TP 01 -Trabalho Prático 1
O problema da rota de distribuição de vacinas

Jean Lucas Almeida Mota

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

jeanlk@ufmg.br

1. Introdução

O problema proposto se baseia na distribuição de vacinas para o vírus Sars-CoV-2, causador da covid-19. Neste problema, as vacinas devem ser transportadas dos centros de distribuição para os postos de vacinação, dada uma lista de centros de distribuição, postos de vacinação e as rotas entre eles. Sob as seguintes condições:

1. As vacinas devem ser mantidas a uma temperatura máxima de -60ºC, sob o risco de perder a eficácia.
2. As vacinas deixam os centros de distribuição a uma temperatura de -90ºC.
3. Rota é o caminho, com direção definida, iniciando em um centro de distribuição e passando por um ou mais postos de vacinação.
4. A cada rota as vacinas perdem uma temperatura XºC a ser definida na entrada.

O sistema desenvolvido deve informar:

1. A quantidade de postos alcançáveis.
2. Uma lista dos postos alcançáveis.
3. Informar se há alguma rota que percorre um mesmo posto de vacinação duas vezes.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++.

2.1 Modelagem e Estruturas de Dados

A fim de garantir praticidade e simplicidade na implementação, a estrutura escolhida para modelar o problema foi o grafo. Pois através do grafo podemos interpretar os centros de distribuição e postos de vacinação como vértices; e as rotas como arestas.

O grafo foi modelado usando listas de adjacência. Por causa do seu custo de espaço O(aresta+vértice), mas principalmente pelo seu custo de tempo ao inspecionar vizinhos O(n). Para fins de praticidade da lista, foi considerado que todo vértice possui uma rota para ele mesmo.

Para verificar as rotas entre os centros e os postos, foi usada o algoritmo Busca por Largura (Breadth First Seach, BFS). Uma vez que é possível através dele garantir a menor rota.

Para verificar a existência de postos visitados mais de uma vez, foi usado uma modificação do algoritmo Busca por Profundidade (Depth First Seach, DFS).
E por fim, a entrada fornecida pelo problema nomeia os postos de vacinação de 1 até n, onde n é um número natural. No entanto, a entrada fornecida não nomeia os centros de vacinação. Dessa forma, para simplificar a implementação, os nomes dos postos foram transladados, usando a seguinte equação $p = p + c$, onde p representa o nome do posto e c representa o número total de centros de distribuição. Os centros de distribuição foram nomeados de 1 até c.

2.2 Implementação

A fim de modularizar a implementação, foi implementado duas classes principais, uma de baixo nível e outra de mais alto nível. São elas:
Classe *GrafoDirecionado*: Classe de mais baixo nível, onde foi implementado a estrutura grafo utilizando lista adjacente. Esta foi implementada usando *vector* da biblioteca padrão C++.
A fim de simplificar a implementação, a árvore retornada pelo algoritmo BFS foi armazenada em uma lista de listas. Onde a lista-n representa os nós que estão à distância n.
A classe *GrafoDirecionado* possui os métodos:

- *Getters* e *Setters* para o número de arestas (m) e o número de nós (n).
- *GetArvore()*: retorna a árvore criada pelo algoritmo BFS. Caso o BFS não seja iniciado previamente, ela retorna nulo.
- *AdicionaAresta(no1, no2)*: Adiciona a aresta do *no1* para o *no2* na lista de adjacência. Caso algum dos vértices não estejam na lista, inicia o mesmo criando uma aresta para ele mesmo.
- *Imprime()*: Imprime através do *cout* a lista de adjancencia.
- *ImprimeComCorrecao(centro)*: Imprime através do *cout* a lista de adjancencia voltando a translação feita ao iniciar o grafo.
- *ImprimeArvore()*: Imprime através do *cout* a árvore retornada pelo BFS. Caso o BFS não seja iniciado previamente, não imprime nada.
- *bfs(s)*: Algoritmo de busca por largura que salva o resultado em uma matriz, onde cada linha n representa uma distância n do nó analisado no bfs. Pseudocódigo:

```
bfs(vertice s){
  cria um map<vertice,booleano> descobertos;
  marca cada vertice em descobertos como falso;
  marca o vertice s em descobertos como true;
  i = 0;
  cria uma listas de listas chamada arvore_auxiliar
  inicializa arvore_auxiliar[i] com s;
  while(arvore_auxiliar[i] != vazio){
      cria uma lista aux auxiliar de nos;
      for(cada nó N em arvore[i]){
          for(cada nó M conectado a N){
              if(descobertos[M]==falso){
                  descobertos[M]==true;
                  adiciona M na lista aux;
              }
          }
```

```
        }
        adiciona a lista aux na arvore_auxiliar[i+1];
        i++
    }
    arvore = arvore_auxiliar; //variável global
}
```

- *ContagemCiclos()*: Usa uma versão modificada do DFS para verificar se há um caminho começando do centro de distribuição que percorre um mesmo posto duas vezes. Independente da temperatura das vacinas. Uma forma análoga de entender esse algoritmo é: identifica se há um ciclo no grafo começando do centro de distribuição. Usa o método auxiliar para recursão *ContagemCiclosRecursivo()*. Pseudocódigo:

```
ContagemCiclos(centro){
    cria um map<vertice,booleano> visitados;
    cria um map<vertice,booleano> revisitados;
    marca cada vertice de visitados como false;
    marca cada vertice de revisitados como false;
    for(cada centro c de distribuição){
        if(ContegemCicloRecursivo(c,visitados,revisitados)){
            retorna existe um ciclo;
        }
    }
    retorna nao existe um ciclo;
}
ContagemCiclosRecursivo(c,visitados,revisitados){
    if(visitados[c]==false){
        visitados[c]==true;
        revisitados[c]==true;
        for(percorre os nos n conectados a c){
            if(visitados[n]==false){
                if(ContagemCiclosrecursivo(n,visitados,revisitados)){
                    retorna existe ciclo;
                }else if(revisitados[n]==true){
                    retorna existe ciclo;
                }
            }
        }
    }
    revisitados[c]=false;
    retorna não existe ciclo;
}
```

Classe *SistemaVacinas*: É a classe de mais alto nível. É responsável por ler a entrada, usar os métodos da classe *GrafoDirecionado* para fazer as operações e imprimir a saída solução do problema. Possui os métodos:

- *Leitura()*: Lê a entrada de dados preenche as variáveis centro, posto, temperatura e distância. Além, é claro, de preencher o grafo principal usando *GrafoDirecionado.AdicionaAresta(no1,no2)*.
- *CalculaRotas()*: Responsável por aplicar o algoritmo BFS em cada centro de distribuição e recolher os postos alcançáveis. Pseudocódigo:

```
CalculaRotas(){
    testa se a temperatura permite percorrer uma distancia maior que ze
ro;
    for(cada centro c){
        grafo.bfs(s); //grafo é uma variavel dessa classe
        arvore recebe grafo.GetArvore() //arvore é uma variavel dessa c
lasse;
        usa o método concatena(arvore,distancia) para pegar todos os ve
rtices que estão a uma dada distancia;
    }
    Ordena a lista de vertices que o método concatena(arvore,distancia)
 criou;
    retira os vertices repetidos dessa lista;
    translada os vertices para ajustar a resposta;
}
```

- *Concatena(arvore,distancia)*: É responsável por preencher uma lista de vértices que estão a uma dada distância usando uma árvore criada pelo algoritmo BFS.
- *Ordena()*: Ordena a lista de vértices que foi preenchida por *Concatena(arvore,distancia)*.
- *ImprimeGrafo()*: Faz a impressão do objeto gráfico com o qual estou trabalhando. Função útil para debug.
- *ImprimeGrafoComCorrecao()*: Faz a impressão do objeto grafo com o qual estou trabalhando transladado. Função útil para debug
- *ImprimeListaPostosPossiveis()*: Imprime a lista de postos alcançáveis. Esse método apenas imprime, logo deve-se antes fazer o uso dos métodos *CalculaRotas(), Concatena(arvore,distancia)* e *Orderna()*.
- *ImprimeNumeroPostosPossiveis()*: Imprime o número de postos alcançáveis. Esse método apenas imprime, logo deve-se antes fazer o uso dos métodos *CalculaRotas(), Concatena(arvore,distancia)* e *Orderna()*.
- *ImprimeContagemCiclos()*: Imprime 1 se há um caminho que passa por um posto mais de uma vez partindo de um centro de distribuição e 2 se não há.

3. Análise de Complexidade de Tempo

A entrada do sistema pode ser interpretada da seguinte forma: Sendo **n** o número de total de centros de distribuição mais o número de postos. E sendo **m** a quantidade de ligações entre eles. Também vamos considerar **c** igual ao número de centros de destruição e **p** igual ao número de postos. Assim temos que **n=c+p** .

O

sistema faz o uso de 5 métodos principais de alto nível, logo somando a complexidade de tempos desses 5 métodos teremos a complexidade total do sistema. Os métodos são:

Leitura(): Ao inicializar o sistema recebendo os dados de entrada, há a operação de construir o grafo. Temos o custo de inicialização **O(n+m)**.
CalculaRotas(): Esse método chama o algoritmo BFS c vezes. Logo **cO(n+m)**.
ImprimeNumeroPostosPossiveis(): Apenas imprime uma saída, **O(1)**.
ImprimeListaPostosPossiveis(): Apenas imprime uma saída, **O(1)**.
ImprimeContagemCiclos(): Executa a versão modificada do DFS c vezes. Logo **cO(2m)**.

Dessa forma, a complexidade total do sistema é dada por:
max(O(n+m) , cO(n+m) , O(1) , O(1) , cO(2m))
max(cO(n+m) , cO(2m))
Assim, se n<=m então o sistema é **cO(2m)**. Se n>m, então o sistema é **cO(n+m)**.

4. Conclusão
Após a leitura do problema proposto, nota-se o uso essencial da estrutura grafo. O uso dessa estrutura simplifica muito a resolução do problema, principalmente por causa dos algoritmos BFS e DFS. O grande esforço deste trabalho estava em adaptar o uso dessas estruturas a fim de torna-las mais especificas para o problema proposto.

Ao finalizar o sistema, ficou evidente a necessidade de entender a fundo como os algoritmos BFS e DFS funcionam, pois é fácil cometer erros lógicos durante sua implementação, dificultando muito o debug.

5. Execução
O programa foi desenvolvido no Windows 10 e testado no Windows 10 e no Linux.

Para execução no Windows foi executado o seguinte comando:
mingw32-make
./tp01
Para execução no Linux foi executado o seguinte comando:
make
./tp01
Observação: Os testes foram executados através do comando:
./tp01 < CasoTeste01a.txt

References

ALMEIDA, M Jussara. Slide Graph Traversal (2021). Disciplina Algoritmos 1 DCC206 UFMG.

Chaimowicz, Luiz e Prates, Raquel. Slide sobre Listas, Pilhas e Filas (2020). Disciplina Estrutura de Dados DCC205 UFMG.

Post sobre a estrutura map. Disponível em https://www.delftstack.com/pt/howto/cpp/how-to-iterate-over-map-in-cpp/ Publicado em 01/11/2017. Acessado em 19/01/2021.

Post sobre concatenação de vetores. Disponível em
https://www.techiedelight.com/concatenate-two-vectors-cpp/
Publicado em janeiro de 2020. Acessado em 19/01/2021.