Diseño y Análisis de Algoritmos: Proyecto - Parte 1

Camilo Morillo Cervantes (202015224)
Juan Sebastián Alegría (202011282)

Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes

1 de abril de 2022

I. Algoritmo de solución.

El algoritmo de solución elegido utiliza tanto recursividad como memoization, esta ultima para no tener que recorrer nuevamente caminos que se recorrieron con anterioridad.
La función solución tiene 7 parámetros de entrada; el numero de pisos, el numero de cuartos, una lista con la energía de cada piso, un diccionario con los portales, el piso actual, el cuarto actual y finalmente un diccionario con los caminos ya recorridos. Esta función recursiva tiene dos casos bases, cuando estamos en el ultimo piso y cuando el piso en el que estamos no tiene la entrada de ningún portal.
En cuanto al caso recursivo este se ejecutara sobre todos los portales de dicho piso, siempre y cuando ese camino no se encuentre en el diccionario de caminos recorridos, por lo tanto por cada portal que este en el piso actual haremos un llamado a la función nuevamente. Lo que cambia de una función recursiva a otra son los parámetros de piso actual y cuarto actual con tal de ejecutar la función en dichos pisos. Dichos llamados de la función recursiva se guardan en una variable llamada costo a la cual se le suma el costo de moverse hacia el cuarto del portal. A partir de los diversos costos conseguidos se conseguirá el mínimo el cual corresponderá al camino de menor costo.
A la vez cada camino de portales recorrido se guardará dentro del diccionario de caminos, para que en caso de que dicho camino se tenga que recorrer de nuevo solamente conseguir el coste de este sin tener que hacer nuevamente su recorrido. A partir de lo anterior tenemos las siguientes poscondiciones y precondiciones de la función:

$\{Q : numPisos, numCuartos, pisActual, cuarActual, portales.size \geq 0 \ \wedge energias.size = numPisos \wedge (\forall k | 0 \leq k < numPisos : energias[k] \geq 0) \wedge portales.size < numPisos \wedge (\forall k, j | 0 \leq k < numPisos \wedge 0 \leq j < 3 : portales[k].size = 3 \wedge portales[k][j] \geq 1)$

$\{R : minimo = shortestPathValue((0,0),(numPisos-1, numCuartos-1))\}$

Se descartó la implementación de una solución alternativa que consistía en utilizar el algoritmo de Dijkstra para calcular el mínimo camino entre los cuartos de la torre. Para este objetivo se crearía un grafo en el cuál cada nodo era representado por una coordenada (cuarto, piso) y conectaba a los demás con un vértice que tenía el costo que requería ir a otro nodo ya sea en el mismo piso o a través de un portal hasta llegar al cuarto final.
La razón por la cuál no se implementó este algoritmo es debido a que requería crear una estructura de datos con una complejidad espacial mucho mayor al diccionario que actualmente utilizamos para almacenar los costos mínimos entre portales. Esta estructura tendría que almacenar no solo los costos mínimos entre portales sino todas las posibles combinaciones de rutas desde el cuarto inicial hasta el final.

II. Análisis de complejidades.

En este caso podemos dividir nuestro programa en dos grandes partes; la lectura y escritura de datos y la función solución. A continuación podemos ver el código del método main() el cual realiza la lectura, escritura de datos y el llamado a la función solución:

```
def main():
    input = sys.stdin.readline
    numberCases = int(input())  # Numero de casos de prueba
    solutions = []
    for c in range(numberCases):
        data = list(map(int, (input().split())))
        numPisos, numCuartos, numPortales = data[0], data[1], data[2]
        energia, portales = list(map(int, (input().split()))), {}
        for _ in range(numPortales):
            infoPortal = input().split()
            piso_i, cuarto_i, piso_f, cuarto_f = tuple(int(e) for e in infoPortal)
            if piso_i - 1 not in portales:
                portales[piso_i - 1] = []
            portales[piso_i - 1].append([cuarto_i - 1, piso_f - 1, cuarto_f - 1])
        # Se llama a la funcion solucion con los datos anteriores
        sol = torreTp(numPisos, numCuartos, energia, portales, 0, 0, {})
        if sol == math.inf:  # Si el coste es infinito entonces la solucion no existe
            solutions.append("NO_EXISTE")
        else:  # Se agrega la solucion a una lista de soluciones
            solutions.append(sol)
    # Output
    for i in solutions:
        print(i)
```

Sea n el numero de casos y m el numero de portales, además de las constantes c_1, c_2, c_3 para asignaciones, comparaciones y operaciones respectivamente (append contara como asignación por su complejidad constante), entonces tenemos la siguiente complejidad :

$T(n,m) = 3c_1 + n(8c_1 + c_2) + nm(7c_1) + n(torreTp)$

Si no tenemos en cuenta la complejidad de la función solución torreTp() entonces la complejidad de la lectura y escritura de datos es de:

$O(nm), \Omega(nm), \Theta(nm)$

Ahora calcularemos la complejidad de la función solución, la cual se encuentra a continuación:

```
def torreTp(numPisos, numCuartos, energia, portales, pisActual, cuarActual, costosMin):
    if pisActual == numPisos - 1:  # Caso base: si nos encontramos en el piso final
        return (numCuartos - 1 - cuarActual) * energia[pisActual]
    # Caso base: no hay portales en el piso actual (dicho camino no es solucion)
    elif pisActual not in portales:
        return math.inf
    else:
        # Variable iniciada en infinito que da el coste del camino con coste minimo
        minimo = math.inf
        # Se verifica cada portal en el piso actual y los caminos que salen de este
        for portal in portales[pisActual]:
            portalKey = str(pisActual) + ',' + str(portal[0])
            # Si no se recorrio el camino se recorre recursivamente
            if portalKey not in costosMin:
                costo = (torreTp(numPisos, numCuartos, energia, portales, portal[1], portal[2],
                costosMin) + (abs(cuarActual - (portal[0]))) * energia[pisActual])
                costosMin[portalKey] = costo - (abs(cuarActual - (portal[0]))) * energia[pisActual]
            else:  # Si se recorrio el camino se saca el costo del diccionario
                costo = costosMin[portalKey] + (abs(cuarActual - (portal[0]))) * energia[pisActual]
            if costo == 0:  # Si el costo el 0 ese sera ya el costo menor ya que no hay costo negativo
                return 0
            if costo < minimo:  # Se retorna el coste minimo de todos estos posibles caminos
                minimo = costo
        return minimo
```

Sea p el numero de pisos y c el numero de cuartos entonces el peor escenario es cuando hay entradas de portales en todos los cuartos de todos los pisos excepto el ultimo piso además de que solo te teletransporten a un solo piso por encima. Teniendo esto en cuenta entonces tenemos la complejidad $O(c^p)$ ya que se comprueban todos los portales de los cuartos de cada piso. Sin embargo como dijimos hay algunos que no se comprueban debido a la memoization lo cual mejora un poco la anterior complejidad. En general entonces tendríamos una complejidad de $O(nc^p)$ donde n es el numero de casos, c el numero de cuartos y p el numero de pisos.

III. Otros escenarios.

Escenario 1 (Portales bidireccionales): Este escenario cambia el problema en el sentido de que ahora es posible acceder a ciertos pisos que antes puede que no sea posible acceder. Ahora podemos subir a un piso y luego bajar a otro piso al cual no había acceso alguno subiendo desde algún portal. Para tener en cuenta estos casos debemos cambiar la implementación para que no solo se verifiquen los portales que tienen entrada en el piso actual si no que también tienen salida en ese piso (la cual ahora también cuenta como entrada).

Escenario 2 (Numero de rutas existentes): En este caso ya no tenemos que sumar los costos si no por cada recursión ir sumándole 1 al total hasta llegar a los casos bases los cuales retornaran 0 si se llega al piso final o -1 si no se llego a un cuarto con la entrada de un portal. Esto ultimo implica le restaremos 1 a nuestro total al cual le habíamos sumado anteriormente 1 en caso de que dicha ruta no nos lleve a al cuarto final balanceando así nuevamente la suma en caso de que la ruta no sirva.

Escenario 3 (Ruta óptima para visitar todos los cuartos): En este escenario se presenta el problema del movimiento entre cuartos, ya que debemos escoger el movimiento mas eficiente entre estos que nos lleve a recorrer todos los cuartos del piso y posteriormente entrar en el portal adecuado. Una vez encontrado el movimiento mas óptimo dentro de un piso entonces debemos tomar el portal que nos lleve al piso inmediatamente superior ya que al ser portales unidireccionales el hecho de ir a pisos mucho más altos implica saltarnos varios pisos y cuartos. Por ello deben estar conectados todos los pisos en orden ascendente en caso contrario no existe dicha ruta óptima para visitar todos los cuartos. De esta forma yendo de piso en piso y recorriendo de forma mas eficiente los cuartos de ese piso para tomar el siguiente portal encontraremos la ruta mas eficiente que nos lleve al ultimo cuarto pasando por todos los anteriores.