Bachelor's degree in Computer Science and Engineering

Teoría Avanzada de la Computación 2023-2024
Grupo 84

Práctica 2

"Algoritmos aplicados a Grafos"

Ignacio Arnaiz Tierraseca – 100428997
Luis Daniel Casais Mezquida – 100429021

Grupo 3

Profesor

Juan Manuel Alonso Weber

Índice

I Resumen **3**

II Desarrollo **4**

1 Estudio de PATH **4**

1.1 Depth-First Search 4

1.2 Floyd-Warshall 5

2 Estudio de K-CLIQUE **8**

3 Estudio K-SAT **10**

III Conclusiones **12**

IV Bibliografía **13**

V Apéndices **14**

Índice de figuras

1 Tiempo de ejecución del algoritmo DFS 5
2 Tiempo de ejecución del algoritmo FW 7
3 Tiempo de ejecución del algoritmo CLIQUE 9
4 Tiempo de ejecución del algoritmo SAT 11

Parte I

Resumen

La presente práctica tiene como objetivo el estudio de posibles algoritmos para resolución de problemas basados en grafos.

En la práctica se ha desarrollado un algoritmo para la generación de grafos sobre los que luego se aplicaran los algoritmos de resolución de problemas, dicho algoritmo toma como entrada el número de nodos que tendrá el grafo y que probabilidad existirá de que los nodos se interconecten, de esta forma un grafo con una probabilidad baja tendrá poca densidad mientras que uno con alta probabilidad se acercará a ser completamente conectado.

Entre los algoritmos desarrollados se encuentran aquellos que buscan encontrar si existe un camino entre dos nodos dados como son DFS y Floyd-Warshall. Así como algoritmos que buscan resolver el problema 'CLIQUE' buscando dentro de un grafo, subgrafos totalmente conectados.

Para cada uno de los algoritmos propuestos se ha realizado una evaluación analítica describiendo su funcionamiento y como debería ser teóricamente su desempeño, así como una evaluación empírica llevando a cabo pruebas para comprobar si el análisis propuesto se ajusta a los resultados reales en términos de complejidad y computabilidad.

Parte II

Desarrollo

1. Estudio de PATH

1.1. Depth-First Search

El algoritmo DFS [1, Figure 6.24] es un algoritmo para recorrer un grafo 'en profundidad', y puede ser aplicado a nuestro problema PATH(u,v) recorriendo el grafo desde el nodo origen (u) y comprobando a cada paso si hemos llegado al nodo destino (v). En caso de que no lleguemos nunca, no hay camino.

Evaluación analítica del algoritmo

Para analizar correctamente el algoritmo y sacar una cota asintótica superior, debemos de analizar el peor caso, el cual se da cuando el u es el primer nodo, y el grafo está completamente conectado, excepto el nodo v, el cual está exclusivamente conectado al último nodo.

Para realizar la evaluación analítica del algoritmo, se divide este en dos secciones a fin de determinar el costo de cada una, ya que se trata de funciones.

- **Inicialización del Algoritmo**: Esta función (`path_dfs`) cuenta con la inicialización de las variables correspondientes a cada uno de los nodos argumento, así como la inicialización de la variable de nodos resultados y la llamada a la función que calculará la distancia DFS, por ello su coste es de 4.
- **Algoritmo DFS**: Esta función (`_path_dfs`) cuenta con una comparación `IF` y con una inicialización de variable cuyo coste total es de 3. En su interior se encuentra un bucle con dos comparaciones `IF` en su interior que suman coste 4, y una llamada recursiva al propio bucle, por ello su coste sería de n^k.

Una vez computada el coste de cada función por separado se puede obtener el coste total del algoritmo en el pero caso siendo:

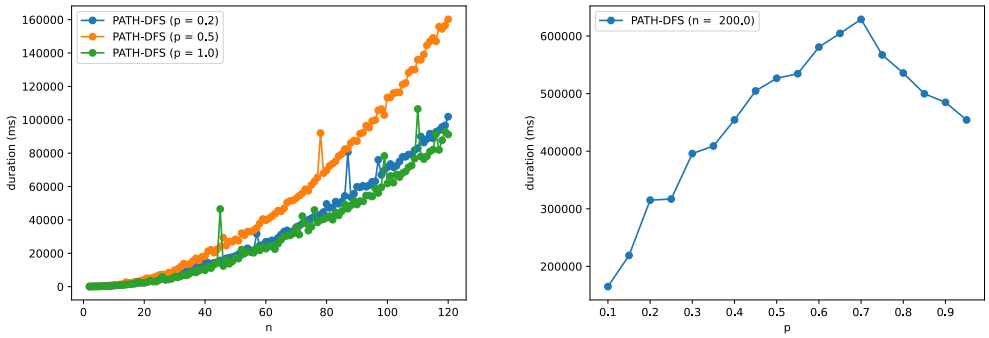
$$T_{\text{DFS}}(n) = 4n^k + 4 \tag{1}$$

La complejidad temporal del algoritmo de Floyd-Warshall, viene dada por los tres bucles anidados donde se calcula la distancia entre nodos siendo esta:

$$O_{\text{DFS}}(n^k) \tag{2}$$

Evaluación empírica

Para evaluar el comportamiento del algoritmo creado y comprobar si las conclusiones desarrolladas en la evaluación analítica son correctas se lleva a cabo una batería de pruebas con diferentes valores tanto para una misma n con diferentes p como a la inversa, a fin de comprobar como afecta cada una al rendimiento.



(a) *Con respecto a n* **(b)** *Con respecto a p*

Fig. 1: *Tiempo de ejecución del algoritmo DFS*

Se observa claramente en la gráfica que para grafos con densidades extremas, ya sean muy conectados o totalmente conectados, el tiempo de computación es notablemente menor que para grafos con densidades medias.

Esto se debe a que en los grafos con densidades bajas, el algoritmo encuentra rápido aquellos caminos por lo que no es posible continuar con la búsqueda, ya que los nodos tiene pocas conexiones En el caso de grafos de alta densidad o totalmente conectados también el algoritmo encontrará rápido un camino entre nodos, ya que habrá pocos o nulos caminos sin salida.

Por otro lado en grafos de densidad media, el algoritmo tendrá que probar con múltiples caminos que pueden ser más largos pese a no conducir al objetivo para luego tener que probar nuevos caminos. Esto se observa también claramente en la segunda gráfica donde para una misma n a medida que p aumenta el tiempo de computación aumenta hasta llegar a un pico entre $p = 0{,}5$ y $p = 0{,}6$ donde se crearán grafos de densidad media, para luego descender a medida que p aumenta hasta llegar al grafo totalmente conectado.

1.2. Floyd-Warshall

El algoritmo Floyd-Warshall [2] permite encontrar el camino más corto entro dos nodos siguiendo un proceso iterativo a través del cual se calcula la distancia entre un nodo a los demás siendo el valor de 0 para si mismo, infinito si no se encuentran conectados y de 1 en este caso si están conectado.

Evaluación analítica del algoritmo

Para analizar el coste computacional del algoritmo desarrollado, se puede dividir en tres secciones claramente diferenciadas para luego sumar sus costes y obtener un total, siendo dichas secciones:

- **Inicialización del Algoritmo**: Inicializar el algoritmo leyendo los parámetros e inicializar los dos nodos a computar el coste es de 7, ya que se trata únicamente de inicialización de variables y una comparación 'IF'.
- **Inicialización de matriz de distancias**: Esta sección del algoritmo cuenta con 2 bucles anidados que calculan la matriz de unos y ceros que indican que nodos se encuentran directamente conectados.
  - **Bucle externo (u)**: Itera a través de cada nodo que actúa como un nodo intermedio en los caminos posibles, con un coste de $2n$, puesto que se modifica el valor de dos variables.
  - **Bucle externo (v)**: Itera sobre todos los nodos posibles como nodo inicial, con un coste de n. En su interior el bucle consta de dos modificaciones de variables, y varias comparaciones 'IF' teniendo en el peor caso la comparación un coste de 3, por lo que el coste total del bucle sería de $5n$.

  El costo de esta sección del algoritmo sería de un total de $10n^2$
- **Cálculo de la distancia entre nodos**: Itera sobre todos los nodos posibles como nodo final.
  - **Bucle externo (k)**: Itera a través de cada nodo que actúa como un nodo intermedio en los caminos posibles, con un coste de n.
  - **Bucle medio (i)**: Itera sobre todos los nodos posibles como nodo inicial, con un coste de n.
  - **Bucle interno (j)**: Itera sobre todos los nodos posibles como nodo final, con un coste de n. Cuenta con una comparación 'IF' de coste 2 por lo que el coste total de este bucle sería de $2n$.

  El costo de esta sección sería de un total de $2n^3$

Si se toma el costo de cada una de las tres secciones que componen el algoritmo, el costo total del algoritmo sería:

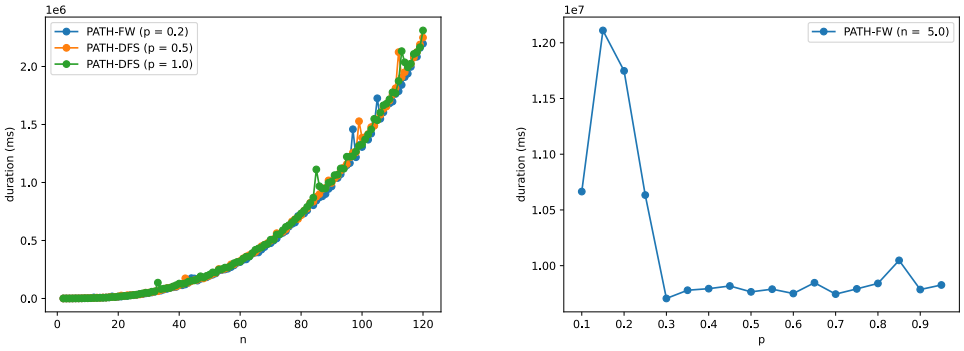
$$T_{\mathrm{FW}}(n) = 2n^3 + 10n^2 + 7 \tag{3}$$

La complejidad temporal del algoritmo de Floyd-Warshall, viene dada por los tres bucles anidados donde se calcula la distancia entre nodos siendo esta:

$$O_{\mathrm{FW}}(n^3) \tag{4}$$

Evaluación empírica

Para evaluar el comportamiento del algoritmo creado y comprobar si las conclusiones desarrolladas en la evaluación analítica son correctas se lleva a cabo una batería de pruebas con diferentes valores tanto para una misma n con diferentes p como a la inversa, a fin de comprobar como afecta cada una al rendimiento.



(a) *Con respecto a n* **(b)** *Con respecto a p*

Fig. 2: *Tiempo de ejecución del algoritmo FW*

Se observa que el costo computacional definido por T(n) se corresponde con los resultados obtenidos, ya que se observa en la gráfica que el comportamiento que sigue el algoritmo se aproxima a ser cúbico. Se observa también que la variación de p, la probabilidad de que dos nodos se encuentren conectados, apenas afecta a los resultados de $T(n)$.

Esto es coherente con la evaluación analítica realizada, ya que para Floyd-Warshall la complejidad temporal viene determinada por el número de nodos y no tanto por la densidad del propio grafo, ya que los bucles se ejecutan para cada nodo y no únicamente ara los nodos conectados, lo que muestra que es un algoritmo adecuado para encontrar caminos entre nodos independientemente de la densidad del grafo dado.

Por otro lado se ha observando que la variación del tiempo para computar una misma n en función de p es mínima y esta apenas tiene influencia, ya que la variación es de pocas decenas de microsegundos.

2. Estudio de K-CLIQUE

El algoritmo 'K-CLIQUE' permite, dado un grafo, determinar si existe un subgrafo de tamaño k totalmente conectado dentro del grafo original [3].

Evaluación analítica del algoritmo

Para analizar el coste computacional del algoritmo desarrollado, se puede dividir en tres secciones claramente diferenciadas para luego sumar sus costes y obtener un total, siendo dichas secciones:

- **Inicialización del Algoritmo**: Inicializar el algoritmo creando el subgrafo que se va a buscar y retornar el resultado final tiene un coste de 3.
- **Verificación de conexión entre nodos**: Esta función permite verificar si un nodo dado es está conectado con el resto de nodos del subgrafo que se está comprobando como CLIQUE, teniendo un coste de n en el peor caso, y al tener dentro dos comparaciones de tipo `IF` el coste ascendería a $5n$.
- **Búsqueda K-CLIQUE**: Esta función tiene un comportamiento recursivo llamándose así misma tratando de añadir cada nodo al subgrafo CLIQUE que se está buscando, comprobando si el nuevo nodo está conectado con todos los del CLIQUE actual. El coste de dicha función sería de n^k al tratarse de una función recursiva, y al contar con dos comparaciones `IF` el coste total sería de $7n^k$.

Una vez calculado los costes de cada sección se puede obtener el coste total. La función de verificación de conexión entre nodos se llama de manera anidada dentro de la función de verificación de K-CLIQUE, por lo que el coste total sería de:

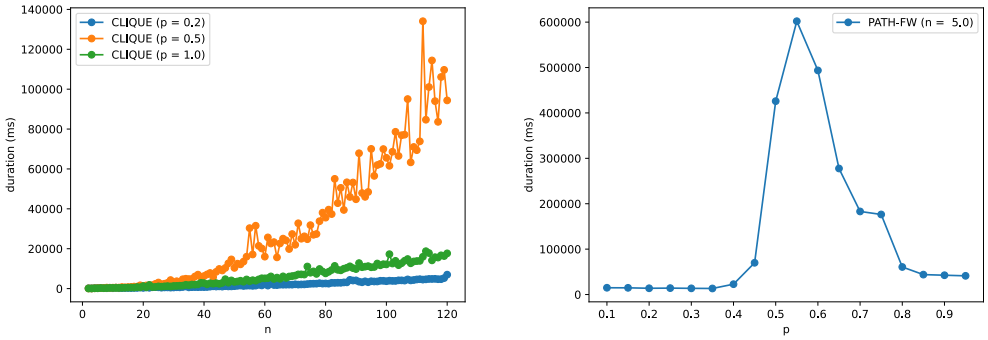
$$T_{\text{K-CLIQUE}}(n) = 7n^k + 5n + 3 \tag{5}$$

El coste en términos de complejidad temporal vendría marcado por el uso de una función recursiva:

$$O_{\text{K-CLIQUE}}(n^k) \tag{6}$$

Evaluación empírica

Para evaluar el comportamiento del algoritmo creado y comprobar si las conclusiones desarrolladas en la evaluación analítica son correctas se lleva a cabo una batería de pruebas con diferentes valores tanto para una misma n con diferentes p como a la inversa, a fin de comprobar como afecta cada una al rendimiento.



(a) *Con respecto a n* **(b)** *Con respecto a p*

Fig. 3: *Tiempo de ejecución del algoritmo CLIQUE*

Se observa claramente como el costo de $T(n)$ es mucho menor para grafos con densidades extremas. Con $p = 0{,}2$ se trata de grafos muy poco conectados donde el algoritmo rápidamente identificará que no es posible encontrar cliques de gran tamaño dentro del grafo. Para $p = 1{,}0$ se trata de un grafo totalmente conectado por lo que el algoritmo rápidamente encuentra que el CLIQUE máximo será el propio grafo.

Por otro lado el tener una $p = 0{,}5$ indica que el grafo tendrá una densidad media por lo que el algoritmo tendrá que probar un mayor número de posibles combinaciones para encontrar el mayor clique.

El gráfico en función de p muestra claramente los puntos expuestos donde el tiempo máximo de cómputo se obtiene para los valores donde el grafo generado tiene una densidad media que exigirá realizar más comprobaciones siendo los valores que exigirán un mayor tiempo aquellos comprendidos entre $p = 0{,}5$ y $p = 0{,}7$, mientras que se observa claramente que para los puntos extremos el tiempo disminuye.

3. Estudio K-SAT

El problema SAT (problema de satisfacibilidad booleana) [4] consiste en determinar si una fórmula booleana es *satisfacible* (existe al menos un caso en el que es cierta).

Para resolver este problema, lo transformaremos en otro problema equivalente, CLIQUE, y resolveremos ese problema en su lugar.

Evaluación analítica del algoritmo

Dado que el algoritmo consiste en transformar el problema en un grafo, y luego encontrar un K-CLIQUE dentro de ese grafo, aquí sólo nos centraremos en la transformación.

Asumiremos que el tamaño del problema, n es igual al número de variables del SAT, k, por lo que $n = k$.

Podemos dividir el algoritmo en dos partes:

- **Inicialización del algoritmo:** Coste de 7.
- **Obtener las variables:** Aquí se analiza la *string* con la definición del problema y se extraen las variables. Se usa una búsqueda mediante *regex*, la cual tiene una complejidad $O(m)$, siendo m el tamaño de la *string* de entrada. Como el tamaño de la entrada es proporcional al tamaño del problema, podemos asumir que tiene un coste n, además de otro coste n para almacenarlo en una estructura de datos, dando un coste total de $2n$.
- **Generación del grafo:** Para generar el grafo se usa un bucle ejecutado n^2 veces, una por variable, en el cual se calcula el contrario de la variable (coste 4), y se conecta el nuevo nodo al resto de nodos anteriores. Esta conexión tiene un coste de $4 \cdot \sum_{i=1}^{n} i = 2n^2 - 2n$, dado que por cada nodo hay que conectarse a los que ya están. El total del coste de este paso es, por lo tanto, $6n^2 - 2n + 7$.

Por lo tanto, el coste total de la transformación sería:

$$T_{\text{Transf}-\text{CLIQUE}}(k) = 6k^2 - 2k + 7 \tag{7}$$

Evaluación empírica

Dado que solo se pidió analizar el caso 3-SAT, para todas las pruebas usaremos $k = 3$. Debido a eso, se decidió hacer pruebas con distintos problemas, o iteraciones, para observar la diferencia de coste entre la resolución del problema y la transformación de un problema en otro.

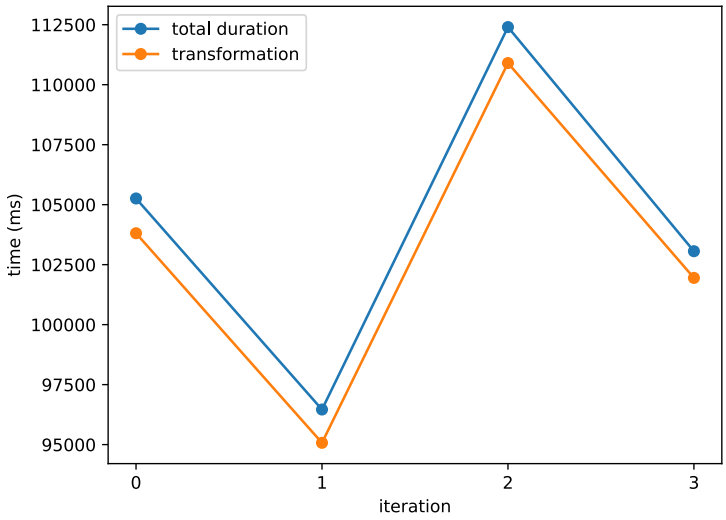


Fig. 4: *Tiempo de ejecución del algoritmo SAT*

En la Figura 4 se muestra el tiempo de ejecución de la transformación de 3-SAT a 3-CLIQUE y el tiempo total del problema, incluyendo la resolución del 3-CLIQUE.

Se observa que el coste de la transformación se lleva la mayor parte del coste de resolución de los problemas, principalmente porque se trata de trabajar con *strings*, lo cual es extremadamente lento comparado con operar otro tipo de estructuras de datos. La resolución del 3-CLIQUE, al ser de un tamaño tan pequeño, es extremadamente rápida.

Parte III

Conclusiones

El desarrollo de la presente práctica nos ha permitido poner en uso los conocimientos teóricos adquiridos no solo a la hora de evaluar el rendimiento y complejidad de algoritmos aplicados a grafos, sino además a llevar a cabo el desarrollo de los mismos buscando el equilibrio entre resultados y coste.

Además, mediante el análisis teórico y empírico y a través de los resultados obtenidos, hemos podido comprobar como en ocasiones los planteamientos teóricos pueden diferir de los resultados reales, siendo necesario un replanteo de la solución propuesta.

Por último, durante el desarrollo de la práctica hemos encontrado diversos desafíos siendo uno de los más relevantes el desarrollo la función generadora de grafos.

Ya que la generación de grafos afecta significativamente a como se evaluaran los diferentes algoritmos, en este caso se decidió que la función generadora de grafos recibiera como parámetro una probabilidad p, esta correspondería a la probabilidad de que entre dos nodos del grafo exista una arista, de esta forma se pueden evaluar los algoritmos para grafos con mayor o menor densidad de aristas fácilmente.

Parte IV

Bibliografía

Referencias

[1] J. E. Hopcroft, J. D. Ullman y A. V. Aho, «Data structures and algorithms,» en Addison-wesley Boston, MA, USA, 1983, vol. 175, cap. 6.5: Traversals of Directed Graphs.

[2] R. W. Floyd, *Algorithm 97: Shortest path*, eng, New York, NY, USA, 1962.

[3] I. M. Bomze, M. Budinich, P. M. Pardalos y M. Pelillo, «The maximum clique problem,» *Handbook of Combinatorial Optimization: Supplement Volume A*, págs. 1-74, 1999.

[4] J. Gu, P. W. Purdom, J. Franco y B. W. Wah, «Algorithms for the satisfiability (SAT) problem,» *Handbook of Combinatorial Optimization: Supplement Volume A*, págs. 379-572, 1999.

Parte V

Apéndices

Instalación y ejecución

Uso del simulador

Compila el simulador con CMake[1] y ninja-build[2]:

```
mkdir build
cd build
cmake .. -G Ninja
cmake --build .
```

Si estás en Windows, te recomendamos instalar WSL2[3] y ejecutar 'en Linux', o con GCC a través de MinGW-W64[4] (puedes encontrar binarios ya compilados en https://github.com/niXman/mingw-builds-binaries).

El simulador saca por `STDOUT` una *string* en formato JSON con los resultados de la simulación especificada. Para ver las opciones, usa el parámetro `--help`:

```
Usage
  src/p2 [options] [SAT-PROBLEM]

OPTIONS
  --n [5]                  size of the graph
  --p [0.5]                probability of an edge between two nodes
  --algorithm ["CLIQUE"]   algorithm to apply (PATH-DFS, PATH-FW, CLIQUE,
                           SAT-CLIQUE)
  --iterations [1]         number of iterations to execute
  --graph [true]           output the graph
  --help [false]           show a list of command-line options

ARGUMENTS
  SAT-PROBLEM              Problem string for the K-SAT algorithm,
                           using parenthesis, * (AND), + (OR) & - (NOT).
                           E.g.: "((c+b+-c)*(a+b+c)*(-a+b+c))"
```

[1] https://cmake.org/
[2] https://ninja-build.org/
[3] https://learn.microsoft.com/es-es/windows/wsl/install
[4] https://www.mingw-w64.org/

Ejecución de los tests en Python

Requiere Python[5] 3.10+.

1. Crea un *virtual enviroment* en la carpeta `.venv/`:

   ```
   python3 -m venv ./.venv
   ```

2. Activa el entorno:

   - Linux:

     ```
     source .venv/bin/activate
     ```

   - Windows (PowerShell):

     ```
     & .\.venv\Scripts\Activate.ps1
     ```

3. Instala las dependencias:

   ```
   pip install -r requirements.txt
   ```

4. Compila el simulador (ver Uso del simulador).

5. Ejecuta el *script* con:

   ```
   python3 src/test.py
   ```

[5] https://www.python.org/