Grafos de Dijkstra: Avaliação de Curtos-Circuitos

André L. P. Guedes[1], **Matheus S. Feitosa**[1], **Matheus T. Batista**[1]

[1] Departamento de Informática
Universidade Federal do Paraná (UFPR) – Curitiba, PR – Brasil

{andre, msf21, mtb21}@inf.ufpr.br

Abstract. *In this paper, we revisit Dijkstra graphs and the concept of structured programming. We present a generalization that accommodates short-circuit evaluations via a new class of graphs: short-circuit graphs. We then introduce a algorithm that contracts these short circuits and reduces this class to Dijkstra graphs.*

Resumo. *Neste trabalho, revisitamos os grafos de Dijkstra e o conceito de programação estruturada. Apresentamos uma generalização desses grafos, admitindo avaliações de curtos-circuitos por meio de uma nova classe de grafos: os grafos de curto-circuito. Introduzimos um algoritmo que contrai esses curtos-circuitos e reduz essa nova classe aos grafos de Dijkstra.*

1. Introdução

Os grafos de Dijkstra (DG), introduzidos em [Bento et al. 2019], são uma classe de grafos inerente ao conceito de programação estruturada, e procuram responder uma questão naturalmente levantada quando se pensa no paradigma: dado um programa, identificar se o mesmo é estruturado. Isto é realizado utilizando o grafo de fluxo de controle (CFG) do programa, e bastaria verificar se o mesmo se enquadra como DG.

Utilizando o compilador GCC para gerar CFGs de programas estruturados, foi diagnosticado um problema: avaliações de curtos-circuitos geram CFGs não isomorfos a grafos de Dijkstra. Após uma revisão da bibliografia, nada foi encontrado sobre o tema. Apresentamos no presente trabalho uma generalização dos DG, capaz de representar avaliações de curtos-circuitos. Também propomos um algoritmo que reduz grafos dessa nova classe a grafos de Dijkstra equivalentes.

Na próxima seção, começamos introduzindo conceitos da teoria dos grafos utilizados de maneira recorrente durante o trabalho. Na Seção 3 apresentamos de maneira mais rigorosa as definições de grafos de Dijkstra e grafos de fluxo de controle. Seção 4 aborda com mais detalhes o problema, além de caracterizar os grafos de curto-circuito. Na Seção 5, apresentamos o algoritmo e sua prova de corretude. A Seção 6 tece críticas sobre o presente trabalho e apresenta os próximos tópicos a serem investigados.

2. Preliminares

Neste texto, exceto quando especificado, todos os grafos são finitos e direcionados. Para um grafo G, denotaremos seus conjuntos de vértices e arcos (pares ordenados de vértices)

Realizado com apoio do CNPq (processo: 420079/2021-1) e bolsas de IC da UFPR/TN.

por $V(G)$ e $A(G)$. Para um arco $a = (x, y)$, x é chamado de cauda e y de cabeça. Para um vértice v, denotaremos por $N_G^+(v)$ e $N_G^-(v)$ seus vizinhos de saída e entrada em G. Diremos que v alcança w se existe caminho de v para w em G. Uma fonte de G é um vértice que alcança todos os demais, enquanto que um sumidouro não alcança ninguém além de si mesmo. O grau de saída de v será denotado por $deg^+(v)$, enquanto $deg^-(v)$ denotará o grau de entrada. Por DFS, entende-se uma busca em profundidade em um grafo G, começando por uma de suas fontes. Por Goto l, entende-se a instrução que desvia o fluxo do programa para etiqueta l.

3. Definição dos Grafos de Dijkstra

Um grafo de fluxo de controle é definido como um conjunto de basic blocks, que, em resumo, são sequências lineares de instruções, iniciando em um bloco de entrada e seguindo até um bloco de saída [Allen 1970]. Com isso, os autores em [Bento et al. 2019] definem a classe dos DGs como uma subclasse dos CFGs, em específico, os CFGs que advêm de programas estruturados. Aqui, entende-se por um programa estruturado aquele que segue o paradigma de programação estruturada, como definido por Dijkstra em [Dijkstra 1968, Dahl et al. 1972]. Para o presente trabalho, é importante se atentar que todos os DGs são direcionados, finitos e constituídos de subgrafos chamados "*statement graphs*". Cada *statement graph* representa uma sequência de instruções, sendo cada uma delas (a) trivial graph; (b) sequence graph; (c) if graph; (d) if-then-else graph; (e) p-case graph, $p \geq 3$; (f) while graph; (g) repeat graph. Seja H um *statement graph*, $s(H)$ e $t(H)$ retornam respectivamente fonte s e sumidouro t conforme descrito na Figura 1. Todo *statement graph* é composto por vértices expansíveis (rotulados por X) ou regulares (rotulados por R).

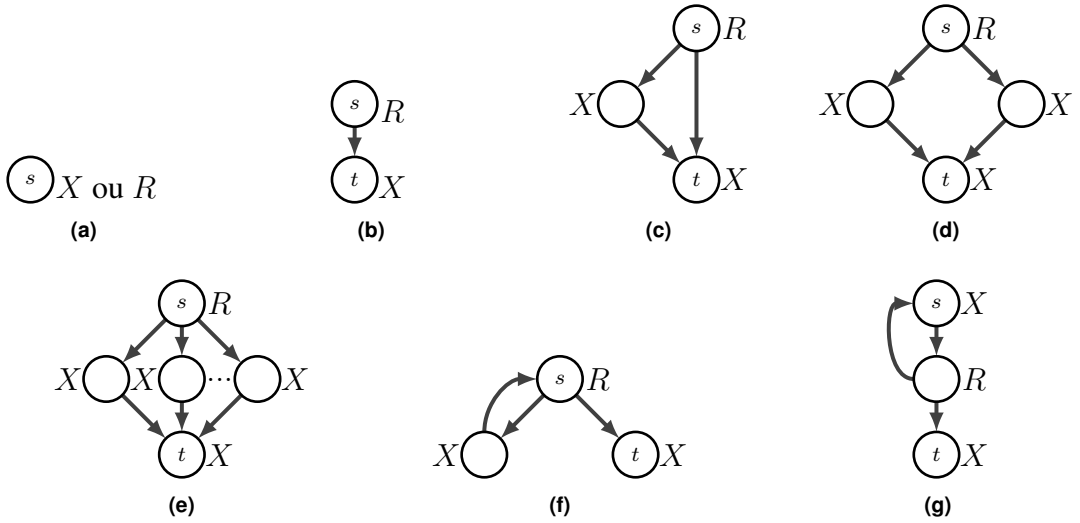


Figura 1. *Statement graphs*, retirado de [Bento et al. 2019]

Um DG é construído a partir de expansões sobre grafos, começando por um *trivial graph*. Em cada etapa da expansão de G em G', escolhe-se um vértice $v \in V(G)$ com rótulo X e troca-se o mesmo por um *statement graph* H, de maneira que $V(G') = V(G) \setminus v \cup V(H)$, $N_{G'}^-(s(H)) = N_G^-(v)$ e $N_{G'}^+(t(H)) = N_G^+(v)$. As demais vizinhanças são preservadas.

4. Problema

Compilando com GCC, programas em C estruturados não geram CFGs equivalentes aos DGs. Em DGs, blocos condicionais e de repetição (Figuras 1c, 1d, 1f e 1g) produzem um único vértice; porém, operadores de curto-circuito (`AND`, `OR`) fazem o GCC gerar múltiplos vértices.[1] Em [Bento et al. 2019], argumenta-se que subexpressões booleanas isoladas não determinam o fluxo nem criam ramificações intermediárias — a expressão deve resultar num único valor (verdadeiro ou falso). A Figura 2 ilustra: DG (Figura 2a) versus CFG (Figura 2b).

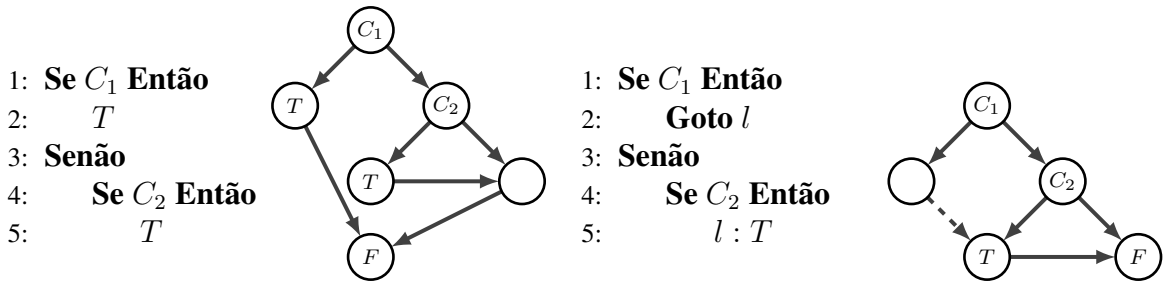
(a) DG *while* ou *repeat* **(b) CFG Com c.c**

Figura 2. Exemplos de CFGs, respectivamente, sem e com c.c

4.1. Simulando Curtos-Circuitos

O objetivo desta seção é mostrar que o comportamento de um curto-circuito nem sempre pode ser simulado em um DG. Sem perda de generalidade, utilizaremos como exemplo o *if graph*.

Basta olhar para o caso `OR`, onde se faz necessário repetição de código (Figura 3a) ou uso de Goto (Figura 3b).



(a) Repetição de Código **(b) Uso de Goto**

Figura 3. c.c `OR` e o *statement if*

5. Solução Proposta

Para apresentar o algoritmo, inicialmente, definiremos alguns conceitos.

5.1. Grafos de Curto-Circuito

Vértice condicional: um vértice v é condicional se possui $deg^+(v) = 2$.

[1]Note que não há nada errado com o GNU Compiler Collection, realmente o fluxo de controle vai ser alterado se houver avaliações de curtos-circuitos.

Vértices condicionais representam as ramificações derivadas de blocos condicionais e de repetição dentro de um CFG. Podemos interpretar cada um de seus arcos de saída como o que ocorre com o fluxo caso a expressão ali seja verdadeira ou falsa. Definimos um grafo de curto-circuito H como um subgrafo acíclico direcionado com apenas uma fonte. Possui apenas 2 vértices condicionais, vizinhos, sendo que esses compartilham 1 vizinho de saída em comum. A Figura 4 representa um grafo de curto-circuito.

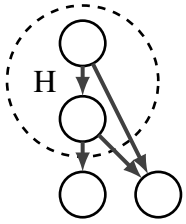


Figura 4. Grafo de Curto-Circuito H

5.1.1. Expansões de Curto-Circuito

Em primeiro momento, definiremos uma versão ponderada dos *statement graphs*, dada pela função peso $w : A(G) \to \{0, 1, \epsilon\}$. Todos os vértices condicionais c terão o peso de seus arcos conforme Figura 5. Os demais arcos recebem peso ϵ. Se um arco tem peso 1, denotaremos sua cabeça por T; se tem peso 0, denotaremos a cabeça por F.

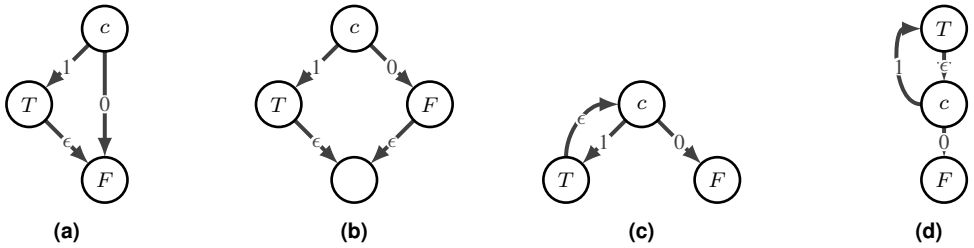
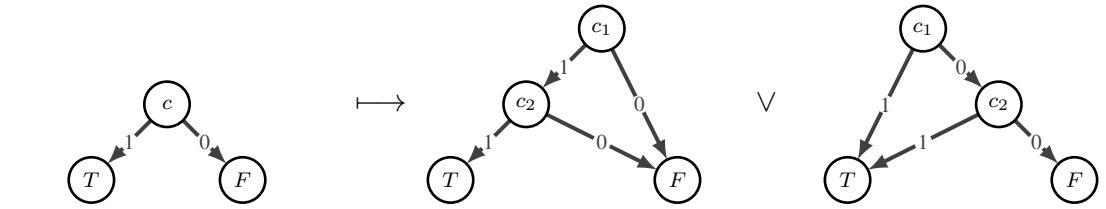


Figura 5. *Statement graphs* ponderados

Para obtenção de um grafo de curto-circuito, comece criando um grafo de Dijkstra G, como definido na Seção 3. Em seguida, seja $c \in V(G)$ vértice condicional. G poderá ser expandido em G' de maneira que: $V(G') = V(G) \setminus c \cup \{c_1, c_2\}$, $N^-_{G'}(c_1) = N^-_G(c)$, $N^+_G(c_2) = N^+_G(c)$, $w(c_2, T) = 1$ e $w(c_2, F) = 0$. A respeito das vizinhanças e pesos de c_1 e c_2, uma das seguintes opções precisará ser escolhida, simbolizando a escolha de um operador lógico `AND` ou `OR`.

$$N^+_{G'}(c_1) = \{c_2, F\} \ \wedge \ w(c_1, c_2) = 1 \ \wedge \ w(c_1, F) = 0 \qquad (\texttt{AND})$$
$$N^+_{G'}(c_1) = \{c_2, T\} \ \wedge \ w(c_1, c_2) = 0 \ \wedge \ w(c_1, T) = 1 \qquad (\texttt{OR})$$

Exemplos de expansão são ilustrados na Figura 6. O processo de expansão pode ser continuado em G', seguindo as mesmas regras descritas anteriormente. Ao final de todas as expansões, podemos desconsiderar os rótulos dos vértices e os pesos das arestas, voltando a versão direcionada do grafo.



(a) Vértice condicional rotulado **(b) Expansão And** **(c) Expansão OR**

Figura 6. Exemplos de expansão em vértices condicionais

5.2. Reconhecimento e Contração de Curtos-Circuitos

Com tudo já estabelecido, o Algoritmo 1 recebe um CFG e contrai curtos-circuitos existentes.

Algoritmo 1 Contração de Curtos-Circuitos(G)

Exija G, Grafo de Fluxo de Controle
1: E_C, conjunto das arestas de ciclo de uma DFS começando na fonte de G
2: $v_1, \ldots, v_n$ Uma ordenação topológica de $G - E_C$
3: **Para** $i \leftarrow n-1$ até 1 **Faça**
4: **Se** $v_i \wedge v_{i+1}$ forem condicionais **Então**
5: **Se** $(v_{i+1} \in N^+(v_i)) \wedge |N^+(v_i) \bigcap N^+(v_{i+1})| = 1 \wedge (deg^-(v_{i+1}) = 1)$ **Então**
6: $G \leftarrow G \downarrow \{v_i, v_{i+1}\}$
7: $i \leftarrow i + 1$
8: **Devolva** G

Prova de corretude: Uma expressão booleana que possui operadores pode ser descrita como $A \star B$, onde $\star \in \{\wedge, \vee\}$ e A, B são expressões booleanas. A partir disso, é possível construir uma árvore de forma recursiva, representando a expressão. O algoritmo realiza uma travessia dessa árvore em ordem topológica inversa, ou seja, das folhas até a raiz. Como o algoritmo contrai sempre as folhas da árvore, em algum ponto da execução, encontraremos uma folha da forma $x_1 \star x_2$, em que x_1 e x_2 são literais pertencentes a alguma das expressões A ou B. A partir desse momento, iniciam-se as contrações. Observe que, a partir do exemplo anterior, $x_1 \star x_2$ torna-se uma folha na nova árvore, podendo ser contraída com outra folha ou expressão, seja ela literal ou composta.

A opção por contrair na ordem topológica inversa não é aleatória: ao contrair duas subexpressões, o algoritmo garante que cada uma já está reduzida a um único vértice, pois a passagem nas subárvores correspondentes contraiu todas as expressões internas.

6. Fechamento e próximos passos

Gostaríamos de tornar mais rigorosas as notações, comentários e provas aqui apresentados, em especial a Seção 4.1. Queremos implementar o Algoritmo 1 integrado aos algoritmos apresentados em [Bento et al. 2019].

Por fim, pretendemos estudar a associação dos nós dos grafos com o conteúdo dos códigos, como comandos e expressões booleanas, com o objetivo de efetivamente representar códigos; posteriormente investigando outras utilidades dessa representação, como, por exemplo, detecção de código plagiado.

Referências

[Allen 1970] Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7):1–19.

[Bento et al. 2019] Bento, L. M., Boccardo, D. R., Machado, R. C., Miyazawa, F. K., Pereira de Sá, V. G., and Szwarcfiter, J. L. (2019). Dijkstra graphs. *Discrete Applied Mathematics*, 261:52–62. GO X Meeting, Rigi Kaltbad (CH), July 10–14, 2016.

[Dahl et al. 1972] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*. Academic Press London and New York.

[Dijkstra 1968] Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148.