Instituto Politécnico do Cávado e do Ave

Licenciatura em Engenharia de Sistemas Informáticos
Estruturas de Dados Avançadas
2023/2024

Fase 1 e 2

Aluno: Pedro Faria (a23290)
Professor: Luís Ferreira

Conteúdo

1 Introdução **1**
1.1 Enquadramento 1
1.2 Problema a resolver 1
1.2.1 Fase 1 - Listas ligadas 1
1.2.2 Fase 2 - Grafos 2

2 Desenvolvimento / Trabalho realizado **3**
2.1 Fase 1 - Listas Ligadas 3
2.1.1 Definição das estruturas 3
2.1.2 Leitura de ficheiro 3
2.1.3 Alteração de Valores 4
2.1.4 Adição e Remoção de linhas e/ou colunas 5
2.1.5 Apresentação da lista ED 5
2.2 Fase 2 - Grafos 7
2.2.1 Definição das estruturas 7
2.2.2 Criação de grafos, vértices e adjacências 8
2.2.3 Inserir vértices e adjacências ao grafo 9
2.2.4 Contagem de caminhos 9

3 Conclusão **10**

4 Trabalhos Futuros **10**

Referências e bibliografia **11**

Lista de Figuras

1 Estruturas definidas para a primeira fase do projeto. 3
2 Funções *PlaceIntLine* e *PlaceLineED* 4
3 Funções para a criação de linhas 5
4 Definição das estruturas para a fase 2 do projeto 7
5 Criação de grafo. 8
6 Funções para a contagem de caminhos 9

1 Introdução

1.1 Enquadramento

O seguinte documento serve para documentar o trabalho desenvolvido para realizar o projeto proposto na unidade curricular de Estruturas de Dados Avançadas.

Este documento descreverá ambas as fases do trabalho realizado, aplicando diversos tópicos e metodologias para resolver e implementar as soluções necessárias.

1.2 Problema a resolver

1.2.1 Fase 1 - Listas ligadas

Pretendeu-se realizar uma solução para o cálculo da soma máxima possível dos inteiros de uma matriz de inteiros com qualquer dimensão, de modo que nenhum dos inteiros selecionados compartilhe a mesma linha ou coluna.

Procura-se implementar as funcionalidades seguintes:

1. Definição de uma estrutura de dados dinâmica ED, recorrendo à definição de listas ligadas, para a representação de uma matriz de valores inteiros;
2. Carregamento para a estrutura de dados da alínea anterior ED dos dados de uma matriz constante num ficheiro de texto. A operação deverá considerar matrizes com qualquer dimensão, sendo os valores separados por vírgulas.
3. Alteração dos inteiros constantes em ED;
4. Inserção de novas linhas/colunas na matriz representada por ED;
5. Remoção de linhas/colunas na matriz representada por ED;
6. Listagem de forma tabular na consola de todos os inteiros constantes na estrutura de dados ED;
7. Cálculo da soma máxima possível dos inteiros em ED de modo que nenhum dos inteiros selecionados compartilhe a mesma linha ou coluna.

1.2.2 Fase 2 - Grafos

1. Definir uma estrutura de dados GR para representar um grafo. Esta estrutura deve ser capaz de representar grafos dirigidos e deve suportar um número variável de vértices. A implementação deve incluir funções básicas para criação do grafo, adição e remoção de vértices e arestas;
2. Após definir a estrutura de dados GR, pretende-se modelar o problema utilizando grafos. Cada elemento da matriz de inteiros será representado por um vértice no grafo. As arestas entre vértices devem representar a possibilidade de somar dois elementos adjacentes na matriz, sob uma regra de conexão específica que poderá ser configurada pelo utilizador (por exemplo, apenas elementos na mesma linha ou coluna, não permitindo diagonais, ou qualquer outra regra);
3. Carregamento para a estrutura de dados da alínea anterior GR dos dados de uma matriz de inteiros constante num ficheiro de texto. A operação deverá considerar matrizes de inteiros com qualquer dimensão, sendo os valores separados por vírgulas.
4. Implementar operações de manipulação de grafos, incluindo procura em profundidade ou em largura, para identificar todos os caminhos possíveis que atendem às regras de conexão definidas. Desenvolver também uma função para calcular a soma dos valores dos vértices num dado caminho;
5. Utilizar as estruturas e algoritmos desenvolvidos para encontrar o caminho que proporciona a maior soma possível dos inteiros na estrutura GR, seguindo a regra de conexão estabelecida. O programa deve fornecer tanto a soma máxima quanto o caminho (ou caminhos, se existirem múltiplos caminhos com a mesma soma máxima) que resulta nessa soma;

Adicionalmente foi criado um repositório no GitHub, para auxiliar na organização e desenvolvimento do projeto. Além do ficheiro zip enviado na submissão do trabalho, todo o código realizado está disponível no seguinte link: https://github.com/PedroF17/EDA

2 Desenvolvimento / Trabalho realizado

2.1 Fase 1 - Listas Ligadas

2.1.1 Definição das estruturas

Nesta primeira etapa do projeto, foram criadas as estruturas que são essenciais para o desenvolvimento deste projeto, pois serão utilizadas para as funções desenvolvidas nos próximos passos.

Para este trabalho, foram criadas duas estruturas *struct* que alojam os dados da matriz. A *struct* Line inclui um inteiro que irá possuir um dos números da matriz, e um apontador da própria *struct*, que aponta para o número seguinte na lista.

A segunda *struct* ED tem um apontador para o ínicio da linha da matriz e um apontador da própria ED para possibilitar a navegação para a próxima linha, como mostra a Figura 1 abaixo.

```
13  typedef struct Line{
14      int num;
15      struct Line* next;
16  }Line;
17
18  typedef struct ED{
19      Line * startLine;
20      struct ED * next;
21  }ED;
```

Figura 1: Estruturas definidas para a primeira fase do projeto.

2.1.2 Leitura de ficheiro

Para possibilitar a leitura da ED através de um ficheiro, foi criada uma função *ReadFile* para receber, processar e armazenar a matriz de forma externa.

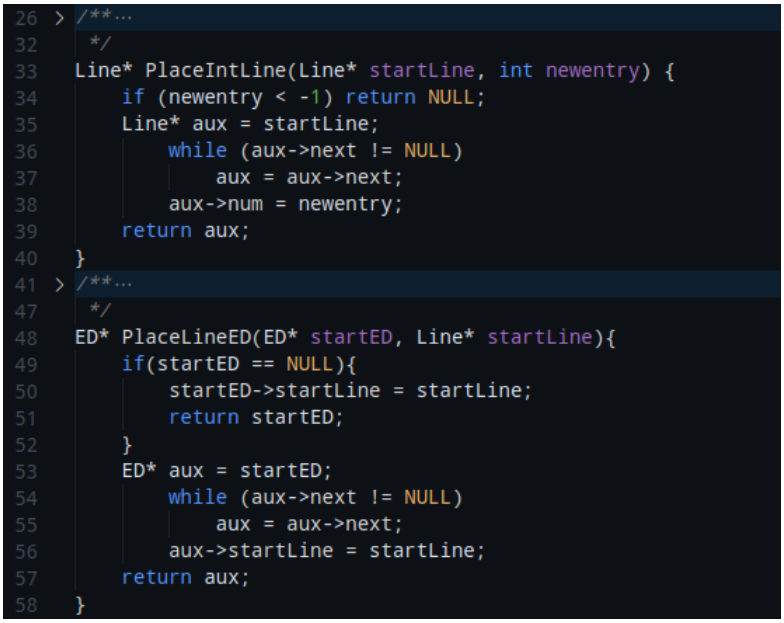

Figura 2: Funções *PlaceIntLine* e *PlaceLineED*

Esta função utiliza uma função auxiliar chamada *splitString*, com o objetivo de separar os valores da matriz dos carácters que não são importantes para o armazenamento na ED. A função *splitString* está incluida no ficheiro *utils.c*, e divide uma string em *tokens*, que são os valores da matriz separados do seu *delimitador* (;).

Além disso, a função *ReadFile* utiliza funções tais como *PlaceIntLine* e *PlaceLineED* após a leitura do ficheiro. Estas funções servem para introduzir os valores lidos dentro da *struct* definida no programa.

Enquanto que a função *PlaceIntLine* procura inserir um inteiro dentro de uma linha, como mostra a Figura 2 entre as linhas 33 e 40, a função *PlaceLineED* coloca uma linha da matriz dentro da lista ligada criada ED, como pode ser visto na Figura 2 a partir da linha 48.

2.1.3 Alteração de Valores

Para possibilitar a alteração de dados dentro da matriz, foi criada uma função *ReplaceNumberInLine*, que, dado dois inteiros, um valor novo e o número a ser substituído, percorrará toda a linha até encontrar o número pedido. Além disso, a função *ReplaceNumberInED* utilizaria a função *Re-*

placeNumberInLine para, além de percorrer uma linha, percorreria todas as linhas até o final da lista ED. Esta função foi desenvolvida.

2.1.4 Adição e Remoção de linhas e/ou colunas

Para a adição de uma linha na ED foram criadas as seguintes funções que mostra a Figura 3.

```
19  Line * NewLine(int num){
20      Line* aux = (Line*)malloc(size: sizeof(Line));
21      if (aux==NULL) return NULL;
22      aux->next = NULL;
23      aux->num = num;
24      return aux;
25  }
26  /**...
32   */
33  Line* PlaceIntLine(Line* startLine, int newentry) {···
40  }
41  /**...
47   */
48  ED* PlaceLineED(ED* startED, Line* startLine){
49      if(startED == NULL){
50          startED->startLine = startLine;
51          return startED;
52      }
53      ED* aux = startED;
54          while (aux->next != NULL)
55              aux = aux->next;
56          aux->startLine = startLine;
57      return aux;
58  }
```

Figura 3: Funções para a criação de linhas

A função *PlaceLineED* é também utilizada para adicionar novas linhas na lista, como mostra a Figura 3 a partir da linha 48. Neste caso, a função *NewLine* é a responsável para alocar o espaço necessário para a nova linha ser colocada na lista ED. (Figura 3, linhas 19 a 25).

2.1.5 Apresentação da lista ED

Para apresentar a lista ED na linha de comandos ao executar o programa, decidiu-se criar duas funções:

- A função *ShowLine* serve para apresentar uma linha, recursivamente, navegando pela *struct Line* e realizando *printf* a cada valor da linha;

- A função *ShowED* utiliza a função *ShowLine* referida anteriormente, para apresentar os números de cada linha até ao final da matriz.

2.2 Fase 2 - Grafos

2.2.1 Definição das estruturas

Nesta segunda fase do projeto, as estruturas foram criadas de forma semelhante, embora sejam de um carácter mais complexo.
Para esta fase, ao invés de serem criadas duas *structs*, foram criadas três:

- A *struct* Vertice contém variáveis para o valor inteiro, o nome do vértice e apontadores para as structs Vertice e Graph;

- A *struct* Adj possui variáveis relevantes para as adjacências ao vértices. Esta contem um inteiro com um indentificador, um float com o peso da adjacência e um apontador para a próxima adjacência;

- A *struct* Graph tem apontadores para os vértices e para as adjacências. Além disso, possui também dois inteiros, um com um número de vértices, e um para o total de vértices do grafo.

```
19  #pragma region Structs
20
21  typedef struct Adj{
22     int code;
23     float weight;
24     struct Adj* next;
25  }Adj;
26
27  typedef struct Vertice{
28     int code;
29     bool visited;
30     char* name;
31     struct Adj* adjacent;
32     struct Vertice* nextVert;
33  }Vertice;
34
35  typedef struct Graph{
36     Vertice* vertices;
37     Adj* adjs;
38     int numVert;
39     int totalVert;
40  }Graph;
```

Figura 4: Definição das estruturas para a fase 2 do projeto

2.2.2 Criação de grafos, vértices e adjacências

Para a criação de grafos, será necessário criar vértices e adjacências. Para isso, foram criadas diversas funções para concretizar este problema.

```
20  Graph* newGraph(int total){
21      Graph* g = (Graph*)malloc(size: sizeof(Graph));
22      if (g != NULL) {
23          g->vertices = NULL;
24          g->numVert = 0;
25          g->totalVert = total;
26      }
27      return g;
28  }
```

Figura 5: Criação de grafo.

A figura 4 representa a função responsável pela criação do grafo. Esta função utiliza o *malloc* para alocar memória suficiente para um grafo, e define os valores da *struct* como NULL, 0 e o valor total de vértices que o grafo possui.

As funções de criação de vértices e adjacências, *newVertice* e *newAdj* respetivamente, funcionam de semelhante maneira, mas com os dados relevantes para cada *struct*.

2.2.3 Inserir vértices e adjacências ao grafo

Para inserir vértices, foi criada uma função que primeiramente verifica se o vértice existe ou se já está inserido no grafo. Após as validações necessárias, a lista de vértices é percorrida até encontrar o último da lista, onde o novo vértice será introduzido. Esta mesma lógica é aplicada na adição de adjacências, mas no contexto dos dados necessários dessa lista.

2.2.4 Contagem de caminhos

As funções *countPaths* e *countPathsName* foram criadas para realizar a contagem de caminhos entre dois vértices. A diferença entre estas funções é o modo de procura de vértices, onde uma utiliza o número inteiro do vértice, onde a outra procura utilizando o nome associado ao vértice, como demonstra a figura abaixo.

```
124  int countPaths(Graph *g, int start, int end, int count){
125      if (g == NULL) return 0;
126
127      if (start == end) return (++count);
128      else {
129          Graph* aux = checkVertice(g, code: start);
130          Adj* a = aux->adjs;
131          while (a) {
132              Graph* g = checkVertice(g, code: a->code);
133              count = countPaths(g, start: g->vertices->code, end, count);
134              a = a->next;
135          }
136      }
137      return count;
138  }
139
140 > /**...
148   */
149  int countPathsName(Graph* g, char* start, char* end, int count){
150      int s = checkVerticeName(g, name: start);
151      int e = checkVerticeName(g, name: end);
152      return countPaths(g, start: s, end: e, count: 0);
153  }
```

Figura 6: Funções para a contagem de caminhos

3 Conclusão

A realização deste projeto permitiu entender como é que certas funções funcionam e também a importância da alocação de memória para a execução dessas funções.

Além disso, defrontei-me com diversos desafios na realização deste projeto, nomeadamente a utilização dos comandos de alocação de memória, a recursividade de funções para percorrer as listas e a implementação de algumas funções dificultaram a execução deste trabalho.

4 Trabalhos Futuros

Mesmo após a entrega deste trabalho, ainda existem várias funções e funcionalidades que ou ficaram por se realizar, ou estão de momento num estado semi-funcional.

A última parte da fase 1 e alguns pontos da fase 2 não foram realizados de forma completa devido a erros ou problemas de alocação de memória.

Estes problemas referidos serão possívelmente resolvidos num futuro próximo, com o objetivo de alcançar um estado de alguma maneira completa ao projeto, aprimorando o conhecimento desenvolvido ao longo da unidade curricular.

Referências e bibliografia

https://www.doxygen.nl/
https://overleaf.com/
https://tex.stackexchange.com/
https://cplusplus.com/reference/clibrary/
https://vscodium.com/
https://stackoverflow.com/