Análisis de Algoritmos

Comparación de Algoritmos de Camino más Corto: Dijkstra, Bellman - Ford, Floyd - Warshall

Nicolás André Durán Encina

Ingeniería Civil en
Informática
Univ. Católica del Maule,
CHILE

Octubre, 2018

Resumen. La utilización de algoritmos para encontrar rutas mínimas entre distintos puntos (origen y destino) se han vuelto cada vez más comunes con el paso del tiempo. Herramientas tecnológicas como sistemas GPS o de navegación han impulsado al constante análisis de los algoritmos utilizados para obtener dichos resultados de búsqueda. Se estudiarán los tiempos de ejecución y complejidades algorítmicas de los algoritmos Dijkstra, Bellman – Ford y Floyd – Warshall para probar hipótesis acerca de su funcionamiento bajo distintas circunstancias establecidas.

Palabras Claves: Grafos, camino más corto, vértices, complejidad algorítmica.

1 Introducción

En la teoría de grafos, el problema del camino más corto es el problema que consiste en encontrar un camino entre dos vértices (o nodos) de tal manera que la suma de los pesos de las aristas que lo constituyen es mínima. Un ejemplo de esto es encontrar el camino más rápido para ir de una ciudad a otra en un mapa. En este caso, los vértices representarían las ciudades y las aristas las carreteras que las unen, cuya ponderación viene dada por el tiempo que se emplea en atravesarlas.

El problema del camino más corto puede ser definido para grafos no dirigidos, dirigidos o mixtos. Un grafo, está definido por dos conjuntos de símbolos: nodos y arcos. Un arco consiste en un par ordenado de vértices y representa una posible dirección de movimiento que puede ocurrir entre vértices. Dicha ruta es una cadena en la que el nodo terminal de cada arco se define para el nodo inicial del siguiente arco. El problema de encontrar la ruta más corta (ruta de longitud mínima) desde el nodo 1 a cualquier otro nodo en un grafo se llama un problema de ruta más corta.

Por lo tanto, el problema de la ruta más corta es el problema de encontrar una ruta entre dos vértices (o nodos) en una gráfica de tal manera que la suma de los pesos de sus bordes constituyentes se minimice.

Para ilustrar el problema del camino más corto, resolveremos exhaustivamente y analizaremos un problema de ejemplo, generando grafos de manera aleatoria a lo largo del informe. En el ejemplo, la ruta más corta entre nodos es análoga al problema de encontrar la ruta más corta entre dos lugares en un mapa como se mencionó anteriormente: los vértices del gráfico corresponden a ciudades y los arcos corresponden a segmentos de carretera, cada uno ponderado por la longitud de su propio segmento.

Cabe recordar que dos vértices son adyacentes cuando poseen una arista común. Un camino en un grafo no dirigido es una secuencia de vértices tal que todo vértice actual es adyacente con el vértice siguiente.

Nuestro problema se centrará en encontrar los caminos más cortos de un vértice origen *v* a todos los demás vértices del grafo. Para ello se utilizarán los algoritmos Dijkstra, Bellman – Ford y Floyd – Warshall en donde se evaluarán 3 hipótesis distintas para analizar el comportamiento que tienen dichos algoritmos al momento de buscar las rutas más cortas entre los nodos de un grafo.

1.1 Estado del arte

Fue el pionero de la programación Edsger W. Dijkstra [ED1959] quien realmente descubrió como resolver el problema central de encontrar caminos más cortos, y su algoritmo homónimo sigue siendo una de las cosas más inteligentes de la informática. Un impecable defensor de la simplicidad y la elegancia en las matemáticas creía más o menos que cada problema complicado tenía una manera de abordarse y la matemática era una herramienta para encontrarla.

El resultado de una explosión de este tipo es que los problemas, como los problemas de ruta más corta, crecen tan rápidamente que se vuelven prácticamente incomputables, lo que lleva una cantidad de tiempo prácticamente infinita para resolver. Solo se necesita un

puñado de nodos en un mapa o gráfico dado para el número de combinaciones posibles para ingresar a los miles de millones, lo que requiere un tiempo demasiado alto e irrazonable.

Cada uno de los algoritmos utilizados tiene sus limitaciones. El algoritmo Dijkstra lo utilizaremos para determinar el camino más corto entre un único vértice origen hasta todos los otros vértices del grafo. A su vez, el algoritmo de Bellman – Ford [BF1958] resuelve el problema desde un vértice de origen si la ponderación de las aristas es negativa. Por su parte, el algoritmo de Floyd – Marshall [FW1959] da solución al problema de forma muy parecida al primer algoritmo mencionado.

Dentro de sus aplicaciones, los algoritmos de los caminos más cortos se aplican para encontrar direcciones de forma automática entre lugares físicos, como las rutas de conducción en sitios de mapas web como Google Maps. Para estas aplicaciones están disponibles rápidos algoritmos especializados.

Si un algoritmo representa una máquina abstracta no determinista como un grafo, donde los vértices describen estados, y las aristas posibles transiciones, el algoritmo del camino más cortos puede ser usado para encontrar una secuencia óptima de decisiones para llegar a un cierto estado final o para establecer límites más bajos en el tiempo necesario para alcanzar un estado dado.

Una aplicación más coloquial es la teoría de los "Seis grados de separación", a partir de la cual se intenta encontrar el camino más corto entre dos personas cualesquiera. Otras aplicaciones incluyen la investigación de operaciones, instalaciones y facilidad de diseño, robótica y transporte.

1.2 Antecedentes: Programación Dinámica

En la Informática, la programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas, como se describe a continuación.

Una subestructura óptima significa que soluciones óptimas de subproblemas pueden ser usadas para encontrar las soluciones óptimas del problema en su conjunto. Por ejemplo, el camino más corto entre dos vértices de un grafo se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

- Dividir el problema en subproblemas más pequeños.
- Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
- Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos ellos mismos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.

Una mala implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a subproblemas que ya han sido resueltos anteriormente. Esto se puede

evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se llama memorización. Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que sabemos que vamos a necesitar de antemano.

1.3 Hipótesis

Existirán distintas hipótesis que iremos viendo a lo largo del informe enfocadas en los algoritmos de camino más corto que trabajaremos y que han sido planteadas luego del estudio del tema de la mano de distintas literaturas.

1. La primera estará enfocada en que la complejidad del algoritmo Dijkstra no depende de la cantidad de aristas del grafo.
2. El peor caso del algoritmo Bellman-Ford se produce cuando el grafo es denso (gran cantidad de aristas).
3. Cuando se requiere calcular las distancias más cortas entre todos los vértices del grafo, independiente de la cantidad de vértices, el algoritmo Floyd-Warshall mejora el tiempo de ejecución de la fuerza bruta.

1.4 Objetivos Generales

Los objetivos que se tienen antes de comenzar el trabajo serán aquellos relacionados con conocer más acerca de los algoritmos de búsqueda del camino más corto para distintos tipos de grafos. Se busca también exponer los rendimientos que tienen estos métodos de distintos factores presentes en los experimentos realizados y como ayudan a la resolución de problemas matemáticos y/o computacionales. Uno de los objetivos principales será el de tratar conceptos de complejidad algorítmica y correlación entre distintas variables para poder interpretar de buena manera los resultados objetivos.

2 Problema Presentado

Se debe determinar si se cumplen o no las hipótesis presentadas con la utilización de grafos y 3 tipos de algoritmos distintos que resuelven problemas parecidos, aunque con ciertas limitaciones para cada uno. Se utilizará el algoritmo Dijkstra, Bellman – Ford y Floyd – Warshall.

Para poder determinar de mejor forma los tiempos de ejecución de los algoritmos, realizaremos 10 iteraciones para los mismos grafos con el fin de obtener sus tiempos de ejecución. De esta forma, se calculará un promedio de tiempo el cual representará una aproximación mucho más cercana al real comportamiento de cada método.

3 Técnicas de Resolución

El último concepto tratado fue el de complejidad algorítmica y pasaremos a explicar como se comporta frente a nuestros algoritmos de búsqueda de caminos más cortos para grafos.

Todos los algoritmos presentarán un mejor y peor caso de ejecución, o en términos más formales, contienen límites inferiores y superiores que son estudiados para comprender su futuro comportamiento [LH2015]. ¿Qué son estos límites? Definen las cantidades máximas y mínimas de iteraciones que deberán realizar cada algoritmo. Utilizaremos la letra "O" mayúscula para referirnos al máximo de instrucciones que tendrá que realizar un algoritmo en base a la cantidad de elementos de entrada, al límite superior o al "peor caso"; y con la letra griega "**Ω**" al límite inferior o el "mejor caso" que presentará el algoritmo.

3.1 Algoritmo Dijkstra

El algoritmo de Dijkstra permite encontrar eficientemente las rutas más cortas desde un nodo de un gráfico general (por lo tanto, puede contener ciclos) a todos los demás nodos. La única restricción es que las longitudes de arco deben ser no negativas. Esta es una suposición más restrictiva que la mera ausencia de ciclos negativos, pero en varias aplicaciones de la vida real tal suposición es satisfactoria.

La idea que subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo como se ha mencionado ya que al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo.

Teniendo un grafo dirigido ponderado de *n* nodos no aislados y el nodo inicial como "*start"*. Una lista de adyacencia (D) de tamaño n irá guardando todas las distancias mínimas desde el nodo inicial *start* hasta el resto de todos los nodos del grafo.

1. Todas las distancias en la lista de adyacencia se deben inicializar con un valor de infinito ya que no se conoce aún cuanto coste se requiere desde el nodo *start* al resto de los nodos del grafo.

2. Para el caso del nodo inicial, la ponderación almacenada en la lista de adyacencia *D* para dicho nodo será de *cero* ya que el costo para ir desde un nodo a él mismo es siempre *cero*.

3. El nodo actual en el que se encuentra el algoritmo deberá revisar todos los nodos adyacentes a este, a excepción de aquellos nodos que ya han sido visitados.

4. Ahora bien, en la lista de adyacencia tendremos el coste menor desde un nodo a otro almacenado. Para actualizar dicha lista en la posición correspondiente deberemos sumar el recorrido que ya se tiene hasta el nodo actual más la distancia al nodo

adyacente (si es que no ha sido recorrido). Si dicho resultado es menor al que se tenía en la lista de adyacencia, deberá ser actualizada.

5. Se marca como ya visitado el nodo actual.

6. Se toma como próximo nodo actual el de menor valor en la lista adyacente y se regresa al paso 3, mientras existan nodos no visitados.

```
función Dijkstra (Grafo G, nodo_salida s)
  //Usaremos un vector para guardar las distancias del nodo salida
al resto
  entero distancia[n]
  //Inicializamos el vector con distancias iniciales
  booleano visto[n]
  //vector de boleanos para controlar los vértices de los que ya
tenemos la distancia mínima
  para cada w ∈ V[G] hacer
     Si (no existe arista entre s y w) entonces
         distancia[w] = Infinito
     Si_no
         distancia[w] = peso (s, w)
     fin si
  fin para
  distancia[s] = 0
  visto[s] = cierto
  //n es el número de vértices que tiene el Grafo
  mientras que (no_estén_vistos_todos) hacer
     vértice = tomar_el_mínimo_del_vector distancia y que no esté
visto;
     visto[vértice] = cierto;
     para cada w ∈ sucesores (G, vértice) hacer
         si distancia[w]>distancia[vértice]+peso (vértice, w)
entonces
            distancia[w] = distancia[vértice]+peso (vértice, w)
         fin si
     fin para
  fin mientras
fin función.
```

Figura 1: Pseudo-Código de Algoritmo Dijkstra

Consideremos ahora la complejidad del algoritmo de Dijkstra. Cuando agregamos un nuevo nodo, actualizamos las etiquetas de todos sus sucesores, lo que requiere O (n) tiempo. La selección de la etiqueta mínima también es O (n). Dado que el ciclo "*While*" debe ejecutarse n veces, la complejidad resultante es O (n^2 + a), con *n* cantidad de nodos del grafo y *a* cantidad de aristas. Si aplicamos los principios del método de *notación asintótica* en donde discriminamos aquellos valores que sean menos significantes para la ecuación, la complejidad temporal algorítmica de Dijkstra resulta como O (n^2).

Dicha complejidad se puede cambiar si mantenemos la lista de etiquetas temporales ordenadas. En este caso, se requiere un esfuerzo de registro n en cada actualización de la etiqueta, pero la etiqueta más baja se puede encontrar en tiempo constante. Dado que cada arista es considerada exactamente una vez por el algoritmo, la complejidad es entonces O (m log n).

3.2 Algoritmo Bellman - Ford

El algoritmo de Bellman-Ford determina la ruta más corta desde un nodo origen hacia los demás nodos en donde para ello es requerido como entrada un grafo cuyas aristas posean pesos. La diferencia de este algoritmo con los demás es que los pesos pueden tener valores negativos ya que Bellman-Ford permite detectar la existencia de un ciclo negativo.

¿Cómo es que trabaja el algoritmo? El algoritmo parte de un vértice origen que será ingresado, a diferencia de Dijkstra que utiliza una técnica voraz para seleccionar vértices de menor peso y actualizar sus distancias mediante el paso de *relajación*, Bellman-Ford simplemente *relaja* todas las aristas y lo hace *n* - 1 veces, siendo *n* el número de vértices del grafo.

Para la detección de ciclos negativos realizamos el paso de relajación una vez más y si se obtuvieron mejores resultados es porque existe un ciclo negativo. Para verificar por qué tenemos un ciclo podemos seguir relajando las veces que queramos y seguiremos obteniendo mejores resultados.

El algoritmo de Dijkstra es un algoritmo voraz (es una estrategia de búsqueda por la cual se sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima.) y la complejidad del tiempo es O (n^2). Dijkstra no funciona para gráficos con aristas de peso negativo, Bellman-Ford trabaja para tales gráficos. Bellman-Ford también es más simple que Dijkstra y combina bien para sistemas distribuidos. Pero la complejidad temporal de Bellman-Ford es O (N·A), con *N* siendo la cantidad de vértices y *A* la cantidad de aristas, siendo mayor a Dijkstra.

El algoritmo funciona de la siguiente manera:

1. Se inicializan las distancias desde el nodo fuente a todos los vértices como infinito y la distancia al nodo inicial en sí misma como 0. Se crea una lista de adyacencia *distancias[ ]* de tamaño N con todos los valores como infinito excepto *distancias [nodo inicial]*.

2. Luego se calculan las distancias más cortas.

    a. Con v como nodo actual y u como nodo adyacente, si se cumple que: distancia [v] > distancia [u] + peso de la arista uv, entonces se actualiza la lista distancia[v] (distancia [v] = distancia [u] + peso del borde uv).

3. Este paso informa si hay un ciclo de peso negativo en el gráfico. Se hace lo siguiente para cada borde u-v:

    a. Si distancia [v] > distancia [u] + peso del borde uv, entonces el grafo contiene un ciclo negativo.

La idea del paso 3 es que el paso 2 garantiza distancias más cortas si el gráfico no contiene un ciclo de peso negativo. Si recorremos todos los bordes una vez más y obtenemos un camino más corto para cualquier vértice, entonces hay un ciclo de peso negativo.

```
Función BellmanFord_Optimizado(Grafo G, nodo_origen s)
       // inicializamos el grafo. Ponemos distancias a INFINITO
menos el nodo origen que
       // tiene distancia 0. Para ello lo hacemos recorriéndonos
todos los vértices del grafo
       para v ∈ V[G] hacer
           distancia[v]=INFINITO
           padre[v]=NULL
       distancia[s]=0
       encolar(s, Q)
       en_cola[s]=TRUE
       mientras Q!=0 entonces
           u = extraer(Q)
           en_cola[u]=FALSE
           // relajamos las aristas
           para v ∈ ady[u] hacer
               si distancia[v]>distancia[u] + peso(u, v) entonces
                   distancia[v] = distancia[u] + peso (u, v)
                   padre[v] = u
                   si en_cola[v]==FALSE entonces
                       encolar(v, Q)
                       en_cola[v]=TRUE
            fin para
        fin mientras
fin función
```

Figura 2: Pseudo-Código de Algoritmo Bellman - Ford

¿Como funciona esto? Al igual que otros problemas de programación dinámica, el algoritmo calcula las rutas más cortas de manera ascendente. Primero calcula las distancias más cortas que tienen a lo sumo un nodo en el camino. Luego, calcula las rutas más cortas con un máximo de 2 nodos, y así sucesivamente. Después de la iteración i-nésima del ciclo externo, se calculan las rutas más cortas con al menos i nodos. Puede haber máximo N - 1 nodos en cualquier ruta, es por eso por lo que el bucle externo se ejecuta N - 1 veces. La idea es, asumiendo que no hay un ciclo de peso negativo, si hemos calculado las rutas más cortas con un máximo de i nodos, entonces una iteración sobre todos los nodos garantiza dar la ruta más corta con un máximo de (i + 1) nodos.

3.3 Algoritmo Floyd - Warshall

El algoritmo de Floyd-Warshall, descrito en 1959 por Bernard Roy, es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. El algoritmo de Floyd-Warshall es un ejemplo de programación dinámica.

El funcionamiento del algoritmo se basa en:

1. Partimos de una matriz, que almacena las distancias entre cada par de vértices conectados, o INF en caso de que no exista conexión.

2. En cada iteración k, comprobamos si la distancia entre los vértices i→j almacenada en la tabla es mayor que la distancia entre i→k más la distancia de k→j.

3. De ser así se actualiza la tabla de distancias, almacenando la nueva distancia. Si desea recuperar el recorrido, también será necesario actualizar la tabla de predecesores, para incorporar k en el camino de i→j.

Tras la iteración k = nVertices, la tabla de distancias contendrá la distancia mínima entre cada par de vértices.

El algoritmo de Floyd-Warshall compara todos los posibles caminos a través del grafo entre cada par de vértices. El algoritmo es capaz de hacer esto con sólo $O(n^3)$ comparaciones (esto es notable considerando que puede haber hasta n^2 aristas en el grafo, y que cada combinación de aristas se prueba). Lo hace mejorando paulatinamente una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima.

```
/* Suponemos que la función pesoArista devuelve el coste del
camino que va de i a j
   (infinito si no existe).

También suponemos que      es el número de vértices y
pesoArista(i,i) = 0
*/

int camino[][];
/* Una matriz bidimensional. En cada paso del algoritmo,
camino[i][j] es el camino mínimo
de i hasta j usando valores intermedios de (1..k-1). Cada
camino[i][j] es inicializado a
*/

procedimiento FloydWarshall ()
   para k: = 0 hasta n – 1

          camino[i][j] = mín ( camino[i][j],
camino[i][k]+camino[k][j])

   fin para
```

Figura 3: Pseudo-Código de Algoritmo Floyd - Warshall

4 Resultados

4.1 Hipótesis 1: La complejidad del algoritmo Dijkstra no depende de la cantidad de aristas del grafo

El algoritmo Dijkstra consiste en buscar el camino más corto desde un nodo inicial hasta el resto de los nodos del grafo. Debido a esta particularidad es que el grafo debe ser completamente conexo y la ponderación de sus aristas siempre positiva. Para el caso de valores negativos, se utilizará el algoritmo de Bellman – Ford descrito más adelante.

Tendremos dos tipos de complejidades para el algoritmo Dijkstra: con *montículos* y sin la utilización de ellos. Los montículos son una estructura de datos del tipo árbol con información perteneciente a un conjunto ordenado. Los montículos máximos tienen la característica de que cada nodo padre tiene un valor mayor que el de cualquiera de sus nodos hijos, mientras que, en los montículos mínimos, el valor del nodo padre es siempre menor al de sus nodos hijos. Por lo tanto, la utilización de montículos ayudan al algoritmo a encontrar el camino más corto en menor tiempo por lo que su complejidad es de O(E + V log V) siendo su mejor caso, con "V" igual a los vértices y "E" las aristas del grafo trabajado. Como mencionamos anteriormente, también se puede desarrollar este método sin la utilización de montículos y con la ayuda de una lista de adyacencia donde almacenaremos todos los caminos más cortos desde el nodo inicial al resto, obteniendo una complejidad algorítmica de $O(V^2 + E)$, con "V" representando a la cantidad de vértices y "E" las aristas nuevamente. Dicha complejidad se aplica para el peor caso posible de este algoritmo.

Como podemos observar ambas complejidades dependen de la cantidad de aristas del grafo. A mayor cantidad de vértices, el algoritmo debiese tardar más en encontrar el camino mínimo ya que se busca encontrar todas las rutas óptimas desde el vértice inicial a cada uno de los otros nodos. Lo mismo ocurre para el caso de las aristas en donde a mayor densidad tenga el grafo, más tiempos tardarán en encontrar la solución ya que existirán más caminos a examinar.

Para entender lo anterior de una manera gráfica se ha probado el algoritmo de Dijkstra con la utilización de grafos con distintas cantidades de vértices (100, 200, 300 y 400) y distintas densidades de aristas (desde un 10% hasta un 90% de densidad del grafo) para cada una de las dimensiones del grafo. ¿Qué es lo que buscamos? Identificar cómo a mayor cantidad de aristas y vértices, los tiempos de ejecución variarán también, como se puede observar el Gráfico 1:

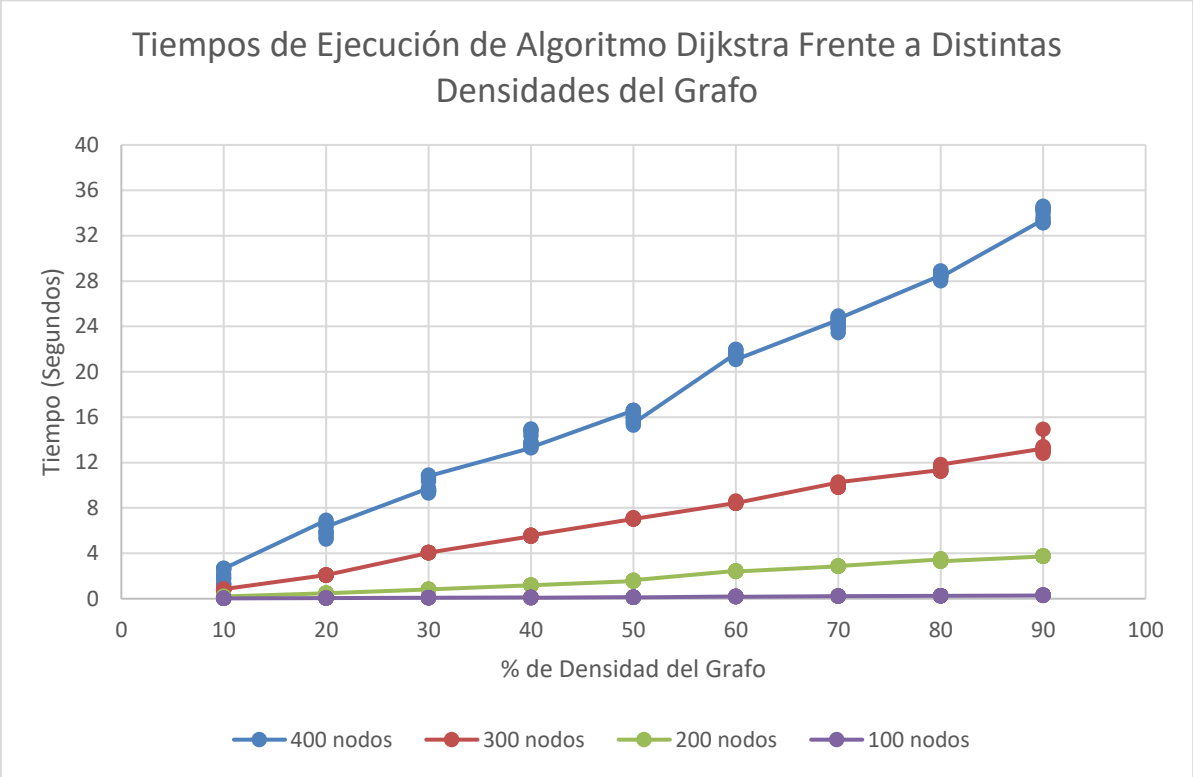

Gráfico 1: Algoritmo de Dijkstra

Como se observa, el aumento en la cantidad de aristas, si seguimos cada una de las líneas de dispersión graficadas, va aumentando en los tiempos de ejecución. Un grafo menos denso (con un 10% de densidad de aristas) tarda menos tiempo en ejecutarse que un grafo más denso (90% de densidad), como se puede ver en el Gráfico 1. Por ejemplo, para el caso de un grafo con 400 Vértices en su interior conectados unos con otros, cuando se tiene una densidad baja, el algoritmo tarda en resolver el problema en un promedio de 1,8 segundos de ejecución. Si comparamos este tiempo cuando el grafo alcanza uno de sus mayores índices de densidad, llega a tardar aproximadamente 34,2 segundos lo cual refleja un cambio exponencial en sus tiempos de ejecución a medida que sus variables aumentan, pero esto no es tan extraño ya que eso se refleja en su complejidad.

Densidad (%)	100 vértices	200 vértices	300 vértices	400 vértices
100	0,3010	3,7505	13,4439	34,2959
90	0,2577	3,3737	11,3805	28,1546
80	0,2190	2,8658	9,9560	24,8212
70	0,1852	2,4250	8,4372	21,9231
60	0,1257	1,5638	7,05161	16,8667
50	0,0992	1,1823	5,5386	14,25865
40	0,07344	0,8106	4,05390	10,5391
30	0,0493	0,4812	2,0715	6,73012
20	0,02690	0,1828	0,8274	1,82997

Tabla 1: Promedios de tiempo (segundos) de Algoritmo Dijkstra

Pero ¿Esto comprueba la hipótesis? Debemos entender que la complejidad algorítmica nunca nos dirá el tiempo exacto que tardará un algoritmo, ni siquiera una aproximación de este ya que influyen diversos factores que afectan los tiempos como el ambiente de trabajo (máquina utilizada, procesador, etc.) en el que se encuentra ejecutando el programa. Para el caso de la complejidad cuando no utilizamos montículos, que es la forma en la que se obtuvieron los resultados anteriores, la complejidad original es de $O(V^2 + E)$. Se observa claramente que depende tanto de la cantidad de nodos o vértices y de la cantidad de aristas presentes en el grafo. De igual forma, si conocemos los principios del método de *Notación Asintótica* sabremos que dicha complejidad se puede "simplificar". La notación asintótica se aplica cuando descartamos los coeficientes constantes y los valores menos significativos. Si graficáramos la función $V^2 + E$ nos daríamos cuenta de que el valor cuadrático de V tenderá a crecer en una mayor proporción que E, por lo que se considera a las aristas como un término menos significativo en nuestra función cuadrática. Por lo tanto, la complejidad algorítmica de nuestro algoritmo Dijkstra se puede expresar como $O(N^2)$. ¿Qué quiere decir esto? Para nuestro método de encontrar los caminos más cortos, la cantidad de vértices que posea el grafo será mucho más significativa que la cantidad de aristas que este tenga.

Si observamos el problema desde otra perspectiva, el algoritmo de Dijkstra para poder concluir su trabajo deberá tomar un nodo inicial y calcular el o los caminos más cortos (si es que existe más de alguno con una sumatoria de ponderación igual) hasta cada uno de los nodos presentes. Esto ocurre sin importar la cantidad de aristas que tenga el grafo, el algoritmo de igual forma deberá encontrar la manera de obtener la ruta mínima a todo el resto de los vértices partiendo por un punto inicial.

Si el grafo es más o menos denso, la complejidad seguirá siendo siempre la misma: $O(V^2)$. Esto se cumple debido a que utilizamos desde un comienzo un grafo *conexo* el cual refiere a que todos los nodos presentes tienen al menos una arista de entrada o salida, teniendo un mínimo de $V - 1$ aristas en total. Sabiendo esto, podemos asegurar que a partir de un vértice inicial podremos encontrar el camino más corto a todo el resto de los vértices del grafo.

Como mencionamos en un comienzo, la complejidad algorítmica y los tiempos de ejecución no significan lo mismo, aunque se relacionan directamente. Esto se vuelve a mencionar ya que a pesar de que la cantidad de aristas influya en la forma en la que los tiempos cambiarán, la complejidad no lo hará ya que se mantiene constante. Sin importar las circunstancias y si el gráfico es completo o disperso, el algoritmo tratará de encontrar siempre las rutas mínimas al resto de los vértices, ya que existe al menos una arista que los una a otro nodo debido a que se utiliza un grafo conexo. De esta forma se confirma que la hipótesis planteada es correcta.

4.2 Hipótesis 2: El peor caso del algoritmo Bellman – Ford se produce cuando el grafo es denso

El algoritmo Bellman – Ford tiene la misma función que el algoritmo Dijkstra que busca a partir de un único vértice inicial todos los caminos mínimos al resto de todos los pares de vértices del grafo. Su particularidad es que puede detectar ciclos con ponderaciones de sus aristas siendo estas negativas, a diferencia del algoritmo Dijkstra.

¿Cómo trabaja? Para detectar el ciclo negativo lo que hace es una comparación: La distancia del nodo predecesor al nodo actual que analizamos debe ser mayor a la distancia

del nodo en el que nos encontramos más el peso entre los dos nodos, es decir, entre el predecesor y el que estamos analizando. Si es que se cumple lo anterior, el algoritmo se seguirá ejecutando.

De igual manera, Bellman – Ford y Dijkstra son bastante similares en su estructura: mientras que Dijkstra busca solo por los vecinos inmediatos de un vértice, Bellman va a través de cada arista en todas las iteraciones que realiza. Gracias a esto último y como ambos algoritmos calculan la distancia de un vértice inicial a todo el resto de los vértices presente en un grafo, es que su complejidad temporal algorítmica es de O(VE) con "V" siendo la cantidad de vértices y "E" la cantidad de aristas presente. Si analizamos la complejidad anterior podemos suponer que a medida que aumentamos la cantidad de aristas y vértices del grafo, sus tiempos de ejecución irán aumentando ¿Pero podemos afirmar que se comportará mejor o peor que el algoritmo Dijkstra?

Si ponemos a prueba el algoritmo de Bellman – Ford utilizando distintas cantidades de aristas y vértices podremos graficarlo para posteriormente realizar un análisis de su comportamiento. Se han utilizado 3 grafos distintos con 100, 200 y 300 vértices respectivamente. Para comenzar a acercarnos a nuestra hipótesis se han tomado distintas densidades del grafo que van desde un 10 hasta un 100% de densidad del grafo. ¿Qué significa lo anterior? A mayor densidad de grafo, los vértices se encontrarán más interconectados entre sí gracias a las aristas que los unen. Un grafo denominado *completo* refiere a que la cantidad de aristas presentes en el grafo es máxima. En palabras sencillas, para cada nodo existirá una cantidad de V – 1 aristas salientes conectadas a los V – 1 vértices restantes, sin contar al nodo actual que estamos describiendo. Esto significa que todos los nodos tendrán una arista que los conecta de manera directa con el resto de ellos. Si observamos el Gráfico 2 se podrá observar el comportamiento del algoritmo Bellman - Ford.

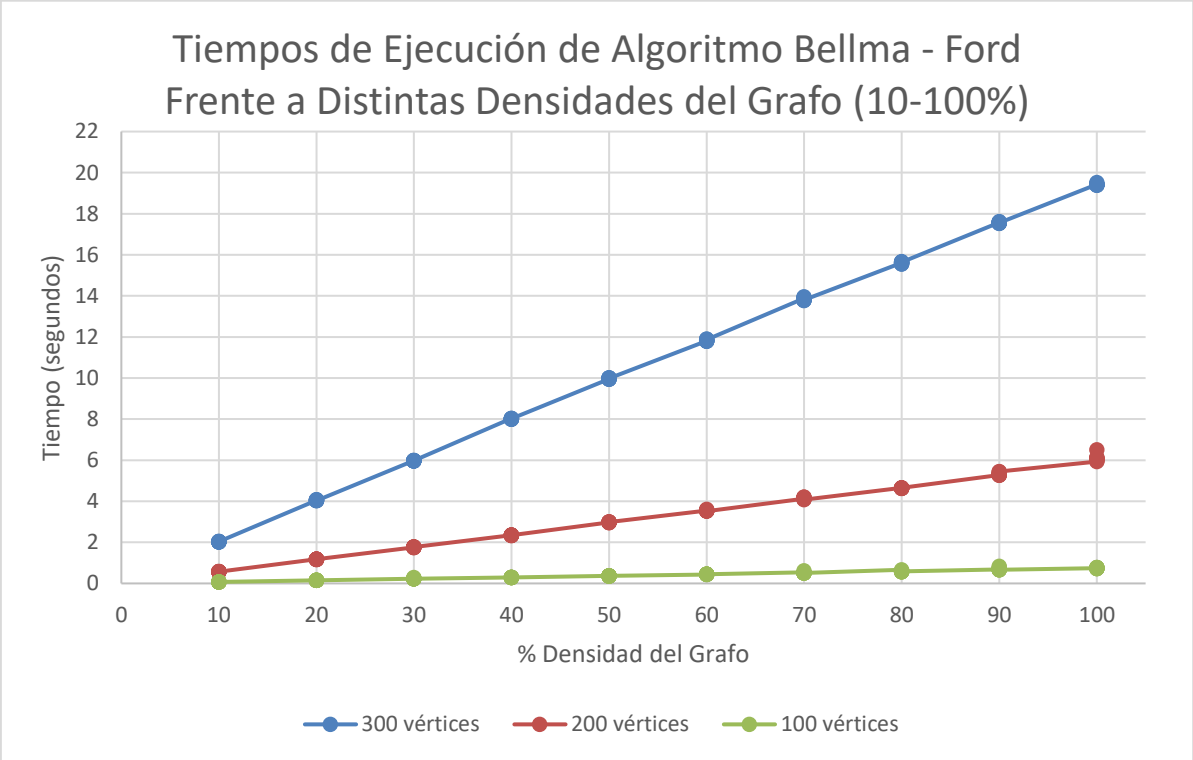

Gráfico 2: Algoritmo Bellman – Ford

Los tiempos como suponíamos aumentan cada vez que aumentamos tanto la cantidad de vértices como la cantidad de aristas del grafo. Esto se debe a que el algoritmo presenta un crecimiento exponencial de sus tiempos de ejecución cada vez que modificamos estas variables. Al igual que el algoritmo de Dijkstra, mientras más denso sea el grafo trabajado, más tiempo tardará en ejecutarse. En la Tabla 2 tenemos los tiempos promedios (para facilitar la lectura) de los resultados de la ejecución del algoritmo anterior.

Densidad (%)	100 nodos	200 nodos	300 nodos
100	0,748203	6,0904	19,4235
90	0,711533	5,3329	17,566
80	0,592079	4,6430	15,613
70	0,523817	4,1418	13,843
60	0,446484	3,5483	11,834
50	0,363917	2,9801	9,9785
40	0,291569	2,3466	8,013
30	0,232874	1,7635	5,9749
20	0,143752	1,1763	4,050
10	0,075069	0,572	2,0324

Tabla 2: Tiempos promedio de Algoritmo Bellman - Ford

Si vemos la Tabla 1 del algoritmo de Dijkstra observaremos medias bastante similares con respecto al algoritmo de Bellman – Ford y es porque como mencionamos, la estructura, la forma en la que trabajan ambos algoritmos, son bastante similares y buscan cumplir el mismo objetivo. Se han comparado dos grafos completos (cada par de vértices está conectado por una arista) de 300 vértices tanto para los resultados obtenidos en el algoritmo Bellman – Ford y su antecesor Dijkstra como se puede observar en el Gráfico 3.

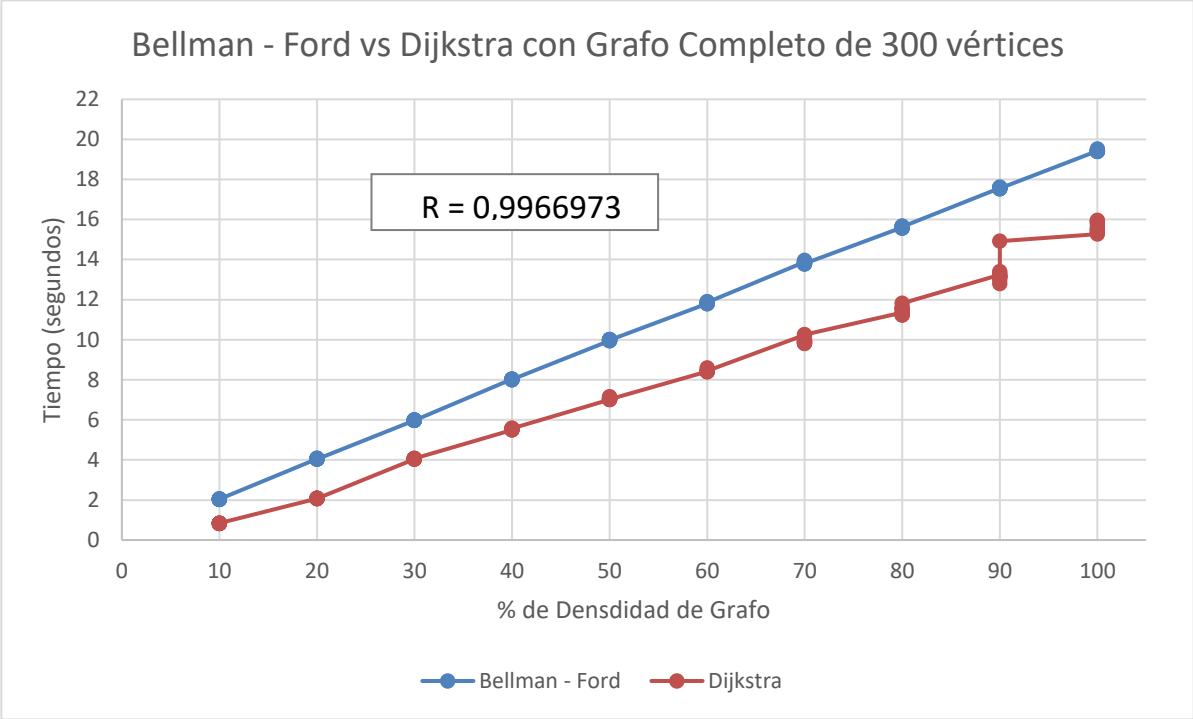

Gráfico 3: Algoritmo Bellman – Ford vs Dijkstra

Como observamos, sus medias y líneas de dispersión tienden a ser bastante similares en comportamiento. Esto se puede comprobar gracias al Coeficiente de Correlación calculado el cual es de 0,99 el cual es cercano a 1. De esta forma se puede decir que ambos presentan una relación lineal casi perfecta y por lo tanto sus medias siempre tenderán a ser parecidas. No es algo tan extraño ya que funcionan ambos algoritmos bajo el mismo principio de encontrar el camino más corto entre vértices.

Ahora bien, si analizamos la complejidad del algoritmo Bellman – Ford nos daremos cuenta de que el peor caso en tiempos de ejecución siempre se dará cuando el grafo sea más denso. Si recordamos la complejidad de O(VE) con *V* equivalente a la cantidad de vértices y *E* a la cantidad de aristas, podríamos buscar el peor caso cuando aumentamos la cantidad de aristas del grafo. El mínimo de aristas que puede tener el grafo para que sea conexo (que todos sus vértices estén conectados por al menos una arista a otro vértice) es de V – 1 pero ¿Qué ocurre con la máxima cantidad con la que podrá trabajar nuestro algoritmo? Si tuviéramos un grafo de 5 vértices y quisiéramos que todos estuviesen conectados con todos, nos daríamos cuenta de que desde cada uno de los vértices debiésemos tener sí o sí 4 aristas salientes conectadas cada una a un vértice distinto lo cual equivale a V – 1 aristas de salida de cada vértice. Si tenemos 5 vértices (V = 5 para este caso) y sabemos que la cantidad de aristas salientes de cada uno es de V – 1, podríamos detectar que la cantidad total de caminos sería de:

$$V \cdot (V - 1)$$

Si lo que buscamos es no contar aquellas aristas repetidas la cantidad máxima de aristas incluidas en un grafo completo y conexo es de:

$$\frac{V\,(V-1)}{2} = \frac{V^2 - V}{2} = E_{\text{máximo}}$$

Si conocemos que la complejidad algorítmica de Bellman – Ford es de O(VE) y que el valor de E como cantidad máxima de aristas es el expresado en la ecuación anterior tendremos lo siguiente:

$$O(V \cdot E) = V \cdot \frac{V^2 - V}{2} = \frac{V^3 - V^2}{2} = O\left(\frac{V^3 - V^2}{2}\right)$$

Como hemos realizado anteriormente y con la ayuda de la Notación Asintótica podemos simplificar la complejidad eliminando las contantes (2) y valores menos significativos (V^2) quedando el algoritmo Bellman – Ford con una complejidad máxima de $O(V^3)$.

Lo que primero se puede observar es que el algoritmo Bellman – Ford tiene una complejidad superior al algoritmo de Dijkstra en su peor caso (complejidad cúbica versus cuadrática). Segundo, podemos determinar que esta complejidad corresponde a su peor caso ya que el grafo cuando se encuentra completo tiene la cantidad máxima de aristas posibles entre sus vértices lo que hace que el algoritmo tarde más en ejecutarse.

De esta manera, con los datos obtenidos, graficados en el Gráfico 2 y calculada la complejidad, comprobamos que la hipótesis se cumple la cual nos decía que mientras más denso se hiciera el grafo, la complejidad algorítmica de Bellman – Ford encontraría su peor caso con $O(V^3)$ para un grafo denso completo y conexo.

4.3 Hipótesis 3: Cuando se requiere calcular las distancias más cortas entre todos los vértices del grafo, independiente de la cantidad de vértices, el algoritmo Floyd – Warshall mejora el tiempo de ejecución de la fuerza bruta

El algoritmo Floyd-Warshall es un algoritmo de ruta más corta para grafos. Al igual que el algoritmo de Bellman-Ford o el algoritmo de Dijkstra, calcula la ruta más corta en un grafo conexo. Sin embargo, Bellman-Ford y Dijkstra son algoritmos de una sola fuente y de ruta más corta. Esto significa que solo calculan la ruta más corta desde un solo vértice inicial. Floyd-Warshall, por otro lado, calcula las distancias más cortas entre cada par de vértices en el grafo de entrada.

El algoritmo Floyd-Warshall es un ejemplo de programación dinámica. Desglosa el problema en subproblemas más pequeños, luego combina las respuestas a esos subproblemas para resolver el gran problema inicial. La idea es esta: o la ruta más rápida de A a C es la ruta más rápida encontrada hasta ahora de A a C, o la ruta más rápida de A a B más la ruta más rápida de B a C.

Floyd – Warshall básicamente funciona con una matriz de adyacencia V · V. Considera cada vértice y decide cuál sería la ruta más corta que pudiera atravesar ese vértice. Esta es una comparación de tiempo constante y una operación de inserción (en una matriz de dos dimensiones) llevada a cabo para todos los elementos v^2 de la matriz.

Esto debe realizarse para cada vértice. Por lo tanto, la complejidad del tiempo resulta ser $O(V^3)$ pero con un valor constante muy pequeño, lo que lo hace extremadamente viable durante la implementación. Una de las particularidades de este algoritmo es que sin importar la densidad del grafo sus tiempos tienden a permanecer constantes.

Al igual que los algoritmos anteriores, utilizaremos grafos con distintas cantidades de vértices (50, 100, 200 y 300) para realizar las comprobaciones de tiempos de ejecución y con variaciones de la densidad de aristas, como se puede observar en el Gráfico 4:

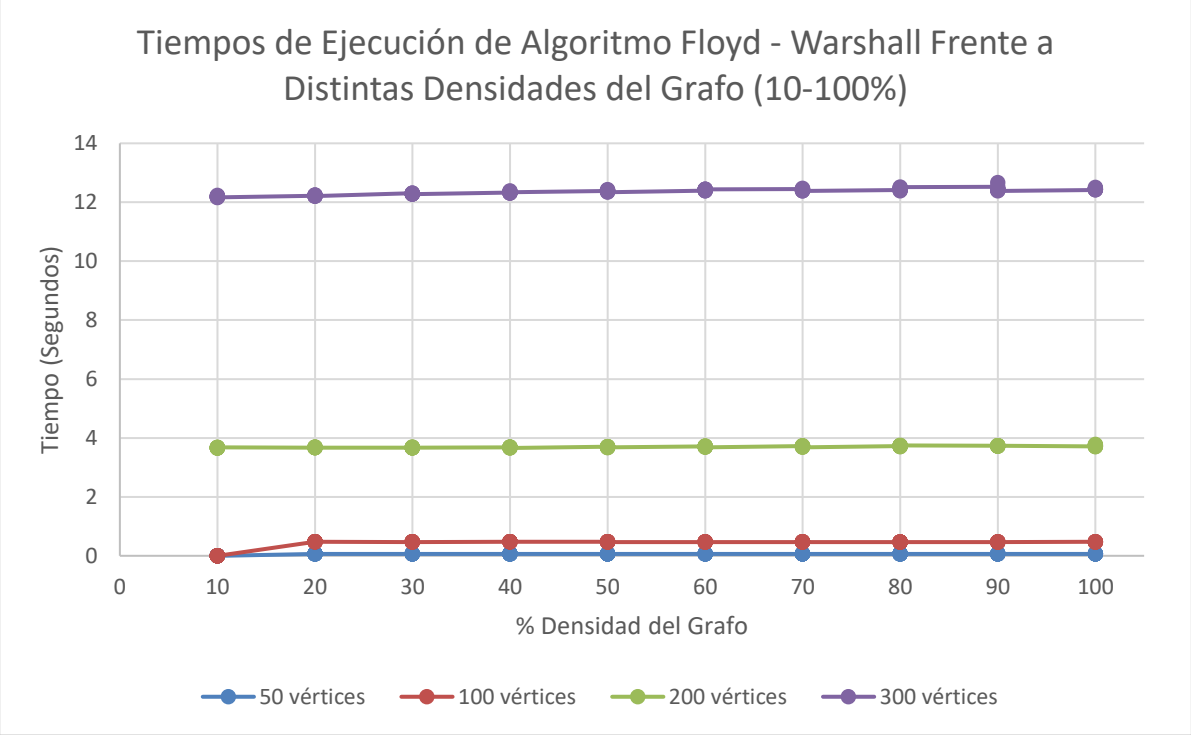

Gráfico 4: Algoritmo Floyd – Warshall

Podemos realizar una comparación con los algoritmos de Dijkstra y Bellman – Ford en sus tiempos de ejecución para ver como se desempeñan con la utilización de grafos con vértices con las cantidades de 20, 50, 100, 200, y 300 y con una densidad del 100% lo que significa que el grafo se encuentra completo y con la cantidad máxima posible de aristas como se observa en el Gráfico 5:

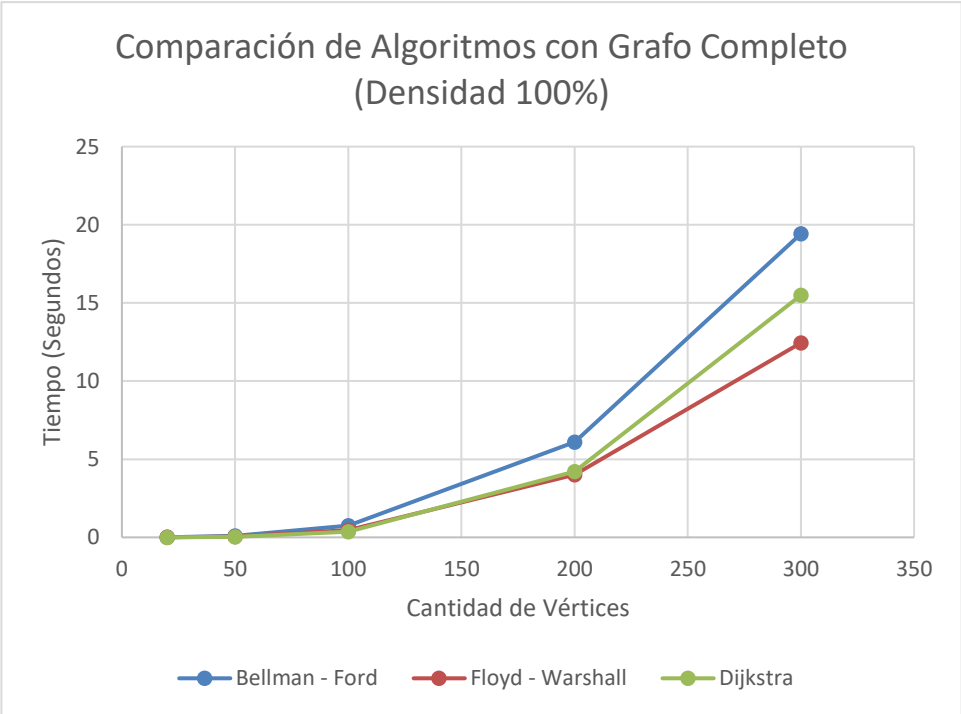

Gráfico 5: Comparación de los tiempos de ejecución bajo distintas cantidades de vértices y un grafo completo

Se puede observar una pequeña diferencia de desempeño del algoritmo de Floyd – Warshall con respecto a los otros. A pesar de esto, no es la forma apropiada de compararlos por lo que los resultados obtenidos en el Gráfico 5 no son los más indicados para el estudio y una posterior conclusión de los resultados. ¿A qué se debe esto? Tanto Dijkstra como Bellman – Ford no están realizando el mismo trabajo que el algoritmo Floyd – Warshall.

Si tomamos como base lo anteriormente señalado y nos enfocamos en nuestra hipótesis deberemos comprobar cómo se comporta el algoritmo de Floyd frente a los dos algoritmos de fuerza bruta: Dijkstra y Bellman – Ford. El problema de esta comparación es que los dos últimos algoritmos mencionados funcionan de manera distinta a Floyd – Warshall ya que requieren un vértice inicial para calcular todos los caminos más cortos al resto de pares de nodos del grafo. Una vez que calculan las rutas mínimas a todos los vértices desde un vértice inicial, los algoritmos dejan de ejecutarse. Lo anterior no ocurre para el caso de Floyd – Warshall donde calcula los mínimos caminos de todos los vértices del grafo. Es por esto que los tiempos calculados anteriormente para los primeros dos algoritmos trabajados en este informe resultan inútiles. Como vemos en la Tabla 3, los tiempos calculados del algoritmo Floyd - Warshall después de haber encontrado todos los caminos más cortos de todos sus pares de vértices son:

Densidad (%)	50 nodos	100 nodos	200 nodos	300 nodos
100	0,0613015	0,4702605	3,80873747	12,4426426
90	0,059613	0,4665741	3,8090649	12,4591161
80	0,0615034	0,4669555	4,15988848	12,4411916
70	0,0608149	0,46880413	3,87126669	12,4193388
60	0,0604104	0,46871523	3,79120761	12,4129037
50	0,0615391	0,47098312	3,76209092	12,3798834
40	0,0612429	0,47439281	3,8631197	12,3333343
30	0,0607379	0,47153873	4,07504388	12,287178
20	0,0614687	0,47269049	3,73799553	12,2201275
10	5,36E-05	0,00017629	0,00037214	12,1877716

Tabla 3: Tiempos promedios de Algoritmo Floyd - Warshall

Para resolver este problema se ha requerido modificar los algoritmos para que calculen no solo las distancias de un nodo inicial al resto de vértices, sino que lo haga también para todo el resto de ellos.

Recordemos que la complejidad del algoritmo Dijkstra con la no utilización de montículos es de $O(V^2 + E)$ y para el caso de Bellman – Ford es de $O(VE)$. Estas complejidades se aplican únicamente para cuando los algoritmos trabajan con un solo nodo inicial. Para realizar el cambio mencionado anteriormente, los algoritmos deberán ir cambiado en cada iteración el vértice inicial desde 0 hasta V – 1, por lo que, si observamos los algoritmos deberán realizar V iteraciones extras para obtener todos los caminos más cortos de todos los vértices. Por lo tanto, las nuevas complejidades quedan de la siguiente forma:

$$O(V^2 + E) \cdot V = V^3 + VE = O(V^3)$$

Para el caso del algoritmo Dijkstra

$$O(V \cdot E) \cdot V = O(V^2 \cdot E)$$

Para el caso del algoritmo Bellman – Ford

De esta manera, ambas complejidades cambian al incorporar la modificación señalada por lo que los tiempos de ejecución aumentarán. En el caso de Dijkstra, su complejidad puede mantenerse constante o mejorar si es que se utiliza montículos ya que representa su mejor caso. En cambio, Bellman – Ford, si el grafo utilizado es completo, la cantidad de aristas máximas alcanzarían la cantidad de $(V^2 – V) / 2$, por lo que la complejidad total pasaría a ser de

$$O(V^4 – V^3) / 2 = O(V^4)$$

Dicha complejidad para el nuevo algoritmo Bellman – Ford traerá consigo tiempos mucho más altos que los obtenidos por Dijkstra y Floyd – Warshall por lo que se puede inferir que no debiese ser el algoritmo óptimo para calcular los caminos mínimos de todos los vértices de un grafo, como se observa en el Gráfico 6. Se han comparado los algoritmos de fuerza bruta modificados y el algoritmo de Floyd – Warshall con grafos de 100 vértices y cambios en la densidad que va desde el 10% (grafo disperso) hasta el 100% (grafo completo).

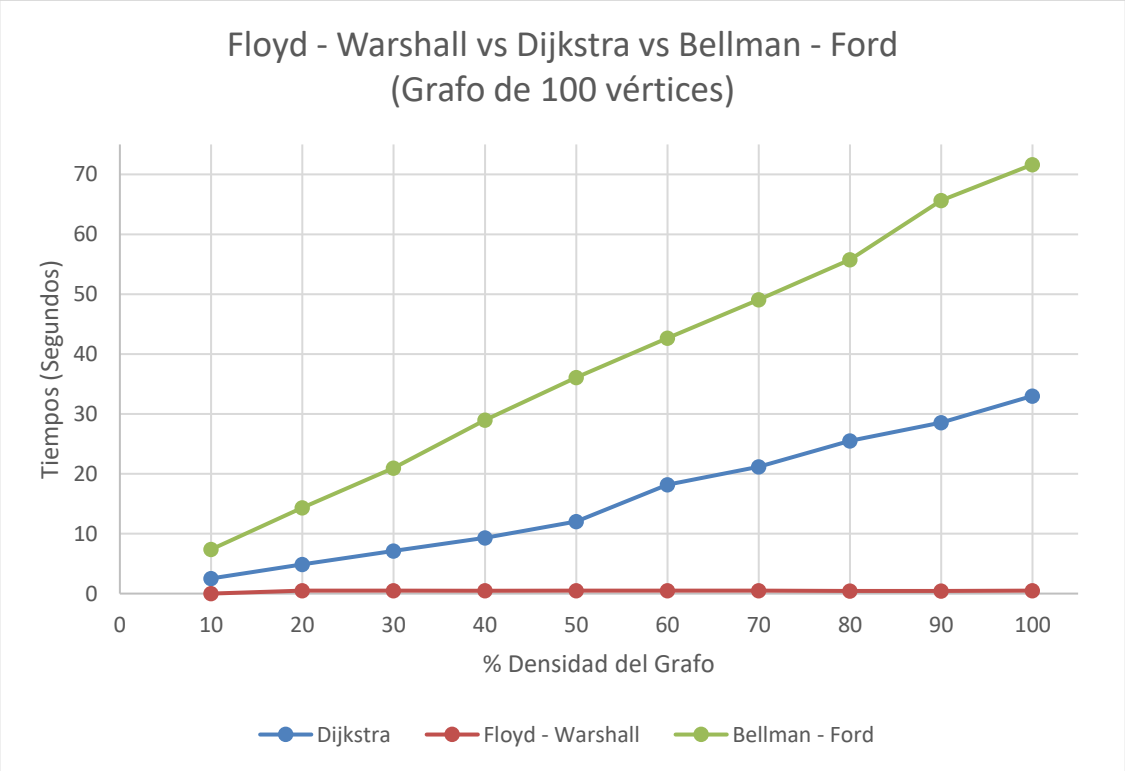

Gráfico 6: Comparación de algoritmos con un grafo de 100 vértices

Densidad (%)	Bellman - Ford	Dijkstra	Floyd - Warshall
100	71,6430242	33,0093431	0,4702605
90	65,6346656	28,5434938	0,4665741
80	55,7477955	25,5302091	0,4669555
70	49,1032329	21,1568364	0,46880413
60	42,6574196	18,163367	0,46871523
50	36,1002734	12,0219445	0,47098312
40	28,9778698	9,28934546	0,47439281
30	20,9607139	7,10910323	0,47153873
20	14,3386227	4,90345652	0,47269049
10	7,37349791	2,53160851	0,00017629

Tabla 4: Promedios de algoritmos con la utilización de un grafo de 100 vértices

Utilizando grafos con pocos vértices se puede ver la diferencia en el rendimiento que tiene el algoritmo Floyd – Warshall con respecto a los otros. Esto sumado a las complejidades que alcanzan los algoritmos de Dijkstra $O(V^3)$ y Bellman – Ford $O(V^4)$ en su peor caso cuando son transformados para trabajar al igual que Floyd, es que se puede comprobar que el algoritmo de Floyd – Warshall $O(V^3)$ mejora los tiempos de ejecución con respecto a los algoritmos de fuerza bruta mencionados. Esto ocurre al menos cuando se utilizan grafos con bajas cantidades de nodos por lo que no se puede aseverar que se siga cumpliendo esta tendencia cuando son utilizados grafos con mayores cantidades de vértices y aristas.

Es importante aclarar que el rendimiento de los algoritmos varía en la forma en la que son implementados por lo que los tiempos de ejecución pueden estar sujetos a los cambios en la estructura de los algoritmos, aunque funcionen de manera parecida. De igual forma, con todo lo anteriormente mencionado, se puede comprobar que la hipótesis es cierta.

5 Conclusiones

Los algoritmos de búsqueda de caminos más corto en grafos son y ha sido una de las investigaciones que más se han realizado en los últimos años. Con el auge de tecnologías de geolocalización y mapas de rutas como los utilizados por Google Maps o empresas de servicio automovilístico como UBER es que se ha hecho indispensable la búsqueda de encontrar nuevos algoritmos o perfeccionar los ya existentes.

Para nuestro caso, después de evaluar tres algoritmos se concluye que el algoritmo Bellman – Ford presenta el peor rendimiento en todos los casos posibles: cuando se utiliza un algoritmo para encontrar el camino más corto desde un único vértice de origen o cuando se utiliza para encontrar las rutas mínimas a partir de todos sus nodos. Esto ocurre ya que depende de la cantidad de aristas que tenga el grafo, siendo en la mayoría de las ocasiones mayor a la cantidad de vértices. Recordemos que debemos utilizar grafos conexos por lo que la cantidad mínima de aristas presente en cualquiera de ellos será de V – 1, con la letra V representando la cantidad de vértices total del grafo. A medida que el grafo se hace más denso, la cantidad máxima de aristas que alcanzaba para que el algoritmo Bellman – Ford pueda trabajar era de $(V^2 – V) / 2$, por lo que en su peor caso como comprobamos en los tiempos de ejecución, el algoritmo llegaba a alcanzar una complejidad de $O(V^3)$, superior a la de su antecesor Dijkstra. Es por esto por lo que presentaba un peor rendimiento al momento de ejecutarse. A pesar de eso, este algoritmo permitía detectar cuando las ponderaciones del grafo tenían costes negativos, a diferencia del algoritmo Dijkstra.

El algoritmo Dijkstra dependía netamente de sus vértices. A pesar de que a medida que el grafo se hacía más denso, cuando la cantidad de aristas que conectaban los vértices aumentaba, los tiempos de ejecución también lo hacían. Esto debido a que su complejidad, sin aplicar el método de notación asintótica, es de $O(V^2 + E)$, por lo que ante cualquier cambio en sus aristas su tiempo también se vería afectado, aunque en menor medida en comparación con el algoritmo de Bellman – Ford.

El algoritmo Floyd – Warshall resultó ser el más óptimo al momento de trabajar con grafos en donde se buscaba la distancia mínima entre todos los pares de vértices, problema que el algoritmo de Dijkstra y Bellman – Ford no podían resolver con su estructura inicial ya que solo calculaban este problema con la utilización de un vértice inicial. Debido a esto, para poder realizar la comparación, se tuvo que modificar los algoritmos de tal manera que pudiesen desempeñar el mismo trabajo que hacía el algoritmo de Floyd. Aún así, este algoritmo lograba encontrar los caminos mínimos de todos sus vértices en menor cantidad de tiempo. Esto se podía producir debido a que la modificación realizada en los algoritmos de Dijkstra y Bellman – Ford no fuese la más indicada, o que la utilización de grafos con baja cantidad de vértices (100 vértices para esa comparación) no fuese la más apropiada.

6 Referencias

I. Cormen, Thomas (2013). Algorithms Unlocked. 7th ed. Cambridge: The MIT Press.

II. Wang, Xiang. (August 2013). Analysis of the Time Complexity of Strassen Algorithm. Paper: IEEE.

III. Vassilevska Williams, Virginia (Nomvember 2011). A Breakthrough On Matrix Product, Paper: IEEE.

IV. Ali Khan, Zafar (August 2016). Comparison of Dijkstra's Algorithm with other Proposed Algorithms, Paper: IEEE.

V. Baggar Chitra (February 2014). A survey of Bellman – Ford algorithm and Dijkstra algorithm for finding shortest path in GIS application. Paper: IEEE.