Algoritmos e Estruturas de Dados

3ª Série

Contar triângulos numa rede

Nº 50493 Bernardo Pereira

Nº 50512 António Paulino

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2022/2023

9/6/2023

Índice

1. Introdução

Pretende-se desenvolver uma aplicação que permita calcular o número de triângulos num grafo, seja este direcionado ou não direcionado. A aplicação deverá inicialmente receber o nome do ficheiro de input (ex: exemplo.txt) e o nome do ficheiro de um dos ficheiros de output (ex: listCounting.txt). De seguida deve retornar o número de total de triângulos e listar para o ficheiro de output especificado, para cada vértice, o número total de triângulos a que pertence. Por fim deve possibilitar que o utilizador ou insira um k e um nome de um outro ficheiro de output, que lista para esse novo ficheiro os k vértices que façam parte do maior número de triângulos existentes, ou termine a aplicação.

2. Contar triângulos numa rede

2.1 Análise do problema

O problema é constituído por um ficheiro em que cada linha contém uma aresta/ligação do grafo. O conjunto de todas as linhas do ficheiro define todas os vértices *V* e ligações *A* do grafo.

As operações necessárias são as seguintes:

1. Leitura do ficheiro de input e criação do grafo que este representa.
2. Criação de um ficheiro output onde se escreverão os vértices inicialmente.
3. Contagem dos triângulos associados a cada vértice.
4. Escrita dos triângulos associados a cada vértice no ficheiro output.
5. Continuação do corrimento do programa de acordo com o enunciado.

A abordagem implementada para as operações foi a seguinte:

1. É criado um leitor para o ficheiro, e são percorridas as linhas do mesmo. Para cada linha que representa uma ligação do vértice A para o vértice B, são adicionados os vértices A e B associados a essa ligação ao grafo, e é adicionada a ligação de A para B ao grafo. Caso o grafo não seja direcionado, também é adicionada a ligação de B para A ao grafo. Este processo repete-se até serem percorridas todas as linhas do ficheiro.

2. É criado um "escritor" para escrever o output do programa.

3. Para cada vértice A, são visitados os seus vértices adjacentes B. De seguida, são visitados os vértices C adjacentes aos vértices B. Se existir uma ligação de A para B, B para C, e C para A, e nenhuma das adjacências dos vértices for o próprio vértice, então existe um triângulo. Nesta situação, são incrementados os valores de contagem de triângulo para os três vértices. Depois de serem verificados todos os possíveis triângulos associados a um vértice, esse vértice é colocado com a cor preta. Sempre que for encontrado um triângulo com um vértice adjacente preto, não é contado o triângulo, o que evita a contagem triplicada de cada triângulo. Apesar disto, é necessário dividir por dois a contagem de triângulos para cada vértice quando o grafo não é direcionado, já que o triângulo é contado uma vez para cada sentido. A contagem de cada triângulo é guardada num HashMap e vão sendo sucessivamente adicionadas as contagens associadas a cada vértice a uma Priority Queue ordenada de acordo com o índice dos vértices (Max-Heap). A contagem de triângulos total é dividida por 3, já que resulta da soma total de todos os triângulos associados a cada vértice.

4. São retirados da Priority Queue todos os elementos e são escritos no ficheiro Output. De seguida é necessário repor todos os elementos, para a criação de uma nova Priority Queue que possibilita a operação 5.'

5. É criada uma nova Priority Queue ordenada de acordo com a quantidade de triângulos associada a cada vértice (Max-Heap) e são adicionados a esta Priority Queue todos os elementos da Priority Queue referida nos pontos anteriores. De seguida são lidos k e o nome do ficheiro fornecidos pelo utilizador, e são retirados da Priority Queue os primeiros k elementos, que são escritos no ficheiro. Após a escrita, são repostos os elementos na Priority Queue para suportar próximas operações que o utilizador queira realizar. Quando k for 0, o programa termina.

2.2 Estruturas de Dados

Neste trabalho foram utilizadas as seguintes estruturas de dados:

HashSet

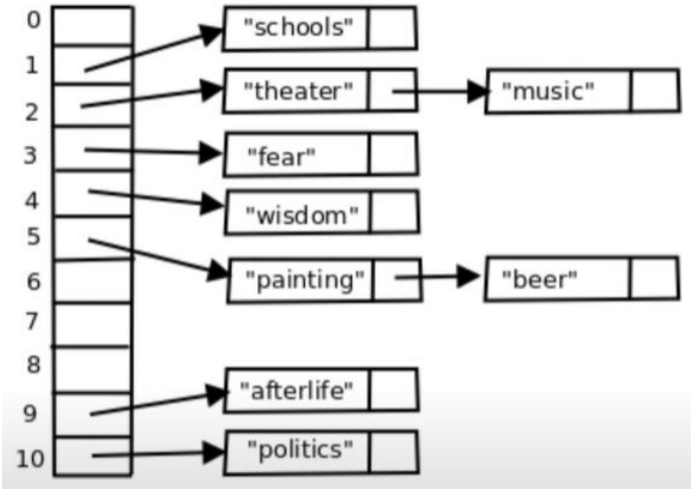

Figura 1: Exemplo da estrutura de dados *HashSet.*

O HashSet é uma estrutura de dados que armazena elementos de forma não ordenada, por meios de indexação através de hash code de uma key. Os elementos são únicos, e não podem ser repetidos. Esta estrutura de dados pode ser utilizada para evitar que sejam armazenados elementos duplicados, e para que a procura de elementos seja mais eficiente.

HashMap

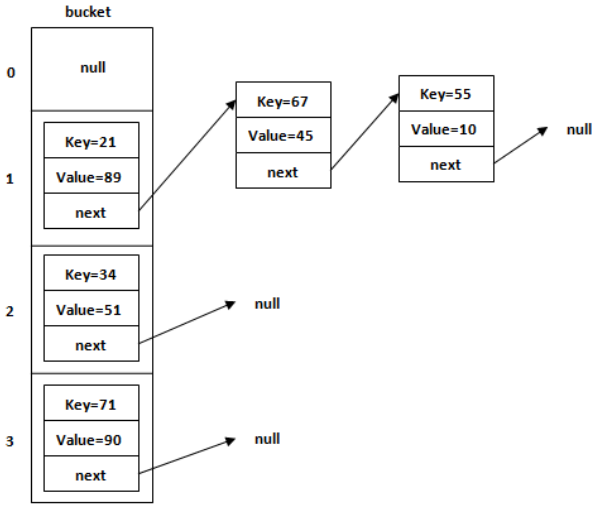

Figura 2: Exemplo da estrutura de dados *HashMap*.

A estrutura de dados HashMap funciona de forma semelhante a HashSet, mas armazena pares chave-valor. Esta estrutura de dados pode ser utilizada quando é necessário associar valores a cada elemento, separadamente da chave.

Mutable List

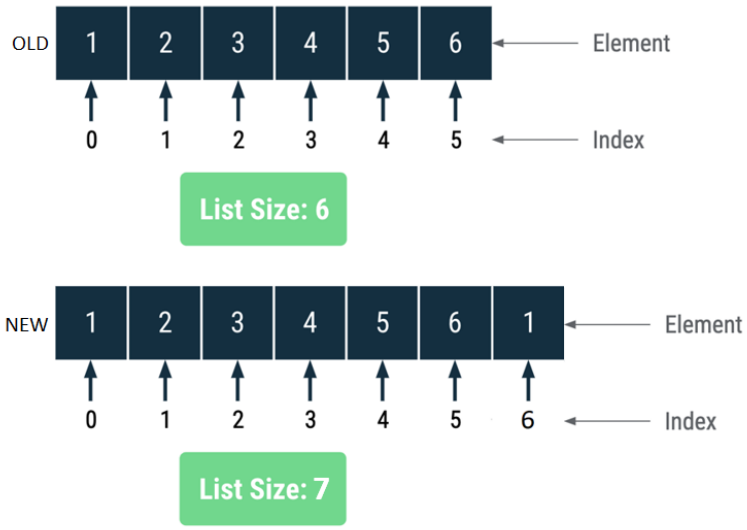

Figura 3: Exemplo da estrutura de dados Mutable List.

A Mutable List é uma estrutura de dados que representa uma lista mutável, ou seja, uma lista cujos elementos podem ser alterados, removidos ou adicionados. Esta estrutura de dados é utilizada quando é necessário fazer alterações a uma lista durante o corrimento de um programa.

Grafo

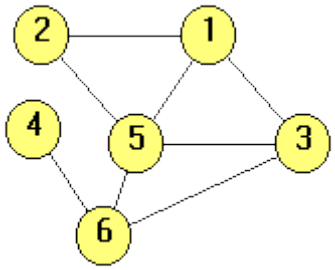

Figura 4: Exemplo da estrutura de dados Grafo

Um grafo é uma estrutura de dados composta por um conjunto de vértices e um conjunto de arestas. Os grafos são utilizados para representar relações entre objetos. Podem ser direcionados ou não, ter valores associados a cada vértice e ter um custo ou peso associado a cada aresta. Um exemplo comum de utilização desta estrutura de dados é em redes sociais.

Priority Queue

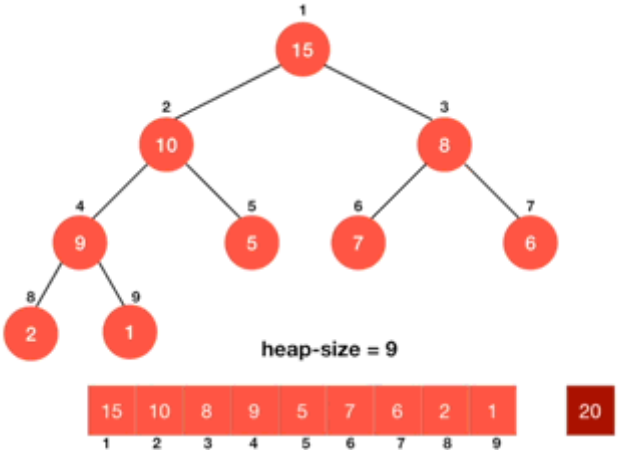

Figura 5: Exemplo da estrutura de dados Priority Queue (heap)

Uma Priority Queue é uma estrutura de dados que mantém uma ordem específica nos elementos, onde o elemento com maior prioridade fica no topo. A prioridade pode ser definida através de uma função de comparação ou por um valor associado a cada elemento. Uma Priority-Queue pode ser utilizada quando têm que ser removidos e adicionados elementos a uma fila durante o corrimento do programa, sempre preservando a prioridade com base na função de comparação.

2.3 Algoritmos e análise da complexidade

Principais algoritmos utilizados nesta implementação:

Priority Queue (Max-Heap), foram utilizadas as funções add e poll, que têm complexidade temporal log(n). As análises dos algoritmos pertencentes a esta estrutura de dados foram realizadas na primeira série.

HashSet/HashMap, foram utilizadas as funções add e contains, que têm complexidade temporal $O(1)$. As análises dos algoritmos pertencentes a esta estrutura de dados foram realizadas na segunda série.

Graph, foram implementadas as funções getEdge, getVertex, addVertex, addEdge, todas com complexidade temporal O (1), descritas abaixo em pseudocódigo.

```
getEdge(id, idAdj):
        vertex = vertices[id]
        if (vertex == null) return null
        if (vertex.adjacencies.contains(Edge(id, idAdj))) return Edge(id, idAdj)
        else return null

getVertex(id):
        return vertices[id]

addEdge(id, idAdj):
        sourceVertex = vertices[id]
        if (sourceVertex == null) return null
        edge = Edge(id, idAdj)
        sourcevertex.adjacencies.add(edge)
        return idAdj

addVertex(id, D):
        existingVertex = vertices[id]
        if(existingVertex != null) return null
        newVertex = Vertex(id, d)
        vertices[id] = newVertex
        return newVertex.data
```

Para a implementação do algoritmo de contagem de triângulos, foram implementadas as funções createGraphFromFile, getTriangleCount, e writeVerticesWithMostTriangles, descritas abaixo.

```
createGraphFromFile(fileName, delimiter, skip, isDirected):
    reader = openFile(fileName)
    graph = new GraphStructure

    for i in 0 until skip:
        reader.readLine()

    line = reader.readLine()

    while line is not null:
        vertex1, vertex2 = split(line, delimiter).toInt()
        graph.addVertex(vertex1, 0)
        graph.addVertex(vertex2, 0)
        graph.addEdge(vertex1, vertex2)
        if not isDirected:
            graph.addEdge(vertex2, vertex1)
        line = reader.readLine()

    reader.close()

    return graph
```

Este algoritmo tem complexidade temporal $O(n)$, em que n é a quantidade de linhas pertencentes ao ficheiro. Isto porque o ficheiro percorre todas as linhas, e para cada realiza operações constantes. Em termos de complexidade espacial é $O(V + A)$, em que V é a quantidade de vértices pertencentes ao grafo especificado pelo ficheiro e A é a quantidade de arestas/ligações, já que cria um grafo com V vértices e todas as ligações que lhe pertencem.

```
getTriangleCount(graph, isDirected):
    vertexTriangleCounts = MutableMap<Int, Int>()
    triangleCount = 0
    pq =  PriorityQueue<Pair<Int, Int>>(compareBy{ it.first })

    for each vertex in graph:
        neighbors = graph.getVertex(vertex.id).getAdjacencies()

        for each neighbor1 in neighbors:
            if (neighbor1.adjacent == neighbor1.id) continue

            vertex1 = graph.getVertex(neighbor1.adjacent)

            if (vertex1.clr == COLOR.BLACK) continue

            neighbors2 = vertex1.getAdjacencies()

            for each neighbor2 in neighbors2:
                if (neighbor2.adjacent == neighbor2.id or graph.getVertex(neighbor2.adjacent).clr == COLOR.BLACK)
                        continue

                if graph.getEdge(neighbor2.adjacent, vertex.id) is not null and neighbor2.adjacent != vertex.id:
                    vertexTriangleCounts[vertex.id] = vertexTriangleCounts[vertex.id] + 1
                    vertexTriangleCounts[neighbor1.adjacent] = vertexTriangleCounts[neighbor1.adjacent] + 1
                    vertexTriangleCounts[neighbor2.adjacent] = vertexTriangleCounts[neighbor2.adjacent] + 1

        vertex.clr = COLOR.BLACK
        divisor = if isDirected 1 else 2
        count = vertexTriangleCounts[vertex.id] / divisor
        pq.add(Pair(vertex.id, count))
        triangleCount += count

    print("Triangulos : " + triangleCount/3)
    return pq
```

Este algoritmo tem complexidade temporal $O(V\ A^2)$, em que V é a quantidade de vértices e A a quantidade de arestas. Isto porque para cada vértice, o algoritmo itera sobre os vizinhos e vizinhos dos vizinhos desse vértice, para verificar se formam triângulos. As restantes operações são constantes, com exceção à adição de elementos à Priority Queue, que tem complexidade temporal $O(log\ n)$.

Em termos de complexidade espacial, este algoritmo tem complexidade O(V), em que V é a quantidade de vértices presentes no grafo. Isto porque para cada vértice, é armazenada a quantidade de triângulos a que pertence.

```
writeVerticesWithMostTriangles(fileName, vertices, k):
    writer = createWriter(FileName)
    result = emptyList()

    for i in 0 until k:
        tmp = vertices.poll()
        writeToFile(writer, tmp.first + " -> " + tmp.second)
        writeNewLineToFile(writer)
        result.add(Pair(tmp.first, tmp.second))

    vertices.addAll(result)

    writer.close()
```

A complexidade temporal deste algoritmo é $O(k)$, em que k é a quantidade de vértices a escrever no ficheiro, ordenados de acordo com a quantidade de triângulos a que pertencem.
A complexidade espacial deste algoritmo é O(k). É criada uma lista temporária que armazena os elementos removidos da Priority Queue de forma a que seja possível repor os elementos, após estes serem removidos da Priority Queue e escritos no ficheiro.

Algoritmo Principal:

```
main(args):
    inputFile = args[0]
    outputFile = args[1]
    isDirected = false

    graph = createGraphFromFile(inputFile, ",", 0, isDirected)

    triangleCounts = getTriangleCount(graph, isDirected)

    writeVerticesWithMostTriangles(outputFile, triangleCounts, triangleCounts.size)

    sortedTriangles = PriorityQueue(compareByDescending { it.second })
    sortedTriangles.addAll(triangleCounts)

    file = outputFile
    k = -1

    while k != 0:
        print("Introduza um número positivo k (max =  sortedTriangles.size  ) ou 0 se quiser terminar a aplicação")
        args = readLine().trim().split(' ')
        k = args[0].toInt()
        if args.size == 2:
            file = args[1]
        if k > 0:
            writeVerticesWithMostTriangles(file, sortedTriangles, k)
```

Este algoritmo tem complexidade temporal $O(V\ A^2)$, em que V é a quantidade de vértices presentes no grafo, e A a quantidade de arestas. Isto deve-se ao facto de que a operação com maior custo é a de contar os triângulos pertencentes a cada vértice.

Este algoritmo tem complexidade espacial $O(V\ +\ A)$, em que V é a quantidade de vértices e A a quantidade de arestas/ligações, já que tem de ser criado um grafo com todos os vértices e ligações que lhe pertencem, e todas as restantes estruturas de dados em que são armazenados elementos têm complexidade O(V), já que são utilizadas para armazenar a quantidade de triângulos a que cada vértice pertence.

3 Avaliação Experimental

Para testar os algoritmos, serão utilizados ficheiros com quantidades de arestas diferentes, sendo que esses ficheiros foram testados por ordem crescente de número de arestas.

Os testes foram realizados numa máquina com o processador Intel I7 12650h, 16 Gb ram 4800mhz.

Ficheiro	p2p-Gnutella05.txt	p2p-Gnutella04.txt	p2p-Gnutella25.txt	facebook_combined.txt	p2p-Gnutella30.txt	Wiki-Vote.txt	p2p-Gnutella31.txt	CA-AstroPh.txt	Email-EuAll.txt	Slashdot0811.txt	Slashdot0902.txt	com-dblp.ungraph.txt	twitter_combined.txt	com-youtube.ungraph.txt
N Arestas	31839	39994	54705	88234	88328	103689	147892	198110	420045	905468	948464	1049866	1768149	2987624
Tempo (ms)	202.5816	230.6355	262.5496	759.7249	373.8548	1197.191	581.2415	1580.566	2868.98	5491.205	5861.693	4326.395	13870.57	32330.1

Tabela 1: Resultados do tempo em milissegundos de execução de algoritmos de ordenação considerando várias amostras. Média de 5 valores por amostra

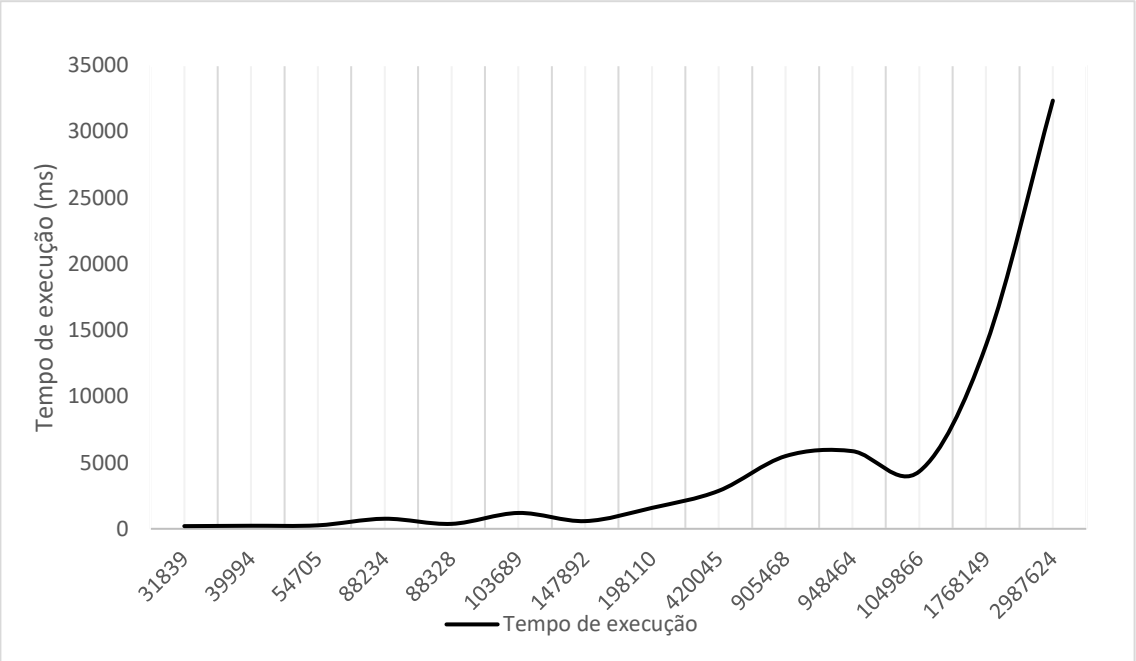

Figura 2: Tempos de execução do algoritmo de contagem de triângulos em função do número de arestas.

A implementação aparenta ter uma complexidade temporal quadrática, quando comparada com o gráfico da figura 6. Verifica-se a análise dos algoritmos na secção 2.3.

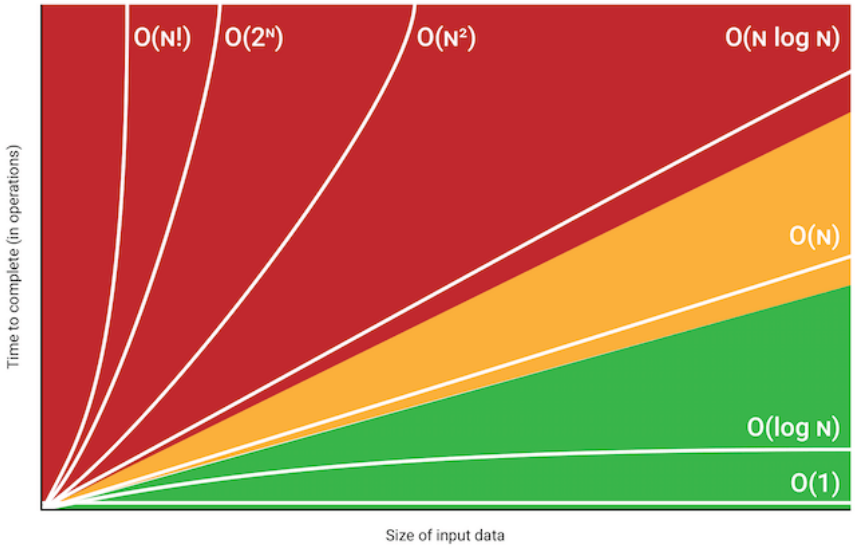

Figura 6: Gráfico de notação O-Grande

4 Conclusões

Neste trabalho foi desenvolvida uma aplicação que permite calcular o número de triângulos num grafo. Foi possível implementar um algoritmo com complexidade temporal aproximadamente quadrática $O(V\,A^2)$ e complexidade espacial aproximadamente $O(V + A)$, em que V é o número de vértices e A a quantidade de arestas. Foi implementado um algoritmo que visita as adjacências de cada vértice e verifica a existência de triângulos. Na implementação deste algoritmo foram utilizadas as estruturas de dados Grafo, HashSet, HashMap, Priority Queue, e Mutable List.