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 1 – Pesquisas em Grafos

Guilherme Saulo Alves

Professores - Jussara Marques, Olga Nikolaevna, Marcos Augusto, Wagner Meira

1-Introdução

O proposito deste trabalho consiste fazer consultas em grafos utilizando busca em largura, busca em profundidade e algoritmo de Dijkstra, para calcular o caminho mais curto. A consulta retorna o caminho percorrido pelas buscas listando os vértices à medida em que são descobertos. Para isso, devemos utilizar o TAD de filas, pilha e Heap.

2-Implementação

Uma das primeiras implementações foi criar estrutura para representar um grafo por lista, em que indica a lista de vértices adjacentes de um vértice. Para busca em largura, foi utilizado o TAD de fila. A busca em profundidade utiliza o TAD de pilhas e o Algoritmo de Dijkstra utiliza o TAD de heap. Os algoritmos serão explicados 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 (comando make). 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 buscalargura.c buscaprofundidade.c dijkstra.c grafo.c
gcc main.o buscalargura.o buscaprofundidade.o dijkstra.o grafo.o -o tp1
./tp1 input.txt output.txt
```

O código está dividido em seis arquivos principais:

- main.c: programa principal;
- grafo.c: funções do TAD grafo por lista de adjacências;
- buscalargura.c: funções do TAD da busca em largura;
- buscaprofundidade.c: funções do TAD da busca em profundidade;
- dijkstra.c: funções do TAD que calcula o caminho mais curto;
- busca.h: declaração das estruturas e cabeçalho das funções utilizadas.

2.2. Funções e Algoritmos

As soluções dos problemas serão apresentadas a seguir, através de pseudocódigos e esquemas ilustrativos.

2.2.1 void BuscaEmLargura(TipoValorVertice u, TipoValorVertice Final, TipoGrafo *Grafo, FILE *saida);

A busca em largura explora as arestas de um grafo com o objetivo de descobrir todos os vértices que são alcançáveis a partir de *u.* A busca expande a fronteira entre vértices descobertos e não descobertos como se fossem ondas geradas por uma pedra ao cair em uma superfície de agua parada. O pseudocódigo abaixo usa uma fila para gerenciar o conjunto de vértices:

```
BuscaEmLargura (u, Final, Grafo)
1       para i ← 1 até Número de Vértices faça
2               cor[i] ← branco
3       cor[u] ← cinza
4       enfileira (u, Fila)
5       enquanto existir vértices na fila && u é diferente de Final faça
6               desenfileira (u, Fila)
7               se existir vértices adjacentes v ao vértice u então
8                       se cor[v]=branco entao
9                               cor[v] ← cinza
10                              enfileira(v, Fila)
11              cor[u]=preto
```

A busca ***BuscaEmLargura*** funciona como se segue. O primeiro ***for*** colore todos os vértices de branco (linha 1 e 2). Quando um vértice é descoberto pela primeira vez, ele se torna cinza (linha 3). A seguir a fila é inicializada com o vértice origem *u* (linha 4). O primeiro anel ***while*** (linha 5) executa enquanto houver vértices cinza, que formam o conjunto de vértices descobertos que ainda não tiveram suas listas de adjacentes totalmente examinadas.

No primeiro comando dentro do anel, o vértice *u* que está no início da fila é desenfileirado (linha 6). O comando ***if*** seguinte examina a lista de vértices *v* adjacentes a *u* e visita *v* se ele for branco (linha 7 e 8). Neste momento, *v* é tornado cinza e o novo vértice cinza é enfileirado (linha 9 e 10). Finalmente, depois que toda a lista de adjacentes de *u* é percorrida, o vértice *u* é pintado de preto e temos o caminho percorrido (linha 11).

2.2.2 void BuscaEmProfundidade(TipoValorVertice u, TipoValorVertice Final, TipoGrafo *Grafo, FILE *saida);

A estratégia seguida pela busca em profundidade é a de buscar, sempre que possível, o mais profundo no grafo. A busca foi implementada de forma iterativa usando pilha. O pseudocódigo abaixo usa uma pilha para gerenciar o conjunto de vértices:

```
BuscaEmProfundidade (u, Final, Grafo)
1       para i ← 1 até Número de Vértices faça
2               cor[i] ← branco
3       cor[u] ← cinza
4       empilha (u, Pilha)
5       enquanto a pilha não estiver vazia faça
6               u ← VerticeTopoPilha
7               enquanto existir vértices v adjacentes à u e v != Final faça
8                       se v ≠ NIL && cor[v]=branco então
9                               cor[v] ← cinza
10                              empilha (v, Pilha)
11                      cor[u] ← preto
12                      desempilha (u, Pilha)
```

O método ***BuscaEmProfundiade*** funciona como se segue. O primeiro ***for*** colore todos os vértices de branco (linha 1 e 2). Quando um vértice é descoberto pela primeira vez, ele se torna cinza (linha 3). A seguir a pilha é inicializada com o vértice origem *u* (linha 4). O primeiro anel ***while*** (linha 5) executa enquanto existir vértices na pilha. A seguir (linha 6) o vértice do topo da pilha é atribuído ao vértice u.

O segundo anel ***while*** (linha 7) executa enquanto houver vértices v adjacentes não descobertos em que v seja diferente do vértice final (condição de parada). O comando ***if*** seguinte examina a lista de vértices *v* adjacentes a *u* e visita *v* se ele for branco (linha 8 e 9). Neste momento, *v* é tornado cinza e o novo vértice cinza é empilhado (linha 9 e 10). Depois que toda a lista de adjacentes de *u* é percorrida, o vértice *u* é pintado de preto e desempilhado da pilha (linha 11 e 12). Se não existir mais vértices na pilha, temos o caminho percorrido.

2.2.3 void Dijkstra(TipoValorVertice *Raiz, TipoValorVertice Final, TipoGrafo *Grafo, FILE *saida);

O algoritmo para apresentar o caminho mais curto é conhecido como algoritmo de Dijkstra. O algoritmo mantém um conjunto S de vértices cujos caminhos mais curtos até um vértice raiz já são conhecidos. Ao final de sua execução, o algoritmo produz uma arvore de caminhos mais curtos a partir de todos os vértices que são alcançáveis a partir de raiz. Veja abaixo:

```
Dijkstra (Raiz, Final, Grafo)
1       para i ← 0 até Número de Vértices faça
2               p[i] ← Infinito
3               antecessor[i]= -1
4       p[raiz]=0
5       Constrói heap sobre vértices do grafo;
6       S=ø
7       enquanto o heap não estiver vazio faça
8               u ← heap.retiraMin();
9               S ← S + u
10              enquanto existir vértices v adjacentes à u
11                      se p[v] > p[u] + peso da aresta (u, v) então
12                              p[v] ← p[u] + peso da aresta (u, v)
13                              antecessor [v] ← u
```

As linhas 1-3 realizam a inicialização dos antecessores e das estimativas de caminhos mais curtos. A linha 4 inicializa a distância do vértice raiz a ele mesmo como sendo zero. A linha 5 constrói o heap sobre todos os vértices do grafo, e a linha 6 inicializa o conjunto solução *S* como vazio. A linha 7 é um anel que executa enquanto o heap não estiver vazio.

A cada iteração do anel nas linhas 8-13 um vértice *u* é extraído do heap e adicionado ao conjunto *S*. Na linha 8 a operação *retiraMin* obtém o vértice *u* que contém o caminho mais curto estimado até aquele momento e o adiciona ao conjunto solução *S*. A seguir, no anel da linha 10, a operação de relaxamento é realizada sobre cada aresta adjacente ao vértice *u*, atualizando o caminho estimado *p[v]* e o *antecessor[v]* se o caminho mais curto para *v* puder ser melhorado usando o caminho por meio de *u*.

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

Para as análises abaixo, adote V e A como número de vértice e arestas.

Busca em Largura: O custo de inicialização do primeiro anel é O(|V|). Seguindo temos a operação de tornar o vértice branco e enfileirar ou desenfilerar. Como essas operações tem custo O(1) cada uma, as operações tem custo total O(|V|). A lista de adjacentes de cada vértice é percorrida apenas quando o vértice é desenfileirado, logo, cada lista de adjacentes é percorrida no máximo uma vez. Já que a soma de todas as listas de adjacentes é O(|A|), o tempo total gasto com as listas de adjacentes é O(|A|). Logo, o método ***BuscaEmLargura*** possui complexidade de tempo total igual a **O(|V|+|A|)**.

Busca em Profundidade: O custo de inicialização do primeiro anel é O(|V|). Seguindo temos a operação de tornar o vértice branco e empilhar ou desempilhar. Como essas operações tem custo O(1) cada uma, as operações tem custo total O(|V|). A lista de adjacentes é percorrida no máximo uma vez, pois é chamada quando atribuímos a u o vértice no topo da pilha. Já que a soma de todas as listas de adjacentes é O(|A|), o tempo total gasto com as listas de adjacentes é O(|A|). Logo, a ***BuscaEmProfundidade*** possui complexidade de tempo total igual a **O(|V|+|A|)**.

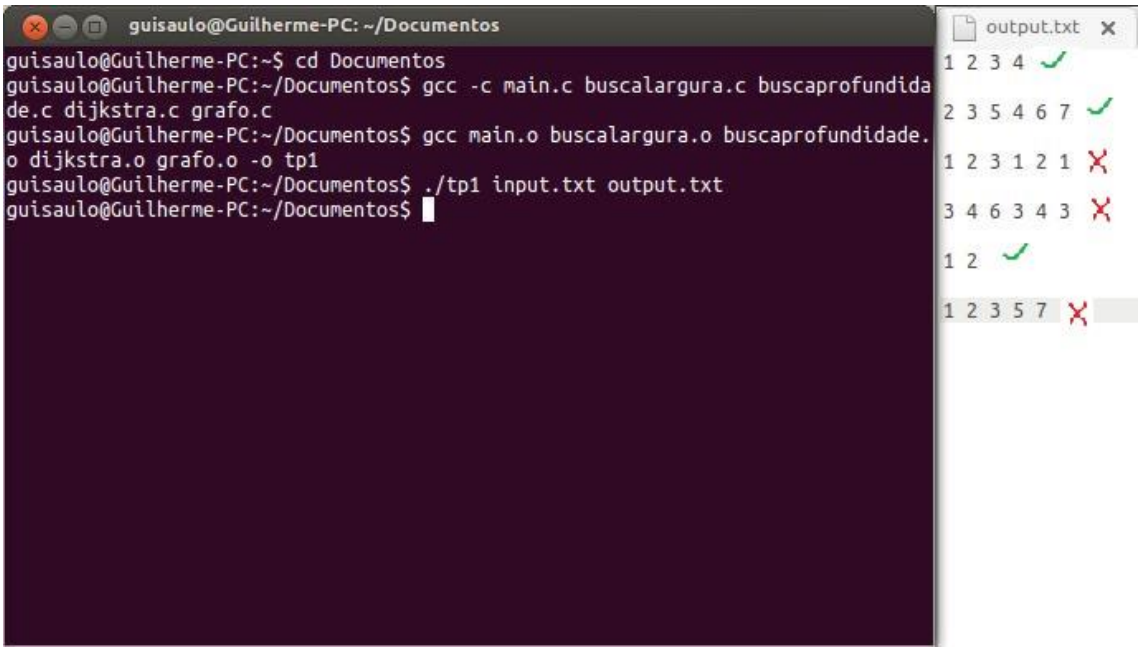
Algoritmo de Dijkstra: A fila de prioridade é implementada como um heap com o método *constroi* com custo O(|V|). O corpo do anel *while* é executado |V| vezes e, desde que o método refaz tem custo O(log|V|), o tempo total para executar a operação retira o item com menor peso é O(|V|log|V|). O *while* mais interno que percorre a lista de adjacentes é executado O(|A|) vezes ao todo, uma vez que a soma dos comprimentos de todas as listas de adjacência é 2|A|. A operação diminuiChave é executada sobre o heap na posição pos[v], a um custo O(log|V|). Logo, o tempo total para executar o algoritmo é **O(|V| log |V| + |A| log |V|) = O(|A| log |V|).**

4-Testes

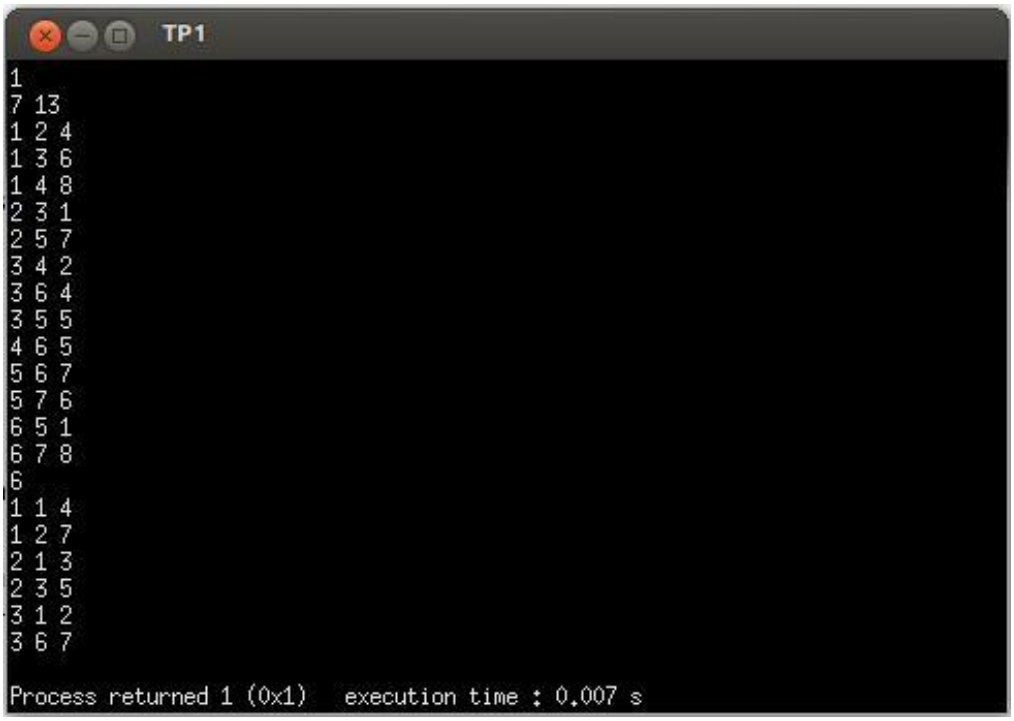
Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória.

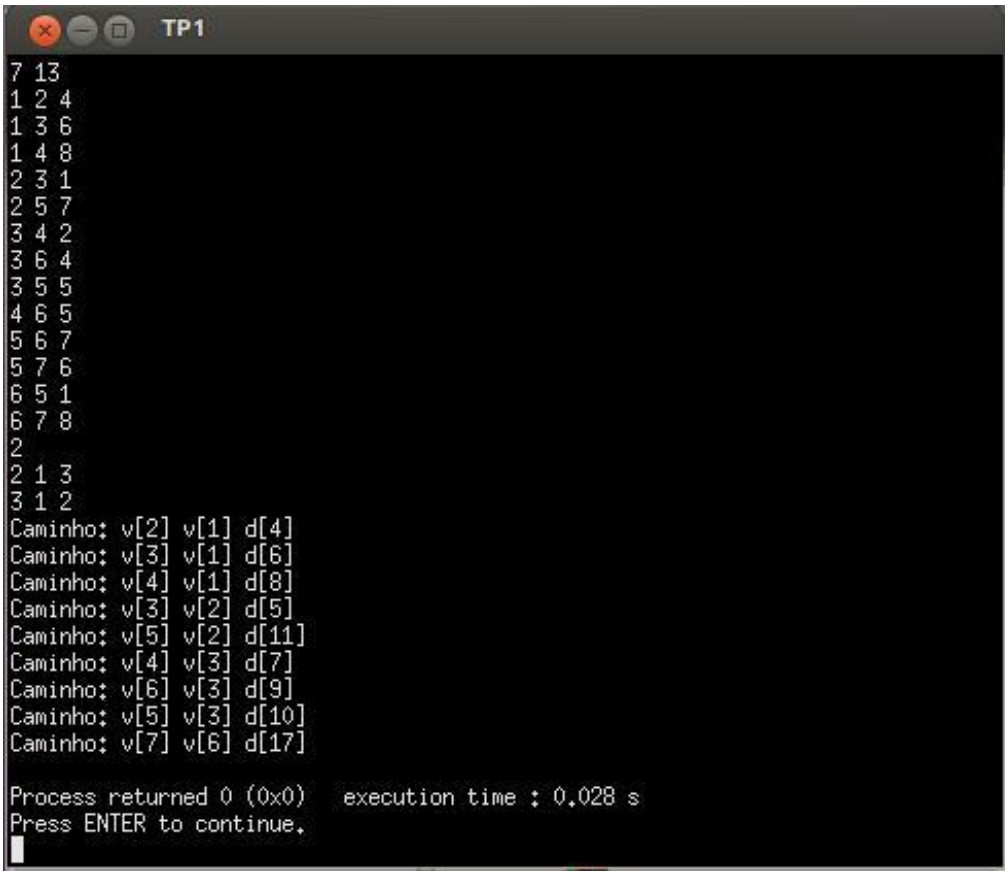
Também foram realizados testes no CodeBlocks durante a elaboração do trabalho para verificar os valores de entrada e saída de algumas variáveis.

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



1- *Teste via Linha de comando:* Lê entrada e imprime resultado no arquivo output.txt. A busca em largura foi implementada corretamente. Porém não foi possível implementar a busca em profundidade corretamente, devido a dificuldades encontradas no TP0. O caminho mais curto usando Dijkstra foi implementado corretamente no caso em que o raiz é o primeiro vértice. Casos em que queremos encontrar o caminho mais curto entre dois vértices, em que a origem seja diferente da raiz, não foi implementado, como no último exemplo.





2 - Testes no *Codeblocks* para verificar se as entradas foram lidas corretamente e para verificar o caminho do algoritmo dijkstra.

5-Conclusão

Com elaboração desse trabalho foi possível relembrar as técnicas de programação aprendidas na disciplina de AEDS 2 e também fixar os novos conceito de buscas em grafos. Percebe-se que a busca em grafos tem enorme importância na ciência da computação e seus algoritmos são essenciais para a área.

Mesmo com problemas de implementação que resultaram na impressão errada de algumas instancias, 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 15 de Março de 2013.

[2] Paulo Feofiloff – Analises de Algoritmos - http://www.ime.usp.br/~pf/analise_de_algoritmos/lectures.html /Acessado em 15 de Março