Solução para Ordenação Topológica em Grafos Direcionados Acíclicos

Lucas Correia de Araujo, Rodrigo Saviam Soffner
GRR20206150, GRR20205092
Acadêmicos em Ciência da Computação
CI1065 - Algoritmos e Teoria dos Grafos
Universidade Federal do Paraná – UFPR

I. Introdução

Nesse relatório consta os detalhes da resolução do trabalho proposto pelo professor Murilo V. G. da Silva, na disciplina de Algoritmos e Teoria dos Grafos (CI1065), com foco no desenvolvimento de uma solução para o problema da Ordenação Topológica de grafos direcionados acíclicos (DAG).

Para contextualizar o projeto realizado, inicia-se com uma breve introdução ao problema apresentado, sendo em seguida exposto a solução utilizada para a resolução da questão, bem como dos detalhes que a tornam eficiente no objetivo citado. Ao fim, consta a explicação da implementação do algoritmo com uso da solução que atinge o tema proposto, além das informações relacionadas ao programa.

II. Problema

Como ponto principal a ser solucionado, foi proposto o problema da Ordenação Topológica de grafos direcionados acíclicos. Dentro da teoria estudada com grafos, pode-se obter uma solução desse problema por meio de uma Busca em Profundidade (DFS), fazendo a leitura inversa do pós-ordem gerado pelo algoritmo de busca.

Entretanto, para esse projeto, foi escolhido outro algoritmo sendo ele o algoritmo de Kahn. A motivação da escolha leva em conta a elegância, a simplicidade de implementação e a objetividade do algoritmo para testar se o grafo é acíclico e também por ter sido um algoritmo especialmente criado para resolver a Ordenação Topológica.

III. Algoritmo de Kahn

Datado dos anos 60, o algoritmo de Kahn possui como princípio determinar a cada instante os vértices que são fonte e inserir na solução. A cada vértice inserido na solução, todos seus arcos correspondentes são removidos do grafo.

A seguir consta o algoritmo proposto por Kahn, que parte do princípio que um grafo direcionado acíclico, um DAG, sempre possui uma fonte e um sumidouro e caso um vértice fonte seja removido o grafo resultante continuara sendo um DAG. [1]

Algorithm 1 Kahn

$L \leftarrow \emptyset$
$S \leftarrow$ vértices sem arcos de entrada
while $S \neq \emptyset$ **do**
  retire v de S
  acrescente v em L
  for todos os arco (v, w) **do**
    retire (v, w) de A
    if w não possui arcos de entrada **then**
      acrescente w em S
    end if
  end for
end while
if $A \neq \emptyset$ **then**
  return Erro
else
  return L
end if

IV. Implementação

A solução foi desenvolvida utilizando a linguagem C++, por ser robusta e com alto desempenho. Para tratamento e armazenamento dos grafos, utiliza-se a biblioteca *GraphViz*, recomendada na especificação do projeto, pela sua usabilidade para leitura de arquivos com formato .dot, juntamente com sua estruturação interna para grafos. [2]

A leitura do grafo é feita pela entrada padrão, stdin, da seguinte maneira:

```
1    Agraph_t * g = agread(stdin, NULL);
```

Listing 1. Captura de grafo .dot

No Listing 1, o ponteiro 'g' recebe o endereço da representação do grafo fornecido pela biblioteca, que será argumento para a função topologicalSort.

Dentro da função topologicalSort, será realizada a ordenação topológica do grafo utilizando o algoritmo de Kahn, primeiramente é inicializado um vetor(vector) para armazenar os graus de entrada dos vértices no grafo. A partir da função 'agnnodes', fornecida pela biblioteca, é possível obter o número de vértices no grafo.

```
1    int vertices_n = agnnodes(g);
2    vector<int> in_degree(vertices_n, 0);
```

Listing 2. Captura do número de vértices do grafo

Também será necessário o uso de uma fila para executar o algoritmo de Kahn, então a estanciamos com tipo 'Agnode_t *', pois teremos que utilizar la para armazenar uma ordem de vértices que iremos processar futuramente, isso é possível guardando os ponteiros dos vértices dentro da nossa estrutura de dados que são do tipo 'Agnode_t'. Como estamos utilizando a linguagem C++ decidimos utilizar as estruturas de dados 'vector' e 'queue' oferecidas pela biblioteca standart do C++.

```
1    queue<Agnode_t *> q;
```

Listing 3. Instanciação de uma fila de vértices

No primeiro processamento, é realizado o calculo dos graus de entrada de todos os vértices, utilizando um *loop* com os parâmetros e funções fornecidas pela biblioteca, de maneira a iterar sobre todos os vértices do grafo e armazenar seus graus de entrada.

Os graus de entrada são acessados pela função 'agdegree(g, n, true, false)', e armazenados na posição do vetor a partir da propriedade 'AGSEQ(n)'. Essa propriedade retorna um índice único para cada vértice, gerado pela biblioteca durante a leitura do grafo. O intervalo desse índice é [1..n], permitindo o uso como indexador para o vetor. Além disso os vértices fontes serão enfileirados em 'q'.

```
    for (Agnode_t *n = agfstnode(g); n; n = agnxtnode(g, n)) {
        in_degree[AGSEQ(n - 1)] = agdegree(g, n, true, false);
        if (in_degree[AGSEQ(n - 1)] == 0)
            q.push(n);
    }
```

Listing 4. Calculo dos graus de entrada dos vértices e enfilamento de vertices fontes

O loop seguinte ira iterar sobre a fila 'q', onde 'u' será um vértice fonte em relação ao grafo ainda não processado. será decrementado o grau de entrada da vizinhança de saída de 'u' e todos vértices fontes formados serão enfileirados em 'q'. Estas verificações e operações são feitas no vetor 'in_degree' que corresponde ao grau de entrada de todos os vértices do grafo. No final o contador 'visited_vertices' será incrementado para a contagem de vértices processados.

```
    while (!q.empty()) {
        u = q.front();
        q.pop();
        top_order.push_back(agnameof(u));

        for (e = agfstout(g, u); e; e = agnxtout(g, e)) {
            v = aghead(e);
            if (--in_degree[AGSEQ(v) - 1] == 0)
                q.push(v);
        }

        visited_vertices++;
    }
```

Listing 5. Processamento principal do algoritmo

Após o processamento dos vértices é feito uma verificação de um condicional para testar se o grafo de entrada é um DAG. Nessa verificação, é avaliado se todos os vértices foram processados, sendo impresso na saída padrão (stdout) a ordenação topológica do grafo em caso verdadeiro, ou uma mensagem de erro caso contrário.

```
    if (visited_vertices != vertices_n) {
        cout << "Error: Graph contains a cycle" << endl;
        return;
    }

```

```
6       size_t i = 0;
7       for (; i < top_order.size() - 1; i++)
8           cout << top_order[i] << ", ";
9       cout << top_order[i];
```

Listing 6. Detecção de grafos cíclicos

V. Erros

Em caso de erros, ocorrerá do algoritmo retornar uma mensagem de erro para a saída padrão de erro (stderr) descrevendo o motivo da falha. Os seguintes erros podem ocorrer:

- ”Error: Could not read graph.”, Caso a biblioteca Graphviz não consiga ler o grafo de entrada. Outra mensagem de erro também será acionada pela biblioteca apontando o motivo do grafo não poder ter sido lido.
- ”Error: Graph is not directed.”, Essa mensagem de erro será retornada caso o grafo de entrada não seja direcionado. Essa informação é provida pela função ’agisdirected(Agraph_t *g)’ da biblioteca Graphviz.
- ”Error: Graph contains a cycle.”, Caso o grafo de entrada possuir um ciclo, detectado dentro do algoritmo de ordenação caso não todos os vértices tenham sido visitados.

VI. Testes

Em seguida será apresentado alguns grafos de exemplo, que serviram de entrada para o algoritmo desenvolvido. Abaixo está contido as representações dos grafos em formato .dot e seus desenhos e as respectivas saídas do algoritmo.

A. *Grafo Direcionado Acíclico (DAG)*

Algoritmo com a seguinte entrada .dot:

```
digraph D {
  0
  1
  2
  3
  4
  5

  2 -> 3
  3 -> 1
  4 -> 0
  4 -> 1
  5 -> 0
  5 -> 2
}
```

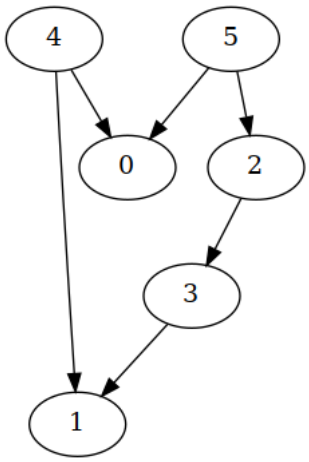

Figura 1. Exemplo de DAG

Saída do algoritmo: 4, 5, 0, 2, 3, 1

Avaliando os dados gerados, nota-se que a saída do algoritmo é realmente uma ordenação topológica do grafo.

```
digraph D {
  0
  1
  2
  3
  4
  5

  2 -> 3
  3 -> 1
  4 -> 0
  4 -> 1
  5 -> 0
  0 -> 2
  2 -> 5
}
```

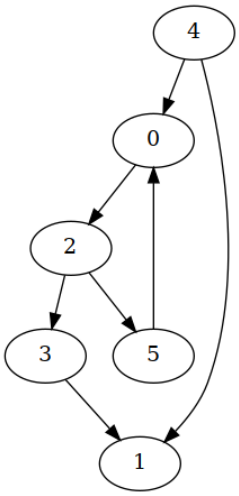

Figura 2. Exemplo de DCG

B. Grafo Direcionado Cíclico (DCG)

Algoritmo com a seguinte entrada .dot:

Saída do algoritmo: Error: Graph contains a cycle.

Avaliando os dados gerados, nota-se que o grafo possui o ciclo direcionado com os vértices 0, 2, 5.

C. Grafo Não-Direcionado

Algoritmo com a seguinte entrada .dot:

```
graph D {
  0
  1
  2
  3
  4
  5

  2 -- 3
  3 -- 1
  4 -- 0
  4 -- 1
  5 -- 0
  5 -- 2
}
```

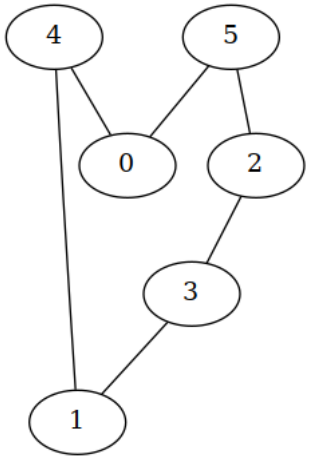

Figura 3. Exemplo grafo não direcionado

Saída do algoritmo: Error: Graph is not directed.

Avaliando os dados gerados, nota-se que o grafo de entrada com formato .dot realmente se trata de um grafo não direcionado.

VII. Anotações

Foram utilizadas as referências fornecidas na disciplina para explicação da parte teórica da solução [3], além das seguintes para desenvolvimento do conteúdo no relatório. [4] [5]

Referências

[1] "Teoria dos grafos - aula 15," UFPO - Prof. Marco Antonio, 10 2019. [Online]. Available: $www.decom.ufop.br/marco/site_media/uploads/bcc204/15_aula_15.pdf$

[2] "Kahn's algorithm for topological sorting," Geeks For Geeks, 06 2023. [Online]. Available: $www.geeksforgeeks.org/topological-sorting-indegree-based-solution/$

[3] "Teoria dos grafos - aula 26," UFPR - Prof. Murilo Silva, 01 2023. [Online]. Available: $www.inf.ufpr.br/murilo/grafos/26.pdf$

[4] "Topological sorting," Algorithims for Competitive Programming, 01 2023. [Online]. Available: $cp-algorithms.com/graph/topological-sort.html$

[5] "Material de teoria dos grafos," UFRN, 02 2015. [Online]. Available: $www.dimap.ufrn.br/\ prolo/Disciplinas/13I/DIM0111.0-AEDII/materiais/grafos/$