ESTRUCTURA DE DATOS Y ALGORITMOS 2

PROYECTO #1 – 11208 AIRPLANE SCHEDULING

PROFESOR: JUAN GUILLERMO LALINDE PULIDO

KRISTIAN RESTREPO OSORIO

EAFIT

2023

Descripción del algoritmo. ¿Por qué funciona?

El algoritmo se desarrolló en Python 3, en el entorno de desarrollo de Visual Basic.

Una de las bases del funcionamiento del algoritmo es la ponderación de las casillas. Estas nos permiten identificar dos situaciones en cada casilla que nos posicionamos: que existe un camino, y que es el más corto. Así, bajo este concepto teórico, podemos llegar a la conclusión de que si una casilla está ponderada a la hora de hacerle la revisión significa que existe un camino hacia su punto de salida. Pero de aquí surge una pregunta, ¿Cómo se pondera toda la matriz?, primero, se crea una matriz de pesos, del mismo tamaño que la matriz de símbolos, con valor por defecto de NONE, posteriormente, iteramos sobre la matriz original buscando las coordenadas de los puntos entradas/salidas (==) del aeropuerto para añadir estos a un deque (una cola doblemente abierta) y además, en la matriz de pesos, en dichas coordenadas, asignarles el valor de 0, ya que no se necesita "pisar" ningún parqueadero para llegar allí. Una vez teniendo en el deque las coordenadas iniciales, se llama a la función ponderar para asignarle un peso a todas las casillas bajo cierto comportamiento.

Mientras existan casillas que evaluar, siempre se van a ejecutar los siguientes pasos:

1. Se ubica en la casilla actual, y se mira en cada vecino de que tipo es.
    - Si es un bloqueo, le asigna de peso a esa casilla infinito (en el código está como -11), y no agregamos a ese vecino a el deque con las casillas por evaluar.
    - Si es una casilla en blanco, le asignamos a ese vecino el mismo peso de nuestra casilla actual, y lo agregamos al inicio del deque, así, estaríamos asegurando que vamos a evaluar primero las casillas con el mismo peso.
    - Si es parqueadero, le asignamos a ese vecino el peso de nuestra casilla actual + 1, (ya que el camino más corto para llegar a dicha posición es "pasando" por un parqueadero), y lo agregamos al final del deque, para que, una vez acabada la evaluación de los adyacentes con el mismo peso, se empiece a evaluar los adyacentes con un peso mayor, y así sucesivamente.

Los puntos que son inaccesibles los deja como NONE, ya que como hay bloqueos que los rodean, no se puede acceder a dichas casillas, con lo que no hay una conexión entre un punto entrada/salida y las mismas.

Cuando todas las casillas de las matrices hayan sido evaluadas, se retorna la matriz de pesos.

Cabe recalcar que el orden del ingreso de las casillas al deque es de suma importancia, ya que nos permiten asegurar que el peso que va a tener dicha casilla va a ser el peso del camino mas corto hacia la misma (ya que miramos los pesos de manera ascendente).

Esta misma función la utilizamos para recalcular los pesos una vez ingresan aviones, ya que cuando ubicamos el avión, marcamos dicha casilla en la matriz original como un bloqueo, y se vuelve a hacer todo el proceso de ponderación, como si esa casilla estuviera en bloqueo desde un inicio. Se sobre entiende que no es una manera optima de abordar el problema, puesto que estamos evaluando los pesos de toda la matriz cada que se aterriza un avión, mientras que podríamos estar evaluando únicamente una parte de dicha matriz, que son los caminos que se ven afectados por el parqueo de

dicho avión, pero se pretende en un futuro darle una mejor implementación al algoritmo, empezando por el cambio de esta "re ponderación".

Así, una vez teniendo los pesos cada que aterriza un avión, podemos saber si un avión puede parquear en un lugar (que esté ponderado con un numero > 0 y que sea parqueadero), como también, si puede despegar, mirando si alguno de sus vecinos esta ponderado, con lo que sabemos que existe una ruta desde ese punto a un lugar de salida (Esto lo realiza la función de backtracking explicada más abajo).

En orden de ejecución del código, primero llamamos a la función principal, la cual lee el input y crea la matriz de símbolos, luego se crea y pondera la matriz inicial de pesos, y luego llamamos a la función del back tracking "resolver_problema" para establecer los puestos de parqueo para que cada avión pueda ingresar y salir sin chocarse con algún "obstáculo". Se ingresa 0 para dejar de ingresar casos.

Con lo que el algoritmo, principalmente gracias a la función de ponderar y a la del backtracking, funciona (en la mayoría de los casos probados, solo en uno se quedó en infinito), puesto que retorna, si ha de existir, una solución valida para ubicar los aviones, y de no ser así, retorna que no existe solución para dicho aeropuerto con esa lista de eventos.

Descripción del mecanismo de backtracking: ¿Cuál es el criterio que utiliza para definir que no tiene sentido continuar con la exploración de la solución y se debe deshacer la última decisión tomada?

El criterio que se utiliza para definir que no tiene sentido continuar con la exploración y se debe deshacer la ultima decisión tomada, es cuando no hay lugar donde parquear el avión, o cuando el avión/es que se parquearon en cierta posición anteriormente, a la hora de realizar la salida de alguno, este no tenga una conexión con un punto de salida, esto lo verificamos si en los vecinos de la posición actual en la que nos encontramos, alguno tiene peso, si alguno tiene peso mayor o igual a 0 significa que hay un camino desde la casilla actual al punto de salida, con lo que en la posición actual también existe dicho camino para salir, así, finalmente retornando "True", y por el contrario, si ninguno de sus vecinos tiene un peso mayor o igual a 0 significa que tenemos que devolvernos a parquear los aviones en un lugar distinto para encontrar ubicaciones que permitan las salidas de los aviones, en este caso se retorna "False".

Para entender un poco más sobre la función del backtracking implementada en el algoritmo ("resolver_problema"), se explica a continuación paso a paso:

Establecemos un caso base para la recursión, que en este caso es cuando se haya recorrido toda la lista de eventos, ya que se logró parquear y todos esos aviones tuvieron una salida satisfactoria, con lo que se resolvió el problema adecuadamente.

Luego, recorremos la lista de eventos:

1. Se saca el evento actual
2. Si el evento actual es mayor a 0, es decir, cuando un avión está aterrizando:
    - Se recorre la matriz de símbolos buscando un parqueadero ponderado (>0)

- Al encontrar el parqueadero, se guarda el numero del parqueadero en una variable temporal
- Se añade a un diccionario como llave el evento actual y como valor, las coordenadas de donde se parqueó el avión, y que numero tiene dicho parqueadero. Guardamos estos valores en un diccionario con la razón de que a la hora de hacer el backtracking, podamos devolver el valor que tenia dicha casilla para cambiar el parqueadero del avión asignado (en caso de ser una solución no factible).
- La matriz de símbolos en la posición donde vamos a parquear la marcamos como un bloqueo, ya que, un avión parqueado para los siguientes eventos representa un lugar por el que no se puede pasar ni parquear, lo cual es equivalente.
- Se añade el parqueadero a la lista de soluciones.
- Se re pondera la matriz de pesos, ya que dicho bloqueo cambia el peso de las casillas que su camino más corto a la salida pasaba por donde se parqueó el avión.
- Ahora se hace el llamado recursivo a la función. Si el llamado recursivo de la función retorna “True”, significa que no se debe modificar donde se parqueó el avión, porque se recorrió la lista de eventos completamente y los aviones que parquearon, pudieron salir satisfactoriamente, con lo que se retorna “True” en el cuerpo del condicional. De lo contrario, si el llamado recursivo retorna “False”, significa que no se encontró parqueadero para un avión o algún avión parqueado en la lista de eventos no pudo despegar satisfactoriamente, con lo que se debe acomodar nuevamente el avión,
Y por ello, se elimina dicha posición donde se parqueó el avión de la lista de soluciones, marcamos la casilla de la matriz de símbolos nuevamente con el parqueadero que tenía en esa posición antes de que se parqueara el avión, se re pondera la matriz con el parqueadero nuevamente libre, y se elimina del diccionario dicho evento, ya que no fue factible, y dicho avión no va a parquearse en el lugar que se le había asignado anteriormente. Así luego, ver el siguiente parqueadero disponible y hacer el mismo proceso.

- Si no se encuentra parqueadero para un avión en toda la matriz, se añade el evento actual a la lista de eventos (al inicio, porque se tiene que reevaluar ese evento), y se retorna “False”, para reajustar el posicionamiento de los aviones parqueados o bien, al no haber parqueaderos para el ingreso de aviones definitivamente, retornar que no hay solución.

3. Si el evento actual es menor a 0, es decir, cuando un avión está despegando:
   - Se guarda en una variable el valor absoluto de mi evento actual, ya que el evento actual en el diccionario se guardó con el valor positivo cuando el avión aterrizó.
   - Para cada vecino de la posición actual del avión que va a salir (obtengo la posición accediendo al valor del diccionario), si es una casilla valida (que se encuentre dentro del rango de la matriz), está ponderada, es distinto de infinito y su peso es mayor o igual a 0, significa que dicho vecino tiene un camino hacia el punto de salida, con lo que mi casilla actual también, por lo tanto, vuelvo a marcar dicha casilla como parqueadero disponible y pondero nuevamente la matriz en base a este parqueadero.
   Como el avión salió satisfactoriamente, hago el llamado recursivo a la función para que siga evaluando los elementos de la lista de eventos, si se recorre la lista de eventos

satisfactoriamente, se retorna “True” (por el caso base), con lo que se retorna “True” nuevamente ya que se accede al cuerpo del condicional, para indicar que dicho avión despego satisfactoriamente. Si el llamado recursivo retorna “False”, al ser un despegue, no se debe seguir verificando si existe un camino para los vecinos, puesto que ya sabemos que como se parquearon los aviones no fue una solución factible, así que retornará para todos los vecinos “False”, por lo tanto, para que no se evalué el resto de los vecinos, se rompe el ciclo, añadimos el evento actual al inicio de la lista de eventos, ya que se debe reevaluar dicho evento posicionando los aviones del siguiente modo, y se retorna “False”.

Finalmente, la función del backtracking retorna “True” o “False”, según si pudo ubicar los aviones satisfactoriamente de manera que todos pudiesen despegar según la lista de eventos. Este valor llega a la función principal e imprime las ubicaciones de cada avión en caso de un “True” o un “No” en caso de no ser posible.

¿Cómo cambiaría el algoritmo si en lugar de preguntar por una las soluciones preguntara por todas las soluciones? Justifique su respuesta.

Una vez llegado al caso base, en nuestro algoritmo original retornamos “True”, porque encontró una solución, y, por lo tanto, finaliza la función. Ahora bien, si se quiere buscar todas las posibles soluciones, debemos cambiar que, una vez encontrado un posicionamiento correcto de los aviones, es decir, si nos encontramos en el caso base, se imprima la lista de solución, y siga buscando posibles soluciones, ya que al no retornar “True” al anterior llamado recursivo, este seguirá buscando una posible combinación de aviones parqueados, así, tomara dicha lista de parqueos que es factible, la imprimirá, y retorna como si la misma no fuera factible, continuando el llamado recursivo.

Importante:

El código pasa todas las pruebas de uDebug para las matrices pequeñas en un tiempo satisfactorio, al igual que para las matrices grandes, excepto para el caso 2 y el caso 21, los cuales tarda mucho en procesar (el caso 21 tarda unos 5 minutos, el 2 se queda cargando). Con lo que el algoritmo para algunos casos no cumple con el límite de tiempo, y por esto mismo, no pasa la prueba de Online Judge.

Algunos casos de prueba de uDebug de las matrices grandes, la respuesta que tienen es errónea, por lo tanto, se realizó pruebas de escritorio para estas matrices con la respuesta de mi código, y la solución satisfacía el problema planteado.

Material extra: (Pseudocodigos realizados en el proceso de la solucion del problema, para implementarlos posteriormente en el codigo):