CENTRO UNIVERSITÁRIO UNIRUY WYDEN
Campus Paralela

FELIPE SOUZA SANTOS

ANÁLISE E PROJETO DE ALGORITMOS

ORIENTADOR: HELENO CARDOSO

2024

Salvador/BA

FELIPE SOUZA SANTOS

ANÁLISE E PROJETO DE ALGORITMOS

Trabalho apresentado à disciplina Análise e Projeto de Algoritmos do Curso de Engenharia de Computação do Centro Universitário Uniruy Wyden - Campus Paralela, como requisito parcial para a obtenção de nota final.

Orientador: Professor Msc Heleno Cardoso, Doutorando Ciência da Computação PPGCOMP UFBA

2024

Salvador/BA

Sumário

RESUMO

Este trabalho apresenta uma abordagem detalhada sobre a análise e o projeto de algoritmos, abordando sua importância e aplicação na área de computação. O estudo examina os principais conceitos que fundamentam o desempenho dos algoritmos, como complexidade de tempo e espaço, estruturas de dados, e técnicas de otimização. Além disso, são discutidas as etapas de projeto de algoritmos, com foco em práticas que visam a criação de soluções eficientes e escaláveis para problemas complexos. A relevância desse tema é ressaltada pela crescente demanda por sistemas computacionais rápidos e robustos, capazes de processar grandes volumes de dados de forma eficaz. Ao final, são sugeridas direções futuras para pesquisa, considerando avanços em áreas como computação quântica, eficiência energética e segurança.

Palavras-chave: Algoritmos. Estudo. Computação. Soluções.

ABSTRACT

This work presents a detailed approach to the analysis and design of algorithms, addressing their importance and application in the area of computing. The study examines the main concepts that underlie the performance of algorithms, such as time and space complexity, data structures, and optimization techniques. In addition, the algorithm design steps are discussed, focusing on practices that aim to create efficient and scalable solutions to complex problems. The relevance of this topic is highlighted by the growing demand for fast and robust computing systems, capable of processing large volumes of data effectively. At the end, future directions for research are suggested, considering advances in areas such as quantum computing, energy efficiency and security.

Keywords: Algorithms. Study. Computing. Solutions.

1. INTRODUÇÃO

Para início do conteúdo, é de suma importância entender o conceito de **algoritmo** e a sua importância no mundo da computação, além das suas aplicações. Um algoritmo pode ter diferentes definições, tendo a principal como uma sequência finita de passos ou instruções não ambíguas para solucionar um problema. Com um algoritmo, são realizadas operações precisas para obter soluções e resolver um contratempo. No mundo computacional, o estudo dos algoritmos é fundamental visto que exercem um papel importante no desenvolvimento de software e na resolução de problemas computacionais. Basicamente, os algoritmos são a espinha dorsal da computação, permitindo que problemas sejam solucionados de forma eficaz e escalável. Sem a presença dos algoritmos, seria impossível realizar até mesmo tarefas simples. Eles são fundamentais para garantir a automação de processos, a precisão e a velocidade no mundo globalizado. Algoritmos possuem características importantes como:

- Entrada: São os dados ou informações de entrada, recebidas para o processamento.
- Saída: Resultado produzido, sendo ao menos uma quantidade após a execução do algoritmo.
- Instruções: Passos detalhados que definem as ações a serem executadas pelo algoritmo para atingir seu objetivo e resolver um problema.
- Condição de parada: Assegura que o algoritmo não continuará de forma indefinida e que o resultado será produzido em um tempo finito.

Algoritmos também possuem propriedades essenciais para garantir que estejam corretos, eficientes e adequados para resolução de um problema específico. Eles devem satisfazer os seguintes critérios:

- Finitude: O algoritmo termina após um número finito de passos. Essa propriedade garante que o algoritmo não entre em um ciclo infinito e sempre termine em um tempo razoável, gerando um resultado.
- Definitude: Cada instrução do algoritmo deve ser clara e sem ambiguidade. A definitude garante que todas as instruções do algoritmo sejam compreensíveis e precisas. Esse processo é fundamental para que o algoritmo seja executado de forma correta.
- Eficácia: Faz com que o algoritmo seja executável de forma ágil e eficiente, aprimorando o uso de recursos computacionais.

Dadas as propriedades importantes dos algoritmos, onde nos mostram que sem elas, o algoritmo poderia ser ambíguo, ineficiente ou até mesmo inutilizável, é imprescindível destacar exemplos práticos, do simples ao complexo, que podem ser resolvidos com algoritmos. Nas atividades diárias, o algoritmo pode se encaixar na recomendação de cardápios, controle de temperatura de ambientes e até mesmo na realização de compras de forma online. Levando em consideração o controle de temperatura de ambientes, o problema principal seria ajustar de forma automática a temperatura de um ambiente para manter um conforto térmico mais adequado. Para isso, seria utilizado um algoritmo de controle de sistemas, tendo como exemplo o termostato, onde ajustes automáticos da temperatura de um ambiente seriam realizados com base no desejo do usuário e na leitura de sensores de movimento. Já na situação de compras online, existem sites que utilizam algoritmos para realizar comparativos de preços de produtos, mostrando as opções mais baratas, de acordo com a vontade do cliente.

2. ALGORITMOS E ESTRUTURAS DE DADOS

Para análise e projeto de algoritmos de forma prática e eficaz, a escolha de estrutura de dados é uma decisão significativa. Existem maneiras importantes pelas quais as estruturas de dados influenciam a eficiência dos algoritmos, sendo a complexidade de espaço uma delas. Na complexidade de espaço, destaca-se o uso de memória, onde algumas estruturas de dados, como listas encadeadas, podem consumir mais memória devido à necessidade de armazenar ponteiros, enquanto vetores podem ser mais compactos, porém, exigem espaço contíguo. Além disso, pode ocorrer o crescimento dinâmico de estruturas como listas dinâmicas, onde podem aumentar seu tamanho conforme necessário, mas podem também ter um custo adicional durante operações de redimensionamento.

Deve-se salientar a importância na escolha de estrutura de dados para melhor competência dos algoritmos, visto que essas estruturas são essenciais para a organização e manipulação de informações em um software. Elas fornecem mecanismos básicos que permitem que os algoritmos funcionem de maneira satisfatória.

- Pilhas (stacks): É uma estrutura de dados em que o acesso é restrito ao elemento mais recente na pilha, seguindo o princípio conhecido como LIFO (Last In, First Out), onde o último elemento adicionado é o primeiro a ser removido. Tem como operações básicas o

push (adicionar), *pop* (remove) e *peek (*acessar o topo). Além disso, não proporciona o acesso direto, ou seja, apenas o topo pode ser acessado diretamente.

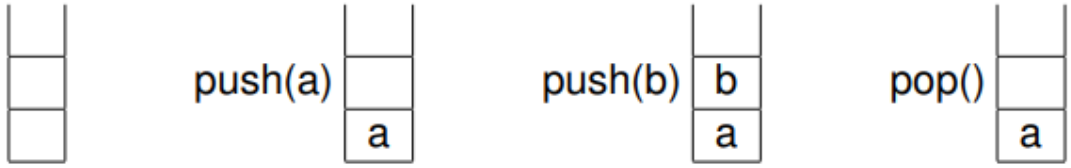


Imagem 1. Exemplo de estrutura de dados de Pilhas

- Filas (queues): É uma estrutura em que o acesso é restrito ao elemento mais antigo, seguindo o princípio conhecido como FIFO (First In, First Out), onde o primeiro elemento adicionado é o primeiro a ser removido. Tem como operações básicas a adição (*enqueue*) e a remoção (*dequeue*). Além disso, não proporciona o acesso direto, ou seja, apenas o início e o fim da fila podem ser acessados diretamente.

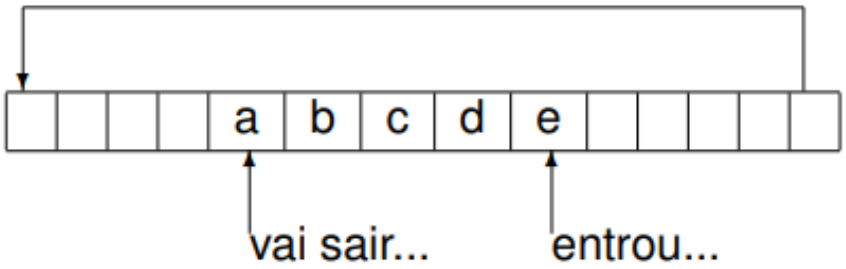


Imagem 2. Exemplo de estrutura de dados de Filas

- Listas encadeadas: Uma coleção de elementos chamados **nós**, onde cada nó contém um valor e um ponteiro para o próximo nó na sequência. Tem como características o acesso sequencial, onde O(n) para acessar um elemento em uma posição específica e o tamanho dinâmico, podendo crescer e encolher conforme necessário. Essa estrutura pode ser usada na implementação de filas, pilhas e listas dinâmicas.

Além das estruturas citadas, existem outras importantes, como árvores, grafos, heaps e arrays. Logo, a escolha da estrutura de dados correta depende das operações que precisam ser realizadas, da complexidade dos dados e dos requisitos de desempenho. É de extrema importância o entendimento das estruturas fundamentais, uma vez que, de acordo com as suas características, podem ajudar na criação de soluções otimizadas para problemas relacionados à computação e até mesmo na aplicação de atividades diárias como organização de horários e planejamento de rotas de trânsito.

3. PARADIGMA DA DIVISÃO E CONQUISTA

A ideia do paradigma dividir-e-conquistar é fazer com que o problema principal seja dividido em problemas menores, ou seja, em “subproblemas” menores, sendo estes resolvidos parte a parte e, ao final, realizar a combinação para a resolução definitiva. Destaca-se a importância de **Dividir** para conquistar, onde o problema é dividido em determinado número de subproblemas. Já a **Conquista** trata-se de resolver os subproblemas recursivamente. Além disso, é interessante **Combinar** as soluções fornecidas pelos subproblemas, a fim de produzir a solução para o problema original. Logo, esse método é significante na resolução de problemas complexos de forma eficiente, principalmente em casos onde a divisão do problema em partes menores facilita a solução. Na imagem abaixo, é possível visualizar um passo do método citado, sendo que cada passo de dividir cria dois subproblemas (importante destacar que alguns algoritmos criam mais de dois):

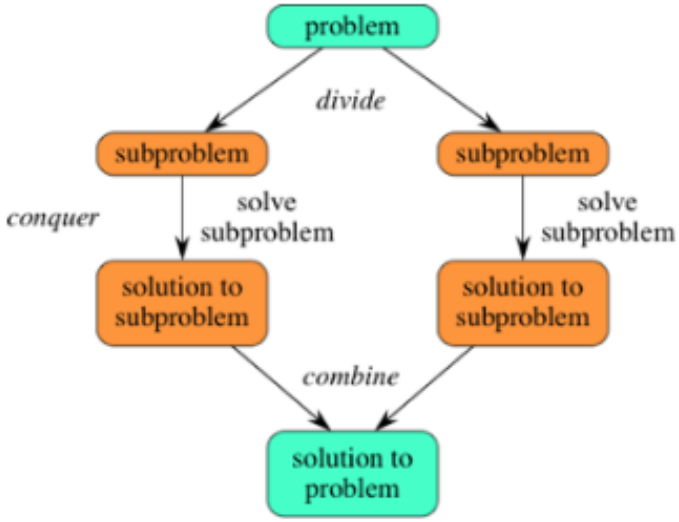


Imagem 3. Surgimento de subproblemas utilizando o método dividir-e-conquistar

Discorrendo de forma aprofundada sobre a construção de soluções, a técnica de divisão e conquista segue três etapas principais:

- Divisão: Primeiro passo, onde ocorre a divisão do problema em subproblemas menores. Essa etapa geralmente continua recursivamente até que os problemas menores se tornem pequenos, de modo que possam ser resolvidos diretamente.
- Conquista: Resolução dos subproblemas menores de maneira recursiva, ou seja, aplicar a mesma técnica de resposta que foi usada para o problema inicial, porém, em instâncias menores do problema.
- Combinação: Após a resolução dos subproblemas, as soluções são combinadas, de forma que a solução final do problema original seja obtida.

Para melhor entendimento do método de divisão e conquista, existem dois algoritmos de ordenação que são baseados neste paradigma: MergeSort e QuickSort. Contudo, eles diferem em seus mecanismos de divisão, conquista e combinação, de acordo com a análise apresentada a seguir. O MergeSort utiliza a técnica para ordenar um array seguindo as três etapas principais.

Exemplo: Ordenar [38, 27, 43, 3, 9, 82, 10] com **MergeSort**

1. Dividir em [38, 27, 43] e [3, 9, 82, 10];

2. Dividir novamente [38, 27, 43] em [38] e [27, 43] e assim por diante;
3. Combinação dos subarrays: [38] e [27, 43] em [27, 38, 43] e [3, 9, 82, 10] em [3, 9, 10, 82];
4. Combinação final dos subarrays: [27, 38, 43] e [3, 9, 10, 82] em [3, 9, 10, 27, 38, 43, 82].

O algoritmo MergeSort possui vantagens e desvantagens, tendo como a vantagem principal a preservação da ordem dos elementos iguais, situação que é importante em alguns cenários. Já como desvantagem, tem a necessidade de espaço adicional que seja proporcional ao tamanho do array em questão, o que pode ser um problema em casos de limite de memória. Já o QuickSort, em comparação com o MergeSort, difere na maneira de como o array é dividido. Veremos a situação no exemplo abaixo:

Exemplo: Ordenar [38, 27, 43, 3, 9, 82, 10] com **QuickSort**

1. Necessário escolher um pivô -> 10;
2. Particionamento do array: [3, 9, 10] e [38, 27, 43, 82];
3. Recursão às duas partições [3, 9] já está ordenado e [38, 27, 43, 82] será dividido novamente, com a escolha de outro pivô, até que tudo esteja ordenado.

O algoritmo QuickSort possui vantagens e desvantagens, tendo como a vantagem principal a rapidez na prática devido ao número menor de operações de movimentação de dados. Já como desvantagem, o algoritmo não é estável, o que significa que a ordem relativa de elementos iguais pode ser modificada. Logo, nota-se que a escolha pelo algoritmo MergeSort é estável e eficiente, porém pode exigir mais espaço na memória. Já no algoritmo QuickSort, o destaque é para a rapidez, além de necessitar de menos memória. Todavia, o desempenho depende de uma boa escolha do pivô para evitar o pior caso de $O(n^2)$. Ambos os algoritmos são utilizados e a escolha deles depende das características do problema apresentado e das restrições do sistema em uso.

4. GRAFOS

No mundo da computação, os grafos são elementos importantes pois são uma abstração computável para boa parte das relações da realidade e da imaginação humana. Um grafo é composto por um conjunto discreto de elementos que representam a existência de algo material

ou imaginário. Há o relacionamento entre estes elementos e também uma regra, ou um conjunto de regras, definindo estas relações pela lógica. Outros elementos do conjunto discreto em questão são interconectados por relações de um padrão de incidência bem definido, e estas relações fazem parte do grafo - elas são expressas por *arestas,* enquanto os elementos interconectados são expressos por *vértices*, de acordo com a imagem a seguir:

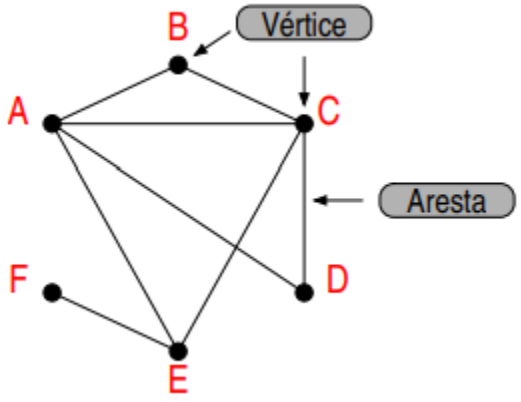


Imagem 4. Objetivos especiais: vértices e arestas

A teoria dos grafos, proposta por Leonhard Euler em 1736, traz a concepção deste universo de ideias com vértices e arestas. Esta teoria matemática trata de relações entre elementos de conjuntos discretos. Ela é empregada em algoritmos para abstrair objetos do mundo real ou imaginário que são inter-relacionados de alguma maneira. A representação computacional de um grafo deve usar uma estrutura que corresponda de forma única a um grafo dado e que possa ser armazenada e manipulada em um computador. Há critérios que devem ser seguidos em algumas estruturas, sendo um deles a quantidade de memória.

- Listas de arestas: Podemos representar um gráfico de forma simples, com uma lista, ou arranjo e |E| arestas. Para representar uma aresta, deve existir um arranjo de dois números de vértice, ou um arranjo de objetos que contém os números dos vértices sobre os quais as arestas são incidentes. Caso as arestas tenham pesos, um terceiro elemento ao arranjo deve ser adicionado, fornecendo o peso da aresta. Como cada aresta contém apenas dois ou três números, o espaço total para uma lista de arestas é θ(E).

- Matrizes de adjacências: Com uma matriz de adjacência, podemos descobrir se uma aresta está presente em um tempo constante, ou simplesmente procurando pela entrada correspondente na matriz.
- Listas de adjacências: Combina as matrizes de adjacência com as listas de arestas. Ela é usada para armazenar a estrutura de um grafo de forma eficaz, especialmente em termos de uso, sendo mais adequada para grafos esparsos (poucas arestas em relação ao número de vértices). Nessa estrutura, cada vértice do grafo tem uma lista que armazena os vértices adjacentes a ele, ou seja, os vértices aos quais ele está conectado por uma aresta.

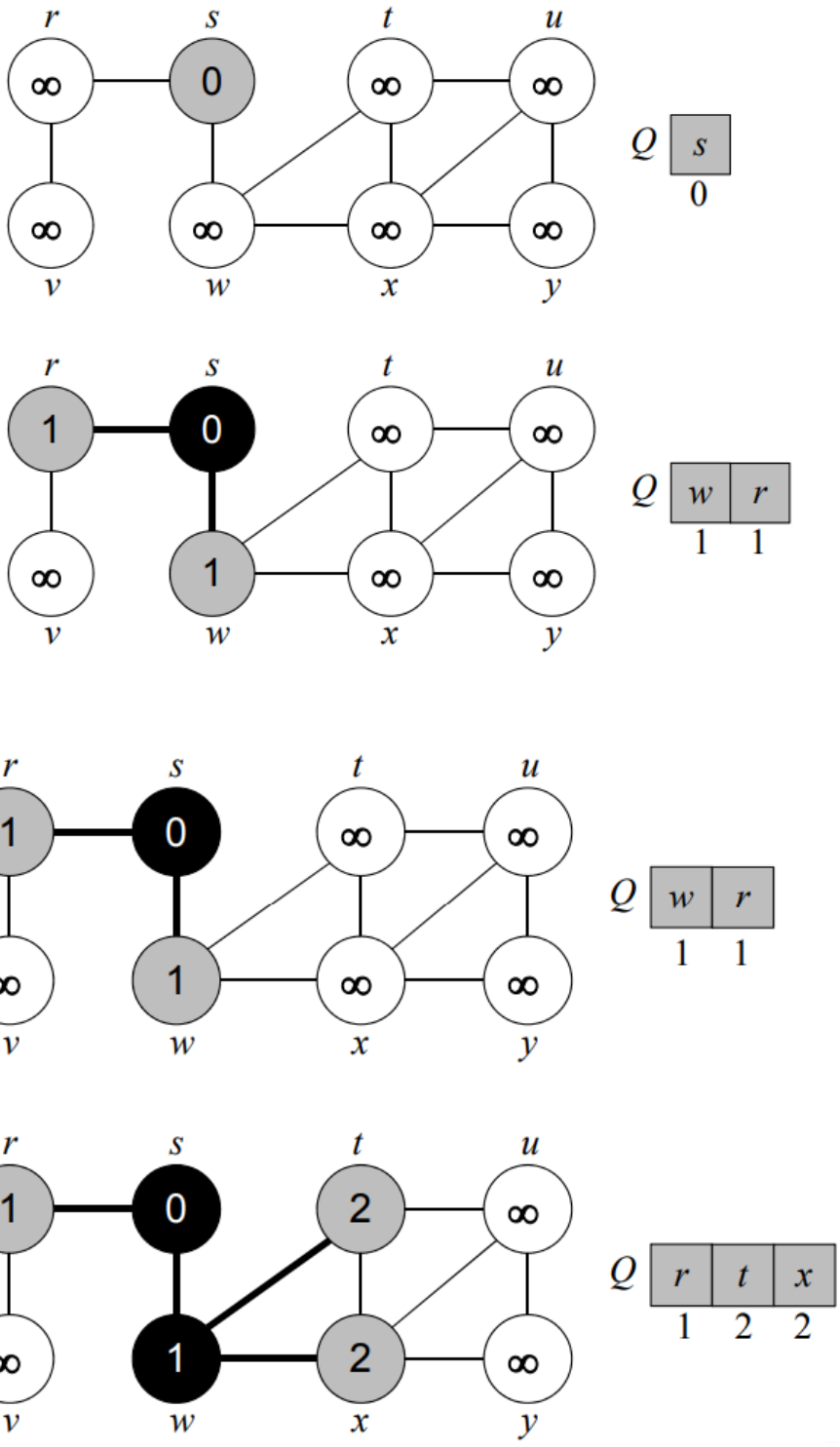
A tabela abaixo apresenta a diferença entre matriz de adjacência e lista de adjacência:

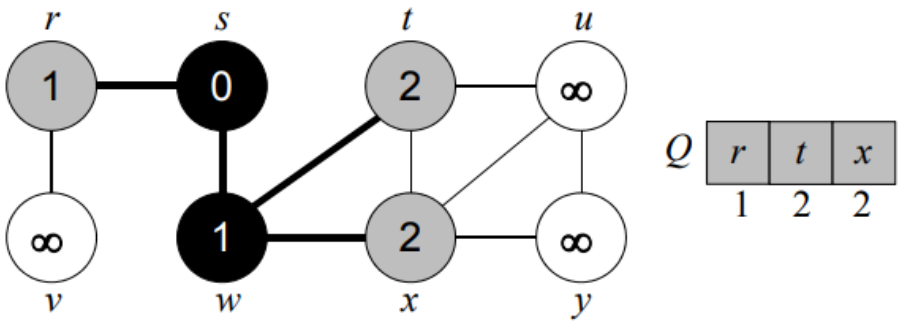
CRITÉRIO	MATRIZ DE ADJACÊNCIA	LISTA DE ADJACÊNCIA
ESTRUTURA	Matriz Bidimensional (Tamanho V V, onde o V É Número De Vértices)	Lista de listas, onde cada vértice aponta para seus vizinhos
ESPAÇO OCUPADO	$O(V^2)$, onde V é o número de vértices	O(V + E), onde E é o número de arestas (mais eficiente para grafos esparsos).
IDEAL PARA GRAFOS	Densos (com muitas arestas)	Esparsos (com poucas arestas em relação ao número de vértices)
VERIFICAÇÃO DE EXISTÊNCIA DE ARESTA	O(1) (acesso direto via índices da matriz)	O(deg(v)), onde `deg(v)` é o grau do vértice (precisa percorrer a lista)
INSERÇÃO DE ARESTA	O(1) (modificação direta na matriz)	O(1) (adiciona elemento na lista correspondente)
REMOÇÃO DE ARESTA	O(1) (remoção direta na matriz)	O(deg(v)) (precisa percorrer e remover o elemento da lista)
ACESSO AOS VIZINHOS DE UM VÉRTICE	O(V) (precisa percorrer a linha correspondente do vértice)	O(deg(v)) (acesso direto à lista de vizinhos)
ESPAÇO PARA GRAFOS NÃO DIRECIONADOS	$O(V^2)$	O(V + E) (não há duplicação de arestas, pois a direção é única)
COMPLEXIDADE DE BUSCA	$O(V^2)$ (por conta de percorrer a matriz)	O(V + E) (mais eficiente para grafos esparsos)
USO PRÁTICO	Melhor quando o grafo é muito	Melhor para grafos esparsos

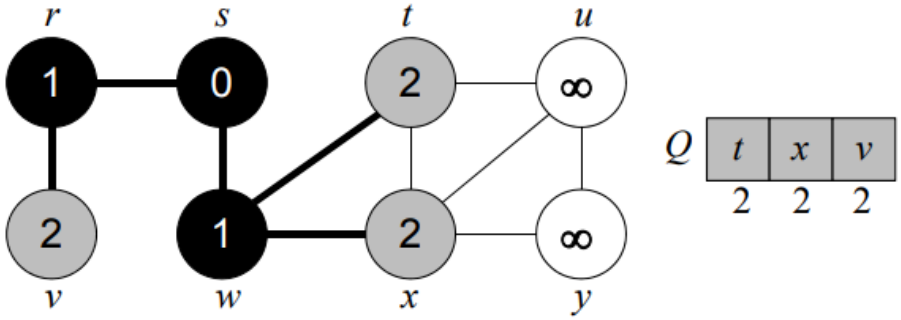
	denso ou para grafos muito pequenos.	(com muitos vértices e poucas arestas).

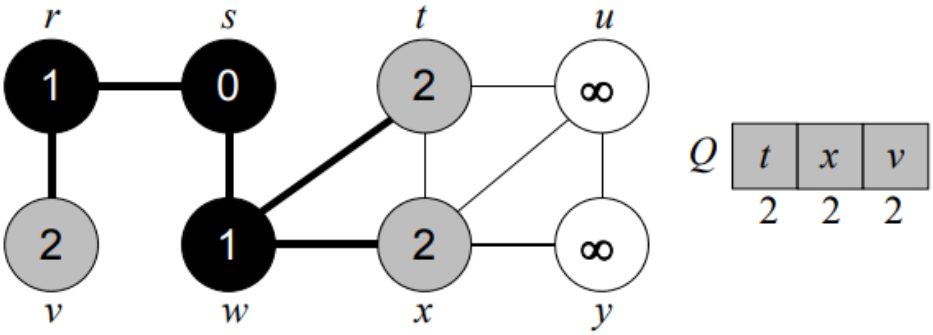
Para melhor entendimento ao realizar uma busca em um grafo, existem dois algoritmos super importantes, onde a análise será feita a seguir. Mas qual o principal objetivo de realizar busca em grafos? Basicamente, os objetivos da busca são determinar quais vértices são alcançáveis através de um vértice inicial, ou seja, examinar as arestas do grafo de forma sistemática, e verificar se um determinado objeto está presente no grafo. Detalhando sobre os métodos de busca, temos:

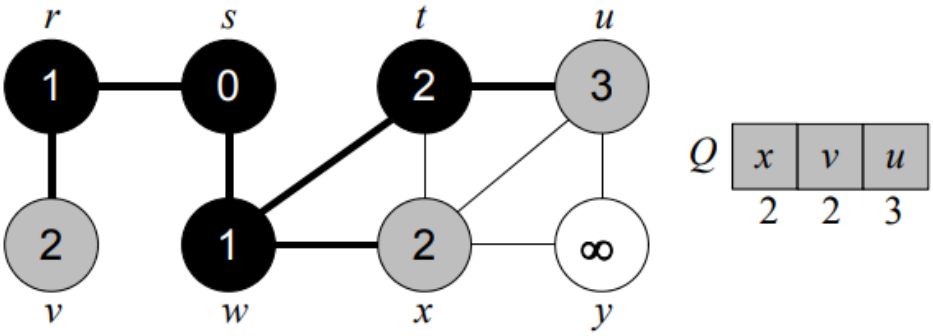
- Breadth-First Search (BFS): O método de busca em largura é um dos algoritmos mais simples para se pesquisar um grafo e é arquétipo de muitos algoritmos de grafos importantes. Esse método baseia-se partindo de um vértice *s* (origem) explorando sistematicamente as arestas do grafo e determinando cada vértice acessível a partir da origem. Para entender a sua lógica de funcionamento, deve-se definir o nó inicial, marcar como visitado e adicioná-lo à fila. Depois disso, enquanto a fila não estiver vazia, remove-se o primeiro nó da fila e para cada *v* deste primeiro nó (nomeando o primeiro nó como *u)*, se *v* não foi explorado, marcamos *v* como explorado e o adicionamos à fila. Isso deve ser repetido de outro inicial, se houver.

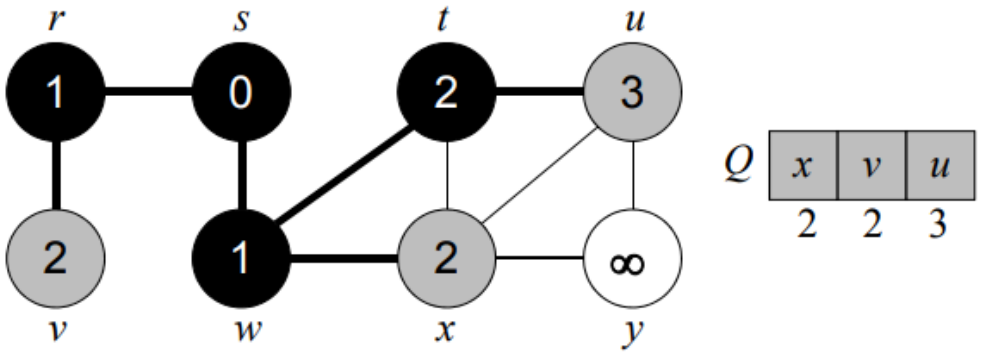

r
s
t
u
∞
0
∞
∞
∞
∞
∞
∞
v
w
x
y
Q
s
0
r
s
t
u
1
0
∞
∞
∞
1
∞
∞
v
w
x
y
Q
w
r
1
1
r
s
t
u
1
0
∞
∞
∞
1
∞
∞
v
w
x
y
Q
w
r
1
1
r
s
t
u
1
0
2
∞
∞
1
2
∞
v
w
x
y
Q
r
t
x
1
2
2

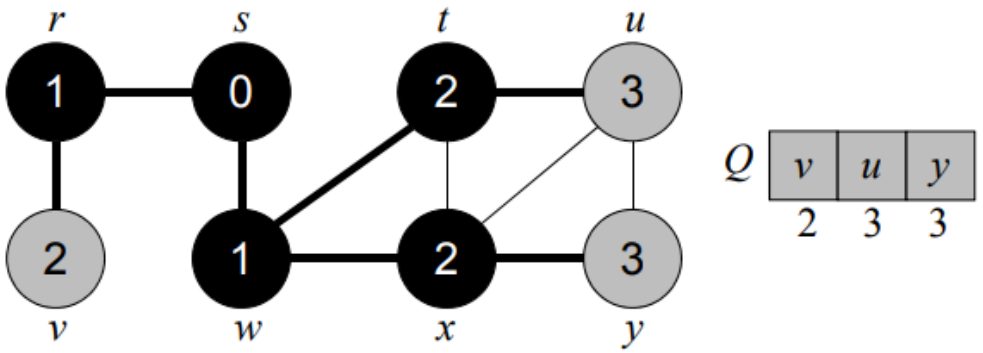

r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q
r t x
1 2 2

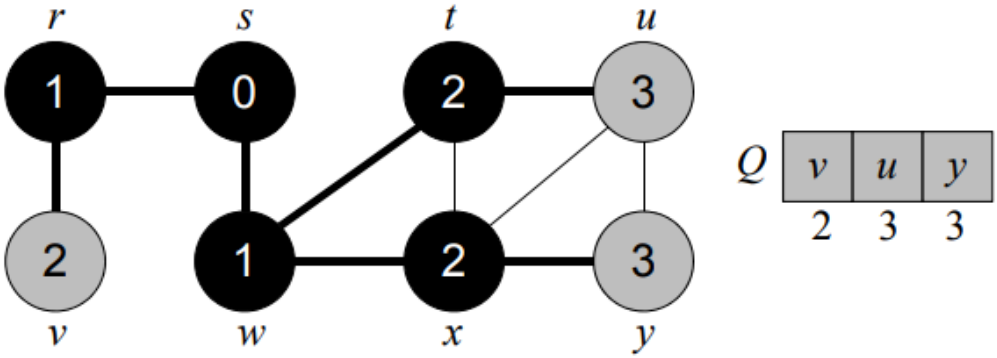

r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2

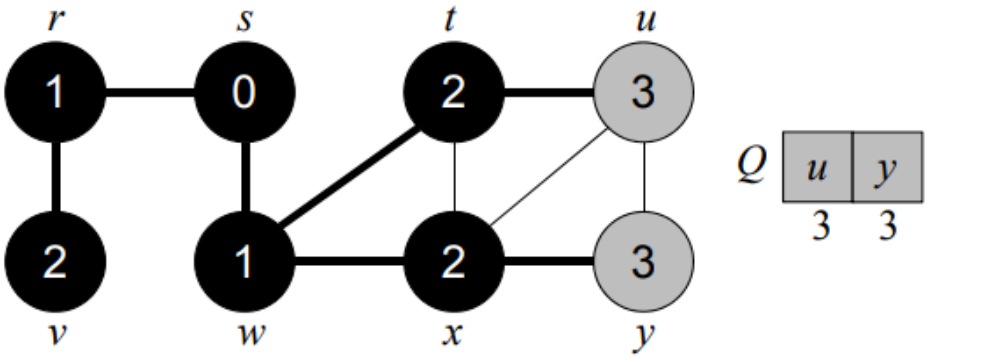

r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2

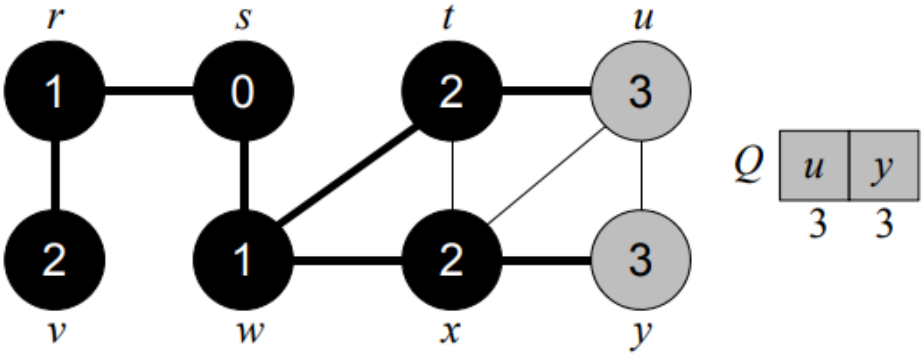

r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3

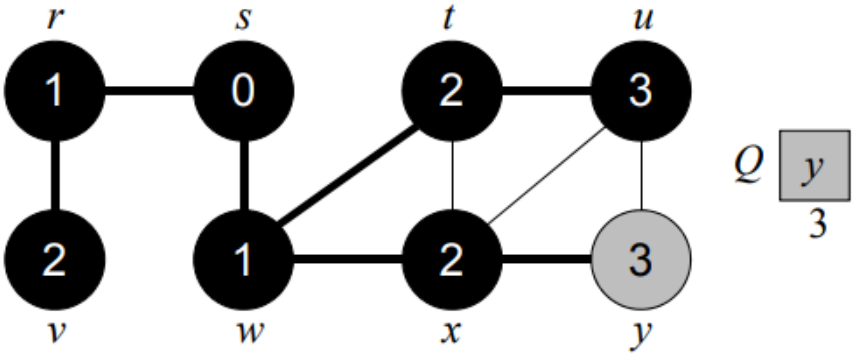

r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3

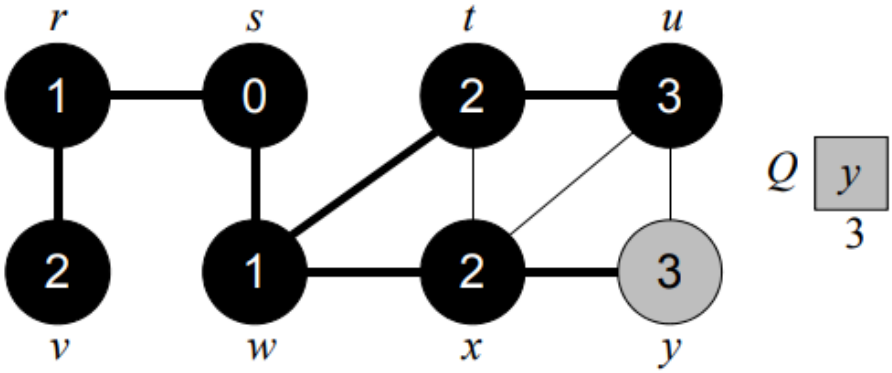

r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3

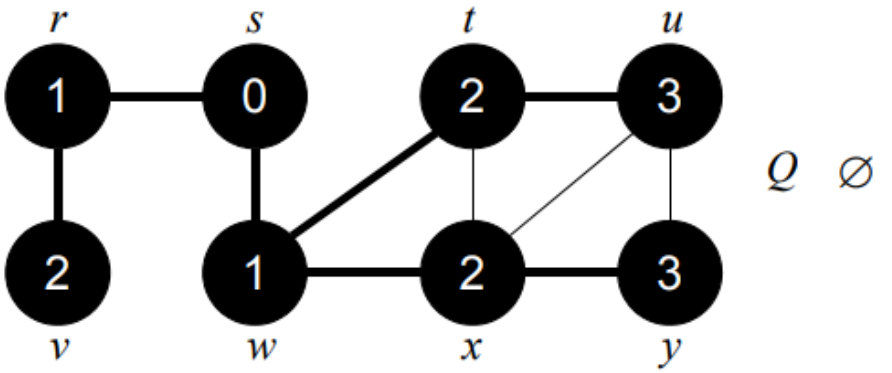

r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3


r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3


r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3

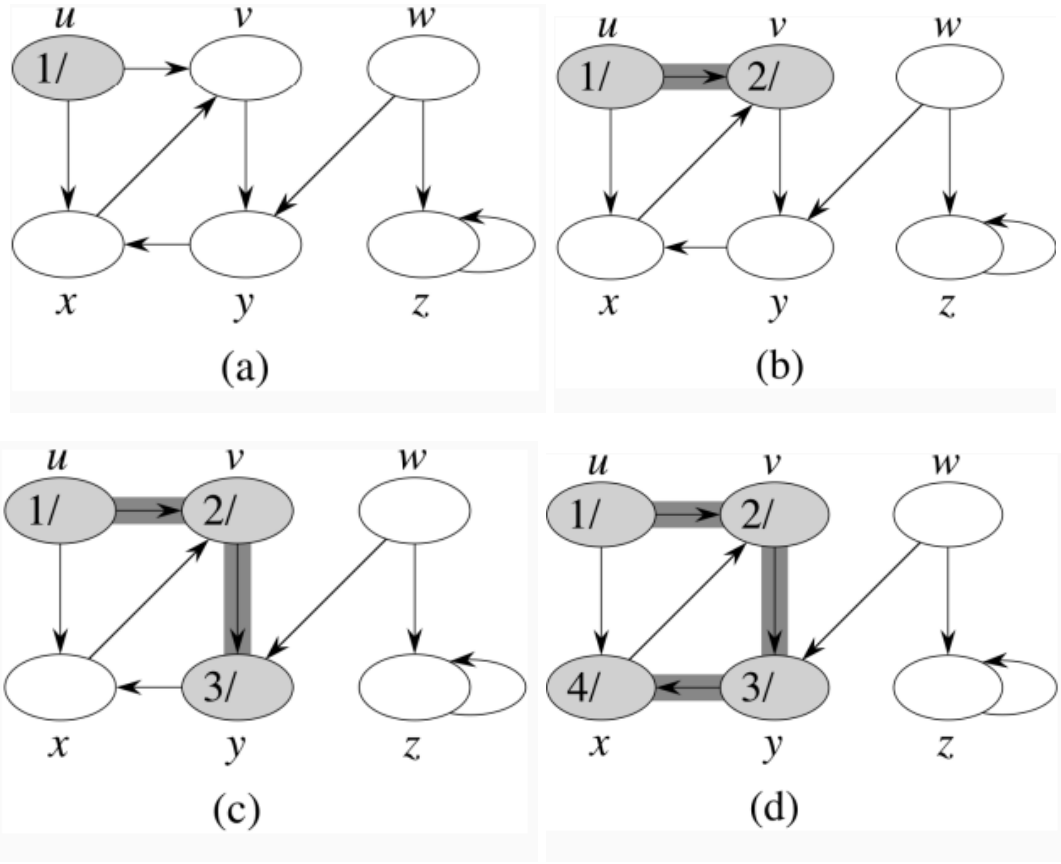

r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3

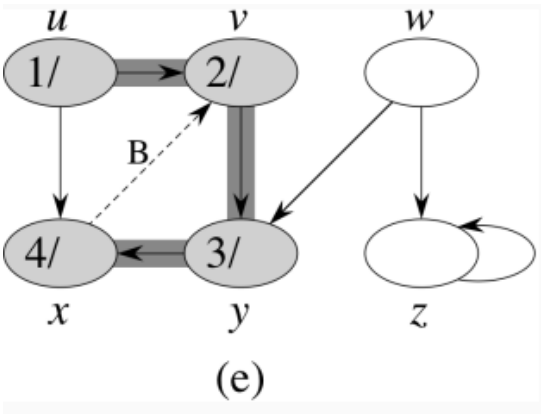

r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3

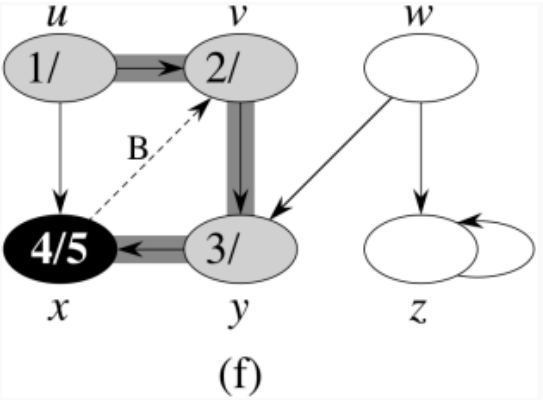

r s t u
1 0 2 3
2 1 2 3
v w x y
Q ∅

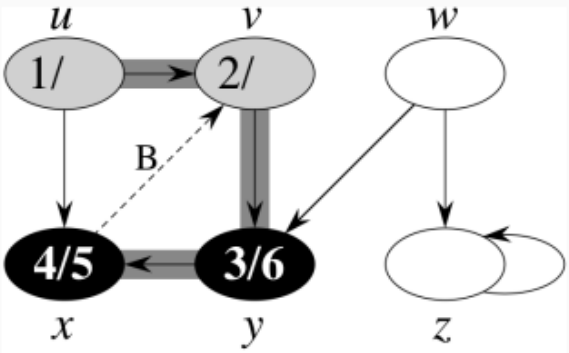
Em resumo, a técnica de busca em largura é poderosa para exploração de grafos em camadas e busca de caminhos mínimos, sendo uma base para muitas aplicações que exigem análise de conectividade e descoberta de distâncias em redes.

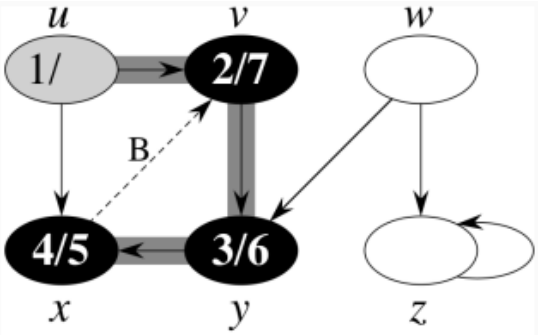
- Depth-First Search (DFS): O método de busca em profundidade é um algoritmo para caminhar no grafo. Seu núcleo se concentra em buscar, sempre que possível, o mais fundo do grafo. As arestas são exploradas a partir do vértice *v* mais recentemente descoberto que ainda possui arestas não exploradas saindo dele. Quando todas as arestas adjacentes ao vértice tiverem sido exploradas, deve ser feita a técnica de backtracking, onde a busca "anda para trás" para explorar vértices do qual "*v*" foi descoberto. O processo continua até que sejam descobertos todos os vértices que são alcançáveis a partir do vértice original. **Caso todos os vértices sejam descobertos, então é o fim da busca.** Caso contrário, o processo continua a partir de um novo vértice de origem ainda não descoberto (grafos desconexos). Abaixo, há um exemplo de execução da busca citada:

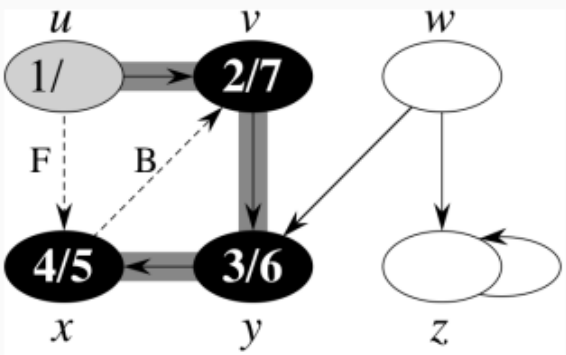


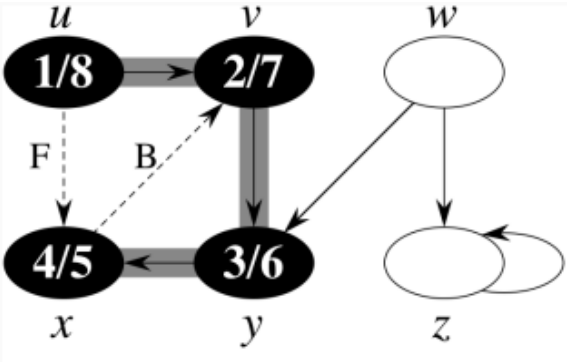

u
v
w
1/
2/
B
4/
3/
x
y
z
(e)

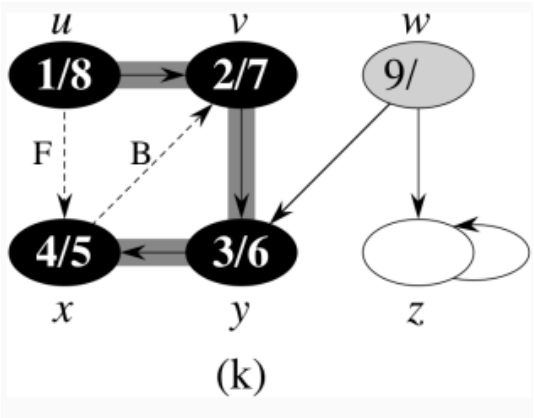

u
v
w
1/
2/
B
4/5
3/
x
y
z
(f)

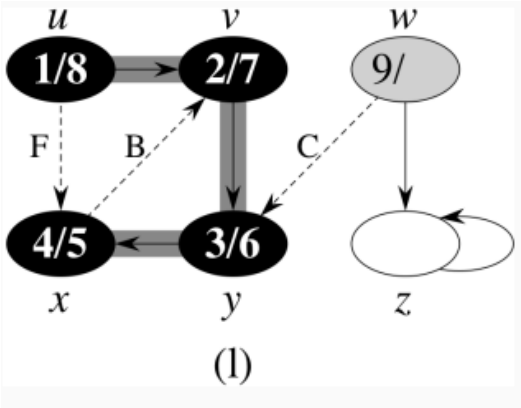

u
v
w
1/
2/
B
4/5
3/6
x
y
z
(g)

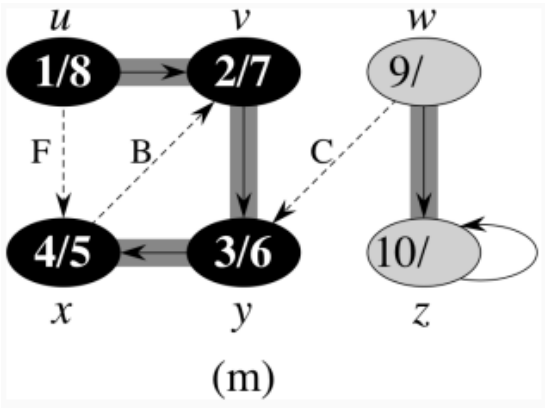

u
v
w
1/
2/7
B
4/5
3/6
x
y
z
(h)

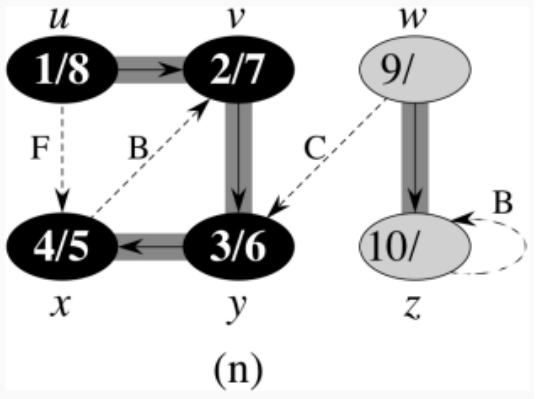

u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z
(i)

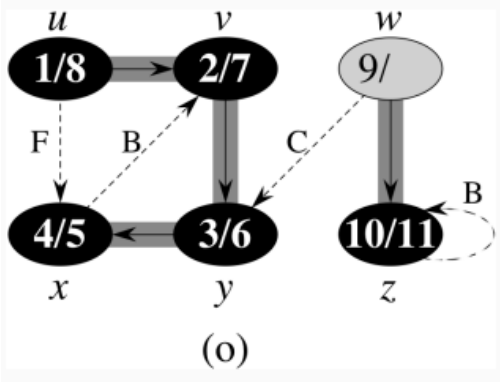

u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z
(j)

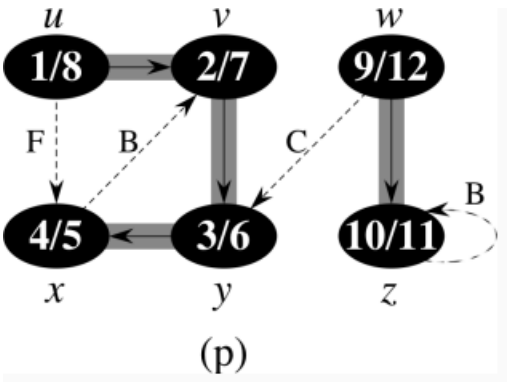












Tempo de execução:

- Os laços nas linhas 1 a 3 e nas linhas 5 a 7 de DFS demoram tempo Θ(V), sem contar o tempo das chamadas a DFS-Visit.
- O procedimento DFS-Visit é chamado exatamente uma vez para cada vértice, isto porque DFS-Visit é chamado para vértices brancos (vértice ainda não visitado), e no início de

DFS-Visit o vértice é pintado de cinza (visitado, mas seus adjacentes ainda não foram visitados).

- Durante a execução de DFS-Visit (v), o laço nas linhas 4 a 7 é executado G.Adj[v] vezes, como o $\sum_{v \in V} |G.Adj[v]| = \Theta(E)$, o custo total da execução das linhas 4 a 7 de DFS-Visit é $\Theta(E)$.
- Portanto, o tempo de execução de DFS é $\Theta(V + E)$.

Em resumo, a busca em profundidade é um método útil para encontrar componentes conectados, detectar ciclos e solucionar problemas de caminhamento e conectividade. Embora não busque o caminho mais curto, ela é eficaz para explorar todos os caminhos de um grafo, sendo amplamente usada em muitas aplicações de análise de grafos e árvores. A seguir, será apresentada uma tabela comparativa, especificando as aplicações e complexidades dos métodos estudados.

CRITÉRIO	BFS	DFS
DESCRIÇÃO	Explora o grafo por camadas, visitando os vizinhos do vértice atual antes de passar para o próximo nível.	Explora o grafo em profundidade, avançando em um caminho até que não haja mais vizinhos inexplorados, retrocedendo então para explorar novos caminhos.
ESTRUTURA UTILIZADA	Fila	Pilha (recursão ou estrutura de pilha explícita)
APLICAÇÕES COMUNS	- Encontrar o caminho mais curto em grafos não ponderados - Encontrar todos os vértices a uma certa "distância" do vértice inicial - Resolver problemas de conectividade em redes	- Detectar ciclos em grafos direcionados e não direcionados - Resolver problemas de conectividade - Caminhamento em árvores
COMPLEXIDADE DE TEMPO	O(V) (para armazenar a fila e o vetor de visitados)	O(V) (para armazenar a pilha de recursão e o vetor de visitados)

TIPO DE CAMINHAMENTO	Em largura, expandindo por camadas	Em profundidade, expandindo por caminhos
ADEQUADO PARA	- Encontrar a menor distância entre vértices em grafos não ponderados - Grafos grandes onde se busca a menor quantidade de nós a partir da raiz	- Explorar todos os caminhos possíveis - Grafos ou problemas onde a solução pode estar mais "profundamente" oculta
DESVANTAGENS	- Não é eficiente em grafos muito grandes devido ao uso de memória com a fila	- Pode atingir níveis de recursão profunda em grafos grandes, levando ao risco de estouro de pilha
PREFERÊNCIA	- Quando o problema exige a menor distância em um grafo não ponderado.	-Quando for necessário explorar todos os caminhos ou fazer uma ordenação em DAGs.
OUTRAS CARACTERÍSTICAS	Garante o menor caminho em termos de número de arestas em grafos não ponderados	Não garante o menor caminho, mas visita todos os vértices e arestas

Em suma, a BFS é ideal para encontrar o menor caminho em termos de arestas e explorar grafos em **largura.** Já o DFS é mais eficiente para explorar **todos os caminhos** de um grafo, especialmente em problemas de conectividade e busca em profundidade.

5. CLASSES DE PROBLEMAS

Para introduzir, é de extrema importância entender o que são os problemas NPC (NP-Completos) e a importância deles para resolução de problemas. Problemas NP-Completos são problemas de decisão que pertencem tanto à classe NP (Nondeterministic Polynomial time) quanto à classe NP-difícil. Informalmente, um problema está na classe NPC se estiver em NP e for tão “difícil” quanto qualquer problema em NP. Eles são conhecidos por serem desafiadores em termos de tempo de execução, já que **não existe um algoritmo conhecido que os resolva de forma eficiente (em tempo polinomial) para todos os casos.** Para que um problema seja considerado NP-completo, ele deve atender a dois critérios:

1. Estar em NP: A solução pode ser verificada rapidamente.
2. Ser NP-difícil: Caso seja possível resolver um problema NP-completo de uma forma diferente, conseguiremos resolver todos os problemas em NP da mesma forma. Utilizando outros termos, o problema NP-completo é um dos mais difíceis de NP.

A imagem abaixo mostra como acredita-se que seja a relação das classes de problemas:

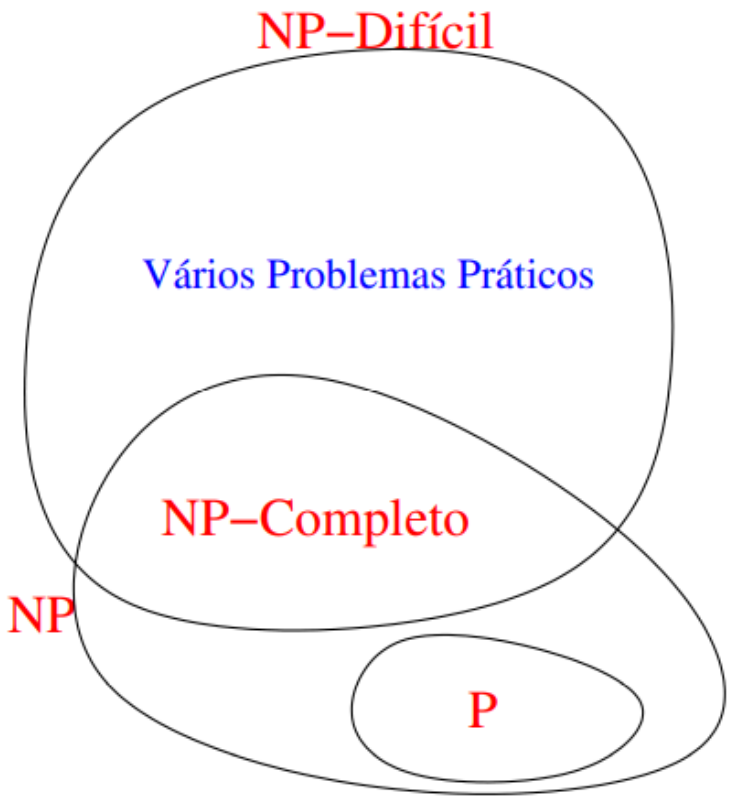


Imagem 5. Relação das classes de problemas

Existem problemas clássicos que se encaixam que se encaixam como NP-completude como o problema da Mochila (Knapsack Problem), onde deve-se decidir como preencher uma mochila com itens de diferentes valores e pesos de modo a maximizar o valor total sem ultrapassar a capacidade de peso, podendo ser aplicado em problemas de orçamento e na criptografia. Além do Knapsack problem, há o problema do Caixeiro Viajante (Traveling Salesman Problem), onde é dado um conjunto de cidades e as distâncias entre elas e deve ser encontrado o caminho mais

curto que visita cada cidade exatamente uma vez e retorna à cidade inicial. Esse exemplo pode ser útil em roteamento de veículos e otimização de rotas de entrega.

Para melhor análise das classes de problemas, é importante discutir a importância de verificar soluções em tempo polinomial. Esta verificação é um conceito central para a teoria da complexidade computacional e está diretamente ligada à classe de problemas **NP**. A verificação de forma rápida de uma solução é importante por inúmeras razões, sendo elas a eficiência em validação de soluções, já que nos problemas classificados como NP, mesmo que não haja uma maneira eficiente de **encontrar** uma solução, **verificar** se uma solução proposta é correta pode ser feito rapidamente (em tempo polinomial). Isso permite que, se tivermos uma solução candidata, possamos validá-la sem precisar gastar muito tempo, tornando a verificação prática em muitos contextos. Além dessa razão, ela também pode servir como base para algoritmos aproximados e heurísticos. Para muitos problemas NP-completos, como a otimização de rotas ou alocação de recursos, as soluções exatas podem ser inviáveis para grandes instâncias. A verificação em tempo polinomial permite que algoritmos heurísticos e aproximados possam gerar soluções plausíveis que, mesmo que não sejam as ideais, possam ser verificadas rapidamente para validar se atendem a um conjunto mínimo de requisitos.

6. ALGORITMOS DE APROXIMAÇÃO

Algoritmos de aproximação, em linhas gerais, são algoritmos que não necessariamente produzem uma solução ótima, mas soluções que estão dentro de um certo fator da solução ótima. O desenvolvimento dos algoritmos de aproximação surgiu em resposta à impossibilidade de se resolver satisfatoriamente diversos problemas de otimização NP-difíceis. Estes algoritmos possuem características importantes como:

- **Qualidade da Aproximação**: Algoritmos de aproximação buscam uma solução cujo valor seja próximo do valor ótimo. A diferença entre o valor da solução obtida e o valor ótimo é quantificada por uma **razão de aproximação** ou **fator de aproximação**.
- **Tempo Polinomial**: São algoritmos desenhados para rodar em **tempo polinomial**, tornando-os aplicáveis a problemas onde os métodos exatos não são viáveis.

- **Limite de Desempenho**: Alguns algoritmos de aproximação vêm com garantias teóricas sobre a qualidade da aproximação em relação ao ótimo. Por exemplo, um algoritmo de aproximação com um fator de aproximação de 2 para um problema de minimização significa que a solução obtida não será pior que o dobro do valor ótimo.

Como esses algoritmos podem ser úteis para Problemas NP-difíceis? Basicamente, eles são fundamentais quando soluções exatas são inviáveis, onde oferecem soluções viáveis mesmo para instâncias grandes em problemas NP-difíceis e são importantes devido a flexibilidade de aplicação, onde, em muitos problemas práticos, uma solução próxima do ótimo é aceitável e suficiente para atingir os objetivos, como na logística, planejamento de redes, e alocação de recursos. Como exemplos de algoritmos de aproximação, temos:

1. **Problema de Partição (Partition Problem)**:

- **Descrição**: Dado um conjunto de números, o problema é verificar se é possível dividi-lo em dois subconjuntos de soma igual.
- **Algoritmo de Aproximação**: O **algoritmo de balanceamento de cargas** é usado para encontrar uma aproximação onde as somas dos subconjuntos são próximas, mas não necessariamente iguais. Em muitas aplicações, essa aproximação é suficiente.
- **Aplicação**: Na divisão de recursos financeiros, alocação de carga em sistemas distribuídos e particionamento de dados.

2. **Problema de Cobertura de Conjuntos (Set Covering Problem)**:

- **Descrição**: Dado um conjunto universal e uma coleção de subconjuntos, o objetivo é escolher o menor número de subconjuntos cuja união cobre todo o conjunto universal.
- **Algoritmo de Aproximação**: O **algoritmo ganancioso (greedy)** é usado para selecionar conjuntos iterativamente, garantindo uma cobertura completa. Esse algoritmo possui uma razão de aproximação de **O(log n)**, onde n é o tamanho do conjunto universal.
- **Aplicação**: Encontrado em otimização de recursos, como cobertura de rede de sensores e planejamento de distribuição de recursos.

Por fim, os algoritmos de aproximação são ferramentas essenciais na resolução de problemas NP-difíceis, pois fornecem soluções rápidas e aceitáveis em tempo polinomial. Para problemas de otimização onde uma solução exata não é necessária ou prática, os algoritmos de aproximação são vitais e aplicáveis em setores como logística, inteligência artificial, finanças, telecomunicações, entre outros.

7. ESTRUTURAS DE DADOS AVANÇADAS

Usadas para otimizar o acesso, armazenamento e a manipulação de dados em aplicações complexas, as estruturas de dados avançadas melhoram a eficiência de algoritmos que precisam lidar com grandes volumes de dados ou operações complexas. Essas estruturas permitem que tarefas que seriam lentas ou inviáveis com estruturas mais simples (como arrays e listas) sejam executadas de forma rápida e eficiente, resolvendo problemas específicos de maneira otimizada. É importante destacar o porquê de utilizar as estruturas de dados avançadas; elas oferecem melhor desempenho em operações de busca, inserção e remoção, o que é essencial para aplicações de alta velocidade, além de oferecerem maior flexibilidade na manipulação de dados complexos e na redução de uso de memória em determinadas tarefas. Sobre as principais estruturas de dados, destacamos as Árvores B e Árvores Red-Black, na qual serão explicadas de forma detalhada a seguir.

- Árvores B (B-trees): São estruturas usadas para implementar tabelas de símbolos muito grandes. Uma árvore B pode ser composta por uma generalização de árvores binárias onde cada nó pode ter mais de dois filhos. Elas foram projetadas para minimizar o número de acessos ao disco, sendo muito eficientes para armazenamento e recuperação de dados em sistemas de arquivos e bancos de dados. Como propriedades, temos:
    1. **N-ária e balanceada**: Uma árvore B de ordem mmm permite até mmm filhos por nó.
    2. **Altura mínima**: A árvore é balanceada em altura, garantindo que todas as folhas estejam no mesmo nível.
    3. **Garantia de espaço**: Cada nó (exceto a raiz) deve estar pelo menos metade cheio.

4. **Nó com múltiplas chaves**: Cada nó possui até m−1m-1m−1 chaves, usadas para dividir o nó em subárvores.

Para manter o balanceamento da estrutura e a garantia de armazenamento e backup dos dados em sistemas que lidam com grandes volumes, as Árvores B possuem operações como:

1. **Busca**: A busca em uma árvore B percorre os nós e compara a chave procurada com as chaves em cada nó, podendo acessar múltiplos filhos até chegar à folha desejada. A complexidade de busca é O(logn)O(\log n)O(logn).
2. **Inserção**: A inserção ocorre sempre em uma folha. Quando um nó está cheio, ele é dividido e a chave intermediária sobe ao nível superior, mantendo a árvore balanceada.
3. **Remoção**: Para remover um elemento, o nó pode ser fundido com outro para manter o balanceamento e a propriedade de mínimo de chaves por nó, realizando também movimentações e reequilíbrios.

As Árvores B são usadas em situações que demandam alta eficiência em operações de busca e manipulação de dados em discos, devido ao alto custo de acessos ao disco:

1. **Sistemas de gerenciamento de banco de dados**: Para indexação e recuperação rápida de dados.
2. **Sistemas de cache e memória secundária**: A estrutura permite uma divisão eficiente de blocos de dados, otimizando o acesso.
3. **Sistemas de arquivos**: Para organizar arquivos e diretórios, permitindo leitura e escrita rápidas.

🤔 *Curiosidade: De onde vem o "B" de "B-árvore"?*

Resposta: As B-árvores foram inventadas e batizadas por Bayer e McCreight no ano de 1972, porém, eles não explicaram o motivo da escolha do nome.

- Árvores Red-Black: É uma árvore binária de busca em que cada nó é constituído dos seguintes campos: **COR** (1 bit): pode ser vermelho ou preto; **KEY** (e.g. inteiro): indica o valor de uma chave; **LEFT, RIGHT:** ponteiros que apontam para a subárvore esquerda e

direita, resp; **PAI**: ponteiro que aponta para o nó pai. O campo pai do nó raiz aponta para nil. Essa estrutura utiliza um mecanismo de **balanceamento dinâmico** para manter a eficiência em operações de busca, inserção e remoção.

O ponteiro pai facilita a operação da árvore rubro-negra, conforme imagem abaixo:

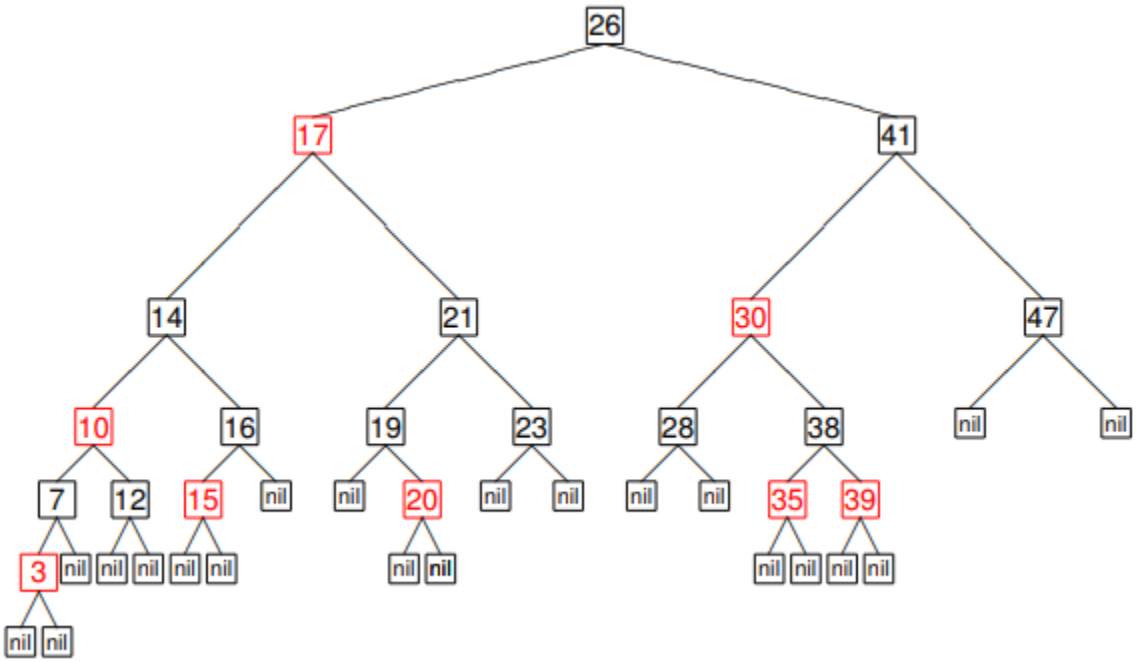


Imagem 6. Operação da árvore Red-Black

A estrutura apresenta propriedades importantes como:

1. **Cor do nó**: Cada nó é vermelho ou preto.
2. **Raiz e folhas pretas**: A raiz e todas as folhas (nós nulos) são pretas.
3. **Nós vermelhos**: Um nó vermelho não pode ter filhos vermelhos (não há duas arestas vermelhas consecutivas).
4. **Profundidade preta**: Todo caminho da raiz até uma folha ou nó nulo tem o mesmo número de nós pretos.
5. **Balanceamento dinâmico**: Essas regras mantêm a árvore aproximadamente balanceada, garantindo operações eficientes.

Para garantir que a estrutura de dados seja eficiente em tempo **de execução** e **aproximadamente balanceada**, as Árvores Red-Black possuem operações como:

1. **Busca**: A busca é realizada de forma semelhante a uma árvore binária de busca, mas mantendo a árvore balanceada conforme as regras de coloração. A complexidade de busca é O(logn)O(\log n)O(logn).
2. **Inserção**: Após inserir um nó, ele é inicialmente colorido de vermelho. Caso essa inserção viole as propriedades de coloração, são realizados ajustes, como **rotações** e **trocas de cor**, para restaurar o balanceamento.
3. **Remoção**: A remoção também pode violar as regras da árvore, exigindo rotações e ajustes de cores para restaurar as propriedades da Árvore Red-Black.

Além disso, elas são úteis em situações onde é necessário um balanceamento dinâmico com inserções e remoções frequentes:

1. **Compiladores**: Para a organização de tabelas de símbolos, permitindo que elementos sejam rapidamente inseridos e buscados.
2. **Sistemas de linguagem de programação**: As Árvores Red-Black são utilizadas em várias bibliotecas padrão para implementação de conjuntos (sets) e mapas (maps).
3. **Sistemas de armazenamento e cache em memória principal**: Especialmente em sistemas operacionais e gerenciadores de memória, que precisam manter dados balanceados e de fácil acesso.

Logo, as **Árvores B** são ideais para armazenamento em disco e grandes bases de dados, enquanto as **Árvores Red-Black** são preferidas para estruturas dinâmicas em memória, oferecendo eficiência e balanceamento dinâmico.

Como já citado, as estruturas de dados servem para organizar, manipular e acessar dados em diversos formatos de acordo com os objetivos do projeto. Elas estão presentes em contextos distintos no mundo do desenvolvimento de software e tecnologia em geral, cumprindo funções cruciais em soluções digitais. Existem estruturas de dados clássicas como **Vetores (Arrays)**, **Listas Ligadas**, **Pilhas e Filas**, **Árvores de Busca Binária (BSTs)** que ajudam a entender o comportamento em questões de eficiência, memória e aplicabilidade a diferentes tipos de problemas. Abaixo, há uma tabela comparativa mostrando as principais diferenças entre as estruturas, principalmente nas operações:

Estrutura	Acesso	Inserção	Remoção	Busca	Complexidade de Espaço	Aplicações
Vetor (Array)	O(1)	O(n)	O(n)	O(n)	O(n)	Armazenamento de dados fixos, acesso rápido por índice
Lista Ligada	O(n)	O(1) no início ou fim	O(1) no início ou fim	O(n)	O(n)	Manipulação dinâmica de dados, inserções e remoções frequentes
Pilha	O(n)	O(1)	O(1)	O(n)	O(n)	Implementação de recursão, navegação em histórico
Fila	O(n)	O(1)	O(1)	O(n)	O(n)	Processamento sequencial de tarefas, sistemas de filas de espera
Árvore de Busca Binária (BST)	O(log n) na média, O(n) no pior caso	O(log n) na média, O(n) no pior caso	O(log n) na média, O(n) no pior caso	O(log n) na média, O(n) no pior caso	O(n)	Estruturas hierárquicas, sistemas de busca, manipulação de dados estruturados

Árvore Red-Black	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(n)	Estruturas de dados em memória, implementação de dicionários e conjuntos
Tabela Hash	O(1)	O(1)	O(1)	O(1)	O(n)	Implementação de dicionários, busca rápida de dados sem ordem

Cada estrutura de dados é projetada para situações específicas: arrays são úteis para acesso rápido e direto; listas ligadas, para manipulação dinâmica; pilhas e filas, para acesso restrito sequencial; árvores (BST e Red-Black), para armazenamento de dados ordenados com balanceamento; e tabelas hash, para acesso direto e rápido. A escolha ideal depende das operações mais frequentes e do tipo de aplicação em que a estrutura será usada.

8. ANÁLISE DE ALGORITMOS

A notação assintótica é uma forma de descrever o comportamento de um algoritmo em termos de seu tempo de execução ou uso de memória, dependendo do tamanho da entrada, nnn. Ela é uma ferramenta fundamental na análise de algoritmos, pois permite estimar a eficiência de um algoritmo ao aumentar a quantidade de dados a serem processados. Além disso, ela é crucial para comparar algoritmos, pois oferece uma métrica abstrata que se foca em como um algoritmo escala. Na prática, ajuda desenvolvedores a escolher a estrutura de dados e o algoritmo mais adequado para suas necessidades, considerando a quantidade de dados e os requisitos de eficiência. Existem notações assintóticas que são usadas para descrever a eficiência de algoritmos, geralmente em relação ao tempo de execução ou uso de memória, conforme o tamanho da entrada *n:*

- Notação O (Big O): Representa o **limite superior** do crescimento da função, ou seja, o **pior caso** de execução do algoritmo.

- Notação Ω (Big Omega): Representa o limite inferior do crescimento da função, ou seja, o **melhor caso** de execução do algoritmo.
- Notação Θ (Big Theta): Indica que o tempo de execução do algoritmo está **exatamente entre os limites superior e inferior**. Isso significa que o comportamento do algoritmo, em termos de crescimento, é constante e previsível.

Exemplo Prático:

Considere um algoritmo de ordenação com complexidade $O(n^2)$:

- No **pior caso** (Big O), ele leva tempo quadrático para ordenar os elementos.
- No **melhor caso** (Big Omega), ele pode realizar a ordenação em menos tempo, dependendo dos dados, mas nunca será mais rápido que seu limite inferior.
- Se o tempo de execução for consistente em todos os casos como $n \log n$, então ele tem complexidade $\Theta (n \log n)$, sendo previsível.

Essas notações são essenciais na escolha de algoritmos, visto que ajudam a entender a eficiência e escalabilidade dos métodos, especialmente com o crescimento do tamanho dos dados de entrada. A análise de algoritmos de ordenação clássicos é fundamental por vários motivos, pois ajuda a entender as diferentes abordagens para organizar dados e os impactos dessas abordagens em termos de eficiência e recursos. Esses algoritmos (como BubbleSort, InsertionSort, MergeSort e QuickSort) são amplamente usados e servem como base para algoritmos mais complexos.

1. BubbleSort

- **Descrição**: Compara cada par de elementos adjacentes e os troca se estiverem na ordem errada. Esse processo se repete até que a lista esteja ordenada.
- **Complexidade de Tempo**:
  - **Pior Caso**: $O(n^2)$
  - **Caso Médio**: $O(n^2)$
  - **Melhor Caso**: O(n) (caso a lista já esteja ordenada)
- **Complexidade de Espaço**: O(1) (ordenamento in-place)

- **Características**: Simples de implementar, mas ineficiente para grandes conjuntos de dados. Também pode ser útil para listas pequenas ou quase ordenadas.
- **Aplicabilidade**: Utilizado principalmente para fins didáticos, dado que é fácil de entender e implementar, mas raramente usado em aplicações práticas.

2. InsertionSort

- **Descrição**: Insere elementos de uma lista desordenada em uma lista ordenada à medida que percorre a estrutura.
- **Complexidade de Tempo**:
    - **Pior Caso**: $O(n^2)$
    - **Caso Médio**: $O(n^2)$
    - **Melhor Caso**: O(n) (caso a lista já esteja ordenada)
- **Complexidade de Espaço**: O(1) (ordenamento in-place)
- **Características**: Simples e eficiente para listas pequenas, além de ser estável para listas quase ordenadas.
- **Aplicabilidade**: Usado em sistemas onde a lista é pequena ou quase ordenada.

3. MergeSort

- **Descrição**: Divide a lista em sub listas até que cada sublista tenha apenas um elemento, depois as combina ordenadamente para obter a lista final.
- **Complexidade de Tempo**:
    - **Pior Caso**: $O(n \log n)$
    - **Caso Médio**: $O(n \log n)$
    - **Melhor Caso**: $O(n \log n)$
- **Complexidade de Espaço**: $O(n)$ precisa de espaço adicional para as listas temporárias na combinação)
- **Características**: Sua ordenação é estável e previsível, além de ser eficiente para grandes listas, porém exige espaço extra.

- **Aplicabilidade**: Muito usado para ordenar grandes volumes de dados, especialmente quando a estabilidade e a eficiência consistente em O(n log n) são essenciais. É usado frequentemente em algoritmos híbridos e bases de dados.

4. QuickSort

- **Descrição**: Escolhe um elemento (o pivô) e organiza os elementos de forma que os menores que o pivô fiquem antes e os maiores fiquem depois, repetindo o processo nas sublistas.
- **Complexidade de Tempo**:
    - **Pior Caso**: O(n^2) (ocorre quando o pivô é sempre o menor ou maior elemento, mas é raro com estratégias de pivô eficientes)
    - **Caso Médio**: O(n log n)
    - **Melhor Caso**: O(n log n)
- **Complexidade de Espaço**: O(n log n)(em média, para chamadas recursivas, mas O(n) no pior caso)
- **Características**: Geralmente mais rápido que MergeSort em prática, por ser in-place, sua ordenação é instável, porém eficiente para grandes listas.
- **Aplicabilidade**: Um dos algoritmos de ordenação mais utilizados devido à sua eficiência em tempo médio e ao baixo uso de memória. É amplamente usado em implementações padrão de bibliotecas.

9. PROGRAMAÇÃO DINÂMICA

A programação dinâmica é ideal para problemas que exibem subproblemas sobrepostos e subestrutura ótima. Ela aborda um dado problema dividindo-o em subproblemas mínimos, solucionando os subproblemas, guardando os resultados parciais, combinando subproblemas menores e sub-resultados para obter e resolver problemas maiores, até recompor e resolver o problema original. Ela possui características importantes como:

1. Divisão em Subproblemas sobrepostos: Ao contrário da abordagem de divisão e conquista (como o MergeSort ou o QuickSort), onde os subproblemas são independentes, a programação dinâmica é aplicada quando os subproblemas se repetem. Em vez de resolver o mesmo subproblema várias vezes, a solução é calculada uma única vez e armazenada para referência futura, o que reduz o tempo de execução.
2. Subestrutura Ótima: Significa que uma solução ótima para o problema principal pode ser formada a partir das soluções ótimas de seus subproblemas. Essa característica permite construir a solução a partir de partes menores, garantindo que a solução geral seja a melhor possível.
3. Memorização (Caching): Na programação dinâmica, as soluções dos subproblemas são armazenadas em uma estrutura de dados como uma tabela (ou cache), evitando cálculos repetidos. Isso é conhecido como "memorização" e é especialmente útil em problemas de recursão, reduzindo significativamente a complexidade de tempo.
4. Redução de Complexidade Computacional: Em problemas onde a recursão simples leva a uma explosão exponencial de complexidade, a programação dinâmica pode reduzir a complexidade para uma ordem polinomial, tornando problemas intratáveis viáveis para a resolução.
5. Abordagem Bottom-Up ou Top-Down:

   5.1. A **abordagem Bottom-Up** (iterativa) resolve todos os subproblemas menores primeiro e, a partir deles, constroi a solução do problema maior, geralmente preenchendo uma tabela para armazenar as soluções intermediárias.

   5.2. A **abordagem Top-Down** resolve o problema de maneira recursiva com memorização, começando do problema principal e resolvendo apenas os subproblemas necessários.

Existem exemplos já citados anteriormente que utilizam a programação dinâmica, como o Problema da Mochila (Knapsack), onde é preciso selecionar itens com pesos e valores específicos para maximizar o valor total, sem ultrapassar a capacidade da mochila e o Problema de Caminho mínimo, onde deve-se encontrar o caminho mais curto ou o custo mínimo para atravessar uma matriz ou grafo. Explicando detalhadamente sobre o primeiro exemplo citado, é

um problema de otimização onde é preciso selecionar itens com valores e pesos específicos para maximizar o valor total sem ultrapassar a capacidade máxima de peso que a mochila suporta:

Formulação do Problema

- Dado:
    - **n itens**, onde cada item *i* tem um **peso *w*** e um **valor *vi***
    - Uma **capacidade máxima da mochila *W***.
- Objetivo:
    - Selecionar uma combinação de itens que maximize o valor total sem que a soma dos pesos ultrapasse *W*.

Variantes do Problema da Mochila

1. **Mochila Binária (0-1 Knapsack)**:
    - Cada item pode ser escolhido **apenas uma vez**.
    - O objetivo é encontrar uma seleção de itens que maximize o valor total sem exceder a capacidade.
    - **Exemplo**: Se temos três itens com pesos 10,20,30 e valores 60,100,120 e uma capacidade de 50, o desafio é escolher itens que maximizem o valor total.
2. **Mochila Fracionária (Fractional Knapsack)**:
    - Diferente da versão 0-1, os itens podem ser **divididos em frações**.
    - O problema pode ser resolvido usando uma estratégia gulosa (greedy), ordenando os itens pela relação valor/peso e selecionando os itens de maior valor/peso até preencher a capacidade da mochila.

Solução com Programação Dinâmica (para a Mochila Binária)

A abordagem de programação dinâmica é ideal para a versão 0-1, onde é necessário explorar todas as combinações de itens.

1. **Definir a Recorrência**:
    - Seja **D P |i| |j|** o valor máximo obtido considerando os primeiros ***i*** itens com capacidade ***j*** da mochila.

- Para cada item iii:
  - **Caso 1**: Não incluir o item *i*– então o valor máximo é o mesmo que para *i* - 1 itens e capacidade *j*, ou seja, **D P [i - 1] [j]**
  - **Caso 2**: Incluir o item *i*– então o valor máximo é o valor do item *i* mais a melhor combinação que se ajusta na capacidade remanescente (*j*−*wi*), ou seja, vi + **D P [i - 1] [j - *wi*]**
  - Assim, a relação de recorrência fica: DP[i][j]=max(DP[*i* - 1][*j*], *vi* + DP [*i* - 1][j-wi]

2. **Construir a Tabela de DP**:
   - Preencher uma tabela de tamanho n×W, onde *n* é o número de itens e W é a capacidade da mochila.
   - O valor final DP[n][W] será o valor máximo possível para os itens e a capacidade total.

Complexidade

- **Tempo**:O(n×W)
- **Espaço**: O(n×W), podendo ser otimizado para O(W) em uma abordagem de espaço comprimido.

Em suma, a programação dinâmica é especialmente útil em algoritmos de otimização e é amplamente aplicada em inteligência artificial, bioinformática, processamento de linguagem natural e outras áreas que envolvem cálculos intensivos.

10. ALGORITMOS GULOSOS

Algoritmos gulosos (ou algoritmos greedy) são usados em problemas de otimização onde é interessante realizar um conjunto de melhores soluções locais, possuindo como objetivo obter uma solução ótima global, tomando como base um determinado parâmetro. Os parâmetros que definem o que será a melhor solução durante a interação irão variar para cada situação. Eles possuem características importantes como:

1. Irrevogabilidade: As escolhas feitas pelo algoritmo são permanentes e não são revisadas ou desfeitas. Isso significa que o algoritmo não explora todas as possibilidades, mas apenas uma sequência de decisões em direção ao objetivo final.
2. Escolhas Locais Ótimas: Em cada passo, o algoritmo faz a escolha que parece ser a melhor, com base em alguma medida de otimização (como custo mínimo e valor máximo, por exemplo). A decisão é feita com informações disponíveis no momento e sem olhar para as etapas futuras
3. Eficiência: Eles tendem a ser mais rápidos e simples que outros métodos de otimização, como a programação dinâmica ou a busca exaustiva, já que não envolvem múltiplas iterações sobre subproblemas ou reavaliação de escolhas.
4. Equívoco na garantia de solução ótima: Em muitos casos, a escolha local ótima não leva a uma solução global ótima. Portanto, os algoritmos gulosos funcionam bem apenas para certos tipos de problemas onde a estratégia gulosa é garantida para alcançar a solução ótima.

O uso de algoritmos gulosos são úteis em problemas que possuem a propriedade gulosa (ou propriedade de subestrutura ótima), onde a escolha ótima local também é a melhor escolha global. Isso permite que a estratégia gulosa produza uma solução global ótima. Problemas que atendem a essa característica incluem problemas onde a estrutura não permite que uma escolha local comprometa o resultado global. Existem exemplos práticos que utilizam algoritmos gulosos para execução, como:

1. Problema da Mochila Fracionária (Fractional Knapsack): Neste problema, o objetivo é maximizar o valor dos itens em uma mochila com capacidade limitada, podendo fracionar itens. Como solução gulosa, pode-se ordenar os itens pela razão valor/peso e adicionar os itens de maior razão até preencher a mochila.
2. Problema da Árvore Geradora Mínima (Minimum Spanning Tree): Pode ser resolvido otimamente utilizando-se procedimentos gulosos no sentido de sempre escolher, sucessivamente, as arestas de menor custo. Algoritmos gulosos como o Algoritmo de Prim e o Algoritmo de Kruskal são usados para resolver esse problema e, devido à propriedade gulosa, esses algoritmos encontram a solução ótima.

11. ESTADO DA ARTE

Uma pesquisa como essa permite focar na criação de algoritmos com menor complexidade de tempo e espaço. Isso é fundamental em áreas como aprendizado de máquina, processamento de big data e inteligência artificial, onde a eficiência é crucial. Com o crescimento da IA, algoritmos específicos como redes neurais e aprendizado por reforço têm sido continuamente otimizados e adaptados para problemas mais complexos e aplicações práticas. Destacando a segurança e criptografia, há algoritmos robustos para esse quesito, especialmente com a pesquisa em criptografia quântica, que explora novos paradigmas de segurança em função do avanço dos computadores quânticos. Já para problemas NP-difíceis, algoritmos de aproximação, heurísticas e meta-heurísticas são utilizados para obter soluções eficientes mesmo que não sejam exatamente ótimas.

Levando em conta alguns desafios do mundo atual, o desenvolvimento de algoritmos sustentáveis e energeticamente eficientes é importante visto que a sustentabilidade computacional deve ser priorizada. Além disso, a escalabilidade é um outro desafio importante visto que, com a quantidade de dados aumentando exponencialmente, algoritmos que lidam bem com grandes volumes de dados continuam a ser desenvolvidos.

Para obter sucesso ao elaborar um projeto de software, existem etapas importantes que devem ser seguidas e que podem variar, dependendo da metodologia utilizada (Scrum, Kanban). Abaixo, cada fase será descrita:

Fase 1: Planejamento e Análise de Requisitos

1. **Definição do Escopo do Projeto**: Identificar o problema a ser resolvido e definir os limites e as metas do projeto.
2. **Levantamento e Análise de Requisitos**: Entrevistar stakeholders e documentar as necessidades. Esse documento servirá como base para todas as decisões subsequentes.
3. **Análise de Viabilidade e Riscos**: Avaliar a viabilidade técnica e econômica e identificar riscos potenciais.

4. **Pesquisa e Seleção de Tecnologias**: Determinar as ferramentas, linguagens de programação e frameworks a serem usados.

Fase 2: Arquitetura e Design

1. **Definição da Arquitetura do Software**: Escolher a arquitetura mais adequada (por exemplo, arquitetura de microsserviços, arquitetura monolítica).
2. **Modelagem de Dados**: Estruturar o banco de dados e as relações entre as entidades.
3. **Design de Interface e UX**: Criar esboços de interfaces e design UX para garantir que a experiência do usuário seja eficaz.
4. **Definição de Algoritmos e Estruturas de Dados**: Selecionar os algoritmos e estruturas de dados mais eficientes para a aplicação, levando em consideração os requisitos de desempenho.

Fase 3: Implementação e Desenvolvimento

1. **Divisão em Sprints ou Módulos**: Se a metodologia é ágil, dividir o projeto em sprints. Caso contrário, dividir em módulos lógicos para implementação.
2. **Desenvolvimento de Código**: Escrever e organizar o código para cada módulo, com testes unitários e integração contínua.
3. **Revisão de Código e Documentação**: Implementar revisões para manter a qualidade e a padronização, além de documentar o processo para futura referência.

Fase 4: Teste e Validação

1. **Testes Unitários e de Integração**: Garantir que cada unidade e módulo funcione conforme esperado.
2. **Testes Funcionais e de Aceitação**: Realizar testes para garantir que os requisitos do usuário estejam atendidos.
3. **Testes de Performance e Carga**: Verificar se o software atende aos requisitos de desempenho e se mantém estável sob cargas elevadas.

Fase 5: Implantação e Manutenção

1. **Preparação do Ambiente de Produção**: Configurar servidores, bancos de dados e qualquer infraestrutura necessária.
2. **Deploy do Software**: Transferir o software para o ambiente de produção.
3. **Treinamento e Documentação do Usuário**: Fornecer treinamento para os usuários e documentação para facilitar o uso.
4. **Monitoramento e Manutenção Contínua**: Monitorar a aplicação e realizar correções de bugs ou melhorias com base no feedback do usuário.

Por fim, é importante salientar o impacto dos algoritmos no mundo globalizado, uma vez que o desenvolvimento de um projeto de software bem estruturado é fundamental para alinhar a pesquisa acadêmica com as aplicações práticas. Esse processo estruturado melhora a qualidade, a eficiência e a escalabilidade dos softwares, possibilitando aplicações mais rápidas e acessíveis em diversas áreas da computação.

12. CONCLUSÃO

Por conseguinte, a análise e projeto de algoritmos permite prever, otimizar e focar na escolha de métodos e estruturas de dados ideais, permitindo que problemas complexos sejam resolvidos com precisão e eficiência, garantindo que recursos como tempo e memória sejam utilizados da melhor forma possível. A relevância desses estudos vai além da teoria: eles são aplicados em diversas áreas, como inteligência artificial, segurança de dados, e análise de grandes volumes de informação, e possibilitam a criação de softwares e sistemas que atendem às exigências do mundo atual. Para o futuro, a pesquisa em algoritmos deve avançar em áreas como computação quântica e principalmente algoritmos de segurança, refletindo a necessidade crescente de soluções tecnológicas sustentáveis, seguras e que se adaptem a contextos dinâmicos e desafiadores. Assim, o estudo contínuo de análise e projeto de algoritmos é essencial para o progresso da tecnologia e para o enfrentamento de novos desafios computacionais.

13. REFERÊNCIAS

ELER, Prof. Danilo Medeiros. Projeto e Análise de Algoritmos Introdução. São Paulo: Unesp, 2021. Color. Disponível em: https://daniloeler.github.io/teaching/PAA2020/files/PAA-aula01-Introducao.pdf. Acesso em: 26 out. 2024.

https://github.com/CloudEducationBrazil/WydenAlgoritmosEComplexidade/

LOUREIRO, Antonio Alfredo Ferreira. **Projeto e Análise de Algoritmos Análise de Complexidade**. Minas Gerais, 2020. Color. Disponível em: https://homepages.dcc.ufmg.br/~loureiro/alg/111/paa_01AnaliseDeComplexidade.pdf. Acesso em: 27 out. 2024.

FERRAZ, Filipe Chagas. **Grafos e algoritmos**. 2020. Disponível em: https://medium.com/programadores-ajudando-programadores/os-grafos-e-os-algoritmos-697c1fd4a416. Acesso em: 20 out. 2024.

ACADEMY, Khan. **Algoritmos de divisão e conquista**. Disponível em: https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms#:~:text=dividir%20e%20conquistar&text=Esse%20paradigma%2C%20dividir%2De%2D,para%20resolver%20o%20problema%20original.. Acesso em: 26 out. 2024.

ANDRADE, Ana Paula de. **O que é e como funciona a Estrutura de Dados Pilha**. 2021. Disponível em: https://www.treinaweb.com.br/blog/o-que-e-e-como-funciona-a-estrutura-de-dados-pilha. Acesso em: 28 out. 2024.

COZMAN, Fabio Gagliardi; MARTINS, Abio Gagliardi Cozman Thiago. **Estruturas de Dados — Pilhas, Filas, Listas**. São Paulo, 2020. 38 slides, color. Disponível em:

https://edisciplinas.usp.br/pluginfile.php/3499152/mod_resource/content/1/estruturas.pdf. Acesso em: 22 out. 2024.

OLIVEIRA, Valeriano A. de; RANGEL, Socorro; ARAUJO, Silvio A. de. **Teoria dos Grafos**: departamento de matemática aplicada. São Paulo, 2012. 34 slides, color. Disponível em: https://www.ibilce.unesp.br/Home/Departamentos/MatematicaAplicada/docentes/socorro/representacaografos.pdf. Acesso em: 26 out. 2024.

HERMUCHE, Anwar. **Métodos de Busca em Grafos — BFS & DFS**. 2024. Disponível em: https://medium.com/@anwarhermuche/m%C3%A9todos-de-busca-em-grafos-bfs-dfs-cf17761a0dd9#:~:text=Quando%20falamos%20em%20realizar%20uma,DFS%20ou%20Busca%20em%20Profundidade).. Acesso em: 18 out. 2024.

UNIVASF. **Grafos – Busca em largura**. 2022. Disponível em: http://www.univasf.edu.br/~marcelo.linder/arquivos_aed2/aulas/aula24.pdf. Acesso em: 21 out. 2024.

BARBOSA, Marco A L. **Busca em profundidade**. 2022. Disponível em: https://malbarbo.pro.br/arquivos/2022/6898/04-busca-em-profundidade.pdf. Acesso em: 23 out. 2024.

DESCONHECIDO. **Análise de Algoritmos**: np-completude. NP-Completude. 2021. Disponível em: https://www.comp.uems.br/~chastel/analise/np-completude.pdf. Acesso em: 23 out. 2024.

MIYAZAWA, Prof. Flávio Keidi. **Algoritmos de Aproximação**. Disponível em: https://www.ic.unicamp.br/~fkm/problems/algaprox.html. Acesso em: 24 out. 2024.

FEOFILOFF, Paulo. **Árvores B (B-trees)**. 2019. Disponível em: https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/B-trees.html. Acesso em: 19 out. 2024.

SONG, Siang Wun. **Árvore Rubro-Negra**. São Paulo, 2008. Color. Disponível em: Árvore Rubro-Negra. Acesso em: 18 out. 2024.

PONTI, Prof. Moacir A.. **Notação Assintótica e Complexidade**: scc201/501 - introdução à ciência de computação ii. São Paulo, 2020. 39 slides, color. Disponível em: https://edisciplinas.usp.br/pluginfile.php/5767771/mod_resource/content/1/scc0201_M2_Slides_NotacaoAssintotica.pdf. Acesso em: 19 out. 2024.

WILHELM, Professor Volmir Eugênio. **Problema da Árvore Geradora Mínima**. Paraná, 2023. 6 slides, color. Disponível em: https://docs.ufpr.br/~volmir/PO_II_11_arvore_geradora_minima.pdf. Acesso em: 23 out. 2024.