Trabalho Prático
Fase 2

Licenciatura de Engenharia de Sistemas Informáticos

Estrutura de Dados Avançadas

Aluno: Diogo Araújo Machado - 26042

Professor: Luis Ferreira

2023/2024

Índice

Introdução

Este documento descreve o trabalho realizado para a segunda fase do trabalho prático da unidade curricular de Estrutura de Dados Avançadas, e pretende aplicar conceitos avançados de teoria de grafos e programação em C para resolver um problema computacional com grau de complexidade elevado, relacionando estruturas de dados, algoritmos de procura e técnicas de otimização.

Trabalho desenvolvido

Estruturas de dados

Para a representação do grafo eu utilizei uma lista ligada constituída por três estruturas diferente: vértices, adjacências e para o grafo. O tipo de grafo utilizado foi um grafo orientado pesado.

```
#pragma region ESTRUTURA_GRAFO
typedef struct Adjacencia {
    int codVert;                //Codigo do vertice adjacente
    int peso;                   //Peso da adjacencia
    bool visitado;
    struct Adjacencia* next;    //Apontador para outra adjacencia
}Adjacencia;

typedef struct Vertice {
    int cod;                    //Codigo do vertice
    int numAdj;                 //Numero de adjacencias do vertice
    bool visitado;
    Adjacencia* adjacencias;    //Lista de adjacencias do vertice
    struct Vertice* next;       //Apontador para o vertice seguinte
}Vertice;

typedef struct Grafo {
    Vertice* inicio;            //Inicio da lista dos vertices do grafo
    int numVert;                //Numero de vertices do grafo
}Grafo;
#pragma endregion
```

Figura 1: Estruturas de dados utilizadas

A estrutura dos vértices contem duas variáveis do tipo inteiro, uma para o código do vértice e uma para o número de adjacências do respetivo vértice, uma variável booleana para marcar o vértice como visitado (variável essa que será utilizada nos algoritmos de procura), e por fim dois apontadores, um para uma lista de adjacências e outro para o vértice seguinte. De seguida temos a estrutura das adjacências que contem também duas variáveis do tipo inteiro, uma para o código do vértice adjacente e outra para o peso da adjacência, uma variável booleana para marcar a adjacência como visitada e um apontador para a próxima adjacência. A estrutura do grafo é uma estrutura que apenas tem um apontador para a lista dos vértices e uma variável do tipo inteiro com o número de vértices do grafo.

Leitura de ficheiro

Para obter os dados para o grafo eu utilizei a leitura através de um ficheiro .csv com uma lista de adjacências e fiz um algoritmo de leitura que lê caracter a caracter identificando o separador de modo a dividir os pesos das diferentes adjacências.

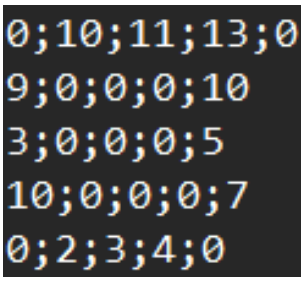

Figura 2: Ficheiro csv com a lista de adjacências.

Como se pode observar na figura 2 temos uma lista de adjacências em que cada linha e coluna representa um vértice.

```
while (1) {
    ch = fgetc(fp);
    if (ch != ';' && ch != '\n' && ch != EOF) {
        strncat(valorFich, &ch, sizeof(char));
        continue;
    }
    peso = atoi(valorFich);
    if (peso != 0) {
        valorFich[0] = '\0';
        novaAdj = CriaAdjacencia(codAdj, peso);
        inicioAdj = InsereAdjacencia(inicioAdj, novaAdj);
        numAdj++;
    }
    codAdj++;
    if (ch == '\n' || ch == EOF) {
        novoVert->numAdj = numAdj;
        novoVert = InsereAdjacenciaVertice(novoVert, inicioAdj);
        inicioVert = InsereVertice(inicioVert, novoVert);
        inicioAdj = NULL;
        novoVert = NULL;
        codAdj = 0;
        numAdj = 0;
        vert++;
        if (ch == EOF) {
            break;
        }
        novoVert = CriaVertice(vert);
        continue;
    }
}
```

Figura 3: Ciclo do algoritmo de leitura do ficheiro csv

Como se pode observar na figura 3 eu crio um ciclo while que apenas para quando a leitura do caracter encontra EOF, variável do sistema que representa o fim do ficheiro. No início do ciclo a função lê um caracter e identifica se não é um '\n', um ';' ou a variável do sistema EOF. Vai colocando o valor lido num buffer que depois quando encontra algum dos caracteres referidos através da função *atoi* é convertido para um número inteiro e adicionado na variável do tipo inteiro peso. Se o peso for 0 não cria adjacência e apenas incrementa o código do vértice da adjacência, caso contrário é criada a adjacência, e inserida na lista de adjacências. Quando o fgetc encontra um '\n' ou o EOF é inserida a lista de adjacência no vértice correspondente, as variáveis são inicializadas a 0. Caso seja o fim do ficheiro o ciclo while termina e acaba assim a leitura do ficheiro, caso contrário cria um vértice em memória e continua a leitura.

Funções de manipulação do grafo

Ao longo do trabalho foram criadas diversas funções de manipulação do grafo, tais como a adição e remoção de vértices e adjacências, a de procurar vértice e as de verificar a existência de vértices e adjacências.

Vou usar como exemplo a função de remover vértice que recebe os seguintes parâmetros:

```
Grafo* RemoveVertice(Grafo* grafo, int codVert)
```

Figura 4: Parâmetros da função de remover vértice

A função recebe como parâmetros o grafo e o código do vértice que se pretende remover e de seguida vai verificar se existe o vértice, usando a função que verifica se existe o vértice, função simples que apenas percorre a lista dos vértices todos e compara o código dos vértices com o código do vértice inserido por parâmetro para verificar se existe. Em caso de existir continua a função e testa diferentes condições. Caso o vértice que se pretende eliminar seja o primeiro, criamos um apontador auxiliar para o vértice seguinte e limpa a lista de adjacências do vértice que será eliminado e de seguida elimina o vértice. Após isso ainda elimina o vértice de todas as listas de adjacências dos restantes vértices. Tudo isso pode ser visto figura seguinte:

```
Vertice* auxVert = grafo->inicio;
if (codVert == 0) {
    Vertice* proxVert = auxVert->next;
    if (auxVert->adjacencias != NULL) {
        LimpaListaAdjacencias(auxVert);
    }
    free(auxVert);
    grafo->inicio = proxVert;
    grafo->numVert--;
    Vertice* vertPointer = grafo->inicio;
    while(vertPointer != NULL){
        grafo = RemoveAdjacencia(grafo, vertPointer->cod, codVert);
        vertPointer = vertPointer->next;
    }
    return grafo;
}
```

Figura 5: Caso o vértice que se pretende eliminar seja o primeiro do grafo

Caso o vértice que se pretenda eliminar seja o único vértice do grafo a função limpa o vértice e a sua lista de adjacências e retorna o grafo vazio.

```
if (auxVert->next == NULL) {
    if (auxVert->adjacencias != NULL) {
        LimpaListaAdjacencias(auxVert);
    }
    free(auxVert);
    grafo->numVert--;
    return grafo;
}
```

Figura 6: Caso o vértice seja o único do grafo

E por fim temos os restantes casos em que se procura o vértice correspondente e quando se encontra elimina a lista de adjacências desse vértice e o vértice em todas as listas de adjacências em todos os outros vértices.

```
while (auxVert->cod != codVert) {
    auxVert = auxVert->next;
    antVert = antVert->next;
}

if (auxVert->adjacencias != NULL) {
    LimpaListaAdjacencias(auxVert);
}

antVert->next = auxVert->next;
free(auxVert);
grafo->numVert--;
Vertice* vertPointer = grafo->inicio;
while (vertPointer != NULL) {
    grafo = RemoveAdjacencia(grafo, vertPointer->cod, codVert);
    vertPointer = vertPointer->next;
}
return grafo;
```

Figura 7: Restantes casos

Funções de procura

Como função de procura eu utilizei a procura em profundidade criando o algoritmo de Depth First Traversal. Para isso utilizei uma estrutura auxiliar para a stack, como na figura seguinte:

```
typedef struct Stack {
    int id;
    struct Stack* next;
}Stack;
```

Figura 8: Estrutura da stack

A estrutura apenas tem uma variável do tipo inteiro para o código do vértice e um apontador para o valor seguinte da stack.

```c
Stack* DepthFirstTraversal(Grafo* grafo, int codIni) {
    Vertice* auxVert = ProcuraVerticeCod(grafo->inicio, codIni);
    auxVert->visitado = true;
    Stack* stack = CriaStackValor(auxVert->cod);
    Stack* registo = CriaStackValor(auxVert->cod);
    Stack* novo = NULL;
    Stack* novoReg = NULL;
    registo = InsereNaStack(NULL, registo);
    stack = InsereNaStack(NULL, stack);
    while (!IsStackEmpty(stack)) {
        auxVert = ProcuraVerticeCod(grafo->inicio, StackPeek(stack));
        auxVert->visitado = true;
        Adjacencia* auxAdj = auxVert->adjacencias;
        while(auxAdj != NULL) {
            Vertice* vertAdj = ProcuraVerticeCod(grafo->inicio, auxAdj->codVert);
            if (auxAdj->visitado == false && vertAdj->visitado == false) {
                auxAdj->visitado = true;
                novo = CriaStackValor(vertAdj->cod);
                novoReg = CriaStackValor(vertAdj->cod);
                stack = InsereNaStack(stack, novo);
                registo = InsereNaStack(registo, novoReg);
                break;
            }
            auxAdj = auxAdj->next;
            if (auxAdj == NULL) {
                stack = RemoveValorStack(stack);
            }
        }
    }
    return registo;
}
```

Figura 9: Algoritmo DFT

O algoritmo começa por procurar o vértice por onde se pretende começar e coloca um apontador a apontar para o vértice e guarda o valor em duas stack's, uma de registo e outra que se manipula ao longo do ciclo. Dentro do ciclo temos apontadores para as adjacências do vértice e para o vértice e verificamos qual a primeira adjacência que não tenha sido usada para um vértice que não tenha sido visitado. Quando encontrado coloca-se o valor na stack e o valor da variavel booleana da adjacência e do vértice torna-se true e continua o ciclo com o vértice adjacente. Chegando a um vértice que não tenha nenhum vértice adjacente usamos o backtracking tirando o valor da stack e voltando aos vértices anteriores até chegar a um com adjacências para vértices que ainda não tenham sido visitados. Quando a stack ficar vazia significa que já se visitou todos os vértices do grafo e, portanto, o ciclo termina e a função retorna a stack de registo com a ordem dos vértices visitados.

Para além deste algoritmo criei também um algoritmo que percorre um grafo de um vértice de início até ao vértice destino calculando o peso da deslocação.

```
int CalculoSomaEntreDoisVertices(Grafo* grafo, int codIni, int codDest) {
    Vertice* auxVert = ProcuraVerticeCod(grafo->inicio, codIni);
    auxVert->visitado = true;
    Stack* stack = CriaStackValor(auxVert->cod);
    Stack* novo = NULL;
    stack = InsereNaStack(NULL, stack);
    int soma = 0;
    auxVert = ProcuraVerticeCod(grafo->inicio, StackPeek(stack));
    while (auxVert->cod != codDest) {
        auxVert->visitado = true;
        Adjacencia* auxAdj = auxVert->adjacencias;
        while (auxAdj != NULL) {
            Vertice* vertAdj = ProcuraVerticeCod(grafo->inicio, auxAdj->codVert);
            if (auxAdj->visitado == false && vertAdj->visitado == false) {
                soma += auxAdj->peso;
                auxAdj->visitado = true;
                novo = CriaStackValor(vertAdj->cod);
                stack = InsereNaStack(stack, novo);
                auxVert = ProcuraVerticeCod(grafo->inicio, StackPeek(stack));
                break;
            }
            auxAdj = auxAdj->next;
            if (auxAdj == NULL) {
                int vertFim = StackPeek(stack);
                stack = RemoveValorStack(stack);
                int vertIni = StackPeek(stack);
                Vertice* vertRem = ProcuraVerticeCod(grafo->inicio, vertIni);
                Adjacencia* adjRem = vertRem->adjacencias;
                while (adjRem->codVert != vertFim) {
                    adjRem = adjRem->next;
                }
                soma -= adjRem->peso;
                auxVert = ProcuraVerticeCod(grafo->inicio, StackPeek(stack));
            }
        }
    }
    return soma;
}
```

Figura 10: Cálculo do peso entre dois vértices

Este algoritmo é semelhante ao DFT, mas ao invés do ciclo while terminar quando a stack acaba ele apenas acaba quando chegar ao vértice destino. A stack regista na mesma o caminho percorrido e retira o vértice da stack em caso de backtracking, mas para além disso vai somando os pesos das adjacências e subtraindo no caso de retroceder o vértice. Por fim retorna o valor da soma das adjacências.

Reflexões

Sobre o trabalho, eu acho que podia ter desenvolvido a função DFT para descobrir todos os caminhos e não apenas um e criado a função de calcular a soma máxima das adjacências. Para além disso eu tentei desenvolver a função DFT mas recursiva mas acabei por deixar de lado por não conseguir fazer backtracking aos vertices.

Conclusão

Com este trabalho obtive diversos conhecimentos com as listas ligadas e com os grafos, aprendi uma maneira mais eficiente de fazer leitura de ficheiros csv através da leitura caracter a caracter e evoluí a nível de desenvolvimento de algoritmos. No futuro pretendo melhorar ainda mais o desenvolvimento de algoritmos, dividir as minhas funções em várias funções, tornando o código mais limpo e de fácil leitura, e melhorar os meus conhecimentos de programação no geral.

Anexos

Enunciado do trabalho: