DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2: Llenalo con super

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Martino, Maximiliano	123/17	maxii.martino@gmail.com
Blufstein, Marcos	300/17	mjblufstein@dc.uba.ar
Peretti, Olivia	359/17	operetti@dc.uba.ar
Pironio, Nicolás	37/17	npironio@dc.uba.ar

Resumen: en el siguiente trabajo se propone e implementa, mediante distintas técnicas algorítmicas, una solución al problema de ir de una ciudad a otra ciudad gastando el menor dinero posible en nafta. El objetivo del presente trabajo es comparar los diferentes algoritmos propuestos para resolver el problema, en función de distintas instancias del problema.

Palabras clave: camino mínimo, Floyd-Warshall, Dijkstra, Bellman-Ford.

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

En este trabajo resolveremos el problema de encontrar un camino para ir de una a ciudad a otra minimizando el costo del combustible. Para modelar el problema computacionalmente, utilizamos la siguiente representación: tendremos un grafo G, con n nodos y m aristas. Cada nodo representa una ciudad distinta y cada arista una ruta entre dos ciudades. Además cada arista m_j tendrá asignado un valor l_j que representa la cantidad de litros de nafta que necesita nuestro vehículo para recorrer dicha ruta. Por otro lado, cada nodo n_i también tendrá un valor asignado c_i, que representa el costo de la nafta por litro en dicha ciudad.

La capacidad del tanque de nuestro auto será de 60 litros, y nuestro algoritmo devolverá el costo mínimo de combustible partiendo de una ciudad y terminando en otra.

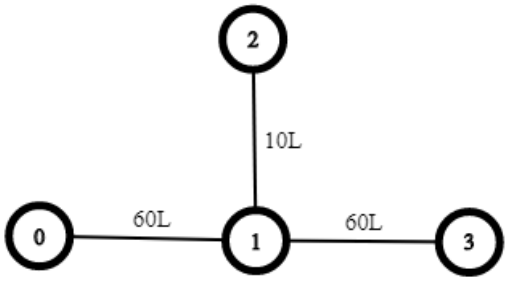
Para resolver este problema implementaremos cuatro algoritmos distintos.

1. Algoritmo de Dijkstra.
2. Algoritmo de Dijkstra con cola de prioridad.
3. Algoritmo de Bellman-Ford.
4. Algoritmo de Floyd-Warshall.

El programa toma como entrada un archivo cuya primera línea debe tener dos enteros n y ms correspondientes a la cantidad de ciudades y a la cantidad de rutas respectivamete. Luego habrá n líneas. Cada una con un entero c_i que representa el costo del litro de combustible en la ciudad i. Luego deberán haber m líneas con tres entero a_i, b_i y l_i. a_i y b_i indicarán las ciudades conectadas por la ruta, y l_i la cnatidad de litros necesarios para recorrerla. Asumiremos que todos los valores en la entrada serán enteros positivos. Finalmente, el archivo debe tener una línea más. La última línea deberá tener el identificador del método con el que se quiere correr el programa. Los identificadores son los siguientes:

1. d: Dijkstra.
2. dp: Dijkstra con cola de prioridad.
3. bf: Bellman-Ford.
4. fw: Floyd-Warshall.

La salida, como pide el enunciado, consistirá de $n * (n - 1)$ líneas. Cada línea tendrá tres enteros a_i, b_i y s_i. a_i y b_i indicarán las ciudades consideradas. s_i indicará el costo mínimo para ir por todas las ciudades comenzando en a_i y finalizando en b_i. Las líneas de salida estarán ordenadas según el orden lexicográfico de las ciudades pertinentes. [1] Veamos un ejemplo:



Supongamos que en este caso los costos de combustible para las ciudades 0, 1, 2 y 3 son respectivamente 1, 50, 10, 50. Veamos las distancias de 0 a todos los demas. En primer lugar, tenemos una única forma de llegar al nodo 1, cargando 60L en un total de 60$. Luego, la distancia de 0 a 1 es 60. Una vez que estamos en 1, tenemos

[1] Extraído del enunciado.

dos opciones. Podemos cargar 10L e ir a 2 (por 500$), o cargar 60 e ir a 3 (por 3000$). Notemos que esta no es la mejor estrategia para llegar a 3, ya que podríamos ir a 2, cargar 60L, volver a 1, cargar 10L y de ahí pasar a 3 (por 1600$). Esto nos resulta en costos totales de 60$, 560$ y 1660$.

Sin embargo, notemos que el camino de 0 a 3 resulta ser un camino no simple, lo que imposibilita la utilización de algunos algoritmos de camino mínimo, como Dijkstra. Por este motivo es que la transformación al grafo con 61*n nodos resulta de utilidad, ya que allí, este camino se vuelve simple. Nos permite estar en una misma ciudad más de una vez, pero el estado del tanque será distinto.

2. Desarrollo

2.1. Transformación del grafo original

Para resolver el problema planteamos la siguiente modificación del grafo original: cada ciudad c, antes representada por un único nodo, será ahora representada por 61 nodos distintos. Llamemos a cada uno de estos nodos c_i, con $0 \leq i \leq 60$. Cada uno de estos significa "estoy en la ciudad c con i litros de nafta en mi tanque".

En cuanto a las aristas, las dividiremos en dos grupos:

- Tipo A: Las que van de un c_i a c_{i+1}. Estas van a tener como peso el costo de cargar un litro de nafta en la ciudad c.
- Tipo B: Las que van de una ciudad a otra. Estas aristas tendrán peso cero ya que al viajar no gasto dinero. Dos nodos c_i, c'_j, estarán conectados si y sólo si existe una ruta de c a c' con distancia $i - j$.

Veamos que hallar un camino mínimo en el grafo, es equivalente a hallar una solución de nuestro problema. Comencemos por ver que existe una biyección entre los caminos de ambos grafos.

Demostración: Llamemos G al grafo original y G' al modificado. Y llamemos s al vértice de salida, y d al vértice de destino.

Veamos primero que todo camino del grafo original está en el grafo modificado.

Sea C un camino de s a d en G. Construyamos un camino C' en G' que represente unívocamente a C. Supongamos que estoy en el i-ésimo vértice del camino, v_i. Ahora, si en C cargamos k_i litros de nafta en v_i entonces vamos a tener k_i aristas de tipo A moviéndose entre los distintos nodos que representan a v_i en C'. Notemos que esta elección es única, pues estando en una ciudad con l litros, solo puedo cargar un litro de nafta, y hay una única arista que representa esta transición. Luego, cuando pasemos de v_i a v_{i+1}, le vamos a agregar a C' una arista de tipo B entre v_i y v_{i+1}. Esta arista también es única, ya que ir de una ciudad a otra tiene un costo fijo. Luego, si repetimos este proceso para cada v_i en C, obtenemos el C' que buscábamos.

Ahora notemos que si tomamos un camino C' en G', las aristas de tipo A representan cuánta nafta cargo en un camino del grafo G, y las aristas de tipo B representan el movimiento entre ciudades. Sabiendo esto, la construcción del camino C en G resulta análogo.

Luego, si realizamos un algoritmo de caminos mínimos sobre el grafo modificado, el costo mínimo encontrado coincide con el costo mínimo del grafo original, que es la respuesta al problema que queríamos resolver.

2.2. Dijkstra

Utilizaremos Dijkstra para calcular distancia mínima de un nodo a todos sus vecinos. Algunas particularidades de Dijkstra son:

1. Dijkstra tiene como precondición que no haya ejes de peso negativo, lo cual nuestro problema garantiza.
2. Es similar al algoritmo de Prim, pero generalizado para una función $c_\bullet$ creciente.
3. Si tomamos la función c_+ este algoritmo resuelve el problema de camino mínimo desde un nodo a los demás.

El pseudocódigo de Dijkstra[2] es el siguiente:

Algorithm 1 Dijkstra$_\bullet$

```
1: procedure Dijkstra•(grafo G, nodo salida, funcion c•)
2:     res ← crearVectorDeTamañoConValor(cantNodos, indefinido)
3:     res[salida] ← elementoNeutro(c•)
4:     for i=1,...,cantNodos-1 do
5:         (x,y) ← arista segura de G con mínimo c•(xy)      ▷ Consideramos x ya definido, y no definido.
6:         res[y] ← c•(x,y)
7:     end for
8:     return res
9: end procedure
```

[2] fuente: teórica.

En particular para la función c_+:

Algorithm 2 Dijkstra+

```
1: procedure Dijkstra+(grafo G, nodo salida)
2:     res ← crearVectorDeTamañoConValor(cantNodos, indefinido)
3:     res[salida] ← 0
4:     for i=1,...,cantNodos-1 do
5:         (x, y) ← arista segura de G con mínimo c+(xy)      ▷ Consideramos x ya definido, y no definido.
6:         res[y] ← c+(x, y)                                  ▷ res[y] = res[x]+peso(xy)
7:     end for
8:     return res
9: end procedure
```

Complejidad[3]:

Sea n la cantidad de vértices, y m la cantidad de aristas:

- Caso ralo: usar una cola de prioridad sobre heap con todas las aristas seguras. Costo $T(n+m) = \mathcal{O}(mlgn)$
- Caso general: usar diccionario de costo que contenga el valor de la mejor arista segura para cada $y \notin V(T)$. Costo $T(n + m) = \mathcal{O}(m + nlgn)$.
- Caso denso: implementar el diccionario de costos sobre un vector. Costo $T(n + m) = \mathcal{O}(n^2)$.

La implementación fue extraída de internet[4] y modificada para adaptarla a nuestras estructuras. Como fue implementado sobre vector la complejidad es $\mathcal{O}(n^2)$.

Notemos que Dijkstra nos devuelve el camino mínimo de un vértice hacia todos sus vecinos. Trasladado a nuestro problema, esto nos daría el costo mínimo de combustible que necesitamos para partir de una ciudad y llegar a cualquier otra. Sin embargo el problema nos pide saber esto mismo pero partiendo de cualquier ciudad. Es por esto que debemos llamar a Dijkstra una vez por nodo, es decir llamamos a Dijkstra n veces, lo cual transforma nuestra complejidad en $\mathcal{O}(n^2)$.

Sin embargo, nos faltan analizar unos detalles más, ya que no corremos el algoritmo sobre el grafo original, sino sobre una modificación.

En primer lugar la cantidad de nodos del grafo aumenta. Si la cantidad de ciudades era n, la cantidad de nodos del grafo modificado será $61 * n$. Pero como 61 es una constante, las complejidades teóricas no se modifican, sin embargo en la práctica la diferencia entre correr Dijkstra con n nodos, y con $61 * n$ nodos, puede ser notoria.

Algo similar sucede con la cantidad de aristas. Dada una ruta del grafo original, como mucho se va a replicar 120 veces. Esto sucede cuando la arista tiene costo 1L, entonces voy de $a_{60} \rightarrow b_{59}$, ...$a_1 \rightarrow b_0$ y viceversa. Además, se agrega una cantidad fija de 60*n aristas, las de tipo A. Sin embargo, como asumimos que el grafo es conexo, n=O(m). Luego la cantidad de aristas en el grafo modificado es $\mathcal{O}(\texttt{cantidadDeRutas})$, por lo tanto esto tampoco afecta la complejidad teórica.

Finalmente tiene sentido preguntarnos sobre qué nodo empezamos y sobre cuál terminamos, ya que antes cada nodo represtaba una única ciudad, y en el nuevo grafo, hay 61 nodos que representan la misma ciudad. La respuesta es que por cada ciudad c nos interesa salir de un solo nodo, el que representa estar en la ciudad c con 0 litros de nafta. Por esta razón, solo es necesario llamar a Dijkstra, $\texttt{cantidadDeCiudades}$ veces[5]. Además en cuanto a los nodos de llegada solo nos interesa el caso de llegar con 0 litros, ya que en otro caso habríamos cargado nafta de más y sería subóptimo.

2.3. Dijkstra con cola de prioridad

Este algoritmo tiene el mismo pseudocódigo que la sección anterior. Es Dijkstra con una implementación particular de cola de prioridad. Por esta razón, como expusimos anteriormente, la complejidad teórica de cada

[3] fuente: clases teóricas.
[4] fuente: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
[5] Por esta razón, en main.cpp el for aumenta de a 61.

llamado es $\mathcal{O}(mlgn)$.

También como en el algoritmo, anterior lo llamamos n veces para computar los caminos mínimos de cada vértice hacia todos los demás. Luego la complejidad total es $\mathcal{O}(nmlgn)$.

2.4. Bellman-Ford

Al igual que Dijkstra, Bellman-Ford calcula los caminos mínimos desde un vértice a todos los otros.
Su complejidad temporal es $\mathcal{O}(mn)$, y la espacial $\mathcal{O}(n+m)$.
Pseudocódigo del algoritmo[6]:

Algorithm 3 Bellman-Ford

```
1: procedure Bellman − Ford(grafo G, nodo salida)
2:     Crear un diccionario D^0 con D^0[salida] = 0 y D^0[y] = ∞, y ≠ v
3:     for i=1,...,cantNodos-1 do
4:         Crear un diccionario D^{i−1}[y] = mín{ D^i[x] + peso(xy) / x ∈ N^in[y] } ∀ y ∈ V(G).
5:     end for
6:     return D^{n−1}
7: end procedure
```

Por lo mismo que en Dijkstra, hacemos cantidadDeCiudades llamados a Bellman-Ford, y la complejidad teórica es $\mathcal{O}(mn^2)$, con n la cantidad de ciudades y m la cantidad de rutas.

2.5. Floyd-Warshall

Este algoritmo se diferencia de los anteriores en que en vez de calcular las distancias mínimas de un vértice hacia los otros, calcula las de todos los vértices hacia todos los otros. Por esta razón, no hay que llamarlo más de una vez.

La complejidad de este algoritmo es $\mathcal{O}(n^3)$. Sin embargo, hay que notar que n no es la cantidadDeCiudades, sino que es 61*cantidadDeCiudades. Aunque la complejidad teórica no cambie, esto puede tener un impacto en el tiempo de ejecución en la práctica. Notemos además que en los algoritmos anteriores hacemos $cantidadDeciudades$ llamados, salteando otros 60 nodos por ciudad que no nos interesan para resolver el problema. Al llamar a Floyd-Warshall una única vez, estos $60 * cantidadDeCiudades$ problemas que habíamos salteado, se resuelven.

[6] fuente: teórica.

3. Implementación

En esta sección explicaremos las estructuras elegidas para representar los datos.

3.1. Arco

Para representar los ejes de un grafo creamos la struct arco, con los siguientes atributos:

- peso
- cola
- cabeza

Si tenemos una arista m que va de un nodo `a`, a un nodo `b` con un valor `k`. Luego podemos representar m de la siguiente forma:

- peso = k
- cola = a
- cabeza = b

3.2. Grafo

Para representar grafos elegimos una representación basada en lista de adyacencias. Para esto, creamos la estructura grafoAd con los siguientes atributos:

- listaAd: es un vector de listas. Cada posición `i` del vector guarda una lista con los arcos salientes del nodo `i`. Es decir, guarda los vecinos a los que se puede acceder desde `i`, con el peso de la arista que lo permite[7].
- cantNodos: es un entero que guarda la cantidad de nodos del grafo.
- cantAristas: es un entero que guarda la cantidad total de aristas del grafo.

Y los siguientes métodos:

- grafoAd: es el constructor de la clase. Inicializa cantNodos y cantAristas en cero, y la lista de adyacencias como un vector vacío.
- agregarNodo: agrega una nueva posición al vector, con una lista vacía.
- agregarEje: agrega una arista al grafo, con el peso y los nodos especificados en los parámetros.

[7] Además por ser eje guarda la cola de la arista, que en este caso será `i` para todos los ejes de la lista.

3.3. Lectura de la entrada

A partir de la entrada del programa, generamos el grafo modificado. Para lograrlo, generamos un grafo con n'=61*n nodos, donde el nodo i representa al nodo i/61 del grafo original, en un estado con i mod 61 Litros de nafta. Por ejemplo, si teníamos un grafo con 100 nodos, en el grafo modificado el nodo 176 representa a la ciudad 2 con 54 litros en el tanque.

Sabiendo esto, para cada costo C_i de combustible leido generamos 61 vertices, donde cada uno está unido al siguiente con una arista de costo C_i. De esta manera, logramos generar todas las aristas de tipo A.

Por último, para cada ruta leida, generamos las aristas de tipo B. Si una ruta es representada con los enteros a_i, b_i, l_i (ambas ciudades seguidas del costo de la ruta), vamos a decir que una arista representa un movimiento valido si a_i representa un estado con K litros en el tanque, y b_i representa un estado con k-l_i litros. Luego, simplemente debemos conectar un nodo de la ciudad a_i a otro de b_i (y viceversa) si representan un movimiento valido.

Presentamos el pseudocodigo a continuación:

Algorithm 4 crear grafo modificado

```
1:  procedure crearGrafoModificado
2:      i ← 0
3:      for i < cantidadDeCiudades do
4:          G.agregarNodo()
5:          j ← 1
6:          precioNafta ← precioNaftaEn(i)
7:          for j < 61 do                                   ▷ Generamos aristas de tipo A
8:              G.agregarNodo()                             ▷ El nodo de estar en la ciudad i con j litros
9:              a ← 61 * i + j − 1                          ▷ Tener j-1 litros en la ciudad i
10:             b ← siguienteDe(a)                          ▷ Tener j litros en la ciudad i
11:             G.agregarEje(precioNafta, a, b)             ▷ Costo de pasar de tener j-1 litros, a tener j en la ciudad i
12:         end for
13:     end for
14:     i ← 0
15:     for i < cantidadDeRutas do
16:         a ← extremoADeArista(i)
17:         b ← extremoBDeArista(i)
18:         l ← cantidadDeLitrosDeAaB(i)
19:         j ← 0
20:         for j < 61 do                                   ▷ Generamos aristas de tipo B
21:             if j-l ≥ 0 then                             ▷ Si puedo ir de a a b con la nafta que tengo
22:                 a' ← a*61+j
23:                 b' ← b*61+j-l
24:                 G.agregarEje(0, a', b')                 ▷ La arista de viajar de la ciudad a a b con l litros de nafta
25:                 a' ← a*61+j-l
26:                 b' ← b*61
27:                 G.agregarEje(0, b', a')                 ▷ La arista de viajar de la ciudad b a a con l litros de nafta
28:             end if
29:         end for
30:     end for
31:     return G
32: end procedure
```

Complejidad: $\mathcal{O}(n + m)$

3.4. Impresión por salida estándar

Para imprimir el resultado, simplemente hay que ser cuidadosos y recordar que nuestro nuevo grafo contiene nodos que no nos interesan (Solo queremos empezar y terminar en estados en los que tenemos 0 Litros en el tanque). De esta forma, si terminamos con una matriz de distancias, solo debemos imprimir aquellas posiciones cuyas columnas o filas sean iguales a 0 modulo 61.

4. Experimentación

4.1. Variación en la cantidad de nodos

En esta experimentación, generamos grafos de la siguiente manera:

- n fue variando entre 10, 100, 200, 300, 400 y 500.
- m siempre mantuvo la proporción m = $\frac{n*(n-1)}{4}$
- los costos de nafta fueron elegidos de manera aleatoria en el rango [1,100]
- los largos de las rutas en litros fueron tomados de manera aleatoria en el rango [1,60]

La idea es ver como se comportan los distintos algoritmos, cuando vamos teniendo un grafo denso cada vez más grande. Es importante tener en cuenta que ambas versiones de Dijkstra y Bellman-Ford fueron corridos n veces (ya que el output del programa tiene todas las distancias posibles) mientras que Floyd se corre una única vez.
Los resultados obtenidos fueron los siguientes:

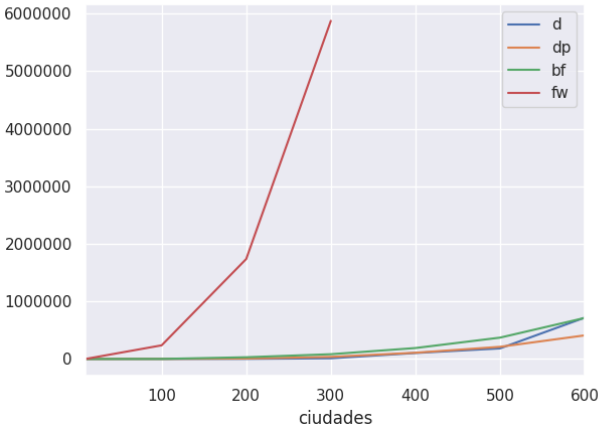


Figura 1: Comparación del problema con los 4 algoritmos

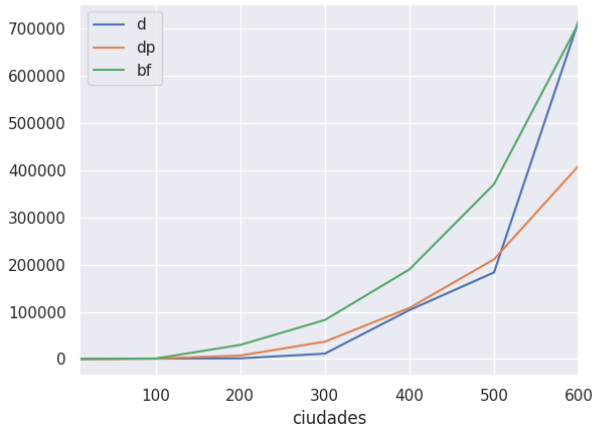


Figura 2: Comparación del problema quitando el algoritmo de Floyd

Como podemos observar, el algoritmo de Floyd crece de manera mucho más acelerada que los otros algoritmos, a tal punto que no pudo ser corrido para los casos más grandes. Esto puede deberse a que el grafo modificado que usamos para resolver el problema tiene 61*n nodos, por lo que la matríz que debe resolver Floyd resulta muy grande. De esta manera, está calculando las distancias mínimas tomando como origen a cualquier estado (cantidad de litros iniciales) de una ciudad, cuando sólamente nos importa tomar el estado del tanque vacío como origen. Esto no afecta tan marcadamente a los otros algoritmos, ya que únicamente los corremos con dicho estado como origen.

En este caso, sabemos que m=$\mathcal{O}(n^2)$, por lo que las complejidades de cada algoritmo corrido n veces son:

- Dijkstra: $\mathcal{O}(n^3)$
- Dijkstra con cola de prioridad: $\mathcal{O}(n^3 * lgn)$
- Bellman-Ford: $\mathcal{O}(n^4)$
- Floyd: $\mathcal{O}(n^3)$

Como se ve en el gráfico, todas las curvas tienen un crecimiento polinomial, como se esperaba. Lo que quizas es mas inusual, es el comportamiento del algoritmo de dijkstra sin cola de prioridad. En estos casos, se esperaría que tenga una mejor performance que los demás, pero vemos que tiene un crecimiento considerablemente más acelerado. Esto probablemente se deba a que, aunque m=$\mathcal{O}(n^2)$, en el grafo modificado la constante oculta sea lo suficientemente pequeña como para que n^2 sea considerablemente peor a m en la práctica.

4.2. Variación en la proporción de aristas

En esta experimentación, generamos grafos de la siguiente manera:

- n fue fijado en 100.
- m fue variando entre 10, 100, 500, 1000, 1500, 2000, 3000, 4000
- los costos de nafta fueron elegidos de manera aleatoria en el rango [1,100]
- los largos de las rutas en litros fueron tomados de manera aleatoria en el rango [1,60]

En este experimento analizamos que sucede si, dado un conjunto fijo de ciudades, vamos construyendo nuevas rutas. Al igual que en el experimento anterior, todos los algoritmos salvo el de Floyd fueron corridos n=100 veces.
Veamos los resultados:

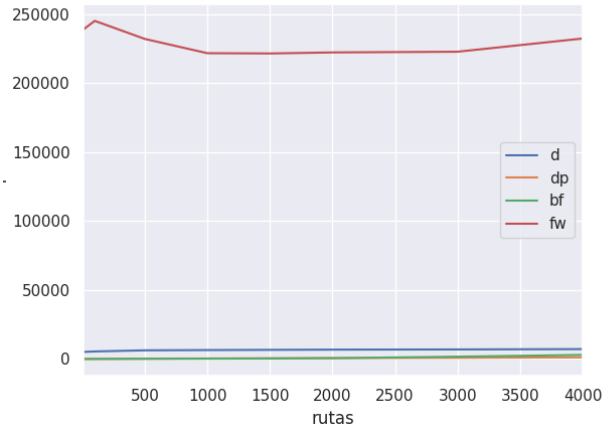


Figura 3: Comparación del problema con los 4 algoritmos

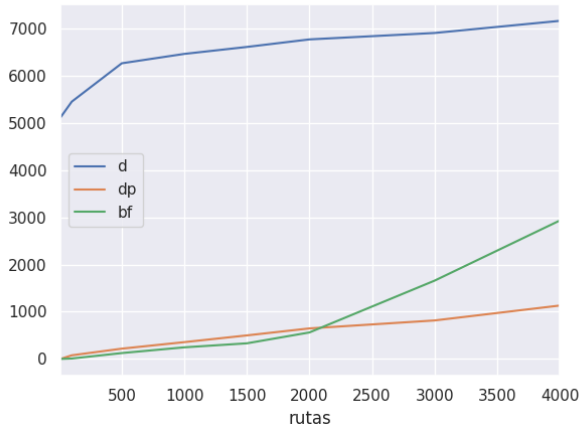


Figura 4: Comparación del problema quitando el algoritmo de Floyd

Nuevamente, el algoritmo de Floyd fue mucho más lento que los demás. Sin embargo, podemos ver que en este caso el tiempo de computo de Floyd se mantiene constante. Esto era de esperarse, pues la complejidad del mismo depende únicamente de n, que fue fijado en 100. En cuanto a los otros algoritmos, asumiendo que en este caso n es una constante, las complejidades teóricas en cada caso son:

- Dijkstra: $\mathcal{O}(1)$
- Dijkstra con cola de prioridad: $\mathcal{O}(m)$
- Bellman-Ford: $\mathcal{O}(m)$

Sin embargo, es importante notar que existe un $\mathcal{O}(m)$ en la complejidad del algoritmo de Dijkstra sin cola de prioridad, ya que se recorren todas las aristas. Esto explica el crecimiento de su curva en el gráfico. En cuanto a Bellman-Ford y Dijkstra con cola de prioridad, si recordamos que el m va acompañado de un n y de un log(n) respectivamente, podemos explicar por qué difieren las pendientes de sus rectas.

5. Conclusión y trabajos futuros

En los resultados de los experimentos, notamos que en la mayoría de los casos obtenemos mejores tiempos utilizando el algoritmo de Dijkstra con cola de prioridad. Hubo casos en los que tanto Bellman-Ford como la otra versión de Dijkstra obtuvieron mejores resultados. Sin embargo, en ambos experimentos, su crecimiento asintótico parece ser mucho menor. En cuanto a posibles mejoras para solucionar el problema, en un comienzo consideramos una estrategia greedy para solucionarlo. El mismo consistía en completar el grafo de entrada (si dos ciudades no estan directamente conectadas, agregamos una arista entre ellas con peso igual al camino mínimo entre ambas, medido en litros. Si dicho costo era mayor a la capacidad del tanque, el peso sería infinito), y en el hecho de que, dado un camino fijo de una ciudad A a otra ciudad B, hay una estrategia óptima para cargar combustible:

Si $s = u_1, u_2, ..., u_l$ representa las paradas de recarga de un posible camino, la siguiente es una estrategia óptima para decidir la cantidad de gas que se va a llenar en cada parada: en la parada u_l cargamos nafta suficiente para alcanzar al nodo destino con el tanque vacío; para $j < l$:

1. Si $c(u_j) < c(u_{j+1})$, entonces en u_j llenamos el tanque.
2. Si $c(u_j) \geq c(u_{j+1})$, entonces en u_j llenamos lo suficiente para llegar a u_{j+1}

Sin embargo, hubo casos que nos trajeron problemas al utilizar este método, como por ejemplo el grafo mostrado en la introducción, ya que terminabamos con caminos no simples. Pensamos en posibles soluciones a éste problema, pero complicaban considerablemente la implementación, por lo que decidimos abandonar la estrategia por la presentada en el trabajo.