ESTRUCTURA DE DATOS Y ALGORITMOS 2

INFORME AVANCE SEMANA

PROYECTO #1 – 11208 AIRPLANE SCHEDULING

PROFESOR: JUAN GUILLERMO LALINDE PULIDO

KRISTIAN RESTREPO OSORIO

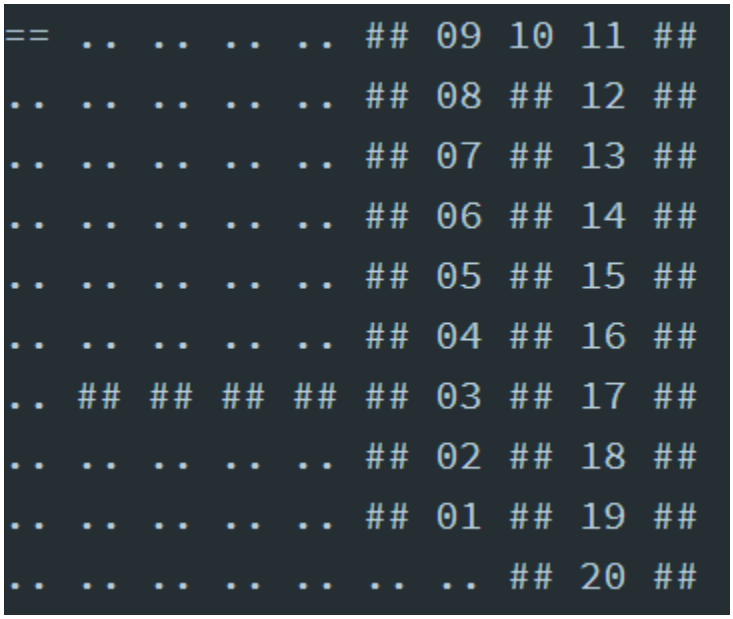
EAFIT

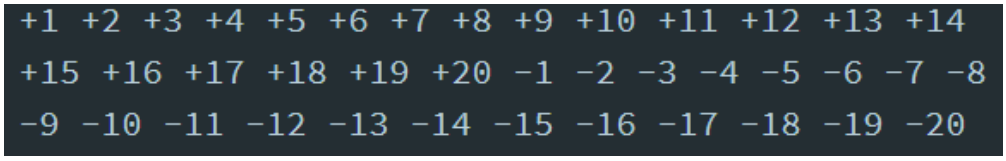
2023

Se encontró un error en el código que hacía que varias casos de prueba no retornaran la solución adecuada, lo que se corrigió, es que a la hora de que un avión despegara, se desmarcaba la casilla actual y se llamaba recursivamente a la función mirando el resto de eventos, si un evento no se cumplía, solo añadíamos el evento que se quitó nuevamente a la lista de eventos y se retornaba falso (en despegar), pero, la casilla que se había desmarcado a la hora de despegar la seguíamos dejando desmarcada, cuando ya ese evento queríamos deshacerlo, por lo tanto, se añadió que si un avión finalmente no pudo despegar, en esa posición se vuelve poner el "##", ya que no despegó.

Durante la semana se fueron desarrollando ciertas ideas, las cuales fueron:

1. Como el código con los casos de prueba planteados arroja respuestas correctas, pero hay casos en los que demora un tiempo considerable, se buscó formas de optimizar el mismo. Después de realizar un análisis a la función que realiza el backtracking, se evidenció que podría llegar a ser más optimo parquear los aviones en las casillas de mayor peso, esto debido a que, al ser bloqueados, se involucran en menor medida en la ruta hacia otro parqueadero, y podría llegar a ser más optimo para muchos casos, que simplemente recorrer la matriz y donde encuentre un parqueadero pone el avión sin ningún criterio que ayude a que se realicen menos pasos (como se estaba realizando para la entrega anterior). Veámoslo con un ejemplo:





Para el caso de este aeropuerto, en base a nuestra anterior lógica, se parquearían los aviones recorriendo la matriz de la esquina superior izquierda hasta la esquina inferior derecha, es decir, primero parqueamos un avión en el 09, luego en el 08, luego en el 07, y así sucesivamente, ya que, si se tapa de primero el 09, los parqueaderos del 10 al 20 quedarán marcados como inaccesibles, ya que no existe una ruta desde un punto entrada/despegue hacia los mismos. Como la lista de eventos ingresa primero los 20

aviones, al hacer el llamado recursivo, nos daremos cuenta de que no se pudieron ubicar todos los aviones satisfactoriamente, y prueba una nueva combinación, ubicando el primer avión no en 09, sino en 10. Y como vemos, tan solo parquear los aviones requiere de muchas correcciones en los llamados recursivos ya que lo estamos ubicando en una posición, sin ningún criterio, donde afecta a la ruta de muchos parqueaderos hacia el punto de salida. Ahora bien, si ubicamos nuestro primer avión en la casilla de mayor peso, todos los aviones se pudieran parquear satisfactoriamente, y ahora sí, se podría revisar si la posición es optima para los despegues, lo cual, con dicho criterio a la hora de ubicar los aviones, se reducen la cantidad de posibilidades para buscar una manera optima de parquear los aviones. Se sabe que la propiedad no es absoluta, y que no siempre parqueando los aviones en las casillas de mayor peso llevara a que se realicen menos pasos a la hora de la búsqueda de la solución, pero, es una solución con más razonamiento para establecer un posicionamiento.

Se realizaron los siguientes cambios en el código para implementar esta idea:

- La función que se encarga de ponderar, ahora, cada que mira en los vecinos de la casilla actual, y ese vecino es un parqueadero, además de asignarle el peso a la casilla, añadirlo a la deque, y marcarlo como visitado, también añade a una lista ("parqueaderos_mayor_peso"), una tupla, la cual, en primera posición tiene las coordenadas del parqueadero, y en segunda posición, tiene el peso de dicha casilla. Una vez ponderada toda la matriz, guarda en otra lista el contenido de "parqueaderos_mayor_peso", pero organizada de mayor a menor según la ponderación. Y finalmente, se retorna tanto la matriz ponderada, como la lista de parqueaderos organizados según su peso.
- A la función del backtracking se le pasa además la lista de parqueaderos organizada, y a la hora de realizar la búsqueda de donde parquear el evento actual, se recorre la lista de parqueaderos (ya no se recorre la matriz entera). También, para acceder a las coordenadas del parqueadero actual en el ciclo, se ingresa a la lista de parqueaderos en la posición actual, y se utiliza el primer elemento de la tupla que son las coordenadas del mismo, así, se puede seguir llevando el mismo funcionamiento que teníamos antes con el doble ciclo recorriendo toda la matriz, pero ahora, recorriendo únicamente la lista de parqueaderos.

2. Se planteo la idea de cambiar la matriz de ponderaciones que se tenia por un diccionario, el cual almacenara como llave las coordenadas de las casillas, y como valor una tupla, que tuviera el símbolo de la casilla (parqueadero, entrada/salida, bloqueo o pista), y su peso, esto con el fin de evitar ejecutar tantos ciclos creando matrices, cada que se re ponderaba el aeropuerto. Así, se eliminaba la función que creaba una matriz nueva con el tamaño de la matriz original y se enviaba una copia en cada reponderación del diccionario con los pesos de todas las casillas en "None" para ser ponderados nuevamente. Finalmente, se identifico que haciendo esto no se generaba ningún cambio, pues constantemente estábamos generando copias de la tabla de hash, lo cual era equivalente a crear matrices nuevas una y otra vez cada que se re ponderaba, por lo tanto, se ignoro dicha solución y se continuo con lo que se llevaba anteriormente.

Prueba de lo mencionado anteriormente **(código que se modificó y no se va a utilizar):**

```
from collections import deque
import time
## Función que evalúa si la posición a la cual se quiere acceder se
encuentra dentro de la matriz
def casilla_es_valido(filas,columnas,nueva_x,nueva_y):
    return nueva_x>=0 and nueva_y>=0 and nueva_x<filas and nueva_y<columnas

## Función que evalúa si la posición a la que se quiere acceder es una
casilla en negro (bloqueo)
def casilla_es_casilla_negra(matriz_simbolos,x,y):
    if matriz_simbolos[x][y] == '##':
        return True

## Función que evalúa si la posición a la que se quiere acceder es una
casilla en blanco (pista)
def casilla_es_blanca(matriz_simbolos,x,y):
    if matriz_simbolos[x][y] == '..':
        return True

## Función que evalúa si la posición a la que se quiere acceder es un
parqueadero
def casilla_es_parqueadero(matriz_simbolos,x,y):
    return matriz_simbolos[x][y].isdigit()

## Función que evalúa si la posición a la que se quiere acceder es una
casilla gris (entrada-salida ó aeropuerto)
def casilla_es_aeropuerto(matriz_simbolos,x,y):
    if matriz_simbolos[x][y] == '==':
        return True

## Función en la que se asigna un peso a cada casilla desde su respectivo
aeropuerto (entrada/salida). Primero, se guarda en el deque (pesos) las
coordenadas de los puntos
## entradas/salidas (antes de ingresar a la funcion), luego, si la posicion
adyasente a mi actual es casilla en blanco, añado esta en el inicio del
deque, y le asigno el mismo peso de mi actual,
## si es parqueadero, lo añado en el final de mi deque, y le asigno el peso
de mi actual + 1, y si es bloqueo, no lo añado al deque y le asigno el valor
de infinito (-11).
def ponderar (matriz_simbolos,hashh,pesos,movimientos,filas,columnas):
    visitados = deque()
    parqueaderos_mayor_peso = []
    while(len(pesos)>0):
        actual_coordenada = pesos.popleft()
```

```
        for i in range(len(movimientos)):
            nueva_x = actual_coordenada[0] + movimientos[i][0]
            nueva_y = actual_coordenada[1] + movimientos[i][1]
            if casilla_es_valido(filas,columnas,nueva_x,nueva_y) and
(nueva_x,nueva_y) not in visitados:
                if
casilla_es_casilla_negra(matriz_simbolos,nueva_x,nueva_y):
                    hashh[(nueva_x,nueva_y)] =
(matriz_simbolos[nueva_x][nueva_y],-11)
                    visitados.append((nueva_x,nueva_y))
                if casilla_es_blanca(matriz_simbolos,nueva_x,nueva_y):
                    hashh[(nueva_x,nueva_y)]=
(matriz_simbolos[nueva_x][nueva_y],hashh[(actual_coordenada[0],actual_coorde
nada[1])][1])
                    pesos.appendleft((nueva_x,nueva_y))
                    visitados.append((nueva_x,nueva_y))
                if casilla_es_parqueadero(matriz_simbolos, nueva_x,nueva_y):
                    hashh[(nueva_x,nueva_y)]=
(matriz_simbolos[nueva_x][nueva_y],hashh[(actual_coordenada[0],actual_coorde
nada[1])][1]+1)
                    pesos.append((nueva_x,nueva_y))
                    visitados.append((nueva_x,nueva_y))
                    parqueaderos_mayor_peso.append(((nueva_x,nueva_y),hashh[
(nueva_x,nueva_y)][1]))
    sorted_list = sorted(parqueaderos_mayor_peso,key = lambda x:
x[1],reverse = True)
    return hashh,sorted_list


## Función que guarda la matriz vacía a ponderar, crea el deque, y agrega al
mismo las coordenadas de los puntos entradas/salidas, para posteriormente,
## enviarlo a la función ponderar para asignar peso a cada casilla. La lista
de movimientos indica como se puede mover el avión.
def pre_ponderacion(matriz_simbolos,filas,columnas,hashh,pesos):
    movimientos = [[0,1],[1,0],[0,-1],[-1,0]]
    return ponderar(matriz_simbolos,hashh,pesos,movimientos,filas,columnas)

def resolver_problema(matriz_simbolos, hashh, eventos, filas, columnas,
lista_solucion, diccionario,hashh_aux,pesos_aux,pq):
    if len(eventos)==0:
        return True
    else:
        evento_actual = eventos.popleft()
        if evento_actual>0:
            for i in range(len(pq)):
```

```
                if hashh[(pq[i][0][0],pq[i][0][1])][1] != None:
                    if
casilla_es_parqueadero(matriz_simbolos,pq[i][0][0],pq[i][0][1]) and
hashh[(pq[i][0][0],pq[i][0][1])][1]>0:
                        temporal = matriz_simbolos[pq[i][0][0]][pq[i][0][1]]
                        diccionario[evento_actual] =
((pq[i][0][0],pq[i][0][1]),temporal)
                        matriz_simbolos[pq[i][0][0]][pq[i][0][1]] = '##'
                        lista_solucion.append(temporal)
                        hashh,pq =
pre_ponderacion(matriz_simbolos,filas,columnas,hashh_aux.copy(),pesos_aux.co
py())
                        if
resolver_problema(matriz_simbolos,hashh,eventos,filas,columnas,lista_solucio
n,diccionario,hashh_aux,pesos_aux,pq):
                            return True
                        else:
                            lista_solucion.pop()
                            matriz_simbolos[diccionario[evento_actual][0][0]
][diccionario[evento_actual][0][1]] = diccionario[evento_actual][1]
                            hashh,pq =
pre_ponderacion(matriz_simbolos,filas,columnas,hashh_aux.copy(),pesos_aux.co
py())
                            diccionario.pop(evento_actual)
            eventos.appendleft(evento_actual)
            return False
        else:
            evento_actual_2 = abs(evento_actual)
            movimientos = [[0,1],[1,0],[0,-1],[-1,0]]
            for i in range(len(movimientos)):
                nueva_x = diccionario[evento_actual_2][0][0] +
movimientos[i][0]
                nueva_y = diccionario[evento_actual_2][0][1] +
movimientos[i][1]
                if casilla_es_valido(filas,columnas,nueva_x,nueva_y):
                    if hashh[(nueva_x,nueva_y)][1] != None and
hashh[(nueva_x,nueva_y)][1]!=-11:
                        if hashh[(nueva_x,nueva_y)][1]>=0:
                            matriz_simbolos[diccionario[evento_actual_2][0][
0]][diccionario[evento_actual_2][0][1]] = diccionario[evento_actual_2][1]
                            hashh,pq =
pre_ponderacion(matriz_simbolos,filas,columnas,hashh_aux.copy(),pesos_aux.co
py())
```

```
                            if
resolver_problema(matriz_simbolos,hashh,eventos,filas,columnas,lista_solucio
n,diccionario,hashh_aux,pesos_aux,pq):
                                return True
                            break
            matriz_simbolos[diccionario[evento_actual_2][0][0]][diccionario[
evento_actual_2][0][1]] = "##"
            eventos.appendleft(evento_actual)
            return False




## Función principal, que se encarga de crear la matriz de simbolos
(aeropuerto en general), almacenar los eventos en una lista,
## generar la primera ponderación, hacer el llamado al algoritmo que
soluciona el problema e imprimir la solución.
def principal():
    numero_caso = 0
    while(True):
        nfc = input().split()
        numero_aviones = int(nfc[0])
        if numero_aviones == 0:
            break
        else:
            numero_filas,numero_columnas = int(nfc[1]),int(nfc[2])
            hash_simbolos = {}
            pesos = deque()
            lista_solucion = []
            diccionario = {}
            numero_caso+=1
            matriz_simbolos = []
            for i in range(numero_filas):
                fila = input().split()
                matriz_simbolos.append(fila)
                for j in range (len(fila)):
                    if fila[j] == "==":
                        hash_simbolos[(i,j)]=(fila[j],0)
                        pesos.append((i,j))
                    else:
                        hash_simbolos[(i,j)]=(fila[j],None)
            pesos_aux = pesos.copy()
            hash_simbolos_aux = hash_simbolos.copy()
            eventos = deque(map(int, input().split()))
```

```
            ponderacion_inicial,pq=
pre_ponderacion(matriz_simbolos,numero_filas,numero_columnas,
hash_simbolos,pesos)
            inicio = time.time()
            resultado = resolver_problema(matriz_simbolos,
ponderacion_inicial, eventos, numero_filas, numero_columnas, lista_solucion,
diccionario,hash_simbolos_aux,pesos_aux,pq)
            if resultado:
                print("Case {}: Yes".format(numero_caso))
                for i in range(len(lista_solucion)):
                    print(lista_solucion[i], end=" ")
                print()
            else:
                print("Case {}: No".format(numero_caso))
            final = time.time()
            print(final-inicio)

if __name__ == '__main__':
    principal()
```

De hecho, se verificó que esta solución no optimizaba mucho el código en comparación con el que se mandó en un principio, tomando los tiempos de ambos con un caso de prueba en específico, que es del aeropuerto con el que se realizo el ejemplo anteriormente, de estos, obtuvimos los siguientes resultados (tiempo en segundos):

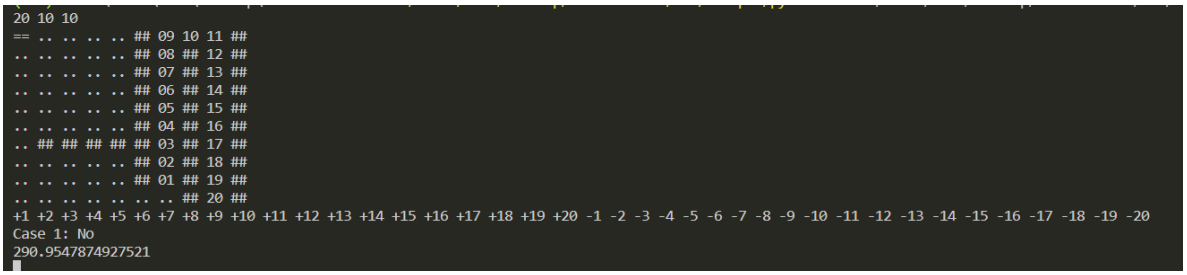
CODIGO ANTERIOR (sin ninguna modificación de prioridad a la hora de parquear, etc.):

```
20 10 10
== .. .. .. .. ## 09 10 11 ##
.. .. .. .. .. ## 08 ## 12 ##
.. .. .. .. .. ## 07 ## 13 ##
.. .. .. .. .. ## 06 ## 14 ##
.. .. .. .. .. ## 05 ## 15 ##
.. .. .. .. .. ## 04 ## 16 ##
.. ## ## ## ## ## 03 ## 17 ##
.. .. .. .. .. ## 02 ## 18 ##
.. .. .. .. .. ## 01 ## 19 ##
.. .. .. .. .. .. .. ## 20 ##
+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11 +12 +13 +14 +15 +16 +17 +18 +19 +20 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20
Case 1: No
307.38053846359253
```

CODIGO MODIFICADO 1 (con el diccionario, y la prioridad):

```
20 10 10
== .. .. .. .. ## 09 10 11 ##
.. .. .. .. .. ## 08 ## 12 ##
.. .. .. .. .. ## 07 ## 13 ##
.. .. .. .. .. ## 06 ## 14 ##
.. .. .. .. .. ## 05 ## 15 ##
.. .. .. .. .. ## 04 ## 16 ##
.. ## ## ## ## ## 03 ## 17 ##
.. .. .. .. .. ## 02 ## 18 ##
.. .. .. .. .. ## 01 ## 19 ##
.. .. .. .. .. .. .. ## 20 ##
+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11 +12 +13 +14 +15 +16 +17 +18 +19 +20 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20
Case 1: No
298.7348699569702
```

CODIGO MODIFICADO 2 (únicamente con la prioridad al parquear los aviones)



Se puede ver cómo, al seguir utilizando matrices ponderadas, y parqueando los aviones con la propiedad del de mayor peso, se obtiene el mejor tiempo, por lo tanto, se dejó únicamente esa solución como implementación realizada al código.

También se empezó a intentar cambiar el código de lenguaje, en este caso, a C++, puesto que puede haber una mejora en cuanto a los tiempos de ejecución del programa, estando en un lenguaje de programación compilado y no interpretado.

Se realizaron algunos avances en cuanto a la implementación del código en este lenguaje, pero, a razón de la falta de experiencia en el mismo, no se ha podido implementar en su totalidad el proceso que se lleva hasta este momento en Python. Estoy capacitándome en el uso de este lenguaje (del cual solo tuve experiencia en el curso de lenguajes de programación el semestre pasado), y además, estoy aprovechando herramientas como “ChatGPT”, y las distintas plataformas que ofrecen tutoriales y cursos para la aceleración del aprendizaje.

En conclusión, finalmente al código se le agregó una “prioridad” a la hora de encontrar un parqueadero en donde ubicar el avión, esto mejoró un poco en tiempo a la hora de buscar una solución al problema, además, de que también se solucionó un problema que hacía que no retornara la respuesta correcta en algunos casos de prueba. (Se siguen probando casos de pruebas en busca de fallas, y minimizar los tiempos para algunos casos que son excesivos).