Análise dos Algoritmos Naive e Warshall para Encontrar Bases e Antibases com Fecho Transitivo

Daniel de Rezende Leão, Caio Elias Rodrigues Araujo, Victor de Souza Friche Passos

May 15, 2023

Link para o Repositório no GitHub

1 Introdução

Neste trabalho, apresentamos a análise dos algoritmos Naive e Warshall para encontrar bases e antibases com fecho transitivo em um grafo. A motivação para este estudo foi o Trabalho 1 da disciplina de Teoria dos Grafos e Computabilidade, ministrada pelo professor Zenilton Kleber Gonçalves do Patrocínio Júnior, no curso de Engenharia de Software da Pontifícia Universidade Católica de Minas Gerais.

Utilizamos uma matriz de adjacência para representar o grafo, onde cada elemento da matriz indica a presença de uma aresta entre dois vértices. Essa escolha foi feita devido à simplicidade de implementação e à facilidade de visualização das relações entre os vértices.

Durante nossa análise, nos deparamos com alguns problemas sem solução em relação à eficiência dos algoritmos. A complexidade desses algoritmos pode se tornar impraticável quando aplicados em grafos de grande escala, especialmente quando o número de vértices é muito superior ao número de arestas.

Neste relatório, descreveremos os algoritmos Naive e Warshall, suas implementações em Java, análise de complexidade e compararemos seus tempos de execução em diferentes grafos. Além disso, discutiremos os resultados obtidos e faremos considerações sobre a eficiência e eficácia desses algoritmos.

2 Algoritmo Naive

2.1 Implementação

```
1   void naiveSearch() {
2       for (int i = 0; i < V; i++) {
3           boolean[] visited = new boolean[V];
4           DFS(i, visited);
5       }
6   }
7
8   /**
9   * Executa uma busca em profundidade a partir de um vrtice especfico.
10  *
11  * @param v o vrtice de onde a busca comear.
12  * @param visited o conjunto de vrtices visitados.
13  */
14  void DFS(int v, boolean[] visited) {
15      visited[v] = true;
16      for (int n = 0; n < V; n++) {
17          if (adj[v][n] && !visited[n]) {
18              DFS(n, visited);
19          }
20      }
21  }
```

Listing 1: Implementação do algoritmo Naive

2.2 Análise de Complexidade

A complexidade do algoritmo Naive é dada por: $O(V^3)$

Isso ocorre porque o algoritmo realiza um loop aninhado triplo para cada vértice do grafo, resultando em uma complexidade cúbica em relação ao número de vértices.

3 Algoritmo Warshall

3.1 Implementação

```
1   void warshall() {
2       boolean[][] reach = new boolean[V][V];
3
4       for (int i = 0; i < V; i++) {
5           System.arraycopy(adj[i], 0, reach[i], 0, V);
6       }
7
8       for (int k = 0; k < V; k++) {
9           for (int i = 0; i < V; i++) {
10              for (int j = 0; j < V; j++) {
11                  reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
12              }
13          }
14      }
15  }
```

Listing 2: Implementação do algoritmo Warshall

3.2 Análise de Complexidade

A complexidade do algoritmo Warshall é dada por: $O(V^3)$

Assim como o algoritmo Naive, o algoritmo Warshall também possui uma complexidade cúbica em relação ao número de vértices do grafo. Ele realiza um loop aninhado triplo para cada vértice do grafo, tornando a complexidade do mesmo nível.

É importante observar que, embora ambos os algoritmos tenham a mesma complexidade, o algoritmo Warshall é mais eficiente em termos de tempo de execução na prática devido à sua abordagem otimizada.

4 Experimentos

4.1 Configuração dos Experimentos

Os experimentos foram realizados utilizando os seguintes grafos:

- `Grafo 1: grafo_100.txt`
- `Grafo 2: grafo_1000.txt`
- `Grafo 3: grafo_10000.txt`
- `Grafo 4: grafo_100000.txt`

Para cada grafo desses foram criadas 100 vezes mais arestas que o número de vértices e sem arestas repetidas. E ambos os algoritmos foram rodados com um grafo estruturado dentor de uma matriz de adjacência.

4.2 Resultados

A Tabela 1 apresenta os tempos de execução dos algoritmos Naive e Warshall para cada um dos grafos:

Table 1: Comparação de tempo de execução dos algoritmos em ms

Grafo (nº de vértices)	**Naive (ms)**	**Warshall (ms)**
Grafo 1 (100)	6.128.300ms	7.035.100ms
Grafo 2 (1000)	670.149.700ms	703.510.000ms
Grafo 3 (10.000)	tende ao ∞	tende ao ∞
Grafo 4 (100.000)	tende ao ∞	tende ao ∞

5 Conclusão

Apesar de ter nos deixado malucos em busca de uma solução que tornasse possível analisar os grafos maiores, esse trabalho foi uma ótima forma de provar como alguns problemas na computação simplesmente ainda não tem soluções alcançáveis.

Após tentarmos diversas vezes ficou evidente que não seria possível alcançar uma solução para os grafos 3 e 4 caso fossem gerados com mais que uma aresta por vértice. A proporção de crescimento cúbica na complexidade dessa solução gera uma impossibilidade para os hardwares de hoje conseguirem concluir os cálculos em um tempo alcançavel.

Esperamos ter conseguidos deixar o algoritmo o melhor otimizado o possível para alguém com nossas habilidades.

6 Referência

Agradecemos ao professor Zenilton Kleber Gonçalves do Patrocínio Júnior pela orientação neste trabalho.