Algoritmos e Estruturas de Dados

3ª Série

(Problema)

Análise de ocorrência de menções em tweets

N. 50548 Nome: Gonçalo Amigo
N. 50518 Nome: Rita Monteiro
N. 50500 Nome: Humberto Carvalho

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2023/2024

05/06/2024

Índice

1. Introdução.....3

2. GenomeSequencing.....4

2.1 Grafo de Bruijin.....5

2.2 Percurso Euleriano.....5

2.2.1 Teoria do Algoritmo de Hierholzer.....6

2.2.2 Implementação.....6

2.2.3 Estrutura de Dados:.....6

2.2.4 Algoritmo.....7

2.2.5 Análise do Código.....7

2.2.6 Potenciais Melhorias.....7

2.2.7 Conclusão.....7

2.3 Estruturas de Dados utilizadas.....8

3. Avaliação Experimental.....9

4. Conclusões.....11

1. Introdução

O projeto a implementar baseia-se no desenvolvimento duma aplicação na linguagem *Kotlin* que analisa a sequência dum código genético de seres vivos, denominado por Genoma. Esta implementação através do grafo de Bruijn e do percurso euleriano no mesmo.

O relatório divide-se em diversas seções, todas com o propósito de dar um maior entendimento ao utilizador da aplicação desenvolvida. Sendo essas seções:

- Descrição do problema em mãos
- Análise de cada método
- Quais as estruturas de dados mais adequadas para a implementação dos métodos
- Tempo de execução
- Análise experimental dos algoritmos implementados com testes
- Conclusão

2. GenomeSequencing

Foi proposto o desenvolvimento de uma aplicação que permita de um dado valor inteiro k, e um ficheiro com a listagem dos k-mers, obter o grafo de *Bruijin* e o percurso Euleriano que representa o genoma.

De forma a implementar o grafo de *Bruijin* e o percurso Euleriano inicialmente é necessário a criação do ADT Grafo, que habilita à implementação da estrutura grafo em memória. Para tal implementação foi necessário seguir a interface descrita no ficheiro *Graph.kt* que apresenta todos os métodos a desenvolver no âmbito da class *GraphStructure*, sendo esses:

1. Class Vertex
2. função addVertex()
3. função addEdge()
4. função getVertex()
5. iterador

Class Vertex:

A class Vertex, representa um vértice no grafo. Para a implementação desta classe foi tido em conta a interface Vertex que descreve os métodos e variáveis referentes a vértices, sendo essas a propriedade id, que identifica o vértice (é o valor do vértice em questão) e a função getAdjacencies() que retorna uma lista mutável com todas as adjacências ao vértice com o id específico. Tendo assim implementada a class Vertex.

Função addVertex():

A função addVertex(), retorna um booleano que indica se é possível adicionar ou não um vértice ao grafo. Para se adicionar um vértice ao grafo, tem de se saber inicialmente se o mesmo já pertence ou não ao grafo, caso não pertença será adicionado ao grafo, incrementando o tamanho *size* e retornando verdadeiro, caso o vértice já esteja presente no grafo não será adicionado novamente, retornando assim falso.

Função addEdge():

A função addEdge() tem como principal objetivo adicionar uma aresta ao grafo caso seja possível. Tendo em conta os dois identificadores dos vértices, recebidos em parâmetro, dependendo se os vértices já se encontram no grafo, será criada uma insistência de adjacência na lista correspondente às adjacências do primeiro identificador e do segundo.

Função getVertex():

A função getVertex irá procurar no HashMap que representa o grafo em questão, o vértice identificado através do id recebido em parâmetro.

Função Iterator():

A função iterator() tem como principal objectivo implementar um iterador de modo a percorrer a lista de vértices adjacentes de cada *Vertex<I>.*

2.1 Grafo de Bruijin

Para a implementação do grafo de Bruijin, é necessário ter em conta o conceito do mesmo. O grafo de Bruijin é definido como uma estrutura que implementa um grafo ao qual os vértices representam os diferentes K-mers e as adjacências dos vértices formam um K-mer igualmente, apenas de diferente tamanho. No desenvolvimento deste método que implementa o Grafo foi necessário inicialmente fazer a leitura do ficheiro que contém a lista dos *K-mers* e posteriormente para a construção do mesmo, cada vértice corresponde a um prefixo de tamanho no intervalo de 1 a K ou a um sufixo no intervalo de 0 a K - 1, sendo chamada a função *addVertex* em ambos os casos. Quando ambos os vértices estiverem no grafo é inserida a aresta correspondente entre os dois graças ao método *addEdge* e posteriormente escrito num ficheiro de output (todas as escritas e leituras de ficheiros foram feitas através a biblioteca *java.io.File* da linguagem Java).

2.2 Percurso Euleriano

Para encontrar um percurso euleriano em um grafo é necessário aplicar o algoritmo de Hierholzer. Um percurso euleriano é um caminho percorrido em um grafo que visita todas as arestas exatamente uma vez.

2.2.1 Teoria do Algoritmo de Hierholzer

O algoritmo de Hierholzer é utilizado para encontrar percursos eulerianos em grafos. O mesmo é aplicável em grafos que são fortemente conexos e onde todos os vértices possuem grau par. O algoritmo pode ser resumido nas seguintes etapas:

- Seleção do vértice inicial:
    - Inicia-se em qualquer vértice do grafo.
- Construção do caminho:
    - Se o vértice atual possui arestas não visitadas, uma aresta é escolhida e removida, e o vértice adjacente é empilhado.
    - Se o vértice atual não possui arestas não visitadas, ele é removido da pilha e adicionado à lista do percurso.
- Repetição:
    - As etapas são repetidas até que a pilha esteja vazia.

2.2.2 Implementação

O código descrito no ponto 2.2.4 descreve o algoritmo de Hierholzer para encontrar um percurso euleriano em um grafo representado pela estrutura *GraphStructure<String>*.

2.2.3 Estrutura de Dados:

➔ EulerianPath:
- ◆ Lista mutável que armazenará o percurso euleriano final.

➔ Stack:
- ◆ Pilha que será usada para manter o controle dos vértices durante a construção do percurso.

➔ initialVertex:
- ◆ Seleciona o vértice inicial do grafo.

2.2.4 Algoritmo

O código começa por criar duas listas de string vazias em que uma é a lista que armazenará o percurso euleriano final e a outra é a pilha utilizada para manter o controle dos vértices durante a construção do percurso. De seguida selecionamos um vértice inicial arbitrário a partir do grafo, consequentemente adicionamos o vértice inicial à pilha, em que por fim entra no ciclo principal do algoritmo que continua até que a pilha esteja vazia. Posteriormente obtemos o objeto vértice correspondente ao último elemento adicionado no grafo, e que analisa se o vértice tem arestas não visitadas, caso haja arestas não visitadas, remove a primeira aresta e empilha o vértice adjacente, caso contrário se não houver arestas não visitadas, remove o vértice do topo da pilha e adiciona à lista do percurso euleriano.
Ao terminar o ciclo, retorna o percurso euleriano na ordem correta, invertendo a lista.

2.2.5 Análise do Código

O código implementa eficientemente o algoritmo de Hierholzer com uma complexidade de tempo linear em relação ao número de arestas do grafo. A estrutura de dados utilizada (lista mutável para a pilha e o percurso) é apropriada para as operações necessárias (adição e remoção de elementos).

2.2.6 Potenciais Melhorias

- ➔ Verificação de Conectividade:
  - ◆ Adicionar uma verificação inicial para garantir que o grafo é fortemente conexo e que todos os vértices possuem grau par antes de iniciar o algoritmo.

- ➔ Tratamento de Erros:
  - ◆ Implementar manejo de exceções para lidar com possíveis erros, como grafos desconexos ou vértices inexistentes.

2.2.7 Conclusão

O algoritmo de Hierholzer é uma solução robusta para encontrar percursos eulerianos em grafos que atendem aos critérios necessários. A implementação fornecida em Kotlin é direta e eficaz, seguindo de perto a lógica teórica do algoritmo. Este relatório forneceu uma análise detalhada da implementação, destacando sua eficiência e sugerindo melhorias potenciais para garantir uma aplicação mais abrangente e segura.

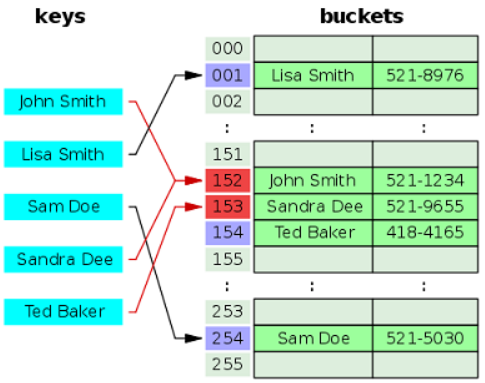
2.3 Estruturas de Dados utilizadas

As estruturas de dados utilizadas no âmbito do projeto foram o HashMap, MutableList e conceito de Grafo.

HashMap:

- Como já foi referido no âmbito da série 2, um HashMap é uma estrutura que armazena valores de acordo um par entre uma chave e um valor associado a essa mesma chave (<I, Vertex<I>?>), tendo uma posição na tabela, de acordo com uma função de dispersão, que é calculada de acordo com um cálculo matemático.

- A estrutura HashMap em termos de complexidade apresenta valores bastante eficientes, sendo essa a razão para o qual foi escolhida para a implementação base do Grafo definido pelo Abstract Data Type GraphStructure (já explicado anteriormente)

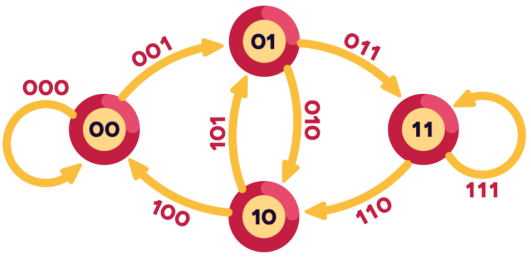


MutableList:

- A estrutura MutableList permite a implementação de uma lista ao qual se pode alterar os elementos e tamanho da mesma. Desta forma, é possível de um modo eficiente adicionar elementos sem ter que criar novas instâncias de uma lista como acontece nas lista imutáveis.

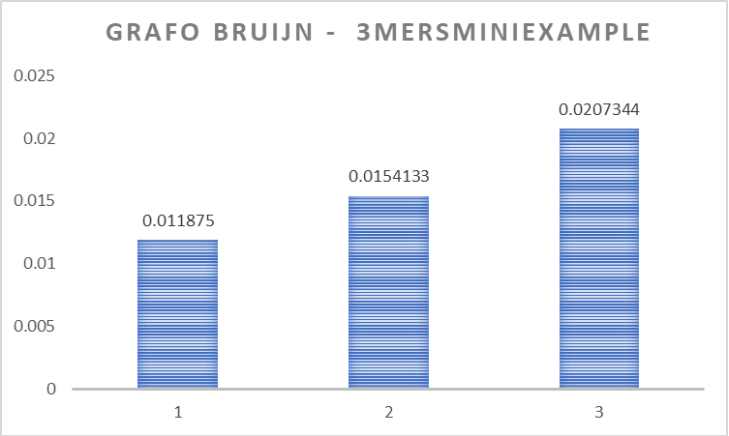
Grafo:

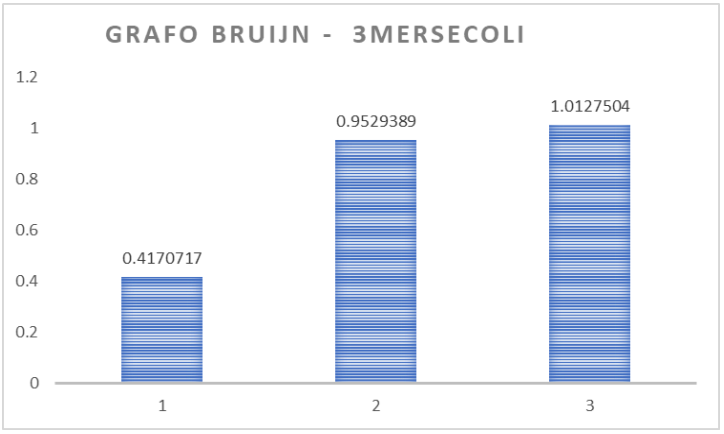
- Um grafo é uma estrutura que permite a ligação entre diferentes nós, desta forma, de modo a chegar a um nós específico existem vários caminhos aos quais podem dar ao nó desejado. Por estas razões o grafo é bastante utilizado.

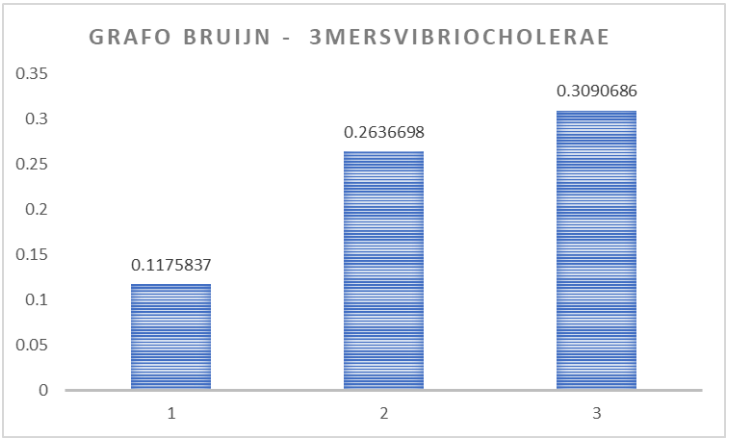


3. Avaliação Experimental

De acordo com as implementações para o grafo de Bruijn com auxílio da biblioteca *kotlin.time.Duration* foi possível saber o tempo que demora em média a ser executado ambos os algoritmos. Nos seguintes gráficos encontram-se as demonstrações referentes aos tempos como também para ficheiros de diferentes dimensões.







As unidades de cada gráfico estão em segundos. Desta forma é possível observar que mesmo para ficheiros com um enorme valor de k-mers o algoritmo implementado em relação tanto ao grafo de Bruijn é bastante eficiente, mantendo sempre valores numa média constante sem grandes alterações.

4. Conclusões

Em suma, com o desenvolvimento deste trabalho foi possível adquirir competências no âmbito de diferentes estruturas de dados, principalmente dos grafos e quão fácil foi a gestão e procura das arestas e vértices do mesmo, tanto como ter o conhecimento sobre esta estrutura que hoje em dia é bastante utilizada nas redes sociais, em termos de implementação da relação de cada usuário aos seus amigos próximos.

Em termos gerais, como grupo, o trabalho desenvolvido colocou em prática novas estruturas de dados que previamente não teriam sido utilizadas no âmbito de outras disciplinas e diferentes maneiras de pensar uma vez que neste momento, como programadores, temos de pensar numa implementação que faça o pretendido mas não só, que também seja eficiente e que utilize menos recursos.