UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Universidade Federal de Minas Gerais

Turma: Ciência da Computação **Prof.:** Vinícius

Nome: Rubia Alice Moreira de Souza
Matrícula: 2022043507

Emparelhamento

Belo Horizonte
2023

1. Introdução

O problema a ser resolvido consiste em encontrar a maior quantidade possível de atribuições de pessoas a empregos, sendo que uma pessoa pode ter apenas um emprego e um emprego pode ser ocupado por apenas uma pessoa. Assim, este é um problema clássico chamado de Maximum Bipartite Matching, por estarmos sempre tratando de dois conjuntos disjuntos, os de trabalhadores e de empregos, que se relacionam.

Todo o histórico de desenvolvimento, pode ser acompanhado pelos commits neste repositório: UFMG-ALG-Emparelhamento. Ele apenas se tornará público após a data final de entrega deste trabalho.

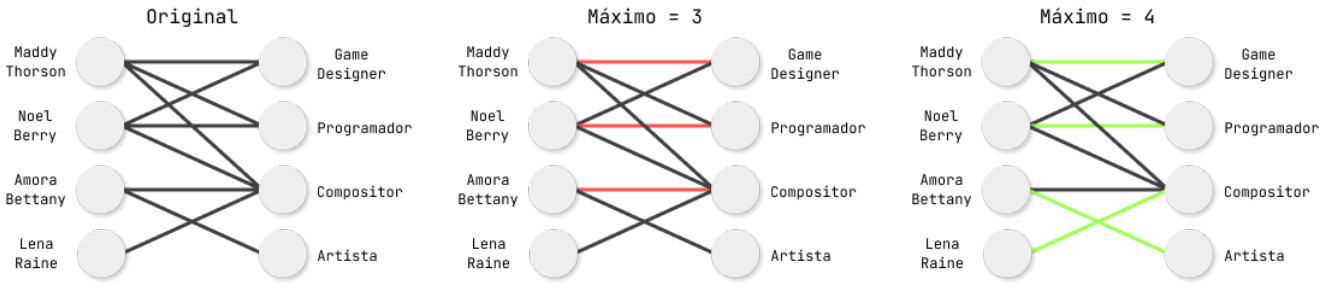
2. Modelagem

O problema foi modelado utilizando grafos de modo que cada vértice representa um trabalhador ou um emprego e as arestas apenas existem entre esses dois tipos. Cada aresta representa uma qualificação de um trabalhador para um determinado emprego.

2.1. Solução Gulosa

Foram feitas duas implementações para o algoritmo guloso. A primeira realiza um processo semelhante a uma BFS. Ela percorre cada vértice de emprego e procura o primeiro trabalhador ligado a ele que está disponível. Então, os dois são agrupados e marcados como “removidos” para não serem visitados novamente.

Nessa primeira implementação, a precisão do algoritmo em relação ao exato foi de 93.5885%, em média, para os 13 casos de testes fornecidos para o trabalho. Os erros do algoritmo acontecem devido às escolhas que impedem agrupamentos melhores no futuro. Um exemplo deste tipo de erro seria em casos como este:



Durante a execução do algoritmo, ele tenta agrupar os vértices apenas na ordem que eles aparecem. Isso fará com que Amora seja atribuída como Compositora. No entanto, essa decisão faz com que a única opção para Lena seja removida. Ao final da execução, o algoritmo irá informar que o máximo de agrupamentos seria 3. Contudo, se selecionarmos a Lena antes da Amora como Compositora teríamos o resultado correto, que é 4.

Com isso em mente, eu implementei um segundo algoritmo que segue a mesma estratégia, mas prioriza aqueles trabalhadores e empregos que possuem menos possibilidades (arestas) para serem agrupados primeiro.

Inicialmente, temos um heap binário mínimo que define a ordem em que os vértices de empregos serão visitados. A cada iteração obtemos o menor elemento do heap e verificamos os trabalhadores associados a ele. Ao encontrar o trabalhador com menos possibilidades disponível para o emprego da iteração atual, agrupamos os dois e marcamos como removidos.

Depois, para cada emprego que o trabalhador selecionado podia desempenhar, reduzimos a quantidade de pessoas aptas a fazê-lo e atualizamos os seus valores no heap. Além disso, para cada trabalhador apto a fazer o emprego agrupado, diminuímos a quantidade de trabalhos que poderiam fazer.

Utilizando essa métrica para selecionar os vértices, o algoritmo foi capaz de ter uma precisão de 100% em relação ao algoritmo exato para os testes fornecidos. No entanto, isso não é uma garantia que ele sempre funcionará com 100% de precisão para TODOS possíveis casos.

Por fim, é importante notar que ambos algoritmos são gulosos, pois, uma vez que um trabalhador e um emprego são agrupados, eles não são considerados novamente durante sua execução. Por causa dessa característica, podem ocorrer agrupamentos que restringem possibilidades futuras e que reduzem a quantidade máxima encontrada de agrupamentos.

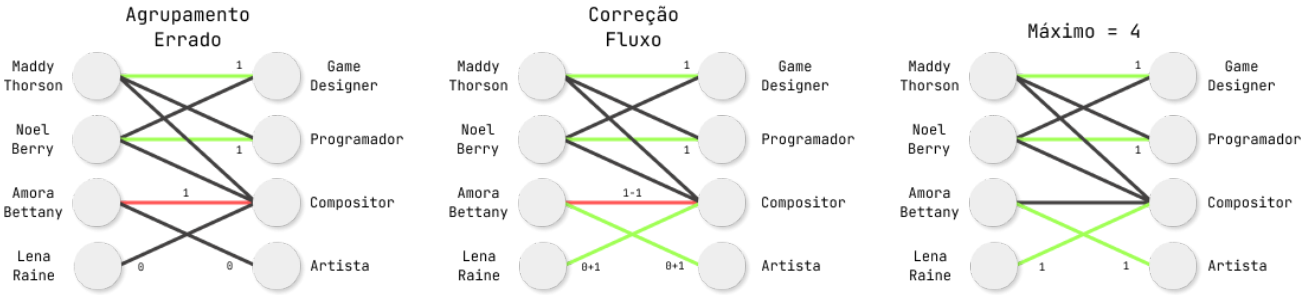
Ambas as implementações estão disponíveis no código fonte do trabalho enviado. A primeira está implementada no método EncontrarMaximumMatchingSemPrioridade(). Já a segunda no método EncontrarMaximumMatchingComPrioridade(). Entretanto, a segunda foi escolhida para ser a implementação final por ter uma precisão maior e uma complexidade não muito superior à primeira. Caso tenha interesse em testar o outro método, basta alterar a chamada na main.

2.2. Solução Exata

A solução exata também foi implementada de duas formas diferentes, sendo que ambas implementações se baseiam no algoritmo de fluxo máximo. Isso pois, por meio dele, é possível se “arrepender” de agrupamentos já feitos ao utilizar arestas residuais. Além disso, as arestas saturadas ao final do algoritmo irão indicar os agrupamentos que foram feitos.

Inicialmente, a fim de aplicar esse algoritmo, primeiro devemos adaptar o grafo original do problema. Primeiro, criamos dois vértices. Um deles será a fonte e o outro o sumidouro. Então, adicionamos uma aresta da fonte para cada trabalhador e uma aresta de cada vértice de emprego para o sumidouro. Depois, definimos que todas as arestas possuem uma capacidade unitária. Isso irá garantir que nenhum fluxo passe pela mesma aresta. Assim, um trabalhador estará ligado a, no máximo, um emprego e um emprego estará ligado a, no máximo, um trabalhador. Por fim, geramos as arestas residuais. Agora, com o novo grafo gerado, podemos executar o algoritmo de fluxo nele.

O primeiro algoritmo que foi implementado foi o de Ford-Fulkerson, utilizando uma DFS para encontrar os caminhos aumentantes. A soma do valor mínimo de cada caminho aumentante encontrado será igual a quantidade máxima de agrupamentos que poderão ser feitos. Ao executar o algoritmo no exemplo anterior, teríamos a seguinte configuração:



Ao tentar enviar fluxo pelo vértice da Lena, será encontrado um caminho aumentante com uma aresta residual que passa pelo emprego de Compositor até a Amora e pelo emprego Artista. Como uma aresta residual foi utilizada no caminho, o fluxo da aresta original será removido, desfazendo o agrupamento. Já as novas arestas percorridas pelo caminho aumentante farão os novos agrupamentos que irão maximizar a quantidade de atribuições.

Além desse, também implementei o algoritmo de Edmonds Karp, que consiste em um algoritmo de fluxo com a mesma lógica, mas que utiliza uma BFS para encontrar o caminho aumentante. Ambos estão disponíveis no código fonte do trabalho. O primeiro está implementado no método EncontrarMaximumMatchingFordFulkerson(). Já o segundo no método EncontrarMaximumMatchingEdmondsKarp(). Entre as duas, a primeira foi escolhida para ser mantida como a implementação final, pois o seu desempenho foi muito superior à segunda, como será explicado na próxima parte.

3. Complexidade

Para a análise de complexidade, podemos considerar o pior caso em que temos um grafo em que todos os trabalhadores podem desempenhar todos os empregos. Assim, se temos “E” empregos e “T” trabalhadores, a quantidade de vértices “n” seria a soma dos dois (n = E + T). Já a quantidade de arestas “m” seria o produto entre eles (m = E * T).

3.1. Algoritmo Guloso

3.1.1. Implementação 1: Sem prioridade

Considerando o primeiro algoritmo de fluxo, o seu comportamento é igual ao de uma BFS, em que verificamos cada vértice e aresta apenas uma vez. Primeiro, temos um custo inicial de O(n) para definir todos os vértices como não visitado. Depois, temos um custo de O(m) para encontrar um trabalhador disponível dentre os vizinhos de um emprego. Assim, a complexidade do algoritmo é O(m + n).

3.1.2. Implementação 2: Com prioridade

Já para a segunda implementação do algoritmo guloso, nós temos o custo extra de manipular o heap mínimo. Tanto a inserção quanto a atualização de elementos terão um custo de O(log(n)) no pior caso. Como devemos inicializar o heap inserindo os vértices, temos um custo de O(n*log(n)).

Além disso, após agrupar um trabalhador a um emprego, devemos decrementar a quantidade de trabalhadores aptos das outras vagas e ajustar o heap, que geraria um custo de O(m*log(n)).

Portanto, a complexidade final será O((m+n)*log(n)). O que faz bastante sentido, pois este algoritmo se assemelha muito ao algoritmo de Dijkstra por utilizar um heap binário mínimo para organizar a ordem que os vértices serão visitados e ter uma estrutura parecida.

3.1.3. Tempo de execução

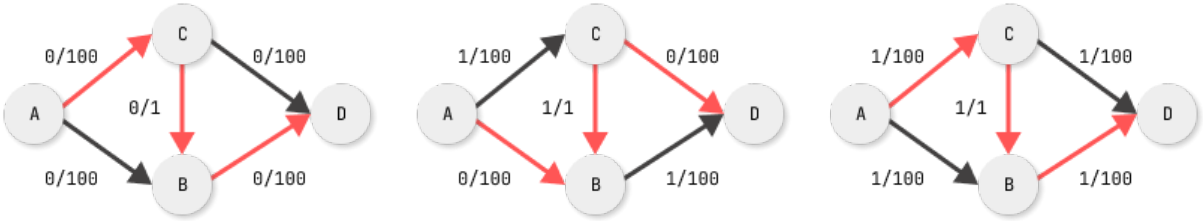
Com relação ao tempo de execução dos casos de teste, ambas as implementações foram capazes de executar todos os 13 casos de teste em sequência com menos de 1 segundo. Considerando que a segunda implementação apresentou uma precisão maior que a primeira e um tempo de execução semelhante, mesmo com o custo logarítmico a mais em sua complexidade, ela foi escolhida para ser mantida como a versão principal para executar os testes do algoritmo guloso no envio final do trabalho.

3.2. Algoritmo Exato

3.2.1. Implementação 1: Ford-Fulkerson

Para o algoritmo de Ford-Fulkerson, temos que a complexidade depende da quantidade de vezes que ele irá iterar para encontrar o fluxo máximo, sendo que cada iteração tem um custo de O(m) para encontrar um caminho aumentante. Assim, a complexidade no pior caso seria O(m * f), sendo “f” o valor do fluxo máximo.

No entanto, sabendo que o pior caso do algoritmo ocorre quando o caminho aumentante fica alternando entre arestas até que fiquem totalmente preenchidas, como no caso abaixo:



É possível notar que para o contexto que temos, este caso não acontecerá de forma tão grave, pois todas as arestas têm capacidade 1. Assim, sabemos que o caminho aumentante irá sempre preencher ou esvaziar completamente uma aresta, impedindo que fique alternando por uma quantidade grande de iterações entre um mesmo conjunto de caminho aumentantes.

3.2.2. Implementação 2: Edmonds-Karp

Já para o algoritmo de Edmonds-Karp, ele consegue reduzir a quantidade de iterações sempre pegando o caminho com menos arestas da fonte até o sumidouro por meio da BFS. Por causa disso, a sua complexidade deixa de depender do valor do fluxo máximo e passa a ser: $O(n * m^2)$.

3.2.3. Tempo de execução

Ao testar ambas as implementações com os casos de testes fornecidos, foi perceptível uma grande diferença no tempo de execução entre o algoritmo de Ford-Fulkerson e Edmonds-Karp.

Enquanto que a primeira implementação foi capaz de executar todos os casos de teste em sequência em 2,21 segundos, a segunda levou cerca de 17,27 segundos nos laboratórios do ICEX.

Assim, para garantir que todos os testes executassem em menos de 1 segundo, o algoritmo de Ford-Fulkerson foi escolhido como o principal para calcular o resultado exato na versão final do código enviado do trabalho.

Acredito que o principal motivo para o melhor desempenho do Ford Fulkerson é o fato das arestas terem capacidade máxima de 1, evitando que seu pior caso ocorra como dito anteriormente.

4. Comparação Algoritmo Guloso e Exato

Ao avaliar a precisão dos algoritmos gulosos em relação ao exato, é possível notar que a sua solução é bem próxima da ótima, com uma margem de erro de 6.4115%. Isso torna-se bem interessante ao comparar a diferença de complexidade entre eles.

Uma perda de precisão de, aproximadamente, 6.5% para conseguir reduzir a complexidade do problema a um nível linear pode ser muito vantajosa a depender do contexto da aplicação.

Além disso, caso seja necessário um pouco mais de precisão, mas ainda garantir um desempenho rápido, aplicar a métrica de priorizar os empregos e trabalhadores com menos possibilidades apenas incrementa a complexidade por um fator logarítmico, garantindo uma performance que ainda é mais rápida que o algoritmo exato.

Portanto, utilizar os algoritmos gulosos pode ser uma escolha boa para projetos que podem ter uma margem de erro, mas precisam de ser bem eficientes.

Por outro lado, para aplicações críticas que não podem dar margem a erro, o ideal é utilizar um algoritmo de fluxo a fim de encontrar a solução exata para todos os casos. Por mais que haja uma métrica, sempre é possível que o algoritmo guloso tome uma decisão errada que afete o resultado final negativamente.