UNIVERSIDADE FEDERAL MINAS GERAIS
Algoritmos - Trabalho Prático 2

Nome: João Vítor Santana Depollo
Matrícula: **2021039751**

Descrição do Problema

Este trabalho consiste em fazer a relação de **um para um** entre vagas de emprego disponíveis e usuários de uma plataforma. Neste cenário, o algoritmo elaborado recebe as ofertas de emprego que todos candidatos recebem e relaciona a quantidade máxima entre vagas de emprego e usuários.

Modelagem - Entrada

Para otimizar os resultados e chamadas dos algoritmos, as ofertas de vagas de todos os usuários são ordenadas de acordo com a quantidade de vagas que cada usuário recebeu. Desta forma, os usuários que têm poucas vagas não vão necessitar que outros usuários troquem seus empregos para liberarem as vagas destes outros usuários.
Para processar a entrada, o algoritmo roda um loop de acordo com a quantidade O de ofertas feitas. Como é necessário os usuários pela quantidade de vagas ofertadas, em cada iteração do loop é feita uma verificação se o nome recebido da iteração atual é diferente do nome da passada. Se sim, o nome é adicionado a lista com todas as ofertas que ele recebeu, se não, a vaga ofertada é adicionada a uma lista deste usuário.
Abaixo, segue o pseudocódigo da implementação:

```
 name, job, lastName;
cin >> name >> job;
lastName = name;
for (int i = 1; i < ofers; i++)
{
    cin >> name >> job;
    if (name == lastName)
    {
        userJobsOfer->addJob(job);
    }
    else
    {
        userJobs.push_back(userJobsOfer);
        userJobsOfer = new UserJobsOfer(name, job);
    }
    if (i == ofers - 1)
    {
        userJobs.push_back(userJobsOfer);
    }
    lastName = name;
}
```

Como a entrada é constituída por um único loop que depende da quantidade de ofertas ***f*** de vagas (quantidade de ofertas), esta parte do programa tem uma ordem de complexidade de ***O(f)***.

Após essa etapa, é iniciado o processo de **montagem do grafo**, que utiliza o método de ordenação automático do stl(***O(n*logn)***) e adiciona ao grafo as ofertas de emprego de um respectivo usuário, agora não utilizado o **nome do usuário** como chave, mas o índice que ele possui nesta lista ordenada de ofertas, permitindo um acesso mais rápido a memória de oferta. Como a montagem do grafo é relativa a quantidade de ofertas de emprego, esta etapa possui complexidade de ***O(n)***. Logo, até dado momento, o algoritmo tem a complexidade da ordenação que foi feita, no caso, ***O(n*logn)***.

Modelagem - Guloso

Pensando na otimalidade da questão, foi implementado um algoritmo guloso que pareia um usuário com a primeira vaga disponível que ele possui. Para melhorar o resultado deste algoritmo e otimizar o algoritmo exato, é feita uma ordenação das ofertas de emprego, dessa forma, os primeiros usuários a serem escolhidos são aqueles que possuem menos vagas, logo, o algoritmo aumenta as escolhas de pessoas que provavelmente precisam que outros usuários troquem de empregos para que suas vagas fiquem disponíveis.
Abaixo, segue o pseudocódigo da implementação do algoritmo guloso:

```
function guloso(U):
    lista_vagas = lista_vagas();
    Para todos u usuários em U:
        Para todas vaga v de orfertas em u:
            if(!lista_vagas.ocupada(v)):
                lista_vagas.ocupar(v);
                break;
return v.size;
```

Considerando ***n*** como o número de usuários **U** e **m** como o número de vagas **V**, o algoritmo atinge uma ordem de complexidade de ***O(n+m)***, pois é feita a verificação para todas as vagas de todos os usuários. Ao considerar a ordenação como melhoria do algoritmo,a diferença do resultado é considerável:

Execução com ordenação por quantidade de ofertas:

```
Average proximity from optimal: 98.8578%
```

Execução sem ordenação por quantidade de ofertas:

```
Average proximity from optimal: 94.5035%
```

Neste teste, alguns resultados sem a ordenação chegam a ter uma proximidade de 80% do algoritmo ótimo.

Modelagem - Exato

Pensando em um modelo de grafo bipartido, o algoritmo elaborado itera por todos usuários e se possível, atribui uma vaga para tal usuário. Neste cenário, para ocupar o maior número possível de vagas, o algoritmo executa da seguinte forma:

Para todas as vagas que o usuário recebeu, o algoritmo verifica se a vaga analisada está sendo usada por um usuário, se sim, são feitas chamadas recursivas desse mesmo procedimento para tentar liberar essa vaga do usuário que a ocupa. Caso contrário, se a vaga estiver livre, ela é marcada como ocupada pelo usuário analisado.
Abaixo, segue pseudocódigo do algoritmo implementado:

```
function podeContratarUsuario(user, visited){
    para vagas v de user:
        visited[user][v] = true;
        if(!visited[user][v]):
            se v.empty:
                v.ocupar(user)
                return 1;
        return podeContratarUsuario(v.usarioOcupador);
    return -1;
}
function exato(){
contador = 0;
para todos usuários u em U:
    visited = list_visited();
    contador+=podeContratarUsuario(u, visited)

}
```

Para todos os usuários, é feita a chamada do procedimento que verifica se o usuário pode receber uma vaga ou não. Se sim, adiciona ao contador de vagas preenchidas. Considerando ***f*** como o número de ofertas de emprego, as chamadas são feitas para todas as ofertas dos usuários. Dessa forma, o algoritmo exato atinge a ordem de complexidade de ***O(n*f)***, a maior complexidade de todo o algoritmo.

Análise entre o algoritmo exato e guloso:

A ordenação feita propicia vantagens para ambos os algoritmos, porém o algoritmo exato não afetará seu resultado caso a ordenação seja removida, porém poderia aumentar bastante o número de chamadas da função que verifica se um usuário pode ocupar a vaga, o que deixa o algoritmo mais lento. Já o algoritmo guloso, não tem a sua complexidade alterada se a entrada for pré processada ou não, porém ele pode ocupar vagas com usuários que possuem muitas outras ofertas. Dessa forma, perde-se a melhora feita. O fato do algoritmo guloso simplesmente preencher a vaga que ele recebe, faz com que ele tenha uma complexidade melhor em relação ao algoritmo guloso.

Comparativo de complexidade:

- **Algoritmo guloso: *O(n+m)***
- **Algoritmo exato: *O(n*f)***