# Trabalho Prático 2

**Gabriela Tavares Barreto**

Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil

gbarreto@ufmg.br

## 1 Introdução

Um grafo bipartido é um tipo especial de grafo em que os vértices podem ser divididos em dois conjuntos disjuntos($A$ e $B$ por exemplo), de forma que todas as arestas do grafo conectem um vértice de $A$ a um vértice de $B$. O objetivo desse trabalho prático consistia em resolver o problema do emparalhamento máximo em um grafo bipartido, ou seja, encontrar o maior conjunto de arestas não adjacentes no grafo.

Esse problema foi apresentando na seguinte forma: uma empresa possui uma lista de pessoas desempregadas e uma lista de vagas de empregos. De que maneira é possível distribuir as vagas de forma a minizar as pessoas que ficam sem emprego? A resposta deste problema se encontra em modelar a relação emprego-desempregado como um grafo bipartido, e executar um algoritmo de emparelhamento máximo no grafo que simboliza essas relações. Mais detalhes sobre a solução a seguir.

## 2 Método

O programa foi desenvolvido na linguagem C++, e compilado pelo G++ da GNU Compiler Collection, usando o WSL. O computador utilizado tem como sistema operacional o Windows 11, com 12 GB de RAM.

### 2.1 Solução

A especificação do trabalho pedia duas soluções, uma gulosa(que fosse subótima) e uma ótima. Para a solução ótima foi implementado o algoritmo de Edmonds-Karp. A seguir serão descritas em alto nível como cada uma funciona.

#### 2.1.1 Método Guloso

Um algoritmo guloso é uma abordagem de resolução de problemas que faz em cada etapa da resolução escolhas locais ótimas, na esperança de obter uma solução global ótima. Como um algoritmo guloso toma decisões com base na informação disponível no momento atual, sem considerar as consequências futuras, nem sempre ele gera soluções ótimas.

Para o problema apresentado, foi implementado um algoritmo guloso que trabalha de acordo com o grau de cada vértice. Isso é feito da seguinte forma, dado um grafo de entrada $G$:

1. Para cada vértice $v$ de $G$, é contabilizado a quantos outros vértices ele se conecta, ou seja, qual é seu grau.
2. Seguindo a ordem crescente de grau, cada vértice é visitado.
3. Ao visitar um vértice $v$, é verificado dentre os seus vértices vizinhos qual vértice $u$ que tem o menor grau.
4. A aresta que liga $v$ a $u$ é adicionada a solução. Nota-se que uma aresta é adicionada a solução se e somente se ambos vértices que a compõe não tiverem sido usados na solução ainda.

Esse método se caracteriza como guloso porque ao verificar os graus dos vértices um por um, faz escolhas ótimas locais, que não consideram o grafo como um todo, sendo que podem haver outras combinações de arestas que, mesmo envolvendo vértices com graus maiores, resultem em um emparelhamento melhor.

### 2.1.2 Método Exato: Eomonds-Karp

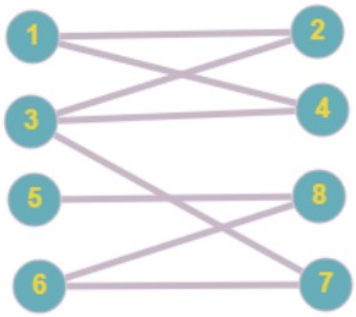


Figura 1: Grago Bipartido $G$

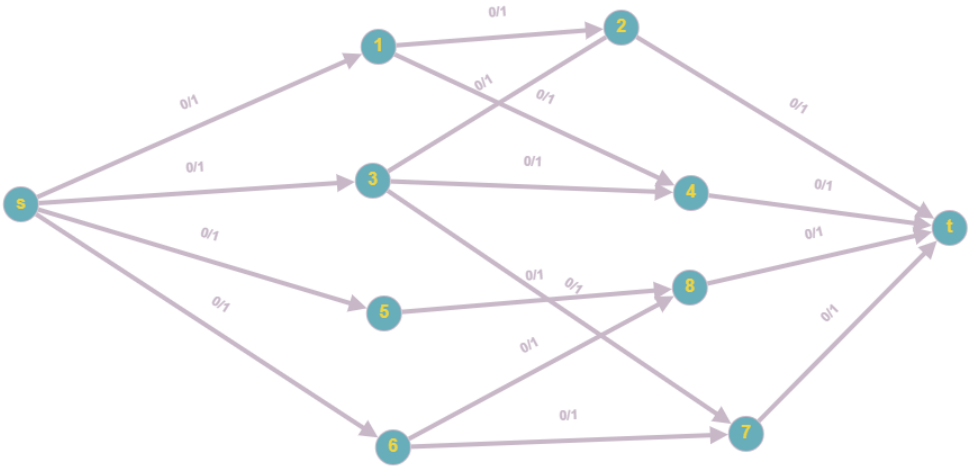


Figura 2: Grafo Residual $G_r$

O algoritmo de Edmonds-Karp é um algoritmo de fluxo máximo que se baseia no algoritmo Ford-Fulkerson. Ele utiliza a estratégia de busca em largura (BFS) para encontrar caminhos aumentantes no grafo residual. O algoritmo de Edmonds-Karp segue os seguintes passos:

1. Inicialize o fluxo máximo como 0.
2. Enquanto houver um caminho aumentante no grafo residual:
   (a) Encontre um caminho aumentante usando uma busca em largura (BFS) no grafo residual.
   (b) Calcule a capacidade residual mínima ao longo do caminho aumentante.
   (c) Atualize o fluxo máximo somando a capacidade residual mínima ao fluxo máximo atual.
   (d) Atualize as capacidades residuais das arestas ao longo do caminho aumentante.
3. Retorne o fluxo máximo.

O algoritmo de Edmonds-Karp não é especificamente projetado para resolver o problema do emparelhamento máximo em grafos bipartidos. No entanto, é possível adaptar o algoritmo para encontrar o emparelhamento máximo usando o conceito de fluxo máximo.

Para aplicar o algoritmo de Edmonds-Karp no problema do emparelhamento máximo, podemos representar o grafo $G$ bipartido nas partições $A$ e $B$ como um grafo residual. Além de ter os vértices e arestas de $G$, o grafo residual $G_r$ tem também os vértices $s$(fonte) e $t$(sumidouro). Tal que $s$ tem arestas direcionadas para os vértices da partição $A$ e os vértices de $B$ tem arestas extras direcionadas a $t$. As capacidades das arestas no grafo de fluxo são definidas como 1, indicando que apenas um emparelhamento é permitido entre os vértices. Um exemplo dessa representação está ilustrado nas figuras 1 e 2 dessa subseção.

Após a execução do algoritmo de Edmonds-Karp no grafo residual, o emparelhamento máximo pode ser obtido verificando as arestas em $G_r$ que também existem em $G$ e que têm fluxo igual a 1. Cada uma dessas arestas representa uma aresta no emparelhamento máximo.

# 3 Complexidade

Foi escolhido fazer a representação do grafo na forma de uma lista de adjacências, por essa estrutura fazer um uso mais eficiente de memória em relação a uma matriz de adjacências. A lista requer apenas o armazenamento das arestas existentes no grafo. Além disso, por ser uma representação mais flexível, ela permite armazenar outras informações sobre o grafo com facilidade. Isso foi útil na resolução do problema, pois o método guloso usa os valores pré-computados de grau dos vértices, e o Edmonds-Karp verifica a capacidade das arestas para funcionar.

Ademais, a lista de adjacências tem complexidade $O(d(v))$ para verificar as adjacências de um vértice $v$, o que é potencialmente menor que $O(V)$, que é a complexidade de fazer o mesmo em uma matriz de adjacências, principalmente no caso de grafos esparsos. Portanto, a escolha dessa representação contribui para otimizar a complexidade do algoritmo de pareamento.

## 3.1 Método Guloso

Analisando o método guloso por partes:

1. Construção da fila de prioridades:
   - O laço `for` para adicionar os vértices ao heap mínimo percorre todos os vértices do grafo uma vez, resultando em uma complexidade de $O(V)$, onde $V$ é o número de vértices.
   - A operação `push` na fila de prioridades tem uma complexidade de $O(\log V)$ para cada elemento inserido.
   - Portanto, a construção da fila de prioridades tem uma complexidade total de $O(V \log V)$.

2. Iteração principal do algoritmo:
   - O laço `while`, que faz o desenfileiramento do heap, executa no máximo $V$ vezes, pois a fila de prioridades pode conter no máximo $V$ elementos.
   - A cada iteração do `while` é removido um elemento do heap mínimo, fazendo com que ele tenha que se reordenar, e isso tem complexidade $O(\log V)$.
   - Dentro do while, o loop `for` percorre os vértices vizinhos do vértice atual $v$, buscando o vizinho de menor grau. Portanto, a complexidade do laço `for` é $O(d(v))$, onde $d(v)$ é o grau daquele vértice.
   - Assim a complexidade total do `while` é

$$\sum_{n=1}^{V} \log V + d(n) = V \log V + 2E = O(V \log V + E)$$

3. Outras operações:
   - As operações de marcação de vértices e incremento do contador de arestas têm complexidade $O(1)$.

Assim, podemos afirmar que a complexidade total do algoritmo é:

$$O(V \log V) + O(V \log V + E) + O(V) = O(V \log V)$$

Portanto, a complexidade do algoritmo guloso apresentado é $O(V \log V)$.

### 3.2 Busca em largura

Analisando por partes:

1. Inicialização: A inicialização do algoritmo envolve a criação de vetores e estruturas de dados auxiliares, como o vetor de visitados e a fila. Essas operações têm complexidade O(V), onde V é o número de vértices do grafo residual.
2. Loops: temos dois loops, um `while`, que ocorre em função do número de elementos na fila, e o `for` que perorre os vizinhos de cada vértice. Portanto a complexidade desses loops aninhados é:

$$\sum_{i=1}^{V} 1 + d(v) = V + E = O(V + E)$$

3. Operações adicionais: As operações de marcação de vértices e atualização do vetor mãe tem complexidade $O(1)$.

Portanto, a complexidade do algoritmo é:

$$O(V) + O(V + E) = O(V + E).$$

### 3.3 Edmonds-Karp

Analisando a complexidade em partes:

1. Inicialização: A inicialização do algoritmo envolve a criação de estruturas de dados auxiliares, como o vetor mae. Essa operação tem complexidade $O(V)$ onde $V$ é o número de vértices do grafo residual.

2. Loop principal: O loop principal é executado enquanto houver caminho aumentante encontrado pelo algoritmo BFS. A complexidade desse loop depende do número de iterações, que pode ser no máximo $E$, já que a complexidade do BFS tende a ser dominada pelo número de arestas.
3. Loop interno `while`: para cada caminho aumentante encontrado, suas arestas tem as capacidades atualizadas. Para cada aresta $uv$ do caminho aumentante, são percorridos os vizinhos de $u$ até encontrar o vértice $v$ para atualizar a capacidade, e percorre-se também todos vizinhos de $v$ até achar $u$ e atualizar novamente a capacidade. Esse loop tem uma complexidade $O(d(v)+d(u))$. Levando em conta os dois loops aninhados, obtem-se a seguinte complexidade:

$$\sum d(v) + d(u) = V(2E + 2E)$$

Por fim, a complexidade do algoritmo é dada por:

$$E.(V.E) = O(V.E^2)$$

# 4 Análise de resultados

De acordo com os resultados encontrados na análise de complexidade, podemos afirmar que o método guloso é assintoticamente menos custoso que o método exato, já que:

$$O(V \log V) < O(V.E^2)$$

Por mais que a solução subótima nem sempre dê uma resposta exata, ela gera uma resposta próxima do valor esperado, e em um tempo relativamente menor. No contexto da vida real, como no da empresa apresentada, pode ser mais interessante usar o algoritmo guloso para parear os usuários aos empregos, visto que ele será mais veloz e menos custoso para a empresa implementar. Agora, se o enfoque é garantir o melhor paramento possível entre usuários e empregos, para otimizar o atendimento do serviço da LinkOut, o algoritmo de Edmonds-Karp também é uma opção válida.

De forma a deixar mais evidente a diferença entre os algoritmos, pode-se verificar a tabela abaixo, gerada a partir dos resultados dos casos de teste disponibilizados.

| **Guloso** | **Exato** | **% de proximidade** |
|---|---|---|
| 4 | 5 | 80.0000% |
| 4 | 4 | 100.0000% |
| 742 | 845 | 87.8100% |
| 684 | 700 | 97.7100% |
| 850 | 884 | 96.1500% |
| 1581 | 1627 | 97.1700% |
| 4 | 4 | 100.0000% |
| 8 | 8 | 100.0000% |
| 3 | 5 | 60.0000% |
| 24 | 25 | 96.0000% |
| 21 | 22 | 95.4500% |
| 19 | 19 | 100.0000% |
| 77 | 117 | 65.8100% |
| 308 | 330 | 93.3300% |
| **Média** | | 90.6735% |

Ainda sobre o método guloso, embora essa abordagem heurística seja rápida e simples, ela pode levar a resultados subótimos ou ficar presa em mínimos locais. É necessário avaliar cuidadosamente o problema em questão e considerar os trade-offs entre eficiência e precisão para decidir se o algoritmo guloso é a melhor escolha frente a um algoritmo exato.

## 5 Instruções de compilação e execução

Para compilar o programa, deve-se seguir os seguintes passos:

- Acessar o diretório TP02-Template-CPP;
- Utilizando um terminal, digitar make, de forma a gerar o arquivo executável **tp2**;
- Para a execução, chamar o TP0 seguido de $<$ e passando um arquivo de entrada do tipo .txt com os dados do grafo como parâmetro.