**Universidad Peruana de Ciencias Aplicadas**

**UPC**



**CURSO**: Complejidad Algorítmica - CC184

Sección: CC42

**Trabajo Parcial**

Profesor: Luis Martín Canaval Sánchez

Integrantes:

* Maguiña Bernuy, Richard José (U202021097)
* Orive Pais, Francisco Javier (U201713987)
* Huerta Pahuacho, Junior Alfredo (U201620846)
* Montenegro Patiño, Raúl David (U20171C285)

Mayo 2021

**Índice**

Tabla de contenido

**Introducción3**

**Objetivos3**

Objetivo Principal3

Objetivos Específicos3

**Marco Teórico4**

Algoritmo de Dijkstra4

Algoritmo DFS7

Algoritmo BFS7

Algoritmo de Fuerza Bruta8

**Análisis de algoritmos9**

Algoritmo de Dijkstra9

Algoritmo DFS10

Algoritmo BFS11

Algoritmo de Fuerza Bruta13

**Conclusiones15**

**Bibliografía16**

**Introducción**

El presente proyecto busca resolver el problema del *vendedor viajero*. Este consiste en hallar la ruta más corta con la que visitar todas las ciudades de una lista dada, de manera que al finalizar se regrese a la ciudad de partida. Para ello se nos ha proporcionado un dataset que contiene todos los centros poblados del Perú en riesgo de inundación. Dicho dataset considera los valores de: departamento, provincia, distrito, centro poblado y posición geográfica.

Nuestra motivación recae en la idea de aportar una idea de solución a un problema de gran magnitud como lo es el *vendedor viajero*. Pese a que los algoritmos que vayamos a utilizar para este proyecto no sean los más eficientes para realizar este trabajo, nos prepara para trabajar con implementaciones más avanzadas en el futuro.

**Objetivos**

Objetivo Principal:

* Encontrar la ruta más corta entre los centros poblados de un distrito perteneciente a una provincia seleccionada por el usuario.

Objetivos Específicos:

* Hacer uso de los algoritmos de Dijkstra, DFS, BFS y Fuerza Bruta para la solución de esta entrega (Un algoritmo por integrante del equipo)
* Hacer uso del lenguaje de programación Python, dentro del entorno de desarrollo de Google Colab para la solución del problema.
* Posteriormente, expandir estas implementaciones para obtener la ruta, no solo a nivel distrital, sino, a nivel de todo el Perú.

**Marco Teórico**

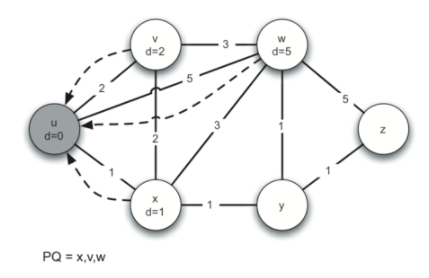
En primer lugar, para la lectura de toda la información contenida en el dataframe, se hizo uso de la librería *pandas*. De esta manera, mediante un arreglo obtuvimos los nombres y la cantidad de provincias existentes en la data. Luego, por medio de un arreglo, se almacenó el nombre de los distritos pertenecientes a una provincia seleccionada por el usuario y, a través de un diccionario, se almacenó el nombre de los distritos, junto a la cantidad de centros poblados existentes en este.

En segundo lugar, se hizo uso de la fórmula Haversine para calcular la distancia entre un centro poblado y otro, pues sus posiciones se representaban a mediante su *Latitud* y *Longitud*. Para esta operación, se hizo uso de funciones que podemos encontrar en la librería *math*. Asimismo, para plasmar en un grafo esta información, fue necesario importar la librería *networkx*, así como la librería de apoyo creada por el profesor del curso llamada *graph stuff*. Dicho grafo tiene como vértices los centros poblados de un distrito y sus respectivas aristas son el resultado obtenido del cálculo de la distancia, las cuales son consideradas como el peso de esas aristas.

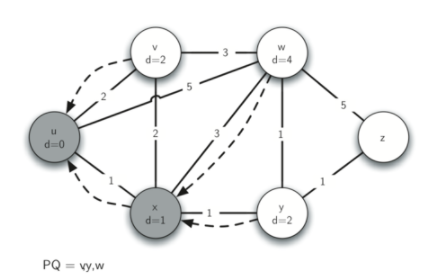
Por último, cada uno de los integrantes implementó los algoritmos y técnicas que se exponen a continuación:

1. **Algoritmo de Dijkstra:** se trata de un algoritmo iterativo que otorga la ruta más corta desde un nodo (o vértice) particular que será el punto de origen hacia todos los demás nodos del grafo. Esto es similar al resultado de un BFS. Para hacer un seguimiento del costo total desde el nodo inicial a cada destino se hace uso de un diccionario de distancias, el cual inicia en cero para el nodo inicial e infinito para los demás vértices. Dicho algoritmo actualiza estos valores hasta que representan el camino de menor peso desde el origen hasta el vértice, momento en el cual se devuelve el diccionario de distancias. Ocurre una iteración por cada vértice en el grafo; Sin embargo, el orden de iteración es controlado mediante una **cola de prioridad.** El valor utilizado para determinar en orden de los objetos en dicha cola es la distancia desde nuestro nodo inicial. Al usar una cola de prioridad, se asegura que mientras exploramos vértice tras vértice se haga siempre con aquel que tenga la menor distancia.  
     
   Las entradas a la cola de prioridad son tuplas de (distancia, vértice), las cuales permiten mantener una cola de vértices ordenados según la distancia. Cuando la distancia a un vértice que ya se encuentra en la cola es reducida, es necesario actualizar la distancia y por lo tanto darle una nueva prioridad, lo cual se consigue añadiendo una nueva entrada a la cola para el mismo vértice. Es importante resaltar que dicho algoritmo solo funciona cuando los pesos (distancias) son todos positivos. A continuación, se muestra un ejemplo de cómo interactúa el algoritmo:

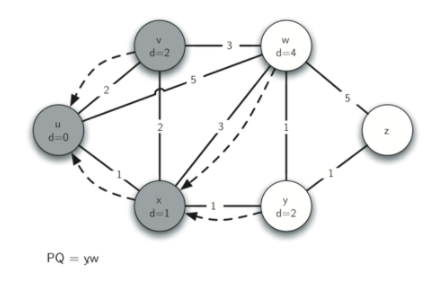
Dado un vértice inicial *u* y los adyacentes *v, w, x.* Consideramos las distancias (el costo) de los adyacentes como infinito. El nuevo costo para llegar a ellos a través del vértice inicial son todos sus costos directos, así que se actualiza el costo para cada uno de los tres nodos.



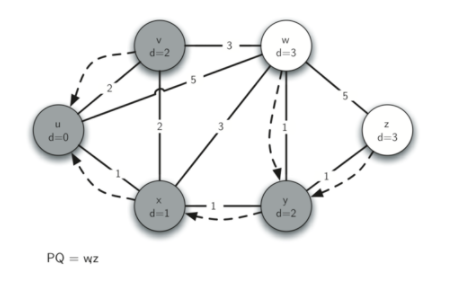
En la siguiente iteración de un bucle while se examinan los vértices adyacentes a *u.* El vértice x es el siguiente ya que tiene el menor costo entre todas las opciones y por lo tanto será la primera entrada a quitar de la cola de prioridad. En x analizamos a sus vecinos *u, v, w*, *y.* Por cada nodo vecino se comprueba si la distancia a ese vértice a través de x es inferior a la distancia previamente conocida. En este caso dicha situación ocurre dado que *y* está inicializada con distancia infinita; Sin embargo, para *u, v,* sus distancias son cero y dos. No obstante, si en lugar de ir desde *u* hacia *w* se recorre desde *x* hacia *w* obtenemos una distancia inferior, por lo que se actualiza *w* con una nueva distancia y se añade otra entrada a la cola de prioridad.



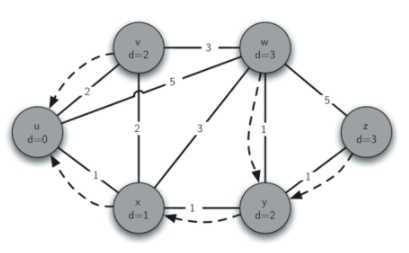
A continuación, se analizan los vértices vecinos a *v*. Al no ser necesario que se produzcan cambios se avanza al vértice *y.*



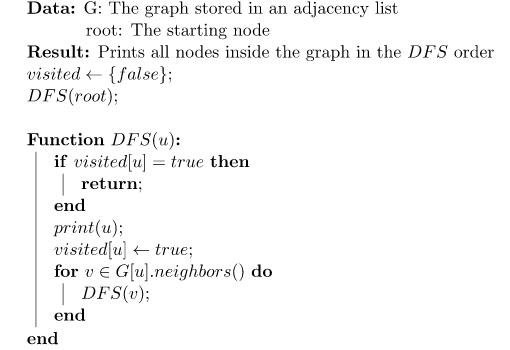
Tras analizar desde y se encuentra que es mejor ir tanto a *w* como a *z*, por lo que se ajustan las distancias.



Finalmente, tras analizar w y z no se producen cambios. Esto vacía la cola de prioridad y así concluye el algoritmo.

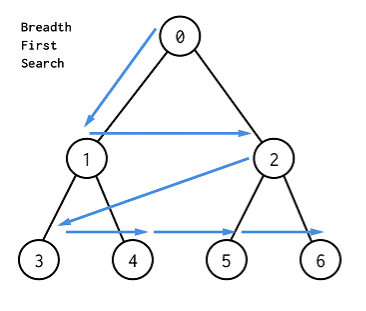


1. **Algoritmo DFS**: El algoritmo de recorrido en profundidad o DFS por su nombre en inglés, *Depth First Search*, permite explorar los vértices de un grafo de forma que primero se visitan los nodos adyacentes al visitado recientemente (Escobar & Giraldo, 2005). Este método de recorrido permite una implementación de manera recursiva, la cual se muestra en el siguiente pseudocódigo:



La complejidad de la implementación básica del DFS se basa en el número de vértices del grafo, el cual denominaremos *V*, y el número de aristas del mismo, denominado *A*. Por tanto, la complejidad de este algoritmo es O(V+A).

1. **Algoritmo BFS**: El algoritmo *Breadth-first search* se usa para la búsqueda en árboles o grafos. Para ello se necesita proveer de un nodo raíz que puede ser cualquier punto del árbol o grafo. Luego se empezarán a explorar todos los vecinos adyacentes del nodo raíz. Luego, se regresa al primer nodo explorado de los vecinos de la raíz, este nodo pasará a ser la nueva raíz y se explorarán todos sus vecinos adyacentes que no hayan sido explorados ya por otro nodo anterior. En caso de que ya no haya más nodos que visitar. Se regresará al nodo raíz inicial



1. **Fuerza Bruta**: En ciencias de la computación se conoce también como búsqueda exhaustiva o un algoritmo de generar y probar, esta técnica consiste básicamente en probar todos los posibles caminos para obtener soluciones y verificar cada uno de ellos hasta encontrar la solución.

Este método para resolver el problema está basado en utilizar la “potencia bruta” y no utiliza técnicas avanzadas para mejorar la eficiencia.

A continuación, mostraré una implementación básica donde se utiliza la técnica de fuerza bruta donde se prueban las soluciones y si no cumple las condiciones se pasa al siguiente candidato para la solución.

*c* ← *first*(*P*)

**while** *c* ≠ Λ **do**

**if** *valid*(*P*, *c*) **then**

*output*(*P*, *c*)

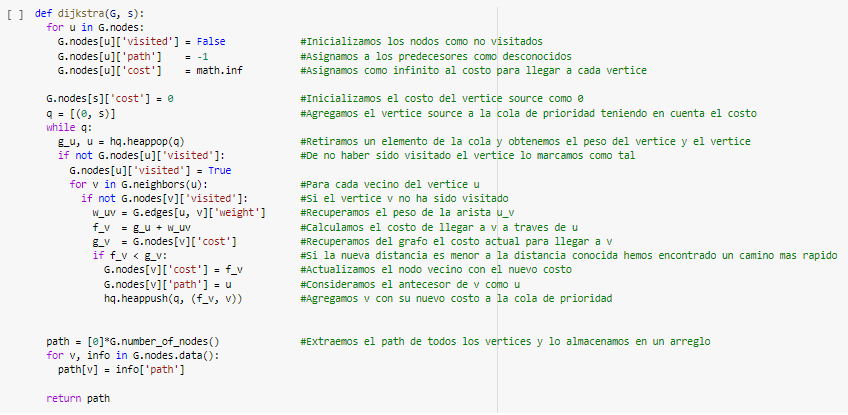
*c* ← *next*(*P*, *c*)

**end while**

El tiempo de complejidad de este algoritmo es expresado como **O(mn).** Para entender esto imaginemos que vamos a buscar una cadena de “n” caracteres en una cadena más grande de “m” caracteres y entonces tomaría n \* m intentos (en el peor de los casos) llegar a la solución.

**Análisis de algoritmos**

1. Algoritmo de Dijkstra (Implementado por **Francisco Orive**):



El diccionario de distancias toma un tiempo de O(V) ya que añadimos cada vértice en el grafo al diccionario creado antes de la función arriba mostrada.

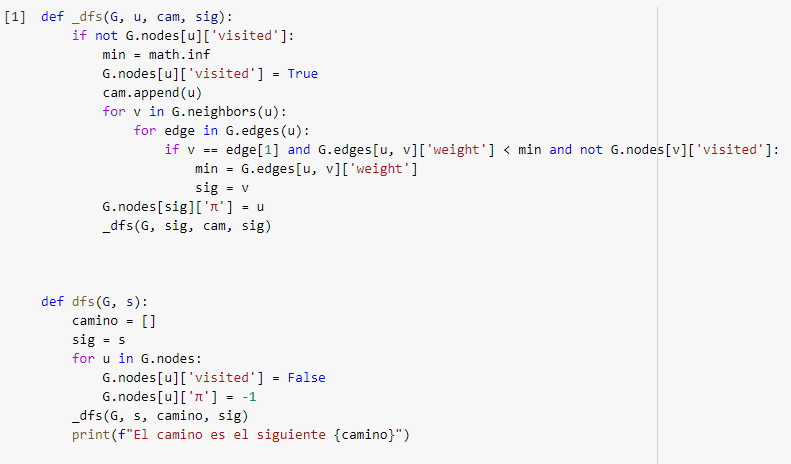
El bucle while se ejecuta una vez por cada entrada que se añade a la cola de prioridad y dichas entradas solo ocurren cuando exploramos una arista, por lo tanto, hay como mucho O(E) iteraciones en el bucle.

El bucle for se ejecuta como máximo una vez por cada vértice. Dicho bucle itera sobre las aristas salientes, por lo que el bucle while se ejecuta en tiempo O(E).

Finalmente, al considerar cada operación de la cola de prioridad, tanto la adición como la sustracción, obtenemos un tiempo O(log E). Dicho tiempo adicionado a lo anterior mencionado implica que el tiempo de ejecución es el siguiente: O(V + ElogE). Donde V son los vértices y E las aristas (Edges).

Mencionar que otra de las opciones para desarrollar este algoritmo sin estructuras de datos es utilizando un doble for. Sin embargo, su tiempo de ejecución sería de O(n^2), lo cual resulta menos eficiente que si ejecutamos el logarítmico gracias a las colas de prioridad.

1. Algoritmo DFS (Implementado por **Richard Maguiña**)



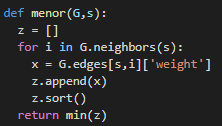
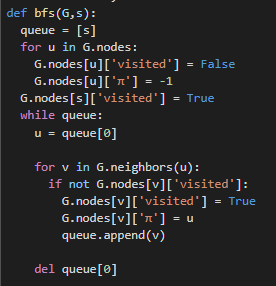
En la presente implementación podemos observar que el inicio del algoritmo es de manera corriente, se declara una lista vacía para almacenar el camino y se marcan todos los nodos como no visitados. Para este caso se creó una variable que almacenará el valor numérico del nodo el cual sería el siguiente en ser visitado. Hasta el momento se tiene una complejidad de O(V), en donde V es el número de vértices existentes en el grafo.

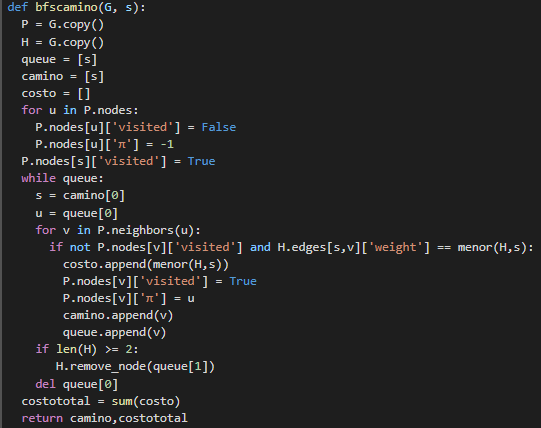
En el inicio del método recursivo “\_dfs” se realiza la condición pasa saber si el nodo enviado como parámetro de la función está marcado como visitado. En caso no lo esté, se inicializa la variable *min*, la cual almacenará el menor valor entre las aristas salientes del nodo actualmente analizado. Además, se marcará al nodo como visitado y se agrega al arreglo *camino*.

El siguiente paso consiste en analizar a cada uno de los vértices vecinos del vértice que se está analizando. Además, se tiene que utilizar un loop adicional para conocer todas las aristas salientes de este mismo nodo para analizarlas. En caso el peso de esta arista sea menor al almacenado en la variable *min* y, además, el nodo vecino (analizado en ese momento) no haya sido visitado, se asignará el nuevo valor mínimo y declaramos al nodo vecino como el siguiente en ser analizado. Sabemos que en este caso se toma en cuenta que cada uno de los nodos del grafo tiene una cantidad de V-1 vértices vecinos, pues todos se encuentran conectados entre sí. Por tanto, en el caso del doble for loop estaríamos encontrando una complejidad de O((V-1)^2), o más simplificado como O(V^2).

Lo último a realizar sería asignar el nodo padre del siguiente vértice que será analizado, así como realizar el llamado recursivo de la función “\_dfs” para terminar encontrando el camino final. Ante todo ello, terminamos encontrando una complejidad O(V) + O(V^2) el cual se termina simplificando como O(V^2).

1. Algoritmo BFS (Implementado por **Junior Huerta**)





El algoritmo bfs usado para esta forma es un revisado en clase sin embargo este algoritmo no funciona con pesos. Lo único que devuelve el algoritmo original es la unión de todos los nodos. Allí están todos los nodos unidos entre sí. Todos los nodos vecinos del nodo base tienen como punto de origen el nodo raíz inicial. (Primera imagen)

Para poder hallar el orden correcto de esta búsqueda con pesos. Se implementó un algoritmo que halle el menor peso del nodo vecino. (Segunda imagen)

En la tercera imagen podemos apreciar una modificación del algoritmo original implementado en clase este algoritmo guarda dos listas, un camino y un costo. La lista de camino guarda el orden en el que debe seguir la visita a cada vecino. La lista de costo guarda los pesos de cada vecino visitado. Tras la implementación de esto. Se agrega como primer elemento a la lista de caminos, el nodo raíz elegido

A su vez se crearán dos grafos temporales para evitar que las alteraciones que se harán en las siguientes líneas de código alteren el grafo original. Estos grafos copiaran el grafo inicial.

Primero se inicializa que “s” viene a ser el primer elemento del camino. En el bucle while se inicializa siempre que se seguirá siendo el primer elemento del camino. Ya que a diferencia del *queue* este se va borrando. En el For no se asignará por un orden, si no que el algoritmo buscará el nodo vecino con el menor peso posible en uno de los grafos temporales. Luego ese mismo costo se agrega a la lista de costos. Tras esto se asigna a la cola de caminos, el vecino elegido con menor peso.

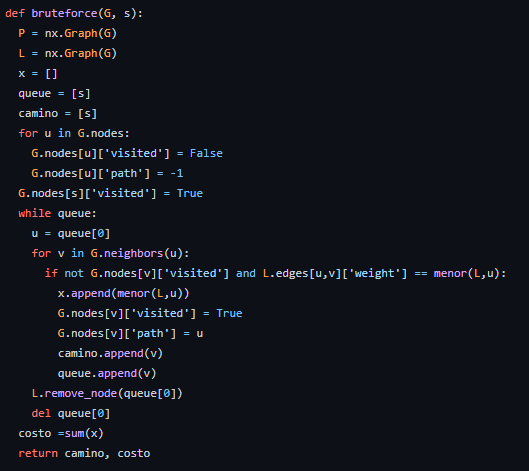
Tras esto en el grafo temporal usado para encontrar el nodo menor se le eliminará. El nodo vecino y todos sus vértices del grafo temporal. De esta forma se repetirá el ciclo while con el siguiente nodo vecino con menor peso. Esta eliminación se repetirá hasta que solo quede un nodo vecino. De esta forma el ciclo se parará con la cola del bfs normal.

Al finalizar el ciclo se sumarán todos los componentes de la lista de costos para hallar el costo total.

Finalmente se regresarán las listas de caminos y costos.

La complejidad para este algoritmo será de O(V+E) donde V son los vértices o nodos y E la cantidad de aristas. Todo esto debido a que se visitará cada nodo para hallar sus vecinos.

1. Fuerza Bruta (implementado por **Raúl Montenegro**)



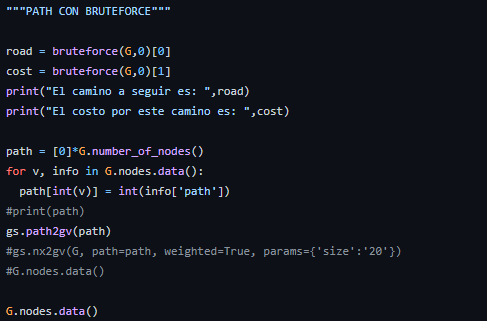
En este algoritmo podemos observar que se crea una copia del grafo para luego ser utilizada y buscar comparar el costo de llegar de una ciudad actual “u” las ciudades vecinas de esta y el cálculo del menor de estos.

En las siguientes líneas podemos observar que se asigna un atributo llamado “visited” para comprobar si el nodo actual ya ha sido recorrido o visitado, además asignamos un atributo llamado “path” para almacenar el nodo o ciudad de la que viene el recorrido o anterior a esta. En la línea final está marcado el atributo “visited” como True para señala que el nodo inicial del recorrido ha sido visitado.

En las siguientes líneas iniciamos el recorrido para comprobar si los nodos vecinos del nodo actual visitado cuentan con el menor costo para hacer el recorrido y agregándolos al camino, así como también se marca al nodo ya comprobado cómo visitado asignando “True” a su tributo “visited”.

Tenemos unas variables adicionales para almacenar los nodos en el orden en que se han recorrido y el costo total del recorrido final.

Finalmente retornamos el “camino” y el “costo” con los datos agregados en los mismos.



En estas líneas agregamos a unas nuevas variables obtenidas el valor de la función anteriormente implementada para ser mostradas como el camino a seguir “road” y el costo de este camino “cost”.

**Conclusiones**

El problema del *agente viajero* es en general muy complicado de resolver debido a las restricciones y variaciones de un grafo con el que nos podamos encontrar; sin embargo, resultó ser un ejercicio que prendió la curiosidad de todos.

Cada uno de los integrantes del equipo tuvo algunas dificultades en el desarrollo de sus algoritmos, pues el objetivo principal de este problema era encontrar la ruta más corta en un mismo grafo, por lo que cada uno tuvo que realizar cambios en la implementación de los mismo. Sin embargo, la gran variedad de herramientas que nos proporcionaba el entorno de desarrollo de *Google Colab*, permitieron simplificar algunas operaciones como la lectura de datos y su muestra en una tabla completa, así como la gráfica de los centros poblados que podíamos encontrar en un distrito seleccionado.

Aún es necesario retocar los algoritmos e implementarlos a nivel de toda la nación. No obstante, creemos que a medida que avance el curso, descubriremos nuevas estrategias que permitirán resolver en una medida adecuada y aceptable esta problemática. Cada algoritmo tiene sus pros y sus contras y su aplicación depende de las restricciones de los valores que hemos analizado, en especial los asignados a la latitud y longitud.

**Bibliografía**

Escobar, A. L. & Giraldo, E. (2005). *Implementación de grafos para el cálculo de la regulación en redes de distribución radiales.* Recuperado de: <https://www.redalyc.org/pdf/849/84911698008.pdf> [Consulta: 3 de mayo de 2021]

GeekforGeeks. (2021, marzo 31). *Dijkstra's algorithm.* Recuperado de: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/> [Consulta: 3 de mayo de 2021]

GeeksforGeeks. (2020, diciembre 4). *Breadth First Search or BFS for a Graph.* Recuperado de: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/> [Consulta: 3 de mayo de 2021]

GeekforGeeks (2020, noviembre 4) Traveling Salesman Problem (TSP) Implementation. Recuperado de: <https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/> [Consultado: 2 de mayo de 2021]

Free Code Camp (2020, enero 6). Brute Force Algorithm Explained. Recuperado de: <https://www.freecodecamp.org/news/brute-force-algorithms-explained/> [Consultado: 3 mayo de 2021]