Caminho Euleriano

Katielen Silva[1] **e Luiz Antônio Cubas Junior**[1]

[1]Departamento Acadêmico de Informática
Universidade Tecnológica Federal do Paraná (UTFPR)
Avenida Sete de Setembro, 3165 - Rebouças - 80.230-901 – Curitiba – PR – Brazil

{katielens,ljunior}@alunos.utfpr.edu.br

1. Problema

O trabalho consiste em implementar um algoritmo que resolva o problema do ciclo/caminho/circuito euleriano, utilizando protótipos e códigos para grafos disponibilizados nas aulas de Estruturas de Dados 2.

2. Descrição

A codificação mostrou-se desafiadora ao percebermos que não era uma simples execução do algoritmo de busca em profundidade já implementado em sala de aula. De certo modo este achismo é uma meia verdade. Utilizamos funções já implementadas de visitação em profundidade, contudo diversas modificações tiveram que ser feitas, além de algumas funções novas.

Para a criação do grafo e visitação em caminho euleriano dos vértices, foi utilizada a linguagem C. Além disso, a base da solução encontrada por nossa equipe para o problema é o algoritmo de busca em profundidade visto em sala de aula, utilizando uma lista encadeada para representar os vértices e suas adjacências – no caso cada vértice seria uma posição de um vetor, em que cada uma abriga um endereço de memória para uma lista encadeada contendo as adjacências –.

3. Funções Novas

3.1. `verificaCircuito`

Esta implementação simplesmente verifica se determinado grafo informado é ou não um circuito euleriano. Segundo a teoria, um circuito euleriano acontece apenas em grafos conexos onde todos os vértices são pares e as arestas são utilizadas apenas uma vez no caminho, além de todos os vértices terem que necessariamente ser visitados, podendo ocorrer mais de uma vez, bem como todas as arestas têm que ser utilizadas, sem repetição. Logo nossa função verifica o grau de cada vértice/nodo, buscando aquele que não tem grau par, caso essa condição seja satisfeita é retornado 0, do contrário, 1. Para a `verificaCircuito` funcionar, uma função auxiliar foi criada com o nome de `grauVertice`.

3.2. `grauVertice`

Esta aplicação apenas conta o número de elementos na lista de adjacências de cada vértice retornando a quantidade verificada à função `verificaCircuito`.

3.3. `deletaAresta`

Integrante das funções principais, esta é talvez a mais importante função criada por nós para o desenvolvimento do caminho euleriano. Como dito de início, acreditávamos que a implementação do trabalho seria apenas a execução do algoritmo de busca em profundidade sobre um grafo conexo qualquer, logo não pensava-se em "eliminar arestas" (na realidade elimina-se nodos de listas encadeadas) do grafo. Acreditava-se que "pintar" (atribuir um valor inteiro a parte à cada nodo para identificar a passagem do algoritmo pelos vértices) fosse o suficiente.

No decorrer do processo de codificação notamos a necessidade de deletar os caminhos já utilizados do grafo, pois apenas pintar os nodos faria o algoritmo retornar ao vértice já visitado pela mesma aresta, visto que no circuito euleriano os vértices podem repetir, mas as arestas não. Assim desenvolveu-se o algoritmo.

De maneira geral, ele busca a aresta que liga um vértice a outro, dependendo da direção pode ser vértice pai e vértice filho, e vice versa, e executa a função da biblioteca padrão, `free`, eliminando o nodo pai e o nodo filho das respectivas listas de adjacências. Assim aquele caminho não existe mais no grafo, evitando a repetição de arestas.

Como adendo, a função retorna o identificador do vértice que atualmente está na pilha de recursão da função `DFS_Visit_Trabalho2`, esta que será explicada mais tarde. Ele é nomeado de "filho" aqui. Este retorno é necessário para evitar o acesso inválido à posições de memória em `DFS_Visit_Trabalho2`. A `deletaAresta` é executada a cada execução daquela, desde que haja(m) aresta(s) para explorar em determinado vértice.

4. Funções Alteradas

4.1. `DFS_Visit_Trabalho2`

Assim como na função `DFS_Visit` vista em aula, a nossa modificação também visita os vértices do grafo por recursão. Contudo, para resolver o problema deste trabalho fizemos as seguintes modificações:

- Agora a função recebe também por parâmetro uma variável identificada como "raiz". Isto é necessário para a implementação de uma condição comentada mais a frente.
- No início da função, uma instrução imprime o identificador dos vértices visitados.
- Não há mais a "coloração" de cinza e preto dos vértices visitados e terminados, respectivamente.
- O laço de repetição para acesso as adjacências ainda permanece, com exceção da condição principal de acesso (*if*). Agora somente vértices com identificadores diferentes da raiz, este é o motivo do novo parâmetro comentado a pouco, e com

grau maior que zero (verificação feita com auxílio da função `grauVertice`) são aceitos para a chamada da recursão. Foi mantida nesta mesma condição a verificação de cor branca no vértice atual, apesar de ser desnecessária, ela mantém a base da `DFS_Visit` original.

- Entretanto, antes da recursão é necessário deletar a aresta utilizada anteriormente, assim a função `deletaAresta` é utilizada, retornando um valor que é guardado em "vIdentificadorSalvo", esta variável é importante, como foi comentado na descrição da aplicação `deletaAresta`. Foi mantida a atribuição do valor que identifica o pai do vértice atual, pois ele é necessário para a chamada daquela função de eliminação de arestas.
- Além da condição principal, uma alternativa (*else*) foi implementada. Ela é similar à aquela, com a diferença de tratar casos onde a raiz foi inserida na lista de adjacências de determinado vértice por fim ou em qualquer posição diferente da última. No caso, em uma lista encadeada a raiz ficaria na "cabeça" da lista ou próxima desta, fazendo o algoritmo, sem este tratamento, visitá-la antes do final do mesmo ou antes de utilizar as outras arestas de determinado vértice que tem ligação com a raiz. Logo, esta alternativa "pula" a visitação da raiz, deixando-a para o final do programa, como volta do circuito ou para quando não houver outro caminho para seguir na volta da recursão.

A função `DFS_Visit_Trabalho2` é essencial para o funcionamento do algoritmo. Levamos um bom tempo para codificá-la de forma correta, ou de uma forma que resolvesse o problema proposto. Foi nesta aplicação que descobriu-se que o algoritmo acessava posições inválidas de memória no retorno da recursão. Pois após a execução da `deletaAresta` nodos deixam de existir, mas na volta da chamada a verificação de continuidade do laço, `v != NULL`, não é executada novamente, acessando assim posições inválidas e chamando aplicações com endereços inválidos nos parâmetros. Tal problema foi contornado com o uso da função `grauVertice`, verificando se o números de nodos no vértice atual é maior que 0, na condição principal e na alternativa. E com isto evitou-se problemas de memória, que eram muitos.

4.2. `Busca_Profundidade_Trabalho2`

Com poucas alterações em relação a `Busca_Profundidade` original, aqui inserimos uma condição que comprova a existência de um circuito euleriano no grafo informado, usando a função desenvolvida pela equipe, `verificaCircuito`. Caso a condição não seja satisfeita, não há visitação de vértices e a mensagem "O grafo informado não possui um circuito euleriano" é exibida na tela.

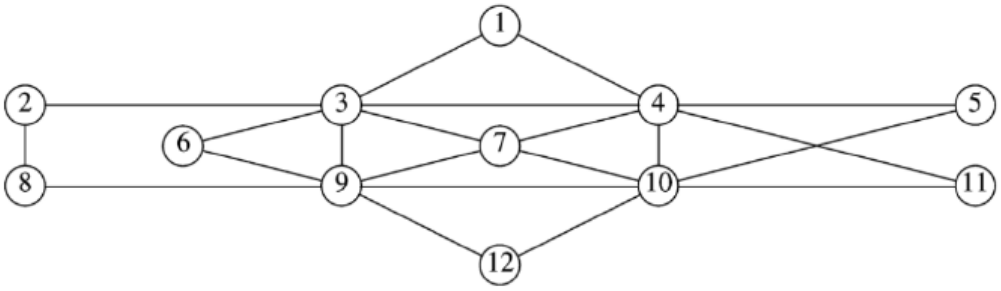
Além disso, inserimos também mais um parâmetro na chamada da função, um inteiro de nome raiz, equivalentemente à função `DFS_Visit_Trabalho2`. Tal inserção fez-se necessária pois queríamos ter o controle de onde o circuito começa, além deste valor ser necessário na função `DFS_Visit_Trabalho2`, como descrito anteriormente. Portanto, o que é recebido aqui por parâmetro, no caso do inteiro "raiz", também é recebido naquela, pois estas duas funções são dependentes uma da outra.

Como adendo, foi retirado o *loop* que tratava casos de grafos não conexos na busca em profundidade original, uma vez que na especificação do trabalho há a informação da possibilidade de desconsiderar grafos desconexos na implementação.

5. Testes

Os seguintes testes foram realizados tendo como base os grafos da especificação do trabalho com números identificadores dos vértices inseridos de forma aleatória, mas sem repetição, tanto nos vértices das figuras, com exceção do grafo 1 que já está numerado na especificação, quanto no vetor de adjacências.

5.1. Grafo 1

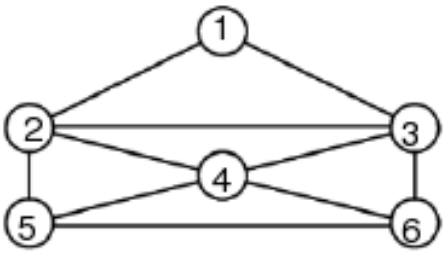


Saída do algoritmo:

```
[luluhacker@blulight Trabalho 2]$ make T2
make: 'T2' is up to date.
[luluhacker@blulight Trabalho 2]$ ./T2
Listas de adjacencias de G:
Nó 1 :  3  4
Nó 2 :  3  8
Nó 3 :  1  4  2  7  6  9
Nó 4 :  1  3  5  7  11  10
Nó 5 :  4  10
Nó 6 :  3  9
Nó 7 :  4  10  3  9
Nó 8 :  2  9
Nó 9 :  3  7  6  8  10  12
Nó 10 :  12  9  7  4  5  11
Nó 11 :  10  4
Nó 12 :  9  10

Caminho euleriano:  5  4  1  3  4  7  10  12  9  3  2  8  9  7  3  6  9  10  4  11  10  5
[luluhacker@blulight Trabalho 2]$
```

5.2. Grafo 2

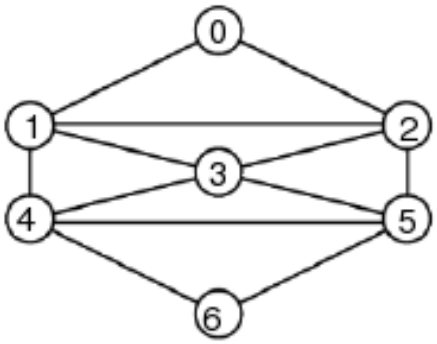


Saída do algoritmo:

```
[luluhacker@blulight ~]$ cd ljunior\@alunos.utfpr.edu.br/
[luluhacker@blulight ljunior@alunos.utfpr.edu.br]$ cd UTFPR/Estruturas\ de\ Dados\ 2/
[luluhacker@blulight Estruturas de Dados 2]$ cd Trabalho\ 2
[luluhacker@blulight Trabalho 2]$ make T2
cc     T2.c   -o T2
[luluhacker@blulight Trabalho 2]$ ./T2
Listas de adjacencias de G:
Nó 1 :  2  3
Nó 2 :  3  1  4  5
Nó 3 :  1  2  5  4
Nó 4 :  2  3  5
Nó 5 :  3  2  4

O grafo informado não possui um circuito euleriano.
[luluhacker@blulight Trabalho 2]$
```

5.3. Grafo 3

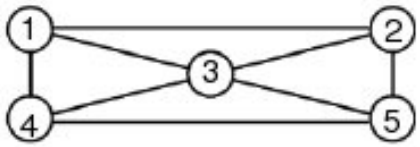


Saída do algoritmo:

```
[luluhacker@blulight ~]$ cd ljunior\@alunos.utfpr.edu.br/
[luluhacker@blulight ljunior@alunos.utfpr.edu.br]$ cd UTFPR/Estruturas\ de\ Dados\ 2/
[luluhacker@blulight Estruturas de Dados 2]$ cd Trabalho\ 2
[luluhacker@blulight Trabalho 2]$ make T2 && ./T2
make: 'T2' is up to date.
Listas de adjacencias de G:
Nó 0 :  1  2
Nó 1 :  0  2  3  4
Nó 2 :  0  1  3  5
Nó 3 :  1  2  4  5
Nó 4 :  1  3  5  6
Nó 5 :  2  3  4  6
Nó 6 :  4  5

Caminho euleriano:  2  0  1  3  4  1  2  3  5  4  6  5  2
[luluhacker@blulight Trabalho 2]$
```

5.4. Grafo 4



Saída do algoritmo:

```
[luluhacker@blulight Trabalho 2]$ make T2
cc     T2.c   -o T2
[luluhacker@blulight Trabalho 2]$ ./T2
Listas de adjacencias de G:
Nó 1 :  2  3  4
Nó 2 :  1  3  5
Nó 3 :  1  2  4  5
Nó 4 :  1  3  5
Nó 5 :  2  3  4

O grafo informado nao possui um circuito euleriano.
[luluhacker@blulight Trabalho 2]$
```

6. Conclusões

Apesar de ter sido trabalhoso codificar e resolver o problema do circuito euleriano, aprendemos muito. Melhoramos a nossa percepção do funcionamento de uma recursão e melhoramos também nosso conhecimento em relação ao gerenciamento de memória em programas codificados em linguagem C que utilizam alocação dinâmica. Foi um real desafio e também uma conquista do autorreconhecimento das nossas próprias capacidades em relação a programação. Ao final, passadas todas as frustrações, gostamos muito deste trabalho.