Estructura de Datos y Algoritmos 2

Laboratorio Nro. 1: Grafos

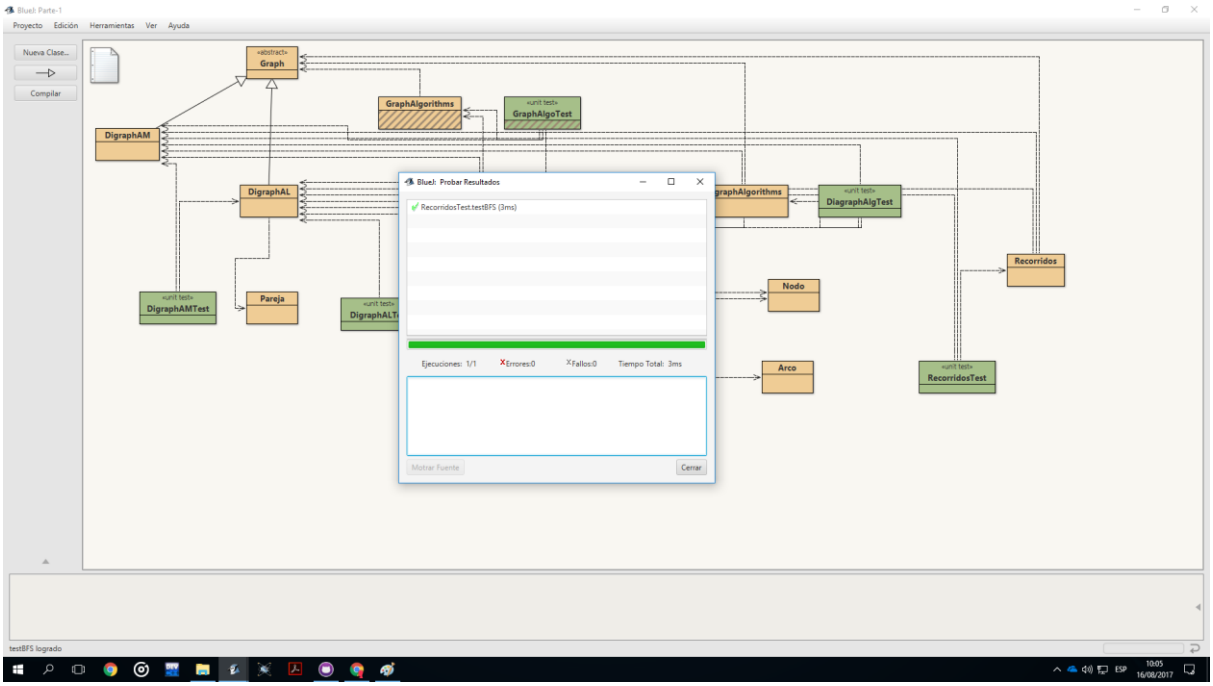
Docente: Dr. Mauricio Toro Bermúdez

Juan Gonzalo Quiroz Cadavid
Universidad Eafit
Medellín, Colombia
jquiro12@eafit.edu.co

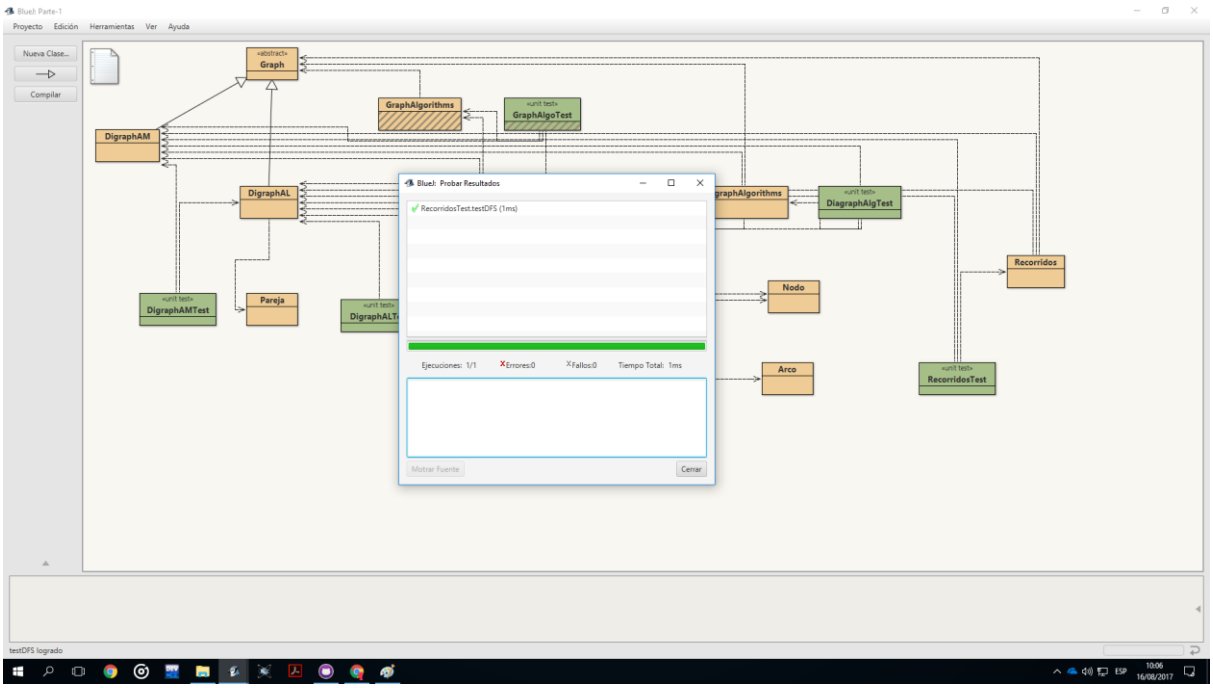
Alejandro Díaz Cano
Universidad Eafit
Medellín, Colombia
adiazc@eafit.edu.co

3.1 Incluyan una imagen de la respuesta de las pruebas del numeral 1.6 PARTE EN UNIÓN CON MI COMPAÑERO DE TRABAJO.

BFS

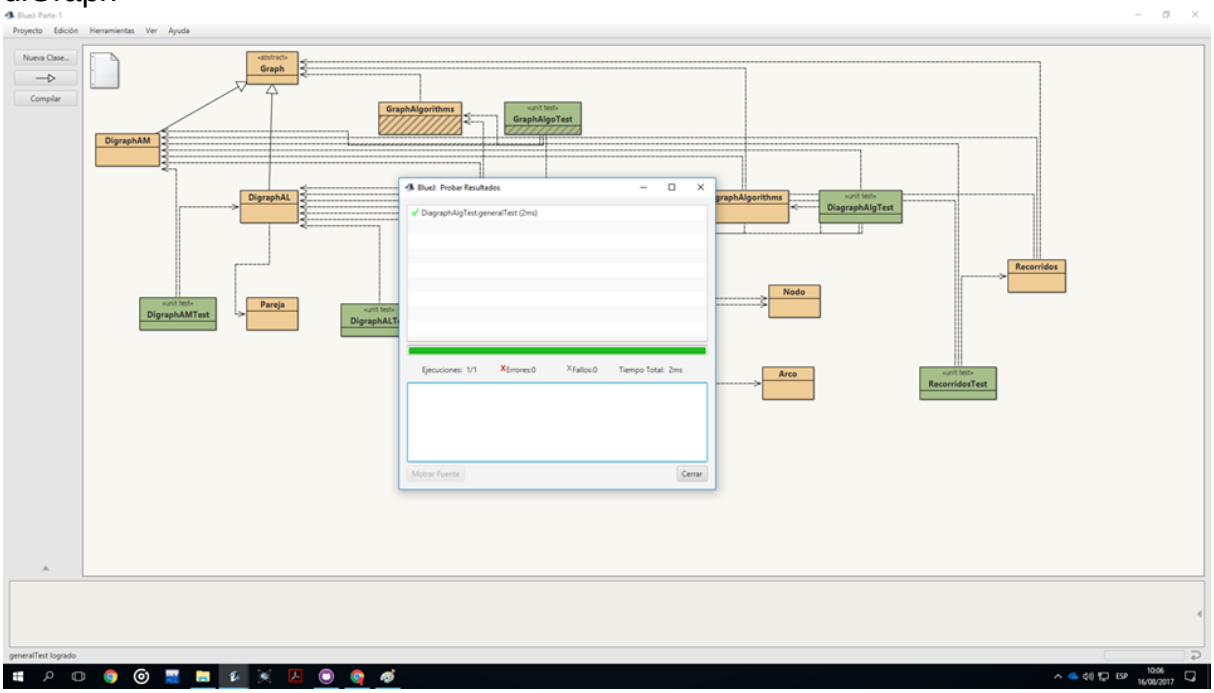


DFS



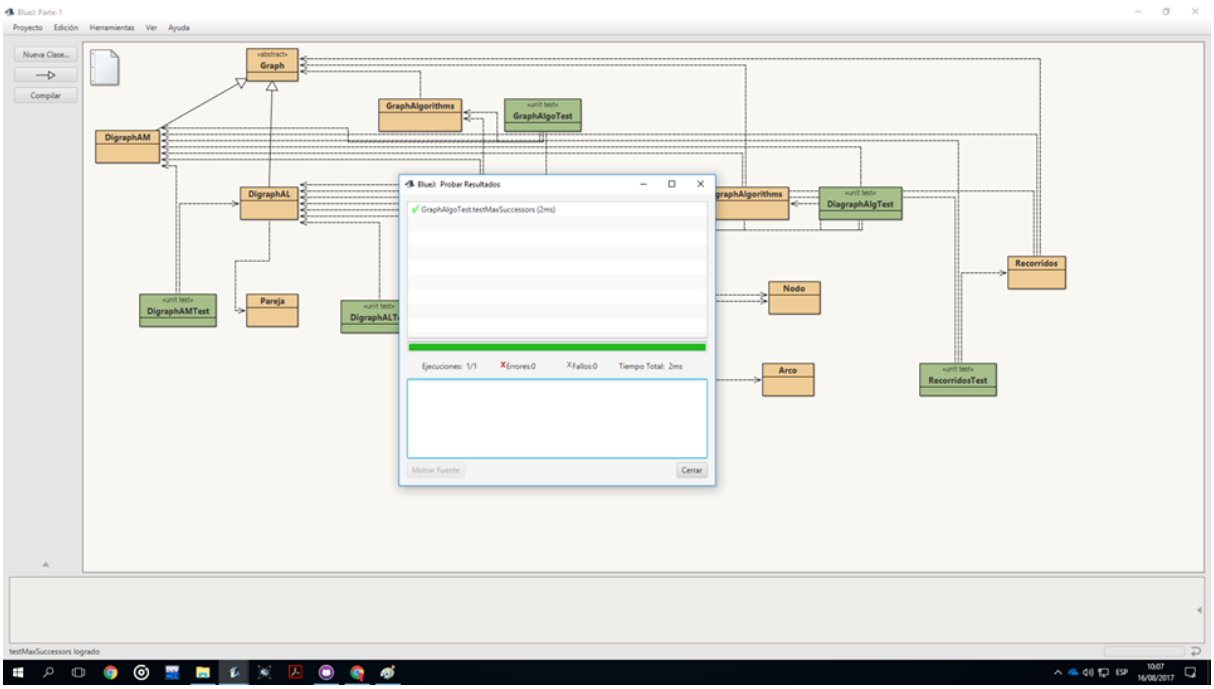
DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

GeneralTest de diGraph

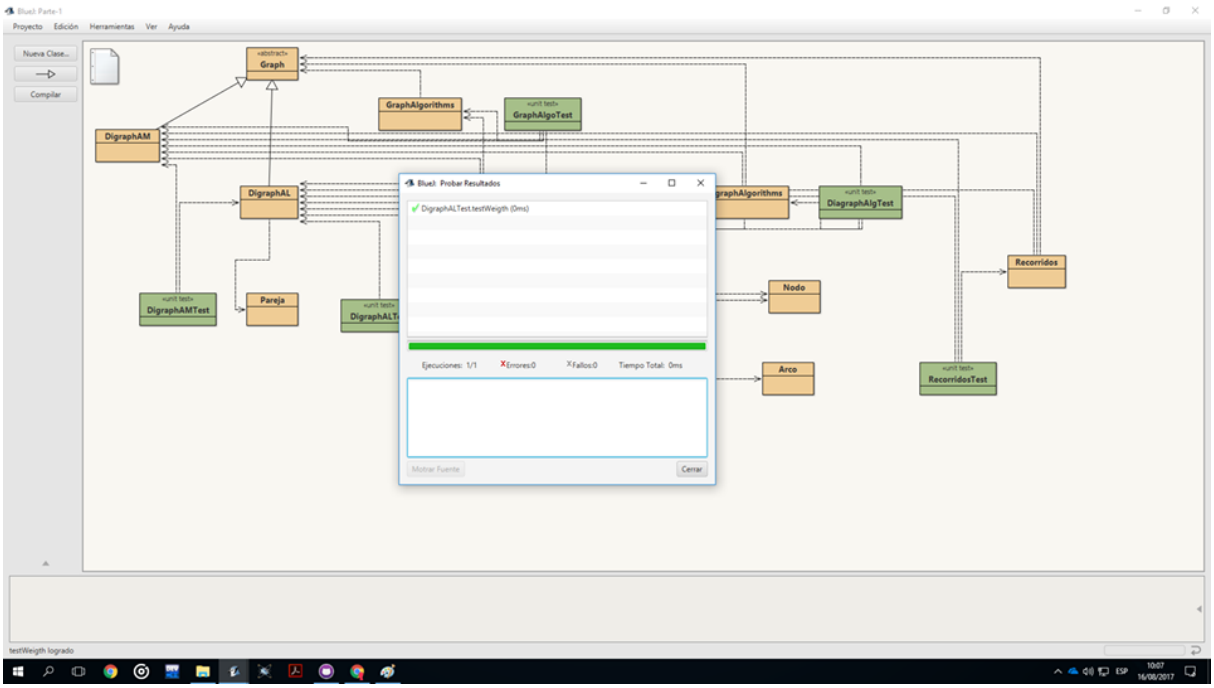


MaxSu

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

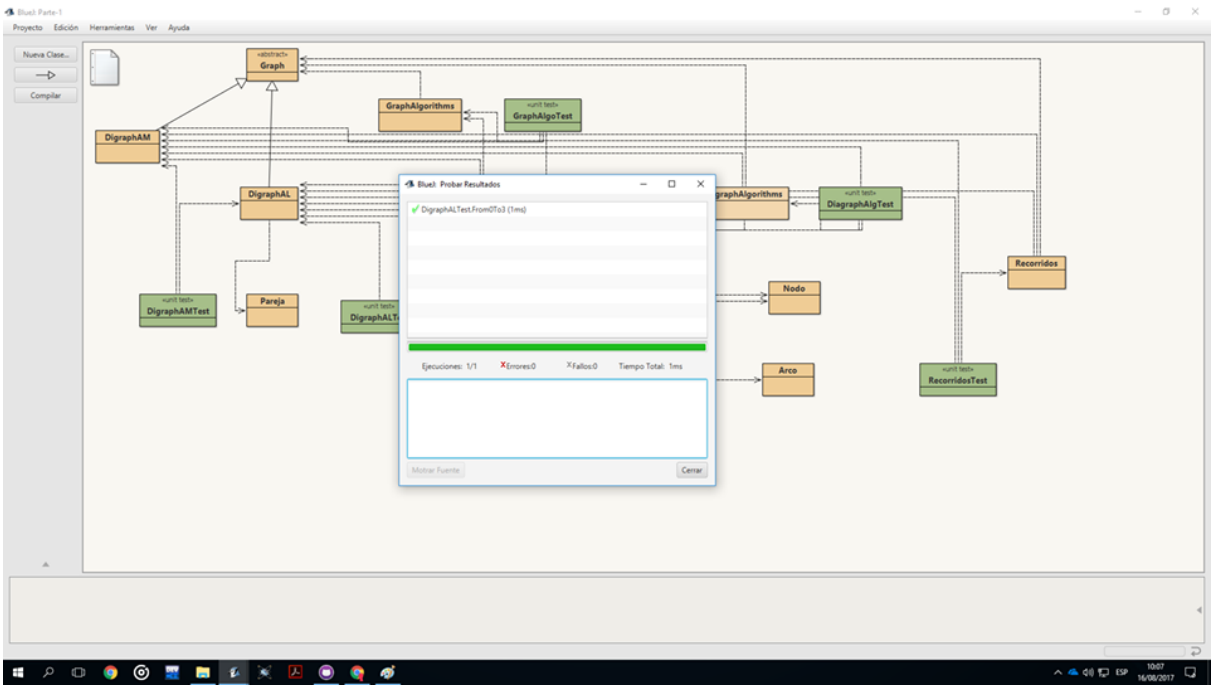


Test de Tamaño

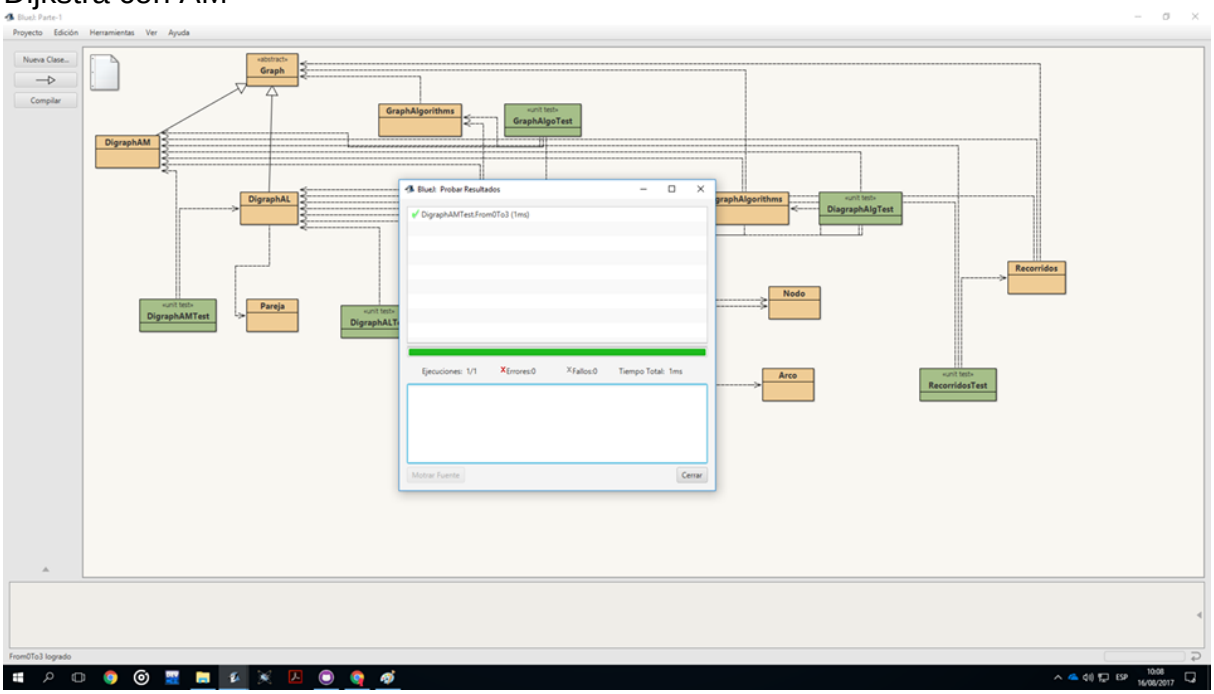


DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

Test Dijkstra



Dijkstra con AM



DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

3.2 Escriban una explicación entre 3 y 6 líneas de texto del código del numeral 1.1.
Digan cómo funciona, cómo está implementado el grafo con matrices y con listas
R// Tenemos una variable de tipo entera llamada graphAM es Matriz y le pasamos un parámetro de “n” x “n” El source es el índice del nodo y tenemos un destino al nodo que está conectado y obtenemos el valor. En el weight almacenamos el tamaño del arco entre ambos nodos y con el getSuccessors se obtienen los índices de los nodos sucesores y los introduce en un arraylist retornando el todos los índices de nodos sucesores.

3.3 ¿En qué grafos es más conveniente utilizar la implementación con matrices de adyacencia y en qué casos en más convenientes listas de adyacencia? ¿Por qué?

R//*La Matriz de adyacencia es más útil para grafos no dirigidos por que se tiene que recorrer todo el grafo para poder saber sus conexiones y así generar sus conexiones completas y sus costos (algo parecido al algoritmo de Dijkstra, pero lleno completamente).

*Las “Listas de adyacencia” son mas útiles para grafos dirigidos por qué se puede tener un nodo en una lista principal que apunta a otra lista que son los nodos con lo que hace conexión haciendo que el código sea más óptimo; porque si se crea una matriz los demás espacios con los que no hacen contacto quedan en 0 o null y esto sería un espacio utilizado inútilmente.

3.4 Para representar el mapa de la ciudad de Medellín del ejercicio del numeral 1.4,
¿qué es mejor usar, Matrices de Adyacencia o Listas de Adyacencia? ¿Por qué?

R//.Es mejor usar listas de Adyacencia porque nos ayudara a ahorrar recursos de máquina, a diferencia de la matriz que se llenara completamente sea necesaria o no, tomara mayor cantidad de espacio.
Mientras que la lista de adyacencia solo tomara los recursos necesarios para la cantidad de datos que se tengan.

3.5 Teniendo en cuenta lo anterior, respondan: ¿Qué es mejor usar, Matrices de
Adyacencia o Listas de Adyacencia? ¿Por qué?

R//.Es mejor usar listas de adyacencia ya sé que van llegando en una especie de lista de listas sin ser “n” x “n” si no que puede ser “i” x “n” con lo cual nos ahorraríamos una gran cantidad de recursos por que no vamos a tener espacios en nulo o vacíos como pueda ser con la matriz.
Lo cual implica consumir los recursos que son y no más de los que realmente serian como matriz.

3.6 Teniendo en cuenta lo anterior, para representar la tabla de enrutamiento,
Respondan: ¿Qué es mejor usar, Matrices de Adyacencia o Listas de Adyacencia?

R//.Para una tabla de enrutamiento si es “n”x”n” es mejor utilizar una matriz de adyacencia, pero si no es “n”x”n” se desea un algoritmo realmente rápido se puede basar más fácilmente en listas de adyacencia.

3.7 Teniendo en cuenta lo anterior, para recorrer grafos, ¿en qué casos conviene
Usar DFS? ¿En qué casos BFS?

R//.Es mejor Usar DFS (Depth First Search) cuando se desea obtener todos los elementos de un grafo no dirigido y dirigido, el recorrido se hace en profundidad y el tipo de recorrido es pre-orden

BFS (Breadth First Search) se utiliza cuando se quiere recorrer un grafo en amplitud con el fin de buscar cómo llegar de un nodo a otro sin tanto costo y el recorrido es por niveles.

3.8 (Conozco PRIM, KRUSKAL, DIJKSTRA) ¿Qué otros algoritmos de búsqueda existen para grafos? Basta con explicarlos,
No hay que escribir los algoritmos ni programarlos.
R//.
-Algoritmo de Bellman-Ford
El algoritmo de Bellman-Ford (algoritmo de Bell-End-Ford) genera el camino más corto en un grafo dirigido ponderado (en el que el peso de alguna de las aristas puede ser negativo(El Algoritmo Dijkstra es mas rápido que este)).

Explicación: inicializamos el grafo. Ponemos distancias a INFINITO menos el nodo origen que tiene distancia 0 y buscamos cada arista del grafo tantas veces como número de nodos -1 haya en el grafo y comprobamos si hay ciclos negativo y lo retornamos.

-Algoritmo de Boruvka
El Algoritmo de Borůvka es un algoritmo para encontrar el mínimo árbol de expansión en un grafo ponderado en el que todos sus arcos tienen distinto peso.
Explicación: Comenzar con un grafo “G” en el que todos sus arcos tengan distinto peso, y un conjunto vacío de arcos “T” mientras los vértices de “G” conectados por “T” sean disjuntos: Crear un conjunto vacío de arcos “S” para cada componente: crear un conjunto vacío de arcos “S” para cada vértice “v” en el componente: Agregar el arco de menor peso desde el vértice “v” a otro vértice en un componente disjunto a “S” añadir el arco de menor peso en “S” a “E” añadir el conjunto resultante “E” a “T”. El conjunto de aristas resultante “T” es el árbol de expansión mínimo de “G”.

-Algoritmo de búsqueda A*
Busca el camino de menor coste entre un nodo origen y uno objetivo
Explicación: Recorremos el/los caminos y nos quedamos con el camino de menor coste

-Algoritmo de Christofides
Es un algoritmo aproximado que permite resolver instancias del problema del viajante de comercio (designado convencionalmente por su acrónimo en inglés, TSP) en donde los pesos de las aristas del grafo satisfacen la desigualdad triangular. Fue desarrollado en 1976 por Nicos Christofides, profesor del Imperial College London

Explicación:
Sea “A” el conjunto de aristas de la solución óptima del TSP para G. Como (V, A) es conexo, contendrá varios árboles recubridores T, y por tanto, w(A) ≥ w (T). Además, sea B el conjunto de aristas de la solución óptima del TSP para el grafo completo sobre los vértices de O. Como los pesos asociados a las aristas son "triangulares" (visitar más nodos no reduce, en ningún caso, el coste total), se tiene que w(A) ≥ w (B). Se demuestra así que existe un apareamiento perfecto de vértices de O con peso tal que w (B)/2 ≤ w(A)/2, de forma que este límite constituye una cota superior para M (ya que M es un apareamiento perfecto de mínimo coste).

Debido a que O debe contener un número par de vértices, existe apareamiento perfecto. Sea e1, ..., e2k el (único) camino euleriano en (O,B). Es evidente que tanto e1, e3, ...,e2k-1 como e2, e4, ..., e2k son apareamientos perfectos, y que el peso de (al menos) uno de ellos es menor o igual que w(B)/2. Así, se tiene que w(M)+w(T) ≤ w(A) + w(A)/2, y de la desigualdad triangular se deduce que el algoritmo se aproxima en 3/2 al óptimo.

Algoritmo de Cuthill-McKee
En el subcampo matemático de la teoría de matrices, el algoritmo de Cuthill-McKee es un algoritmo para reducir el ancho de banda de una matriz simétrica dispersa. El algoritmo invertido de Cuthill-McKee (RCM, por las siglas inglesas Reverse Cuthill-McKee) es el mismo algoritmo pero con los índices resultantes invertidos. En el ámbito práctico, emplear este último es generalmente una mejor solución.
Explicación: Dada una matriz simétrica n×n, se visualiza como la matriz de adyacencia de un grafo. El algoritmo de Cuthill-McKee es entonces una remuneración de los vértices del grafo con el objetivo de reducir el ancho de banda de su matriz de adyacencia. El algoritmo produce una n-dupla ordenada R de vértices, que es el nuevo orden de los vértices. Primero, se elige un vértice periférico x y se realiza la asignación R := ({x}).A continuación, para i = 1, 2, ... se iteran las próximas instrucciones mientras |R| < n.
se construye el conjunto de Adyacencia Ai de Ri(SIendoRi el i-enesimo componente de R) Excluyendo aquellos vértices que ya estuvieran en R
Ordenar Ai, siendo una ordenación ascendente de los vectices
Añadir Ai al conjunto resultado R.
En otras palabras, numerar los vértices de acuerdo a un particular búsqueda en anchura transversal, donde los vértices vecinos son visitados en orden de menor a mayor.

-Algoritmo de Edmond
En teoría de grafos, el Algoritmo de Edmond es un algoritmo para encontrar una arborescencia de peso mínimo (a veces llamado de óptima derivación). Es el equivalente dirigido del árbol recubridor mínimo. El algoritmo estuvo propuesto independientemente primero por Yoeng-Jin Chu y Tseng-Hong Liu (1965) y posteriormente por Jack Edmonds (1967).
Explicación

El algoritmo toma como entrada un grafo directo $D = \langle V, E \rangle$ donde V es el conjunto de nodos, E es el conjunto de aristas dirigidas, $r \in V$ es un vértice distintivo llamado *raíz*, y un peso real $w(e) e \in E$. Esto devuelve una arborescencia A arraigada en r como el peso mínimo, donde el peso de arborescencia se define como la suma de sus pesos: $w(A) = \sum_{e \in A} w(e)$.

El algoritmo tiene una descripción recursiva. Permite denotar la función $f(D, r, w)$ que devuelve la arborescencia arraigada en r de peso mínimo. Primero retiramos cualquier arista de E cuya destinación es r. También podemos sustituir cualquier conjunto de aristas paralelas (aristas entre el mismo par de vértices en la misma dirección) por una sola arista con un peso igual al mínimo de los pesos de estas aristas paralelas.

Ahora, para cada nodo v que no sea la raíz, encuentre la arista entrante a v d$\pi(v)$. Si el conjunto de aristas $P = \{(\pi(v), v) \mid v \in V \setminus \{r\}\}$ no contiene ningún ciclo, entonces $f(D, r, w) = P$.

Por otra parte, P contiene el último ciclo. Escoja arbitrariamente uno de estos ciclos y llámelo C. Ahora definiremos un nuevo grafo dirigido ponderado$D' = \langle V', E' \rangle$ en el cual el ciclo C se "contrae" en un nodo de la siguiente manera:

Los nodos de V' son los nodos de V no en C mas un nuevo nodo denotado v_C.

- Si (u, v) es una arista de E con $u \notin C$ y $v \in C$ (una arista que entra en un ciclo), entonces incluiremos en E' una nueva arista $e = (u, v_C)$, y define $w'(e) = w(u, v) - w(\pi(v), v)$.
- Si $(u, v) E$ con $u \in C$ y $v \notin C$ (una arista que se aleja del ciclo), luego incluya E' una nueva arista $e = (v_C, v)$, y defina $w'(e) = w(u, v)$.
- Si (u, v) es una arista de E con $u \notin C$ y $v \notin C$ (una arista no relacionada con el ciclo), luego incluya en E' una nueva arista $e = (u, v)$, y defina $w'(e) = w(u, v)$.

Por cada nodo en E', recordaremos a qué arista en E corresponde.

$f(D', r, w')$. Puesto que A' es una arborescencia que abarca, cada vértice tiene exactamente una arista entrante. Sea (u, v_C) el único arista entrante en v_C en A'. Esta arista corresponde con una arista $(u, v) \in E$ con $v \in C$. Elimina la arista $(\pi(v), v)$ de C, rompiendo el ciclo. C. Por cada arista en A', marque esta correspondiente arista en E. Ahora defina $f(D, r, w)$ como el conjunto de aristas marcadas que forman una arborescencia de extensión mínima.

Observe que $f(D, r, w)$ $f(D', r, w')$, con D' D. Halle $f(D, r, w)$ para un gráfico de un solo vértice es trivial (es solo D en sí mismo), por lo que el algoritmo recursivo siempre termina.

Algoritmo de Floyd-Warshall

En informática, 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 algoritmo de Floyd es muy similar, pero trabaja con grafos ponderados. Es decir, el valor de la "flecha" que representamos en la matriz puede ser cualquier entero o infinito. Infinito marca que no existe unión entre los nodos. Esta vez, el resultado será una matriz donde estarán representadas las distancias mínimas entre nodos, seleccionando los caminos más convenientes según su ponderación ("peso"). Por ejemplo, si de "A" a "B" hay 36 (km), pero de "A" a "C" hay 2(km) y de "C" a "B" hay 10 (km), el algoritmo nos devolverá finalmente que de "A" a "B" hay 12 (km).

Los pasos a dar en la aplicación del algoritmo de Floyd son los siguientes:

* Formar las matrices iniciales C y D.

* Se toma k=1.

* Se selecciona la fila y la columna k de la matriz C y entonces, para i y j, con i≠k, j≠k e i≠j, hacemos:

Si (Cik + Ckj) < Cij → Dij = Dkj y Cij = Cik + Ckj

En caso contrario, dejamos las matrices como están.

* Si k ≤ n, aumentamos k en una unidad y repetimos el paso anterior, en caso contrario páramos las interacciones.

* La matriz final C contiene los costes óptimos para ir de un vértice a otro, mientras que la matriz D contiene los penúltimos vértices de los caminos óptimos que unen dos vértices, lo cual permite reconstruir cualquier camino óptimo para ir de un vértice a otro.

-Algoritmo de Johnson

El algoritmo de Johnson es una forma de encontrar el camino más corto entre todos los pares de vértices de un grafo dirigido disperso. Permite que las aristas tengan pesos negativos, si bien no permite ciclos de peso negativo.

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

Funciona utilizando el algoritmo de Bellman-Ford para hacer una transformación en el grafo inicial que elimina todas las aristas de peso negativo, permitiendo por tanto usar el algoritmo de Dijkstra en el grafo transformado. Su nombre viene de Donald B. Johnson, quien fuera el primero en publicar la técnica en 1977.
Explicación:

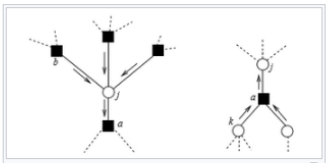
El algoritmo de Johnson consiste en los siguientes pasos:

1. Primero se añade un nuevo nodo q al grafo, conectado a cada uno de los nodos del grafo por una arista de peso cero.
2. En segundo lugar, se utiliza el algoritmo de Bellman-Ford, empezando por el nuevo vértice q, para determinara para cada vértice v el peso mínimo $h(v)$ del camino de q a v. Si en este paso se detecta un ciclo negativo, el algoritmo concluye.
3. Seguidamente, a las aristas del grafo original se les cambia el peso usando los valores calculados por el algoritmo de Bellman-Ford: una arista de u a v con tamaño $w(u, v)$, da el nuevo tamaño $w(u, v) + h(u) – h(v)$
4. Por último, para cada nodo s se usa el algoritmo de Dijkstra para determinar el camino más corto entre s y los otros nodos, usando el grafo con pesos modificados.

En el grafo con pesos modificados, todos los caminos entre un par de nodos s y t tienen la misma cantidad $h(s) – h(t)$ añadida a cada uno de ellos, así que un camino que sea el más corto en el grafo original también es el camino más corto en el grafo modificado y viceversa. Sin embargo, debido al modo en el que los valores $h(v)$ son computados, todos los pesos modificados de las aristas son no negativos, asegurando entonces la optimalidad de los caminos encontrados por el algoritmo de Dijkstra. Las distancias en el grafo original pueden ser calculadas a partir de las distancias calculadas por el algoritmo de Dijkstra en el grafo modificado invirtiendo la transformación realizada en el grafo.

-Algoritmo de propagación de creencias
El algoritmo de propagación de creencias (en inglés, belief propagation algorithm), también conocido como el algoritmo suma-producto, es un algoritmo de paso de mensaje para realizar inferencia sobre modelos gráficos tales como redes bayesianas, campos aleatorios de Markov y factor graph. Es ampliamente utilizado en los campos de inteligencia artificial y teoría de la información y ha mostrado cierto éxito experimental en aplicaciones tan diferentes como: análisis de paridad de códigos,1 aproximaciones de energía libre , coloreado de grafos y satisfacibilidad booleana.

Existen variantes del algoritmo de propagación de creencias para diferentes modelos gráficos (Redes bayesianas, y campos aleatorios de Markov). El modelo gráfico usado en adelante es llamado *factor graph*, esto no representa ninguna limitación pues estos modelos son equivalentes y es posible la conversión entre estos.[7] Un *factor graph* es un grafo bipartito con dos tipos de nodos: nodos variables, usualmente denotados por las letras $i, j, k, \ldots$ y representados gráficamente con círculos, y nodos factores, usualmente denotados por las letras $a, b, c, \ldots$ y representados gráficamente como cuadrados, existe una aristas entre un nodo variable i y un nodo función a si y solo si i se encuentra en el argumento de $f(x_a)$. Llamaremos V al conjunto de nodos variables y F al conjunto de los nodos factores.



Representación parcial de un factor graph. Resulta útil imaginarse que dentro de los nodos factores (cuadrados) radica una función $f_a(x_a)$ que expresa la interacción entre las variables que residen en su vecindad

El algoritmo funciona enviando "mensajes" entre los nodos variables y nodos factores. De forma más específica si $i \in V$ y $a \in F$ el mensaje enviado desde i hacia a es la evaluación de una función $n : V \to \mathbb{R}$ y de forma alterna el mensaje enviado desde a hacia i es la evaluación de una función $m : F \to \mathbb{R}$. Estos mensajes contiene la influencia de una variable sobre otra. El cálculo de los mensajes posee diferente forma según los nodos emisores y receptores.

Un mensaje enviado desde una variable i a un nodo función a es el producto de todos los mensajes que recibe la variable i de los nodos factores vecinos b excepto a. $n_{i\to a}(x_i) = \prod_{b\in N(i)/a} m_{b\to i}(x_i)$ donde $N(i)/a$ es el conjunto de los nodos factores vecinos de i a excepción del nodo a. Si este conjunto es vació entonces $n_{i\to a}(x_i)$distribuye de uniformemente sobre el conjunto de valores posibles de la variable i.

Un mensaje enviado desde un nodo factor a hacia un nodo variable i es el producto de la función $f_a(x_a)$ evaluada en la vecindad de a a excepción de i que esta marginalizado al valor x_i. $m_{a\to i}(x_i) = \sum_{x_a/x_i} f_a(x_a) \prod_{j\in N(a)/i} n_{j\to i}(x_j)$ Si $N(a)/i$ es vacío entonces $m_{a\to i}(x_i) = f_a(x_i)$. La suma sobre x_a/x_i es la suma sobre todo el conjunto de todos los posibles valores de las variables vecinas de a a excepción de i que se encuentra marginalizada al valor x_i.

Inicialmente el conjunto de mensajes iniciales se escogen de manera aleatoria, luego siguiendo cierto esquema iterativo de actualización (usualmente tomar un nodo aleatorio y actualizar los mensajes a sus vecinos) con respectos a los mensajes anteriores se computan los nuevos mensajes. Si el modelo gráfico posee forma de árbol, existe un esquema de actualización que garantiza convergencia (ver siguiente sección). Si se cumple que el proceso converge, usualmente que los mensajes dejen de cambiar, que no se garantiza en grafos generales, es posible obtener la distribución marginal computada por la propagación de creencias.

La distribución marginal estimada de cada nodo variable es proporcional al producto de los mensajes de los nodos vecinos. $P(x_i) = \frac{1}{\alpha_v} \prod_{a\in N(i)} m_{a\to i}(x_i)$ En el caso de los nodos función la distribución conjunta de las variables que son parte de su vecindad es proporcional al mensaje recibido por el nodo. $P(x_a) = \frac{1}{\alpha_f} f_a(x_a) \prod_{a\in N(i)} n_{i\to a}(x_i)$ donde α_v y α_f son constantes de normalización $\alpha_v = \sum_{x_i} \prod_{a\in N(i)} m_{a\to i}(x_i)$ $\alpha_f = \sum_{x_a} f(x_a) \prod_{i\in N(a)} n_{i\to a}(x_i)$

-Algoritmo del vecino más próximo

El algoritmo del vecino más próximo fue, en las ciencias de la computación, uno de los primeros algoritmos utilizados para determinar una solución para el problema del viajante. Este método genera rápidamente un camino corto, pero generalmente no el ideal.
Abajo está la aplicación del algoritmo del vecino más próximo al problema del viajante.
Estos son los pasos del algoritmo:
elección de un vértice arbitrario respecto al vértice actual.
descubra la arista de menor peso que ya este conectada al vértice actual y a un vértice no visitado V.
convierta el vértice actual en V.
marque V como visitado.
si todos los vértices del dominio estuvieran visitados, cierre el algoritmo.
vaya al paso 2.
La secuencia de los vértices visitados es la salida del algoritmo.
El algoritmo del vecino más próximo es fácil de implementar y ejecutar rápidamente, pero algunas veces puede perder rutas más cortas, que son fácilmente notadas con la visión humana, debido a su naturaleza más "ávida". Como norma general, si los últimos pasos del recorrido son comparables en

longitud al de los primeros pasos, el recorrido es razonable; si estos son mucho mayores, entonces es probable que existan caminos mucho mejores.

-Algoritmo MINIMAX
En teoría de juegos, minimax es un método de decisión para minimizar la pérdida máxima esperada en juegos con adversario y con información perfecta. Minimax es un algoritmo recursivo.
El funcionamiento de minimax puede resumirse en cómo elegir el mejor movimiento para ti mismo suponiendo que tu contrincante escogerá el peor para ti.

Pasos del algoritmo minimax:
Generación del árbol de juego. Se generarán todos los nodos hasta llegar a un estado terminal.
Cálculo de los valores de la función de utilidad para cada nodo terminal.
Calcular el valor de los nodos superiores a partir del valor de los inferiores. Según nivel si es MAX o MIN se elegirán los valores mínimos y máximos representando los movimientos del jugador y del oponente, de ahí el nombre de minimax.
Elegir la jugada valorando los valores que han llegado al nivel superior.
El algoritmo explorará los nodos del árbol asignándoles un valor numérico mediante una función de evaluación, empezando por los nodos terminales y subiendo hacia la raíz. La función de utilidad definirá lo buena que es la posición para un jugador cuando la alcanza. En el caso del ajedrez los posibles valores son (+1,0,-1) que se corresponden con ganar, empatar y perder respectivamente. En el caso del backgammon los posibles valores tendrán un rango de [+192,-192], correspondiéndose con el valor de las fichas. Para cada juego pueden ser diferentes.
Si minimax se enfrenta con el dilema del prisionero escogerá siempre la opción con la cual maximiza su resultado suponiendo que el contrincante intenta minimizarlo y hacernos perder.

Algoritmo de Dinic

El algoritmo de Dinic es un algoritmo de Tiempo polinómico para la computación de un Flujo maximal en una red de flujo, concebida en 1970 por el científico de la computación Yefim Dinitz, israelí de origen soviético.1 El algoritmo es ejecutado en un tiempo de {\displaystyle O(V^{2}E)} {\displaystyle O(V^{2}E)} y está basado en el Algoritmo de Edmonds-Karp, el cual a su vez se ejecuta en un tiempo {\displaystyle O(VE^{2})} {\displaystyle O(VE^{2})}, y utiliza trayectorias de aumento más cortas. La introducción de los conceptos

nivel de grafo y bloqueo de flujo es lo que define el rendimiento del algoritmo de Dinic.

Se tiene el grafo $G = ((V, E), c, s, t)$ que será una red de flujo con $c(u, v)$ es la capacidad y $f(u, v)$ el flujo de el arco (u, v).

La **capacidad residual** es un mapeo $c_f : V \times V \to R^+$ definido como,

1. Si $(u, v) \in E$,
   $c_f(u, v) = c(u, v) - f(u, v)$
   $c_f(v, u) = f(u, v)$
2. $c_f(u, v) = 0$ de otra manera.

El **grafo residual** es el grafo $G_f = ((V, E_f), c_f|_{E_f}, s, t)$, donde

$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

La **trayectoria de aumento** es una ruta $s - t$ en el grafo residual G_f.

Se define $\text{dist}(v)$ como la longitud del camino más corto desde s hasta v en G_f. Entonces el **nivel del grafo** de G_f es el grafo $G_L = (V, E_L, c_f|_{E_L}, s, t)$, donde

$E_L = \{(u, v) \in E_f : \text{dist}(v) = \text{dist}(u) + 1\}$.

El **bloqueo del flujo** es un $s - t$ flujo f de manera tal que el grafo $G' = (V, E'_L, s, t)$ con $E'_L = \{(u, v) : f(u, v) < c_f|_{E_L}(u, v)\}$ no tiene ninguna ruta $s - t$.

Información tomada de: https://es.wikipedia.org/wiki/Categor%C3%ADa:Algoritmos_de_grafos

Trabajo en Equipo y Progreso Gradual (Opcional)

a) Puntos Extra: Actas de compromiso: Mi Compañero Juan Gonzalo Quiroz y yo acordamos dividirnos el trabajo en un comienzo y yo Alejandro Díaz elegí los puntos 1.7- 1.8- 2.1- 3.3- 3.4- 3.7 - 3.8- 3.9 -3.10 3.11; pero mas adelante en la sesión 3 eran preguntas acerca del código entonces un cambio el plan mi compañero esta vez hacia todo el punto 1 y yo todo el punto 3.

b) el reporte de los cambios en el código se encuentra en el github https://github.com/eafit-201610012010/st0247-201610012010

c) El reporte de cambios del informe en laboratorio igual se encuentra en el github antes con el nombre de lab1.pdf y ya el definitivo con el nombre de ED1LabPlantillaInforme.pdf

d) todo el seguimiento y el plan de trabajo lo hicimos en planner https://tasks.office.com/eafit.edu.co/ES-ES/Home/Planner#/plantaskboard?groupId=b288a1ef-aa38-477d-b81a-1645e5339e79&planId=s1ghxlols0WKuc_-XOH-FGQAD19J