**IMT – Instituto Mauá de Tecnologia**

**Tópicos Avançados em Estrutura de Dados – Tarefa T11**

**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](#_Toc120909320)

[1.1 Palavras-chave 3](#_Toc120909321)

[2. Introdução 3](#_Toc120909322)

[3. Desenvolvimento do Tema 3](#_Toc120909323)

[3.1 Planaridade de Grafos 3](#_Toc120909324)

[3.2 Implementação de Busca em Profundidade de Grafos 4](#_Toc120909325)

[3.2.1 Definição 4](#_Toc120909326)

[3.2.2 Desempenho 5](#_Toc120909327)

[3.3 Implementação do Algoritmo de PRIM 5](#_Toc120909328)

[3.3.1 Implementação do algoritmo 6](#_Toc120909329)

[3.3.2 Implementação eficiente 6](#_Toc120909330)

[3.4 Implementação do Algoritmo de Kruskal 7](#_Toc120909331)

[3.5 Implementação do Algoritmo de Dijkstra 7](#_Toc120909332)

[4. Conclusões ou considerações finais 7](#_Toc120909333)

[5. Referências Bibliográficas 7](#_Toc120909334)

# 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

# Introdução

# 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

## Implementação do Algoritmo de Dijkstra

# Conclusões ou considerações finais

# 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