Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Msc. Bidkar Pojoy

Juan González-Campo 21077
Pedro Marroquín 21801
Paulo Sánchez 21401

Algoritmo de Dijkstra

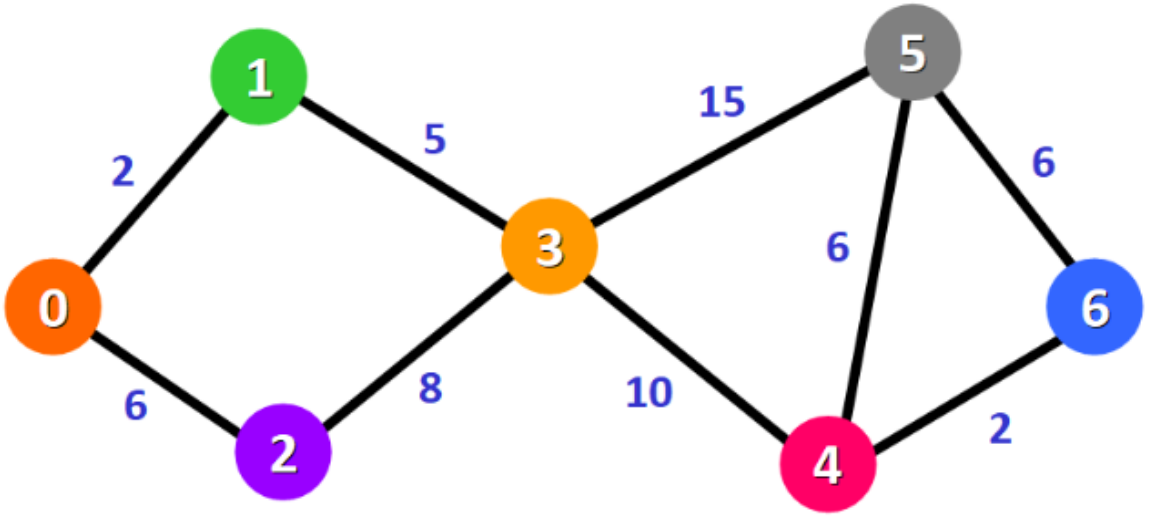
Definición de Problema

A partir de un grafo etiquetado, se busca estimar el costo del camino más corto desde un vértice dado al resto. Lo importante de este algoritmo es solucionar problemas de optimización, generalmente cuando se está representando una distribución geográfica. Donde las aristas tengan un coste (precio, distancia o similares) de la conexión entre dos lugares y sea fundamental determinar las rutas más cortas.

Algo que cabe destacar de este algoritmo es que genera uno a uno los caminos de un nodo al resto por orden creciente de la longitud. Además usa un conjunto S de vértices donde, a cada paso del algoritmo, se almacenan los nodos para lo que ya se conocen el camino mínimo y devuelve un vector indexado por vértices. De tal manera, que para cada uno de los vértices indexados se pueden determinar el coste del camino más convenientes a tales vértices (Salas, 2008).

Un ejemplo del algoritmo mencionado con anterioridad, es el siguiente grafo:

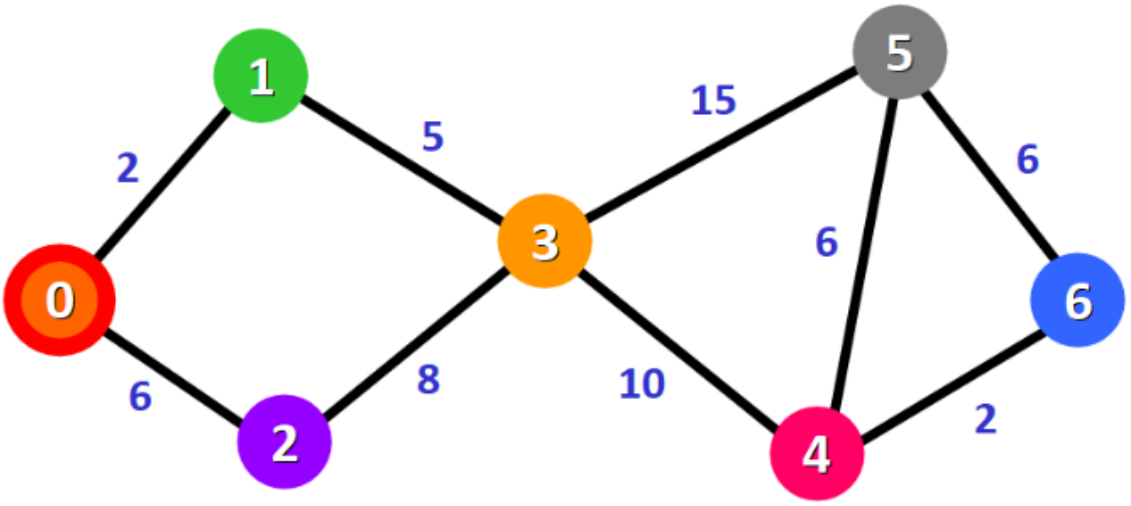
Gráfico 1: Ejemplo de Problema



El algoritmo va a generar el camino más corto del nodo 0 para todo el resto de nodos en el grafo. Inicialmente tenemos la lista de distancias, donde la distancia del nodo inicial hacia sí mismo es 0, para este caso puntual, el nodo inicial es 0 pero puede ser cualquier otro nodo a elección. Además, la distancia desde el nodo de partida hacia el resto de nodo no se ha determinado todavía, por lo que usamos el símbolo infinito para representar esto inicialmente.

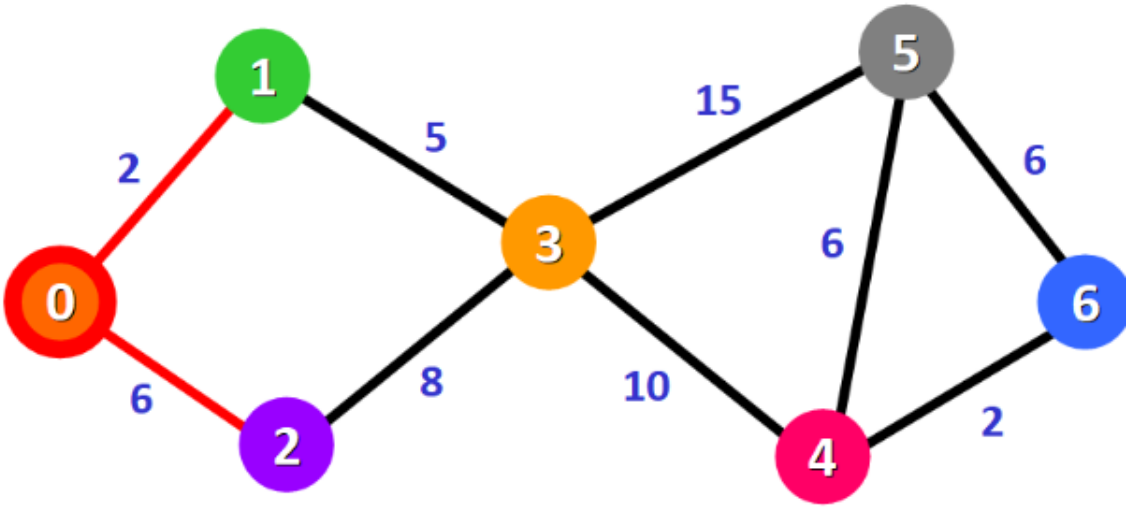
Asimismo, se genera una lista para llevar control sobre los nodos que aún no se han visitado. Dado que empezamos desde el nodo 0, podemos marcar este nodo como visitado.

Gráfica 2: Descartando nodo 0



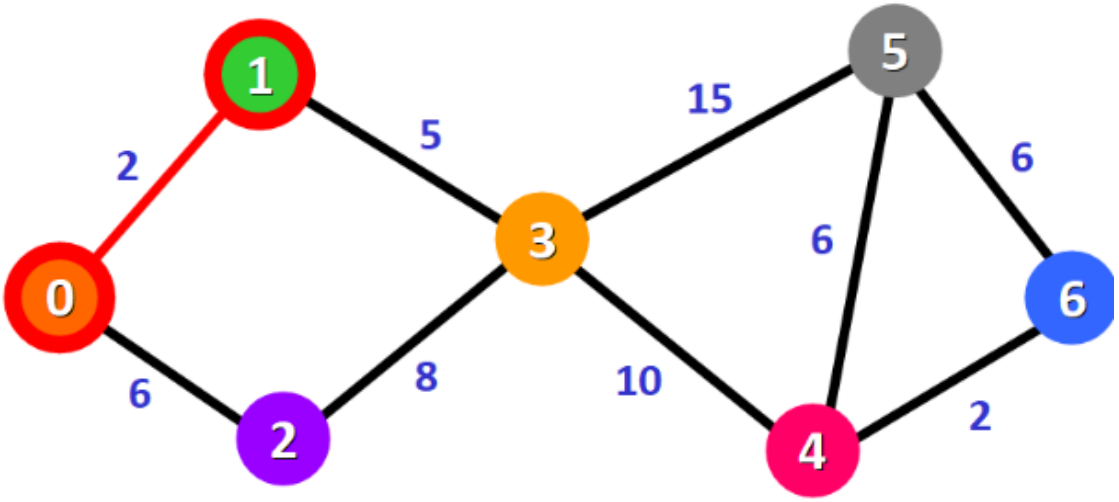
Ahora se necesita revisar la distancia del nodo 0 a sus nodos adyacentes. Como es evidente, estos son los nodos 1 y 2.

Gráfica 3: Revisando los nodos adyacentes



Se tiene que actualizar la lista de las distancias, desde el nodo 0 hacia el nodo 1 y nodo 2 con los pesos de las aristas que los conectan con el nodo 0. Ahora se descarta el nodo 1 y el nodo 2 de los nodos por visitar. Dado que el nodo 1 tiene la distancia más corta desde el nodo inicial lo agregamos al camino (Cassingena, 2020).

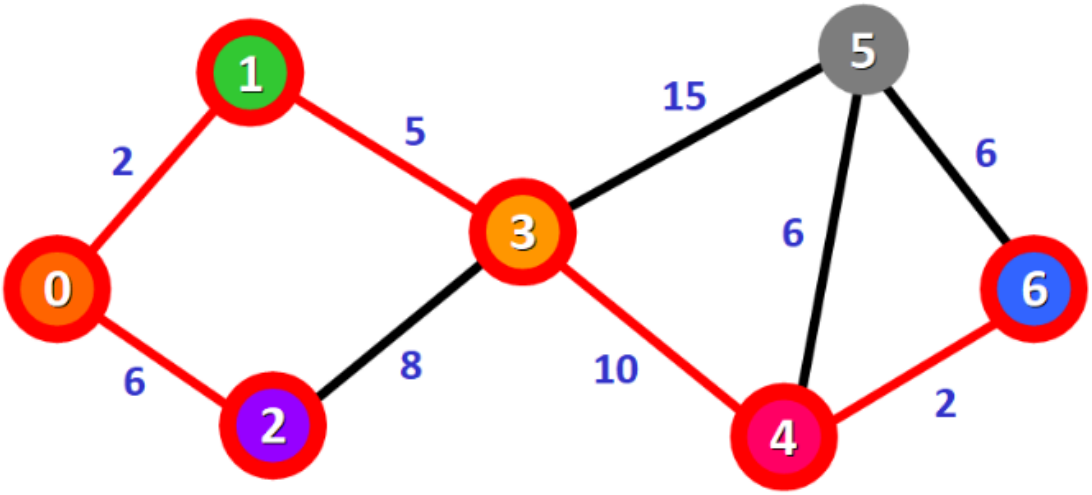
Gráfica 4: Primer componente del camino



Ahora de nuevo analizamos los nodos adyacentes para encontrar el camino más corto para alcanzarlos. El único nodo adyacente a 1 es el nodo 3, por lo que se procede a actualizar la distancia en el listado de distancias. Luego a partir de los nodos adyacentes a 0 encontramos el camino más corto hacia 3, agregando las distancias al listado.

Evaluamos los nodos adyacentes de 3, el nodo con el camino más corto es el nodo 4, por lo que se marca como visitado. Ahora estimamos los nodos adyacentes de 4, marcamos el nodo con la distancia más corta como visitado en este caso el nodo 6. Por lo que el camino màs eficiente es el siguiente:

Gráfica 5: Camino más corto



En nuestro caso para poder probar las implementaciones en condiciones similares, consideraremos solamente los casos de grafos completamente conectados, es decir si un grafo tiene 4 nodos, cada uno de ellos está conectado a los otros 3 nodos.

La aplicabilidad de este algoritmo es para resolver problemas principalmente de geolocalización, principalmente para hacer rutas de repartición. Por ejemplo, para repartir correo en una ciudad con cierta cantidad de camiones, cuál sería la forma más eficiente de

llevarlo a cabo. Otra aplicación relevante es para el diseño de aplicaciones de navegación, tales como Google Maps y Waze. Así como para el cableado de computadora, donde se tiene que conectar los diferentes componentes de una computadora a una placa. Los módulos que están conectados en la placa de un ordenador con cierta cantidad de pines tienen que ser conectados (Matai et al., 2010).

Algoritmos de Solución

Programación Dinámica:

Generalmente no es la técnica típicamente utilizada para el algoritmo Dijkstra, ya que este se basa en un enfoque de búsqueda exhaustiva. Sin embargo, es importante resaltar que la programación dinámica se emplea para resolver problemas de caminos más cortos con ciertas características.

Por ejemplo, si se tiene un grafo con una alta cantidad de nodos y aristas. Donde se necesita encontrar múltiples caminos más cortos entre diferentes pares de nodos en el grafo. Por lo que, un acercamiento con programación dinámica puede resolver este problema de manera eficiente y asegurar que sea correcto.

Greedy:

El algoritmo Dijkstra generalmente se describe como un algoritmo greedy. Es el camino más corto de un único nodo inicial que explora el grafo desde ese mismo nodo seleccionando los vértices con la distancia más pequeña. Esto es la esencia de un algoritmo greedy, ya que hace localmente la elección óptima en cada paso, con la esperanza que este va a llevar globalmente a la solución más eficiente. Sin embargo, la implementación greedy del algoritmo no necesariamente lleva a la respuesta correcta en su forma más simple.

Divide and Conquer:

Esto involucra hacer el problema en problemas más pequeños, resolviendo cada subproblema independientemente, y luego combinando las soluciones para los subproblemas para generar la solución del problema original. Dado que el algoritmo Dijkstra involucra explorar nodos en un grafo y actualizar distancias, este no estrictamente cumple con el acercamiento divide and conquer (*Dynamic Programming and Divide and Conquer*, s. f.); sin embargo, se pueden utilizar los grandes rasgos de la técnica para resolver el problema, por supuesto de una manera poco eficiente.

Por lo tanto, el enfoque que se tomó para poder hacer una división que tuviese sentido para resolver dos subproblemas fue de separar en 2 subgrafos de igual tamaño; es decir, a la mitad. Para hacer la separación a la mitad, se generó un algoritmo de búsqueda de los dos nodos con menor distancia entre ellos; de esta manera, se podría garantizar que al resolver los dos subgrafos y luego querer conectarlos de nuevo, la conexión entre ellos tendría el valor mínimo posible. Por la estructura del algoritmo de separación, este método tiene 2 condiciones. Primero, todos los nodos deben estar conectados entre sí, para asegurar que al hacer la separación no se tenga en un mismo subgrafo un nodo que no tenga conexión al resto de nodos. Segundo, debe ser una potencia de 2; de esta manera, se garantiza que se podrá seguir dividiendo en 2 hasta llegar al caso base.

Análisis Teórico

Programación Dinámica:

Primero se debe considerar el While Loop inicial, el cual visita cada nodo; es decir, si se tienen N nodos, se visitarán N. Luego consideramos la función min(), la cual en este caso en particular itera sobre N nodos al tener un grafo completamente conectado. Finalmente se tiene el for loop, que explora N - 1 nodos, pues explora a todos los vecinos de un nodo en particular.

Por esto tenemos una complejidad teórica de $O(N^2)$, pues el while loop, la función min() y el for loop tienen complejidad lineal, la cual se simplifica a una complejidad cuadrática.

Greedy:

Primero se debe considerar el While Loop inicial, el cual visita cada nodo es decir, si se tienen N nodos, se visitarán N. Luego se considera el primer for loop, en el cual nuevamente se pasan por los N nodos. Finalmente se tiene el segundo for loop, que explora (en nuestro caso, debido a que restringimos el problema a grafos completamente conectados) N - 1 nodos, pues explora a todos los vecinos de un nodo en particular.

Por esto tenemos una complejidad teórica de $O(N^2)$, pues el while loop y los for loops dentro de este tienen complejidad lineal, que se simplifica a una complejidad cuadrática, similarmente al algoritmo dinámico.

Divide and Conquer:

Los casos base tienen complejidad de $O(1)$, sin importar que sea retornar un único nodo y distancia 0 o una lista de 2 nodos y la distancia entre ellos, puesto a que obtener las llaves de un diccionario de 2 llaves es trivial su duración. Por lo tanto, la complejidad será definida por el algoritmo en casos generales; es decir, cuando realiza particiones y resuelve los subgrafos.

La sección del algoritmo para determinar los 2 nodos con distancia mínima, para así separar entre ellos, consiste en 2 for loops anidados recorriendo todos los nodos y todas las conexiones entre los nodos. Ya que nuestro algoritmo está especificado para grafos totalmente conectados (cada nodo tiene una conexión con los demás), significa que esto tiene una complejidad de $\Theta(N^2)$.

Separar los grafos en 2 mitades tiene una complejidad de $O(N)$, por lo que su complejidad será acotada por $O(N^2)$. Como se ha comprobado en clase, agregarle una cantidad de menor potencia a un complejidad, en este caso es $cN^2 + cN$, no altera la complejidad mayor; es decir, permanece siendo $O(N^2)$.

Finalmente, dividimos por la mitad al grafo original y resolvemos dos subgrafos para posteriormente volverlos a juntar entre los dos nodos que tenían distancia mínima. Por lo tanto, la ecuación de recursión será la siguiente:

$$T(N) = 2T(\tfrac{N}{2}) + \Theta(N^2)$$

Para resolver esta ecuación de recurrencia se utilizará el Master Method. La ecuación de recurrencia cae dentro del tercer caso del Master Method; su resolución se puede observar en la siguiente imagen:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 2} = n^1$$

$$f(n) = \Theta(n^2) = \Omega(n^{\log_2 2 + \varepsilon})$$

Aprobar:

$$2 \cdot f\left(\frac{n}{2}\right) \leq c f(n) \text{, con } c < 1$$

$$2 \cdot \left(\frac{n}{2}\right)^2 = 2\frac{n^2}{4} = \frac{n^2}{2} \leq cn^2$$

$$1 > c \geq \frac{1}{2}$$

$\Rightarrow$ se cumple condición de regularidad

$$\therefore T(n) = \Theta(n^2)$$

□

Se concluye que $T(N) = \Theta(N^2)$.

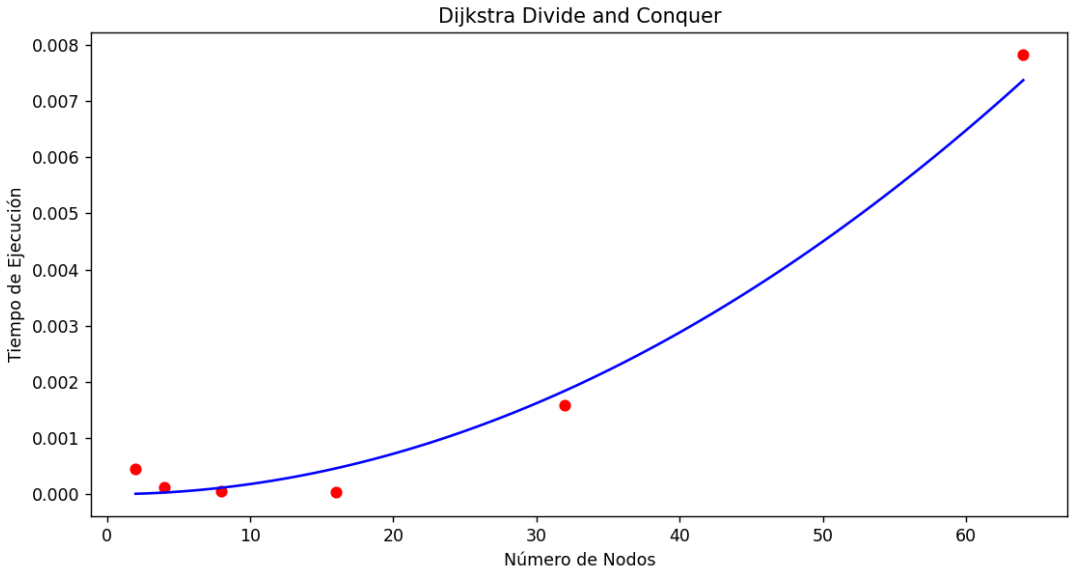
Análisis Empírico

Entradas:

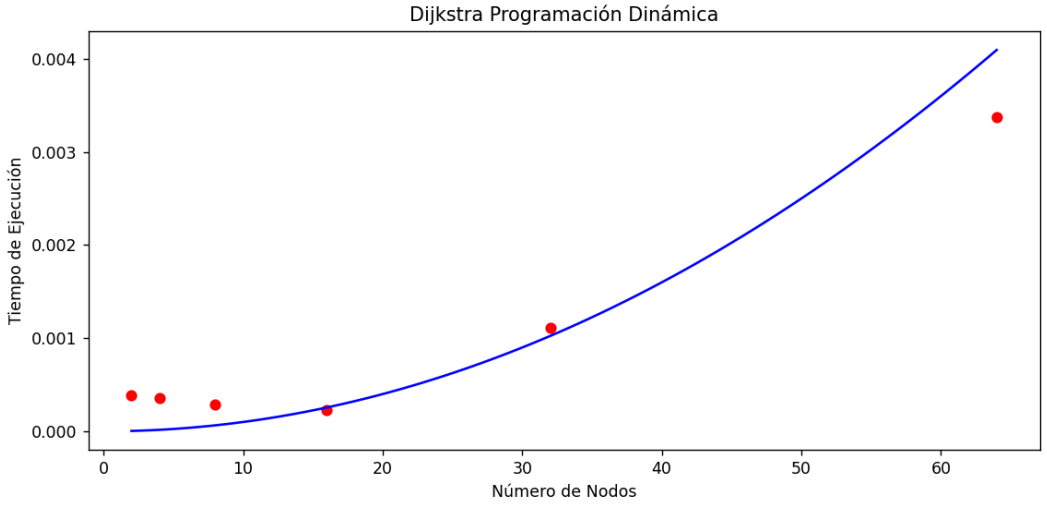
Las entradas se generaron a través de un programa en python, en el archivo generategraph.py. Estas se encuentran dentro del archivo graphs.txt.

Gráficos de tiempos de ejecución

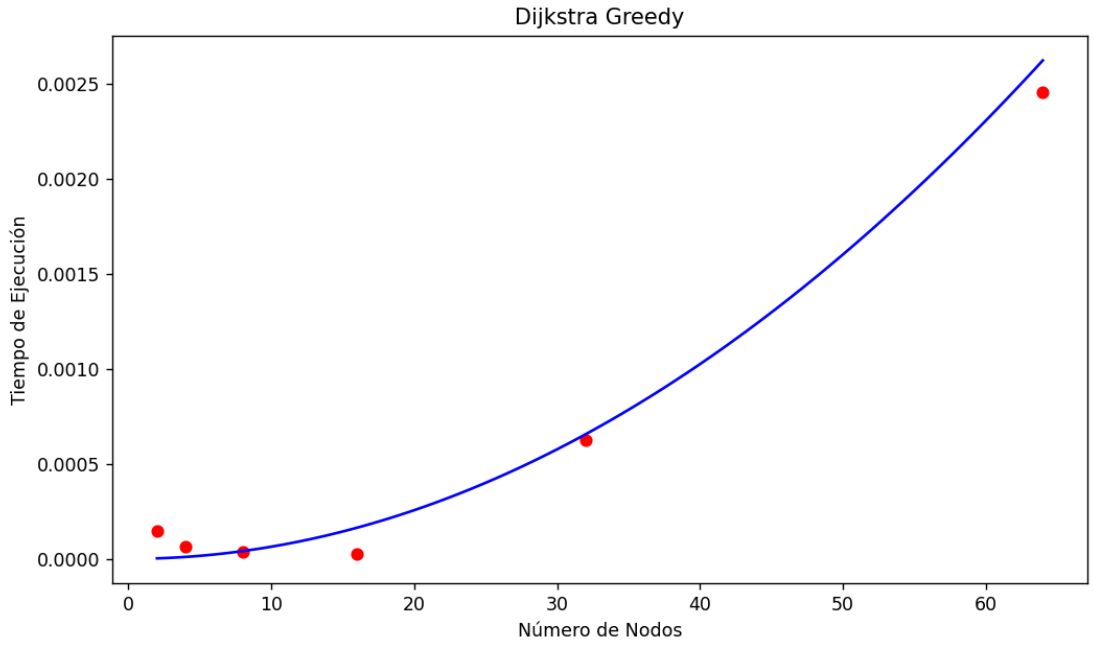
Divide and Conquer



Dynamic Programming



Greedy



Análisis

Se utilizó como entrada para las tres variantes de algoritmos, grafos entre 2 nodos y 64 nodos. Todas las variantes del algoritmo tuvieron un comportamiento adecuado al que se determinó a nivel teórico $O(n^2)$ mostrado con las lineas azules, que son funciones cuadráticas de la forma cn^2 con constantes 0.00000064 para el algoritmo greedy, 0.000001 para el de programación dinámica y 0.0000018 para divide and conquer. Sin embargo esta complejidad simplemente nos indica el comportamiento del tiempo de ejecución relativo al algoritmos por sí mismo, no en relación a otros algoritmos lo cual se puede notar con las constantes..

Es por esto que observamos en las gráficas que mientras el comportamiento es el mismo, la magnitud del tiempo difiere entre cada algoritmo. El algoritmo divide and conquer tiene el tiempo más alto de ejecución, variando entre 0.002 y 0.008. Seguido por el algoritmo de programación dinámica entre 0.002 y 0.004 y finalmente el más bajo es el greedy entre 0.00001 y 0.0025.

Es importante no solamente saber la complejidad del algoritmo, sino que también se debe de tener en cuenta el costo real de la ejecución, pues como se mencionó, la complejidad simplemente nos indica el comportamiento del algoritmo relativo a sí mismo, no contra otros algoritmos.

Fuentes

Cassingena, E. (2020, septiembre 28). *Dijkstra's Shortest Path Algorithm—A Detailed and Visual Introduction*. freeCodeCamp.Org. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

Dynamic Programming and Divide and Conquer. (s. f.).

Matai, R., Singh, S., & Lal, M. (2010). Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches. En D. Davendra (Ed.), *Traveling Salesman Problem, Theory and Applications*. InTech. https://doi.org/10.5772/12909

Salas, A. (2008). *Acerca del Algoritmo de Dijkstra*.