Documentação Trabalho Prático 1 da Disciplina de Algoritmos 1

Daniel Oliveira Barbosa
Matrícula: 2020006450

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

1. Introdução

A rede social profissional LinkOut está com um problema de pareamento dos seus usuários com as vagas compatíveis. Precisamos encontrar o número máximo de pareamentos entre usuários e vagas compatíveis usando dois métodos: um método guloso e outro exato. O método guloso deverá ter menor complexidade porém não garantirá uma solução ótima. Já o método exato deverá garantir a solução exata, porém terá uma complexidade maior. Aqui nessa documentação, entraremos em detalhes sobre ambas as soluções desenvolvidas para resolver o problema do LinkOut.

2. Solução Gulosa

A solução gulosa tenta resolver o problema de pareamento de usuários e vagas através da modelagem como um problema em grafos. O grafo é construído a partir dos usuários e vagas disponíveis, onde cada um é representado como um vértice em um grafo bipartido. Uma partição é composta pelos usuários e a outra pelas vagas. A compatibilidade entre os usuários e vagas é representada por arestas de ida (da partição usuários para a partição vagas) e volta (da partição vagas para a partição usuários) entre o usuário e vaga no grafo. A ida e a volta é feita para modelar a compatibilidade não direcional entre a vaga e o usuário.

Para lermos a entrada e gerarmos o grafo, usamos a estrutura de dados de mapas desordenados (unordered map) para armazenar os usuários e vagas únicas. Dado que esse mapa é implementado usando hash, podemos inserir e verificar se um vértice (usuário ou vaga) existe em O(1). Para armazenar o grafo, utilizamos uma lista de adjacências, implementado como um vetor alocado dinamicamente de vectors. Cada vértice do grafo é associado a uma lista que contém os seus vértices vizinhos, ou seja, aqueles que são compatíveis com ele.

OBS.: tome N como o número de vértices do grafo e M o número de arestas.

O algoritmo guloso faz as seguintes operações:

Primeiro, os vértices do grafo são ordenados em ordem ascendente de grau, ou seja, do vértice com menos vizinhos para o vértice com mais vizinhos. Essa ordenação é feita pelo sort() (da STL) com complexidade O(N * log(N)) e é realizada para priorizar o pareamento com vértices de menor grau. Isso aumenta as chances de vizinhos com menor número de vizinhos serem pareados. Dado que vértices de maior número de vizinhos possuem mais chances de pareamentos, priorizar o pareamento de vértices de menor grau aumenta a probabilidade de se realizar mais pareamentos como um todo.

Após a ordenação, iteramos sobre esses vértices na ordem ascendente de grau em O(N). Para cada vértice, testamos se ele já está pareado (verificando o vetor de vértices já pareados em O(1)).

Se já estiver pareado, passamos para o próximo vértice na ordem ascendente de grau. Caso não esteja pareado, percorremos a lista de vizinhos desse vértice e realiza-se o pareamento com o primeiro vizinho não pareado encontrado em O(grau(vértice)) no pior caso.

Se o vértice e seu vizinho tiverem sido pareados, incrementamos o número de pareamentos. Caso todos os vizinhos do vértice estejam pareados, passamos para o próximo vértice da ordem ascendente de grau.

Esse processo continua até que todos os vértices tenham sido percorridos na ordem ascendente de grau. Após o algoritmo terminar, ele retorna o número de pareamentos feitos, que não é necessariamente a solução ótima (mas pode ser).

No caso do algoritmo guloso, sua complexidade depende principalmente da ordenação dos vértices, que requer O(N * log(N)) operações. Em seguida, o processo de pareamento envolve percorrer a lista de adjacências, resultando em uma complexidade de O(N + M). Portanto, a complexidade do algoritmo guloso como um todo é

$$O((N + M) + (N * \log(N))) = O(N * \log(N))$$

A solução gulosa é mais rápida em comparação com a solução exata, porém não garante a obtenção da solução ótima. O trade-off está na troca da exatidão do resultado pela velocidade e complexidade de execução.

a. Solução Exata

A solução exata também modela o problema como um problema de grafo. Porém, diferente do algoritmo guloso, modelamos o problema como um problema de fluxo máximo, onde o fluxo máximo do grafo corresponderá também ao número de pareamentos máximo entre vagas e usuários no grafo bipartido.

O grafo da solução exata é uma cópia do grafo da solução gulosa, com a adição de vértices de source (origem) e sink (destino) e arestas adicionais do source para os usuários e vice versa, e também das vagas ao sink e vice versa. Essas adições permitem a aplicação do algoritmo de Ford-Fulkerson para encontrar o número máximo de pareamentos em um grafo bipartido. Assim como na solução gulosa, utilizamos uma lista de adjacências para representar o grafo, implementado como um vetor alocado dinamicamente de vectors.

A escolha entre uma busca em profundidade (DFS) e uma busca em largura (BFS) é determinante para a velocidade de execução e complexidade do algoritmo. No caso deste problema, optou-se por utilizar a DFS. A BFS não se tornou uma opção interessante dado que ela necessariamente visita todos os vértices não visitados a cada nível de distância do source até encontrar o sink. Porém o caminho aumentante não conterá necessariamente todos os vértices de cada nível até o sink. Dessa maneira, todos os outros vértices a cada nível que não fazem parte do caminho aumentante mas que são visitados, são visitados desnecessariamente. Já a DFS não incorre nesse problema pois não visita todos os vértices a cada nível de distância, apenas vai procurando até encontrar o vértice mais profundo, que no caso seria o sink. Assim, a DFS retorna com o caminho aumentante sem ter que visitar todos os outros vértices desnecessariamente. A DFS da forma que foi implementada possui complexidade $O(N + M)$.

O algoritmo de Ford-Fulkerson faz as seguintes operações:

Primeiro, inicia-se com um fluxo nulo em todas as arestas do grafo. Em seguida, fazemos uma chamada à DFS para encontrarmos um caminho aumentante no grafo.

Como sabemos que o caminho aumentante necessariamente possui uma capacidade mínima de 1, atualizamos o fluxo do grafo por esse caminho aumentante e incrementamos o fluxo máximo (o número de pareamentos) do grafo.

Repetimos esse processo de chamar uma DFS para encontrar o caminho aumentante, atualizar o fluxo do grafo e incrementar o fluxo máximo (o número de pareamentos) até não haver mais caminhos aumentantes no grafo.

No final do algoritmo retornamos o fluxo máximo do grafo, que corresponde ao número máximo de pareamentos compatíveis entre usuários e vagas.

A implementação do algoritmo de Ford-Fulkerson também requer estruturas de dados auxiliares, como um vetor para marcar os vértices visitados durante a DFS, e um vector para armazenar o caminho aumentante encontrado.

A complexidade do algoritmo de Ford-Fulkerson depende principalmente do número de caminhos aumentantes encontrados, que é limitado pelo fluxo máximo do grafo. Geralmente isso é indesejado, porém como o fluxo máximo no nosso problema é limitado superiormente pelo mínimo entre o número de usuários e vagas devido ao grafo ser bipartido e ter fluxo de no máximo 1 em cada aresta, torna-se o ideal. Dado que a DFS possui complexidade O(N + M), e seja f o fluxo máximo, a complexidade do algoritmo exato como um todo é de:

$$O(f * (N + M))$$

A solução exata garante a solução ótima, porém é mais lenta em comparação com a solução gulosa. O trade-off está na troca da velocidade e complexidade de execução pela exatidão do resultado.

3. Conclusão

Em conclusão, a resolução do problema de pareamento de usuários e vagas envolveu a implementação de dois algoritmos: um algoritmo guloso e outro exato. A solução gulosa proporciona uma abordagem rápida, mas não garante a solução ótima. Já a solução exata, baseada no algoritmo de Ford-Fulkerson, garante a solução ótima, mas possui complexidade e velocidade de execução menos favorável.

Durante o desenvolvimento dessa solução, foi necessário compreender a modelagem do problema como um grafo bipartido e utilizar estruturas de dados adequadas, como lista de adjacências, mapas, vectors e outros. Também foi necessário entender os benefícios e malefícios de se usar uma BFS ou uma DFS em casos específicos do Ford-Fulkerson. Além disso, a escolha do algoritmo adequado, seja guloso ou de fluxo máximo, dependeu do trade-off entre velocidade de execução e obtenção da solução ótima.

Com base nessas informações, é possível compreender melhor o funcionamento das soluções e tomar decisões informadas sobre qual abordagem adotar dependendo das necessidades específicas do contexto.