Remoção de Transitividade

Daniel Valadares de Souza Felixl[1], Felipe Augusto Maciel Constantinol[1], Gustavo Silvestre Almeida Conceiçãol[1], Larissa Valadares Silqueiral[1]

[1]Pontifícia Universidade Católica de Minas Gerais

1. Introdução

A remoção de transitividade em grafos direcionados é uma operação que busca simplificar a estrutura do grafo, removendo ciclos e estabelecendo um único caminho entre quaisquer pares de vértices alcançáveis. Essa técnica é amplamente utilizada em diversas áreas para a resolução de problemas complexos.

Diante desse contexto, o objetivo deste relatório é apresentar uma solução em C++ para a redução de transitividade em grafos direcionados. O algoritmo implementado visa eliminar ciclos e loops no grafo, resultando em um novo grafo no qual existe apenas um caminho entre todos os pares de vértices alcançáveis.

A solução proposta visa simplificar a estrutura do grafo, estabelecendo uma relação clara de atingibilidade entre os vértices, o que é essencial para a análise de caminhos únicos e a resolução eficiente do problema.

2. Algoritmo

O código implementa a classe *Digraph* para representar o grafo direcionado e realizar as operações necessárias. A classe possui métodos para criar o grafo, gerar seu grafo transposto e calcular o fecho transitivo de cada vértice do grafo. Além desses, há métodos auxiliares que permitem visualizar essas etapas, facilitando a compreensão e depuração do processo de redução transitiva.

O construtor da classe *Digraph* é responsável por inicializar um objeto da classe. Ele recebe como parâmetro o número de vértices do grafo e utiliza esse valor para configurar a estrutura interna do objeto. Essa estrutura, por sua vez, é uma matriz de adjacência (cuja escolha é justificada na seção 4 deste artigo) que inicializa todos os seus elementos como falso. O código que implementa esse construtor pode ser visualizado na Figura 1.

```
// Constructor
Digraph(int numVertices) {
    this->V = numVertices; // definindo numero de vertices
    this->E = 0; // definindo numero de arestas
    adjMatrix = new bool*[V]; // Alocar espaço para primeira coluna
    for (int u = 0; u < V; u++)
    {
        adjMatrix[u] = new bool[V]; // Alocar espaço para cada linha
        for (int v = 0; v < V; v++)
            adjMatrix[u][v] = false; // Iniciar todos os vertices como falso
    }
}
```

Figura 1. Construtor da classe Digraph

O método *removeCycleUtil*, que pode ser visualizado na Figura 2, é uma função auxiliar que utiliza a técnica de busca em profundidade (DFS) para identificar e remover

ciclos no grafo. Esse método percorre os vértices, visitando todos os seus vizinhos. Se um vértice já foi visitado e está presente na pilha de recursão, isso indica a existência de um ciclo no grafo. Nesse caso, o ciclo é removido remapeando as arestas apropriadas. Ele é chamado pelo método *removeCycle*, que itera sobre todos os vértices do grafo, garantindo que todos os possíveis ciclos sejam eliminados.

```
// Metodo auxiliar para removeCycle()
bool removeCycleUtil(int u, bool visited[], bool* recStack, bool* flag) {
    if (!visited[u]) { // testar se ja visitado
        // Definir Vertice como visitado e marca como parte do stack
        visited[u] = true;
        recStack[u] = true;
        for (int v = 0; v < V; v++) {
            v = v == u ? v + 1 : v; // Linha para nao remover loops
            if (adjMatrix[u][v]) {
                if (!visited[v]
                && removeCycleUtil(v, visited, recStack, flag))
                {
                    if (!*flag) {
                        removeEdge(u, v);
                        *flag = true;
                    }
                    return true;
                }
                else if (recStack[v])
                {
                    if (!*flag) {
                        removeEdge(u, v);
                        *flag = true;
                    }
                    return true;
                }
            }
        }
    }
    // Remove vertice da pilha caso ja tenha sido visitado
    recStack[u] = false;
    return false;
}
```

Figura 2. Remoção de Ciclo

O método *createTraverse* é responsável por criar o grafo transposto, invertendo todas as arestas do grafo original. O método percorre os vértices do grafo original e adiciona as arestas de forma invertida no grafo transposto, para cada par de vértices conectados. O código que implementa esse método pode ser visualizado na Figura 3, abaixo. É válido mencionar que esse método permite analisar e avaliar as relações de alcançabilidade entre os vértices do grafo.

```
// Criar Grafo Transposto
Digraph createTraverse () {
    Digraph traverse(V); // Iniciar grafo
    for (int u = 0; u < V; u++)
        for (int v = 0; v < V; v++)
            if (adjMatrix[u][v])
                traverse.addEdge(v, u); // Adcionar arestas
    return traverse; // Retorna grafo transposto
}
```

Figura 3. Criação de Grafo Transposto

Por fim, tem-se o método *transitiveClosure*, que é responsável por criar um novo grafo que representa a relação de transitividade do grafo atual. Ele utiliza o algoritmo de fecho transitivo para determinar todas as arestas necessárias para garantir a transitividade do grafo. Esse algoritmo, que pose ser observado na Figura 4, é fundamental para obter um grafo com apenas um caminho entre quaisquer pares de vértices, eliminando a redundância de caminhos e simplificando a estrutura do grafo.

```
// Criar Grafo com realacao de transitividade
Digraph transitiveClosure () {
    Digraph closure(V); // Iniciar grafo
    for (int u = 0; u < V; u++) // Colocar arestas ligadas de u pra v
        for (int v = 0; v < V; v++)
            if(adjMatrix[u][v])
                closure.addEdge(u, v);
    for (int k = 0; k < V; k++)
        for (int u = 0; u < V; u++)
            for (int v = 0; v < V; v++)
                if (adjMatrix[u][v] || (adjMatrix[u][k] && adjMatrix[k][v]) || u == v)
                    closure.addEdge(u, v);
    return closure;
}
```

Figura 4. Relação de Transitividade

A aplicação sequencial desses métodos, conforme descrito na próxima subseção, oferece uma abordagem sistemática para alcançar o objetivo da redução transitiva de forma eficaz e confiável.

2.1. Redução da Transitividade

Adotando a abordagem delineada no artigo "*The transitive reduction of a directed graph*", o algoritmo *findTransitiveReduction()* foi desenvolvido para alcançar a redução transitiva de um grafo direcionado. Para uma visualização direta da implementação do algoritmo, a Figura 5 apresenta o código correspondente. Os passos descritos abaixo descrevem o procedimento realizado.

1. Encontrar o grafo acíclico equivalente de G
   Inicialmente, é criado um novo grafo G1 que é uma cópia do grafo original G. Para esse novo grafo, é chamada a função *removeCycles()*, que faz com que G1 se torne um grafo acíclico.

2. Remover loops de G1
   Nesta etapa, a função *removeLoops()* é chamada no grafo G1 para remover quaisquer loops presentes no grafo.

3. Calcular M1 e M2, e M3 = M1M2
   No passo 3, é criado um novo grafo G^T2 que representa o grafo transposto de G1. Isso é feito chamando a função *createTraverse()* em G1, que retorna um grafo que possui as arestas direcionadas invertidas em relação a G1. Em seguida, é calculado o produto matricial entre as matrizes de incidência G1 e G^T2, gerando uma nova matriz M3.

4. Criar G3 a partir de M3
   Na quarta etapa, é criado um novo grafo G3 com base na matriz de incidência M3. O novo grafo G3 terá uma aresta entre dois vértices se e somente se os vértices correspondentes em G1 e G^T2 possuírem uma aresta na matriz M3. Isso é feito percorrendo as matrizes de adjacência de G1 e G^T2 e atribuindo o valor true a G3 apenas se o valor correspondente em M3 for verdadeiro.

5. Calcular Gt1 = G1 - G3
   Neste passo, é criado um novo grafo G^T1 a partir do grafo G1 com a remoção das arestas presentes em G3. Isso é feito percorrendo as matrizes de adjacência de G1 e G3 e atribuindo o valor de G1 a G^T1 se o valor correspondente em G3 for falso, caso contrário, atribui-se o valor false.

6. Criar GT a partir de G^T1
   Finalmente, o grafo GT é definido como uma expansão cíclica canônica de G^T1. A expansão cíclica canônica de um grafo é um processo pelo qual as arestas adicionais são inseridas no grafo para formar um ciclo que passa por todos os vértices.

```
Digraph findTransitiveReduction() {
    Digraph g1(V); // Grafo Aciclico do Original

    // Copiar grafo original para G1
    for (int u = 0; u < V; u++) {
        for (int v = 0; v < V; v++)
            g1.adjMatrix[u][v] = adjMatrix[u][v]; // Copiar valor de cada aresta
    }

    g1.removeLoops(); // Remover loops de g1   lico

    g1.removeCycles(); // Transforma G1 em aciclico

    // O objeto G1 representa M1, ja M2 sera representada por gt2

    Digraph gt2 = g1.createTraverse(); // Pegar grafo com relacao de transposto de g1

    // Computar M3 como M1 uniao com M2
    Digraph m3(V);
    for (int u = 0; u < V; u++) {
        for (int v = 0; v < V; v++)
            m3.adjMatrix[u][v] = g1.adjMatrix[u][v] && gt2.adjMatrix[u][v];
    }

    // Computar Gt1 como de g1 com a remocao das arestas de m3
    Digraph gt1(V);
    for (int u = 0; u < V; u++) {
        for (int v = 0; v < V; v++)
            gt1.adjMatrix[u][v] = m3.adjMatrix[u][v] ? false : g1.adjMatrix[u][v];
    }

    // Definir Gt como a expansao ciclica canonica de gt1
    Digraph gt = gt1;

    // Retornar grafo com transitividade reduzida
    return gt;
}
```

Figura 5. Remoção de Transitividade

3. Testes Realizados

Para testar a implementação do algoritmo, foram criados casos de teste representativos que incluem diferentes tipos de grafos, como grafos com e sem ciclos, e também variaram o tamanho dos grafos. Os valores da matriz de adjacência foram inseridos manualmente para cada caso de teste.

3.1. Teste 1

O primeiro teste consistiu em um grafo com alguns ciclos e um loop. A figura abaixo mostra o grafo criado e a matriz de adjacência correspondente:

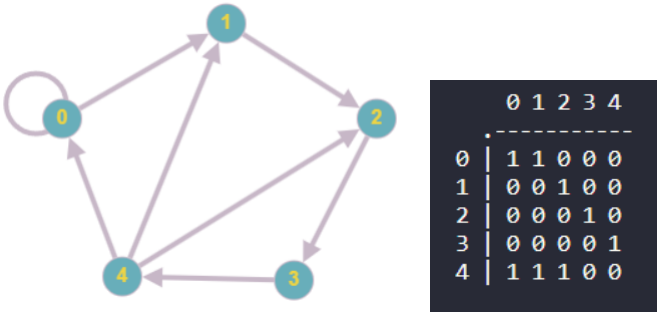

Figura 6. Teste 1 - Grafo de Entrada

Após a criação do grafo de teste, a função de redução de transitividade foi chamada. A matriz de adjacência resultante após a aplicação do algoritmo é exibida na figura abaixo:

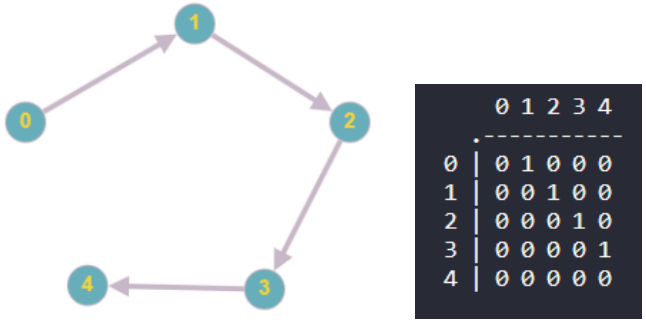

Figura 7. Teste 1 - Resultado

Foram verificados os seguintes pontos em relação ao resultado:

- O algoritmo removeu com sucesso todos os ciclos e loops presentes no grafo.
- A estrutura da matriz de adjacência foi corretamente atualizada para refletir as alterações no grafo.
- Os resultados alcançados foram satisfatórios e estão de acordo com os objetivos propostos no artigo base.

3.2. Teste 2

O segundo teste foi realizado em outro grafo para garantir que o resultado obtido no teste anterior não fosse específico apenas para aquele caso. A figura abaixo mostra o grafo criado.

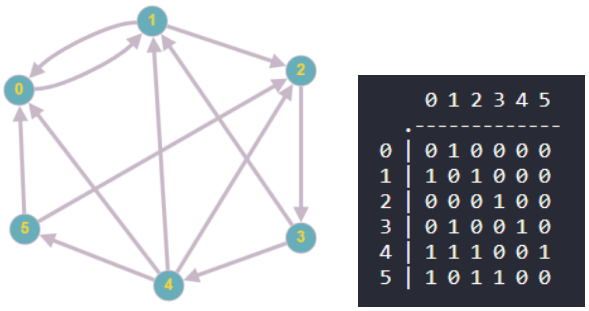

Figura 8. Teste 2 - Grafo de Entrada

A seguinte figura mostra o grafo resultante, juntamente com sua matriz de adjacência.

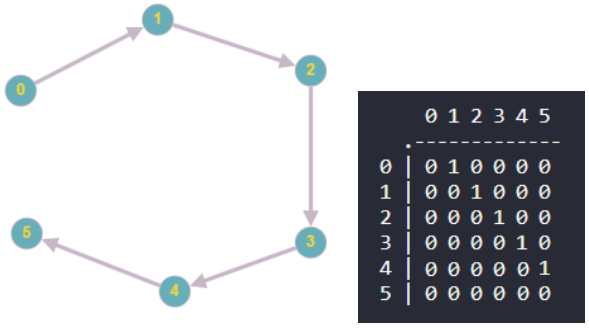

Figura 9. Teste 2 - Resultado

Mais uma vez, o algoritmo removeu com sucesso todos os ciclos presentes no grafo, e a matriz de adjacência foi atualizada corretamente. Os resultados obtidos neste teste também atenderam aos objetivos propostos.

Dessa forma, nos dois casos de teste realizados, o algoritmo demonstrou sua eficácia na remoção de ciclos e loops nos grafos, com a devida atualização da matriz de adjacência. Os resultados obtidos foram consistentes e satisfatórios.

4. Estrutura de Dados

A escolha da matriz de adjacência como estrutura de dados para representar o grafo direcionado foi baseada em suas vantagens distintas. Essa estrutura permite um acesso rápido à informação de adjacência, pois é possível verificar a existência de uma aresta entre dois vértices em tempo constante, simplesmente acessando a posição correspondente na matriz. Além disso, a matriz de adjacência é eficiente para grafos esparsos, ou seja, quando o número de arestas é relativamente pequeno em comparação com o número total de vértices.

5. Conclusão

A implementação do algoritmo de redução de transitividade em um grafo direcionado demonstrou resultados promissores e uma abordagem eficaz para lidar com a transitividade. Os algoritmos de remoção de ciclos, criação do grafo transposto e cálculo do fecho transitivo foram aplicados com sucesso, resultando em grafos reduzidos e otimizados.

Além disso, foi realizado um teste adicional em um grafo não direcionado. Embora a teoria sugira que o passo a passo dos algoritmos seja compatível com grafos não direcionados, foi constatado que nossos algoritmos não são capazes de resolver o problema para esse tipo de grafo. Isso ocorre devido ao método de remoção de ciclos baseado no destino (dst), que não leva em consideração a direção das arestas. Assim, é necessário realizar adaptações nos algoritmos ou buscar outras abordagens para resolver o problema em grafos não direcionados.