

Trabajo Practico Algoritmos

Alan Rodriguez

October 19, 2022

Contents

1	Algoritmo goloso para el problema del viajante de comercio	1
2	Algoritmo anterior aleatorio	4
3	Búsqueda local	5
4	Inconvenientes con el tp	7
5	Bibliografía	7

Abstract

1 Algoritmo goloso para el problema del viajante de comercio

Heurística

El camino se reliazará eligiendo el vecino mas cercano (arista de menor peso)

Pasos a seguir

- Arrancamos de un nodo.
- lo marcamos como visitado.
- elegimos la arista de menor peso.
- no se puede volver a elegir un nodo ya visitado.
- repetir hasta visitar todos los nodos.

$$V = \{1, 2, 3, 4\}$$

$$E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$

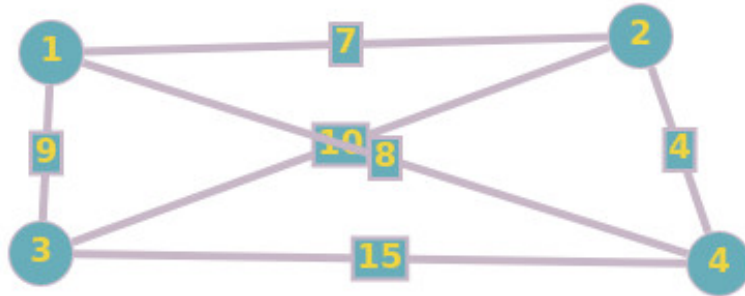


Figure 1: Grafo No Dirigido Completo

Representación

la representacion de grafo elegida es por matriz de adyacencia

	1	2	3	4
1	-	7	9	8
2	7	-	10	4
3	9	10	-	15
4	8	4	15	-

$$\text{Grafo} = \begin{vmatrix} - & 7 & 9 & 8 \\ 7 & - & 10 & 4 \\ 9 & 10 & - & 15 \\ 8 & 4 & 15 & - \end{vmatrix}$$

```
public class GrafoMatriz {
    public int [][] matAdyacentes;
    ...
}
```

Código en Java

- usare un *nodoActual* que será el nodo que estamos parados;
- me guardaré la posicion del minimo en *posicionDelMinimo*;
- el camino resultante lo guardare en *caminoRecorrido*
- y para saber que nodo visité usareé *yaLoRecorri*

```

public static int [] recorridoViajanteDeComercio(GrafoMatriz
    grafo) {
    int nodoActual = 0; //O(1)
    int posicionDelMinimo=0; //O(1)
    int tamGrafo = grafo.numeroDeVertices(); //O(1)
    int [] caminoRecorrido = new int[tamGrafo]; //O(1)
    boolean [] yaLoRecorri = new boolean[tamGrafo]; //O(1)
    inicializarEnFalso(yaLoRecorri); //O(n)
    inicializarEnCero(caminoRecorrido); //O(n)
    while (!yaLoRecorri[nodoActual]) { //peor de los casos:
        O(n)
        float [] misAdyacentes = grafo.adyacentesDe(
            nodoActual); //O(1)
        yaLoRecorri[nodoActual] = true; //O(1)
        insertarEnSiguientePosicionLibre(caminoRecorrido
            , nodoActual); //peor de los casos O(n)
        float pesoMinimo = grafo.pesoMasAlto(nodoActual)
            ; // O(1)
        for (int indice = 0; indice < tamGrafo; indice
            ++){ //todo el if se ejecuta tantas veces
            como nodos haya O(m)
            if (!yaLoRecorri[indice]) {
                if (misAdyacentes[indice] <
                    pesoMinimo) {
                    pesoMinimo =
                        misAdyacentes[indice
                            ];
                    posicionDelMinimo =
                        indice;
                }
            }
        }
        nodoActual = posicionDelMinimo; //O(1)
    }
    return caminoRecorrido;
} // el while que ejecuta un for define el orden del metodo : O(
n.m)
}

```

Ordenes de Complejidad

inicializar variables int $O(1)$

inicializar cada arreglo $O(n)$ siendo n el tamaño del arreglo

for dentro del while $O(n)$ siendo n la cantidad de vertices que tiene el nodo

while en el peor de los casos (si se ejecuta siempre) es $O(n)$ siendo n el tamaño del grafo pero tiene un for dentro de $O(n)$ por lo que queda $O(n^2)$

tenemos entonces: $O(n^2) + 2O(n) + O(1)$ por definición no importa el valor de las constantes, lo que crecerá más rápido será n^2 por lo que es $O(n^2)$

2 Algoritmo anterior aleatorio

- Se tendrán en cuenta los primeros mejores caminos.
- Se recibirá por parametro una *cotaMinima* y tendré una variable *cotaMaxima* que dependerá del tamaño del Grafo.
- Usaremos una funcion *Ramdom* que me dara un número, ese numero será que posicion elijo de los mejores para seguir.

```
public static int [] recorridoViajanteDeComercioAleatorio(
    GrafoMatriz grafo) {
    int nodoActual = 0; //O (1)
    //int posicionDelMinimo=0; //O(1)
    int tamGrafo = grafo.numeroDeVertices(); //O(1)
    int porcentajeDelGrafo = (tamGrafo * 10) / 100; //O(1)
    if (porcentajeDelGrafo <= 0) porcentajeDelGrafo = 1; //O(1)
    // para arreglos muy peque os que no pueda sacar
    // cierto porcentaje
    int [] caminoRecorrido = new int [tamGrafo]; //O
    (1)
    boolean [] yaLoRecorri = new boolean[tamGrafo];
    //O(1)
    inicializarEnFalso(yaLoRecorri); //O(n)
    inicializarEnCero(caminoRecorrido); //O(n)
    while (!yaLoRecorri[nodoActual]) { //peor de los casos:
    O(n)
        float [] misAdyacentes = grafo.adyacentesDe(
            nodoActual); //O(1) me guardo el peso de las
            aristas adyacentes al nodo actual
        Candidato [] misCandidatos = new Candidato [
            tamGrafo];
        yaLoRecorri[nodoActual] = true; //O(1)
        insertarEnSiguientePosicionLibre(caminoRecorrido
            , nodoActual); //peor de los casos O(n)
        for (int indice = 0; indice < tamGrafo; indice
            ++ ) {
            if (!yaLoRecorri[indice]) {
                Candidato talCual = new
                    Candidato(misAdyacentes [
                        indice], indice);
                misCandidatos[indice] = talCual;
            }
        }
        else {
            Candidato valorAlto = new Candidato
                (999999999, indice);
            misCandidatos[indice] = valorAlto;
        }
    }
```

```

    }
    Arrays.sort(misCandidatos);
    //double x = (Math.random() * ((max-min)+1))+min;
    double sig = Math.random() * ((porcentajeDelGrafo) + 1)
        + porcentajeDelGrafo;
    int siguiente = (int)sig-1;
    nodoActual = misCandidatos[siguiente].getNumero(); //O(1)
    }
    return caminoRecorrido;
}

```

Ordenes de Complejidad

inicializar variables int $O(1)$

inicializar cada arreglo $O(n)$ siendo n el tamaño del arreglo

for dentro del while $O(n)$ siendo n la cantidad de vertices que tiene el nodo

while en el peor de los casos (si se ejecuta siempre) es $O(n)$ siendo n el tamaño del grafo pero tiene un for dentro de $O(n)$ por lo que queda $O(n^2)$

tenemos entonces: $O(n^2) + 2O(n) + O(1)$ por definición no importa el valor de las constantes, lo que crecerá más rápido será n^2 por lo que es $O(n^2)$

3 Búsqueda local

- la búsqueda local recibe una solución inicial que en principio es la mejor solución
- de esa solución se buscan los vecinos que son combinaciones de la solución propuesta, pero no se realizan grandes cambios (vecinos lejanos)
- de haber una solución óptima que mejore a la mejor solución, se vuelve a correr el algoritmo de búsqueda local

```

public static int[] busquedaLocal ( GrafoMatriz grafo) {
    int[] solucionHastaAhora = recorridoViajanteDeComercio(
        grafo);
    int[] mejorSolucion = solucionHastaAhora.clone();
    float costoOptimo = recorrerElGrafo(grafo, mejorSolucion)
        ; // hasta ahora el unico costo que tengo
}

```

```

List<int[]> vecinos = new ArrayList<>();
for(int iterador=0; iterador < mejorSolucion.length;
    iterador++){
    int[] vecino = generarVecino(iterador,
        mejorSolucion.length, solucionHastaAhora);
    vecinos.add(vecino); //agrego al vecino
}
for (int[] vecino : vecinos) {
    float otroCosto = recorrerElGrafo(grafo, vecino);
    ;
    if (otroCosto <= costoOptimo) {
        costoOptimo = otroCosto;
        mejorSolucion = vecino;
    }
}
return mejorSolucion;
}

```

los vecinos los voy generando haciendo permutaciones

- el primero con el segundo
- y así continuo con el siguiente con su consecutivo
- el último con el primero

```

private static int[] generarVecino(int indice, int
    tamañoArreglo, int[] laMejorSolucion) {
    int[] vecinoNuevo = laMejorSolucion.clone(); //O
    (1) //le copio todos los valores
    if (indice < tamañoArreglo-1){ //peor de los
        caso se ejecuta dos instrucciones de orden 1
        //swappeo el del indice actual y su siguiente
        vecinoNuevo[indice]= laMejorSolucion[
            indice + 1]; //O(1)
        vecinoNuevo[indice +1] = laMejorSolucion[indice
            ]; //O(1)
    }
    else {
        // swappeo el primero con el ultimo
        vecinoNuevo[indice]= laMejorSolucion[0]; //O(1)
        vecinoNuevo[0] = laMejorSolucion[indice]; //O(1)
    }
    return vecinoNuevo;
} // ejecuta 3 asignaciones de orden constante -> O(1)

```

encontrar la mejor solución involucra:

- de la mejor solución no debo contar las aristas que no están más y que están en la solución propuesta
- de la solución propuesta debo sumar aquellas aristas que no tiene la mejor solución

4 Inconvenientes con el tp

1. me costó plasmar lo que entendía en teoría a la práctica, me demoré mucho definiendo el dominio y redefiniendolo para poder codificar una solución
2. entendí las explicaciones que me dieron ante las consultas realizadas, pero pasarlo a código no me fue fácil

5 Bibliografía

- Explicaciones de lo realizado en cada punto:
Apuntes de cursada: Algoritmos - Universidad Nacional de Quilmes - 1er Semestre 2022
- Estilo y Formatos con LaTeX:
<http://minisconlatex.blogspot.com/2012/04/escribir-codigo-de-programacion-en.html>
- Modelo de representación de grafos, consultas varias:
Libro de M.A.Weiss, “Estructuras de Datos en Java”
Libro de Luis Joyanes Aguilar Ignacio Zahonero Martínez “Estructuras de datos en Java”
- Herramienta para dibujar el grafo:
<https://graphonline.ru/es/>