. Ejemplos de ejecución

5. Algoritmo de aproximación

5.1. Descripción del algoritmo

A continuacion se presenta un algoritmo de aproximación para el problema de la batalla naval.

```
def find_index_of_max(list):

    i = 0
    max = list[0]

    for row, idx in list:
        if row > max:
            max = row
            i = idx
        i += 1

    return i

def verify_position(grid, row, col, rows, cols):

    if row < 0 or row >= len(grid) or col < 0 or col >= len(grid[0]):
        return False

```

```
    if rows[row] == 0 or cols[col] == 0:
        return False

    # if row < 0 or row >= len(grid) or col < 0 or col >= len(grid[0]):
    #     return False

    # Check if position is already occupied
    if grid[row][col] != None:
        return False
    # Check if position is adjacent to a ship
    for i in range(-1, 2):
        for j in range(-1, 2):
            if row + i >= 0 and row + i < len(grid) and col + j >= 0 and col + j < len(grid[0]):
                if grid[row + i][col + j] != None:
                    return False
    return True


def try_to_put_ship_horizontally_in_row(ship, grid, idx_f, rows, cols):

    row_to_put_ship = grid[idx_f] # Lista de la fila [3, None, None, None]
    for idx_c in range(len(row_to_put_ship)):

        can_place = True
        for k in range(ship):
            if not verify_position(grid, idx_f, idx_c + k, rows, cols): # SI PUEDO PONER EL BARCO EN ESA CELDA
                can_place = False
                break

        if can_place:
            for k in range(ship):
                grid[idx_f][idx_c + k] = ship
                rows[idx_f] -= 1
                cols[idx_c + k] -= 1

def aproximation(rows, cols, ships):
    grid = [[None] * len(cols) for _ in range(len(rows))]
    ships.sort(reverse=True)

    for ship in ships:
        indice_fila_max = rows.index(max(rows))
        indice_columna_max = cols.index(max(cols))

        if rows[indice_fila_max] >= cols[indice_columna_max]:

            for j in range(len(cols)):
                if verify_position(grid, indice_fila_max, j, rows, cols):

                    if (cols[j] >= ship) and (indice_fila_max + ship <= len(rows)):
                        can_place = True
                        for k in range(ship):
                            if not verify_position(grid, indice_fila_max, j + k, rows, cols):
                                can_place = False
                                break
                        if can_place:
                            for k in range(ship):
                                grid[indice_fila_max][j + k] = 1 # Se pone el barco
                                rows[indice_fila_max] -= 1
                                cols[j + k] -= 1
                            break
        else:
              for i in range(len(rows)):
                if verify_position(grid, i, indice_columna_max, rows, cols):

                    if (rows[i] >= ship) and (indice_columna_max + ship <= len(cols)):
                        can_place = True
```

```
85                          for k in range(ship):
86                              if not verify_position(grid, i + k, indice_columna_max,
    rows, cols):
87                                  can_place = False
88                                  break
89                          if can_place:
90                              for k in range(ship):
91                                  grid[i + k][indice_columna_max] =  1 # Se pone el
    barco
92                                  rows[i + k] -= 1
93                                  cols[indice_columna_max] -= 1
94                              break
95
96      return grid
```

5.2. Análisis del algoritmo

El algoritmo de aproximación para el problema de la batalla naval tiene una complejidad de $O((n+m) \cdot b \cdot k)$, donde n es el número de filas, m es el número de columnas del tablero, b es la longitud del barco y k es la cantidad de barcos que hay. Ya que el algoritmo recorre los barcos, compara cual tiene más demanda, si la columna o la fila, que luego define si poner el barco horizontal o vertical.

La aproximación que nos da el algoritmo es el siguiente:

demanda total: 11

Demanda cumplida: 2

demanda optima: 4

---

demanda total: 18

Demanda cumplida: 12

demanda optima: 12

---

demanda total: 53

Demanda cumplida: 18

demanda optima: 26

---

demanda total: 14

Demanda cumplida: 0

demanda optima: 6

---

demanda total: 40

Demanda cumplida: 20

demanda optima: 40

---

demanda total: 58

Demanda cumplida: 18

demanda optima: 46

---

demanda total: 67

Demanda cumplida: 18

demanda optima: 40

---

demanda total: 120

Demanda cumplida: 38

demanda optima: 104

---

demanda total: 247

Demanda cumplida: 62

demanda optima: 172

---

demanda total: 360

Demanda cumplida: 88

demanda optima: 202

Como podemos ver, no es el mejor algoritmo para acercarse a la solución óptima, que sería maximizar la demanda cumplida. Sin embargo, en algunos casos llega como por ejemplo en el segundo ejemplo ilustrado anteriormente

6. Medición empírica

7. Conclusiones