DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Llenalo con super

Algoritmos y Estructura de Datos III

Primer Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Amigo, Martín Ignacio	368/17	martinamigo@protonmail.com
de Renteria, Dago Martín	036/17	momasosa1@gmail.com
Festini, Santiago Alberto	311/17	festini.santiago21@gmail.com
Regnier, Melissa	052/17	melissaregnier.98@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo práctico, buscamos utilizar los algoritmos aprendidos de camino mínimo en una aplicación concreta de un problema que podría presentarse en la vida real. Para esto, una vez modelado el problema en forma de grafo, aplicaremos los algoritmos de Dijkstra, Bellman-Ford y Floyd-Warshall para así poder comparar sus rendimientos.

1.1. Contexto y motivación

El problema que resolveremos consiste en abaratar costos de nafta para una empresa que desea distribuir un producto entre ciertas ciudades donde tanto la distancia entre ellas como el costo de cargar nafta en cada una difiere. La empresa desea saber el costo mínimo entre cualquier par de ciudades, sabiendo que se cuenta con un auto con un tanque de 60 litros. La empresa cuenta con el mapa de las ciudades y las distancias de las rutas que las unen, además de los precios de la nafta en cada ciudad. A partir de esto, deberemos construir nuestra solución.

1.2. Abstracción y simplificación

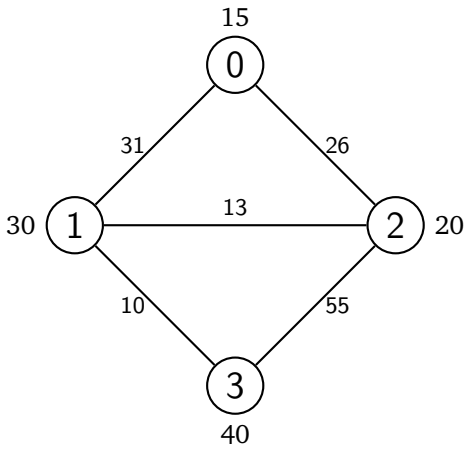
Para facilitar el entendimiento y poder abstraernos de los detalles particulares, definiremos las siguientes notaciones:

- n : cantidad de ciudades
- m : cantidad de rutas bidireccionales existentes entre las ciudades
- l_i : cantidad de litros de nafta que supone recorrer la $i - ésima$ ruta con $0 \leq i < m$. Este valor lo acotaremos por 60 ya que es la capacidad máxima de nuestro tanque y por tanto no tiene sentido considerar rutas que no podamos recorrer en su totalidad. Además, consideramos que es mayor que 0 ya que sino dos ciudades distintas estarían en el mismo lugar y se tratarían de la misma, resultando en ciudades duplicadas.
- c_i : costo de la nafta en la ciudad $i - ésima$ con $0 \leq i < n$. Este valor es positivo ya que se trata del precio de la nafta que por lo general no la regalan ni te pagan por llevártela.

1.3. Ejemplificación

A continuación presentamos un ejemplo simple en donde contamos con $n = 4$, $m = 5$ y los siguientes costos:

- $c_0 = 15 \quad c_1 = 30 \quad c_2 = 20 \quad c_3 = 40$



Tomemos como ejemplo el camino desde la ciudad 1 a la ciudad 0. Hay una ruta existente entre ambas, con una cantidad necesaria de 31 litros para recorrerla. Si observamos el resto de las aristas, vemos también que este el camino de menor distancia entre ambas ciudades. Como $c_1 = 30$, el costo total de este camino es de \$930. Sin embargo, si buscamos minimizar el costo vemos que podríamos ir de la ciudad 1 a la 2 con costo \$390 y luego de la 2 a la 0 con costo \$520 cargando 13 y 26 litros respectivamente. El costo total de este nuevo camino es de \$910, resultando menor que el camino de distancia menor antes propuesto. Si se observan el resto de los caminos posibles, se verá que este es el camino que minimiza el costo.

Analicemos ahora el camino desde la ciudad 2 a la ciudad 3. Vemos que el camino de menor distancia es ir hasta la ciudad 1 y desde allí a la 3. Este también resulta el camino de menor costo. En cuanto al valor del mismo, podemos ver que si bien podría cargar sólo la cantidad necesaria para llegar hasta la ciudad 1 y luego allí cargar lo faltante para ir a la ciudad 3, esto resulta en un costo de \$560 lo cual es mejorable. Si cargamos en cambio toda la nafta necesaria en la ciudad origen, que tiene nafta más barata, el costo total termina siendo \$460 con lo cual logramos ahorrar \$100.

El resultado final para cada par de ciudades sería el siguiente:

	0	1	2	3
0	0	465	390	615
1	910	0	390	300
2	520	260	0	460
3	1310	400	790	0

donde la fila i representa el costo de ir desde la ciudad i al resto.

2. Desarrollo

2.1. Modelización

Para poder resolver el problema, primero debemos llevar a cabo una modelización apropiada del problema en grafos que nos permita aplicar los algoritmos de camino mínimo tradicionales sin tener que realizar modificaciones $ad\ hoc$ a los mismos. Llamaremos G al grafo modelador.

Una aproximación inicial sería modelar cada ciudad como un vértice, y cada ruta bidireccional como dos aristas con direcciones opuestas entre los vértices correspondientes a las ciudades que une. Sin embargo, esto supone mantener no sólo pesos en las aristas sino también en los vértices, lo cual nos impide aplicar los algoritmos de camino mínimo sin realizar modificaciones no triviales. Por otra parte, un grafo modelado de esta forma no cumpliría con la optimalidad de los subcaminos que es necesaria para realizar cualquiera de los algoritmos.

Entonces, como es necesario deshacernos de los pesos en los vértices sin perder esa información, decidimos representar cada ciudad con 60 vértices. De esta forma, $v_{i,j}$ con $0 \leq i < n$ y $0 \leq j < 60$, es un vértice que representa al estado de la $i - ésima$ ciudad cuando llegamos a ella con j litros restantes

de nafta en el tanque. Notemos que no podría llegar a una ciudad desde otra con 60 litros ya que esto implicaría haber necesitado 0 litros para llegar a ella y, como dijimos en la sección anterior, consideramos que todas las rutas requieren cantidades positivas de litros de nafta.

Así, podemos definir tanto las aristas como sus pesos entre dos vértices cualesquiera unívocamente. Supongamos que existe una ruta entre la ciudad 1 y la ciudad 2; recordemos que cada una de ellas tiene 60 vértices en nuestro grafo modelador. Desde un vértice cualquiera $v_{1,i}$ podremos llegar a la ciudad 2 cargando distinta cantidad de litros de nafta. Esta cantidad de litros, nombrémosla q, dependerá mayormente de dos cosas. Por un lado, de cuántos litros sean necesarios para recorrer la ruta que conecta 1 y 2, llamémoslo l_0 considerado que se trata de la ruta 0. Por otro lado, de la capacidad máxima del tanque, ya que no podremos cargar más cantidad que la soportada por el mismo. Por lo tanto, si queremos saber qué posibles q tenemos, podemos decir que:

$$Q = \{q \mid l_0 \leq i + q \leq 60\}$$

donde Q es el conjunto de todos los q posibles para ir desde el vértice $v_{1,i}$ a los vértices de la ciudad 2. No hay valor q' válido que no exista en Q ya que de existir, no cumpliría que $q' + i \leq l_0$ y por tanto no tendría suficiente nafta para recorrer la ruta o $q' + i \geq 60$ en cuyo caso no podría haber cargado esa cantidad q' en un primer lugar. Notar que la cantidad de q válidos está acotada por 61 ya que puedo cargar como mínimo 0 litros y como máximo 60.

Ahora bien, para saber qué aristas debemos definir en nuestro grafo G que modelen la existencia de la ruta entre las ciudades 1 y 2, consideremos los elementos de Q. Para cada valor q existente, la cantidad de nafta total que hay en el tanque antes de comenzar a recorrer la ruta es de $i + q$. Por lo tanto, para saber cuántos litros tengo al llegar a la ciudad 2 luego de recorrer dicha ruta, sólo queda restarle l_0, ya que ese era el costo de la ruta en litros de nafta. Como los vértices modelan tanto la ciudad como la cantidad de litros que tenemos una vez en la ciudad, las aristas que debemos definir son:

$$v_{1,i}\ v_{2,j} \textit{ donde } j \in \{i + q - l_0 \mid q \in Q\}^1$$

Lo único que resta para terminar la modelización de dicha ruta es ponerle el peso correspondiente a cada arista que definimos. Esto se trata simplemente de multiplicar la cantidad de litros cargada (q) por el costo de la nafta en la ciudad en la que lo cargué, en este caso c_1 ya que se trata de la ciudad 1. Por lo tanto el peso de la arista $v_{1,i}\ v_{2,j}$ es $q * c_1$ con el q correspondiente a cada determinado valor de j.

Si este mismo procedimiento lo llevamos a cabo para cada $v_{1,i}$ con $0 \leq i < 60$, tenemos todas las posibles maneras de llegar desde la ciudad 1 a la ciudad 2 mediante esa ruta, ya que podríamos llegar a la ciudad 1 desde alguna otra con una cierta cantidad i de litros de nafta en nuestro tanque.

Esto mismo puede generalizarse a toda ruta existente entre dos ciudades a y b cualesquiera, teniendo entonces una manera de definir todas las aristas necesarias para que G modele correctamente nuestro problema. De esta forma, para poder averiguar el costo mínimo necesario para llegar desde una ciudad i a cualquier otra, sabemos que "llegaremos" a la ciudad origen con una cantidad 0 de litros de nafta (ya que no venimos de ninguna otra ciudad) y por lo tanto nuestro punto de partida será desde el vértice $v_{i,0}$. Por la forma en que modelamos el grafo G, basta con aplicar un algoritmo de camino mínimo de uno a todos desde todos los vértices $v_{i,0}$ con $0 \leq i < n$. La razón por la que aplicar un algoritmo de camino mínimo nos otorga la respuesta correcta es que, a la hora de modelar el grafo, este fue diseñado de manera tal que contempla todas las posibles combinaciones de cantidades de nafta cargada en cada ciudad en cada camino de rutas existente en el mapa original. Esto es así ya que para cada ciudad consideramos todos los valores posibles válidos de cantidad de nafta a cargar en la misma, definiendo las aristas correspondientes para cada ruta existente. Así, en este caso vale el principio de optimalidad ya que para un camino mínimo todos sus subcaminos son óptimos. Consideremos un camino mínimo P del vértice $v_{i,j}$ a un vértice $v_{p,q}$. Tomemos un subcamino Q del mismo, de $v_{i,j}$ a $v_{a,b}$. Si existiese un camino Q' distinto de costo menor, entonces en mi camino original podría reemplazar Q con Q' siendo este nuevo camino P' válido ya que puedo llegar a la ciudad a con una cantidad b de litros y por lo tanto, puedo realizar el mismo camino de $v_{a,b}$ a $v_{p,q}$ que realizaba en P. Este nuevo camino P' tendría un costo menor que P y por ende P no sería un camino mínimo en un primer lugar. A continuación, presentamos un pseudo-código del algoritmo para analizar su complejidad.

```
// Inicialización de G' en lista de aristas
para i entre 0 y n-1                                          O(n)
  c_i ← input                                                 O(1)
para i entre 0 y m                          O(m)
  grafo G' ← arista i-ésima, ida y vuelta   O(1)
// Modelización en grafo G
grafo G en lista de adyacencias con 60 * n vértices           O(60 * n)
para cada arista (a,b) en G'                O(m)
  para cada i entre 1 y 60                  O(60)
    para cada j válido                      O(60)
      defino arista v_{a,i} v_{b,j} con peso q * c_a          O(1)
calculo camino mínimo de toda ciudad a a toda ciudad b        O(CM)
```

Para analizar la complejidad, se encuentran a la derecha de cada operación los costos que estas mismas implican. Podemos ver entonces que la inicialización del grafo utilizando los parámetros pasados como $input$ nos cuesta $O(n)$ para definir los costos c_i en una estructura auxiliar y $O(m)$ para definir todas las aristas (tomando como aristas no dirigidas a las rutas existentes entre ciudades y a los vértices como las ciudades). Por lo tanto, la inicialización tiene una complejidad total de $O(n+m)$. En cuanto a la modelización del input en nuestro grafo G, primero debemos inicializar la estructura de lista de adyacencias, que nos cuesta la cantidad de vértices a inicializar, en este caso $O(60*n) \in O(n)$. Luego, queda el proceso de agregar las aristas correspondientes. Vemos en el pseudo-código que para cada arista definida en G' ($O(m)$) realizamos 60 iteraciones que a su vez realizan a lo sumo 60 iteraciones (recordemos que la cantidad de j válidos era $O(60)$) costando cada una $O(1)$ amortizado ya que se trata de agregar al final del vector correspondiente al vértice $v_{a,i}$ el vecino $v_{b,j}$ con su costo q que se puede obtener mediante un simple cálculo (ver ecuación 1). Por lo tanto, el costo total es de $O(m*60*60) \in O(m)$. Por último, se debe realizar el algoritmo de camino mínimo de todos a todos, lo cual llamaremos CM. La complejidad total del algoritmo es de $O(n+m+m+CM) \in O(n+m+CM)$. Vemos entonces que el costo dependerá tanto de la cantidad de ciudades como de las rutas entre ellas como así también del algoritmo elegido para calcular caminos mínimos.

Para resolver el problema de calcular los caminos mínimos una vez terminada la modelización apropiada del problema, utilizamos tres algoritmos diferentes: Dijsktra (con dos implementaciones posibles, una para el caso ralo y otra para el caso denso), Bellman-Ford y Floyd.

2.2. Algoritmos de camino mínimo

Las primeras tres implementaciones de algoritmos presentados a continuación calculan el camino mínimo de un vértice a todo el resto. Es por esto que se debe ejecutar n veces para obtener las distancias entre todo par de ciudades. Notar que para saber la distancia desde una ciudad origen i se debe utilizar el vértice $v_{i,0}$ como origen, como ya mencionamos anteriormente. A su vez, las únicas distancias calculadas que nos interesan para nuestro resultado final son aquellas a los vértices $v_{j,0}$ con $0 \leq j < n, i \neq j$ para la ciudad j. Esto es así ya que no puede existir un valor menor a éste entre todos los vértices de la ciudad j $v_{j,l}$ porque esto implicaría que habría llegado con l litros de más, que en algún punto podría no haber comprado y por lo tanto haber gastado menos.

El esquema para calcular las distancias con estos algoritmos es el siguiente:

```
Inicializo diccionario D                                  O(n * n)
para cada ciudad i                                        O(n)
  Inicializo el diccionario de costos                       O(60 * n)
  Algoritmo que calcula costo mínimo que modifica costos    O(A)
  para cada otra ciudad j                                   O(n)
    D[i][j] ← costos[v_{j,0}]                                 O(1)
```

Si sumamos las complejidades, nos queda una complejidad total de $O(n^2+n+60*n+A+n) \in O(n^2+A)$ siendo A el costo de cada algoritmo particular. El costo de $O(n^2)$ tiene sentido ya que es lo mínimo necesario para definir cada costo buscado.

Por otra parte, vale notar que a partir de este momento vamos a estar hablando de algoritmos que toman como input a nuestro grafo G modelado y por lo tanto sería lógico referirnos a las complejidades utilizando valores de n' y m' como cantidad de vértices y aristas del grafo G respectivamente. Sin embargo, como $n' = 60 * n$ y $m' \leq 60 * 60 * m$ podemos decir que $n' \in O(n)$ y $m' \in O(m)$. Es decir, estaremos utilizando tanto n como m en vez de n' y m' para referirnos a complejidades que dependan de la cantidad de vértices y aristas. Esto nos ahorrará una innecesaria conversión de n' y m' a n y m a la hora de calcular las complejidades totales.

2.2.1. Dijkstra caso ralo

Este algoritmo calcula caminos mínimos de un vértice a todo el resto, por lo que se debe ejecutar n veces para obtener las distancias de todos a todos. Notar que para saber la distancia desde una ciudad origen i se debe utilizar el vértice $v_{i,0}$, como ya mencionamos anteriormente. Para más información del algoritmo y su demostración, ver [1].

A continuación adjuntamos el pseudo-código de esta implementación de Dijkstra con cola de prioridad (que tiene mejor complejidad que la otra implementación en el caso ralo). El algoritmo recibe el grafo G y el vértice v a partir del cual se debe calcular los costos mínimos y un diccionario de costos el cual se modifica y como precondición comienza con valores en indefinido.

```
Dijkstra (grafo G, vértice v, diccionario costos)
  Inicio una cola de prioridad Q                                   O(1)
  costos[v] ← 0                                                    O(1)
  agrego vecinos de v a Q con costos de G                          O(d(v))
  mientras Q no esté vacía                                         O(m)
    saco el mínimo m de Q                                            O(log(m))
    si vértice(m) no está definido en costos                         O(1)
      costos[vértice(m)] ← costo(m)                                    O(1)
      agrego vecinos de vértice(m) a Q con los nuevos costos           O(d(vértice(m)))
```

Como podemos ver, el costo total es de $O((m + n) * log\ n)$ ya que realizo $O(m)$ veces la operación de sacar el mínimo y agrego vecinos $O(n)$ veces, una por cada vértice agregado al árbol que construye Dijkstra (todos los vértices son agregados sólo una vez), por lo que cuesta $O(n * agregar\ a\ la\ cola) \in O(n * log\ m) \in O(n * log\ n)$.

En el caso ralo, es decir, cuando $m \in O(n)$, la complejidad del algoritmo es $O(n * log\ n)$.

La complejidad total de nuestro algoritmo utilizando esta implementación de algoritmo de camino mínimo es $O(n + m + (n + m)log\ n) \in O((n + m)log\ n)$.

2.2.2. Dijkstra caso denso

Esta implementación de Dijkstra utiliza un diccionario de costos en vez de una cola de prioridad, lo cual implica que la búsqueda del mínimo costo cueste $O(n)$. El pseudo-código de esta implementación de Dijkstra es el siguiente:

```
DijkstraDenso (grafo G, vértice v, diccionario costos)
  n ← tamaño de G                                                          O(1)
  costosActuales ←diccionario de tamaño n inicializado con valores infinitos   O(n)
  costosActuales[v] ← 0                                                    O(1)
  costosActuales ←defino a los vecinos j de v con el costo de vj           O(n)
  para todo i entre 1 y n                                                  O(n)
      w ←arista con mínimo costo definido en costosActuales no definidas en costos   O(n)
      costosActuales ←defino a los vecinos j de w con el costo de wj                 O(d(w))
      costos[w] ←costosActuales[w]                                                   O(1)
```

El algoritmo realiza operaciones que cuestan $O(n)$ para definir valores iniciales y luego $O(n)$ iteraciones para definir el árbol de caminos mínimos en las que busca el mínimo costo alcanzable desde el árbol -definidos en costosActuales- y define los nuevos costos de los vecinos del vértice que se agrega, lo que

cuesta $O(n)$. Es decir, esto da una complejidad de $O(n^2)$, lo cual mejora la complejidad de la anterior implementación en el caso denso cuando $m \in \theta(n^2)$.

La complejidad total de nuestro algoritmo utilizando esta implementación de algoritmo de camino mínimo es $O(n + m + n^2) \in O(n^2 + m) \in O(n^2)$.

2.2.3. Bellman Ford

Para ampliar información sobre el algoritmo y su demostración, ver $página\ 651$ de [1]. A continuación presentamos un pseudo-código del algoritmo utilizado para poder analizar su complejidad, este recibe el mismo input que Dijkstra con la excepcion de que los valores definidos en el diccionario de costos se inicializan en infinito.

```
BellmanFord(grafo G, vértice v, diccionario costos):
  edges ← lista de aristas del grafo G que es una lista de adyacencias      O(n + m)
  costos[v] = 0                                                             O(1)
  modificado ←verdadero                                                     O(1)
  para i = 1, . . . , n  1 y modificado es verdadero                        O(n)
      modificado ←falso                                                     O(1)
      para toda arista (x,y,c) en edges:                                    O(m)
          si costos[x] + c < costos[y]                                      O(1)
              modificado ←verdadero                                         O(1)
              costos[y] ←costos[x] + c                                      O(1)
  return costos
```

Como podemos ver en las complejidades comentadas en cada línea de pseudo-código, la complejidad total es la suma de la complejidad de transformas a lista de aristas ($O(n + m)$) y la complejidad de las $O(n)$ iteraciones de costo $O(m)$. Es decir, $O(n + m + n * m) \in O(n * m)$.

2.2.4. Floyd

Este algoritmo, a diferencia de los anteriores, calcula los caminos mínimos de todos a todos. Es decir, no es necesario ejecutarlo n veces. Por esta razón, el $input$ que toma es distinto: en vez de un diccionario de costos de $O(n)$, recibe un diccionario de $O(n^2)$ ya que es en este en donde define los costos de todos los vértices a todos los vértices. El mismo está inicializado con todos sus valores en infinito. Presentamos el pseudo-código a continuación; para mayor detalle y una demostración apropiada ver [1].

```
Floyd(grafo G, diccionario D)
  n ← tamaño de G                                  O(1)
  para todo vértice i:                             O(n)
      D[i][i] ←0                                   O(1)
      para todo vecino j de i:                     O(d(i))
          D[i][j] ← costo de arista ij en G        O(1)
  para k, i, j entre 0 y n:                        O(n^3)
      si D[i][k] + D[k][j] < D[i][j]               O(1)
          D[i][j] ←D[i][k] + D[k][j]               O(1)
  return D
```

En cuanto a la complejidad, tenemos $O(n)$ iteraciones que realizan operaciones de costo $O(d(i))$, lo cual implica una complejidad de $O(n + m)$, y luego un triple for anidado de $O(n)$ iteraciones cada uno, con un costo de $O(1)$ cada una de ellas, lo cual implica un costo final de $O(n^3)$. Por lo tanto la complejidad es de $O(n + m + n^3) \in O(n^3)$.

La complejidad total de nuestro algoritmo utilizando esta implementación de algoritmo de camino mínimo es $O(n + m + n^3) \in O(n^3)$.

3. Experimentación

3.1. Metodología

Para llevar a cabo la experimentación de las distintas hipótesis ejecutamos los respectivos algoritmos en grafos generados automáticamente. Para esto hicimos uso de la herramienta `erdos_renyi_graph(n, p)`, de la biblioteca `networkx` de python, que crea un grafo de Erdös-Rényi $G_{n,p}$, en el que n corresponde a la cantidad de vértices en el grafo generado, y p corresponde a la probabilidad de que una arista exista [2]. Nótese que la densidad del grafo va a depender de forma directa de p. De esta forma fuimos generando grafos conexos para n de 10 a 100, incrementando de a 10, y para p igual a 0.2, 0.4 y 0.8. A su vez, los costos de cada arista fueron definidos generando números entre 1 y 60 de forma aleatoria. Véase que ninguna arista puede tener costo mayor a 60, porque modelaría una ruta que no puede ser transitada de forma completa. Por último, definimos el costo de la nafta en cada ciudad generando de forma aleatoria números entre 1 y 200. Este límite fue elegido de forma arbitraria, teniendo en cuenta que un límite muy bajo causaría muy poca variación entre los costos de las ciudades, y un límite muy alto crearía situaciones demasiado irreales.

Con estos grafos como entrada, ejecutamos cada algoritmo 30 veces (para cada grafo) y medimos el tiempo que tardó cada uno en promedio (sin contar el tiempo usado en cargar los datos). Escogimos esta metodología con el fin de minimizar el ruido que se podría generar por cuestiones de scheduling del procesador, entre otros factores.

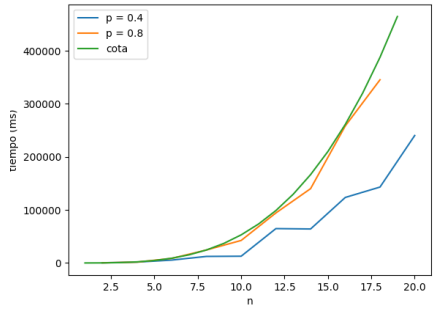
Todas las experimentaciones fueron realizadas en los equipos de los laboratorios del Departamento de Computación, siguendo la metodología descrita arriba, excepto en los casos aclarados debajo, donde el costo temporal hacía imposible la finalización de algunas ejecuciones.

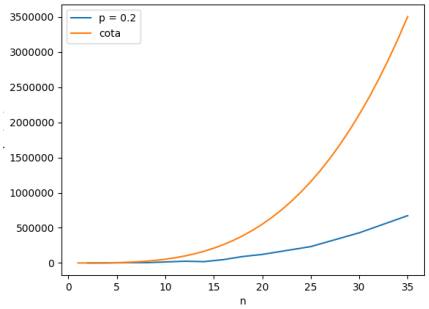
3.2. Complejidad de Dijkstra Caso Ralo

La primer hipótesis a corroborar es aquella que refiere a la complejidad temporal del algortimo *Dijkstra Caso Ralo* en caso peor. En secciones anteriores, explicamos que la misma es $O((n + m) * log(n))$. Como el peor caso en *Dijkstra de uno a todos* depende del orden en el que se encuentran los costos a partir del vértice inicial, no lo podemos generalizar para *Dijkstra de todos a todos*, es decir, al iterar n veces el primero, porque sería necesario encontrar un grafo tal que partiendo de cualquier vértice respete este orden. Tal grafo sería muy difícil de diseñar, y posiblemente de implementar. Por esto, alcanza con experimentar sobre grafos aleatorios conexos cualequiera.

Para corroborar nuestra hipótesis procedimos según la metodología recién descripta, pero con n de 2 a 20, con incrementos de 2. Luego, también con $n = 25$, 30 y 35 y $p = 0.2$. Estos valores fueron decididos teniendo en cuenta los altos costos temporales de este algoritmo, que habrían imposibilitado la experimentación en el tiempo disponible.

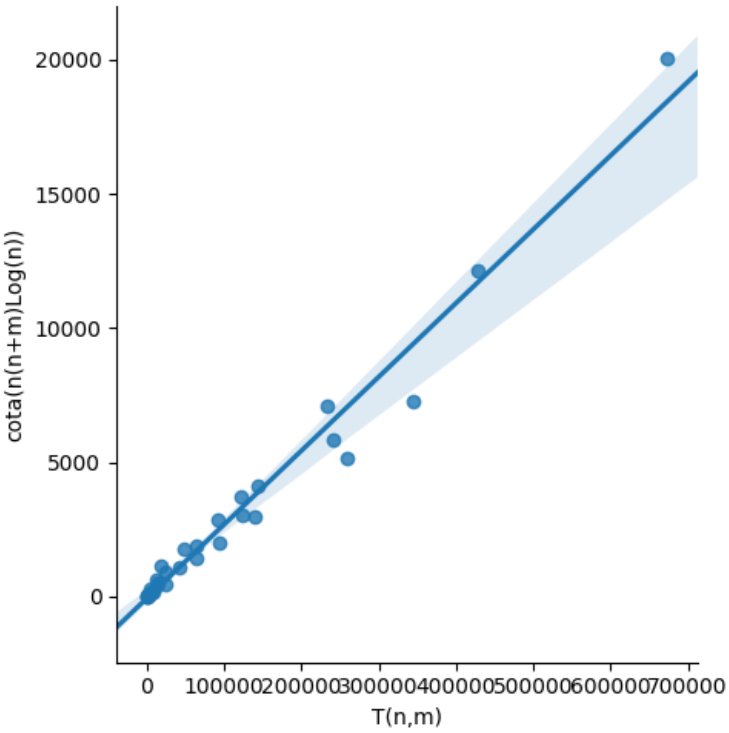
A continuación exponemos dos gráficos, en los que se puede ver la relación entre los tiempos de ejecución y el tiempo esperado, acotando m por n^2, de forma que el mismo quede sólo en función de n. Vale decir que también, para esta cota, se multiplicó por un coeficiente para ajustar la escala. Así, en el gráfico de la izquierda, $cota = n^2 * log(n) * 210$, y en el de la derecha, $cota = n^3 * log(n) * 23$.





Se puede ver cómo el crecimiento del tiempo de ejecución crece con la misma rapidez que el tiempo esperado. Para analizar esto de forma más rigurosa, confeccionamos un gráfico de correlación lineal entre los tiempos de ejecución $T(n, m)$ para cada grafo y el tiempo esperado $cota(n, m) = n * (n + m) *$

$log(n)$. Si nuestra hipótesis es correcta, luego estos tiempos van a presentar una fuerte correlación lineal, acompañado de un coeficiente de Pearson alto. El resultado fue el siguiente:



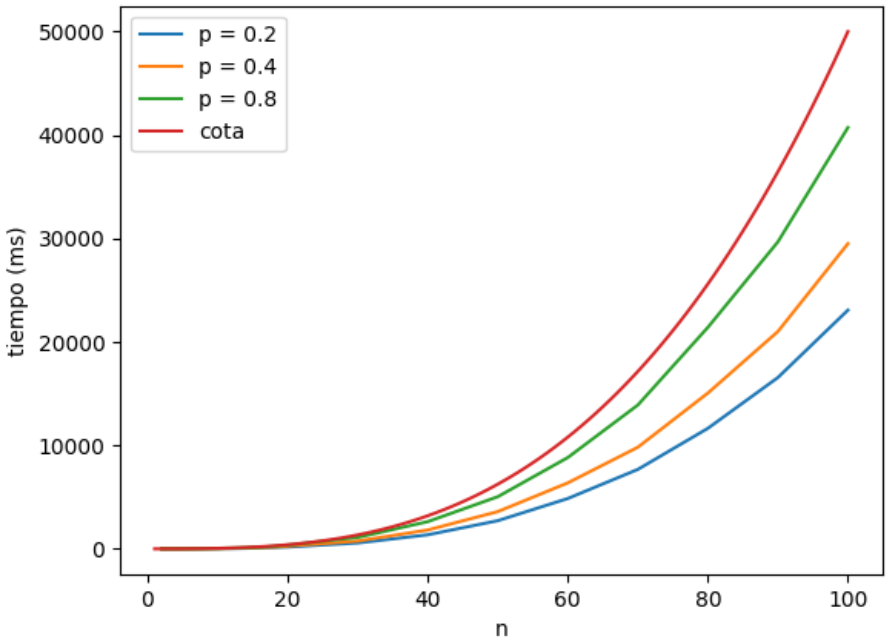
Este gráfico, junto con un coeficiente de Pearson igual a 0.9866151207138689, evidencia una correlación lineal fuerte, según lo esperado. De todas formas, si estos resultados no son suficientemente cercanos a los esperados, vale recordar que un ajuste de constantes podría mejorarlos aún más. Este ajuste podría ser necesario dado que el grafo que recibe el algoritmo de Dijkstra no es exactamente igual al grafo original, sino que tiene más vértices y más aristas, luego de pasar por el modelado.

3.3. Complejidad de Dijkstra caso Denso

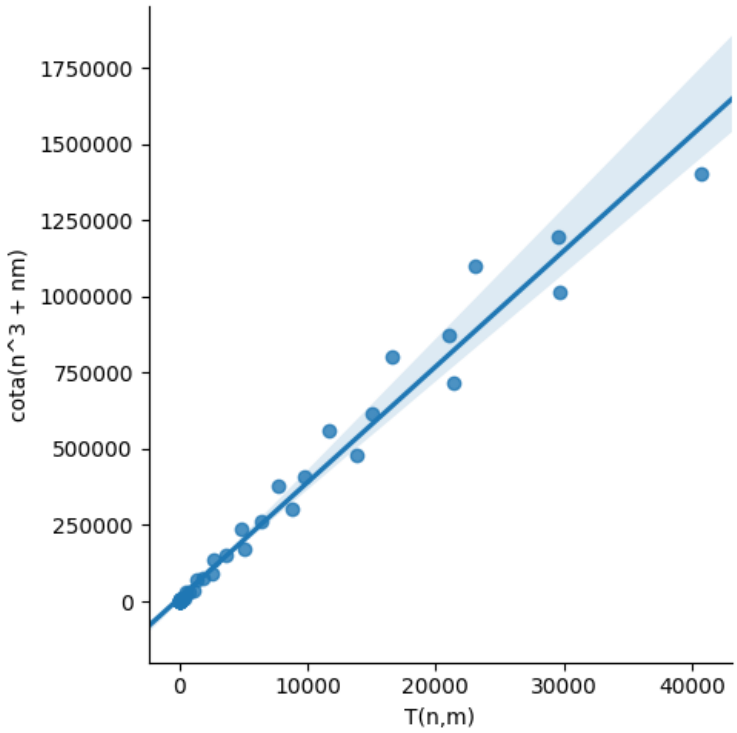
Con anterioridad mostramos que la complejidad temporal de *Dijkstra Caso Denso* es $O(n^2+m)$ en caso peor. Como lo ejecutamos n veces, la complejidad del algoritmo sería $O(n*(n^2+m)) \in O(n^3+n*m)$. Siguiendo un razonamiento similar al expresado en la anterior subsección, podemos concluir que basta con experimentar sobre grafos conexos cualesquiera para corroborar esta hipótesis.

Para esto realizamos experimentaciones según lo descrito en la subsección *Metodología*. Luego, para hacer más exhaustiva las mismas, y gracias a los bajos costos temporales de este algoritmo, seguimos el mismo procedimiento para n de 2 a 20, incrementando de a 2.

A continuación se exhiben los resultados de la susodicha experimentación. En el mismo, $cota = n^3 * 5e^-2$. Nótese que se omite m, dado que $O(n^2+m) = O(n^2)$, y para que quede en función de n. Asimismo, se la multiplica por un coeficiente para ajustar la escala. Si nuestra hipótesis es correcta, esperamos ver que los tiempos de ejecución crecen de forma similar a los tiempos esperados.



Se puede ver claramente que los tiempos de ejecución siguen la forma del tiempo esperado. Para poder corroborar de forma certera nuestra hipótesis, veamos que efectivamente hay una correlación lineal entre los tiempos de ejecución $T(n, m)$ para cada algoritmo y la cota de complejidad $cota(n, m) = n^3 + n * m$.



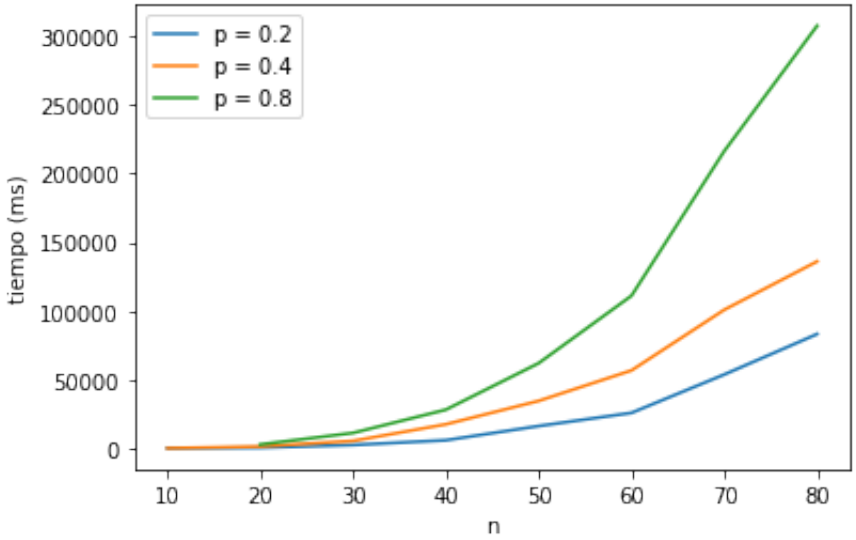
Este gráfico y un coeficiente de Pearson igual a 0.9880138800385881 parecen corroborar que la cota es apropiada, si bien no del todo ajustada.

3.4. Complejidad de Bellman Ford

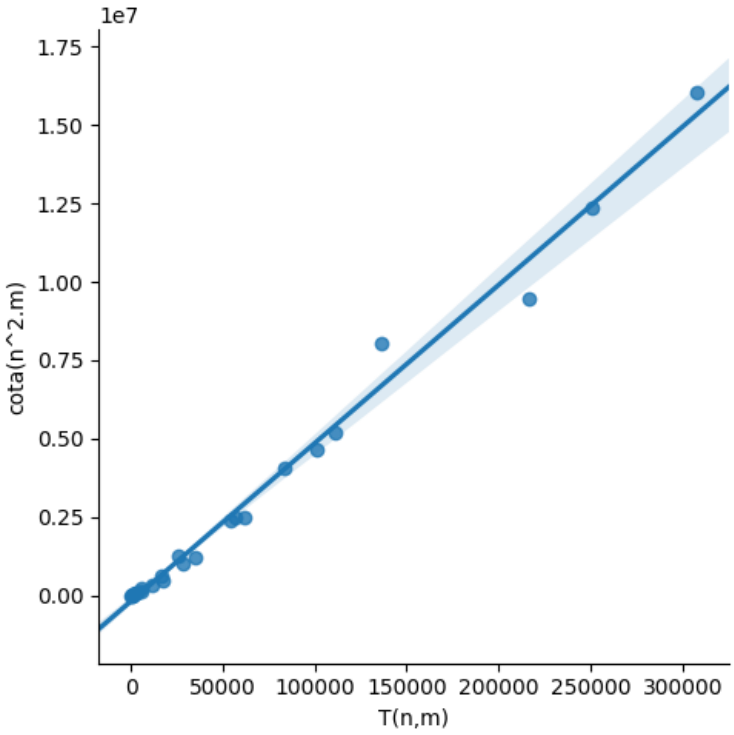
Para el algoritmo *Bellman Ford*, planteamos como hipótesis que su complejidad temporal en peor caso es $O(nm)$. Si analizamos cuál es el peor caso, podemos concluir que para *Bellman Ford de uno a todos* es

aquel en el que el peso de las aristas está ordenado de forma descendiente a partir del vértice inicial. Sin embargo, cuando iteramos este algoritmo desde los distintos vértices para obtener el camino mínimo de todos a todos, esa cualidad se pierde. Para esto, podemos ver a modo de ejemplo que el mismo camino ordenado que en un sentido cuesta k iteraciones (con k igual a la longitud del camino), en el sentido contrario cuesta 1 iteración (es el mejor caso). De esta forma, alcanza con experimentar sobre grafos conexos aleatorios cualesquiera para corroborar nuestra hipótesis.

A continuación se exponen los resultados de la experimentación, clasificados según el parámetro p utilizado para generar los grafos, directamente relacionado con la densidad de los mismos. La misma se hizo siguiendo la metodología detallada anteriormente, pero con n de 1 a 80, debido al elevado costo temporal de las ejecuciones con más vértices. Según nuestro análisis, esperamos que a medida que n y p crecen, también lo haga el tiempo de ejecución.



Este gráfico muestra que a mayor cantidad de vértices y a mayor densidad del grafo, mayor es el tiempo de ejecución. Veamos ahora con más rigurosidad cuál es el orden de esta relación. Para ello, analicemos si efectivamente hay una correlación lineal entre el tiempo de ejecución para cada grafo, y su cota $n * m$.

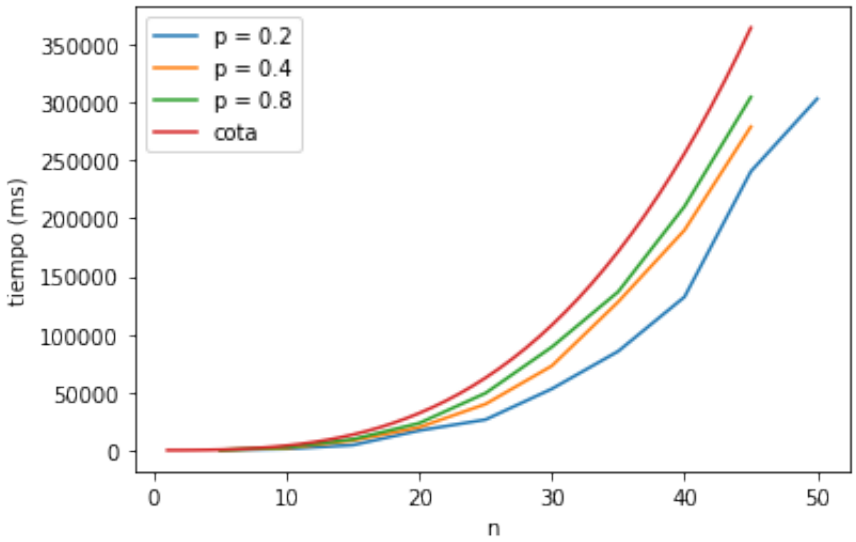


La correlación lineal esperada se puede observar en el gráfico, y es corroborada por un coeficiente de Pearson igual a 0.9945075378130618. Así, la experimentación confirma nuestra hipótesis.

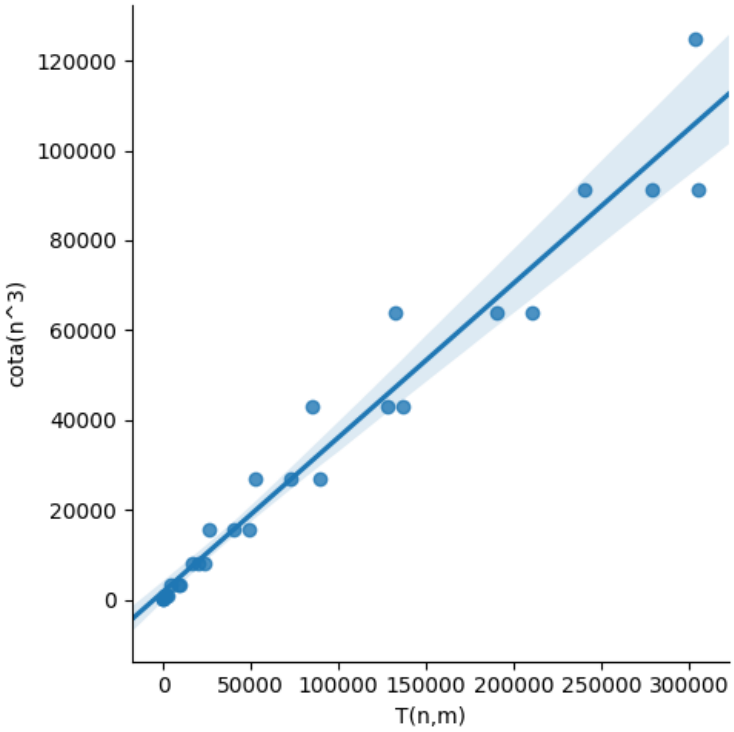
3.5. Complejidad de Floyd

Como mostramos en secciones anteriores, el análisis de los algoritmos arroja como resultado teórico que la complejidad temporal de este algoritmo en peor caso es $O(n^3)$. Observando este algoritmo con detenimiento, es claro que, dados un n y un m, no existe un peor caso en particular, o, lo que es lo mismo, todos los grafos (de n vértices y m aristas) realizan la misma cantidad de pasos. Por tanto, para corroborar que la complejidad temporal es como dedujimos, alcanza con experimentar sobre grafos aleatorios conexos cualesquiera.

Siguiendo la metodología descripta, experimentamos con n de 5 a 50, con incrementos de a 5. A continuación se exhiben los gráficos de los tiempos de ejecución junto al gráfico del tiempo esperado (multiplicado por un coeficiente para ajustar la escala) $cota = 4 * n^3$. Esperamos ver que los tiempos medidos crecen a la par de los tiempos esperados.



Es claro en el gráfico cómo los tiempos de ejecución efectivamente crecen igual de rápido que el tiempo esperado. Para terminar de corroborar esta hipótesis, veamos que hay una correlación lineal entre ambos.



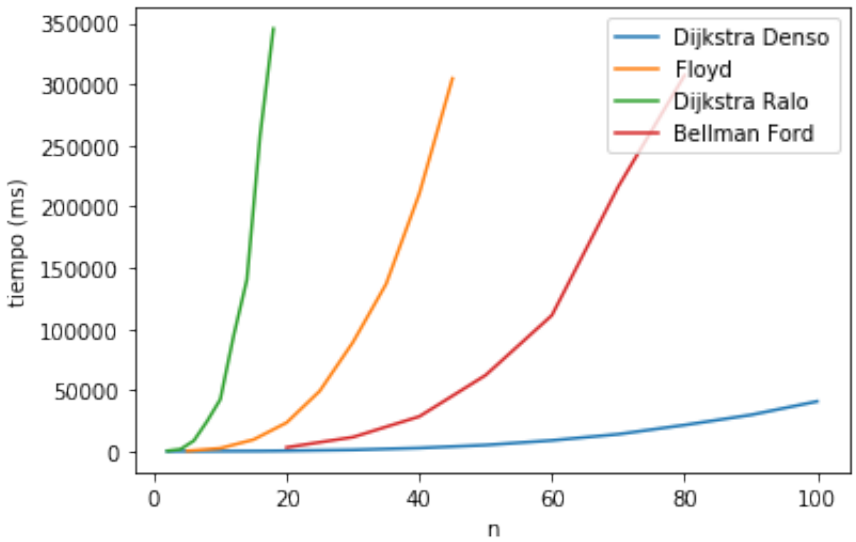
Este gráfico, junto con un coeficiente de Pearson igual a 0.9790472583236288, confirman que se trata de una cota correcta, si bien no ajustada. Si se observa con detenimiento, se puede distinguir que, para distintas alturas, hay tres puntos alineados horizontalmente. Esto se explica recordando que, como se explicó en la metodología, para cada n experimentamos con 3 valores distintos de m. Así, podemos inferir que la diferencia con los resultados esperados se da porque según la cota, las tres instancias deberían costar lo mismo, pero en la práctica los tiempos de ejecución dependen de m. Esto puede parecer contradictorio con el análisis temporal del algoritmo, pero el conflicto se resuelve al considerar que, en nuestras mediciones, contamos el tiempo de modelización dentro del tiempo de ejecución, y el modelización sí depende de m.

3.6. Comparación

Ya hemos analizado cuál es la complejidad temporal de cada algoritmo por separado, en términos asintóticos. Lo que nos resulta de interés ahora es comparar los tiempos de ejecución de cada uno, para los valores de n y m que manejamos.

A partir del análisis asintótico, podríamos inferir que *Dijkstra Denso* y *Floyd*, con complejidad $O(n^3)$, presentan los menores tiempos; seguidos por *Dijkstra Ralo*, con complejidad $O(n * (n + m) * log(n))$; y por último *Bellman Ford*, con complejidad $O(n^2 * m)$.

Sin embargo, podemos advertir que a la hora de mirar los tiempos de ejecución vamos a encontrar algunas diferencias con este análisis. En primer lugar, es pertinente considerar que *Floyd* no recorre los n vértices del grafo original, sino los $60 * n$ vértices del modelo. De esta forma, la constante que desaparece en el análisis asintótico es al menos $60^3 = 216000$, nada pequeña. Por esto, podemos esperar que este algoritmo no presente el buen rendimiento que podríamos inferir.



Estos resultados nos dicen muchas cosas: Por un lado, según lo esperado, *Dijkstra Denso* presenta, y por mucho, los menores tiempos de ejecución. Por otro lado, según esperábamos, *Floyd* presenta tiempos no muy buenos. Con un poco de sorpresa, vemos que *Dijkstra Ralo* es aún peor que *Floyd*. Asumimos que, a pesar de que este último tenga muy elevadas constantes, la mala complejidad de *Dijkstra Ralo* pesa más. Por último, así como esperábamos, *Bellman Ford* presenta un peor desempeño que *Dijkstra Denso*.

4. Discusión y conclusiones

Hemos analizado el problema planteado, presentamos soluciones, vimos que eran correctas y analizamos sus complejidades, teórica y experimentalmente. De esta forma, nos llevamos algunas conclusiones y algunas preguntas a responder en futuros trabajos.

En primer lugar, vimos que el algoritmo *Dijkstra Denso* presenta los menores tiempos de ejecución, mientras que *Dijkstra Ralo* presenta los peores.

Por otro lado, si bien *Floyd* tenía una muy buena complejidad teórica, vimos que en la práctica no presentó un buen desempeño. Esto nos ayuda a entender el alcance y las limitaciones del análisis asintótico. Además, aparece como desafío buscar la forma de implementarlo haciendo que recorra sólo los vértices originales, ahorrando todo el trabajo innecesario que ejecuta la implementación actual.

Además, vemos como una opción interesante repetir la experimentación excluyendo el tiempo de modelado de la medición. Así, podríamos obtener resultados que sólo tengan en cuenta la parte del algoritmo en la que difieren las distintas soluciones, dado que las cuatro realizan el modelado de la misma manera. Aparte, suponemos que algunas experimentaciones darían conclusiones más certeras (veáse el anális del gráfico de correlación lineal de Floyd).

Por último, nos preguntamos si utilizar instancias reales para experimentar, en vez de generar instancias aleatorias, podría darnos resultados más apropiados para el contexto de uso. En caso de ser complicado obtener una amplia cantidad de instancias reales, se podría basar los parámetros utilizados para generar los casos aleatorios en los valores que tomarían en las situaciones de uso real. Por ejemplo: ¿cuántas ciudades se suelen tener que recorrer?,¿cuánto suele salir la nafta?,¿cuánto varía el costo de la nafta entre ciudades?

Referencias

[1] *Introduction to Algorithms* by T. Cormen, C. Leiserson, R. Rivest, and C. Stein. The MIT Press, 2nd edition, (2001)

[2] *On Random Graphs. I.* by Erdős, P.; Rényi, A. (1959). Publicationes Mathematicae. 6: 290–297.