FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico N°2

Nombre: Melina Lazzaro **Padrón:** 105931
Nombre: Nahuel Castro **Padrón:** 106551
Nombre: Ana Gutson **Padrón:** 105853
Nombre: Nicolás Pinto **Padrón:** 105064
Nombre: Iván Litteri **Padrón:** 106223

Fecha de Entrega: 20/10/2021
Grupo: liomessi30

Lineamientos básicos

- El trabajo se realizará en grupos de cinco personas.
- Se debe entregar el informe en formato pdf y código fuente en (.zip) en el aula virtual de la materia.
- El lenguaje de implementación es libre. Recomendamos utilizar C, C++ o Python. Sin embargo si se desea utilizar algún otro, se debe pactar con los docentes.
- Incluir en el informe los requisitos y procedimientos para su compilación y ejecución. La ausencia de esta información no permite probar el trabajo y deberá ser re-entregado con esta información.
- El informe debe presentar carátula con el nombre del grupo, datos de los integrantes y y fecha de entrega. Debe incluir número de hoja en cada página.
- En caso de re-entrega, entregar un apartado con las correcciones mencionadas

Índice general

Capítulo 1

Minimizando costos

1.1. Objetivos

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 dos depósitos en un sentido. 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.

Aclaración: No existen ciclos negativos!

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.
3. Analizar en qué situaciones una solución es mejor que otras
4. Crear un ejemplo con 5 depósitos y mostrar paso a paso cómo lo resolvería el algoritmo de Johnson.
5. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología greedy? Justifique
6. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología de programación dinámica? Justifique.
7. Programar la solución usando el algoritmo de Johnson.

Formato de los archivos:

El programa debe recibir por parámetro el path del archivo donde se encuentran los costos entre cada depósito. El archivo debe ser de tipo texto y presentar por renglón, separados por coma un par de depósitos con su distancia.

Ejemplo: "depósitos.txt"

Debe resolver el problema y retornar por pantalla la solución. Debe mostrar por consola en en que ciudad colocar el depósito. Además imprimir en forma de matriz los costos mínimos entre cada uno de los depósitos.

1.1.1. Algoritmo de Johnson

El algoritmo de Johnson es un algoritmo cuya finalidad es hallar el camino más corto entre todos los pares de vértices de un grafo dirigido disperso. Permite que las aristas tengan pesos negativos, pero no que los ciclos tengan peso negativo.

Se basa en dos algoritmos:

- El algoritmo de Bellman - Ford, el cual realiza una transformación en el grafo inicial, eliminando todas las aristas de peso negativo, y detectando todos los ciclos negativos;
- El algoritmo de Dijkstra, usándolo en el grafo resultante de utilizar el algoritmo anterior.

¿Cómo funciona?

Para aplicar el algoritmo, se siguen los siguientes pasos:

Pasos para aplicar el algoritmo de Johnson

1. Se añade un nodo q al grafo, el cual está conectado a cada uno de los nodos del grafo por una arista de peso cero.
2. Se utiliza el algoritmo de *Bellman - Ford*, empezando por el nuevo vértice q, para determinar para cada vértice v el peso mínimo $h(v)$ del camino de q a v.
   $\rightarrow$ Si en este paso se detecta un ciclo negativo, el algoritmo concluye.
3. Se actualiza el peso de las aristas del grafo, usando lo calculado anteriormente: una arista de u a v con peso $w(u, v)$, tiene como nuevo peso
   $$w(u, v) + h(u)–h(v)$$
4. Para cada nodo s se usa el algoritmo de *Dijkstra* para determinar el camino más corto entre s y los otros nodos, usando el grafo actualizado anteriormente.

Como todas las aristas del grafo resultante tienen un peso actualizado con el mismo agregado de $h(u)–h(v)$, se asegura de que el camino más corto en el grafo original lo sea también en el grafo actualizado.

También, debido al modo en el que se computan los valores $h(v)$, se asegura de que todos los pesos de las aristas son no negativos.

```
def johnson(grafo: Graph) -> tuple(dict, dict):
    grafo_BF = Graph.desde_grafo(grafo)
    grafo_BF.agregar_vertice("0")
    for v in grafo_BF:
        grafo_BF.agregar_arista("0", v, 0)

```

```
    distancias_BF, _ = bellman_ford(grafo_BF, "0")

    for u in grafo:
        h_u = distancias_BF[u]
        for v in grafo.adyacentes_a(u):
            h_v = distancias_BF[v]
            w = grafo.peso_de_arista_entre(u, v)
            grafo.borrar_arista(u, v)
            grafo.agregar_arista(u, v, w+h_u-h_v)

    padres_D = {}
    distancias_D = {}
    for v in grafo:
        distancias_D[v], padres_D[v] = dijkstra(grafo, v)

    for v in distancias_D:
        for w in distancias_D[v]:
            distancias_D[v][w] = distancias_D[v][w] - distancias_BF[v] + distancias_BF[w]

    return distancias_D, padres_D
```

¿Es óptimo?

Cuando se analizan algoritmos, lo que interesa de ellos es su optimalidad y su eficiencia.
La optimalidad refiere sobre si, al utilizar un algoritmo en específico para resolver un problema, siempre se obtiene el resultado correcto; por otro lado la eficiencia (tanto en tiempo y espacio) tiene que ver con la cantidad de operaciones que se realizan para resolver el problema en cuestión.

Para el problema que interesa resolver, el algoritmo recomendado, Johnson, es óptimo ya que sabemos con precisión que el grafo de nuestro problema no posee ciclos negativos. En caso de poseerlos, deberíamos reclinar de esta implementación, ya que este algoritmo no admite ciclos negativos: en caso de hallarlos, el algoritmo concluye.

Además, al lograr que los pesos de las aristas no sean negativos, se asegura la optimalidad de los caminos encontrados por Dijkstra, y por ende, de Johnson.

1.1.2. Comparación de complejidad temporal y espacial entre los algoritmos propuestos

Los algoritmos propuestos para resolver el problema en cuestión son:

- El algoritmo de *Bellman - Ford*.
- El algoritmo de *Floyd - Warshall*.
- El algoritmo de *Johnson*.

Se procede entonces a medir el uso de recursos de estos algoritmos.

Al medir el uso de recursos de un algoritmo, se puede hacer, entre otras formas, mediante su complejidad temporal, es decir, cuánto demora el algoritmo en terminar su ejecución dada una entrada v, y su complejidad espacial, midiendo el uso de memoria operativa requerida por el algoritmo (tanto el código como los datos con los que éste opera).

Se analizan las complejidades temporales y espaciales de los algoritmos propuestos.

Bellman - Ford

El algoritmo de Bellman - Ford es un algoritmo para encontrar el camino más corto en un grafo dirigido ponderado. Normalmente es utilizado cuando hay aristas con peso negativo.

Si el grafo contiene un ciclo de coste negativo, el algoritmo lo detecta, pero no encuentra el camino más corto que no repite ningún vértice.

El algoritmo consiste en hacer varias veces lo siguiente: recorrer todas las aristas. Si la arista (u, v) de longitud L es tal que la distancia que teníamos guardada desde el origen s hasta el nodo u más la longitud de la arista L, es menor a la distancia que teníamos guardada hasta el nodo v, se actualiza la información de la distancia hasta el nodo v como la suma mencionada.

Si el grafo tiene V nodos, entonces, alcanza con recorrer sus aristas $V - 1$ veces. Por ende, la complejidad temporal del algoritmo es

$$\mathcal{O}(V.E)$$

con V: vértices E: aristas.

Al implementar el algoritmo, se utiliza una matriz, cuyo tamaño depende de la cantidad de vértices que tenga el grafo, por lo que entonces la complejidad espacial del mismo será de

$$\mathcal{O}(V)$$

con V: vértices.

```python
def obtener_aristas(grafo) -> list:
    aristas = []
    for v in grafo:
        for w in grafo.adyacentes_a(v):
            aristas.append((v, w, grafo.peso_de_arista_entre(v, w)))
    return aristas

def bellman_ford(grafo, origen) -> tuple(dict(str, dict), dict(str, dict)):
    distancias = {}
    padres = {}
    for v in grafo:
        distancias[v] = float('inf')

    distancias[origen] = 0
    padres[origen] = None
    aristas = obtener_aristas(grafo)
    for _ in range(len(grafo) - 1):
        for v, w, peso in aristas:
            if distancias[v] + peso < distancias[w]:
                distancias[w] = distancias[v] + peso
                padres[w] = v

    for v, w, peso in aristas:
        if distancias[v] + peso < distancias[w]:
            raise ValueError('El grafo tiene ciclos negativos')

    return distancias, padres
```

Dijkstra

```python
def dijkstra(grafo, origen) -> tuple(dict, dict):
    distancias = {v: float('inf') for v in grafo}
    padres = {}

    distancias[origen] = 0
    padres[origen] = None
    cola = PriorityQueue()

    for v in grafo:
        cola.put(v)

    while not cola.empty():
        v = cola.get()
        for w in grafo.adyacentes_a(v):
            if distancias[w] > distancias[v] + grafo.peso_de_arista_entre(v, w):
                distancias[w] = distancias[v] + grafo.peso_de_arista_entre(v, w)
                padres[w] = v
                cola.put(w)

    return distancias, padres
```

Floyd - Warshall

El algoritmo de Floyd - Warshall es un algoritmo para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. Este algoritmo es un claro ejemplo de programación dinámica, ya que desglosa el problema en subproblemas más pequeños, luego combina las respuestas a esos subproblemas para resolver el gran problema inicial.

Este algoritmo compara todas las rutas posibles a través del gráfico entre cada par de vértices. Es capaz de hacer esto con solo V^3 comparaciones. Esto es notable considerando que puede haber hasta V^2 aristas en el gráfico y se prueba cada combinación de aristas. Lo logra mejorando gradualmente una estimación en la ruta más corta entre dos vértices, hasta que se sabe que la estimación es óptima.

Una forma de representar el algoritmo es mediante una matriz cuadrada de VxV, dando lugar a una complejidad espacial de

$$\mathcal{O}(V^2)$$

con V: vértices.

Es debido a todo lo anteriormente mencionado que su complejidad temporal es de

$$\mathcal{O}(V^3)$$

con V: vértices,

dependiendo completamente del número de vértices del gráfico.

Como se verá más adelante, esto lo hace especialmente útil para un cierto tipo de gráfico y no tanto para otros tipos.

Johnson

Como se dijo previamente, Johnson está comprendido por dos algoritmos, cuyas complejidades ya conocemos. Se procede a analizar entonces cada uno de los pasos.

- El primer paso del algoritmo es simple: consiste en añadir un vértice, con una arista por cada vértice existente. Por ende, la complejidad es de $\mathcal{O}(V)$
- El segundo y tercer paso se ejecutan con la complejidad del *algoritmo Bellman - Ford*, es decir, $\mathcal{O}(VE)$.
- El paso final se ejecuta con la complejidad del *algoritmo de Dijkstra*, el cual es implementado por cada vértice del grafo. Si se decide implementar el *algoritmo de Dijkstra por montículos de Fibonacci*, entonces por cada iteración, se requiere de $\mathcal{O}(V \log V + E)$ tiempo para completar para una complejidad total de

$$\mathcal{O}(V^2 \log V + VE)$$

con V: vértices E: aristas.

A la hora de ser implementado, Johnson utiliza:

- Un grafo, cuya complejidad es de $\mathcal{O}(VE)$.
- Dos diccionarios de complejidad $\mathcal{O}(V)$.
- Dos diccionarios de complejidad $\mathcal{O}(V^2)$.

Por ende, la complejidad espacial del algoritmo es de

$$\mathcal{O}(V^2)$$

con V: vértices.

Complejidad	Bellman - Ford	Johnson	Floyd - Warshall
Temporal	$\mathcal{O}(VE)$	$\mathcal{O}(V^2 \log V + VE)$	$\mathcal{O}(V^3)$
Espacial	$\mathcal{O}(V)$	$\mathcal{O}(V^2)$	$\mathcal{O}(V^2)$

1.1.3. Comparación entre situaciones

Con base en lo analizado anteriormente, se llega a la siguiente conclusión.

El algoritmo *algoritmo Bellman - Ford* es un algoritmo que calcula las rutas más cortas desde un único vértice de origen a todos los demás vértices en un dígrafo ponderado, detectando los ciclos negativos.

El algoritmo *algoritmo Floyd - Warshall* calcula las rutas más cortas de cada nodo a todos los demás. Se utiliza cuando cualquiera de todos los nodos puede ser una fuente, por lo que desea que la distancia más corta para llegar a cualquier nodo de destino desde cualquier nodo de origen. Esto solo falla cuando hay ciclos negativos. Su complejidad temporal depende solamente de los vértices del grafo.

El *algoritmo de Johnson* calcula las rutas más cortas de cada nodo a todos los demás. Concluye al detectar un ciclo negativo. Su complejidad temporal depende tanto de los vértices como de la cantidad de aristas del grafo.

Se procede, entonces a dividir los algoritmos sugeridos en dos grupos, para después comparar:

- Algoritmos para calcular las rutas más cortas de una sola fuente (single-source),
- Algoritmos para calcular las rutas más cortas de todos los pares (all-pairs).

Dentro del primer grupo se encuentra el *algoritmo Bellman - Ford*, mientras que en el segundo grupo están los algoritmos de *Floyd - Warshall* y *Johnson*.

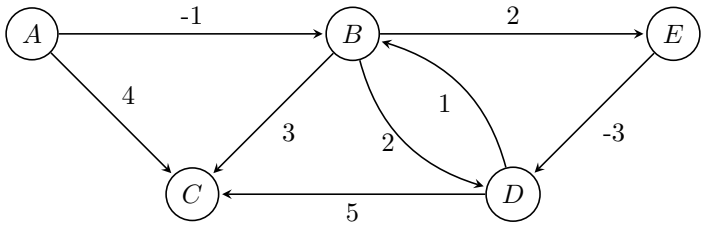
Se realiza la comparación de los algoritmos correspondientes al segundo grupo.

Si se observa la complejidad de los algoritmos, se nota que la complejidad temporal de *Floyd - Warshall* depende totalmente de la cantidad de vértices del grafo, mientras que la de *Johnson* depende tanto de la cantidad de aristas como de la de vértices.

Tras todo el análisis anterior, se concluye finalmente que *el algoritmo de Floyd - Warshall* es más efectivo para gráficos densos (muchos vértices), mientras que el *algoritmo de Johnson* es más efectivo para gráficos dispersos (pocos vértices).

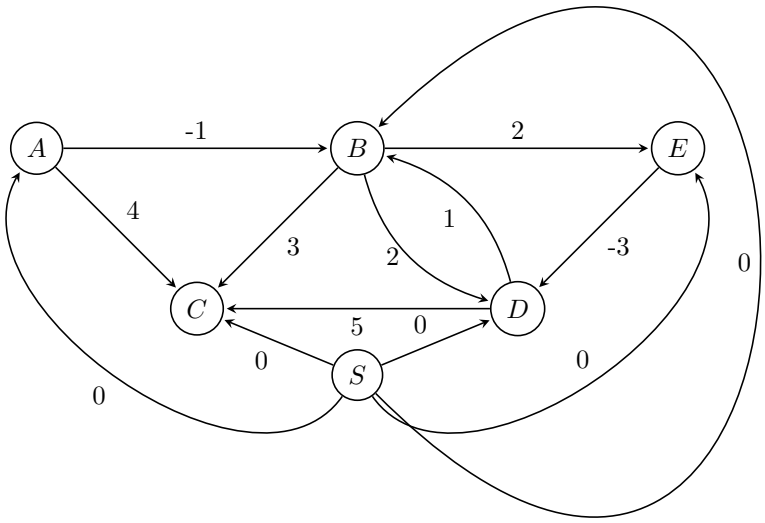
1.1.4. Ejemplo

Se cuenta con el siguiente gráfico ponderado dirigido, de cinco vértices.



Se procede a implementar el *algoritmo de Johnson*, recordando los pasos para la realización del mismo (ver **1.1.1**).

Paso 1. Se añade un nodo q al grafo, el cual está conectado a cada uno de los nodos del grafo por una arista de peso cero.



En este caso se añade el nodo S, conectado a A, B, C, D, E mediante aristas de peso nulo.

Paso 2. Se utiliza el algoritmo de *Bellman - Ford*, empezando por el nuevo vértice q, para determinar para cada vértice v el peso mínimo $h(v)$ del camino de q a v. Si en este paso se detecta un ciclo negativo, el algoritmo concluye

El algoritmo de Bellman - Ford realiza V iteraciones. Por cada una de estas iteraciones, se va a ver cada una de las aristas del grafo, realizando una actualización de la distancia entre los nodos si del origen al destino se puede mejorar la distancia. Si al concluir las V iteraciones todavía se puede mejorar la distancia, significa que hay un ciclo negativo.

En el algoritmo de Johnson interesa ver los caminos mínimos desde el vértice S, entonces se parte de lo siguiente, en donde solo se conoce la distancia de S a sí misma, mas no la de S hasta los otros nodos (se coloca ∞) :

Padre	Distancia
S = None	S = 0
	A = ∞
	B = ∞
	C = ∞
	D = ∞
	E = ∞

En la primera iteración las aristas que mejoran la distancia son todas las adyacentes a S resultando

Padre	Distancia
S = None	S = 0
A = S	A = 0
B = S	B = 0
C = S	C = 0
D = S	D = 0
E = S	E = 0

Ahora se vuelve a iterar y se ve que las aristas $A \to B$, $E \to D$ mejoran las distancias actuales, quedando

Padre	Distancia
S = None	S = 0
A = S	A = 0
B = A	B = -1
C = S	C = 0
D = E	D = -3
E = S	E = 0

Se itera nuevamente y se ve que la distancia a B es mejorada por la arista $D \to B$ con lo que se obtiene:

Padre	Distancia
S = None	S = 0
A = S	A = 0
B = D	B = -2
C = S	C = 0
D = E	D = -3
E = S	E = 0

Por más que se siga iterando las distancias no mejorarán, por lo que se concluye que el algoritmo no tiene ciclos negativos y esos son sus caminos mínimos.

	q	v_0	v_1	v_2	v_3	v_4
0	0	inf	inf	inf	inf	inf
1	0	0	-1	0	-3	0
2	0	0	-2	0	-3	0
3	0	0	-2	0	-3	0
4	0	0	-2	0	-3	0

Paso 3. Se actualiza el peso de las aristas del grafo, usando lo calculado anteriormente: una arista de u a v con peso $w(u, v)$, tiene como nuevo peso $w(u, v) + h(u)–h(v)$

Se cuenta con:

- $w(u, v)$: peso de la arista original del grafo;
- u: vértice de origen;
- $h(u)$: distancia mínima que hay desde el vértice S hasta el vértice u según el algoritmo de Bellman - Ford;
- v: vértice de destino;
- $h(u)$: distancia mínima que hay desde el vértice S hasta el vértice v según el algoritmo de Bellman - Ford;

y a partir del conocimiento de estos datos, se calcula el nuevo peso de la arista.

Entonces, por ejemplo, para la arista $A \to B$ se tiene que:

- $w(A, B) =$ -1;
- $h(A) = 0$;
- $h(B) =$ -2;

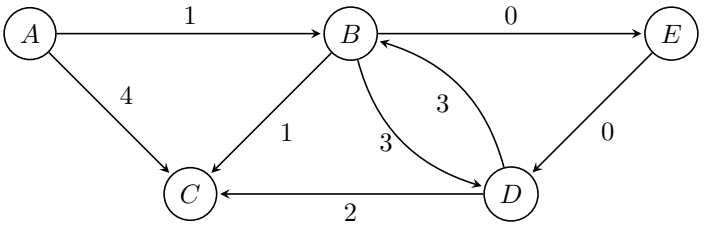
obteniendo así un nuevo peso de $-1 + 0 - (-2) = 1$.

Luego, para la arista $E \to D$ se tiene que:

- $w(E, D) =$ -3;
- $h(E) = 0$;
- $h(D) =$ -3;

obteniendo una nueva arista $E \to D$ con peso igual a cero.

Así se prosigue con el resto de aristas hasta obtener todos los nuevos valores, alcanzando el siguiente gráfico:



Paso 4. Para cada nodo s se usa el algoritmo de Dijkstra para determinar el camino más corto entre s y los otros nodos, usando el grafo actualizado anteriormente.

Para el algoritmo de Dijkstra se elige un vértice y se marcan las distancias desde éste al resto: la distancia a sí mismo es cero y la del resto es infinito (ya que no se conocen todavía). También se necesita saber cuál es el padre de ese vértice (de donde se llega a este); al iniciar el algoritmo nuestro vértice no tiene padre puesto que partimos de ahí.

A medida que se va iterando se actualizan las distancias y los padres, dependiendo de si al hacerlo mejoran las distancias que se tenían anteriormente. Esto se realiza encolando en un heap cada mejora que se hace. Las nuevas iteraciones se hacen desde los elementos encolados en el heap.

Con eso dicho, se plantea Dijsktra desde A, se coloca al padre de A como ninguno y la distancia de A a A es igual a cero, mientras que el resto de vértices tienen distancias iguales a infinito.

Padre	Distancia
A = None	A = 0
	B = ∞
	C = ∞
	D = ∞
	E = ∞

Heap	Vacío

Para la primera iteración, se actualizan las distancias de B y C que serán 1 y 4 respectivamente, y su padre será A

Padre	Distancia
A = None	A = 0
B = A	B = 1
C = A	C = 4
	D = ∞
	E = ∞

Heap	(B, 1), (C, 4)

Se desencola el vértice C y se itera con él.

Como C no aporta ninguna mejora puesto que para salir de C todas las distancias son infinito, se continúa al siguiente vértice del heap, el cual es B.

La distancia de B a E aporta una mejora, puesto que conlleva 0 más la distancia que ya traía B, obteniendo un 1; la distancia de B a D también con un costo total de 4; y finalmente también lo hace la distancia de B a C que ahora vale 2.

Padre	Distancia
A = None	A = 0
B = A	B = 1
C = B	C = 2
D = B	D = 4
E = B	E = 1

Heap	(E, 1), (C, 2), (D, 4)

Nuevamente se desencola el nodo con distancia mínima que es E, con distancia 1 y se observan sus adyacentes.

Se ve que mejora la distancia a D, obteniendo ahora un valor de 1, por lo que se actualiza el padre y se vuelve a encolar quedando

Padre	Distancia
A = None	A = 0
B = A	B = 1
C = B	C = 2
D = E	D = 1
E = B	E = 1

Heap	(C, 2), (D, 4), (D, 1)

Ahora se desencola D y se nota que esto no realiza mejora alguna; tampoco la habrá al desencolar C y D.

Se recuerda que Dijkstra es un algoritmo para calcular las rutas más cortas de una sola fuente (single-source), por ende, al igual que se aplicó el algoritmo con el vértice A como origen, debe aplicarse con los demás vértices. El resultado de eso es el siguiente

Padre	Distancia
B = None	B = 0
A = A	A = 1
C = B	C = 2
D = E	D = 1
E = B	E = 1

Padre	Distancia
C = None	C = 0
A = A	A = ∞
B = B	B = ∞
D = E	D = ∞
E = B	E = ∞

Padre	Distancia
D = None	D = 0
A = A	A = ∞
B = B	B = 3
C = E	C = 2
E = B	E = 3

Padre	Distancia
E = None	E = 0
A = A	A = ∞
B = B	B = 3
C = E	C = 2
D = B	D = 0

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

Greedy es un paradigma algorítmico en el cual se construye una solución pieza por pieza, eligiendo siempre como siguiente pieza a la que ofrece el beneficio más obvio e inmediato.

Johnson está compuesto básicamente por dos algoritmos, como ya se ha hecho mención antes: *algoritmo Bellman - Ford* y el *algoritmo de Dijkstra*.

Se analiza el algoritmo de Dijkstra, el único algoritmo no analizado en este trabajo.

Dijkstra es un algoritmo para hallar la ruta más corta desde una fuente. Siempre se elige la solución local más óptima.

Se tiene una matriz, o cualquier estructura de datos, de distancias donde todas las longitudes son infinitas. Desde el nodo inicial, se establece ese nodo en visitado y se pasa por sus nodos vecinos, actualizando sus nuevos valores en la estructura de datos de distancia si fuera necesario (si la nueva ruta es más corta que la ruta existente a ese nodo). Luego, se recorre la matriz de distancias, se encuentra el nodo más cercano al árbol actual y se repite hasta que se hayan visitado todos los nodos.

Vemos que utiliza en su funcionamiento una metodología Greedy ya que siempre elije el vértice 'más liviano' o 'más cercano' en $V - S$ para agregar al conjunto S (siendo S el conjunto de los vértices cuyos pesos finales de la fuente s ya se han determinado).

Entonces, se concluye que al incluir Johnson a Dijkstra, y utilizar éste una metodología greedy, Johnson entonces también utiliza una metodología Greedy.

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

La programación dinámica es principalmente una optimización sobre la recursividad simple. Siempre que se vea una solución recursiva que tiene llamadas repetidas para las mismas entradas, se puede optimizarla mediante la programación dinámica.

La idea es simplemente almacenar los resultados de los subproblemas, de modo que no se tengan que volver a calcular cuando sea necesario más adelante. Esta simple optimización reduce la complejidad del tiempo de exponencial a polinomio.

Como se dijo en el punto anterior, Johnson está compuesto básicamente por dos algoritmos: *algoritmo Bellman - Ford* y el *algoritmo de Dijkstra*.

Bellman - Ford, como ya se ha dicho anteriormente, es un algoritmo que calcula las rutas más cortas desde un único vértice de origen a todos los demás vértices en un dígrafo ponderado, detectando los ciclos negativos. Para conocer su implementación ver **1.1.2, subsección Bellman - Ford**.

Se analiza y se nota que:

- Calcula las rutas más cortas de forma ascendente.
- Los valores intermedios se almacenan y se utilizan para los valores del siguiente nivel.
- Primero calcula las distancias más cortas para los caminos más cortos que tienen como máximo un borde en el camino, y lo almacena.

- Luego, calcula las rutas más cortas con un máximo de 2 aristas, y así sucesivamente.
- Después de la i- ésima iteración del bucle exterior, se calculan las rutas más cortas con un máximo de i aristas.

Se ve con claridad que sigue el enfoque de programación dinámica.

Entonces, se concluye que al incluir Johnson a Bellman - Ford, y seguir éste una metodología de programación dinámica, Johnson entonces también utiliza una metodología de programación dinámica.

Capítulo 2

Un poco de teoría

2.1. Objetivos

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?

Pista: probablemente no haya una respuesta correcta para este problema, solo justificaciones correctas.

2.1.1. Greedy, división y conquista, y programación dinámica

Programación Dinámica

La programación dinámica consiste en la división de un problema en subproblemas. Una vez dividido en estos subproblemas, se buscan las soluciones óptimas de los mismos para encontrar la solución óptima global del problema inicial utilizándolos. Para evitar recalcular los subproblemas que aparecen repetidos se utiliza la memoización, la cual es una técnica que consiste en almacenar los resultados de los subproblemas calculados para evitar repetir su resolución cuando vuelva a requerirse, consiguiendo reducir significativamente la complejidad temporal de la solución.

División y Conquista

Division y conquista está basado en la resolución recursiva de un problema dividiéndolo en subproblemas cada vez mas pequeños hasta que puedan ser resueltos directamente. Finalmente, las soluciones de estos subproblemas se combinan para formar la solución del problema general. Programacion Dinamica consiste en dividir el problema en subproblemas mas pequeños, resolverlos y combinar las soluciones obtenidas para calcular la solución del problema inicial. Ademas, se van guardando los resultados de los subproblemas para

no calcularlos mas de una vez.

Division y conquista y programacion dinamica comparten la dinamica de dividir el problema en subproblemas mas pequeños, aun asi, el uso de una u otro depende de los subproblemas; en caso de que estos no suelan repetirse, division y conquista es una mejor alternativa, en caso contrario, se deberia usar programacion dinamica. Programacion dinamica resuelve el problema desde abajo hacia arriba, es decir, el problema no puede ser resuelto hasta que se resuelvan todos los subproblemas. En cambio, greedy resuelve el problema de arriba hacia abajo, es decir, va reduciendo el problema tomando la mejor solución local. Ademas, programacion dinamica debe resolver cada posibilidad para resolver el problema, por lo tanto es mas caro que Greedy. Aun asi, muchos problemas no pueden resolverse con un algoritmo Greedy, por lo que se debe utilizar programacion dinamica. Por lo tanto, primero se deberia intentar resolver el problema con un algoritmo greedy, y en caso de no poder, utilizar programacion dinamica.

Greedy

La idea básica detrás de estos algoritmos es subdividir el problema en subproblemas con una jerarquía entre ellos y resolver estos subproblemas de forma miope, sin considerar la instancia general sino el caso particular del subproblema actual. Es decir, tomar una decisión local óptima esperando que esa decisión lleve a una solución global óptima. Para tomar estas decisiones óptimas de utiliza una heurística. [1]

Un algoritmo greedy es una estrategia de busqueda por la cual se sigue una heurística consistente en elegir la opcion optima en cada paso local esperando de esta manera llegar a una solución general óptima. [1]

2.1.2. ¿Qué algoritmo elegiría para resolver el problema?

Sabemos que en este caso la complejidad temporal del algoritmo de Programación Dinámica es mejor que la del algoritmo Greedy, aún así, Programación Dinámica ocupa mucho más espacio en memoria ya que guarda los resultados de los subproblemas para no volver a calcularlos denuevo. Teniendo esto en cuenta, si quisieramos priorizar el tiempo de ejecución de nuestro algoritmo, la Programación Dinámica sería la mejor opción. En caso contrario, donde nuestros esfuerzos de optimización se encuentren en la memoria y los recursos de la CPU, convendría utilizar el algoritmo Greedy.

Bibliografía

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms (tercera edición), MIT Press (2009).

[2] J. Kleinberg, E. Tardos, Algorithm Design, Addison Wesley (2006).

[3] Florida International University (Web).

[4] Los artículos de Donald B. Johnson (Publicaciones de ACM)

[5] Johnson, Donald B. (1977), .Efficient algorithms for shortest paths in sparse networks", Journal of the ACM