FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

[75.29 / 95.06] Teoría de Algoritmos I
Primer cuatrimestre de 2021
Fecha de entrega 20 de Octubre de 2021, 19:00
Primera Entrega

Alumno	Padrón	Email
Fernando Martin Tejelo	98911	ftejelo@fi.uba.ar
Mariano Lazzarini	106352	mlazzarini@fi.uba.ar
Tobias Luciano Rivero Trujillo	106302	trivero@fi.uba.ar
Martin Pata Fraile de Manterola	106226	mpata@fi.uba.ar
Matias Gabriel Fusco	105683	mfusco@fi.uba.ar

Índice

1. Parte 1: Minimizando costos

Una empresa productora de tecnología está planeando construir una fábrica para un producto nuevo. Un aspecto clave en esa decisión corresponde a determinar dónde la ubicarán para minimizar los gastos de logística y distribución. Cuenta con N depósitos distribuidos en diferentes ciudades. En alguna de estas ciudades es donde deberá instalar la nueva fábrica. Para los transportes utilizarán las rutas semanales con las que ya cuentan. Cada ruta une de ida y vuelta dos depósitos. No todos los depósitos tienen rutas que los conecten. Por otro lado, los costos de utilizar una ruta tienen diferentes valores. Por ejemplo hay rutas que requieren contratar más personal o comprar nuevos vehículos. En otros casos son rutas subvencionadas y utilizarlas les da una ganancia a la empresa. Otros factores que influyen son gastos de combustibles y peajes. Para simplificar se ha desarrollado una tabla donde se indica para cada ruta existente el costo de utilizarla (valor negativo si da ganancia).

Los han contratado para resolver este problema.

Han averiguado que se puede resolver el problema utilizando Bellman-Ford para cada par de nodos o Floyd-Warshall en forma general. Un amigo les sugiere utilizar el algoritmo de Johnson.

Se pide:

1. Investigar el algoritmo de Johnson y explicar cómo funciona. ¿Es óptimo?
2. En una tabla comparar la complejidad temporal y espacial de las tres propuestas.Analizar en qué situaciones una solución es mejor que otras
3. Crear un ejemplo con 5 depósitos y mostrar paso a paso cómo lo resolvería el algoritmo de Johnson.
4. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología greedy? Justifique
5. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología de programación dinámica? Justifique
6. Programar la solución usando el algoritmo de Johnson.

1.1. Investigar el algoritmo de Johnson y explicar cómo funciona. ¿Es óptimo?

El algoritmo de Johnson encuentra el camino más corto entre todos los pares de vértices de un grafo dirigido disperso y permite aristas con pesos negativos, pero no ciclos negativos. Usa el algoritmo de Bellman-Ford para hacer una transformación en el grafo inicial para eliminar las aristas de peso negativo, permitiendo usar el algoritmo de Dijkstra en el grafo resultante, para obtener el camino mínimo.

Consiste en los siguientes pasos:

1. Se añade un vértice q al grafo,el cual tendrá un arista a todos los demás vértices del grafo con peso igual a cero.
2. Se usa Bellman-Ford, empezando por el vértice q, para determinar para cada vértice v el peso mínimo $h(v)$ del camino de q a v. Si se detecta un ciclo negativo, el algoritmo concluye.
3. Después a las aristas del grafo original se les cambia el peso usando los valores calculados por Bellman-Ford: una arista de u a v con tamaño $w(u, v)$, tendrá un nuevo tamaño $w'(u, v) = w(u, v) + h(u) - h(v)$
4. Por último, para cada vértice se usa el algoritmo de Dijkstra para determinar el camino más corto, usando el grafo resultante del paso anterior, para poder encontrar el óptimo.

En el grafo con pesos modificados, todos los caminos entre un par de nodos s y t tienen la misma cantidad $h(s)-h(t)$ añadida a cada uno de ellos, así que un camino que sea el más corto en el grafo original también es el camino más corto en el grafo modificado y viceversa. Sin embargo, debido al modo en el que los valores $h(v)$ son computados, todos los pesos modificados de las aristas son no negativo, por lo tanto tenemos un grafo que cumples con los requisitos para la utilización del algoritmo Dijkstra en el nuevo grafo. Como dicho grafo cumple con las condiciones para utilizar Dijkstra de manera óptima y el mismo es un algoritmo óptimo por consiguiente obtenemos que Johnson sea óptimo, permitiendo la resolución de camino mínimo en un grafo que tenga pesos negativos.

1.2. En una tabla comparar la complejidad temporal y espacial de las tres propuestas.Analizar en qué situaciones una solución es mejor que otras

Propuesta	Complejidad temporal	Complejidad espacial
Bellman-Ford	$O(V * E)$	$O(V)$
Floyd-Warshall	$O(V^3)$	$O(V^2)$
Johnson	$O(V^2 log(V) + V * E)$	$O(V * E + V^2)$

Sean V número de vértices y E número de aristas

Bellman-Ford, a diferencia de Floyd-Warshall y Johnson, solo busca el Árbol recubridor mínimo desde un nodo especifico, en otras palabras, busca el camino más corto a cada uno de los nodos, pero desde cierto nodo especificado.
Por tener que hacer solo esto, el algoritmo resulta tener una complejidad temporal y espacial bastante mejor que Floyd, y Johnson. Como contra parte esta solución es a un problema de dominio mas reducido por lo tanto por si solo podría no resolver la problemática que queramos tratar.

Para compararlos de manera más justa, podríamos decir que se podría usar V veces para obtener el camino mínimo entre cualesquiera 2 nodos, pero el problema de esto es que pierde todas las ventajas que tenía, haciendo que la complejidad temporal termine siendo $V^2 * E$; lo que lo volvería mucho más costoso que otros métodos

Floyd-Warshall busca el camino más corto entre cada uno de los nodos.Tiene la ventaja de que la complejidad no aumenta con la cantidad de aristas, por lo que podría llegar a ser mejor para grafos muy densos, aunque esta ventaja no es demasiado grande ya que Johnson llega a una complejidad V^3 también para un grafo totalmente denso.
En algunos casos en el cual podría llegar a ser una mejor idea, su utilización en comparación a Johnson, es en casos donde la implementación del grafo que se este utilizando, disponga de un gran coste, el agregado de aristas al grafo, ya que Johnson basa su funcionamiento en agregar aristas al grafo para operar, por lo tanto con una implementación de grafo que implique un 'agregar arista' costoso afectaría su ejecución, a diferencia de Floyd-Warshall el cual no tiene esta necesidad.

Johnson pareciera ser el que mejor algoritmo de todos, ya que busca el camino mínimo entre todos los pares de nodos, y con una complejidad algorítmica de $V^2 log(V)$ para grafos dispersos, y V^3 para un grafo totalmente denso, así que en el peor de los casos no llega a ser más lento que Floyd-Warshall. Pero sufre en cuanto a complejidad espacial. Ya que espacialmente ocupa lo equivalente a Bellman-Ford, y V veces Dijkstra. Y dependiendo de lo denso que sea el grafo, terminaría con una complejidad entre $V * E$, y V^2. Por lo tanto podríamos concluir que disponiendo de una cantidad de espacio en principio muy poco limitada, el algoritmo de Johnson es una solución realmente óptima.

1.3. Crear un ejemplo con 5 depósitos y mostrar paso a paso cómo lo resolvería el algoritmo de Johnson.

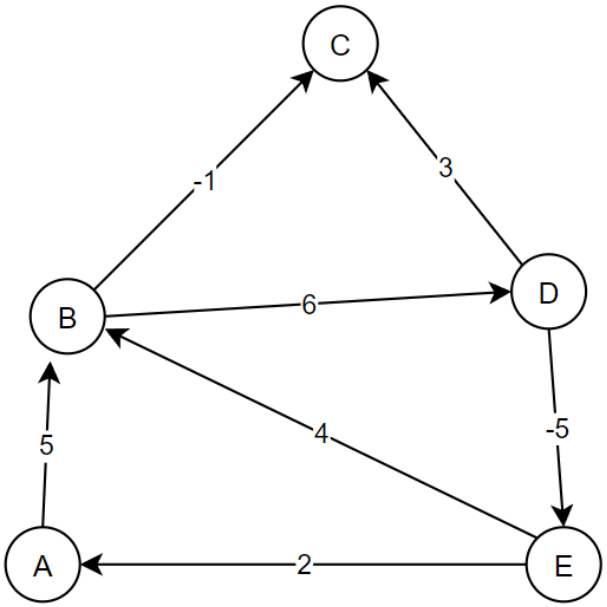
Partiendo de la problemática para la cual se nos ha contratado, creamos un ejemplo para simular como la resolveríamos utilizando el algoritmo de Johnson.

A continuación los pasos del algoritmo de Johnson:

1. Agregar nodo q con aristas de peso 0 a cada uno de los nodos
2. Ejecutar Bellman-ford, para encontrar el camino mínimo desde q, hasta el resto de los nodos.
3. Una vez encontrados todos los caminos mínimos, se modifica cada arista del grafo original, con el valor dado por la siguiente fórmula: $w(u, v) + h(u) - h(v)$. Siendo u el nodo de salida, v el de llegada, w el peso original, y h el camino mínimo desde q calculado por bellman; y se elimina el vértice q, y sus aristas.
4. Aplicar Dijkstra para cada uno de los nodos.
5. Recorrer lo devuelto por Dijkstra, y recalcular el peso original de cada nodo, para obtener la distancia real.

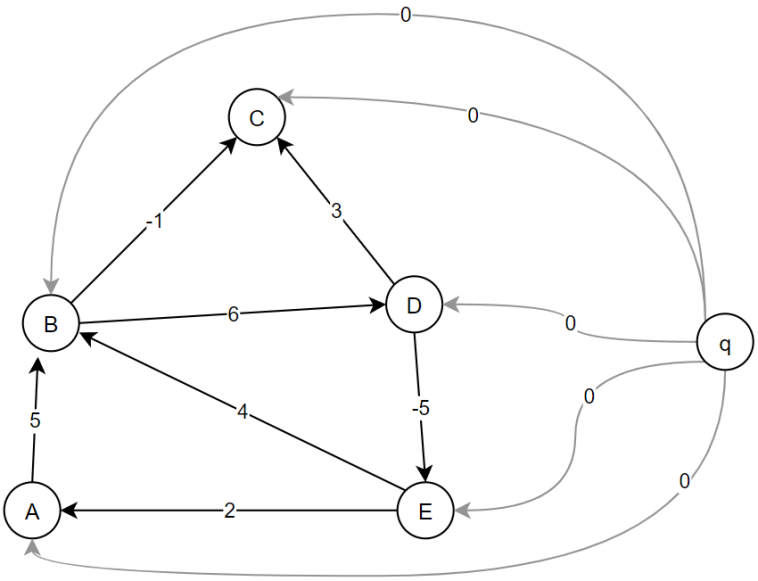
Grafo inicial:

Creamos un grafo direccionado de 5 vértices. Este contiene aristas de peso positivo y negativo, pero no incluye ciclos negativos que haría que el algoritmo no funcione.



1.3.1. Paso 1:

Agregamos un nodo **q** con aristas de peso 0 a cada uno de los nodos. Este nuevo nodo apunta a todos los nodos originales del grafo pero ninguno de estos apunta a **q**, por lo tanto para el grafo original el nuevo nodo "no existe".



1.3.2. Paso 2:

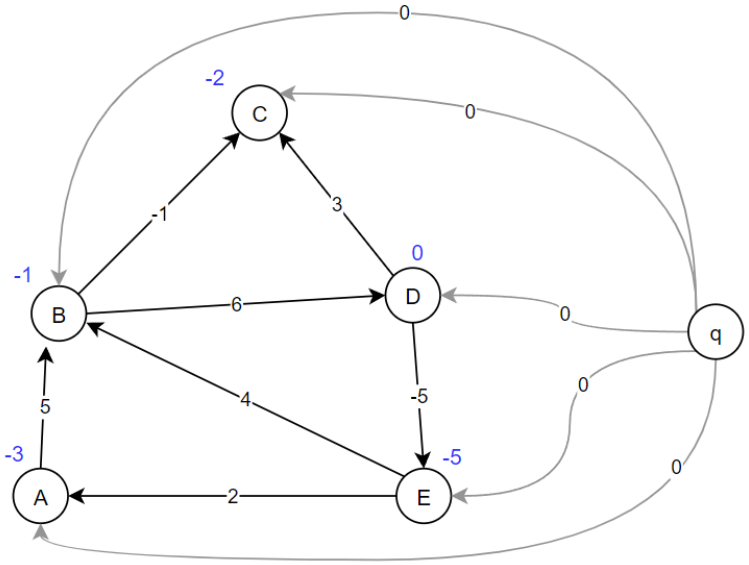
Ejecutar Bellman-ford, para encontrar el camino mínimo desde q, hasta el resto de los nodos.

Primero anotamos que para ir desde q hasta q en 0 pasos, se recorre 0 distancia, y para ir desde q hasta el resto de los nodos en 0 pasos, se recorre ∞ distancia (ya que no se pueden acceder en 0 pasos).
Pasamos en la segunda iteración, en la que buscamos el paso mínimo desde q hasta cada nodo atravesando una o menos arista, en este caso es 0 para todas. En la tercera iteración, buscamos el camino mínimo desde q, atravesando 2 o menos aristas. Para hacer esto, buscamos si es menor el camino de que teníamos en la iteración anterior, o ir primero a un antecesor del nodo al que queremos llegar, y luego al nodo al que queremos llegar. Para saber la distancia que se agregarían con este nuevo nodo extra, miramos la distancia que recorre hasta q en el mismo vector que anotamos las distancias y predecesores. Si la distancia con este nodo extra es menor, anotamos la nueva distancia, y el nodo predecesor al que llegamos.

```
Ejemplo con camino 2:
    q->E->A = 2  > 0 no se cambia
    q->A->B = 5  > 0 no se cambia
    q->E->B = 4  > 0 no se cambia
    q->D->C = 3  > 0 no se cambia
    q->B->C = -1 < 0 se cambia
    q->B->D = 6  > 0 no se cambia
    q->D->E = -5 < 0 se cambia
```

Para las siguientes iteraciones, seguimos haciendo lo mismo, fijándonos si es más corto el camino que ya tenemos, o pasando por algún otro predecesor, y anotamos el nuevo predecesor y distancia.

1.3.2.1 Resultado de Paso 2



Distancias

dist	q	A	B	C	D	E
0	0	∞	∞	∞	∞	∞
1	0	0	0	0	0	0
2	0	0	0	-1	0	-5
3	0	-3	-1	-1	0	-5
4	0	-3	-1	-2	0	-5
5	0	-3	-1	-2	0	-5
6	0	-3	-1	-2	0	-5

Predecesores

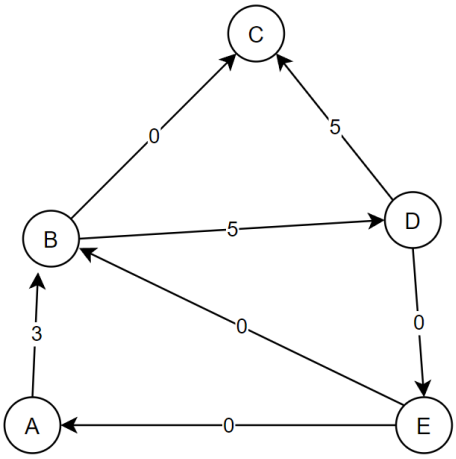
dist	q	A	B	C	D	E
0	q	N	N	N	N	N
1	q	q	q	q	q	q
2	q	q	q	B	q	D
3	q	E	E	B	Q	D
4	q	E	E	B	Q	D
5	q	E	E	B	Q	D
6	q	E	E	B	Q	D

Es imposible que exista con más de 5 aristas recorridas, pero igualmente se calcula, ya que si este camino de 6 aristas es distinto al de 5, quiere decir que existe un ciclo negativo en el grafo. En este grafo, no existe ningún cambio, por lo que podemos asegurar de que no existen ciclos negativos.

1.3.3. Paso 3

Una vez encontrados todos los caminos mínimos, se modifica cada arista del grafo original, con el valor dado por la siguiente fórmula: w(u,v) + h(u) - h(v). Siendo u el nodo de salida, v el de llegada, w el peso original, y h el camino mínimo desde q calculado por Bellman; y se elimina el vértice q, y sus aristas.

Por ejemplo, para la arista A->B: $w(A,B) + h(A) - h(B) = 5 + (-3) - (-1) = 3$



1.3.4. Paso 4

En este paso se aplica Dijkstra a todos los nodos.
Los pasos para implementar el algoritmo de Dijkstra son los siguientes:

1. Elegir un vértice para comenzar. La distancia del mismo será 0 y la del resto de los vértices la inicializamos en ∞.
2. Calcular la distancia hacia los vértices adyacentes sumando los pesos de cada arista que los conecta con la distancia del vértice actual. Si existen distancias previamente calculadas pero conozco una nueva que me lleve a un vértice ya conocido que sea menor, lo actualizo.
3. Marco el vértice actual como visitado y me muevo hacia el vértice con menor distancia conocida.
4. Repito hasta llegar al vértice objetivo.

1.3.4.1 A continuación un ejemplo con el nodo A como inicio:

Armamos una tabla con todos los nodos del Grafo, al ser A el nodo inicial seteamos su distancia a 0 y como nodo previo Nil, porque no es el inicial. Al resto de nodos los setamos con una distancia a infinito y seguimos lo pasos explicados arriba

Node	d[v]	nodoA
A	0	Nil
B	∞	Nil
C	∞	Nil
D	∞	Nil
E	∞	Nil

Nos fijamos los nodos adyacentes a **A** que es **B**:

- Chequeamos la nueva distancia de B: $distancia\_actual[A] + distancia[B] = 0 + 3 = 3 <$ ∞(Distancia Actual). Entonces actualizamos la distancia de B a 3 y ponemos A como nodo previo.

Node	d[v]	nodoA
A	0	Nil
B	5	A
C	∞	Nil
D	∞	Nil
E	∞	Nil

Ahora vamos al nodo **B** por ser el próximo nodo no visitado de menor distancia. De B tenemos **2** nodos adyacentes **C** y **D**:

- Para **C** tenemos que su nueva distancia es: $distancia\_actual[B] + distancia[C] = 3 + 0 = 3 < \infty$(Distancia Actual). Entonces actualizamos la distancia de C a 3 y ponemos B como nodo previo.
- Para **D** tenemos que su nueva distancia es: $distancia\_actual[B] + distancia[C] = 3 + 5 = 8 < \infty$(Distancia Actual). Entonces actualizamos la distancia de C a 8 y ponemos B como nodo previo.

Node	d[v]	nodoA
A	0	Nil
B	3	A
C	3	B
D	8	B
E	∞	Nil

Ahora vamos al nodo C por ser el próximo nodo no visitado de menor distancia y este no tiene nodos adyacentes

Node	d[v]	nodoA
A	0	Nil
B	3	A
C	3	B
D	8	B
E	∞	Nil

Ahora vamos al nodo D por ser el próximo nodo no visitado de menor distancia y este tiene como adyacente **C** y **E**

- Para **C**: $distancia\_actual[D] + distancia[C] = 8 + 5 = 13 > 3$. No hacemos nada ya que la distancia actual es menor que la nueva calculada.
- Para **E** tenemos que $distancia\_actual[D] + distancia[E] = 8 + 0 = 8 < \infty$(Distacia Actual). Entonces actualizamos la distancia de E a 8 y ponemos D como nodo previo.

Node	d[v]	nodoA
A	0	Nil
B	3	A
C	3	B
D	8	B
E	8	D

Para E que es el último nodo no visitado tenemos que sus nodos adyacentes son **A** y **B**:

- Para **A**: $distancia\_actual[E] + distancia[A] = 8 + 0 = 8 > 0$. No hacemos nada ya que la distancia actual es menor que la nueva calculada.
- Para **B**: $distancia\_actual[E] + distancia[B] = 8 + 0 = 8 > 3$. No hacemos nada ya que la distancia actual es menor que la nueva calculada.

Node	d[v]	nodoA
A	0	Nil
B	3	A
C	3	B
D	8	B
E	8	D

1.3.4.2 Resultado del Paso 4

Si estos pasos se aplican a todos lo nodos las distancias obtenidas por Dijkstra con el grafo modificado son:

o.	A	B	C	D	E
A	0	3	3	8	8
B	5	0	0	5	5
C	∞	∞	0	∞	∞
D	0	0	0	0	0
E	0	0	0	5	0

1.3.5. Paso 5

Una vez que se obtienen las distancias por Dijkstra, se pueden volver a obtener las distancias originales, pudiendo volver al valor original de cada nodo utilizando la fórmula $w'(u, v) - h(u) + h(v)$

Distancias obtenidas por Dijkstra con los pesos originales.

o.	A	B	C	D	E
A	0	5	4	11	6
B	3	0	-1	6	1
C	∞	∞	0	∞	∞
D	-3	-1	-2	0	-5
E	2	4	3	10	0

1.4. Pseudocódigo

A continuación se adjunta el Pseudocódigo de Johnson y sus algoritmos auxiliares:

1.4.1. Johnson

```
Johnson(grafo):
    creamos un vertice q con arista de peso 0 a todos
    los vertices del grafo.

    aplicamos BellmanFord(grafo):
        donde para cada vertice definimos el peso minimo desde q-v
        para cada union (u,v) con peso w en grafo hacemos:
                peso de esa union = peso union + pesoU - pesoV

        //en este punto ya todos los pesos son positivo

    crear nueva matriz D de distancias inicializada a infinito

    para cada nodo v del grafo modificado:
        aplicar Dijkstra desde v
        camino_minimo = Dijkstra(grafo)

    retornar camino_minimo
```

1.4.2. Dijkstra

```
Dijkstra(grafo, nodo_origen):
    Se crean 2 vectores, uno de distancia y otro de nodo padre
    se crea un set Q de nodos.

    por cada nodo V en el grafo:
        dist[v] = infinito
        prev[v] = indefinido
        agregar v a Q

    dist[nodo_origen] = 0

    mientras Q no esta vacio:
        u = nodo en Q con el minimo de dist[u]
        sacar u de Q

    para cada vecino de u (v) que todavia esta en Q:
        si dist[u] + peso_arista(u,v) < dist[v]:
            dist[v] = dist[u] + peso_arista(u,v)
            prev[v] = u

    retornar dist[], prev[]
```

1.4.3. Bellman-Ford

```
Bellman_Ford(grafo, nodo_origen):
    creamos 2 vectores de tamanio v, uno de distancias y otro de
    predecesores
    // No se necesita el vector de predecesor si solo
    // se necesita distancia y no recorrido.

    Para cada nodo V del grafo:
        distancia[v]=INFINITO
        predecesor[v]=NULL

    distancia[nodo_origen]=0

    // relajamos cada arista del grafo tantas veces
    // como numero de nodos-1 haya en el grafo

    Desde i=1 hasta (cantidadNodos(Grafo)-1):
        Para cada par de nodos (u,v) conectados por una arista:
            si distancia[v] > distancia[u] + peso_arista(u, v) entonces:
                distancia[v] = distancia[u] + peso_arista(u, v)
                predecesor[v] = u

    // comprobamos si hay ciclos negativo
    Para cada par de nodos (u,v) conectados por una arista:
        si distancia[v] > distancia[u] + peso_arista(u, v) entonces:
            //Quiere decir que hubo cambios por lo tanto
            retornar Error:existe un ciclo negativo

    //Si llegamos aca entonces no existe ningun ciclo negativo,
    // y se pudieron obtener las distancias minimas

    retornar distancia, predecesor
```

1.5. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología greedy? Justifique.

Si, utiliza una metodología greedy, ya que en su implementación hace uso de un algoritmo de tipo greedy como lo es Dijkstra. Pero en su totalidad no es un algoritmo de tipo greedy, ya que tiene la particularidad de utilizar otros algoritmos en su implementación como es(Bellman-Ford) **el cual no es de tipo greedy**, por lo tanto no se podría afirmar que Johnson sea un algoritmo de tipo greedy al cien por ciento.

1.6. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología de programación dinámica? Justifique

De igual manera que lo justificado anteriormente. Si, Johnson hace uso de programación dinámica al tener en su implementación un algoritmo auxiliar que hace uso de dicha metodología, pero como en el caso anterior. No se podría afirmar que es un algoritmo,que enteramente tenga metodología de programación dinámica por la presencia de Dijkstra.

1.7. Programar la solución usando el algoritmo de Johnson.

Junto al presente informe, se adjunta los archivos necesarios para la ejecución del programa. A continuación se agregan las instrucciones para su uso.

Requisitos:

- Python 3
- Un archivo .txt o .csv con formato: **nodoOrigen,nodoDestino,pesoArista**

Instrucciones de ejecución: Teniendo en una misma carpeta los 3 archivos .py, se debe llamar mediante la consola a "python main.py <ruta>". Se puede especificar una ruta completa o solo el nombre del archivo si este se encuentra en la misma carpeta.

1.7.1. Modificaciones al algoritmo original de Johnson

Por cuestiones de claridad de la implementación, decidimos modificar ligeramente el algoritmo de Johnson. En el algoritmo original, luego de hacer V veces Dijkstra, uno recorre los nodos devueltos por Dijkstra, y va recalculando las distancias originales de las aristas (con la ecuación de $w'(u,v) - h(u) + h(v)$), a la vez que va recalculando las distancias totales entre cada uno de los nodos.

En nuestra versión del algoritmo, decidimos que en vez de volver a recorrer lo devuelto por Dijkstra para recalcular las distancias, modificaríamos Dijkstra, para que devuelva las distancias ya recalculadas. Para hacer esto, Dijkstra además de recibir el grafo que recorre y el nodo de inicio, recibe las distancias al nodo q que calculamos previamente con Bellman-Ford. Además, Dijkstra tendría 2 vectores de distancia, uno para el grafo modificado, y otro para el grafo original.Entonces a la vez que recorre el grafo modificado, y almacena las distancias a cada nodo en el grafo modificado, y el predecesor a ese nodo; también almacena en un tercer vector las distancias que se hubieran recorrido en el grafo original; usando las distancias del grafo, y las distancias a Q que calculo Bellman-Ford a partir de la ecuación $w'(u,v) - h(u) + h(v)$. Este nuevo vector de distancias originales no interfiere de ninguna manera en la ejecución de Dijkstra, ya que solo cumple la meta de almacenar información. Cuando Dijkstra termina de ejecutarse, devuelve el vector de predecesores, y el vector de distancias para el grafo original, y de esta manera nos evitamos el tense que recalcular las distancias para los pesos originales, sino que Dijkstra ya se ocupo de hacerlo mientras se ejecutaba.

2. Parte 2: Un poco de teoría

1. Hasta el momento hemos visto 3 formas distintas de resolver problemas. Greedy, división y conquista y programación dinámica.

   a) Describa brevemente en qué consiste cada una de ellas

   b) Identifique similitudes, diferencias, ventajas y desventajas entre las mismas. ¿Podría elegir una técnica sobre las otras?

2. Tenemos un problema que puede ser resuelto por un algoritmo Greedy (G) y por un algoritmo de Programación Dinámica (PD). G consiste en realizar múltiples iteraciones sobre un mismo arreglo, mientras que PD utiliza la información del arreglo en diferentes subproblemas a la vez que requiere almacenar dicha información calculada en cada uno de ellos, reduciendo así su complejidad; de tal forma logra que $O(PD) < O(G)$. Sabemos que tenemos limitaciones en nuestros recursos computacionales (CPU y principalmente memoria). ¿Qué algoritmo elegiría para resolver el problema?

2.1. Hasta el momento hemos visto 3 formas distintas de resolver problemas. Greedy, división y conquista y programación dinámica.

2.1.1. Describa brevemente en qué consiste cada una de ellas

División y conquista: La forma de resolver una problemática por división y conquista consiste en 'achicar' un problema de cierto tamaño dividiéndolo en subproblemas de un tamaño menor, para luego resolver dichos subproblemas y que la unión de cada solución sea la solución del problema inicial. En general esta forma de trabajar va de la mano con el método de recurrencia y tiene 3 partes base:

- Caso base: resolución del caso trivial en nuestro problema.
- División del problema: donde dividimos nuestro problemas y creamos la recursión.
- Unión de los subproblemas.

En algunos casos permite la resolución de una problemática en complejidad total mayor a una resolución mas bruta. Justificada por el Teorema maestro.

Greedy: Utilizado principalmente para problemas de optimización, su enfoque en relación a su nombre, voraz, avaricioso etc, se debe a la forma que tiene la metodología de tratar problemas. Además presentan la característica de ser fáciles de pensar e implementar. Su forma de trabajar es partir de un dominio general X, y del mismo ir a un subproblema 'y' buscar la mejor forma de resolver 'y', sin tener en cuenta los demás problemas, realizados ni por realizar. Además una vez tomada una acción, esta no es revisada ni replanteada, se pasa al siguiente caso. De esta forma avanzan por los subproblemas hasta tener el problema general resuelto. Esta manera de avanzar conlleva ciertos problemas, y constituye una de las desventajas de la técnica Greedy. Su metodología hace que, para algunos problemas, sea imposible alcanzar una solución mediante un algoritmo Greedy. Es por eso que se debe chequear que el problema cumpla con ciertos requisitos antes de emplear esta técnica. Estos requisitos son la Elección Greedy' y la 'Subestructura Óptima'. Otra cuestión a tener en cuenta con este tipo de algoritmos es que para cada problema que cumpla con estas características existen diferentes algoritmos greedy, pero algunos no son óptimos, y en algunos casos puede que ninguno.

En general las resoluciones del tipo greedy presentan:

- conjunto de candidatos: elementos seleccionables.
- Solución parcial: candidatos seleccionados.
- Función de selección: determina el mejor candidato del conjunto de candidatos seleccionables.
- Función de factibilidad: determina si es posible completar la solución parcial para alcanzar la solución del problema.
- Criterio que define que es una solución: indica si la solución parcial obtenida resuelve el problema.
- Función Objetivo: valor de la solución alcanzada.

Con estos elementos su forma general de proceder es:

1. Se parte de un conjunto vacío.
2. De la lista de candidatos, se elige el mejor de acuerdo a **función de selección**.
3. Comprobamos si se puede llegar a una solución con el candidato elegido, por medio de su **función de factibilidad**.
4. si no llegamos a una solución elegimos otro candidato y repetimos hasta solucionar el problema o quedarnos sin candidatos.

Programación dinámica: Al igual que la programación Greedy, la metodología de programación dinámica es utilizada en general para problemas de optimización. Permite resolver problemas mediante secuencia de decisiones y a la vez lo realiza descomponiendo el problema en varios subproblemas de tamaño menor. La forma de trabajar de la programación dinámica, es dividir en problema en subproblemas hasta llegar a los problemas de atómicos o casos base, luego desde este punto se resolverán cada uno de estos subproblemas, escalando desde lo mas chico hasta lo mas grande, con la peculiaridad de que estas soluciones, serán recordadas o almacenadas en algún lugar para evitar recalcular problemas iguales. Es aquí donde la programación dinámica muestra su potencial ya que al hacer esta técnica de memorizar, se evitan muchas operaciones, las cuales afectan la complejidad total, lo que nos puede acercar a una solución óptima. La gran contra parte de esta técnica es un aumento en el uso de memoria, el cual en algunos casos podría generarnos problemas.

2.1.2. Identifique similitudes, diferencias, ventajas y desventajas entre las mismas. ¿Podría elegir una técnica sobre las otras?

Si bien con lo descrito anteriormente, mencionamos algunas de las ventajas o desventajas de las mismas, a continuación nos centraremos en ellas.

En general por lo descripto en el apartado de programación dinámica, esta podría sonar similar a tanto programación greedy como división y conquista, lo cual es verdad. Programación dinámica Utiliza ambos conceptos de trabajo con algunas diferencias, en el caso de Greedy la diferencia radica en que se realizan múltiples decisiones hasta el final del recorrido, donde se decidirá cual es mejor de ellas. En el caso de división y conquista, la diferencia es mas sencilla, radica en la memorización de soluciones y reutilización de las mismas para evitar calcular nuevamente problemas ya resueltos.

La división y conquista, puede permitir resolver problemas complejos con un solución bastante sencilla. Por otro lado la tarea de idear y analizar estos algoritmos en casos generales puede ser bastante difícil y requiere muchas practica dominarlo ya que aveces es necesario cambiar el problema a resolver por uno mas difícil para poder encontrar solución.

La programación Greedy tiene como ventaja, la característica de ser soluciones fáciles de pensar e implementar, por lo cual pueden ser bastante tentadoras, pero traen como contrapartida alta dificultad en demostrar que verdaderamente sean óptimos, ya que incluso al llegar a una solución esta podría no ser óptima, por lo tanto siempre hay que tener en cuenta esto a la hora de resoluciones greedy.

Finalmente, la programación dinámica, tiene como principal problema el uso extra de memoria, que en casos donde la misma este limitada o el problema tenga un tamaño considerablemente enorme podría traernos problemas. Además hay que tener en cuenta que el espectro de uso del mismo no es tan general como en los otros métodos.

Elegir un método sobre el otro, dependerá de cada dominio que se quiera resolver, ya que podemos tener bloqueos o recursos limitados que podría hacer a una opción inviable o poco recomendable por ejemplo, si contáramos con una cantidad muy limitada de memoria, recurrir a programación dinámica quedaría descartado.En casos donde la problemática no sea de optimización podríamos perder mas tiempo en tratar de demostrar que nuestra solución greedy verdaderamente nos entrega un solución óptima. De igual forma podemos encontrarnos con la dificultad de encontrar un caso base o un modo de división del problema para poder achicar el mismo.

2.2. ¿Qué algoritmo elegiría para resolver el problema?

El algoritmo a elegir, dependerá principalmente de que tiene una limitación mayor, si la memoria, o el CPU. Suponiendo que la limitación de la memoria (ya sea de espacio o velocidad) sea mucho mayor que la del CPU, convendría utilizar el algoritmo Greedy, ya que no depende del acceso a memoria para resolver ninguno de los subproblemas. En cambio, si la limitación de memoria no es tan grande, o menor que la del CPU, terminaría conviniendo Programación Dinámica, que suele tener una complejidad temporal menor, a cambio de tener una complejidad espacial mayor, gracias al almacenamiento y uso de subproblemas repetidos.

2.2.1. Conclusión

Como conclusión final sobre el informe podemos destacar 2 aspectos. Por un lado respecto a la primera parte del trabajo en relación al camino mínimo, el algoritmo de Johnson es una herramienta muy versátil y útil, permitiendo el peso negativo en los grafos y demostrando que algunas veces la aplicación de uno o mas algoritmos de forma auxiliar y con ciertas características permite encontrar una nueva forma óptima de resolver problemáticas. Con respecto a la parte 2, podemos encontrar que hay un amplio abanico de herramientas y metodologías para resolver problemas, ninguna de ellas, sin embargo, es el arma definitiva, con esto queremos decir que ninguna herramienta suplanta a ninguna otra. Es mas, cada una de ellas aumenta las problemáticas que podemos encarar y a su vez, vuelve mas eficiente nuestro trabajo como desarrolladores.

3. Refencias

- DECASAI, departamento de ciencias de la computacion e I.A, Universidad de Granada, "Algoritmos Greedy, analisis y diseño de algoritmos"
- Universidad de Zaragoza, Porfesor de algoritmos, Javier Campos, "Algoritmica basica"
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms - Third Edition", 2009
- Alex Chumbley, Karleigh Moore, Jimin Khim - brilliant.org/wiki/johnsons-algorithm