Trabalho Prático
Implementação Grafos em C

Enrique George Rodrigues Nº 28602

Instituto Politécnico do Cávado e do Ave
Licenciatura Engenharia de Sistemas Informáticos
Estruturas de Dados Avançadas
24 MAIO 2024

Resumo. Este documento descreve a minha solução de um problema computacional de complexidade média, que inclui a implementação de um grafo em C, aplicando conceitos avançados de teoria dos grafos e programação para resolver o problema. O objetivo foi desenvolver uma solução capaz de calcular o somatório máximo possível de inteiros a partir de uma matriz de dimensões arbitrárias, considerando regras específicas de conexão entre os elementos. Para esta implementação, foi utilizada uma combinação de uma tabela de hash e listas ligadas. Estas estruturas de dados foram escolhidas para garantir eficiência tanto de memória como de desempenho.

Palavras-chave: Programação, Grafos, Tabelas Hash, Listas Ligadas, Linguagem C, DFS, Algoritmo de Dijkstra

1 Introdução

Este documento descreve o trabalho realizado para o segundo trabalho prático da unidade curricular de Estruturas de Dados Avançadas, parte integrante da Licenciatura em Engenharia de Sistemas Informáticos no Instituto Politécnico do Cávado e do Ave. Este segundo trabalho prático foca-se na implementação e manipulação de grafos, aplicando conceitos avançados de teoria dos grafos e programação em C.

As especificações detalhadas do trabalho podem ser encontradas no anexo titulado "Enunciado do Trabalho Prático". O código fonte está disponível no Github, veja https://github.com/Basiiii/Maximal-Graph-Sum-C.

A implementação de estruturas de dados eficientes é fundamental para resolver problemas computacionais complexos, especialmente no contexto da teoria dos grafos. Este documento descreve a solução desenvolvida para um problema que requer a construção e manipulação de um grafo utilizando a linguagem C. A estrutura do grafo foi construída utilizando uma combinação avançada de uma tabela de hash para os vértices com uma lista ligada para controlar colisões e listas ligadas para as arestas adjacentes. Esta abordagem foi escolhida não apenas para otimizar o desempenho, mas também para garantir eficiência na utilização da memória.

Na estrutura do grafo está a tabela de hash, onde cada entrada representa um vértice do grafo. Em caso de colisão, isto é, quando múltiplos vértices são hashados para a mesma posição na tabela, uma lista ligada é utilizada para resolver os conflitos. Cada vértice é representado por uma estrutura que contém um identificador único e uma lista ligada de arestas adjacentes. As arestas, por sua vez, são representadas por uma estrutura que armazena o vértice de destino, o peso da aresta e um apontador para a próxima aresta na lista ligada.

Além da estrutura de dados, este trabalho também implementou algoritmos fundamentais para a manipulação do grafo. Foram utilizados algoritmos como Dijkstra, para encontrar o caminho mais curto entre vértices, e um algortimo de busca em profundidade (DFS), para explorar os vértices e arestas do grafo e calcular o caminho com o peso maior. Estes algoritmos são essenciais para resolver o problema dado e de conectividade dentro do grafo.

Mais tarde neste documento, explicarei a estrutura do código implementado (Sec. 3), destacando as principais decisões do projeto e as funcionalidades implementadas. Esta abordagem visa fornecer uma solução robusta e eficiente para o problema proposto, utilizando conceitos avançados de teoria dos grafos e programação em C.

2 Padrões e Práticas de Programação

Para garantir a consistência, legibilidade e manutenção da base de código, este projeto segue padrões de programação estabelecidos. Estes padrões estão alinhados com o Guia de Estilo Google para C++ [1], adaptado para C, promovendo uniformidade em todo o projeto.

2.1 Convenções de Nomeação

A adoção de uma convenção de nomeação clara melhora a clareza do código. As seguintes convenções foram utilizadas ao longo do projeto:

- **Funções**: `FunctionsLikeThis()`
- **Variáveis**: `variablesLikeThis`
- **Constantes**: `CONSTANTS_LIKE_THIS`
- **Enums**: `EnumsLikeThis`
- **Structs**: `StructsLikeThis`

Evita-se o uso de nomes completamente em maiúsculas, exceto para constantes, de modo a distingui-las claramente.

2.2 Regras de Formatação

A formatação consistente é fundamental para a legibilidade:

- **Indentação**: 2 espaços por nível de indentação, evitando tabs.
- **Comprimento da Linha**: Linhas limitadas a um máximo de 80 caracteres.

- **Chaves**: As chaves de abertura são colocadas na mesma linha da declaração de controlo.
- **Comentários**: Comentários descritivos são utilizados para explicar secções complexas do código, seguindo um estilo consistente.

2.3 Práticas de Programação

Seguir as melhores práticas garante a qualidade do código:

- Utilizam-se nomes claros para variáveis e funções, tornando o código autoexplicativo.
- Escreve-se código claro e conciso, evitando complexidade desnecessária.
- Segue-se o Guia de Estilo Google para C++ [1] para formatação e estilo gerais, garantindo a conformidade com os padrões da indústria.

2.4 Controlo de Versões

Utiliza-se Git para o controle de versões, facilitando a gestão eficaz da base de código:

- Cria-se uma nova branch para cada funcionalidade ou correção de erro.
- As branches são criadas a partir da branch principal para desenvolvimento, com nomes descritivos que refletem a funcionalidade ou erro abordado.
- Evita-se fazer commits diretamente na branch principal, garantindo a estabilidade.
- As branches de funcionalidade são regularmente fundidas de volta na branch principal após testes completos.

2.5 Documentação

Utiliza-se Doxygen para documentar o código, gerando automaticamente a documentação da API:

- Segue-se a sintaxe do Doxygen para comentários, documentando funções, variáveis e blocos de código.
- A documentação inclui descrições breves, descrições de parâmetros, descrições de valores de retorno e exemplos de uso.
- Utilizam-se tags do Doxygen como `@brief`, `@param`, `@return` e `@example` para estruturar corretamente os comentários.
- Define-se o autor nos comentários de cabeçalho de ficheiros para novos ficheiros ou secções significativas de código, com modificações documentadas usando a tag `@section Modifications`.

Seguindo estes padrões e práticas de programação, o projeto mantém uma base de código de alta qualidade, garantindo consistência e facilitando a compreensão, contribuição e manutenção.

3 Estrutura do Grafo

Nesta secção, vou descrever detalhadamente a estrutura do grafo implementado, abordando as escolhas de design e justificando as decisões com base na eficiência e escalabilidade. A implementação foi feita em C, com o uso de uma tabela de hash para gerenciar os vértices e listas ligadas para as arestas, otimizando tanto o desempenho quanto a utilização de memória.

3.1 Grafo

Antes de discutir em profundidade os vértices e arestas, é importante entender a estrutura básica do grafo implementado. A estrutura *Graph* é definida como segue:

```
1        typedef struct Graph {
2                unsigned int numVertices;
3                unsigned int hashSize;
4                Vertex** vertices;
5        } Graph;
```

Amostra de Código 1.1. Estrutura Grafo

Esta estrutura guarda o número atual de vértices no grafo *(numVertices)*, o tamanho da tabela de hash *(hashSize)*, e um apontador para a tabela de hash que contém os vértices *(vertices)*. A tabela de hash facilita o acesso rápido aos vértices, enquanto a estrutura modular permite que o grafo se ajuste dinamicamente ao crescimento do número de vértices.

3.2 Vértices

Os vértices do grafo são guardados numa tabela de hash. A escolha de uma tabela de hash para os vértices foi feita com o objetivo de minimizar o tempo de acesso e procura. A tabela de hash proporciona um tempo médio de acesso $O(1)$, permitindo inserções, remoções e procuras rápidas, o que é crucial para o desempenho do grafo.

Quando vários vértices são atribuídos ao mesmo índice numa tabela hash, isto representa uma colisão. Para gerir estas situações, foi utilizado um método conhecido como encadeamento separado, onde cada índice da tabela hash aponta para uma lista ligada contendo os vértices que têm o mesmo valor da função hash. Isto garante que o acesso aos dados permaneça eficiente mesmo na presença de colisões, mantendo a estrutura geral da tabela hash intacta e permitindo inserções, remoções e buscas rápidas dentro das listas ligadas associadas a cada índice.

```
1        typedef struct Vertex {
2                unsigned int id;
3                Edge* edges;
4                struct Vertex* next;
```

```
5       } Vertex;
```

Amostra de Código 1.2. Estrutura Vértice

3.3 Arestas

As arestas são guardadas em listas ligadas associadas a cada vértice. Esta abordagem foi escolhida por causa da facilidade de inserção e remoção. A utilização de listas ligadas permite operações eficientes de inserção e remoção de arestas. Em grafos dinâmicos, onde as ligações entre vértices podem mudar frequentemente, esta flexibilidade é essencial.

Comparado a uma matriz de adjacências, as listas ligadas utilizam um espaço proporcional ao número de arestas ($O(n)$), tornando-as mais eficientes para grafos esparsos.

As listas ligadas também permitem que cada aresta guarde um peso, o que é fundamental para algoritmos como o de Dijkstra, que necessitam de informações sobre o peso das arestas.

Cada aresta é representada por uma estrutura que inclui o vértice de destino, o peso da aresta e um apontador para a próxima aresta na lista ligada. Esta representação simplifica a implementação de algoritmos de grafos e a manipulação das conexões entre os vértices.

```
1       typedef struct Edge {
2               unsigned int dest;
3               unsigned int weight;
4               struct Edge* next;
5       } Edge;
```

Amostra de Código 1.3. Estrutura Aresta

3.4 Função Hash

A função de hash implementada aplica uma técnica de hash projetada para alcançar uma excelente distribuição estatística. Garante que cada bit de entrada influencie aproximadamente 50% dos bits de saída, visando minimizar colisões ao garantir saídas únicas para entradas distintas. Isto é alcançado por meio de uma aritmética de inteiros eficiente e operações bitwise. O algoritmo incorpora um "número mágico" cuidadosamente selecionado ($0x45d9f3b$), escolhido após testes extensivos para otimização de desempenho.

Este número melhora a qualidade do algoritmo, aproximando-se dos hashes de criptografia AES, embora com potenciais vantagens marginais de velocidade. Embora a normalização do valor de hash para se ajustar ao tamanho da tabela de hash possa introduzir colisões, estas podem ser eficazmente mitigadas com tabelas de hash suficientemente grandes, ou utilizando a técnica de encadeamento separado. O crédito por este algoritmo vai para *Thomas Mueller*, conforme discutido no Stack Overflow [2].

```
1         uint32_t Hash(uint32_t id, uint32_t hashSize) {
2                 // Inicialize position as given id
3                 uint32_t index = id;
4
5                 // Apply the hash function
6                 index = ((index >> 16) ^ index) * 0x45d9f3b;
7                 index = ((index >> 16) ^ index) * 0x45d9f3b;
8                 index = (index >> 16) ^ index;
9
10                // Normalize the hash value to fit in the hash table
11                index %= hashSize;
12
13                return index;
14        }
```

Amostra de Código 1.4. Função Hash

3.5 Notação Big O

As operações principais no grafo são projetadas para serem eficientes, com a seguinte notação Big O:

- **Inserção de Vértice**: $O(1)$ no caso médio, devido ao acesso direto na tabela de hash.
- **Remoção de Vértice**: $O(1)$ no caso médio, utilizando a tabela de hash.
- **Inserção de Aresta**: $O(1)$ para adicionar uma aresta no início da lista ligada de arestas adjacentes.
- **Remoção de Aresta**: $O(1)$ no melhor caso, ou $O(n)$ onde n é o número de arestas adjacentes que são precisos percorrer para remover uma aresta.
- **Busca de Vértice por ID**: $O(1)$ no caso médio, graças ao acesso direto à tabela de hash.
- **Busca de Aresta**: $O(1)$ no melhor caso, ou $O(n)$ onde n é o número de arestas adjacentes que são precisos percorrer para buscar uma aresta.

Esta estrutura e notação Big O são ideais para suportar operações eficientes em grafos de tamanho variável, mantendo a simplicidade e escalabilidade da implementação.

4 Preservação do Grafo

Este projeto inclui funcionalidades para importar, exportar, guardar e carregar grafos, garantindo a flexibilidade e eficiência na manipulação dos dados do grafo.

4.1 Importação de Grafo a partir de Ficheiro CSV

A importação de grafos é realizada a partir de ficheiros CSV com o formato específico, onde cada linha do ficheiro representa um vértice e as suas arestas. O formato é o seguinte:

```
1;958;2
2;29;8
3;849;9;219;6;648;2
4;289;5;27;4;777;4;622;4
5;259;5;597;1
6;269;7
```

Cada linha contém o identificador de um vértice seguido por pares de valores que representam os vértices de destino e os pesos das arestas. Por exemplo:

```
5;259;5;597;1
```

Esta linha indica que o vértice **5** está ligado ao vértice **259** com peso **5** e ao vértice **597** com peso **1**.

Para processar o ficheiro CSV, utilizamos um parser que analisa o ficheiro completo utilizando as funções `strtok_s()` e `atoi()`. Este método é altamente eficiente, permitindo que um grafo aleatório com um milhão de vértices e um número aleatório de arestas seja analisado e carregado em memória em menos de 2 segundos.

4.2 Exportação de Grafo para Ficheiro CSV

A exportação de grafos para ficheiros CSV segue o formato descrito acima, facilitando a leitura e a manipulação dos dados em outras ferramentas ou para futuros carregamentos no sistema.

4.3 Guardar e Carregar um Grafo

Além da importação e exportação em formato CSV, o projeto também suporta guardar e carregar grafos em e de ficheiros binários. Este método é utilizado para preservar o estado do grafo de forma eficiente e rápida, permitindo um carregamento mais rápido em comparação com a leitura de ficheiros CSV.

A gravação de um grafo é realizado escrevendo os dados diretamente para um ficheiro binário. Esta abordagem garante que todos os detalhes do grafo, incluindo os vértices e as arestas com os seus respectivos pesos, sejam gravados de forma compacta.

O carregamento do grafo a partir de um ficheiro binário envolve a leitura dos dados diretamente para a memória. Esta operação é otimizada para garantir que grandes grafos possam ser carregados rapidamente, preservando toda a estrutura e os dados associados.

Estas funcionalidades de importação, exportação, salvamento e carregamento proporcionam uma gestão flexível e eficiente dos grafos, permitindo uma fácil manipulação e preservação dos dados.

5 Algoritmos

A secção de algoritmos apresenta uma série de implementações para procura e análise de caminhos em grafos. O objetivo destes algoritmos é encontrar todos os caminhos possíveis entre vértices específicos, bem como calcular informações relevantes sobre esses caminhos, como o peso total das arestas. Além disso, inclui algoritmos para calcular o caminho mais curto *(minimum shortest path)* e o caminho mais longo *(maximum longest path)* entre os vértices.

5.1 Encontrar Todos os Caminhos Possíveis

Este algoritmo, implementado no ficheiro `search.c`, utiliza uma abordagem de procura em profundidade (DFS) para encontrar todos os caminhos possíveis de um vértice de origem para um vértice de destino num grafo. Durante a execução, mantém uma lista ligada de caminhos encontrados e calcula o peso total de cada caminho.

```
1         typedef struct PathNode {
2                 unsigned int* vertices;  // Array de vértices no
                      caminho
3                 unsigned int* weights;   // Array de pesos das
                      arestas no caminho
4                 unsigned int length;     // Comprimento do caminho
5                 struct PathNode* next;   // Próximo nó da lista
                      ligada de caminhos
6         } PathNode;
```

Amostra de Código 1.5. Definição da Estrutura de Dados para Caminhos

A função `FindAllPaths` é responsável por coordenar a busca, inicializando os parâmetros necessários e chamando `DepthFirstSearch` para iniciar a pesquisa em profundidade a partir do vértice de origem até o vértice de destino. Cada caminho encontrado é adicionado à lista de caminhos usando a função `AddPath`. Após a execução, a função `PrintPaths` é utilizada para imprimir todos os caminhos encontrados, juntamente com o peso total de cada caminho.

```
1         PathNode* FindAllPaths(const Graph* graph, unsigned int src,
              unsigned int dest, unsigned int* numPaths) {
2                 *numPaths = 0;
3                 unsigned int pathCapacity = 10;
4                 PathNode* paths = NULL;
5                 bool* visited = (bool*)calloc(graph->hashSize,
                      sizeof(bool));
6                 unsigned int* pathVertices =
7                 (unsigned int*)malloc(graph->hashSize *
                      sizeof(unsigned int));
8                 unsigned int* pathWeights =
9                 (unsigned int*)malloc(graph->hashSize *
                      sizeof(unsigned int));
```

```
10
11              DFSContext context = { .graph = graph,
12                      .pathVertices = pathVertices,
13                      .pathWeights = pathWeights,
14                      .visited = visited,
15                      .pathIndex = 0,
16                      .paths = &paths,
17                      .numPaths = numPaths,
18                      .pathCapacity = &pathCapacity };
19
20              bool success = DepthFirstSearch(&context, src, dest);
21
22              if (!success) {
23                      FreePaths(paths);
24                      paths = NULL;
25                      *numPaths = 0;
26              }
27
28              free(visited);
29              free(pathVertices);
30              free(pathWeights);
31
32              return paths;
33      }
```

Amostra de Código 1.6. Função para Encontrar Todos os Caminhos

O algoritmo utiliza uma lista ligada de caminhos (`PathNode`) para guardar cada caminho encontrado. A estrutura de dados `DFSContext` é utilizada para manter o estado da busca em profundidade, incluindo vértices visitados, índice do caminho atual e a lista de caminhos encontrados.

5.2 Encontrar o Caminho Mais Curto com Dijkstra

Este algoritmo, implementado no ficheiro `dijkstra-min.c`, utiliza o algoritmo de Dijkstra para encontrar o caminho mais curto de um vértice de origem para um vértice de destino num grafo ponderado. Durante a execução, mantém uma estrutura de heap mínimo para gerenciar os vértices a serem explorados e calcula o peso total do caminho mais curto.

```
1       typedef struct HeapNode {
2               unsigned int vertex;
3               unsigned int weight;
4       } HeapNode;
5
6       typedef struct MinHeap {
7               HeapNode* nodes;
8               unsigned int size;
9               unsigned int capacity;
10      } MinHeap;
```

Amostra de Código 1.7. Definição da Estrutura de Dados para Heap

A função `DijkstraShortestPath` é responsável por coordenar a execução do algoritmo, inicializando as estruturas necessárias e chamando funções auxiliares para inserir vértices no heap e extrair o vértice com o menor peso. Após a execução, a função `PrintShortestPath` é utilizada para imprimir o caminho mais curto encontrado, juntamente com o peso total do caminho.

O algoritmo utiliza uma estrutura de heap mínimo (`MinHeap`) para gerenciar os vértices a serem explorados. A estrutura `HeapNode` é utilizada para representar um nó no heap, com o índice do vértice (`vertex`) e o peso associado (`weight`). A estrutura `MinHeap` mantém um array de elementos `HeapNode` e rastreia o número atual de elementos no heap (`size`) e a capacidade máxima do heap (`capacity`).

A função `DijkstraShortestPath` utiliza Dijkstra para calcular o caminho mais curto de um vértice de origem (`src`) para um vértice de destino (`dest`) num grafo ponderado (`graph`). Após a execução, o caminho mais curto é reconstruído utilizando o array `prev` e armazenado em `path`, com seu comprimento armazenado em `pathLength`.

Explicação:

- **Estruturas de Dados:** `HeapNode` representa um nó no heap com um vértice e um peso, enquanto `MinHeap` é uma estrutura de heap mínimo para Dijkstra.
- **Funções Auxiliares:** `CreateMinHeap`, `InsertNode`, `ExtractMin` são funções auxiliares para manipulação do heap.
- **Algoritmo de Dijkstra:** A função `DijkstraShortestPath` implementa o algoritmo de Dijkstra para calcular o caminho mais curto num grafo ponderado, utilizando um heap mínimo para otimização.
- **Libertação de Memória:** A memória alocada dinamicamente (`dist`, `visited`, `prev`, `minHeap->nodes`, `minHeap`) é libertada no fim da função para evitar *memory leaks*.

Esta versão da função `DijkstraShortestPath` fornece um tratamento robusto de erros e documentação clara, garantindo que falhas na alocação de memória sejam adequadamente tratadas e que a função seja confiável em diversos cenários.

5.3 Encontrar o Caminho Mais Longo com Dijkstra

Este algoritmo, implementado no ficheiro `dijkstra-max.c`, utiliza o algoritmo de Dijkstra para encontrar o caminho mais longo de um vértice de origem para um vértice de destino num grafo. Durante a execução, mantém uma estrutura de heap máximo para gerir os vértices a serem explorados e calcula o peso total do caminho mais longo.

```
1         typedef struct HeapNode {
2                 unsigned int vertex;
3                 unsigned int weight;
```

```
4       } HeapNode;
5
6       typedef struct MaxHeap {
7               HeapNode* nodes;
8               unsigned int size;
9               unsigned int capacity;
10      } MaxHeap;
```

Amostra de Código 1.8. Definição da Estrutura de Dados para Heap Máximo

A função `DijkstraMaxPath` é responsável por coordenar a execução do algoritmo, inicializando as estruturas necessárias e chamando funções auxiliares para inserir vértices no heap e extrair o vértice com o maior peso. Após a execução, a função `PrintLongestPath` é utilizada para imprimir o caminho mais longo encontrado, juntamente com o peso total do caminho.

O algoritmo utiliza uma estrutura de heap máximo (`MaxHeap`) para gerir os vértices a serem explorados. A estrutura `HeapNode` é utilizada para representar um nó no heap, com o índice do vértice (`vertex`) e o peso associado (`weight`). A estrutura `MaxHeap` mantém um array de elementos `HeapNode` e rastreia o número atual de elementos no heap (`size`) e a capacidade máxima do heap (`capacity`).

Explicação:

- **Estruturas de Dados:** `HeapNode` representa um nó no heap com um vértice e um peso, enquanto `MaxHeap` é uma estrutura de heap máximo para Dijkstra.
- **Funções Auxiliares:** `CreateMaxHeap`, `InsertNodeMaxHeap`, `ExtractMax` são funções auxiliares para manipulação do heap máximo.
- **Algoritmo de Dijkstra:** A função `DijkstraMaxPath` implementa o algoritmo de Dijkstra para calcular o caminho mais longo num grafo, utilizando um heap máximo para otimização.
- **Libertação de Memória:** A memória alocada dinamicamente (`dist`, `visited`, `prev`, `maxHeap->nodes`, `maxHeap`) é libertada no final da função para evitar *memory leaks*.

Esta versão da função `DijkstraMaxPath` fornece um tratamento robusto de erros e documentação clara, garantindo que falhas na alocação de memória sejam adequadamente tratadas e que a função seja confiável.

6 Conclusão

Neste trabalho, desenvolvi e implementei com sucesso uma solução para um problema computacional desafiador. Esta solução envolve a criação de um grafo em C, utilizando conceitos avançados de teoria dos grafos e programação para alcançar o resultado desejado. O objetivo principal deste trabalho prático foi desenvolver um método capaz de calcular a soma máxima possível de inteiros a partir de uma matriz de dimensões arbitrárias, respeitando regras específicas que regem as conexões entre os elementos.

Para isso, utilizei uma combinação de tabelas de hash e listas ligadas. Estas estruturas de dados foram escolhidas por causa da sua capacidade de garantir eficiência tanto no uso de memória quanto no desempenho. A tabela de hash facilitou operações rápidas de procura, enquanto as listas ligadas permitiram a travessia eficiente e a manipulação das conexões entre os elementos no grafo.

Através desta implementação, demonstrei o meu conhecimento das teorias subjacentes e das metodologias necessárias para resolver problemas computacionais complexos. A abordagem adotada não apenas alcança a funcionalidade desejada, mas também otimiza o uso de recursos, tornando-a adequada para aplicações do mundo real onde a eficiência é fundamental.

Em resumo, este projeto proporcionou *insights* valiosos sobre a aplicação prática da teoria dos grafos e das estruturas de dados, melhorando ainda mais a minha proficiência em técnicas avançadas de programação. A solução desenvolvida aqui é um testemunho da minha capacidade de enfrentar e resolver desafios computacionais exigentes de forma eficaz.

Agradecimentos Gostaria de expressar o meu sincero agradecimento ao Professor Luís Ferreira, pela sua orientação, apoio e *feedback* valiosos ao longo do desenvolvimento deste projeto. A sua ajuda foi fundamental no desenvolvimento desta solução informática.

References

1. Google Inc.: Google C++ Style Guide. `https://google.github.io/styleguide/cppguide.html` (2024), visto: 2024-05-02
2. Mueller, T.: What integer hash function are good that accepts an integer hash key? (2009), URL `https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key`, visto: 2024-05-14

Todas as ligações foram seguidas pela última vez no dia 24 de maio de 2024.