Trabalho Prático 2
Desemprego

Aline Cristina Pinto
2020031412

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

alinecristinapinto@ufmg.br

1. Introdução

Este trabalho possui como objetivo desenvolver uma *Proof of Concept*, através de duas soluções: uma gulosa (geralmente subótima, mas rápida) e uma exata, capazes de otimizar a rede social fundada por Willie Gates: a LinkOut. Essa rede possui como missão conectar usuários a vagas de trabalho, entretanto o engajamento da mesma não está bom por não ter um balanceamento entre recomendações de vagas entre os usuários. Sendo assim, os algoritmos criados buscam encontrar o maior número possível de pares únicos entre usuários e vagas. Para ambas soluções, foi utilizada uma modelagem com grafo bipartido que será abordada de forma detalhada nesta documentação. Da mesma forma, técnicas de algoritmos gulosos e exatos que também serão explicados e comparados.

2. Modelagem

2.1. Grafo

Para a criação de ambos algoritmos, foi modelado o grafo bipartido G(V, A), tal que :

V = { v | v representa um usuário ou uma vaga }

A = { (v, u) | u e v formam uma associação usuário-vaga, indicando que o candidato atende aos requisitos da vaga }

Esse grafo não possui pesos, logo, é **não ponderado**. Além disso, apesar de um usuário estar para a vaga assim como a vaga está para o usuário, por motivos de simplicidade, o mesmo foi construído como um **grafo direcionado**. Esse direcionamento vai do usuário para a vaga.

O objetivo é determinar o maior número de pares únicos de usuário-vaga.

2.1.1. Estrutura de dados

Para representar esse Grafo, foi optado pelo uso de uma *lista de adjacências*, levando em consideração duas vantagens, sendo essas o custo de memória (ocupa menos memória que uma matriz de adjacência) e a facilidade de sua manipulação dada a liberação do uso da biblioteca *STL - Standard Template Library* em C++.
Foi criado então um vector de Arestas, sendo a Aresta uma struct que possui fonte e destino. A representação final ficou como:

fonte	Aresta(fonte, destino)	Aresta(fonte, destino)	…

Além disso, como os dados são recebidos como string, uma lógica de mapeamento foi criada de tal forma que:

1. Os usuários são representados de 1 ao número de usuários U, logo {1, …, U}
2. As vagas são representadas do posterior do número de usuários até que se compute o número de vagas, logo {U+1, …, U+J}

2.2. Solução Gulosa

Buscando encontrar o maior número possível de combinações, a solução mais intuitiva foi criada, que consiste em dar preferência para os usuários que possuem menos ofertas de vagas, ou seja, consultar os vértices que possuem menos arestas primeiro. Assim, percorre-se os usuários com menos vagas disponíveis selecionando a primeira oferta não ocupada das opções e salva-se em um vetor auxiliar que a vaga foi ocupada. Nota-se que após essa associação, não voltamos atrás caso a vaga selecionada seja melhor associada (que dê mais combinações) a outra pessoa. Por conta disso, essa solução será subótima em alguns casos, visto que o número de associações dependerá da ordem que os dados foram recebidos.
A seguir, temos um pseudo-código que representa essa solução com a complexidade. Por motivos de dificuldades pessoais com C ++, o grafo em si não foi ordenado pelo número de arestas. A solução adotada foi utilizar uma fila de prioridades auxiliar, que possui como par (número de arestas, id usuário):

```
Integer numMatches <- 0                                          O(1)
Integer[] ofertasOcupadas <- {}                                  O(1)
Priority Queue Q

for each usuário U de S                                          O(V)
  Q <- pair<U.sizeAresta, U.fonte>                               O(log V)

while Q IS NOT EMPTY                                             O(V)
    U <- pega o usuário com menos ofertas from Q                 O(1)
    U.pop                                                        O(log V)
    for each oferta V de U  (vertice usuário)                    O(E de U)
         if ofertasOcupadas(V) está desocupada                   O(1)
               ofertasOcupadas[V] = 1                            O(1)
```

```
                numMatches = numMatches + 1                    O(1)
                break                                          O(1)

    return numMatches                                          O(1)
```

Observa-se que as operações que consomem maior complexidade são:

1. A ordenação da fila de prioridade O(V)(for) + O(log V)(push heap) + O(1)(size) = O(V);
2. O while+for interno, sendo o while O(V) e o for O(E) no pior caso, quando o vértice possui todas as arestas e precisa-se percorrer todas para verificar se é possível fazer o match, logo O(V + E)

Logo, temos que a complexidade desse algoritmo guloso será O(V+E).

2.3. Solução Exata

Esse problema também pode ser modelado como um problema de fluxo máximo, no qual a capacidade máxima de cada aresta será um. Sendo assim, poderia-se usar o algoritmo de Ford-Fulkerson para encontrar o fluxo máximo. Porém, como a capacidade de cada aresta varia apenas entre 0 e 1 e o grafo trabalhodo é bipartido, a solução pode ser simplificada utilizando apenas uma adaptação de DFS tenta todas as possibilidades de atribuir uma vaga a um usuário a partir da seguinte forma:

1. Percorre-se todas as vagas dado um usuário apto, para cada vaga:
    a. Se a vaga não foi atribuída a ninguém, atribui-se a esse usuário e a função retorna true;
    b. Se a vaga foi atribuída a outra pessoa p, verifica-se recursivamente se p pode ser atribuída a outra vaga (desconsiderando a vaga atual através de um vetor de visitados). Se sim, atribui-se a vaga ao usuário e a função retorna true;
2. Se a função baseada em DFS retorna true para a função principal, significa que há um caminho possível que pode ser contabilizado.

A seguir, temos um pseudo-código que representa essa solução:

```
begin verificarMatch // baseado em DFS
    for all arestas V                                          O(E)
       if V não foi visitado and V é adjacente a U             O(1)
            visitados(V) = 1
            if V não foi atribuído a ninguém or verificarMatch(S,
      matchVagas[v], visitados, matchVagas) é true
                matchVagas[V] = U                              O(1)
                return true                                    O(1)
```

```
    return false                                                    O(1)
end

begin principal
    numMatches = 0                                                  O(1)
    matchVagas[numOfertas] = {}                                     O(1)
    for each usuário U em S                                         O(V)
       visitados[numOfertas] = {}                                   O(1)
       if verificarMatch(S, U, visitados, matchVagas) é true
            numMatches += 1                                         O(1)

    return numMatches                                               O(1)
end
```

Observa-se que o algoritmo itera sobre cada vértice do grafo (visível no pseudo-código nomeado *principal*), chamando para cada um deles a função *verificarMatch* que aplica uma DFS nas arestas correspondentes (na modelagem usada V será o número de usuários e E o número de ofertas). Sendo assim, a complexidade será de O(V * E).

2.4. Solução Exata vs. Solução Gulosa

Como já discutido anteriormente nas seções 2.2 e 2.3, a complexidade de tempo da solução exata tende a ser um pouco maior O(V*E) do que a gulosa O(V+E) apesar de ambas serem lineares. Para o problema em questão, a solução gulosa encontrada é satisfatória (utilizando as entradas disponibilizada pelos monitores para a realização desse trabalho, notou-se uma confiabilidade de aproximadamente 98,8% da solução gulosa para o resultado exato, o que é um valor bom), logo pode ser útil para aumentar o engajamento da rede social sendo que a mesma dá prioridade para as pessoas que possuem menos ofertas.

Trade-offs

A vantagem do algoritmo exato é poder encontrar o melhor dos casos possíveis. Porém, para entradas maiores, algoritmo exato pode custar mais tempo do que o algoritmo guloso, que dependendo da entrada pode encontrar o mesmo resultado do exato. Além disso, quando o guloso não entrega o valor exato, ele entrega um resultado subótimo que pode ser suficiente para a rede social, logo é uma opção a ser usada.

A seguir temos um comparativo de dois testes utilizando os **algoritmos de forma isolada** para duas entradas de tamanhos diferentes. Para as medições foi utilizado um cronômetro de relógio, portanto, os valores são apenas aproximações:

Algoritmo	Tempo (s) para entrada pequena (test_case0.txt)	Tempo (s) para entrada longa (test_case13.txt)
Exato	< 0,01s	~ 3,62
Guloso	< 0,01s	~ 0,02

Nota-se que para entradas pequenas não há uma diferença perceptível, enquanto para entradas maiores, como o caso de teste 13, o algoritmo exata foi bastante lento comparado ao guloso. Logo, a vantagem do guloso está em seu custo de tempo menor.