Institución : Universidad Nacional Autónoma de México
Sede: Instituto de Investigaciones en Matemáticas Aplicadas
y en Sistemas
Curso : Matemáticas Discretas
Fecha : 3 de diciembre de 2021
Autor : LCD 56
Mail : alumnolcd56@gmail.com

Reporte ejecutivo

1. Planteamiento del problema a resolver

El Sistema de Transporte Colectivo Metro es posiblemente el principal medio de transporte público utilizado en la Ciudad de México. Por lo tanto, es uno de los lugares más potenciales donde se puede sufrir un contagio del virus $SARS-COV-2$, (COVID-19), debido a que es una enfermedad que se propaga por vías respiratorias, (es decir, nariz y boca). Más aún, dicho virus se propaga con mayor facilidad en espacios interiores o en aglomeraciones de personas.

Al ser esta una enfermedad relativamente nueva, los países no cuentan con suficiente información para el tratamiento de la misma, por lo que las principales recomendaciones son permanecer en casa, mantener una distancia de aproximadamente 16 pies entre cada par de personas, utilizar cubre bocas, así como evitar aglomeraciones.

Pese a que el índice de muertes en México se ha visto reducido y hasta controlado, no lo es así el número de casos nuevos registrados de la enfermedad. Es por ello que debería seguir siendo una prioridad, si no el mantenerse en casa, sí el evitar aglomeraciones y por supuesto, seguir las medidas sanitarias como el lavado de manos y uso del cubrebocas.

Con el fin de ofrecer rutas con el menor índice de afluencia, y por tanto de aglomeración de personas, el objetivo del presente proyecto es hallar la ruta, dadas una estación inicial, una estación final y un día de la semana, que sea menos transitada. El fin de esto es intentar minimizar las aglomeraciones sufridas dentro de la red del Sistema de Transporte Colectivo Metro, evitando así posibles contagios.

Otra de las ventajas de las rutas con menor afluencia es que, como se sabe, las estaciones más concurridas suelen significar un traslado más lento de un lugar a otro, por lo que encontrar una ruta con menos afluencia de personas no sólo disminuye el contagio, sino también puede ser clave para disminuir el tiempo de traslado.

Luego entonces queda definido el objetivo del presente proyecto; implementar un programa que ayude a los usuarios a obtener la ruta con menos tránsito de personas, de manera que no deban

comprometer su salud ni su tiempo para trasladarse de un lugar a otro a consecuencia de la cantidad de gente que utiliza el servicio del Sistema de Transporte Colectivo Metro diariamente.

2. Forma de modelarlo matemáticamente y justificación de suposiciones

Se utilzaron dos bases de datos, la primera que contenía los registros de la afluencia diaria por estación y la segunda contenía la ubicación (longitud y latitud) de cada una de las estaciones de la red del metro de la Ciudad de México.

La primera de ellas se utilizó para la aplicación del modelo y resolución del problema, mientras que la segunda base de datos se utilizó para poder ver de manera gráfica los resultados y la red en general del metro.

Por lo que la forma en que se resolvió el problema fue a través de la aplicación de un algoritmo para encontrar la ruta más corta entre dos estaciones, en este caso entendemos por ruta más corta como el camino de la estación inicial a la estación final con menos afluencia.

Dado lo anterior se calcularon las afluencias promedio por día en cada una de las estaciones del metro de la Ciudad de México. Después de obtenerlas se construyó la matriz de adyacencia con pesos, en donde cada peso corresponde a la afluencia que hay entre dos estaciones, en este caso se tomó como dicho peso la afluencia promedio del nodo destino, es decir para la entrada i, j el valor de dicha entrada será 0 si no hay forma de llegar del nodo i al j de manera directa y será la afluencia del nodo j si podemos pasar de i a j de manera directa. Además estamos considerando que no es lo mismo ir de i a j que ir de j a i, por lo que la entrada j, i de la matriz de adyacencia será la afluencia del nodo i.

Una vez construida la matriz de adyacencia con pesos se creó una gráfica dirigida que tiene como nodos a cada una de las estaciones del metro. Y donde cada arista corresponde a la conexión de la estación i con la estación j y donde el peso de cada arista corresponde a la afluencia del nodo en el cuál incide la arista. Una observación importante es que si el nodo i está conectado con el nodo j entonces el nodo j estará conectado con el nodo i. Por lo tanto nuestro grafo contiene 163 nodos y un total de 364 aristas.

Ahora bien, los supuestos que se tomaron fueron los siguientes:

- Los registros utilizados corresponden al periodo de tiempo de $01/01/2021 - 30/09/2021$.
- Se utilizó un grafo dirigido pues la cantidad que viaja de una estación a otra depende del sentido de la línea, es decir no es la misma cantidad de gente que va de Copilco a Universidad

que vicersa, por ejemplo.

- El peso de la arista se tomó como la afluencia del nodo destino, esto se tomó para facilitar la modelación de los datos del problema.

- Se tomaron las afluencias promedio por día de la semana pues la cantidad de personas que toman el metro en un día laboral es mayor que la cantidad de personas que lo utilizan en fines de semana.

3. Enunciado del problema algorítmico formal general a resolver

Sea $G = (V, E)$ una gráfica dirigida ponderada con V el conjunto de vértices y E el conjunto de aristas. Sean u, v dos nodos de G, correspondientes a una estación inicial y otra final y un día de la semana, se debe encontrar el camino menos afluente de u a v bajo el algoritmo de Dijkstra.

4. Propuesta de solución mediante algoritmos combinatorios

Recordando que un algoritmo combinatorio busca lograr su objetivo a través de la reducción del tamaño efectivo del espacio, se observa que el Algoritmo de Dijkstra ciertamente reduce el espacio.

Recordemos el algoritmo de Dijkstra;

Sea G un grafo dirigido ponderado de N nodos no aislados y sea x el nodo inicial. Definiremos un vector D de tamaño N que guardará la distancia entre x hacia el resto de nodos. Además, sea V un vector de N entradas que funcionará para llevar control sobre los nodos ya visitados.

- Inicializamos las distancias de D con un valor relativamente grande (infinito, teóricamente) salvo la entrada correspondiente a x, misma que será 0.
- Sea $a = x$, donde a denota el vértice actual.
- Se recorren los nodos adyacentes de a, salvo los nodos ya visitados. Denotemos a los nodos no visitados como v_i.
- Para el nodo actual, se calcula la distancia desde ese nodo hasta sus vecinos bajo la expresión $d(v_i) = D_a + d(a, v_i)$. Esto es, la distancia tentativa del nodo v_i es la distancia que actualmente tiene el nodo en el vector D más la distancia desde el nodo actual hasta el nodo v_i. Si la distancia tentativa es menor que la distancia almacenada en el vector, entonces se actualiza el vector con esa distancia tentativa.
- Marcamos como visitado al nodo a.
- Se toma como próximo nodo actual el de menor valor en D y se vuelve al tercer paso mientras haya nodos no visitados.

Entonces, ciertamente este algoritmo reduce el espacio, pues, al hacer la comparación de la distancia tentativa de un nodo contra la distancia ya guardada a ese mismo nodo, se está evitando seguir con una exploración innecesaria. Similarmente, el que los nodos tengan una característica de visitado tiene como consecuencia que no se exploren los nodos más que las veces necesarias.

5. Análisis de correctitud y análisis asintótico de tiempo y espacio

Antes de proceder con los análisis de correctitud, tiempo y espacio debemos mostrar el código de la implementación usada para el algoritmo de Dijkstra;

```
def dijkstra(G,ni,nf,d=my_dict):
  u=d[ni]
  v=d[nf]
  masinf=float('inf')
  vertices=list(G.nodes)
  distancias={w:masinf for w in vertices}
  fijos={w:False for w in vertices}
  padres={w:None for w in vertices}
  distancias[u]=0
  fijos[u]=True
  nuevo_fijo=u

  while not(all(fijos.values())):
      # Acualizar distancias.
      for w in G.neighbors(nuevo_fijo):
          if fijos[w]==False:
              nueva_dist=distancias[nuevo_fijo]+G[nuevo_fijo][w]['weight']
              if distancias[w]>nueva_dist:
                  distancias[w]=nueva_dist
                  padres[w]=nuevo_fijo

      # Encontrar el nuevo a fijar.
      mas_chica=masinf
      for w in vertices:
          if fijos[w]==False and distancias[w]<mas_chica:
              optimo=w
              mas_chica=distancias[w]
      nuevo_fijo=optimo
      fijos[nuevo_fijo]=True

      # Cuando fije el vértice final v, dar el camino.
      if nuevo_fijo==v:
          camino=[v]
          while camino[0]!=u:
              camino=[padres[camino[0]]]+camino
          #Obtenemos los nombres de los nodos
          key_list = list(d.keys())
          val_list = list(d.values())
          cam=[]
          for c in camino:
            position = val_list.index(c)
            cam.append(key_list[position])
          return cam
```

5.1. Correctitud

Para justificar la correctitud del algoritmo de Dijkstra empleado, se deben probar que al finalizar la ejecución del algoritmo se debe tener que, para todo $u \in S$ con $u \neq a$

- Cualquier camino $a - u$ tiene peso al menos $L(u) : L(u) \leq p(a - u)$
- Existe un camino de a a u de peso igual a $L(u)$. En particular, $L(u)$ es finita : $L(u) < \infty$.

Del teorema anterior se sigue que:

- El algoritmo termina

  En efecto, probemos por inducción sobre $i \geq 1$ que los elementos de S_i son distintos.

  Para $i = 1$, $S_1 = \{u_0\}$ y en este caso no hay nada que demostrar.

  Supongamos que todos los elementos de S_i son distintos para algún $i \geq 1$, de modo que $z \notin S_i$. Sea x el último vértice en $S_i \subseteq S$ de un camino $u - z$ y y su vecino. Cuando x entró a S, la etiqueta de y se actualizó, luego $L(y) \leq L(x) < \infty$. Por lo tanto, es posible escoger en $T_i = V - S_i$ un elemento con etiqueta mínima finita $\leq L(y)$. Uno de ellos es precisamente el elemento u_i. Entonces definimos $S_{i+1} = \{u_0, u_1, \ldots, u_{i-1}, u_i\}$. Es claro que todos los elementos de S_{i+1} son distintos. Por consiguiente, $i \leq n$ y el algoritmo debe terminar.

- $L(u)$ es el peso de un camino de coste mínimo de a a u para todo $u \in S - \{a\}$.

  En efecto, puede haber más de un camino de coste mínimo. Sin embargo, es fácil ver que todos ellos tienen el mismo peso. Si consideramos uno de estos caminos, su peso q es menor o igual que el peso del camino de a a u cuya existencia se garantiza en la parte B del teorema, esto es, $q \leq L(u)$. De otro lado, según la primera parte de este teorema, todo camino $a - u$ tiene peso al menos $L(u)$, esto es, $q \geq L(u)$. Concluimos que $q = L(u)$. En particular, tomando $u = z \in S$, obtenemos que $L(z)$ es el peso de un camino de coste mínimo de a hacia z.

- Si $u_m = z$, entonces

$$L(u_0) \leq L(u_1) \leq \ldots \leq L(u_j) \leq L(u_{j+1}) \leq \ldots \leq L(u_m)$$

  .

  En efecto, antes de que los elementos u_j y u_{j+1} entren a $S = S_m$, estos se encuentran en $T_j = V - S_j$. Se escoge uj por cuanto este es uno de los elementos con etiqueta mínima, luego $L(u_j) \leq L(u_{j+1})$

Ahora bien, respecto al algoritmo de Dijkstra modificado. En algunas ocasiones nos interesa conocer no solamente el peso de un camino de coste mínimo entre dos vértices dados, sino también la ruta a seguir para llegar de uno de estos dos vértices al otro. En este caso a cada vértice distinto de a le agregamos una etiqueta adicional, es decir, a cada vértice $v = a$ le asociamos una pareja $L(v)$ de la siguiente manera.

En el Paso 1, $L(v) = (\infty, *)$. Si al ejecutar el Paso 2 este vértice es vecino de a, entonces su nueva etiqueta es $L(v) = (p(u, v), u)$. En caso contrario, esta etiqueta inicial es $(\infty, *)$. Supongamos que en determinado momento la etiqueta de cierto vértice $v \notin S$ se actualiza, es decir que v es un vecino de un vértice u que acaba de entrar a S, de modo que $L(u) + p(u, v) < L(v)$. Entonces la etiqueta de v se cambia por $L(v) = (L(u) + p(u, v), u)$. El vértice u es predecesor de un camino de coste mínimo de a a v. De esta manera, si al terminar el algoritmo la etiqueta de v es $L(v) = (L(v), u)$, entonces observamos la etiqueta de u. Sea $L(u) = (L(u), t)$. Entonces un camino de coste mínimo de a a v debe ser dela forma $a, \ldots, t, u, v$. Al observar la etiqueta de t, obtenemos otro predecesor. Procedemos de esta manera hasta llegar a un vértice b cuya etiqueta sea $L(b) = (L(b), a)$. En ese momento habremos obtenido un camino $a, b, \ldots, t, u, v$ de coste mínimo igual a $L(v)$.

5.2. Análisis asintótico de tiempo y espacio

Ahora, para el análisis de tiempo, debemos notar que la implementación utilizada está basada en, básicamente, dos iteraciones anidadas. La primera, verifica el estado de *visitado* de los nodos, mientras que la segunda iteración opera sobre los vecinos de un nodo. El resto de iteraciones se emplea para comparar las distancias tentativas con las ya registradas y para imprimir el camino más corto.

Entonces, notemos que dado el número de vértices V de la gráfica, las iteraciones anidadas dependen de V. Esto es, la complejidad temporal del algoritmo es de $O(V)$.

Siguiendo un razonamiento análogo para el análisis asintótico de espacio, notemos que las variables críticas son aquellas que almacenan los estados de si los nodos han sido *visitados*, así como el arreglo que guarda las distancias entre nodos y aquel que guarda el padre de cierto nodo. Notemos entonces que todos estos arreglos crecen linealmente conforme crece el número de vértices V en la gráfica.

Por esto, la complejidad espacial del algoritmo propuesto es de $O(V)$.

6. Estrategias de diseño de algoritmos o estructuras de datos utilizadas

La estrategia utilizada fue la modificación del algoritmo original de Dijkstra para que únicamente nos diera la ruta con menos afluencia de la estación origen a la estación final y de esta manera pudiéramos ahorrar tanto tiempo como espacio en memoria.

Las estructuras de datos utilizadas fueron listas, diccionarios y dataframes para el almacenamiento de los datos y grafos para la representación de ellos.

7. Aplicación a los datos concretos

Antes de aplicar el algoritmo a los datos debemos tenerlos en el formato requerido, por lo que se realizó la limpieza y transformación de los datos. En nuestro problema en particular la limpieza de los datos fue sumamente extensa, debido a que las bases de datos estaban incompletas, se tenían registros corruptos o inconsistentes con el resto (y no podían ser consideradas como anomalías).El proceso consistió en lo siguiente:

- Eliminar las columnas que no eran relevantes para el proyecto.
- Corregir, a través de diccionarios, los nombres dados a las estaciones y a las líneas del Sistema de Transporte Colectivo Metro, (STC Metro de ahora en adelante), con el fin de establecer un convenio para emplear en el resto de la implementación.
- Debido a que los datos presentados no estaban en el formato adecuado, o Python no los interpretaba como valores enteros, se tuvo que realizar una búsqueda de datos para identificar y modificar aquellos registros que no cumplieran con el estándar establecido.
- La base de datos contaba con diferentes formatos de fecha, algunos no eran reconocidos como una fecha en hojas de cálculo, por lo cual también se tuvo que estandarizar su formato. Más aún, la base contaba con fechas no válidas, mismas que se eliminaron.
- La base de datos original no contaba con los días de la semana, sino únicamente con las fechas, por lo cual se tuvo que crear una columna que guardara el día de la semana correspondiente a la fecha del registro.

Una vez terminado el proceso de limpieza y transformación de los datos, se realizó el proceso para el tratamiento de los datos, que consisitió en lo siguiente:

- Se crearon listas, correspondientes a cada una de las líneas del STC Metro, para guardar en ellas las estaciones correspondientes a cada línea.
- Se creó, a partir de las listas anteriores, un conjunto con todas las estaciones del STC Metro sin contar los transbordos.
- Se creó un diccionario sobre el conjunto anterior empleando los nombres de las estaciones y un índice.

- Se crearon listas auxiliares para obtener las conexiones entre estaciones y, por tanto, tener un mapeo de las estaciones. i.e. esto nos ayuda a crear listas con los elementos;
  Universidad $->$ Copilco ; Copilco $->$ Miguel A. De Quevedo ....

- A partir de las listas anteriores se creó un data frame que guardara la conexión entre una estación y otra, así como el índice correspondiente a la estación de inicio y a la estación final.

Una vez terminado el proceso de tratamiento se realizó la aplicación del algoritmo a nuestros datos, el proceso consistió en lo siguiente:

- Se busca crear la matriz de adyacencia asociada a la red de STC Metro. Esta matriz, además, guardará la afluencia promedio por día de la semana de cada una de las estaciones. Para determinar la afluencia promedio por día de la semana de cada estación, primero debemos sumar cada afluencia por estación y posteriormente dividir entre el número de registros correspondientes a tal estación.

- Una vez obtenida la matriz correspondiente se crea la gráfica correspondiente a la matriz y se manda a llamar la función que contiene el algoritmo de Dijkstra para el día de la semana específico, la estación de partida y la estación a la que queremos llegar.

8. Conclusiones y posible trabajo a futuro

Tras la implementación detallada anteriormente y tras probar la implementación, se concluye que;

- El objetivo principal, (encontrar la ruta menos afluente dadas dos estaciones pertenecientes a la red del Sistema de Transporte Colectivo Metro de la CDMX y un día de la semana), se logró con satisfacción, proporcionando así alternativas al traslado de personas que signifiquen un riesgo menor de contagio y, potencialmente, bajo un tiempo de traslado menor.
- Tras analizar las afluencias promedio, y como conclusión secundaria, se observó que las estaciones con más afluencia son aquellas que fungen como transbordos.
- Como propuestas y trabajos futuros, se propone la implementación de un algoritmo de memoización para determinar los caminos menos afluentes entre cualesquiera dos estaciones en cualquier día. Esto, podría ser implementado corriendo el algoritmo presentado en todos los pares de nodos o con el algoritmo de Floyd-Warshall.
- Se propone dar seguimiento a la base de datos principal con el fin de que los resultados fueran mucho más precisos. Más aún, se propone la implementación de una base de datos con la afluencia diaria tomando en cuenta el horario. Esto, significaría resultados mucho más realistas.

9. Bibliografía

- Frana, Phil (August 2010). «An Interview with Edsger W. Dijkstra». Communications of the ACM 53
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.
- Knuth, D.E. (1977). Ä Generalization of Dijkstra's Algorithm". Information Processing Letters. 6 (1): 1–5. doi:10.1016/0020-0190(77)90002-3.
- Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James B.; Tarjan, Robert E. (April 1990). "Faster Algorithms for the Shortest Path Problem". Journal of the ACM. doi:10.1145/77600.77615.