GFC - Um gerador de grafos de fluxo de controle a partir de códigos de três endereços.

Evaldo de Paula Souza

Visão Geral

Grafos são estruturas de dados que representam o relacionamento entre pares de objetos. Podem ser separados em dois grandes grupos, com base na forma como estabelecem essas relações: Os grafos não-direcionados e os grafos direcionados(também chamados dígrafos).
Em um grafo não-direcionado, cada ligação representa um par não ordenado de objetos, ou vértices. Assim não há diferença entre uma ligação {u,v} e uma ligação {v,u}. Em um grafo direcionado, a ligação é estabelecida entre um vértice de origem e um vértice de destino, assim a ligação {u,v} representa uma informação diferente da ligação {v,u}.
O presente programa implementa um grafo direcionado, pois um grafo de fluxo de controle, gerado a partir de um código de três endereços, deve representar a sequência de passos que o código de três endereços descreve.
Das duas principais formas de se implementar um grafo, através de uma matriz de adjacência ou de uma lista de adjacência, a que mais se adequa ao propósito do programa é a criação de uma lista, pois facilita a representação do fluxo do programa, bem como simplifica a representação de seus caminhos.
A linguagem de programação escolhida para construir esse programa foi a *Java versão 11.0.11*, devido às bibliotecas nativas da linguagem que facilitam o tratamento de textos e de coleções de dados. O programa foi originalmente construído no sistema *Ubuntu 20.04*, no entanto todos os comandos usados são nativos da linguagem *Java*, e portanto deve ser compatível com outros sistemas operacionais, salvo em casos extremos.

Implementação

Em essência, o programa lê um arquivo de texto linha por linha, salvando cada linha em uma estrutura chamada "**Comando**", que verifica se a linha contém "*goto ", "if",* e se é iniciada com um "*L"* , indicando que é uma linha de destino de um direcionamento. Então, tais comandos são ordenados em outra estrutura chamada "**Vertice",** que representa um bloco de instruções terminado em um desvio, condicional ou incondicional. Por fim, esses "vertices" são ordenados em uma estrutura chamada "**Grafo"**, onde seus desvios e destinos são mais claramente determinados.
O que se obtém ao final do processamento é uma estrutura clara e concisa do fluxo que o código de três endereços representa, suas ligações e desvios, bem como seus pontos de entrada e saída. Tudo armazenado em uma estrutura que pode facilmente ser pesquisada e trabalhada, apesar de que tais funcionalidades não foram implementadas neste programa, por fugirem do escopo proposto.

Classe Comando

O início do programa é a classe **Comando**, que é responsável por armazenar exatamente uma linha de texto recebida do documento externo, e obter delas certas informações

necessárias para o processamento posterior das informações. Neste processamento é visto se a linha contém "*goto ", "if",* e se é iniciada com um "*L"*.
Caso contenha "*goto ",* significa que essa linha será a última de um vértice, e também que conterá o destino para o qual o fluxo do programa deverá seguir, o que será importante quando se tratar dos **Vertices**.
Se a linha também contém um *"if"*, significa que além de indicar um destino em outra parte do código, também deve dar sequência de forma linear ao programa, sendo necessário representar tal informação.
Por fim, se a linha se inicia com um *"L"*, é entendido que tal linha representa um destino de *"goto",* que geralmente assume a forma de *"L0", "L1", "L2"*, etc. Essa informação significa que a linha em questão é o início de um vértice. No caso de se ter uma convenção de nomenclatura diferente, é necessário alterar essa parte específica do código, adicionando a nova condição de seleção na função *ifDestino()*.

Vertice

A classe **Vertice** é destinada ao armazenamento de comandos, e a tratar as informações obtidas no primeiro processamento das linhas.
O primeiro comando recebido por um novo **"vertice"** é tratado como seu "*lider*", o primeiro comando de um bloco. O "*líder*" é tanto adicionado à sua própria variável, quanto ao bloco de todos os comandos, por também fazer parte das instruções gerais.
Ao receber um comando, o ***"vertice"*** também decide se ele é o último bloco que deverá processar, usando para isso tanto a informação de que ele contém um *"goto",* quanto se ele é o destino de um *"goto"*. É possível que o "*líder*" seja o único comando em um bloco, e portanto essas informações são conferidas para todo comando recebido, desde o primeiro.
É importante notar que se um comando, recebido após o líder ser determinado, for o destino de um desvio, ele não deve ser incluído no presente vértice, mas sim salvo para ser o líder do próximo bloco. Foi usado para esse procedimento uma variável booleana, possibilitando ao programa lidar com essa situação específica.
Ao encontrar um comando que contenha "*goto*", o "**vertice"** determina que ele será a saída, e também separa (split) a linha de texto em duas partes, uma até a palavra "goto ", incluindo um espaço em braco no final, e o restante dela, normalmente um destino iniciado em "L" e seguido por uma integer. Essa segunda parte é armazenada em uma variável própria, e será usada pela classe **Grafo** para determinar as ligações de cada vértice.
É importante clarificar que a classe **Vertice** depende de uma estrutura externa para ser preenchida, recebendo os **comandos** de uma entidade externa e independente, não processando ela mesma nenhum arquivo de texto. Foi implementada dessa forma para melhor se adequar aos padrões de programação orientada a objeto, e para facilitar a organização geral do código.

Grafo

Então, se tem a classe **Grafo**, que armazena os vértices criados até então. Ela contém uma lista de listas de vértices, que é a representação final do grafo. Cada vértice é armazenado na primeira posição de uma lista, e suas ligações, outros vértices, são armazenados nas posições subsequentes.
Além dos vértices de comandos, um **Grafo** também contém um vértice de entrada e um de saída, denominados "Entry" e "Exit". Tais etiquetas facilitam a leitura do grafo final,

facilitando ao usuário seguir o fluxo representado. Foram criadas na forma de vértices, inseridos na estrutura do grafo, para facilitar uma posterior implementação de algoritmos de busca ou de ligação entre grafos, mesmo que tais funcionalidades não tenham sido implementadas neste programa em particular.
O processo de determinar ligações é feito tendo por base os dados já obtidos pelos comandos e pelos vértices. Primeiramente é utilizada a variável booleana "**fLAG"**, presente no vértice, que determina se deve haver uma ligação entre esse vértice e o próximo. Então é utilizado a variável "**ligacoes"**, também do vértice, onde se busca, entre todos os vértices, aqueles cuja as linhas líderes são iniciadas por essa string.

App

Por fim, se tem a classe **App**, que contém o método *main*, os métodos para leitura de arquivos, e a função para preencher os **Comandos, Vertices** e **Grafos.**
O programa lê todos os arquivos contidos na pasta *"/entradas"*, presente no diretório principal do programa, e os processa, imprimindo no terminal um grafo por arquivo.
A função **"gerador"** é responsável pelo processo de ler um arquivo linha por linha, criando **Comandos** a partir delas, passando esses **Comandos** a um **Vertice**, conferindo se este atendeu algum requisito para ser terminado, e uma vez que o **Vertice** está pronto, armazená-lo em um **Grafo.**
É uma função complexa, que realiza diversas funções, e portanto uma candidata a passar por um processo de refatoração, porém devido à constrição do cronograma, tal não foi realizado.
Por fim, a classe **App** imprime os grafos criados. É importante notar que ao se imprimir vários grafos, esses nem sempre são mostrados em ordem.

Amostras de Processamento

Teste 1:
Código de entrada 1:

```
x = 0
y = 1
if x == 0 goto L2
if y == 0 goto L2
goto L1
L1: z = 1
goto L0
L2: z = 0
L0:
```

Grafo teste1.txt:

B1: x = 0
y = 1
if x == 0 goto L2

B2: if y == 0 goto L2

B3: goto L1

B4: L1: z = 1
goto L0

B5: L2: z = 0

Exit: L0:

Grafo:
Entry-> B1,

B1-> B2,B5,

B2-> B3,B5,

B3-> B4,

B4-> Exit,

B5-> Exit,

A partir do GFC construído, quais são os possíveis fluxos de execução do programa?
Entrada -> B1 ->B2 -> B3 -> B4 -> Exit;
Entrada -> B1 ->B2 -> B5 -> Exit;
Entrada -> B1 -> B5 -> Exit;

Quais são os loops no GFC caso existam?
Não Existem loops neste GFC

Teste 2:
Código de Entrada 2:
```
e = 1
t0 = 3 * 5
t1 = 2 / t0
x = t1
t2 = 67 - 233
t3 = t2 / e
a = t3
t4 = 343.5 * 13.15
t5 = 2.456 + t4
c = t5
if a == c goto L2
goto L1
L1: t = 7
goto L0
L2: t = 1
```

L0:

Grafo teste2.txt:

B1: e = 1
*t0 = 3 * 5*
t1 = 2 / t0
x = t1
t2 = 67 - 233
t3 = t2 / e
a = t3
*t4 = 343.5 * 13.15*
t5 = 2.456 + t4
c = t5
if a == c goto L2

B2: goto L1

B3: L1: t = 7
goto L0

B4: L2: t = 1

Exit: L0:

Grafo:
Entry-> B1,

B1-> B2,B4,

B2-> B3,

B3-> Exit,

B4-> Exit,

A partir do GFC construído, quais são os possíveis fluxos de execução do programa?
Entrada -> B1 -> B2 -> B3 -> Exit;
Entrada -> B1 -> B4 -> Exit;

Quais são os loops no GFC caso existam?
Não Existem loops neste GFC

Teste 3:
Código de Entrada 3:
L1: if i > k goto L2
t1 = i + 1

```
i = t1
x[i] = 0
goto L1
L2: x[0] = 0
```

Grafo teste3.txt:

B1: L1: if i > k goto L2

B2: t1 = i + 1
i = t1
x[i] = 0
goto L1

Exit: L2: x[0] = 0

Grafo:
Entry-> B1,

B1-> B2,Exit,

B2-> B1,

A partir do GFC construído, quais são os possíveis fluxos de execução do programa?
Entrada -> B1 -> B2 -> B1 -> Exit
Entrada -> B1 -> Exit;
Quais são os loops no GFC caso existam?
O loop de B1-> B2 -> B1...

Teste 4:
Código de Entrada 4:

```
 s = 0
 i = 0
 k = 2
L0: ifnot i < 10 goto L2
 j = 0
L3: ifnot j < 10 goto L1
 t0 = s + k
 s = t0
 t1 = j + 1
 j = t1
 goto L3
L1: t2 = i + 1
 i = t2
 goto L0
L2:
```

Grafo teste4.txt:

B1: s = 0
i = 0
k = 2

B2: L0: ifnot i < 10 goto L2

B3: j = 0

B4: L3: ifnot j < 10 goto L1

B5: t0 = s + k
s = t0
t1 = j + 1
j = t1
goto L3

B6: L1: t2 = i + 1
i = t2
goto L0

Exit: L2:

Grafo:
Entry-> B1,

B1-> B2,

B2-> B3,Exit,

B3-> B4,

B4-> B5,B6,

B5-> B4,

B6-> B2,

A partir do GFC construído, quais são os possíveis fluxos de execução do programa?
Entrada -> B1 -> B2 -> B3 -> B4 -> B5 -> B4 -> B6 -> B2 -> Exit;
Entrada -> B1 -> B2 -> Exit;
Entrada -> B1 -> B2 -> B3 -> B4 -> B6 -> B2 -> Exit;
Quais são os loops no GFC caso existam?
Um loop é a sequência B4 -> B5 -> B4…
Outro loop é o de B2 -> B3 -> B4 -> B6 ->B2…
E por fim, B2 -> B3 -> B4 -> B5 -> B4 -> B6 -> B2…

Teste 5:
Código de Entrada 5:

```
a = 1
b = 2
c = 3
L0: if c>0 goto L1
goto L2
L1:t0 = a + 1
a = t0
t1 = c - 1
c = t1
goto L0
L2:
```

Grafo teste5.txt:

B1: x = 0
y = 1
t0 = 3 + 2
t1 = t0 - x
x = t1
t2 = x + y
if t2 != 0 goto L1

B2: if y == 0 goto L2

B3: L1: z = 1
goto L0

B4: L2: z = 0

Exit: L0:

Grafo:
Entry-> B1,

B1-> B2,B3,

B2-> B3,B4,

B3-> Exit,

B4-> Exit,

A partir do GFC construído, quais são os possíveis fluxos de execução do programa?
Entrada -> B1 -> B2 -> B3 -> Exit;

Entrada -> B1 -> B2 -> B4 -> Exit;
Entrada -> B1 -> B3 -> Exit;

Quais são os loops no GFC caso existam?
Não Existem loops neste GFC

Conclusão

A principal dificuldade encontrada foi a organização do programa, manter uma estrutura limpa e coerente enquanto se trabalhava com os elementos do problema. Algumas soluções ainda mostram as marcas dessa dificuldade, e é provável que diversas funções sejam redundantes, ou que seja possível simplificar seu uso ou implementação.
Outro desafio foi acompanhar o fluxo de execução do programa em alguns momentos, pois é necessário manter diversos objetos em mente, e saber o que cada um vai acionar dentro do outro. Essa dificuldade traz à luz o problema do sistema de "flags" usado, onde o valor de uma variável booleana é alterado dependendo da situação. Esse sistema torna difícil a compreensão do que o programa está fazendo em determinado momento, e abre brecha para muitas falhas. Um ponto de melhoria significativo deste programa seria a implementação de um sistema mais eficiente e menos complexo, o que facilitaria a manutenção do código.
Uma funcionalidade que não foi implementada, mas que seria interessante possuir é um sistema para arquivar os grafos criados, que nesta implementação são apenas exibidos no console. Também não foi solucionado o pequeno problema na ordem em que esses grafos são mostrados, que nem sempre seguem a ordem alfabética dos arquivos.
Também não foi implementado todas as *exceptions* que podem surgir durante o funcionamento do programa, como por exemplo o caso em que o usuário adiciona uma nova pasta dentro da pasta *entrada*.
Além disso, é importante notar que este programa é muito sensível a variações nos textos de entrada, processando de forma muito rígida cada linha. Assim, um espaço a mais depois de um *"goto"* ou outra pequena mudança do gênero resulta em uma saída inválida ou defeituosa. Para se obter um programa plenamente funcional, é preciso resolver essa fragilidade do sistema, mas tal correção foge do propósito do presente programa.
No mais, todas as funcionalidades propostas foram implementadas, e nenhuma falha crítica foi encontrada durante o processo de testagem, contudo não descartando a existência de alguma em casos mais extremos, que podem não terem sido abordados durante os testes.