# Editorial TP 02 (Trabalho Prático 02) - Algoritmos I

Victor Deluca Almirante Gomes

7 de julho de 2022

## 1 Introdução

Este documento é um compilado de informações associadas ao Trabalho Prático 2, e inclui informações sobre o problema e as possíveis soluções esperadas, bem como as potenciais complicações de cada uma. Os casos de teste serão disponibilizados em um documento em separado.

### 1.1 Objetivos do trabalho

Dando continuidade aos conceitos abordados no TP01, este trabalho foi elaborado com o propósito principal de incentivar a criatividade da turma. O objetivo foi testar as capacidades de cada estudante para adaptar algoritmos e paradigmas aprendidos em aula para aplicações menos triviais - Sem, é claro, exagerar muito no nível de dificuldade.

Por este motivo, foi escolhido um problema que pode ser resolvido de várias (*Várias*) formas diferentes. O interessante do problema abordado por este trabalho prático é que (quase) todas as soluções possíveis utilizam algum algoritmo estudado no decorrer da disciplina, e até mesmo as exceções podem ser associadas ao paradigma guloso ou de divisão e conquista. Contudo, como iremos ver a seguir, nenhuma dessas soluções é uma aplicação trivial, e cada uma possui seus próprios *caveats*, de forma que este problema é perfeito para os propósitos deste trabalho.

Tendo em vista que o propósito do trabalho foi exercitar a criatividade, todas as possíveis soluções foram aceitas - inclusive soluções por força bruta, na falta de algo melhor. Pelo mesmo motivo, até mesmo soluções incorretas não sofreram penalidades muito altas, haja visto que os casos de teste providenciados pelo enunciado totalizaram 40% do *fator parcial de implementação* (E o terceiro caso de teste era extremamente fraco, permitindo que algumas soluções heurísticas retornassem respostas corretas e que soluções por força bruta terminassem dentro do tempo limite).

## 2 Recapitulação do problema

Uma série de desastres naturais deixou várias cidades no estado de Minas Específicas em situação de emergência. A situação é tão grave que a maioria das rodovias do estado foram destruídas ou sofreram graves desgastes, de forma que é impossível atravessá-las de forma segura sem respeitar um certo limite de peso. Esta situação é um grave obstáculo para a ONG de ajuda humanitária Luz Vermelha, que busca levar suprimentos às regiões mais necessitadas.

Desta forma, o objetivo deste trabalho é, dados pares de cidades $u$ e $v$ na malha rodoviária do estado, com $n$ cidades e $m$ rodovias interligando pares de cidades - onde cada rodovia possui um limite de peso - encontrar o máximo de peso que pode ser transportado de $u$ até $v$ por um único caminhão.

Formalmente, dizemos que um caminho $P = [p_1, p_2, ..., p_k]$ (Onde $p_1 = u$ e $p_k = v$) de $u$ a $v$ pode transportar um peso máximo de $min(w(p_i, p_{i+1}))$, onde $w(i, j)$ indica o peso de uma aresta $(i, j)$. Chamamos esse peso $min(w(p_i, p_{i+1}))$ de "gargalo"do nosso caminho. Desta forma, o objetivo deste trabalho é especificar um algoritmo que encontre $max(min(w(p_i, p_{i+1})))$.

# 3 Soluções possíveis

Este problema é um problema clássico da literatura, conhecido por diversos nomes, incluindo *"Caminho do Gargalo Máximo"*, *"Problema do Caminho mais Largo"* e *"MaxiMin"* (Aludindo à sua variante mais conhecida, *"MiniMax"*). Esta seção é destinada ao compartilhamento e descrição das soluções mais comuns, bem como algumas de suas variantes. Seu objetivo é enriquecer o conhecimento de quaisquer estudantes que se interessem pela leitura deste documento.

## 3.1 Preliminares

Todas as soluções para esse problema envolvem a mesma modelagem: A malha rodoviária deve ser representada como um grafo $G(V, E)$, sob a qual as consultas são respondidas uma a uma.

Para os propósitos desta seção, considere a seguinte terminologia:
$V$ e $E$ são o conjunto de vértices e arestas no grafo, respectivamente;
$|V|$ e $|E|$ são o número de vértices e arestas no grafo, respectivamente;
$Q$ é o número de consultas realizadas;
$u$ e $v$ são os vértices de origem e destino de uma consulta, respectivamente;
Dada uma aresta $(i, j)$, $w(i, j)$ indica a capacidade da aresta $(i, j)$.
Dado um caminho $P = \{i_0, i_1, ..., i_k\}$, $w(P)$ indica o gargalo máximo de P.

## 3.2 Soluções que não funcionam

Algumas soluções que realmente não funcionam para os propósitos deste trabalho são:

- **Fluxo Máximo:** Embora o conceito de capacidades possa remeter a um problema de fluxo máximo, a limitação de *um único caminhão por expedição* nos impede de utilizar esse algoritmo. Não existem modificações no algoritmo de fluxo máximo que o forcem a retornar o fluxo ótimo de um único caminho (Ou melhor, você teria que modificá-lo tanto que ele deixaria de ser um algoritmo de fluxo máximo e passaria a ser uma das soluções descritas nas seções abaixo).
- **BFS/DFS:** Da mesma forma que a ideia anterior, o uso de uma BFS/DFS, por si só, não garante que chegaremos a uma solução ótima;

## 3.3 Força Bruta

Geralmente, a solução mais óbvia para problemas de otimização é um algoritmo de força bruta. Um algoritmo de força bruta irá explorar, uma por uma, todas as soluções possíveis para o problema, e então escolher a melhor entre elas. No contexto deste trabalho, uma solução por força bruta consiste em enumerar todos os caminhos possíveis de $u$ até $v$.

Trivialmente, esse algoritmo irá nos gerar a solução ótima, uma vez que todas as soluções possíveis são exploradas. A análise de complexidade dessa solução é igualmente trivial. O pior caso ocorre em um grafo completo, isto é, quando o número de arestas é $|V| * (|V| - 1)$. O primeiro elemento do caminho pode ser escolhido de $|V| - 2$ formas (Qualquer vértice excluindo a fonte e o destino), o segundo de $|V| - 3$ formas e assim por diante.

Existem duas formas de implementar esta solução: A primeira é por meio de **recursão**, descrita pelo pseudocódigo na figura 1. Criamos uma função recursiva que, a cada passo $i$, escolhe uma aresta $(p_i, j), j \notin P$ para acrescentar ao caminho $P$ atual. A Figura 2 ilustra o processo para encontrar o caminho de 1 a 4 em um grafo completo com 4 vértices.

```
1  int generate_all_paths(current_path):
2      if connects(u,v,current_path):
3          return bottleneck(current_path)
4
5      i <- last_element_in_path(current_path)
6      ans <- 0
7      for j in adj(i):
8          if(not_in_path(current_path,j)):
9              ans <- max(ans,generate_all_paths(current_path + { j })
10
11     return ans
```

Figura 1: Pseudocódigo para o algoritmo de força bruta recursivo

A segunda forma é por meio de uma solução **iterativa**. Criamos uma fila de caminhos e, a cada iteração, seguimos os seguintes passos: Selecionamos o caminho $P$ no topo da fila; Removemos $P$ da fila; Criamos caminhos $P^j = P + (P_i, j)$ para cada aresta adjacente a $P_i$; Acrescentamos todos os caminhos $P^j$ à fila. Note que a ordem de processamento dos caminhos não importa, de forma que poderíamos utilizar uma pilha com o mesmo efeito.

Em ambos os casos, representamos cada caminho $P$ como um vetor de vértices (Ou arestas, como preferir), de forma que todos os elementos adjacentes em $P$ são adjacentes no grafo original (Isto é, $(p_i, p_{i+1})$ é sempre uma aresta no grafo). Também em ambos os casos, um caminho acaba quando chegamos ao vértice destino $v$. Nesse caso, avaliamos o *gargalo* daquele caminho e atualizamos a solução ótima com $min(OtimoAtual, GargaloEncontrado)$.

Ambas as técnicas são equivalentes e ambas irão enumerar todos os caminhos possíveis de $u$ até $v$. A única diferença entre ambas as técnicas é que a segunda implementação, em sua forma mais simples, requer que um vetor de caminhos seja armazenado. Esse vetor irá receber um número beeem grande de elementos, resultando em uma complexidade de $O(N!)$ não somente em tempo, mas também em espaço.

Prós:

- Solução mais intuitiva do trabalho;
- A prova de corretude é completamente trivial;
- Pode ser melhorado com *heurísticas* de forma a obtermos a solução ótima mais rapidamente;
- Apesar do custo elevado, o algoritmo ainda pode ser executado em tempo hábil para grafos esparsos (Isto é, com poucas arestas) - onde o pior caso é matematicamente impossível de ocorrer;

Contras:

- O custo assintótico do algoritmo aumenta exponencialmente com o número de vértices no grafo (Basicamente o pior custo assintótico que um algoritmo pode ter).
- Note que, mesmo em grafos mais esparsos, o algoritmo ainda se torna inutilizável rapidamente conforme o número de vértices aumenta;
- Não se associa a nenhum paradigma visto em aula, contradizendo a minha afirmação na introdução ;-;

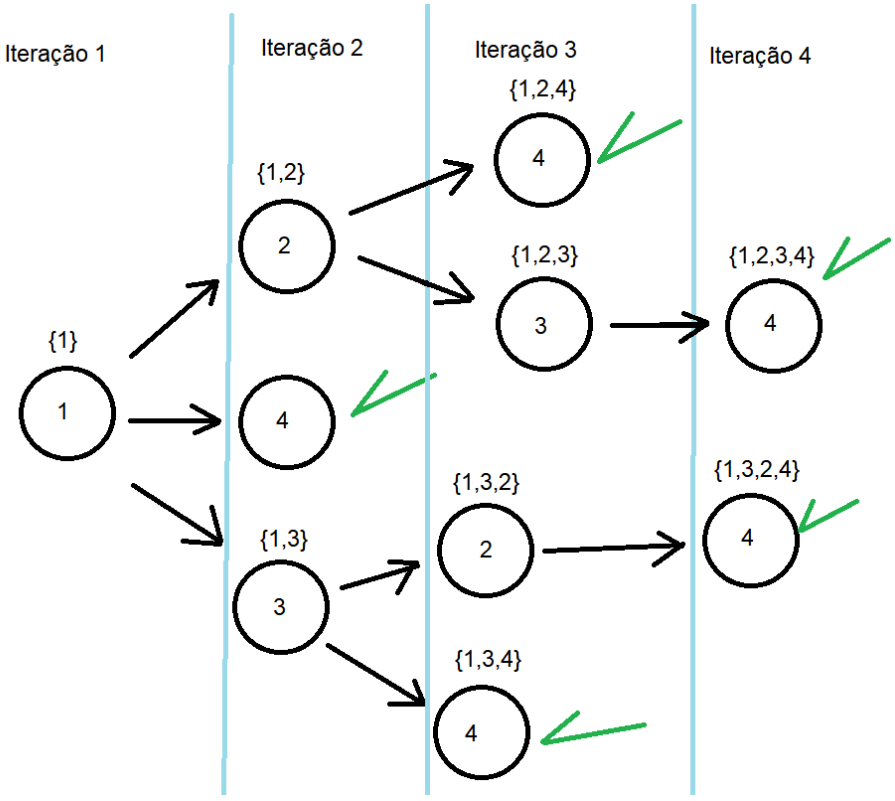


Figura 2: Árvore de recursão para encontrar um caminho de 1 até 4 em um grafo completo com 4 vértices

Na prática, soluções por força bruta dificilmente são utilizadas por conta própria. Isso porque, geralmente, existem algoritmos mais eficientes para resolver o problema. Instâncias em que o algoritmo de força bruta pode ser útil incluem:

- **Para debugar implementações de algoritmos mais complexos** - Naturalmente, um algoritmo de força bruta sempre irá retornar a solução correta, e é relativamente simples de implementar. Por esse motivo, quando não temos casos de teste prontamente disponíveis, podemos elaborar nossos próprios casos de teste e utilizar o algoritmo de força bruta para gerar as soluções esperadas para esses casos de teste.
- **Como *framework* para algoritmos mais eficientes:** Embora algoritmos de força bruta sejam ineficientes, podemos utilizar sua lógica como base para gerar soluções melhores, conforme veremos na subseção a seguir.

Complexidade assintótica: $O(|V| * |V|!)$ por consulta, totalizando $O(Q * |V| * |V|!)$
Número de soluções enviadas: 5
Casos de teste aceitos: 3
Veredito final: 60% de acurácia, tempo limite excedido

Observação: A simples argumentação "O algoritmo explora todas as soluções possíveis"é suficiente para garantir que o algoritmo retorna a solução ótima. É por esse motivo, inclusive, que dizemos que a justificativa é trivial.

### 3.3.1 Força Bruta Otimizada

Uma melhoria natural para o algoritmo apresentado na seção anterior seria evitar a exploração de soluções que nós *sabemos* que não podem ser ótimas. Suponha que nós conhecemos um caminho, não necessariamente ótimo, $O_{tmp}$, com gargalo $G(O_{tmp})$. Naturalmente, nenhum caminho $P$ que possui uma aresta $(i, j)$ tal que $w(i, j) < w(G(O_{tmp}))$ poderá ser ótimo. Dessa forma, podemos simplesmente ignorar caminhos que possuem tais arestas. A Figura 3 descreve nosso código otimizado.

```
1  int generate_some_paths(current_path):
2      if connects(u,v,current_path):
3          return bottleneck(current_path)
4
5      i <− last_element_in_path(current_path)
6      ans <− 0
7      for j in adj(i):
8          if(w(i,j) > ans):
9              if(not_in_path(current_path,j)):
10                 ans <− max(ans,generate_some_paths(current_path + { j })
11
12     return ans
```

Figura 3: Pseudocódigo para o algoritmo de força bruta otimizado recursivo

Vale ressaltar que essa versão do algoritmo ainda é exponencial. Seu pior caso corre quando o gargalo está no final da árvore de recursão, isto é, quando os vértices de menor capacidade se conectam à cidade destino $v$. Nesse caso, o algoritmo precisará explorar a árvore de recursão inteira, e nossa pequena otimização não terá efeito algum.

Na área de Otimização Combinatória, esse tipo de algoritmo é conhecido como Branch-and-Bound: Se sabemos que uma solução irá gerar um valor inferior ao ótimo, para quê processá-la? Embora não possuam efeito na complexidade assintótica do problema, algoritmos de Branch-and-Bound costumam ter bons resultados na prática (Quanto mais otimizações você conseguir aplicar, melhor). Portanto, é um paradigma que vale a pena conhecer, pelo menos.

Prós:

- Solução relativamente intuitiva a partir da solução por força bruta;
- (Provavelmente) Passa em todos os casos de teste;

Contras:

- Custo ainda exponencial;
- Eficácia ainda questionável na vida real (Algoritmos de Branch-and-Bound eficientes costumam ser bem mais elaborados);
- Análise de complexidade confusa;

Complexidade assintótica: $O(|V| * |V|!)$ por consulta, totalizando $O(Q * |V| * |V|!)$
Número de soluções enviadas: 2
Casos de teste aceitos: 5
Veredito final: 100% de acurácia

Observação: Alguns trabalhos argumentaram que esse algoritmo possui tempo polinomial, pois a cada iteração removemos uma solução candidata (Note que existem no máximo $O(|E|)$ valores diferentes para a solução do problema). Essa argumentação é incorreta, pois não leva em conta a altura da árvore de recursão - Como vimos alguns parágrafos atrás, se todos os vértices de capacidade mínima se conectam à cidade destino, os "cortes"na árvore de recursão são irrelevantes. Contudo, a argumentação foi suficiente para me enganar durante o período de correção e, portanto, não sofreu penalidades.

### 3.3.2 Remoção Iterativa de Arestas

Como discutido na subseção anterior, nenhum caminho $P$ que possui uma aresta $(i, j)$ tal que $w(i, j) < w(G(O_{tmp}))$ poderá ser ótimo. Anteriormente, utilizamos este conceito para eliminar caminhos que utilizem a aresta $(i, j)$.

Uma solução ainda mais esperta baseada no mesmo preceito é simplesmente eliminar a aresta $(i, j)$ do grafo. A ideia é simples: Todo caminho que contém a aresta $(i, j)$ possuirá um gargalo de pelo menos $w(i, j)$. Logo, uma vez que identificamos $w(i, j)$ como um possível gargalo, não há necessidade de manter $(i, j)$ no grafo.

Dessa forma, podemos formular um algoritmo iterativo baseado em dois passos: Gerar um caminho arbitrário $P$, de $u$ até $v$; Remover a aresta gargalo $G(P)$ do grafo. Repetimos o processo até que não seja mais possível encontrar caminhos de $u$ até $v$. O pseudocódigo na Figura 4 descreve o processo proposto:

```
1 E' <- E
2 ans <- 0
3 while(path_exists(u,v,E')):
4     P = generate_path(u,v,E')
5     (i,j) = get_bottleneck(P)
6     ans <- max(ans, w(i,j))
7     E' <- E' - (i,j)
```

Figura 4: Pseudocódigo para o algoritmo de remoções iterativas

A função $generate\_path(u, v, E')$ retorna um caminho de $u$ até $v$ utilizando apenas as arestas em $E'$, e pode ser implementada como uma BFS ou uma DFS. Não estamos interessados em um caminho ótimo - qualquer caminho é suficiente.

A análise de complexidade dessa função é bem direta. O loop externo 'while' será executado um máximo de $O(|E|)$ vezes, enquanto as funções $path_exists()$ e $generate_path()$ podem ser implementadas em $O(|V| + |E|)$, resultando em um custo de $O(|E| * (|V| + |E|))$.

Prós:

- Solução relativamente intuitiva;
- Custo polinomial;

Contras:

- Análise de complexidade menos trivial: Requer um limite para o número de vezes que o "while"externo será executado;
- Menos eficiente que quase todas as outras soluções apresentadas em seções posteriores;

Complexidade assintótica: $O(|E| * (|V| + |E|))$ por consulta, totalizando $O(Q * (|E| * (|V| + |E|)))$

Número de soluções enviadas: 4
Casos de teste aceitos: 5
Veredito final: 100% de acurácia

### 3.4 Dijkstra modificado

Possivelmente a solução mais popular para o problema. É a solução mais difundida online, é a solução descrita no GeeksForGeeks e, coincidentemente, foi a solução mais aplicada no decorrer do trabalho. A ideia básica da solução consiste em alterar a condição de relaxamento do algoritmo de Dijkstra de $min(distance[i] + w(i,j), distance[j])$ para $max(min(capacity[i], w(i,j)), capacity[j])$.

A intuição por trás da solução é simples: Da mesma forma que o Dijkstra normal armazena um vetor de distâncias *distance*[], nosso Dijkstra modificado armazenará um vetor de gargalos *capacity*[], onde *capacity*[$i$] indica o gargalo máximo encontrado em todos os caminhos de $u$ até $i$. Para obter o próximo vértice a ser processado, utilizamos uma *max_queue*, análoga à *min_queue* utilizada no Dijkstra clássico.

Contudo, provar que o algoritmo de fato funciona dessa forma é uma tarefa um pouco mais complexa. Para os propósitos deste trabalho, foram aceitos, entre outros, os seguintes argumentos:

- Demonstração (in)formal por indução;
- Demonstração (in)formal por contradição;
- Demonstração de que a alteração na condição de relaxamento não afeta a capacidade do algoritmo de gerar uma solução correta;
- "A demonstração é análoga à demonstração de corretude do Dijkstra normal". De fato, se você substituir a nossa nova condição de relaxamento na prova de corretude do Dijkstra, a prova é basicamente a mesma (Embora fazer essa afirmação demande um nível decente de autoconfiança);

Uma possível demonstração, por indução, é descrita a seguir. Seja $S$ o conjunto de vértices já processados pelo algoritmo de Dijkstra (Isto é, todos os vértices já escolhidos pela fila de prioridades). Vamos mostrar que *capacity*[$i$] contém o gargalo máximo do vértice fonte $u$ para qualquer vértice $i \in S$ (Note que, se demonstrarmos essa afirmação, é garantido que *capacity*[$v$] contém o gargalo máximo para o vértice destino $v$ ao final da execução do algoritmo).

Caso base: Início do algoritmo, $S = \{u\}$,apenas o vértice fonte foi processado. $capacity[u] = \infty$ está trivialmente correto, pois podemos transportar quanta carga desejarmos do vértice fonte até ele próprio.

Passo indutivo: Suponha agora que, em uma dada iteração, *capacity*[$i$] contém o gargalo máximo para qualquer vértice $i \in S$. A fila de prioridade escolhe agora um novo vértice, $j$, para processar. Temos que mostrar que *capacity*[$j$] também contém o gargalo máximo entre todos os caminhos de $u$ até $j$.

Suponha, por absurdo, que exista um caminho estritamente mais curto, digamos $P0$, de $u$ até $j$ que o caminho tomado pelo algoritmo (Que chamaremos de $P$). O caminho não pode consistir apenas em vértices de $S$, pois nossa fila de prioridade escolhe, por definição, o caminho de maior gargalo entre os caminhos já registrados pelo algoritmo.

Pela definição de $P0$, temos $w(P0) > capacity[j]$. Seja $(k,l)$, então, a primeira aresta em $P0$ tal que $l \notin S$, e seja também $P0_l$ um subcaminho de $P0$ que termina em $l$. Naturalmente, $w(P0_l) > w(P0)$. Como $k$ ainda está em $S$, então *capacity*[$k$] contém (Pela hipótese indutiva) o gargalo máximo de $u$ até $k$. Logo, $min(capacity[k], w(k,l)) \geq w(P0_l)$. Por sua vez, $l$ é adjacente a $k$ e, portanto, foi processado pelo algoritmo com $capacity[l] = max(capacity[l], min(capacity[k], w(k,l)))$. Logo, $capacity[l] \geq min(capacity[k], w(k,l))$. Obtemos assim a série de inequações: $capacity[l] \geq min(capacity[k], w(k,l)) \geq w(P0) > capacity[j]$.

Finalmente, uma vez que $j$ foi escolhido antes de $l$ pela fila de prioridades, temos que $capacity[j] \geq capacity[l]$. Dessa forma, obtemos $capacity[j] > capacity[j]$, uma visível contradição. Portanto, o caminho $P0$ não pode existir.

A prova descrita nessa seção foi formulada com base na seguinte prova (Baseada no Dijkstra original): `https://web.engr.oregonstate.edu/~glencora/wiki/uploads/dijkstra-proof.pdf`.

Prós:

- Solução relativamente intuitiva;
- Bastante material online;
- Análise de complexidade análoga ao Dijkstra original;

Contras:

- Demonstração de corretude pouco trivial;

Complexidade assintótica: $O(|E| * log(|V|))$ por consulta, totalizando $O(Q * |E| * log(|V|))$.
Número de soluções enviadas: 58
Casos de teste aceitos: 5
Veredito final: 100% de acurácia

### 3.4.1 Armazenando soluções já visitadas

Como bem sabemos, é fácil modificar o Dijkstra para retornar a distância mínima (Ou, no caso, o gargalo máximo) de um vértice $u$ para todos os outros vértices $i \in S$ do grafo. Podemos melhorar ligeiramente a complexidade assintótica da solução anterior armazenando todas as soluções já encontradas pelo nosso Dijkstra em uma matriz $|V|x|V|$. Dessa forma, serão necessárias apenas $|V|$ execuções do Dijkstra (Uma para cada vértice), e consultas repetidas poderão ser respondidas em $O(1)$. Nosso novo custo será $O(|V| * |E| * log(|V|) + Q)$.

Complexidade assintótica: $O(|V| * |E| * log(|V|) + Q)$.
Número de soluções enviadas: $\approx 4$ (Alguns estudantes não documentaram essa modificação, e eu posso ter esquecido de anotar alguns casos)
Casos de teste aceitos: 5
Veredito final: 100% de acurácia

### 3.4.2 Floyd-Warshall modificado

Da mesma forma que o algoritmo de Dijkstra pode ser modificado para obter o gargalo máximo entre um único par de vértices, o algoritmo de Floyd-Warshall (Que pode ou não estar na ementa de vocês - se não estiver, recomendo a leitura, é uma aplicação bem bacaninha de programação dinâmica) pode ser modificado para obter o gargalo máximo entre todos os pares de vértices.

Por simplicidade, vamos utilizar um argumento empírico dessa vez. A demonstração é análoga à demonstração de corretude do Floyd-Warshall original: Na k-ésima iteração do algoritmo, $w(i, j)$ irá conter o gargalo máximo que pode ser obtido em um caminho de $i$ até $j$ que utilize apenas os vértices $\{1, ..., k\}$. Para tanto, verificamos se $k$ faz parte de um caminho ótimo de $i$ até $j$ (Utilizando apenas os vértices $\{1, ..., k\}$) e atualizamos o valor de $w(i, j)$ adequadamente com $w(i, j) = max(w(i, j), min(w(i, k), w(k, j)))$.

Naturalmente, após $|V|$ iterações, o valor de $w(i, j)$ contém o gargalo máximo que pode ser obtido em um caminho de $i$ até $j$ utilizando todos os $|V|$ vértices, isto é, entre todos os caminhos possíveis no

grafo (Não incluindo vértices repetidos). Com essa nova matriz em mãos, podemos responder a todas as consultas em tempo constante.

```
1 for k in V:
2     for i in V:
3         for j in V:
4             w(i,j) <− max(w(i,j), min(w(i,k), w(k,j)) )
```

Figura 5: Pseudocódigo para o algoritmo de remoções iterativas

Prós:

- Implementação trivial;
- Análise assintótica trivial;
- Melhor custo assintótico possível para os limites fornecidos no enunciado do trabalho (A parte mais pesada do programa deixa de ser a série de consultas);

Contras:

- Requer conceitos de programação dinâmica;
- Demonstração de corretude menos trivial que outras alternativas;

Complexidade assintótica: $O(|V|^3)$ em pré-processamento + $O(1)$ por consulta, totalizando $O(|V|^3+Q)$
Número de soluções enviadas: 0
Casos de teste aceitos: 5
Veredito final: 100% de acurácia

### 3.5 Busca Binária + DFS

A última família de soluções possíveis envolve soluções baseadas em **divisão em conquista**, em particular, no algoritmo de Busca Binária. Suponha que existe um caminho de $u$ até $v$ com capacidade menor ou igual a $C$. Então, trivialmente, existe um caminho de $u$ até $v$ com capacidade menor ou igual a $C' < C$. Similarmente, se não existe um caminho de $u$ até $v$ com capacidade maior ou igual a $C$, então *não existe* caminho de $u$ até $v$ com capacidade maior ou igual a $C' > C$.

Seja $f(u, v, C) = 1$ se é possível viajar de $u$ até $v$ com uma carga de $C$ quilos, e $f(u, v) = 0$ caso contrário. De acordo com o parágrafo anterior, podemos concluir que $f(u, v)$ é *monotonicamente decrescente*. Em particular, a função segue o formato ilustrado pela Figura 7.

Como sabemos, qualquer função monotonicamente crescente/decrescente pode ser resolvida por meio de uma Busca Binária. Em outras palavras, o algoritmo ilustrado pela Figura 6 resolve o problema do gargalo máximo:

Para implementar a função $is\_there\_a\_path()$, podemos utilizar uma simples DFS com a restrição adicional de que apenas arestas com peso maior ou igual a $C$ serão consideradas. Inicializamos a busca utilizando os limites para as capacidades das arestas, isto é, $Cleft = 1$ e $Cright = 100000$. Alternativamente, uma BFS, Dijkstra ou qualquer outro algoritmo que encontra um caminho qualquer entre dois vértices também funciona.

```
int binary_search(u,v,Cleft,Cright):
    if Cleft == Cright:
        return Cleft

    mid <- (Cleft + Cright)/2
    if(is_there_a_path(u,v,C)):
        return binary_search(u,v,mid,Cright)
    else:
        return binary_search(u,v,Cleft,mid)
```

Figura 6: Pseudocódigo para o algoritmo de busca binária

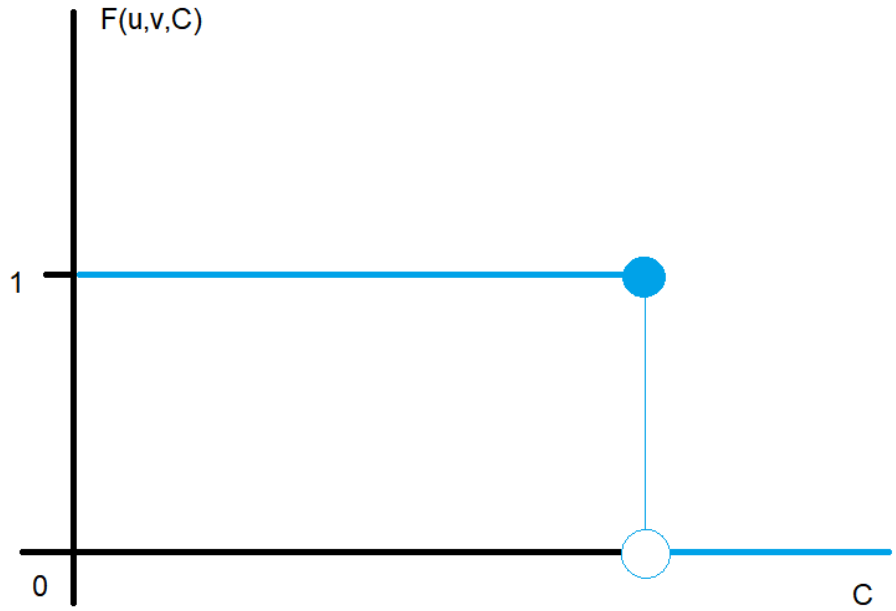


Figura 7: $F(u, v, C)$ é uma função monotonicamente decrescente, o que significa que podemos aplicar busca binária a ela

A análise de complexidade é bastante direta. Seja $B$ a capacidade da maior aresta dada pela entrada. Para cada consulta, realizamos uma busca binária que executa $O(log(B))$ passos, cada qual possui um custo de $O(|V| + |E|)$, totalizando um custo de $O(|V| * log(B) + |E| * log(B))$ por consulta.

Prós:

- Implementação extremamente simples;
- Análise de complexidade relativamente trivial (Poucos detalhes para se preocupar);
- Custo assintótico razoável (Embora ligeiramente inferior aos outros, pois $B$ possui limites maiores do que $|V|$ ou $|E|$);

Contras:

- Possivelmente a solução mais difícil de conceptualizar do trabalho;
- Demonstração de corretude requer alguns termos complicadinhos como "função monotonicamente crescente", ou uma explicação empírica equivalente;

Complexidade assintótica: $O(log(B) * (|V| + |E|))$ por consulta, totalizando $O(Q * log(B) * (|V| + |E|))$
Número de soluções enviadas: 3
Casos de teste aceitos: 5
Veredito final: 100% de acurácia

# 4 Soluções aplicáveis a grafos não-direcionados

Originalmente, o trabalho foi definido sob um grafo não direcionado com o objetivo de dificultar o plágio (Pois existe um número maior de soluções prontas online para o caso não direcionado). Infelizmente, essa ação inviabiliza algumas das soluções mais interessantes para o problema, que se aplicam apenas ao caso não-direcionado.

Originalmente, este editorial havia sido elaborado desconsiderando essa simples modificação, de forma que soluções para o caso direcionado haviam sido inclusas. Para não desperdiçar o meu esforço e também para enriquecer a leitura do documento, tais soluções serão descritas em separado nesta seção.

Vale lembrar, naturalmente, que todas as soluções válidas para o caso direcionado também se aplicam ao caso não direcionado (Pois todo caso direcionado pode ser transformado em um caso não direcionado).

## 4.1 Minimal Spanning Tree + DFS

Proposição: Em um grafo não-direcionado, todas as arestas do caminho de gargalo máximo entre dois vértices $u$ e $v$ estão presentes na *Árvore Geradora Máxima* do grafo.

Prova (Informal): Suponha, por absurdo, que alguma aresta do caminho de gargalo máximo não está presente na Árvore Geradora Máxima do grafo. Então, por definição, o gargalo do caminho também não está presente na Árvore. Seja $(i, j)$ a aresta correspondente a esse gargalo. Note que a Árvore é geradora, isto é, existe um caminho de qualquer vértice para qualquer vértice. Mas se o gargalo não está na Árvore Geradora Máxima, isto significa que existe um caminho que consiste exclusivamente de arestas com maior (Ou igual) capacidade que interliga $i$ e $j$. Podemos substituir $(i, j)$ por esta série de arestas, e obteremos um caminho de maior gargalo. Logo, o caminho original não era um caminho de gargalo máximo.

Agora que sabemos que todas as arestas do caminho de gargalo máximo estão presentes na Árvore Geradora Máxima, a solução se torna bem simples: Basta computar a Árvore Geradora Máxima do grafo, utilizando o algoritmo de Kruskal ou Prim (A implementação é análoga à Árvore Geradora Mínima. Alternativamente, você pode inverter os pesos de todas as arestas e resolver o problema de Árvore Geradora Mínima correspondente), e então retornar para cada consulta o gargalo do único caminho entre $u$ e $v$ nessa árvore. O pseudocódigo é descrito na Figura 8.

```
1 T(V,E) <- minimal_spanning_tree(G(V,E))
2 for u,v in consultas:
3     get_bottleneck_in_tree(T,u,v)
```

Figura 8: Pseudocódigo para o algoritmo baseado em Árvore Geradora Máxima

Onde a função *get_bottleneck_in_tree*() pode ser implementada utilizando uma simples DFS (Vale lembrar que só existe um caminho de $u$ até $v$, pois nosso novo grafo é uma árvore).

Prós:

- Implementação relativamente simples;

- Um dos melhores custos assintóticos para o problema;
- Otimizável, conforme veremos na seção seguinte;

Contras:

- Demonstração de corretude pouco trivial;
- Relativamente difícil de conceptualizar;
- Algoritmo apenas válido para o caso não-direcionado;

Complexidade assintótica: $O(|E| * log(|V|))$ em pré-processamento + $O(|V| + |E|)$ por consulta, totalizando $O(|E| * log(|V|) + Q * (|V| + |E|))$.

Observação: A solução não se aplica ao caso direcionado, onde a Árvore Geradora Máxima não está definida. Seu equivalente em um grafo direcionado, a Arborescência Geradora Máxima, é baseada em um vértice raiz $r$ e, portanto, mais assemelha-se a um Dijkstra que a uma árvore geradora (Pois não garante conectividade entre todos os vértices, sendo portanto necessária uma execução por vértice).

#### 4.1.1 Minimal Spanning Tree + LCA (Leitura altamente opcional)

É possível otimizar nosso algoritmo baseado em Árvores Geradoras Mínimas para responder consultas em $O(log(|V|)$ com um certo nível de pré-processamento e algumas estruturas de dados mais avançadas. Infelizmente, este tópico está muito além não apenas do escopo da disciplina, como também da minha capacidade de explicar em poucas linhas. Encorajo estudantes que possuam interesse no tópico a visitarem a seguinte página para que obtenham mais informações: `https://www.topcoder.com/thrive/articles/Range%20Minimum%20Query%20and%20Lowest%20Common%20Ancestor`

De forma resumida, podemos utilizar algoritmos especializados para encontrar o caminho mais curto entre dois vértices (E, com algumas modificações, o gargalo máximo também) em uma árvore com um custo computacional de $O(log(|V|))$ (Por isso a necessidade de computar a Árvore Geradora Mínima primeiro). Seu uso é altamente desnecessário e eu estou quase feliz que ninguém tenha considerado a hipótese (Embora seja uma bela solução).

Prós:
- Melhor custo assintótico possível na prática;

Contras:

- Implementação altamente complexa;
- Tópico abordado muito além do nível necessário para a turma (Apenas descrito para propósitos de documentação);

Complexidade assintótica: $O(|E| * log(|V|))$ em pré-processamento + $O(log(|V|))$ por consulta, totalizando $O(|E| * log(|V|) + Q * log(|V|))$.

Observação: Estudantes que tenham interesse no tópico também podem referir-se ao seguinte documento: `https://maratonapcauece.files.wordpress.com/2014/09/solucoes_regional_2013.pdf`, páginas 19-24. Este documento contém uma breve explicação sobre o uso de algoritmos especializados para encontrar o gargalo máximo em uma árvore geradora.

### 4.2 Prim modificado

Uma solução análoga à seção 3.4 consiste em visualizar o seu algoritmo não como um Dijkstra modificado, mas como um *Prim* modificado. Como vimos na seção 4.1, o gargalo máximo entre $u$ e $v$ estará sempre contido na árvore geradora máxima do grafo.

Porém, para esse problema, podemos utilizar o algoritmo de Prim também para responder cada consulta. Nesse caso, não precisamos gerar a Árvore Geradora Máxima completa: Basta parar a execução quando alcançarmos o vértice destino $v$. Essa solução ainda está correta pois, em uma árvore, sempre existe um único caminho entre qualquer par de vértices $u$ e $v$, e o nosso Prim modificado já encontra esse caminho sem precisar criar o resto da árvore (Nota: Esse é um exemplo de argumento empírico).

Nosso algoritmo de Prim modificado funciona da mesma forma que o Dijkstra modificado. A vantagem está na forma como visualizamos o algoritmo: Ao visualizar nossa solução como um algoritmo de Prim, podemos aproveitar a estrutura de árvore geradora do algoritmo para apresentar uma prova de corretude mais simples.

Prós:

- Prova de corretude mais simples que o Dijkstra;

Contras:

- Complexidade assintótica inferior à discutida na seção 4.1, pois é necessária uma execução do algoritmo para cada consulta;

Complexidade assintótica: $O(|E| * log(|V|))$ por consulta, totalizando $O(Q * |E| * log(|V|))$.

Observação: Como o algoritmo é basicamente um Dijkstra, a solução ainda funciona em grafos direcionados. Contudo, o argumento sobre árvores geradoras e a prova de corretude associada não funcionam, sendo necessária uma prova alternativa.

## 5 Considerações sobre as correções

Esta seção é dedicada a explicar, em maior detalhe, as penalidades aplicadas durante a correção do TP02 e seus propósitos.

Os seguintes tópicos referem-se às principais penalidades aplicadas no Fator de Implementação (Totalizando 60% da nota):

- **Ausência de instruções de compilação/Leitura não efetuada pela entrada padrão: ~~(-100% compilação / -100% corretude)~~**: Embora eu já tenha explicado essa penalidade algumas vezes no fórum, vale a pena reiterar: Todas as especificações do trabalho estão ali por um motivo, e qualquer trabalho que não atenda a essas especificações pode e deve(ria) ser punido com a anulação completa da nota. Pessoalmente, eu adoraria ter mantido uma penalidade significativa sobre estes trabalhos, porém, devido a intervenção das professoras, foi necessário remover as penalidades novamente.

  Nota adicional: Nenhum dos estudantes que cometeu um desses erros no decorrer do TP01 repetiu o mesmo erro durante o TP02. Fico feliz.

- **Soluções incorretas (-20% a -100% corretude)**: Foram empregados 5 casos de teste, descritos em um documento à parte. Vale notar que uma única consulta respondida incorretamente é motivo para anulação da nota no caso de teste em questão. Isso ocorre porque o que estamos avaliando é a eficiência do seu programa no **grafo apresentado** pelo caso de teste, não para cada consulta individualmente.

  Contudo, essa observação é relativamente irrelevante, haja visto que todos os programas penalizados retornaram várias consultas incorretas para cada caso de teste.

  Os grafos empregados na análise foram: Dois grafos pequenos (Fornecidos no enunciado), uma rede baseada em uma topologia em estrela (Grafo esparso) e dois grafos completos (Densos) com capacidades e números de consultas variados.

- **Tempo limite excedido (-100% tempo de execução)**: Refere-se a trabalhos que não tenham retornado uma solução (Correta ou não) dentro do tempo limite de 5 segundos. Como vimos na seção anterior, um trabalho que não retorna a solução correta dentro do tempo limite ignora os paradigmas estudados em sala de aula em prol de uma solução mais crua e menos eficiente.

- **Código sem comentários (-20% a -75% qualidade de código)**: Refere-se a códigos difíceis de compreender devido à ausência de comentários. A penalidade é heurística (i.e. de acordo com a discrição do avaliador), e baseia-se no quanto o código é difícil de acompanhar.

Os seguintes tópicos, por sua vez, referem-se às principais penalidades aplicadas no Fato de Documentação (Totalizando 40% da nota).

- **Modelagem incorreta (-75% modelagem)**: Refere-se a 'soluções' que não resolvem, de fato, o problema (Vide seção 3.2 para exemplos). A penalidade é de 75% ao invés de 100% levando em conta o esforço empregado em argumentar a corretude de sua solução.

- **Ausência de demonstração de corretude da solução (-75% modelagem)**: Refere-se a soluções corretas, mas cuja documentação não apresenta argumentos para sua corretude. A penalidade é alta pois uma solução correta sem justificativa é tão ruim quanto uma solução incorreta. Reitero: Uma solução correta sem justificativa **não é uma solução**. Uma solução deve funcionar para qualquer caso de teste (Dentro dos limites fornecidos). Se vocês não me apresentam uma justificativa, a solução de vocês não tem essa garantia e, portanto, está basicamente incorreta.

- **Ausência de demonstração de corretude da solução - com exemplos (-50% modelagem)**: Refere-se a trabalhos que tentam argumentar a corretude de sua solução apenas por meio de exemplos. Exemplos não são uma boa prova de corretude: O fato de o seu programa funcionar para um caso de teste não garante que ele irá funcionar para todos os outros casos de teste (Ao contrário de uma argumentação formal ou empírica, como as propostas pela seção 3, que são válidas para qualquer entrada possível). O ganho de 25% em relação aos casos anteriores deriva-se, novamente, do esforço em argumentar a corretude da solução, mostrando ao menos uma compreensão da importância em demonstrar a corretude da solução.

- **Demonstração de corretude insuficiente (-50% modelagem)**: Refere-se a trabalhos que tentam argumentar a corretude de sua solução de outras formas, sem de fato garantir a corretude do trabalho. A justificativa é análoga à penalidade para trabalhos que utilizaram apenas exemplos.

- **Análise de complexidade - Variáveis não nomeadas (-100% análise de complexidade)**: Vale a pena reiterar: Se a análise de complexidade de vocês não explicita o significado de cada variável (Seja 'n', 'V' ou qualquer outro nome), vocês simplesmente não tem uma análise de complexidade

(Ou melhor, a análise de vocês não diz nada sobre o seu algoritmo). Felizmente, poucos trabalhos falharam neste ponto dessa vez.

- **Análise incorreta para a solução das consultas (-50% análise de complexidade)**: Refere-se a análises incorretas para o Dijkstra/Prim/Busca binária/etc. empregado por vocês para resolver as consultas. Penalidade relativamente leve, pois os outros 50% referem-se à análise do programa principal.
- **Análise incorreta para o programa principal (-50% análise de complexidade)**: Refere-se principalmente a trabalhos que não levaram em conta a quantidade $Q$ de consultas (Pessoal, por favor, no segundo TP não) em sua análise. Pode parecer uma penalidade alta, mas existe uma enorme diferença em termos de eficiência entre um código $O(|E| * log(|V|))$ e um código $O(Q * |E| * log(|V|))$.

  Vale a pena lembrar que todos os valores fornecidos na entrada são - do ponto de vista da análise de complexidade - variáveis, e não constantes, pois estão sob o controle da pessoa que fornece a entrada (Ou de outros fatores externos), mas certamente não de vocês enquanto programadores.

Outras penalidades aplicadas foram completamente heurísticas (Exemplo: Apresentação, descrição e qualidade do código), e baseadas no dano causado pela infração à compreensão do trabalho como um todo.