

Universidade Federal de Minas Gerais  
Instituto de Ciências Exatas  
Departamento de Ciências da Computação

**DCC005 - ALGORITMOS E ESTRUTURAS DE  
DADOS III - TURMA TE**

Trabalho Prático 0 – Busca em Profundidade em  
Grafos

Guilherme Saulo Alves  
Professores - Jussara Marques, Olga Nikolaevna,  
Marcos Augusto, Wagner Meira

# 1-Introdução

O proposito deste trabalho consiste em fazer um busca em profundidade iterativa a partir de um vértice inicial até um vértice destino. A consulta retorna o caminho percorrido pela busca listando os vértices à medida em que são descobertos. Para isso, devemos utilizar o TAD de listas, pilhas e Grafos.

## 2-Implementação

Uma das primeiras implementações foi criar estrutura para representar listas (indicam a lista de vértices adjacentes de um vértice), pilhas (para o algoritmo da busca em profundidade iterativa) e grafos. As funções serão explicadas detalhadamente ao longo da documentação.

### 2.1. Considerações Iniciais

O ambiente de desenvolvimento do código foi no CodeBlocks IDE 10.05 no Sistema Operacional Ubuntu Linux 13.04. Também foi usado o compilador GNU GCC Compiler via linha de comando e a ferramenta MakeFile. Não foram feitos testes do programa em ambiente Windows. Segue os comandos para compilar o código :

Linha de comando:

```
cd Documentos
gcc -c main.c busca.c
gcc main.o busca.o -o tp0
./tp0 input.txt output.txt
```

MakeFile

```
cd Documentos
make
```

O código está dividido em três arquivos principais: main.c com o programa principal, busca.c e busca.h com a implementação do TAD da busca em profundidade.

## 2.2. Funções

As funções que formam o TAD serão apresentadas a seguir:

### 2.2.1 void FLVazia(TipoLista \*Lista);

A função é responsável por inicializar uma lista vazia de vértices adjacentes do grafo.

### 2.2.2 int ListaVazia(TipoLista Lista);

Verifica se a lista de adjacência está vazia.

### 2.2.3 void Insere(TipoItem \*x, TipoLista \*Lista);

Insere um vértice depois do último item da lista.

### 2.2.4 void FPVazia(TipoPilha \*Pilha);

Cria uma pilha vazia de vértices.

### 2.2.5 int PilhaVazia(TipoPilha Pilha);

Verifica se uma pilha está vazia.

### 2.2.6 void Empilha(TipoItem x, TipoPilha \*Pilha);

Empilha um vértice no topo da Pilha.

### 2.2.7 void Desempilha(TipoPilha \*Pilha, TipoItem \*Item);

Desempilha um vértice da Pilha.

### 2.2.8 void InsereAresta(TipoValorVertice \*V1, TipoValorVertice \*V2, TipoGrafo \*Grafo);

Insere uma aresta entre 2 vértices.

### 2.2.9 void FGVazio(TipoGrafo \*Grafo);

Cria um Grafo Vazio.

### 2.2.10 int ListaAdjVazia(TipoValorVertice \*Vertice, TipoGrafo \*Grafo);

Verifica se uma lista está vazia.

### 2.2.11 TipoApontador PrimeiroListaAdj(TipoValorVertice \*Vertice, TipoGrafo \*Grafo);

Retorna o primeiro vértice da lista de adjacentes de um dado vértice.

### 2.2.12 void ProxAdj(TipoValorVertice \*Adj, TipoApontador \*Prox, char \*FimListaAdj);

Retorna o vértice Adj apontado por Prox.

### 2.2.13 void BuscaEmProfundidade(TipoGrafo \*Grafo, TipoValorVertice \*Inicial, TipoValorVertice \*Final);

Busca em profundidade iterativa usando pilha em que encontra o caminho entre 2 vértices.

A cada vértice do grafo, este possui uma lista de vértices que estão ligados à ele por meio da lista de adjacência. O algoritmo recebe como parâmetro o vértice inicial e final de consulta.

O seu vértice inicial está inicialmente branco. Então ele é empilhado e logo depois olhamos todos os vértices que fazem parte da lista de vértices do vértice inicial. Se o primeiro for branco, empilhamos e passamos a olhar a lista de vértices desse novo vértice até que o vértice final seja encontrado.

Na implementação, chamamos a função PrimeiroListaAdj, onde passamos como parâmetro o Grafo e o vértice. Essa função retorna o primeiro elemento da lista do vértice. Se ele for cinza, achamos o Prox dele que seja branco. Se não tiver nenhum branco, o vértice é desempilhado da pilha. O algoritmo abaixo exemplifica a ideia:

```
BuscaEmProfundidade(Grafo, Vinicial, Vfinal)
1 para u ← 1 até NumVertices faça
2   cor[u] ← branco
3 cor[Vinicial] ← cinza
4 P ← CRIA-PILHA (r)
5 enquanto P não estiver vazia faça
6   enquanto existir adjacentes a r não branco e diferente de Vfinal faça
7     u ← COPIA-TOPO-DA-PILHA (P)
8     v ← PRÓXIMO (Adj[u])
9     se v ≠ NIL
10      então se cor[v] = branco
11        então cor[v] ← cinza
12          COLOCA-NA-PILHA (v, P)
13      senão cor[u] ← preto
14      TIRA-DA-PILHA (P)
```

## 2.3 Programa Principal

Contém a chamada da função main. Faz uso de todas as estruturas e módulos citados acima, trabalha com o tratamento da entrada e saída, bem como a interpretação dos comandos e verificação da sua consistência.

O programa principal lê do arquivo input.txt, os arquivos de entrada para vários testes. A saída é direcionada para o arquivo output.txt.

### 3-Analise de Complexidade

Podemos supor que cada invocação de `PróximoListaAdj`, `FPVazia`, `PrimeiroListaAdj`, `Empilha`, `Desempilha` consome tempo constante (ou seja, uma quantidade de tempo que não depende do tamanho do grafo).

O bloco de linhas 1-4 consome tempo  $O(V)$  onde  $V$  é o número de vértices. Agora considere o processo iterativo nas linhas 5-14 do algoritmo Busca-em-Profundidade. Cada execução bloco de linhas 6-14, (a) examina lista de adjacentes (linhas 9-12) ou (b) pinta de preto um vértice originalmente cinza (linhas 12-14).

Uma execução de qualquer das duas alternativas consome tempo constante. Como um vértice preto jamais volta a ser cinza, o consumo total das ocorrências de (b) ao longo da execução do algoritmo Busca-em-Profundidade é  $O(V)$ .

Agora considere o consumo total das ocorrências de (a). Ao longo da execução do algoritmo, cada vértice do grafo é examinado no máximo uma vez (a definição de `Próximo` garante isso). Portanto, o consumo total de todas as ocorrências de (a) é  $O(A)$ , sendo  $A$  o número de vertices adjacentes.

Assim, o consumo de tempo do algoritmo Busca-em-Profundidade É  $O(V+A)$ . Portanto, o algoritmo é linear.

### 4- Testes

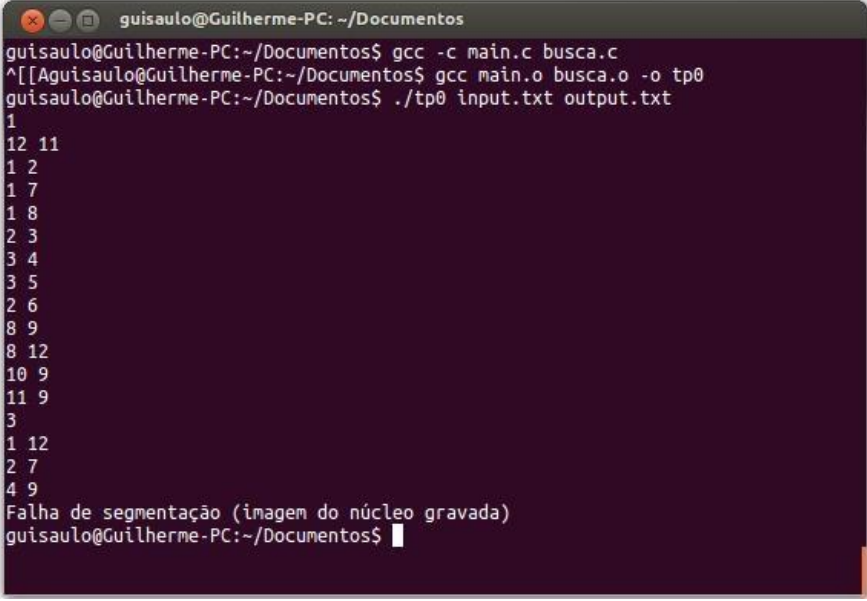
Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória.

Também foram realizados testes no CodeBlocks durante a elaboração do trabalho para verificar os valores de entrada e saída de algumas variáveis.

As figuras abaixo mostram a saída do teste disponibilizado no enunciado do TP:

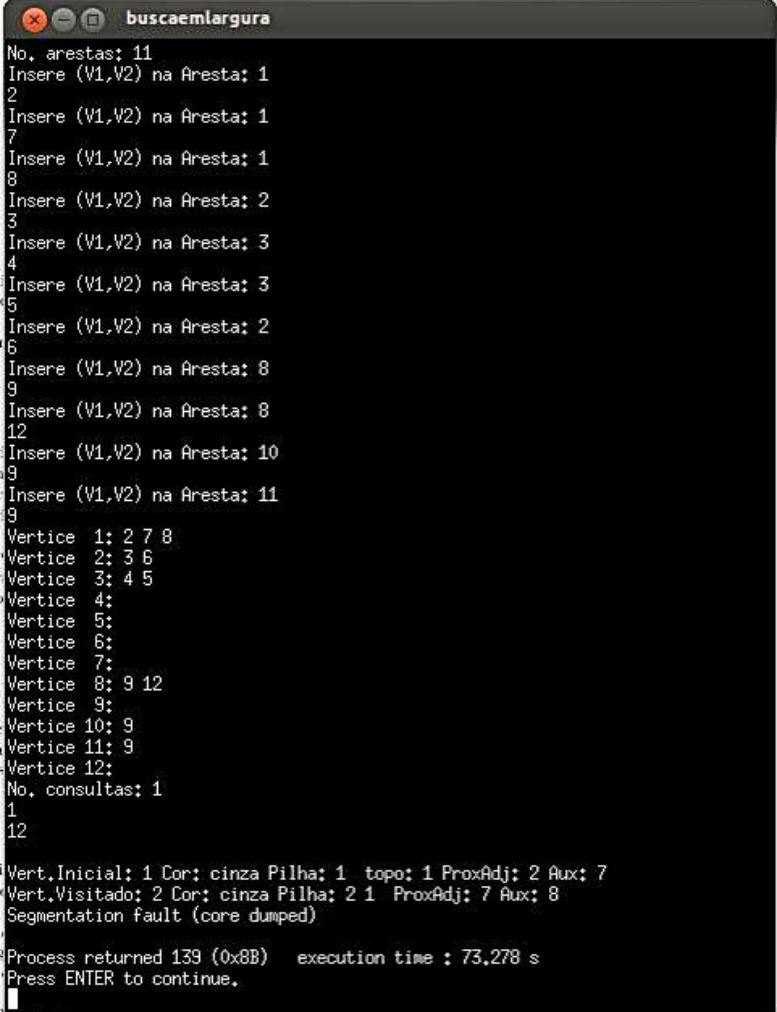
- 1- Teste via Linha de comando: Lê entrada mas ocorre falha de segmentação na função busca. \*

```
1 1
2 12 11
3 1 2
4 1 7
5 1 8
6 2 3
7 3 4
8 3 5
9 2 6
10 8 9
11 8 12
12 10 9
13 11 9
14 3
15 1 12
16 2 7
17 4 9
```



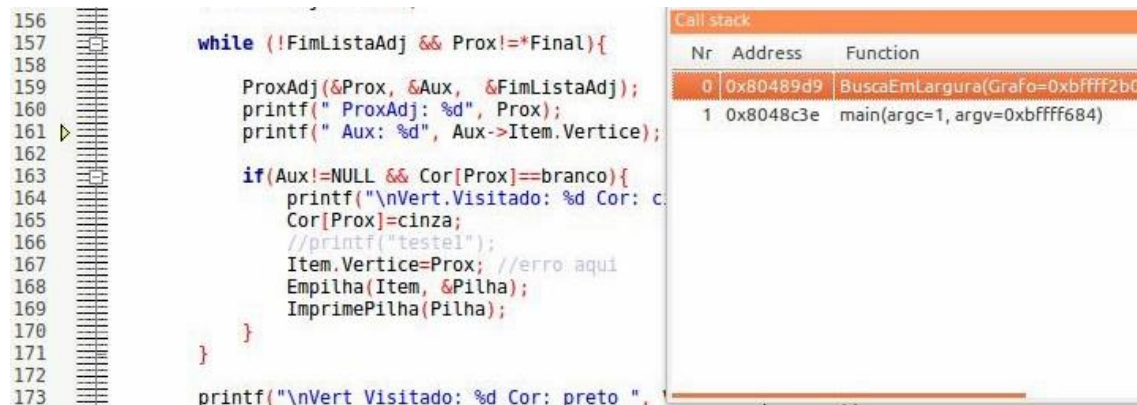
```
guisaulo@Guilherme-PC: ~/Documentos
guisaulo@Guilherme-PC:~/Documentos$ gcc -c main.c busca.c
^[[Aguisaulo@Guilherme-PC:~/Documentos$ gcc main.o busca.o -o tp0
guisaulo@Guilherme-PC:~/Documentos$ ./tp0 input.txt output.txt
1
12 11
1 2
1 7
1 8
2 3
3 4
3 5
2 6
8 9
8 12
10 9
11 9
3
1 12
2 7
4 9
Falha de segmentação (imagem do núcleo gravada)
guisaulo@Guilherme-PC:~/Documentos$
```

- 2 - Teste no Codeblocks, com a verificação dos caminhos das variáveis. Também com erro de segmentação na busca em profundidade. \*



```
buscaemlargura
No. arestas: 11
Insere (V1,V2) na Aresta: 1
2
Insere (V1,V2) na Aresta: 1
7
Insere (V1,V2) na Aresta: 1
8
Insere (V1,V2) na Aresta: 2
3
Insere (V1,V2) na Aresta: 3
4
Insere (V1,V2) na Aresta: 3
5
Insere (V1,V2) na Aresta: 2
6
Insere (V1,V2) na Aresta: 8
9
Insere (V1,V2) na Aresta: 8
12
Insere (V1,V2) na Aresta: 10
9
Insere (V1,V2) na Aresta: 11
9
Vertice 1: 2 7 8
Vertice 2: 3 6
Vertice 3: 4 5
Vertice 4:
Vertice 5:
Vertice 6:
Vertice 7:
Vertice 8: 9 12
Vertice 9:
Vertice 10: 9
Vertice 11: 9
Vertice 12:
No. consultas: 1
1
12
Vert.Inicial: 1 Cor: cinza Pilha: 1 topo: 1 ProxAdj: 2 Aux: 7
Vert.Visitado: 2 Cor: cinza Pilha: 2 1 ProxAdj: 7 Aux: 8
Segmentation fault (core dumped)
Process returned 139 (0x8B) execution time : 73,278 s
Press ENTER to continue.
```

**\* Problema de Segmentação:** Um erro de segmentation fault ocorre sempre que chega no terceiro vértice adjacente. O código chega a ler os dois primeiros vértices, mas no terceiro dá erro de segmentação quando acessamos `Aux->Item.Vertice`, conforme imagem abaixo:



```

156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
while (!FimListaAdj && Prox!=*Final){
    ProxAdj(&Prox, &Aux, &FimListaAdj);
    printf(" ProxAdj: %d", Prox);
    printf(" Aux: %d", Aux->Item.Vertice);

    if(Aux!=NULL && Cor[Prox]==branco){
        printf("\nVert.Visitado: %d Cor: c
        Cor[Prox]=cinza;
        //printf("testel");
        Item.Vertice=Prox; //erro aqui
        Empilha(Item, &Pilha);
        ImprimePilha(Pilha);
    }
}

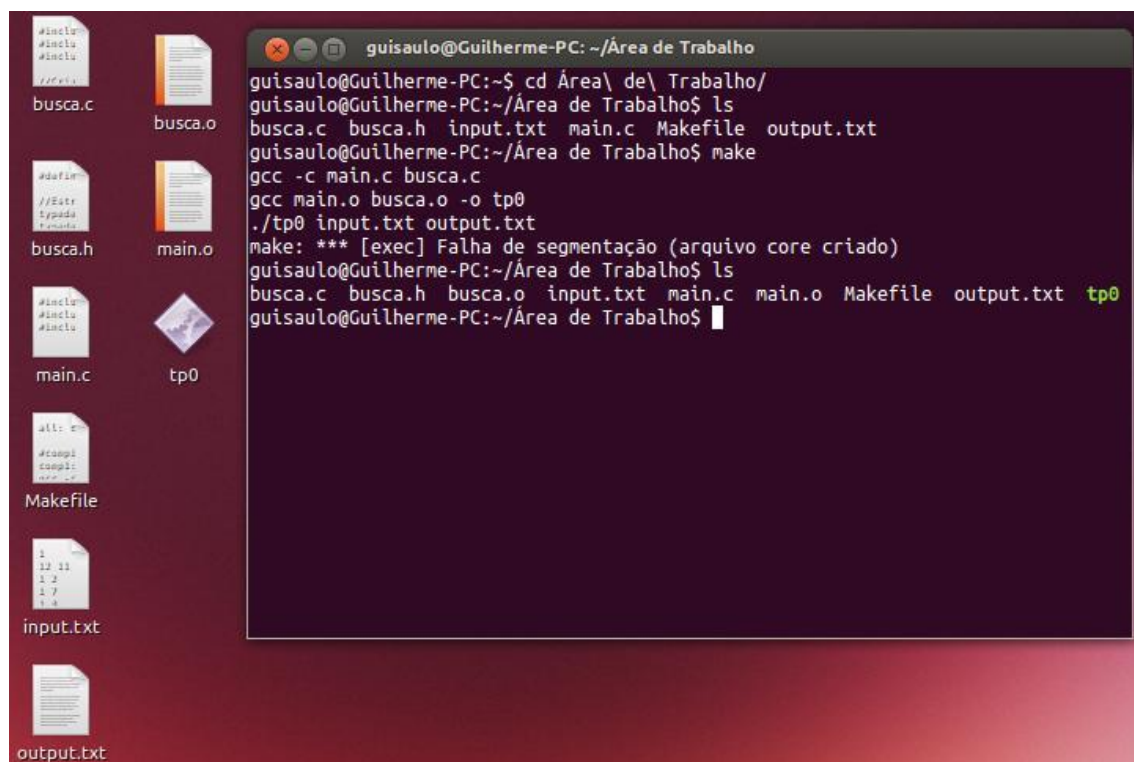
printf("\nVert Visitado: %d Cor: preto ",

```

Nr	Address	Function
0	0x80489d9	BuscaEmLargura(Grafo=0xbffff2b0
1	0x8048c3e	main(argc=1, argv=0xbffff684)

Várias tentativas foram feitas para debugar este erro, porém sem sucesso. Devido à isso, não consegui imprimir o caminho dos vértices no arquivo de saída, pois o bug acontece antes da impressão, mesmo com a ideia correta do algoritmo.

### 3- Teste da compilação via mikefile



```

guisaulo@Guilherme-PC: ~/Área de Trabalho
guisaulo@Guilherme-PC:~$ cd Área\ de\ Trabalho/
guisaulo@Guilherme-PC:~/Área de Trabalho$ ls
busca.c busca.h input.txt main.c Makefile output.txt
guisaulo@Guilherme-PC:~/Área de Trabalho$ make
gcc -c main.c busca.c
gcc main.o busca.o -o tp0
./tp0 input.txt output.txt
make: *** [exec] Falha de segmentação (arquivo core criado)
guisaulo@Guilherme-PC:~/Área de Trabalho$ ls
busca.c busca.h busca.o input.txt main.c main.o Makefile output.txt tp0
guisaulo@Guilherme-PC:~/Área de Trabalho$

```

Files in file manager:

- busca.c
- busca.h
- main.c
- Makefile
- input.txt
- output.txt
- busca.o
- main.o
- tp0

## 5-Conclusão

Com elaboração desse trabalho foi possível relembrar as técnicas de programação aprendidas na disciplina de AEDS 2 e também fixar os novos conceitos buscas em grafos.

Com a elaboração deste trabalho, percebe-se que a busca em profundidade iterativa, usando pilhas, é uma estrutura de dados extremamente eficiente para saber o caminho entre duas localidades.

Mesmo com problemas de segmentação, que resultou na não impressão dos arquivos de saída, foi possível fixar os conceitos de estrutura de dados, modularização, alocação dinâmica de memória e compilação do código através da ferramenta MikeFile.

## 6-Referências

[1] Nívio Ziviani, Projeto de Algoritmos com implementação em Pascal e C - <http://www2.dcc.ufmg.br/livros/algoritmos/> Acessado em 01 de Março de 2013.

[2] Paulo Feofiloff – Analises de Algoritmos - [http://www.ime.usp.br/~pf/analise\\_de\\_algoritmos/aulas/dfs.html](http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dfs.html) /Acessado em 03 de Março