Universidad Simon Bolivar
Curso: CI2613 / Algoritmos y Estructuras III
Trimestre: Septiembre-Diciembre, 2022
Profesor: Wilmer Bandres
Estudiante: Junior Miguel Lara Torres - Carnet: 17-10303

Tarea 3 (20 pts)

1. Teoría (Algoritmos) (5 pts)

- Escriba un pseudocodigo del esquema general de búsqueda de caminos mínimos dado en clase. Explique porqué el esquema funciona para encontrar caminos de costo mínimo desde una sola fuente siempre y cuando no hayan ciclos de costo negativo.

```
FunGeneral ( Grafo G = (V, E), fuente u, funCostos c ) {
    d[v] ← ∞  ∀v ∈ V;
    d[v] = 0;
    while ( ∃e = (x, y) : d[y] > d[x] + c(e) ) {
        e ← Elegir e con la condición del while.
        d[y] ← d[x] + c(e)
    }
    return d;
}
```

El esquema funciona de manera ineficiente ya que puede tomar valores exponenciales en el numero de nodo, pero debido al condicional del while tenemos que al encontrar un nodo tal que se pueda relajar", termino que se agrega en algoritmos posteriores, básicamente encontrar una menor distancia para el nodo procesado. La condición $d[y] > d[x]+c(e)$ que ocurren por la cantidad de arcos permite obtener al final de procesar todos los arcos las distancias mínimas desde la fuente a todo nodo del grafo siempre y cuando se asuma un grafo conexo acíclico. Si en el trayecto existe un ciclo/circuito de costo negativo podemos decir entonces que para una primera vez ese ciclo/circuito lo recorremos 3 veces, pero en otra oportunidad podemos decir que ese ciclo se recorrido 4 o mas veces ,por tanto no existe un camino de costo mínimo a pesar de que existe un camino de u(fuente) a v(cualquier otro nodo del grafo). Por tanto, este algoritmo funciona para grafos con arcos de costos mayores o iguales a 0.

- Escriba el algoritmo de Dijkstra y explique su correctitud y complejidad.

```
Dijkstra ( Grafo G = (V,E). funCostos c, fuente u ) {
    S ← ∅;
    Q ← V;
    d[v] ← ∞  ∀v ∈ V;
    d[u] ← 0;
    while !Q.empty() {
        v ← Extraer nodo mínimo en Q;
        S ← S ∪ {v};
        for e = (v,x) ∈ δ+(v) {
            if d[x] > d[v] + c(e) {
                d[x] = d[v] + c[e];
            }
        }
    }
    return d;
}
```

Complejidad:
Teniendo en cuenta la implementación para la Priority Queue con un árbol rojo-negro/binario, las operaciones toman tiempo $lg_2(]V])$, la altura del árbol, es actualizar la relación de prioridad en el Queue cuando se hace el paso de relajación.

Ahora, para el for interno ubicado dentro del while(Paso de relajación), ocurre que se debe quitar he insertar el nuevo nodo actualizado entonces tiene complejidad $O(2lg_2(|V|))$. Luego extraer el nodo mínimo del Q toma tiempo $O(lg_2(|V|))$ pues debemos ver y eliminar el nodo lo que necesita re-balancear. El while toma tiempo $O(|V|)$. Ahora, el while mas el for tenemos que se toma en total $O(|V|lg_2(|V|)+|E|lg_2(|V|)) = O((|V|+|E|)Lg_2(|V|))$.

Correctitud:
La correctitud se limita a probar por inducción en el costo mínimo de los caminos hacia los nodos del grafo para la proposición siguiente:

Caminos Óptimos de Dijkstra:

Cada vez que Dijkstra agrega un nodo v a S implica que $d[v] = \delta(u,v)$ con u nodo fuente.

La noción base es comenzar cuando se extraer el primer elemento de la Priority Queue el cual es la fuente ya que todos los demás nodos están marcados con la distancia ïnfinito", por tanto se cumple $d[u] = 0 = \delta(u,u)$. Ya luego se relajan todos los arcos adyacentes al nodo fuente lo que garantiza para la próxima iteración que existirá algún nodo con distancia mínima calculada, pues este tendrá que salir de la Priority Queue, pero en particular se van formando caminos de costo minimo, entonces, al sacar un nodo de la Priority Queue debemos tener en cuenta que este forma un camino optimo con la fuente de lo contrario contradice la siguiente propiedad de los caminos mínimos:

Suboptimalidad de Caminos:

En un camino de costo mínimo cualquier sub-camino es también óptimo.

De esto obtenemos que en caso de existir una posterior actualización de un nodo que no se sacó de la Priority Queue entonces no formo el camino optimo con la fuente en análisis anteriores. Ya que de forma abstracta cuando se extrae un nodo v con menor peso calculado entonces lo estamos agregando a un camino optimo que forma con el nodo fuente $P : \langle u, x, y, ..., v \rangle$.

- Escriba el algoritmo de Floyd Warshal y explique la idea detrás del mismo, es decir, que intenta calcular el algoritmo en cada iteración.

```
Floyd-Warshall ( Grafo G = (V = {n_0, n_1, ..., n_{|V|-1}}, E) ) {
    matriz d(|V|,|V|);        d[x,y] ← ∞  ∀x,y ∈ V;
    d[x,x] ← 0  ∀x ∈ V;
    d[x,y] ← c(x,y)  ∀e ∈ E;
    for j = 0 to |V| − 1 {
        for x,y ∈ V {
            d[x,y] = min(d[x,n_j] + d[n_j,x],  d[x,y]);
        }
    }
    return d;
}
```

Este algoritmo trata de calcular en cada iteración i caminos tal que para todo par de nodos existe un camino de costo mínimo en el cual se agrega un nodo n_i para un grafo $G = (V = \{n_0, n_1, ..., n_{|V|-1}\}$. Básicamente en cada iteración i checamos que exista un camino entre un nodo x e y usando el nodo n_i como intermediario y en caso afirmativo verificar si la distancia formada es menor a la actual y actualizarla.

2. Práctica CodeForces (15 pts)

Agrego para cada problema del CodeForces la IDEA GENERAL (que esta en el código) para que esta documentada acá.

- A. Jzzhu and Cities

  Se pide determinar la cantidad de trenes a eliminar tales que sean ineficientes, es decir que no acorten "distancia"para llegar a una ciudad o como también trenes repetidos. Para abordar el problema se usa el algoritmo de Dijkstra que permite calcular los caminos de costo mínimo desde una fuente, en este caso la Çapital- nodo 1. El problema es análogo a determinar los costos para cada ciudad usando tanto las rutasçomo los "trenes.[en] el proceso de Dijkstra.

  Durante el proceso del algoritmo Dijkstra, este marca cuando se obtiene un camino con menor costo al que posee actualmente entonces decimos que este nodo posee 1 camino y es mínimo, cuando este camino sea realmente el optimo entonces cualquier aparición de otro camino igual decimos que tiene 2 caminos mínimos, es decir aumentamos contador de caminos "óptimos".

  Una vez finalizado el calculo de Dijkstra entonces decimos que para la cantidad de trenes dados por el problema, procesamos la ciudad destino y verificamos si esta ciudad posee un costo menor al del tren, si es el caso entonces el tren es innecesario, por otra parte si se da el caso de tener mismo costo entonces verificamos la cantidad de caminos que este posea y mientras sea mayor a uno eliminamos el tren. Básicamente, mientras el contador de caminos óptimos sea mayor que uno entonces tenemos rutas innecesarias de trenes para ese nodo/ciudad.

- B. Checkposts

  Se solicita buscar el costo mínimo a gastar en colocar los Policías de seguridad de las ciudades, como también la cantidad de formas en que se pueden colocar estos respectivos Policías. Dado que existe como propiedad de los Policías poder proteger otras ciudades siempre y cuando pueden viajar a ellas y poder devolverse. Esto es análogo a encontrar componentes fuertemente conexas dentro de las ciudades, se habla de C.F.C. porque los caminos entre ciudades son "vías dirigidas- Arcos dirigidos.

  Asi mismo, tener una C.F.C. permite saber por definición de la misma que todo par de nodo perteneciente a esta permite una conexión (existe un camino). Por tanto, una vez hallada una C.F.C. es determinar en ella cual es el nodo con menor costo y cuantas veces aparece este, ya luego es llevar un acumulador de esto. Recordamos por el principio fundamental del conteo (Discretas 1), se tiene un conjunto de C.F.C. $C_1, C_2, ..., C_k$ y cada una de ellas tiene su dinero mínimo y repeticiones, d_i y r_i con $1 \leq i \leq k$ respectivamente, decimos entonces que las formas totales se calculan como repeticiones de C_1 por Repeticiones de C_2, ... por repeticiones de C_k, es decir $r_1 \times r_2 \times ... \times r_k$. Asi mismo, para el dinero mínimo es la suma $d_1 + d_2 + ... + d_k$.

  Finalmente, la respuesta viene dado por $DineroMinimo = \sum_{i=1}^{k} d_i$ y $Opciones = \prod_{i=1}^{k} r_i$.

- C. Greg and Graph

  Se desea calcular la suma de todas las distancias mínimas para todo par de nodo "SOBRANTES"justo antes de eliminar un nodo del Grafo. Para esto es vital el algoritmo de Floyd-Warshall que determina esto, solo que este algoritmo va agregando nodos en sus iteración k determina un camino de tamaño k de ser posible para todo par de nodos, es decir los tamaños CRECIENTES. Este detalle para el problema planteado es importante ya que al "Paso k"Greg elimina un nodo lo que genera tamaños de caminos "DECRECIENTES".

  Por tanto, una vez iniciada la iteración principal de Floyd-Warshall (que seria camino de tamaño 1 debemos tener en cuenta leer el ultimo nodo a borrar que es cuando Greg solo le quedan caminos de tamaño 1 y al mismo tiempo marcar como visitado dicho nodo a "borrar". Una vez iniciado los fors anidados internos del algoritmo se determinan los caminos mininos correctos para la iteración k (paso k inverso)lo que permite realizar un chequeo, este chequeo revisa si los nodos actuales ya han sido visitados("borrados") esto dice que para el "nodo siguiente a borrar"(el paso k inverso) ya no hay "mas camino"por tanto actualizar con la distancia mínima al paso $N - K + 1$ que recordemos el la forma inversa de la ejecución de Greg.

  Finalmente, el vector de respuestas o bien sea llamado $dist_mins$ (vector de distancias mínimas) tendrá la respuestas para cada paso k que se realizó en el Floyd-Warshall.