Trabalho Prático 1

Algoritmos 1

João Antonio Oliveira Pedrosa
Matrícula: 2019006752

[1] Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil

joao.pedrosa@dcc.ufmg.br

1. Introdução

O trabalho propõe a implementação de uma estrutura de rotas de entrega de vacinas, composta por Centros de Distibuição (CDs) e Postos de Vacinação (PVs).

Dados os conjuntos de Centros de Distribuição, Postos de Vacinação e as ligações entre eles, os seguintes problemas devem ser resolvidos:

- **Definir quais PVs são alcançáveis:** As vacinas possuem um incremento X de temperatura a cada unidade distância percorrida, sendo assim, deve se responder quais PVs são alcançáveis de maneira que a temperatura das vacinas não sofra um incremento maior do que 30^o.
- **Definir se existe rota com repetição:** É preciso responder se existe alguma rota que passa pelo mesmo PV duas vezes.

2. Modelagem Computacional

A estrutura de entrega das vacinas pode ser traduzida para um grafo, no qual os serão os PVs e as arestas serão as ligações que existem entre eles. Para que os CDs também sejam compreendidos nesse grafo, podemos marcar como "Fontes", todos os PVs adjacentes à algum CD. Dessa forma, não precisamos armazenar os CDs como vértices do grafo.

Agora, os dois problemas que precisam ser resolvidos podem ser traduzidos para dois problemas clássicos de teoria dos grafos:

- **Alcançabilidade:** Perceba que, se a temperatura aumenta X graus a cada unidade de distância e nós queremos que a temperatura aumente no máximo 30 graus, então os nós alcançaveis são os nós que ficam a uma distância $\leq \left\lceil \frac{30}{X} \right\rceil$ do CD mais próximo. Então transformamos a tarefa em um clássico problema de distância mínima que podemos resolver com uma BFS com Múltiplas Fontes.
- **Rotas Repetidas:** Existirá uma rota com vértice repetido no grafo, se e somente se, existir um vértice que, a partir de si, consegue chegar em si mesmo. Ou seja, para que exista uma rota com vértice repetido, deve existir um ciclo no grafo. Novamente, temos em mãos um problema clássico de teoria dos grafos, o qual pode ser resolvido com uma DFS. Ao chamarmos a DFS para um vértice, usaremos uma váriavel *booleana* para marcar que estamos na recursão desse vértice. Ao fim da recursão, marcaremos a *booleana* novamente, indicando que a recursão desse vértice acabou. Se chegarmos à um vértice com a recursão em andamento, significa que o grafo possui um ciclo.

3. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

3.1. Função *main*

A main do programa recebe a entrada e instancia a classe **Graph**. Após isso, ela faz uma chamada o método **CalcDist()** e imprime todos os vértice com distância $\leq \left\lceil \frac{30}{X} \right\rceil$. Por fim, uma chamada do método **CheckCycle()** é realizada.

3.2. Class Graph

A única classe utilizada na implementação tem o nome de *Graph* e possui os seguintes atributos:

- **Do tipo `std::vector<vector<int>>`**: Uma lista de adjacências.
- **Do tipo `std::vector<int>`**: Um vetor de distâncias, um vetor de fontes e um vetor de cores.
- **Do tipo `int`**: O tamanho do grafo.
- **Do tipo `bool`**: Um indicador se o grafo possui ciclos.

Além disso, a classe possui os seguintes métodos:

- **CalcDist()**: Calcula a distância de todos os PVs até o CD mais próximo.
- **CheckCycle()**: Checa se existem ciclos no grafo.
- **Dfs(int x)**: Método auxiliar utilizado pelo método **CheckCycle()**
- **AddEdge(int x, int y)**: Adiciona uma aresta de x à y.
- **AddSource(int x)**: Adiciona x à lista de fontes.

3.3. Menor Distância

Para encontrar a menor distância de todos os PVs até o CD mais próximo, foi utilizado, como explicado na parte de modelagem, uma BFS com múltiplas fontes. Uma BFS com múltiplas fontes é uma simples adaptação da BFS que, antes do ínicio do processamento, coloca todas as fontes na fila. Assim, a distância calculada, sempre será a distância à fonte mais próxima, afinal, a BFS não fará calcúlos para vértices repetidos e a fonte mais próxima sempre será a primeira a alcançar o vértice.

Esse algoritmo foi implementado no método **CalcDist()** e foi utilizado uma **`std::queue<int>`** para a implementação da fila. Segue o pseudocódigo do algoritmo utilizado:

Obs: Devido ao pacote do LaTeX utilizado, algumas palavras do pseudocódigo estão em inglês. Mesmo assim, optei por deixar o restante em português, por uma questão de coesão com o restante do documento.

Algorithm 1 BFS com Múltiplas Fontes

INPUT: Lista de Adjacências G e lista F de PVs adjacentes à CDs
OUTPUT: Lista D de distâncias

```
D ← vetor com |G| elementos inicializados com valor ∞
Q ← fila vazia
for all v ∈ F do
    D[v] ← 1
    Insere v em Q
end for

while Q possui algum elemento do
    v ← Elemento no início da fila Q
    Retira v de Q
    for all u ∈ G[v] do
        if D[v]+1 < D[u] then
            D[u] ← D[v] +1
            Insere u em Q
        end if
    end for
end while
return D
```

3.4. Encontrando o Ciclo

Como explicado na parte de modelagem, para o algoritmo de checagem de ciclos, nós iremos utilizar um vetor de cores. Inicialmente, iremos colorir todos os vértices de cinza. Quando começarmos a recursão de um vértice, iremos o colorir de branco e, assim que terminarmos a recursão, colororiremos este mesmo vértice de preto. Então, basta que rodemos uma DFS que faça este processo a partir de cada uma das fontes. O grafo possuirá um ciclo se e somente se, em algum momento, a DFS chegar em um vértice que ainda está colorido de branco.

Na classe, o método que realiza a checagem de ciclos é o **CheckCycle()**. O método utiliza um vetor de inteiros para as cores, sendo 0 para cinza, 1 para branco e -1 para preto. Segue o pseudocódigo do método:

Algorithm 2 Checagem de Ciclos

INPUT: Lista de Adjacências G e lista F de PVs adjacentes à CDs
OUTPUT: Booleana R. $True$ caso o grafo possua ao menos um ciclo. $False$ caso contrário.

```
R ← False
C ← vetor com |G| elementos

procedure DFS(v)
    C[v] ← 1
    for all u ∈ G[v] do
        if C[u] = 1 then
            R ← True
            return
        end if
        if C[u] = 0 then
            DFS(u)
        end if
    end for
    C[v] ← −1
end procedure

procedure CHECKCYCLE
    for all v ∈ F do
        C ← vetor com |G| inicializados com valor 0
        DFS(v)
    end for
end procedure

return R
```

4. Análise de Complexidade

Para essa seção, iremos denotar N como número de PVs do grafo, S como número de PVs adjacentes à CDs e M como o número de arestas. Também iremos denotar G como a lista de adjacências do grafo.

- A leitura da entrada, impressão da saída e construção do grafo, têm complexidade $\mathcal{O}(N + M)$.

- A BFS que realiza o cálculo das distâncias, visita cada vértice uma única vez, sendo assim, ela também têm complexidade $\sum_{v \in [0,N)} |G[v]| \in \mathcal{O}(N + M)$.

- Por fim, a DFS que realiza a checagem de ciclos, de maneira semelhante à DFS, visita cada vértice uma única vez, sendo assim, ela têm complexidade igual à da BFS: $\mathcal{O}(N + M)$. Entretanto, a função **CheckCycle()** realiza uma DFS para cada fonte. Sendo assim, a complexidade final da função é $\mathcal{O}(S \cdot (N + M))$

Sendo assim, separando o algoritmo nessas 3 partes principais, temos uma complexidade de $\mathcal{O}(2 \cdot (N + M) + S \cdot (N + M)) \in \mathcal{O}(S \cdot (N + M))$.

Obs: O valor máximo de M não é dado na documentação, mas é limitado superiormente por $N \cdot (N - 1)$. Da mesma forma, o valor de máximo S, mesmo não sendo informado na documentação, é limitado superiormente por N.