**IMT – Instituto Mauá de Tecnologia**

**Tópicos Avançados em Estrutura de Dados – Tarefa T15**

**Ana Helena Marcacini RA: 20.01305-0**

**Ettore Padula Dalben RA: 20.00387-0**

**Pedro Henrique Hein RA: 20.00134-7**

**Teoria de Grafos e Implementação de Algoritmos em Grafos**

**São Caetano do Sul**

**2022**

Índice

[1. Resumo (Abstract) 3](#_Toc121079607)

[1.1 Palavras-chave 3](#_Toc121079608)

[2. Introdução 3](#_Toc121079609)

[3. Desenvolvimento do Tema 3](#_Toc121079610)

[3.1 Planaridade de Grafos 3](#_Toc121079611)

[3.2 Implementação de Busca em Profundidade de Grafos 4](#_Toc121079612)

[3.2.1 Definição 5](#_Toc121079613)

[3.2.2 Desempenho 5](#_Toc121079614)

[3.3 Implementação do Algoritmo de PRIM 5](#_Toc121079615)

[3.3.1 Implementação do Algoritmo 6](#_Toc121079616)

[3.3.2 Implementação Eficiente 7](#_Toc121079617)

[3.3.3 Primeira Implementação Eficiente 8](#_Toc121079618)

[3.3.4 Segunda Implementação Eficiente 8](#_Toc121079619)

[3.4 Implementação do Algoritmo de Kruskal 9](#_Toc121079620)

[3.4.1 Implementação Ingênua 10](#_Toc121079621)

[3.4.2 Implementação Eficiente com union-find 10](#_Toc121079622)

[3.5 Implementação do Algoritmo de Dijkstra 12](#_Toc121079623)

[3.5.1 Princípio do Algoritmo 12](#_Toc121079624)

[3.5.2 Implementação do Algoritmo 13](#_Toc121079625)

[3.5.3 Primeira Implementação 14](#_Toc121079626)

[3.5.4 Segunda Implementação 15](#_Toc121079627)

[4. Conclusões ou considerações finais 16](#_Toc121079628)

[5. Referências Bibliográficas 16](#_Toc121079629)

# Resumo (Abstract)

Neste artigo, trataremos de três tópicos cruciais das estruturas de dados, os quais são a planaridade de grafos, a busca em profundidade e a implementação de algoritmos a elas relacionados.

## Palavras-chave

Planaridade; grafos; algoritmo; busca; profundidade; pré-ordem; vetor; função; implementação; árvore

# Introdução

A maleabilidade e versatilidade dos modos em que um problema pode ser resolvido alcança uma miríade de possibilidades. O desenvolvimento de um raciocínio que auxilie na eficiente solução de um problema é indispensável para bacharéis em engenharia de computação.

O tratamento e manipulação dos dados contidos em um grafo é fundamental para aprimorar serviços e produtos que usufruem dessa estrutura, assim como o aplicação de suas propriedades e características que permitem os desenvolvedores otimizar operações importantes aos processos.

# Desenvolvimento do Tema

## Planaridade de Grafos

A teoria da planaridade gira em torno dos ciclos de grafos, ciclos, por sua vez, são um número de vértices conectados em uma rede fechada. É importante conhecer os grafos planos e os desenhos, para se determinar um grafo planar.

Um grafo plano é um par (V, E) com as seguintes características:

* Os vértices são um subconjunto finito do plano R²;
* Toda aresta é um arco poligonal entre dois vértices;
* Arestas diferentes têm diferentes conjuntos de pontas;
* O interior de uma aresta não contém vértices nem pontos que pertençam a outra aresta.

Um arco é a união de segmentos de reta finitos no plano semelhante ao intervalo fechado [0;1] da reta. As imagens de 0 e 1 são as pontas do arco.

* Todo grafo plano (V, E) corresponde a um grafo combinatório;
* O conjunto de pontos de um grafo plano é, topologicamente, um subconjunto fechado e limitado do plano R²;
* Uma face de um grafo plano G é qualquer região do conjunto topológico aberto R². Se H é subgrafo de G toda face de H é parte de uma face de G;
* A fronteira topológica de uma face corresponde a um subgrafo.

Um desenho de um grafo G é um grafo plano H isomorfo a G em que os conceitos de isomorfismo entre grafos planos são:

* Isomorfismo topológico, induzido por isomorfismo do plano;
* Isomorfismo combinatório, pode ser estendido a uma bijeção entre faces que preserva incidência entre faces e arestas.

Com isso, define-se um grafo planar como:

* Um grafo combinatório é planar se é isomorfo a um grafo plano, isso é, se admite um desenho;
* Uma coleção F de subconjuntos de E(G) é simples se toda aresta de G pertence a no máximo dois membros de F.

## Implementação de Busca em Profundidade de Grafos

Um algoritmo de busca é qualquer algoritmo que visita todos os vértices de um grafo percorrendo arcos de um vértice a outro. Há muitas maneiras de fazer uma tal busca, portanto são caracterizado pela ordem que visitam os vértices.

O algoritmo de busca em profundidade, ou busca DFS, trata-se de uma generalização do algoritmo com propósito de decidir o alcance entre vértices. Com objetivo de visitar todos os vértices e numerá-los em ordem de descoberta.

A busca em profundidade não resolve problemas específicos. Ela é apenas um pré-processamento, para resolução eficiente de vários problemas concretos. A busca DFS ajuda a compreensão do grafo estudado, revelando sua forma e unindo informações (numeração dos vértices) úteis para responder perguntas.

### Definição

Uma formação comum para o algoritmo de busca em profundidade é:

Visita todos os vértices e todos os arcos do grafo numa determinada ordem e numera cada vértice (o k-ésimo vértice descoberto recebe o número k ).

A busca poderia começar por qualquer vértice, mas é padrão iniciá-la pelo vértice 0. Registra-se a posição dos vértices num vetor indexado pelos vértices.

A ordem de descoberta dos vértices é chamada pré-ordem, para obter permutação dos vértices em pré-ordem basta inverter o vetor indexado.

### Desempenho

Uma função f(G) examina o leque de saída de cada vértice uma só vez. Portanto, cada arco é examinado uma só vez. Assim, se o grafo tem V vértices e A arcos, f(G) consome tempo proporcional a V + A . Esse consumo é proporcional ao tamanho do grafo e, portanto, também ao tempo necessário para ler todas as listas de adjacência.

No caso de grafos representados por matriz de adjacências, a função f(G), combinada com a versão apropriada de DFSr(G), consome tempo proporcional a V² quando aplicada a um grafo com V vértices. Esse consumo é proporcional ao tempo necessário para ler a matriz de adjacências. Se o grafo é esparso, essa segunda versão é mais lenta que a primeira.

## Implementação do Algoritmo de PRIM

O algoritmo de Prim é simples, mas sua implementação eficiente apresenta dificuldades. Dado um grafo não-dirigido conexo G com custos nas arestas, o algoritmo de Prim cultiva uma subárvore de G até que ela se torne geradora.

Franja, neste contexto, é o corte cuja margem é o conjunto de vértices de uma subárvore. A cada iteração começa com uma subárvore S. No início da primeira iteração, S consiste em um único vértice. O processo iterativo consiste no seguinte (enquanto a franja de S não estiver vazia):

* Escolha de uma aresta da franja que tenha custo mínimo;
* Seja x-y a aresta escolhida, com x em S;
* Acrescente a aresta x-y e o vértice y a S.

O algoritmo tem caráter guloso, a cada iteração, abocanha a aresta mais barata da franja sem se preocupar com o efeito a longo prazo, dessa escolha.

### Implementação do Algoritmo

A árvore geradora S do grafo é representada por uma árvore radicada. Para isso, basta selecionar um vértice de S para fazer o papel de raiz e eliminar um dos dois arcos de cada aresta de S. A árvore radicada será representada por um vetor de pais alocado pelo usuário.

Suponhamos que o grafo é representado por listas de adjacência com custos. Para cada vértice V e cada A em G➔adj[V], o custo do arco que liga V a A➔w será V➔c e esse número pode ser positivo ou negativo. Que temos uma constante CONS de valor maior que o custo de qualquer aresta. Por fim, que os dois arcos que compõem cada aresta têm o mesmo custo.

Uma implementação ingênua transforma o algoritmo de Prim em código de maneira direta e literal. O resultado é simples, mas ineficiente.

O desempenho dessa implementação é quadrático, se aplicada a um grafo com V vértices e E arestas, consome tempo proporcional a VE, no pior caso. Pode-se dizer que o tempo é proporcional a V vezes o tamanho do grafo.

A implementação ingênua do algoritmo de Prim é lenta e ineficiente porque cada iteração recalcula toda a franja da árvore, mesmo sabendo que a franja mudou pouco desde a iteração anterior.

### Implementação Eficiente

Para obter uma implementação mais eficiente, é preciso iniciar cada iteração com a franja pronta e atualizá-la no fim da iteração. Mas é difícil fazer isso se a franja for tratada como uma simples lista de arestas. É preciso inventar uma representação mais eficiente e tomar algumas decisões adicionais de projeto.

A fronteira de uma árvore S é o conjunto de todos os vértices do grafo que não pertencem a S mas são vizinhos de vértices de S. O preço de um vértice w da fronteira de S é o custo de uma aresta de custo mínimo dentre as que estão na franja de S e incidem em w. Se a aresta da franja que determina o preço de w é v-w, diremos que v é o gancho de w.

Podemos agora reescrever o algoritmo de Prim em termos de preços e ganchos. Cada iteração começa com uma árvore S e com os preços e ganchos dos vértices que estão na fronteira de S. O processo iterativo consiste no seguinte (enquanto a franja de T não estiver vazia):

* Escolha um vértice y de preço mínimo na fronteira de S;
* Seja x um gancho de y;
* Acrescente o arco x-y e o vértice y a S;
* Atualize os preços e ganchos fora de S.

Para armazenar os valores dos vértices usamos um vetor VAL indexado pelos vértices. Os ganchos podem ser armazenados num vetor alocado para esse fim, mas é melhor armazená-los na parte ociosa do vetor de pais de S, ou seja, nas posições do vetor GAN indexadas pelos vértices da fronteira de S.

Com isso, os elementos de GAN tem a seguinte interpretação: se v está em S então GAN[v] é o pai de v, se v está na fronteira de S então GAN[v] é o gancho de v, e nos demais casos GAN[v] está indefinido. Poderíamos dizer que os elementos de GAN indexados pelos vértices da fronteira são provisórios, estando sujeitos a alterações nas próximas iterações.

### Primeira Implementação Eficiente

No início de cada iteração da primeira implementação eficiente temos:

* O vetor característico TREE do conjunto de vértices da árvore S;
* Um vetor VAL que contém o valor de cada vértice na fronteira de S;
* Um vetor GAN que contém os pais dos vértices de S e os ganchos dos vértices da fronteira de S.

Quando aplicada a um grafo não-dirigido com V vértices e E arestas, a função implementação consome tempo proporcional a V² + E. Como E < V², o consumo de tempo da função é proporcional a V². Como o tamanho de grafos densos é proporcional a V², podemos dizer que esta implementação é linear para grafos densos.

### Segunda Implementação Eficiente

Esta implementação mantém os vértices da fronteira em ordem crescente de valor (ou quase isso), para que não seja preciso procurar o vértice mais barato.

Assim como na primeira implementação do algoritmo de Prim, cada iteração da segunda implementação começa como:

* O vetor característico TREE do conjunto de vértices da árvore S;
* Um vetor VAL que contém o preço de cada vértice da fronteira de S;
* Um vetor GAN que contém os pais dos vértices de S e os ganchos dos vértices da fronteira de S.

Mas, diferentemente da primeira implementação, os vértices que não pertencem a S são mantidos numa fila priorizada "de mínimo".

Os vértices que não pertencem à árvore S ficam armazenados numa fila priorizada "de mínimo" com prioridade ditada pelo valor de cada vértice. A fila é manipulada pelas funções:

* FILA(G➔V): inicia uma fila priorizada para G➔V vértices;
* VAZIO: devolve true se e somente se a fila está vazia;
* INSERE(w,val): insere o vértice w na fila com prioridade VALOR[w];
* DELIMIN(val): retira da fila um vértice y que minimiza VALOR[];
* DESC(w,val): reordena a fila após o valor de VALOR[w] diminuir.

A implementação clássica da fila priorizada usa estrutura de heap. Com essa implementação consome tempo proporcional a (V+E) log( V) no pior caso. Como G é conexo, temos E ≥ V−1 e, portanto, o consumo de tempo é proporcional a Elog V, no pior caso. Assim, esta implementação é apenas um pouco pior que linear. Podemos dizer que ela é linearítmica.

## Implementação do Algoritmo de Kruskal

O algoritmo de Kruskal, publicado por Joseph Kruskal em 1956, forma uma floresta geradora até que ela se torne conexa. Uma subfloresta de um grafo não-dirigido G é qualquer floresta que seja subgrafo não-dirigido de G. Uma floresta geradora de G é uma subfloresta que tenha o mesmo conjunto de vértices que G.

Uma aresta A de G é externa a uma floresta geradora F se A não pertence a F e o grafo F +  A é uma floresta, ou seja, um grafo sem circuitos. Portanto, uma aresta é externa a F se tem uma ponta em uma componente conexa de F e outra ponta em outra componente.

Cada iteração do algoritmo de Kruskal começa com uma floresta geradora F de G. O processo iterativo é muito simples (enquanto existir arestas externas):

* Escolha uma aresta externa que tenha custo mínimo;
* Seja a A aresta escolhida;
* Acrescente A a F.

No início da primeira iteração, cada componente conexa da floresta F tem apenas um vértice. No fim do processo iterativo, F é conexa, uma vez que G é conexo e não há arestas externas a F.

O algoritmo tem caráter guloso: a cada iteração, escolhe a aresta que parece mais promissora localmente sem se preocupar com o efeito final dessa escolha.

### Implementação Ingênua

Uma boa implementação do algoritmo de Kruskal necessita escolher duas estruturas de dados: uma para representar a floresta geradora e uma auxiliar que facilite a decisão se uma aresta é externa a uma dada floresta geradora.

A representação por vetor de pais é incômoda no caso desse algoritmo. Uma simples lista das arestas da floresta, armazenada num vetor, servirá como representação. Cada aresta v-w será representada por uma estrutura de aresta, construída por uma função.

Para decidir arestas externas, um vetor de chefes será usado, de forma que:

* Eleja um dos vértices de cada componente conexa da floresta para ser o chefe da componente;
* Para cada vértice v do grafo, denote por CHEFE[v] o chefe da componente que contém v;
* Uma aresta v-w é externa à floresta se, e somente se, CHEFE[v] ≠ CHEFE[w].

Essas estruturas permitem implementação simples do algoritmo de Kruskal. Tal implementação recebe um grafo não-dirigido conexo G e armazena no vetor FLO as arestas de uma árvore geradora do grafo. FLO terá G➔V-1 elementos. A função usa uma constante INFI de valor maior que o custo de qualquer aresta.

A função ainda faz a união de duas componentes conexas da floresta para que ambas passem a ter o mesmo chefe. Porém, essa função é muito lenta, consumindo V (E + V) unidades de tempo, no pior caso, para processar um grafo não-dirigido com V vértices e E arestas. Como E < V², diz-se que o consumo de tempo é limitado por V³.

### Implementação Eficiente com union-find

A implementação anterior é ineficiente por dois motivos. Pois cada iteração examina todas as arestas a procura da aresta externa mais barata e porque cada iteração examina todos os vértices para atualizar o vetor de chefes.

Para corrigir a primeira ineficiência, basta ordenar as arestas em relação crescente de custo e examinar cada aresta uma só vez. A segunda ineficiência é mais difícil de corrigir, será preciso recorrer à estrutura union-find, que usa um vetor de chefes mais flexível.

A esta implementação começa por invocar uma função que armazena as A arestas do grafo num vetor E[0..E-1]. Em seguida, invoca uma função de ordenação para rearranjar E[0..E-1] em ordem crescente de custos. Por fim, examina as arestas na ordem em que elas estão em E[0..E-1] e escolhe as que farão parte da árvore geradora. Ainda há 3 funções que fazem o seguinte:

* UFINIT inicia a estrutura de chefes fazendo com que cada vértice seja o seu próprio chefe.
* UFCHEF(v) devolve o chefe da componente conexa de F que contém o vértice v.
* UFUNE(v0,w0) faz a união das componentes cujos chefes são v0 e w0 respectivamente.

Para fazer uma implementação eficiente das funções UFCHEF e UFUNE, usa-se as ideias do tipo-de-dados abstrato union-find.

Union-find: A estrutura é uma flexibilização CHE[0..V-1] do vetor de chefes que já usamos acima que consiste no seguinte: para cada vértice v, o vértice CHE[v] não mais precisa ser o chefe de v, desde que seja possível chegar ao chefe iterando CHE, ou seja, fazendo CHE[CHE[v]], CHE[CHE[CHE[v]]], etc. Com essa flexibilização, CHE[v] passa a ser uma espécie de "superior imediato" de v. É claro que v é um chefe se e somente se CHE[v] ≡ v.

Para os chefes de v e w iguais a v0 e w0 respectivamente. Se esses dois vértice são diferentes, a função UFUNE une as componente chefiadas por v0 e w0 em uma só tornando v0 o superior imediato de w0 ou vice-versa. Para evitar que seja necessário iterar CHE muitas vezes para chegar a um chefe, a função UFUNE faz com que o chefe da maior das duas componente seja também o chefe da menor. O tamanho das componentes é mantido em um vetor TAM: se v0 é um chefe então TAM[v0] é o número de vértices chefiados por v0.

Dessa forma, o número de repetições de CHE necessário para chegar a um chefe é menor que log V, sendo V o número de vértices de G. Assim, cada chamada de UFCHEF e UFUNE consome log V unidades de tempo no máximo.

Para o grafo com V vértices e E arestas, a função ordenadora consome tempo limitado por E log E. O restante do código consiste em 2V invocações de UFCHEF e V invocações de UFUNE e, portanto, consome tempo limitado por V log V. Como log E < 2 log V, então o consumo é limitado por (E + V) log V, no pior caso. Como o grafo é conexo, temos E ≥ V−1 e, portanto, o algoritmo consome tempo proporcional a E log V, no pior caso. Diz-se que o algoritmo é linearítmico.

## Implementação do Algoritmo de Dijkstra

Uma árvore de caminhos baratos (ACB) é uma subárvore radicada geradora S de G tal que todo caminho em S que começa na raiz tem custo mínimo em G. Um grafo com custos positivos tem uma ACB com raiz R se e somente se todos os seus vértices estão ao alcance de R. Para escapar dessa condição, pode-se trabalhar com o subgrafo induzido pelos vértices que estão ao alcance de R.

Um algoritmo simples foi descoberto e publicado em 1959 por Edsger W. Dijkstra. Mas, sua implementação eficiente apresenta dificuldades inesperadas.

### Princípio do Algoritmo

Dado um grafo G com custos positivos nos arcos e um vértice W, o algoritmo de Dijkstra cresce uma subárvore radicada em G, a partir do vértice W, até englobar todos os vértices ao alcance de W. Caso todos os vértices de G estejam ao alcance de W. Ao final da execução, a subárvore torna-se geradora.

A franja de uma subárvore radicada S de G é o conjunto de todos os arcos do grafo que têm ponta inicial em S e ponta final fora de S. Em outras palavras, a franja de S é o leque de saída do conjunto de vértices de S.

O algoritmo é iterativo. Cada iteração começa com uma árvore radicada S, com raiz W, e o vetor DIST das distâncias em S a partir de W. No começo da primeira iteração, W é o único vértice de S e DIST[s] vale 0. O processo iterativo consiste no seguinte (enquanto a franja de T não estiver vazia):

* Escolha, na franja de S, um arco x-y que minimize DIST[x] + Cx-y;
* Acrescente o arco x-y e o vértice y a S;
* Faça DIST[y] = DIST[x] + Cx-y .

Para Cx-y como custo do arco x-y. Depois do passo 1, podemos dizer que y é o vértice fora de S que está mais perto de W.

### Implementação do Algoritmo

Uma implementação ingênua do algoritmo é ineficiente quando cada iteração recalcula a franja da árvore radicada, mesmo que seja quase igual à da iteração prévia. Para obter mais eficiência, acrescenta-se à árvore os melhores arcos da franja, ainda que não definitivos e que sejam trocados em próximas iterações. O que leva a uma segunda descrição do algoritmo, num menor nível de abstração.

Uma segunda descrição do algoritmo depende da ideia de vértice maduro. Um vértice do grafo G é considerado maduro caso seu leque de saída já foi examinado. Todos os demais vértices de G são considerados imaturos. É claro que todo vértice maduro pertence à árvore em construção.

Cada iteração do algoritmo começa com um vetor de pais PA que representa uma árvore radicada S com raiz R, um potencial DIST em G, e um conjunto de vértices maduros. No início da primeira iteração, R é o único vértice de S, todos os vértices imaturos, DIST[s] vale 0, e DIST[V] vale infinito para todo V diferente de R. O processo iterativo consiste em (enquanto T tiver vértices imaturos):

* Seja y um vértice imaturo de S que minimiza DIST[];
* Para cada arco y-z de G que está tenso, faça DIST[z] = DIST[y] + C e PA[z] = y;
* Declare y maduro.

Para C = custo do arco y-z. Por definição, um arco y-z está tenso em relação a DIST[] se DIST[y]+C < DIST[z], relaxado se DIST[y]+C ≥ DIST[z], e justo se DIST[y]+C ≡ DIST[z]. Todo arco y-z em que DIST[y] < ∞ e DIST[z] ≡ ∞ está tenso.

Para provar que o algoritmo de Dijkstra está correto, é preciso verificar que as seguintes propriedades invariantes valem no início de cada iteração:

* PA representa uma árvore radicada S, que tem raiz R;
* Todos os arcos de S são justos e DIST[x] < ∞ se e somente se x pertence a S;
* Para todo vértice maduro v, os arcos v-w de G está relaxado em relação a DIST.

Verificados os invariantes, no início da última iteração. Todos os vértices de S estão maduros. Pelo invariante 3, todo arco de G com ponta inicial em S está relaxado em relação a DIST. Para todo arco v-w de G, se v está em S então DIST[w] < ∞ e w está em S, conforme o invariante 2. Todo vértice ao alcance de R em G pertence a S. Se supusermos, para simplificar a discussão, que todo vértice de G está ao alcance de R, podemos concluir que S é geradora de G.

Se nem todos os vértices estiverem ao alcance de R, o algoritmo produz uma ACB do subgrafo de G induzido pelos vértices que estão ao alcance de R.

### Primeira Implementação

A primeira implementação do algoritmo de Dijkstra supõe que o grafo G é representado por listas de adjacência com custos e, portanto, para cada V e cada A em G➔ADJ[V], o arco que vai de V a A➔W tem custo A➔C. O conjunto dos vértices maduros é representado pelo vetor característico MADURO. O infinito é representado pela constante MAX. Supõe que o valor da expressão DIST[V] + A➔C é sempre menor que MAX. Para garantir isso, a soma de todos os custos deve ser menor que MAX, que também garantirá que não há overflow aritmético.

Essa implementação examina cada arco do grafo uma única vez. Quando aplicada a um grafo com V vértices e A arcos, a função consome tempo proporcional a V² + A, no pior caso. Como A < V², o consumo de tempo da função é proporcional a V², no pior caso. Pode-se dizer que a função é linear quando aplicada a grafos densos, uma vez que o tamanho deles é proporcional a V².

### Segunda Implementação

A primeira implementação do começa cada iteração examinando os vértices imaturos, um a um, a procura de algum que minimize DIST. Para acelerar esse processo, esta implementação mantém, uma fila priorizada "de mínimo", os vértices imaturos. É mais simples colocar na fila todos os vértices imaturos, mesmo os que estão fora de S.

A maneira comum de implementar a fila priorizada usa estrutura de heap. Assim como na implementação anterior, se nem todos os vértices estiverem ao alcance de R, o vetor PA produzido pela função representará uma ACB do subgrafo induzido pelos vértices que estão ao alcance de R.

Supondo que o grafo tem V vértices e A arcos. Se a fila priorizada for implementada em um heap, todas as operações sobre a fila serão executadas em tempo limitado por log V. Nesse caso, a implementação consumirá tempo proporcional a (V+A) log V, no pior caso. Se A ≥ V − 1, como acontece em muitas aplicações, o consumo de tempo será proporcional a A log V, no pior caso. Pode-se dizer que é linearítmica.

A segunda implementação é mais sofisticada que primeira, pois usa uma fila priorizada. Apesar disso, ela é assintoticamente mais lenta que a primeira quando A é próximo de V².

Se restringirmos a grafos esparsos, a segunda implementação é assintoticamente mais rápida que primeira. Se restringirmos a grafos densos, a relação se inverte.

# Conclusões ou considerações finais

A compreensão e o entendimento desses códigos, juntamente com toda a carga da teoria dos grafos, são conhecimentos muito relevantes para estruturação de códigos e demais modelos de dados.

A aplicação bem desenvolvida desses conceitos aprendidos ao longo do curso, permite ao profissional uma densa e rica quantidade de recursos excelentes a diversas aplicações no mercado e em seus próprios projetos.

# Referências Bibliográficas

FEOFILOFF, P. Planaridade. Ime.usp.br, 2011, Disponível em: www.ime.usp.br/~pf/mac5827/aulas/planar.html, Aceso em: 29 de novembro de 2022

WIKIPEDIA. Grafo ciclo. wikipedia.org, 2019, Disponível em: https://pt.wikipedia.org/wiki/Grafo\_ciclo, Aceso em: 29 de novembro de 2022

FEOFILOFF, P. Busca em profundidade. Ime.usp.br, 2019, Disponível em: www.ime.usp.br/~pf/algoritmos\_para\_grafos/aulas/dfs.html#performance, Aceso em: 29 de novembro de 2022

FEOFILOFF, P. Algoritmo de Prim. Ime.usp.br, 2019, Disponível em: www.ime.usp.br/~pf/algoritmos\_para\_grafos/aulas/prim.html, Aceso em: 29 de novembro de 2022

FEOFILOFF, P. Algoritmo de Kruskal. Ime.usp.br, 2019, Disponível em: www.ime.usp.br/~pf/algoritmos\_para\_grafos/aulas/kruskal.html, Aceso em: 29 de novembro de 2022

FEOFILOFF, P. Algoritmo de Dijkstra. Ime.usp.br, 2020, Disponível em: www.ime.usp.br/~pf/algoritmos\_para\_grafos/aulas/dijkstra.html, Aceso em: 29 de novembro de 2022