Trabalho Prático 2
Desemprego

Vinicius Trindade Dias Abel
Matrícula: 2020007112

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

viniciustda@ufmg.br

1. Introdução

O problema proposto foi desenvolver um algoritmo que pudesse garantir vagas de emprego para o máximo de usuários da rede social LinkOut (o principal objetivo da LinkOut é conectar seus usuários às vagas de emprego ideais), dado um grafo que representa as qualificações usuário-emprego.

Para resolver o problema, foi implementado um algoritmo guloso que encontra uma solução rapidamente e um algoritmo exato que sempre encontra a melhor solução possível.

2. Método

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de um grafo bipartido, implementado por um vetor de adjacência (escolhido pela facilidade de trabalhar com vetor) e um vetor tipo_vertice que garante que o grafo seja bipartido. Os vértices representam usuários e vagas e as arestas representam as qualificações usuário-emprego.

Para a resolução do problema, foi utilizado um algoritmo guloso, que relaciona o usuário a vaga pela ordem em que aparecem, e o algoritmo Ford-Fulkerson, que resolve o problema encontrando o fluxo máximo do grafo.

O algoritmo guloso utiliza um vetor para armazenar os vértices que já foram visitados e o vetor de adjacência. Já o algoritmo exato utiliza um vetor para armazenar o vértice pai de cada vértice e o algoritmo DFS que, por sua vez, também usa um vetor para armazenar os vértices que já foram visitados.

2.2. Funções

AdicionaVertice

A função recebe como parâmetro uma string vértice que é o nome do vértice e um bool tipo que define se ele é do tipo usuário (true) ou do tipo vaga (false). Se o vértice ainda não foi adicionado, esta verificação é feita percorrendo um unordered_map que atribui um id int para o vértice, então o vértice recebe

um id, é armazenado seu tipo e se o vértice é um usuário, é criada uma aresta entre o vértice e um vértice “Inicio”, se o vértice é uma vaga, é criada uma aresta entre o vértice e um vértice “Fim”. Os vértices “Inicio” e “Fim” são criados para a implementação do algoritmo exato.

AdicionaAresta

A função recebe como parâmetro dois vértices e a aresta entre eles é criada pelo vetor de adjacência utilizando os id desses vértices.

AlgoritmoGuloso

A função utiliza um vetor visitado que armazena se o vértice já foi visitado. O algoritmo percorre o vetor de adjacência e se o vértice for do tipo usuário, ele atribui a primeira vaga disponível (não foi visitada) que o usuário pode ocupar. Assim a função cria as relações usuário-vaga pela ordem em que aparecem.

AlgoritmoExato

A função é uma implementação do algoritmo Ford-Fulkerson, ela utiliza um vetor para armazenar o vértice pai de cada vértice, também utiliza a função DFS de busca em profundidade para verificar se existe um caminho de aumento no grafo, e utiliza a função CalculaPosicao para acessar a posição correta de um vértice v no vetor de adjacência de um vértice u.

DFS

A função é uma implementação do algoritmo DFS, ela utiliza um vetor visitado que armazena se o vértice já foi visitado e uma fila para percorrer o grafo, verificando se existe um caminho de aumento no grafo.

CalculaPosicao

A função recebe um verti u e um vértice v como parâmetro e descobre a posição de v no vetor de adjacência de u.

GetGuloso

A penas chama a função AlgoritmoGuloso e retorna o resultado.

GetExato

A penas chama a função AlgoritmoExato e retorna o resultado.

Main

A função main recebe o número de usuários, o número de vagas e o número de relações usuário-vaga. Em seguida, recebe as qualificações usuário-vaga estradas s por meio de um loop. Dentro do loop, ela chama as funções AdicionaVertice (duas vezes, já que será adicionado um vértice usuário e um vértice vaga) e AdicionaAresta, criando um grafo bipartido. Após a criação do

grafo, são chamadas as funções GetGuloso e GetExato para retornar o número máximo de usuários com vaga.

2.3. Configuração para teste

– Sistema Operacional do computador: Linux Ubuntu;
– Linguagem de programação implementada: C++;
– Compilador utilizado: G++ da GNU Compiler Collection;
– Dados do seu processador: i7;
– Quantidade de memória RAM: 16GB.

3. Análise de Complexidade

AdicionaVertice

A função tem complexidade O(1). Se o vértice não existe, ela atribui um id para o vértice e chama a função AdicionaAresta que também tem complexidade O(1).

AdicionaAresta

A função tem complexidade O(1). Ela adiciona os dois vértices no vetor de adjacência.

AlgoritmoGuloso

A função tem complexidade O(nm). Ela percorre os vetores de adjacência dos vértices usuários, são n usuários e o usuário pode ocupar m vagas.

AlgoritmoExato

A função tem complexidade $O(n^2)$. A complexidade do laço é a que prevalece, e o laço de repetição “while” possui outro laço de repetição “for” interno, levando a complexidade $O(n^2)$. A complexidade das funções DFS e CalculaPosicao não prevalecem em relação a complexidade $O(n^2)$.

DFS

A função tem complexidade O(nm). Ela percorre os vetores de adjacência, onde existem n vértices e cada vértice possui m vértices adjacentes.

CalculaPosicao

A função tem complexidade O(m). Ela percorre o vetor de adjacência de um vértice em especifico.

GetGuloso

A função tem a mesma complexidade do AlgoritmoGuloso.

GetExato

A função tem a mesma complexidade do AlgoritmoExato.

Geral

O TP tem complexidade $O(n^2+nm)$. A função main chama as funções AdicionaAresta n vezes e AdicionaVertice 2n vezes, então o laço de repetição da função main tem complexidade O(n), a inicialização do grafo também tem complexidade O(1). Como as funções GetGuloso e GetExato são chamadas uma vez cada, a complexidade das duas sobressaem, logo a complexidade do TP é $O(n^2+nm)$.

4. Conclusões

Este trabalho lidou com cálculo de fluxo máximo de um grafo, na qual a abordagem utilizada para sua resolução foi o algoritmo Ford-Fulkerson.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados a grafos e algoritmos de grafos.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo a melhor forma para implementar o grafo já que os vértices são do tipo string e a implementação do algoritmo de Ford-Fulkerson.

Para implementar o grafo com vértices do tipo string, criei Unordered Map que atribuía um id do tipo int para cada vértice e então criei o vetor de adjacência utilizando os id. Na implementação do algoritmo Ford-Fulkerson utilizei exemplos como base e adaptei para atender a forma como implementei o grafo, para os casos de teste 8, 9 e 10, retornou um valor próximo ao esperado, entretanto não consegui identificar o erro no código.

5. Bibliografia

- Standard C++ Library reference. Disponível em: https://cplusplus.com/reference/

- Teoria dos Grafos – Fluxo Máximo em redes. Disponível em: https://www.ibilce.unesp.br/Home/Departamentos/MatematicaAplicada/docentes/socorro/aula12_fluxoredes2016.pdf

- Ford-Fulkerson Algorithm for Maximum Flow Problem. Disponível em: https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

- Ford-Fulkerson Algorithm Explained. Disponível em: https://favtutor.com/blogs/ford-fulkerson-algorithm

- Ford-Fulkerson Algorithm. Disponível em: https://www.programiz.com/dsa/ford-fulkerson-algorithm

6. Instruções para compilação e execução

- Acesse o diretório tp1
- Utilizando o terminal, compile o TP 00 utilizando o comando: make
- Com esse comando, será gerado o arquivo tp02.exe e os arquivos *.o
- Utilizando o terminal, teste o tp02.exe utilizando os comandos:
    - make eval: avalia apenas os resultados referentes ao algoritmo exato.
    - make eval_greedy: avalia o quanto o resultado guloso é próximo do ótimo.