Reflexión: Actividad integradora #4

Miguel Arriaga Velasco – A01028570

Campus Santa Fe

Programación de estructuras de datos y algoritmos fundamentales

26 de mayo del 2021

1. Grafos

Los grafos son estructuras de datos no lineal utilizados en la programación. Esta estructura se compone de nodos y arcos:

- Nodo: Elemento básico de información en un grafo.
- Arco: Liga que une dos nodos de un grafo (establece relación entre dos elementos).
- Subgrafo: Es un grafo que contiene a un subconjunto de Nodos y Arcos.

Terminología extra de grafos:

- Nodos Adyacentes: Nodos que tienen un arco que los conecta.
- Vecinos de un Nodo: Todos los nodos que son adyacentes al nodo.
- Camino o trayectoria: Secuencia de nodos, de tal forma que cada par de nodos son adyacentes.
- Trayectoria simple: Camino donde todos los nodos contenidos son distintos.

También se puede distinguir entre dos tipos de grafos:

- Grafo No-Dirigido: Los arcos en el grafo no tienen ninguna dirección particular, es decir, se consideran bidireccionales. Un arco de A a C es igual que uno de C a A.
- Grafo Dirigido: Los arcos tienen dirección. El primer elemento del arco es denominado el "origen" y el segundo el "destino". Un arco de A a C es diferente que uno de C a A.

Algunas de las representaciones de grafos más comunes son:

- Matriz de adyacencias: una matriz en 2D de tamaño N*N, en donde N es el número de nodos en el grafo. Se utilizan valores booleanos para determinar si existen arcos entre los nodos. Ver figura 1.

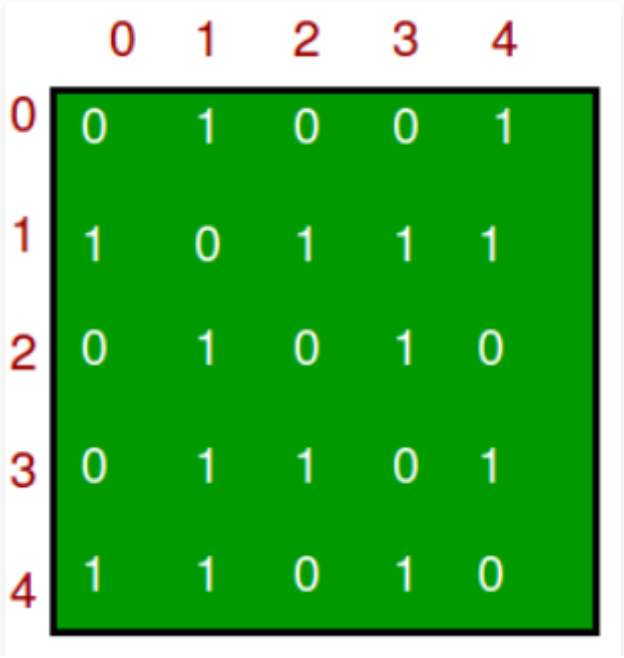

Figura 1. Representación visual de una matriz de adyacencias.

- Lista de adyacencias: se utiliza un arreglo de arreglos o un vector de vectores, en donde el primero almacena a los nodos y el segundo contiene los nodos con los que cada nodo tiene conexiones. Ver figura 2.

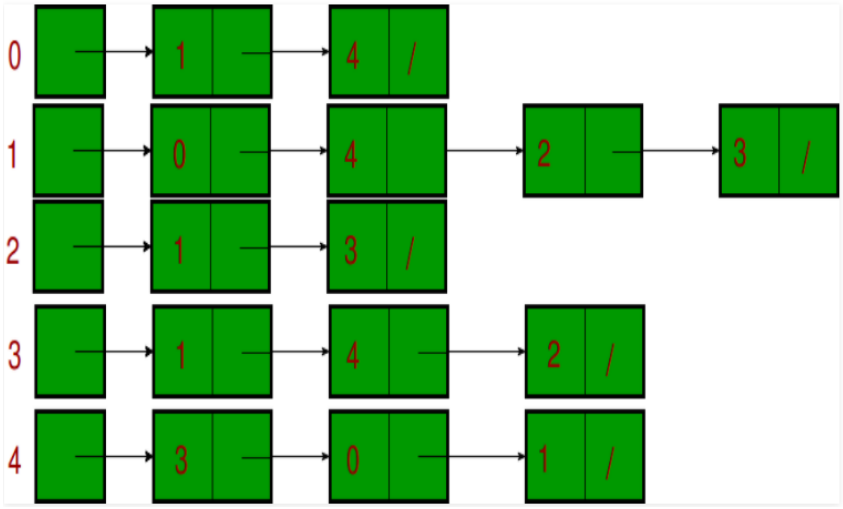

Figura 2. Representación visual de una lista de adyacencias.

- Lista de arcos: Existe una lista de Nodos Vértices y una lista de Arcos. Ver figura 3.

Figura 3. Nodos Vértices (izquierda) y arcos (derecha) en una lista de arcos.

Algunas de las operaciones que se pueden realizar en grafos son:

- Insertar nodo
- Insertar arco
- Borrar nodo
- Borrar arco
- Buscar nodo
- Recorrer grafo
    - Depth First Search (DFS): explora un camino a profundidad antes de continuar con el siguiente.
    - Breath First Search (BFS): va explorando el grafo por capas, cada vez más anchas.
- Ordenamiento topológico: ordenamiento de nodos linear para Grafos Dirigidos Acíclicos (DAG) de forma que por cada arco UV, el nodo U venga antes que V en el ordenamiento.
- Camino más corto: encontrar el camino más corto entre dos nodos, comúnmente se implementa con el algorítmo de Dijkstra, que es una versión modificada de BFS.
- Es bipartita: indica si un grafo es o no es bipartita (cuyos vértices se pueden separar en dos conjuntos disjuntos V1 y V2).

Aplicaciones de los grafos:

- Conectividad, redes de transporte
  - ¿Existe un camino entre dos Nodos?
  - Costo mínimo de conexión para todos los Nodos
  - Ruta óptima de un Nodo a otro
- Autómatas o Diagramas de Estado

Para esta actividad integradora, implementaremos una lista de adyacencias, para representar un grafo dirigido que indique las conexiones que tiene cada dirección IP.

1.1. Complejidad del grafo

Para la clase "bitacoraGraph", que es la clase encargada del almacenamiento y operaciones del grafo, se decidió sacrificar espacio de memoria, para tener un mejor rendimiento en tiempo de ejecución. Se utilizó la lista de adyacencias, que tiene una complejidad espacial de O(n+m), en donde n representa el número de nodos y m representa el número de arcos. Requiere más espacio de memoria que otras representaciones ya que duplica información. Sin embargo, la complejidad de memoria de la clase es aún mayor, ya que se decidió utilizar un mapa, para poder mantener cuenta de los índices de cada una de las direcciones IP. Este mapa tiene una complejidad espacial de O(n) y es el que está incluido en C++. Sin embargo, este mapa permitió que la creación de la lista de adyacencias se hiciera en tiempo lineal, se evalúa que este intercambio de memoria por tiempo es muy satisfactorio en este caso, ya que esta es la única bitácora que se desea almacenar.

La complejidad espacial final del grafo es de:

$$O(n) = 2n + m$$

En donde n representa el número de nodos y m representa el número de arcos. Una posible modificación al programa para reducir el espacio utilizado es con el uso de memoria dinámica. A través de apuntadores se podrían usar las mismas variables de IPs en la lista de adyacencias y en el mapa.

La complejidad temporal del método loadBitacoraGraphList es de:

$$O(n) = n$$

En donde n es el número de líneas del archivo de entrada.

2. Heap

Al igual que en la actividad integradora pasada, se utilizó un Heap para poder acomodar las direcciones IP. Sin embargo, en este caso, la prioridad en el Heap está dada por el número de conexiones (vecinos) de cada nodo. Además, otra diferencia con la actividad pasada es que en esta ocasión se utilizó un Min Heap, para que a la hora de utilizar el método Heap Sort, las IPs queden ordenadas en orden de mayor a menor conexiones.

3. Heap Sort

Según GeeksForGeeks, 2021: "Heap Sort es una técnica de ordenamiento basada basada en la estructura de datos de un Binary Heap. Es similar al ordenamiento por selección donde primero encontramos el elemento mínimo y colocamos el elemento mínimo al principio. Repetimos el mismo proceso para el resto de los elementos. Para ordenar la bitácora se utilizó este algorítmo.

3.1. Análisis de complejidad

Complejidad espacial:

Heap sort realiza un ordenamiento “in-place”, ya que utiliza la misma estructura de datos que guardaba a los datos sin ordenar, para guardar los datos ordenados. Es por esto que su complejidad espacial es constante: O(1).

Complejidad temporal:

Para analizar la complejidad temporal de nuestro algorítmo, primero debemos de analizar la complejidad que nos toma el método “heapifydel” (figura 1), que se llama dentro del método “heapSort”.

```
void Heap::heapifydel(int n, int i)
{
    int smallest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && vect[l].getNumNeighbors() < vect[smallest].getNumNeighbors())
        smallest = l;
    if (r < n && vect[r].getNumNeighbors() < vect[smallest].getNumNeighbors())
        smallest = r;
    if (smallest != i) {
        swap(i,smallest);
        heapifydel(n,smallest);
    }
}
```

Figura 4. Algoritmo “heapifydel”

Este algorítmo lo podemos analizar utilizando las técnicas de análisis empírico de algoritmos recursivos. Se puede observar que solo hay una llamada recursiva, y el parámetro de control “smallest”, se multiplica por un valor constante (2), en cada llamada. Esto nos dice que la complejidad de este método es de:

$$O(n) = (log_2 n)$$

Ahora observemos el algorítmo heapSort (figura 2).

```
void BitacoraHeap::heapSort()
{
    // One by one extract an element from heap
    for (int i = size - 1; i >= 0; i--) {
        // Move current root to end
        swap(0, i);
        // call max heapify on the reduced heap
        heapifydel(i,0);
    }
}
```

Figura 5. Algoritmo “heapSort”

Se puede observar que hay un loop que va desde cero, hasta el tamaño del heap menos uno. Observemos que la operación más costosa dentro del heap es “heapfydel”, ya que “swap” corre en tiempo constante. Ya sabemos que el costo de “heapifydel” es de:

$$O(n) = (log_2 n)$$

Esta operación se repite n-1 veces, quitamos las constantes y obtenemos una complejidad temporal final de:

$$O(n) = (nlog_2 n)$$

Esta complejidad aplica para los peores, mejores y casos promedios. Además, es la misma que la de los algoritmos QuickSort y MergeSort, que se utilizaron en entregas pasadas.

Conclusión y reflexión sobre grafos para la situación problema

La situación problema de este curso es sobre los ataques cibernéticos y la identificación de páginas infectadas. Los grafos son una de las estructuras de datos más utilizadas para encontrar caminos y optimizar procesos. En esta situación problema, un grafo podría ser de gran ayuda para rastrear IPs sospechosas y así identificar de dónde vienen los ataques. Por ejemplo, se podría utilizar alguno de los recorridos (BFS o DFS) para saber qué IPs se conectan entre ellas comúnmente y poder terminar esas conexiones. Además, esto podría ser de utilidad para rastrear IPs de hackers en caso de que estén ocultándola con una VPN o algo similar.

En un problema de esta naturaleza, un grafo es muy eficiente. Se puede recorrer de varias formas y ya existen muchos algoritmos que estan optimizados, por ejemplo, el de Dijkstra. Especialmente pueden ser muy poderosos al combinarlos con otros tipos de estructuras de datos. En este caso se utilizó a la par con un Heap y un Map y esto se logró con una increíble eficiencia temporal. Analizar patrones en los datos y combinar estructuras de datos para escribir buen código, es una habilidad necesaria en esta área.

Referencias

Presentaciones por el Dr. Eduardo Arturo Rodríguez Tello, durante la materia de Programación de estructuras de datos y algoritmos fundamentales

GeeksForGeeks (2021) “Heap Sort”. Recuperado desde: https://www.geeksforgeeks.org/heap-sort/

GeeksForGeeks (2021) “Topological Sorting”. Recuperado desde: https://www.geeksforgeeks.org/topological-sorting/

GeeksForGeeks (2018) “Graph Data Structure And Algorithms”. Recuperado desde: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

GeeksForGeeks (2021) “Graph and its representations”. Recuperado desde: https://www.geeksforgeeks.org/graph-and-its-representations/