Algorítmica (2022-2023)
Grado en Ingeniería Informática
Universidad de Granada

Memoria Práctica 3

Pedro Antonio Mayorgas Parejo
Alejandro Ramos Peña

7 de mayo de 2023

Índice

1. Componentes del algoritmo Greedy

Como ya sabemos, un algoritmo greedy tiene que estar formado de distintos elementos para poder ser formalizado como tal.

En este algoritmo tenemos los siguientes elementos:

- **Conjunto de candidatos:** En este caso, el conjunto de candidatos es el conjunto de aristas del grafo.
- **Conjunto de seleccionados:** El conjunto de seleccionados es el conjunto de nodos que forman el camino euleriano.
- **Función de selección:** La función de selección es la que nos va a permitir elegir la arista que vamos a añadir al camino. En este caso, la función de selección es la que nos va a permitir elegir el nodo al que vamos a saltar.
- **Función de factibilidad:** Es la función que comprueba si se puede obtener una solución. En este caso es la función que comprueba los prerrequisitos de un camino euleriano en un grafo.
- **Función solución:** La función solución es la que nos va a permitir comprobar si hemos llegado al final del camino. En este caso, la función objetivo es la que comprueba si queda alguna arista en el nodo actual.
- **Funcion objetivo:** Es la función que devuelve el camino euleriano. En este caso es la función que devuelve el contenido de la variable de camino.

2. Descripción del algoritmo Greedy con heurística basada en la matriz de adyacencia

El algoritmo está encapsulado en una clase, debido a las enormes necesidades de heurística del algoritmo greedy.

La heurística de este algoritmo es basada en la matriz de adyacencia, donde la matriz de adyacencia nos proporciona información sobre las aristas de los vértices (o nodos) del grafo, a partir de la entrada de la matríz de adyacencia construimos la siguiente información:

- Obtenemos los grados de cada vértice del grafo. Por grado, es la cantidad de aristas que tiene, contabilizadas mediante, la matriz.
- Obtenemos los puntos iniciales, si en la matriz hay dos vértices con grado impar, se debe empezar en uno de los dos y acabar en el otro punto.
- Del fichero de entrada obtenemos el número de nodos.

Una vez obtenida toda la heurísitca ejecutamos el método **fleuryAlgorithm**, el algoritmo se ejecuta en base a lo siguiente:

El algoritmo está preparado para obtener todas las posibles rutas desde todos los vértices.

Se hace una copia de la matriz de adyacencia original y de los grados de los vértices, para no perder la información de la adyacencia durante las distintas búsquedas de rutas.

Pero debido a los limitados requisitos de la práctica solo se hará desde un punto, que se limita en la función de evalIfEulerPath, en dicha función se ha preparado que cuando los nodos sean pares, se introduzca un par aleatorio a partir del atributo de instancia del número de nodos.

1. Si la matriz tiene un solo nodo, se introduce el nodo y se termina la función.
2. Si la matriz tiene mas de un nodo, el algoritmo obtiene el nodo de partida (este ha sido obtenido previamente en evalIfEulerPath y lo almacena en una variable temporal que indexa el vértice en el que está evaluando el siguiente al que debe saltar.
3. Se introduce el nodo inicial en el eulerPath, para almacenar el nodo de partida.
4. Evalúa si el nodo actual tiene un grado mayor que 0, si no lo tiene se termina el bucle y la función.
5. En caso de tener un grado mayor de 0, pasamos a recorrer su matriz de adyacencia, evaluando las siguientes condiciones básicas en todos los requisitos.

- El siguiente vértice, debe tener una arista, esto es que en la matriz de adyacencia vale 1. Una vez evaluado esto, debemos evaluar las siguientes subcondiciones.
  - Si el siguiente vértice tiene una arista y un grado igual a 1, se almacena como el posible Dead end. A continuacuón evaluamos los siguientes puntos.
  - Si el siguiente vértice tiene una arista y un grado mayor que 1, evaluamos a continuación si este tiene un grado 2, para evaluar que no conduzca a otro posible Dead end, esto es que cuando saltemos al siguiente vértice y construyamos el siguiente camino. Que su siguiente salto a su vez no tenga grado 1, esto es evitando vértices que en el grafo no tienen salida y dejen sin evaluar el resto de vértices. Si este punto conduce a un DeadEnd se almacena también como posible salto y pasamos a evaluar al resto de vértices.

    En caso contrario, si el siguiente vértice tiene un grado mayor que 1, lo escogemos directamente como salto válido.
  - Si el siguiente vértice tiene un grado mayor que 2, es un vértice bueno y lo seleccionamos directamente como siguiente salto.
- Si se ha cumplido con una subcondición de dead end y hemos evaluado todos los posibles vértices. Comprobamos cuál tipo de dead end es, si es un dead end de dos saltos esto es que el siguiente vértice tiene otro siguiente vértice que conduce a un final del grafo, escogemos ese en primer lugar para construir el camino. Si no hay uno de dos saltos, se escoge como siguiente el vértice que conduce a un final.

Una vez completada la evaluación, con ese vértice de salto, pasamos a modificar la matriz para notificar que el salto se ha realizado, además se decrementa el grado de dicho vértice, para que en las próximas evaluaciones se escoja el mejor camino posible.

Finalmente se marca como vértice actual el del salto, se reinician las variables de lo vértices de salto, los de dead end y se añade al camino el vértice del salto. Repetimos la iteración evaluando de nuevo las condiciones anteriores a partir de la matriz de adyacencia del nuevo vértice actual para producir el siguiente salto.

Una vez que todos los puntos hayan recorrido al menos 1 vez todo el grafo, el bucle termina cuando el grado del último vértice vale 0, esto es ha sido el último.

Luego dependiendo de si queremos evaluar otros caminos, se reinicia la matriz de adyacencia y la de grados. Para empezar de nuevo si hay otro camino que evaluar desde otro vértice.

3. Algoritmo con heurística de número de aristas incidentes.

La heurística de este algoritmo se basa en coger el primer nodo adyacente al actual en el cual inciden más aristas. De este modo, conseguimos que el algoritmo se dirija hacia los nodos que tienen más aristas incidentes, y por tanto, más posibilidades de que el algoritmo no se quede sin opciones de camino.

Para pasarle como argumento al programa un grafo hemos especificado un formato de archivo .graph de texto que especifica un grafo donde:

1. La primera línea tiene que contener la letra G para que se compruebe que el formato del archivo es correcto
2. La segunda línea indica el número de nodos del grafo.
3. Las siguientes líneas indican las aristas del grafo, donde cada línea tiene primero el número de aristas que tiene el nodo y despúes todos los nodos a los que está conectado, de 0 a n-1.

Además, hemos creado una clase Nodo con distintos métodos que nos van a permitir ir eliminando aristas conforme vayamos recorriendo el grafo y poder obtener el número de aristas incidentes fácilmente

Una vez hemos leído el grafo y lo hemos almacenado en un vector de Nodos, se ejecuta el algoritmo de la siguiente manera:

1. Mientras que el nodo actual tenga alguna arista incidente:
   a) Comprobamos de todos los Nodos adyacentes cual es el que tiene un mayor número de aristas incidentes que no sea un dead end para saber cuál es el siguiente.
   b) Añadimos el nodo actual al camino
   c) Se elimina la arista que une el nodo actual con el nodo siguiente.
   d) Se actualiza el nodo actual al nodo adyacente.
2. Una vez que el nodo actual no tiene aristas incidentes, se añade al camino y se termina el algoritmo.

4. Eficiencia de los algoritmos

4.1. Algoritmo basado en heurística de matriz de adyacencia

La complejidad viene dada por la visita de todos los posibles vértices del grafo, así como la evaluación del vector de grados, para evaluar cuál es el siguiente punto.
Siendo N el número de Vértices que tiene que analizar y E^2 los dos vectores de heurística del grado de los vértices de origen y el de destino.
Complejidad en el peor caso: $O(N+E^2)$.

4.2. Algoritmo basado en heurística de aristas incidentes.

Este algoritmo es más sencillo y fácil de implementar que el que usa como heurística la matriz de adyacencia, pero es menos eficiente, ya que en el peor caso, que es cuando todos los nodos están conectados con todos entre sí, **la complejidad es $O(n^3)$**, ya que en cada iteración del bucle principal (aunque se vayan eliminando aristas) se recorren todos los nodos adyacentes (en los cuales también se analizan todos los nodos adyacentes para comprobar que no sea un dead end) para encontrar el nodo con más aristas incidentes.

5. Cómo compilar y ejecutar la aplicación

Para compilar la aplicación, se debe ejecutar el comando siguiente en la carpeta raíz del proyecto. Esto generará dos ejecutables, arista.bin y adjacency.bin.

```
# On project root
make run
# Or build and manual running
make build
# run matrix adjacency heuristic
./adjacency.bin salida.txt adjacencymatrix.matrix
# run edge heuristic
./arista.bin grafoEjemplo.graph
```

Como parámetros de entrada tienen:

- **arista.bin**: El nombre del archivo de texto que contiene el grafo.
- **adjacency.bin**: El nombre del archivo de texto para la salida y después el nombre del fichero de texto que contiene el grafo de adyacencia.

Además se pueden ejecutar ambos con el grafo de las diapositivas de la práctica con el comando **make run** y con otro grafo de ejemplo.

6. Ejecución del ejercicio 1

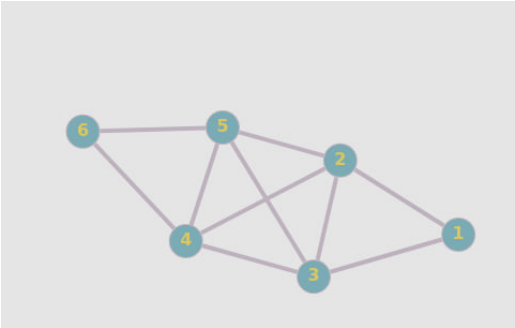

Figura 1: Grafo del ejercicio 1 el ejemplo propuesto

Heurística de matriz de adyacencia

```
./adjacency.bin matrix_heuristic.txt adjacencymatrixnotodd.matrix
ADJACENCY MATRIX
Nodes 6
--------------------------------
0 1 1 0 0 0
1 0 1 1 1 0
1 1 0 1 1 0
0 1 1 0 1 1
0 1 1 1 0 1
0 0 0 1 1 0
DEGREE VERTEX
--------------------------------
2 4 4 4 4 2
--------------------------------
OOD VERTEX
--------------------------------
0
--------------------------------
Ejecutando algoritmo greedy con matriz de adyacencia de heurística
        Tiempo de ejec. (us): 19
Path N: 0
--------------------------------
0 => 1 => 2 => 3 => 1 => 4 => 3 => 5 => 4 => 2 => 0
--------------------------------
Executing all
```

Figura 2: Heurística de matriz de adyacencia aplicadas al grafo 1

Heurística de aristas

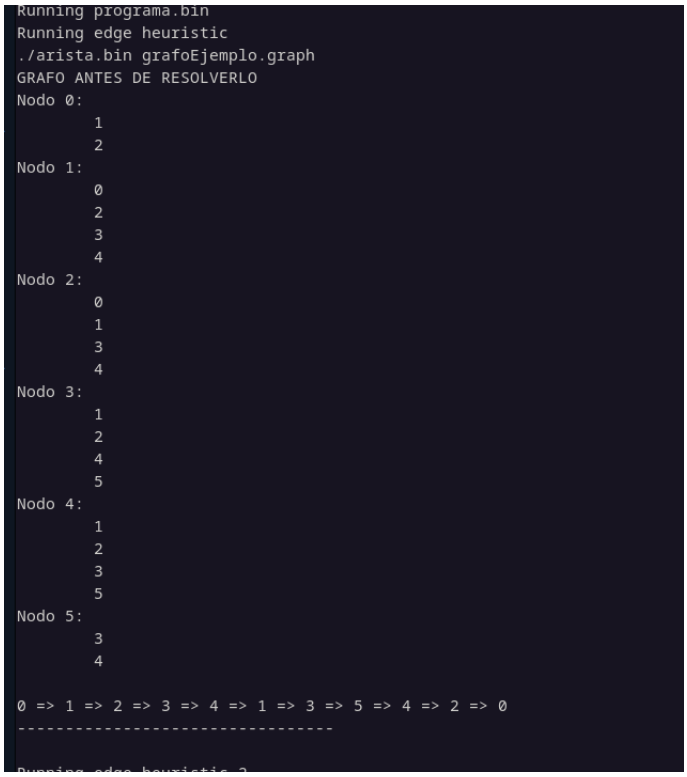

Figura 3: Heurística de aristas aplicadas al grafo 1

7. Ejecución del ejercicio 2

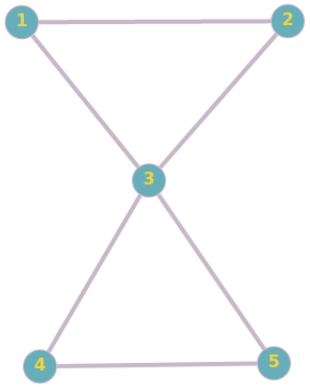

Figura 4: Grafo del ejercicio 2

Heurística de matriz de adyacencia

```
Running adjacency heuristic 2
./adjacency.bin matrix_heuristic.txt adjacencymatrixnotoddej2.matrix
ADJACENCY MATRIX
Nodes 6
--------------------------------
0 1 1 0 0
1 0 1 0 0
1 1 0 1 1
0 0 1 0 1
0 0 1 1 0
DEGREE VERTEX
--------------------------------
2 2 4 2 2
--------------------------------
OOD VERTEX
--------------------------------
0
--------------------------------
Ejecutando algoritmo greedy con matriz de adyacencia de heuristica
       Tiempo de ejec. (us): 16
Path N: 0
--------------------------------
0 => 1 => 2 => 3 => 4 => 2 => 0
--------------------------------
Executing all
```

Figura 5: Heurística de matriz de adyacencia aplicadas al grafo 2

Heurística de aristas

```
Running edge heuristic 2
./arista.bin grafoEjemplo2.graph
GRAFO ANTES DE RESOLVERLO
Nodo 0:
        1
        2
Nodo 1:
        0
        2
Nodo 2:
        0
        1
        3
        4
Nodo 3:
        2
        4
Nodo 4:
        2
        3

0 => 1 => 2 => 3 => 4 => 2 => 0
--------------------------------
```

Figura 6: Heurística de aristas aplicadas al grafo 2