# Reporte tarea 9 - Programación y algoritmos

Erick Salvador Alvarez Valencia, Centro de Investigación en Matemáticas, A.C.

***Index Terms*—Camino más corto, Gráficos por computadora.**

## I. Introducción

En el presente reporte se analizará la implementación del algoritmo de dijsktra, el cual fue utilizado en la presente tarea para encontrar el camino más corto en un grafo dirigido, y posteriormente dibujar el grafo y trazar la ruta del mismo usando la librería de Cairo Graphics. Para complementar de manera apropiada el contenido del texto se brindará una pequeña descripción del algoritmo, su implementación, algunas pruebas realizadas y algunas conclusiones.

mds

Octubre 30, 2017

## II. Algoritmo de Dijkstra

### *II-A. Descripción*

En la presente tarea se trató el problema de la ruta más corta con un grafo dirigido con pesos (distancias) positivos, para tratar este problema se utilizó el algoritmo de dijkstra, dicho algorito usa una técnica *Greedy* para encontrar la ruta más corta entre un nodo y todos los demás. Por técnica Greedy o Voráz nos referimos a que se utiliza el principio de que para que un camino sea óptimo, todos los caminos que contiene también deben serlo.

La idea del algoritmo es en cada iteración ir tomando el nodo con menor distancia al origen, al cual denotamos como $u$, después se marca al nodo $u$ como visitado y se recorren sus vecinos, si estos no han sido visitados aún se evaluan sus distancias. Sea $v$ un vecino del nodo $u$, $w$ la distancia entre $u$ y $v$ y sea $dist(v)$ la distancia del nodo $v$ al nodo origen, por cada vecino $v$ de $u$, si $dist(v) > dist(u) + w$ entonces hacemos $dist(v) = dist(u) + w$. El proceso anterior se llama paso de *relajación*, al realizar el paso de relajación una y otra vez hasta que todos los nodos hayan sido visitados, entonces el algoritmo asegurará que para todo nodo $v$ con algún camino al origen, $dist(v)$ será mínima. A continuación se mostrará la implementación del algoritmo de dijkstra:

**Algorithm 1** Ruta más corta.

```
1:  procedure DIJKSTRA(graph, ori)
2:      dist ← Vector de distancias con respecto a ori.
3:      vis ← Vector de nodos visitados.
4:      fill(dist, INF).
5:      pq ← Cola de prioridad con prioridad mínima.
6:      pq.push(ori, 0).
7:      while pq no esté vacía do
8:          u ← pq.front().first
9:          if !vis[u] then
10:             vis[u] = true
11:             for i ← 1 to graph[u].size() do
12:                 v ← i
13:                 if u == v then
14:                     continue.
15:                 w ← graph[u][v].
16:                 if !vis[v] then
17:                     if dist[v] > dist[u] + w then
18:                         dist[v] = dist[u] + w.
19:                         pq.push(v, dist[v]).
20:     return dist
```

En el Algoritmo 1. podemos ver que se usa una cola prioridad la cual contiene pares para ir obteniendo el nodo con la distancia mínima, los pares están compuestos por el índice del nodo y su distancia con respecto al origen. De esta forma la cola comparará las distancias y nos devolverá el nodo asociado a ellas. Esta no es la única forma de realizar un dijkstra, cualquier estructura que almacene los datos con respecto a un orden nos servirá, tal es el caso del árbol binario de búsqueda, o en el caso más extremo, podemos ir almacenando los datos en un vector e irlo ordenando en cada iteración, pero esto resultaría costoso computacionalmente.

Hay una adaptación que se puede realizar al algoritmo, y la misma implica la obtención del árbol de camino mínimo con respecto al nodo de origen, la idea central es usar un vector de tamaño $V$ donde $V$ es el número de vértices, ese vector indicará para cada nodo $v$ cuál es su antecesor. Cada vez que se realice el paso de relajación podemos actualizar el vector de predecesores de la siguiente forma: $prev[v] = u$ ya que estamos seguros que se tomó esa ruta porque minimiza la distancia. De esta forma podremos obtener la ruta de cualquier nodo al origen con el siguiente algoritmo:

**Algorithm 2** Obtención de ruta.

```
1: procedure GetPath(prev, dest)
2:     if prev[dest] ≠ -1 then
3:         GetPath(prev, prev[dest]).
4:     else
5:         print(dest).
```

## III. Dibujo del grafo y el camino mínimo

Para el dibujo del grafo y su camino mínimo se utilizó la librería *Cairo Graphics* la cual es una biblioteca gráfica de la API GTK+ usada para proporcionar imágenes basadas en gráficos vectoriales. Aunque Cairo es una API independiente de dispositivos, está diseñado para usar aceleración por hardware cuando esté disponible. Cairo posee distintas funciones que ayudan a realizar diferentes gráficas, ya sean figuras geométricas, modificación de imágenes, texto, degradados, etc. Con Cairo se generaron distintas imágenes las cuales contenian grafos hechos a base de círculos, rectas y texto. La idea general era crear el grafo en donde los nodos se distribuían de forma circular, esto por la facilidad de acomodo de los mismos ya que otras topologías implican más esfuerzo por parte de la programación.
Para lo anterior debemos dibujar los nodos en función a un ángulo que se irá aumentando en base a una valor obtenido de una discretización uniforme, posterior a eso se usan los valores de la parametrización de la circunferencia con radio $r$ y ángulo $\theta$. Lo anterior se puede realizar de la siguiente forma:

**Algorithm 3** Dibujo de los nodos.

```
1: procedure DrawNodes(nodes, xIni, yIni, r)
2:     theta ← 0.
3:     dTheta ← 2 * π / nodes.size()
4:     for i ← 1 to nodes.size() do
5:         x ← xIni + r * cos(theta)
6:         y ← xIni + r * sin(theta)
7:         DrawCircle(x, y, radio=40)
8:         theta = theta + dTheta
```

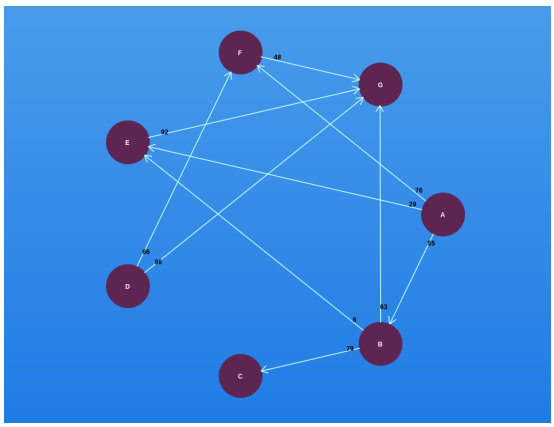
Con lo anterior podremos dibujar los nodos en forma circular y éstos se adaptarán en función al número de los mismos. Cairo provee funciones para dibujo de arcos en donde podremos indicar que dibuje un círculo con ángulo completo. Para el dibujo de aristas se pueden usar las coordenadas de los nodos como punto de partida, la idea es trazar una línea que va desde las coordenadas del nodo $A$ al nodo $B$ pero como se trazaría parte de la recta encima del círculo, hay que empezar desde la orilla del mismo, usando la parametrización del círculo podremos determinar el punto de inicio y el punto final de la siguiente forma: $x_{Ini} = x_0 + r * cos(\theta)$, $y_{Ini} = y_0 + r * sin(\theta)$, $x_{End} = x_1 - r * cos(\theta)$, $y_{End} = y_1 - r * sin(\theta)$. Una vez dibujadas las líneas podemos dibujar las flechas usando un diferente ángulo $\widehat{\theta} = valor\ arbitrario$ y la misma parametrización del círculo.
El valor de los nodos y los pesos de las aristas también se pueden escribir de forma fácil usando las funciones de Cairo para escritura de texto, la idea es guardar los puntos donde se colocaron los círculos y usarlas al escribir lo anteriormente dicho.

## IV. Generación aleatoria

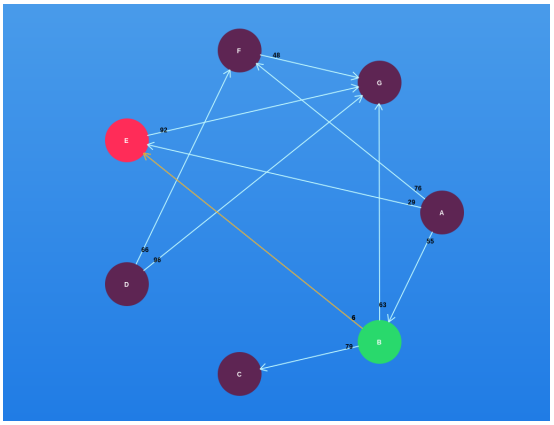
En el programa se trabajó con un grafo dirigido y se almacenó en una lista de adyacencia, la forma de creación fue parcialmente aleatoria ya que se siguió una estructura de matriz triangular, esto para que el grafo fuera dirigido.
La idea es recorrer todos los nodos de manera ordenada y para cada nodo $i$ se recorren todos los nodos desde $i + 1$ hasta $n$, después se decide si existirá conexión mediante una determinada probabilidad $P$, en caso de existir conexión se genera un peso aleatorio $w$ que va desde 1 hasta un cierto límite $W$ y posteriormente se hace la conexión. Esto provocará que se realice las conexiones tipo matriz triangular superior, lo malo de este algoritmo es que los últimos nodos tendrán muy poca probabilidad de tener conexiones salientes dado que siempre se empieza a iterar desde $i + 1$, de hecho el último nodo no tendrá conexiones.

## V. Ejecución y pruebas realizadas

A continuación se mostrarán dos resultados de ejecuciones realizadas al programa, una con un grafo de 7 nodos y la última con un grafo de 13 nodos.

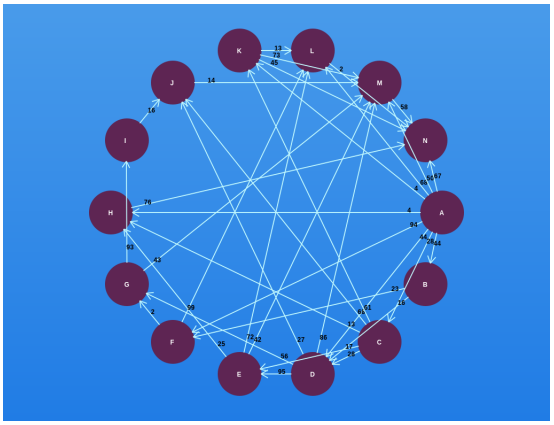



(a) Figura 1. Grafo generado aleatoriamente con 7 nodos.

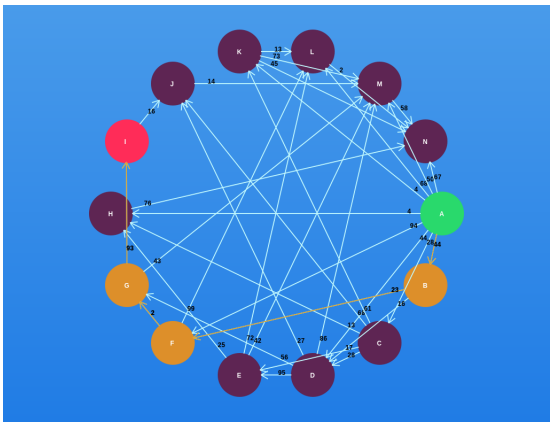



(b) Figura 2. Ruta mínima del grafo generado en la Figura 1.

Podemos ver que en el grafo generado en la Figura 1. todos los nodos son conexos y se eligió como nodos inicial y final B y E, la ruta más corta entre ellos es la que los conecta directamente, de hecho es la única existente por construcción.




(c) Figura 3. Grafo generado aleatoriamente con 14 nodos.




(d) Figura 4. Ruta mínima del grafo generado en la Figura 3.

Para el grafo generado en la Figura 3. se puede notar que el número de aristas aumentó drásticamente, esto porque la probabilidad de conexión entre los nodos $i, j$ no es tan baja. Para la ruta mínima se eligieron como nodos inicial y final el A y el I. La ruta está constituida por los nodos A, B, F, G, I, para este caso se puede notar que existen otras rutas alternativas pero se puede comprobar que la mostrada es la mínima.

## VI. Conclusiones

Para este problema se analizó un algoritmo para búsqueda del camino más corto entre un nodo y todos los demás en un grafo dirigido, a su vez se usó la librería de Cairo Graphics para dibujar el grafo en formato PNG y vectorial(PS). Se proponen dos mejoras para el programa, la primera es en relación ala generación aleatoria del grafo, al estar trabajando con grafos dirigidos, este algoritmo de generación nos da resultados factibles, mas aún hay un gran problema y este se comentó anteriormente, los últimos nodos tienen muy poca probabilidad de tener conexiones salientes, y el último nodo nunca tendrá estas. Para mejorar la generación de grafos se propone el no usar matrices triangulares y, en lugar de ello generar árboles de expansión mínima con el algoritmo de Kruskal o Prim, posteriormente generar ciclos de manera aleatoria. Esto en primera no generará grafos completamente conexos y con aristas mejor distribuidas.

Otra mejora es en relación al dibujo del grafo, principalmente en la impresión de los pesos de las aristas, se puede ver que los mismos siempre están al comienzo de las aristas y por encima de ellas, cuando el grafo es grande y tiene muchas aristas, se puede notar que los pesos se ven muy juntos. Como solución se propone darles un aumento ya sea en la componente $x$ o $y$ dependiendo de la pendiente de la arista asociada o su ángulo $\theta$. De esta forma los pesos no se encontrarán completamente arriba de las aristas y tendrán una mejor distribución.

Como nota final, hay que comentar que se usó la herramienta valgrind para el chequeo de la memoria usada en el programa, y aunque la misma mostró que existen errores de liberación de memoria, no se pudo determinar la fuente de los mismos ya que valgrind indicó que éstos provenían de librerías asociadas al sistema operativo.