UNIDADE II

Análise de Algoritmos

Prof. Me. Roberto Leminski

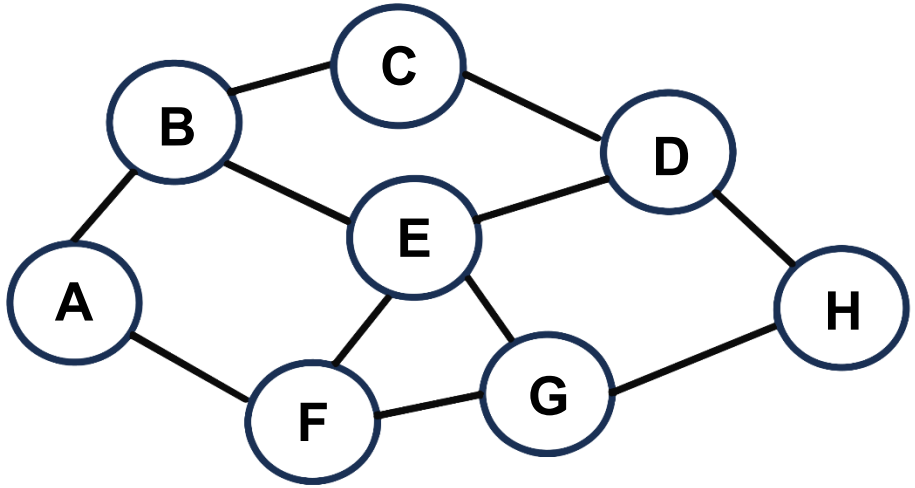
Algoritmos sobre grafos

- Um grafo pode ser considerado como um conjunto finito de elementos distintos e separados, que se conectam entre si.
- Essas conexões não serão necessariamente de todos os elementos com todos os demais elementos, mas é necessário que ao menos algumas conexões estejam presentes.

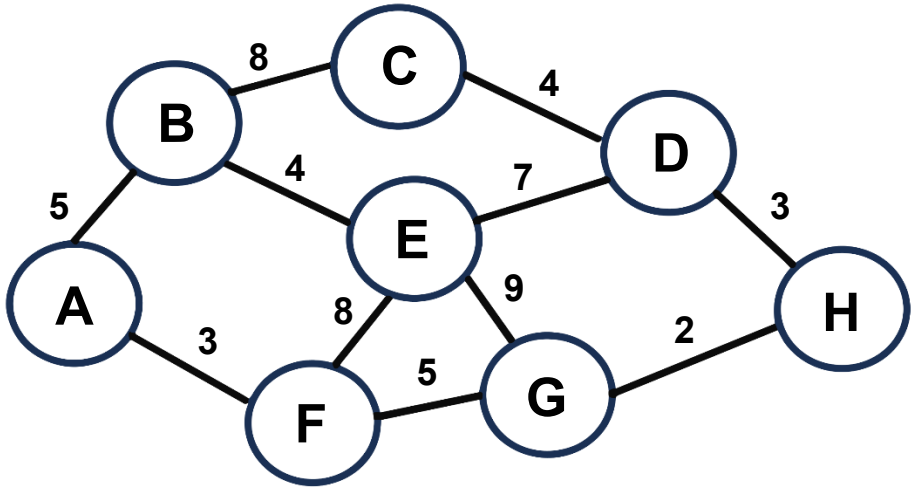
Um grafo G será composto por:

- Um conjunto V de vértices (também chamados de pontos ou nós), que representam os elementos que estabelecem conexão entre si.
    - Um conjunto E de pares não ordenados de vértices distintos chamados de arestas. Cada aresta representa uma conexão individual entre dois elementos de V.

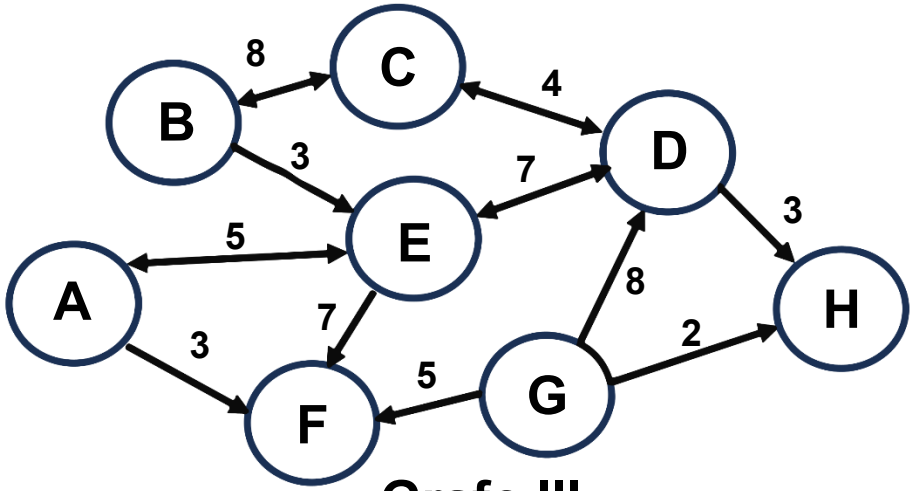
Algoritmos sobre grafos



Grafo I



Grafo II



Grafo III

Fonte: livro-texto

Algoritmos sobre grafos

Grafo I =
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Grafo II =
$$\begin{bmatrix} 0 & 5 & 0 & 0 & 0 & 3 & 0 & 0 \\ 5 & 0 & 8 & 0 & 4 & 0 & 0 & 0 \\ 0 & 8 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 7 & 0 & 0 & 3 \\ 0 & 4 & 0 & 7 & 0 & 8 & 9 & 0 \\ 3 & 0 & 0 & 0 & 8 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 9 & 5 & 0 & 2 \\ 0 & 0 & 0 & 3 & 0 & 0 & 2 & 0 \end{bmatrix}$$

Grafo III =
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 5 & 3 & 0 & 0 \\ 0 & 0 & 8 & 0 & 3 & 0 & 0 & 0 \\ 0 & 8 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 7 & 0 & 0 & 3 \\ 5 & 0 & 0 & 7 & 0 & 7 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 5 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fonte: livro-texto

Algoritmos sobre grafos

- Uma busca por um elemento em um grafo consiste em converter o grafo em uma árvore e aplicar uma busca em largura ou em profundidade (não heurísticas) ou algum outro algoritmo de busca heurístico.
- Um grafo é euleriano se é possível construir um trajeto por ele que passe por todas as arestas do grafo, sem repetir arestas, começando e terminando o trajeto na mesma aresta.
- A solução para esse problema foi definida pelo matemático Leonhard Euler (1707 – 1783), tendo em vista o problema das sete pontes de Königsberg (cidade da Prússia que possuía sete pontes).
    - A solução é bastante simples: o grafo pode ter todas suas arestas percorridas se possuir nenhum ou dois vértices de grau ímpar (o grau de um vértice é o número de arestas que se conectam a ele).
    - No caso de o grafo possuir dois vértices com grau ímpar, o trajeto começará em um desses vértices e terminará no outro.

Algoritmos sobre grafos

- Um problema mais complicado é verificar se um grafo é hamiltoniano.
- Um grafo é hamiltoniano se existe um trajeto que permite passar por todos os vértices do grafo, sem repetir nenhum.
- Se esse trajeto existir e se iniciar e terminar no mesmo vértice, diz-se que existe um ciclo hamiltoniano no grafo. Esse nome deriva do matemático irlandês William Hamilton (1803–1865).
- A definição sobre se o grafo é ou não hamiltoniano é um problema NP-Completo e será tratado mais adiante.

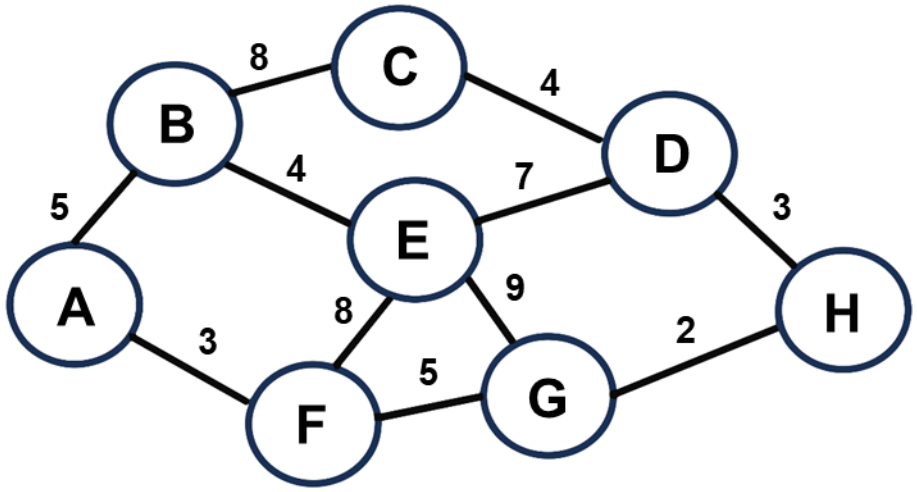
Algoritmos sobre grafos

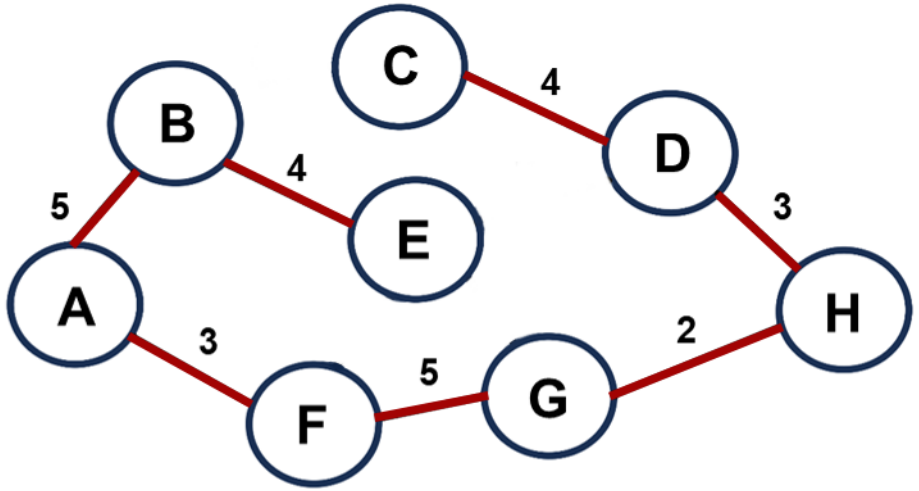
- O Algoritmo de Kruskal é um algoritmo que reduz um grafo conexo ponderado a uma árvore geradora mínima.
- Uma árvore geradora mínima é uma árvore que contêm todos os vértices e apenas as arestas mínimas necessárias, de menor peso. O nome do algoritmo vem do seu criador, o matemático americano Joseph Bernard Kruskal Jr. (1928 – 2010).

Esse algoritmo pode ser definido em duas etapas:

1. Selecionar a aresta de menor peso presente no grafo. Se dois vértices conectados por ela estiverem numa mesma árvore (gerada pelas arestas que permaneceram das iterações anteriores), remova-a. Caso contrário, a aresta permanecerá.
2. Repetir o processo em ordem crescente das arestas até não haver mais arestas disponíveis.

Algoritmos sobre grafos





Fonte: livro-texto

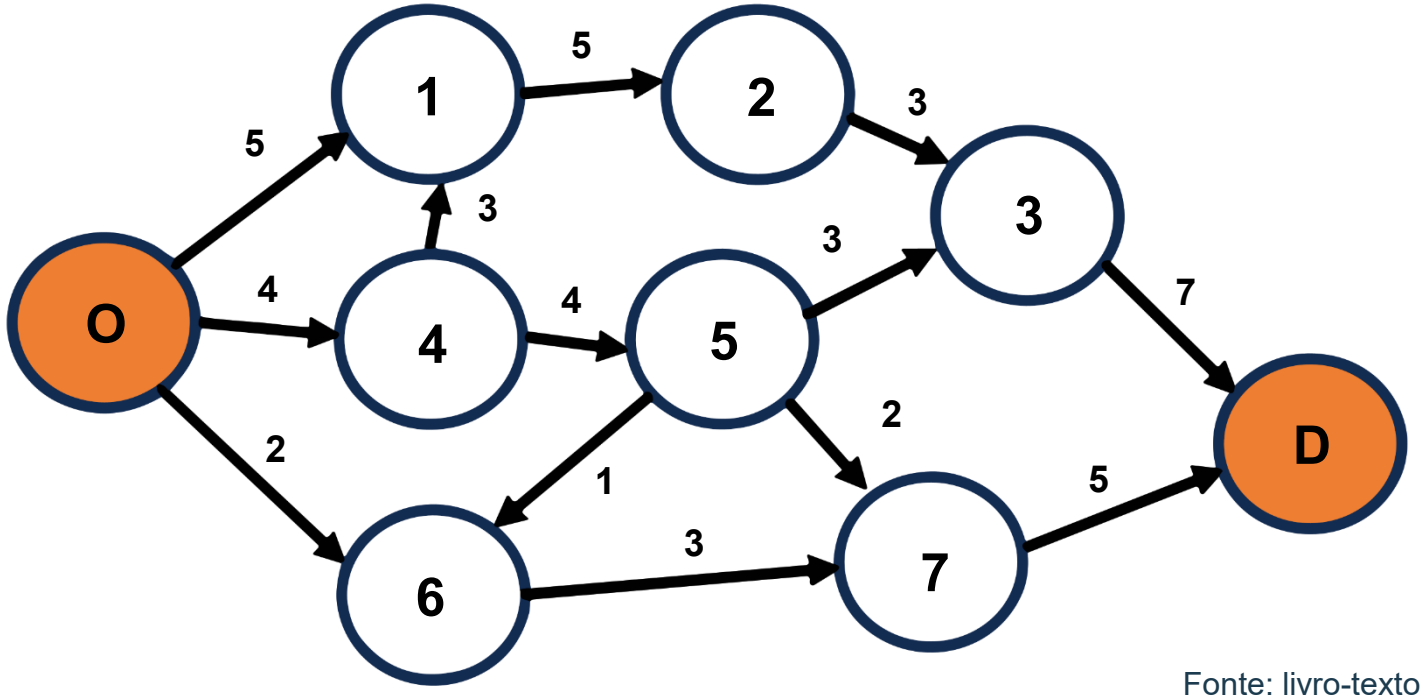
Fluxo em redes

- Podemos definir uma rede como sendo um grafo no qual existem deslocamentos de entidades entre os vértices utilizando exclusivamente as arestas que as conectam como caminho.
- Considere uma rede representada por um grafo ponderado, no qual o peso de cada aresta E é um valor positivo c, que significa a capacidade da aresta. Essa capacidade é o fluxo máximo dos elementos através deste caminho.
- Um dos vértices do grafo é denominado origem (O), que é de onde o fluxo através da rede parte. Um segundo nó é denominado destino (D), que é para onde os fluxos se destinam. O objetivo, nesse problema, é calcular qual o maior fluxo possível entre os vértices de origem e destino da rede.

Fluxo em redes

Dentre as muitas aplicações práticas envolvendo esse tipo de algoritmo, podemos citar:

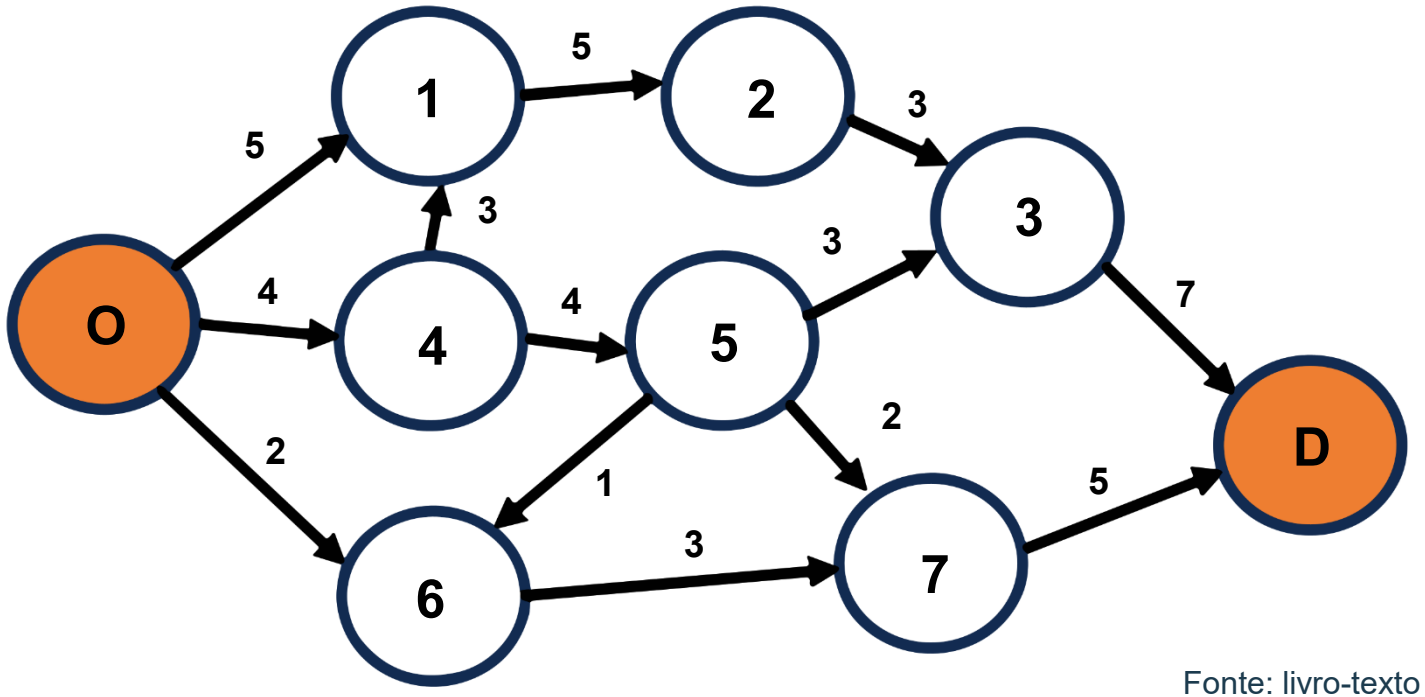
- controle de escoamento de fluidos em redes de distribuição ou em aplicações industriais;
- fluxo de trânsito de veículos em ruas de uma região;
- comunicação entre componentes de uma rede de computadores;
- comunicação dentro de um sistema computacional distribuído.



Fonte: livro-texto

Fluxo em redes

- Um problema que surge é a saturação de arestas e vértices: cada aresta comporta um fluxo máximo, que é indicado pelo seu peso (lembrando que é um grafo ponderado).
- Em cada nó do grafo, com exceção de O e D, a somatória dos fluxos que chegam nele deve ser igual à somatória dos fluxos que saem.



Fonte: livro-texto

Fluxo em redes

- Para calcular o fluxo máximo em redes, utilizamos o algoritmo de Ford-Fulkerson.
- Esse algoritmo recebe o nome de seus criadores, os matemáticos americanos Lester Randolph Ford Jr. e Delbert Ray Fulkerson, que o publicaram em 1956.
- Um conceito importante para esse algoritmo é a capacidade residual de uma aresta: essa é a diferença entre a capacidade da aresta e o fluxo que está passando por ela em dado momento.
- Se uma dada aresta possuir uma capacidade residual igual a zero em dado momento, ela é dita saturada.

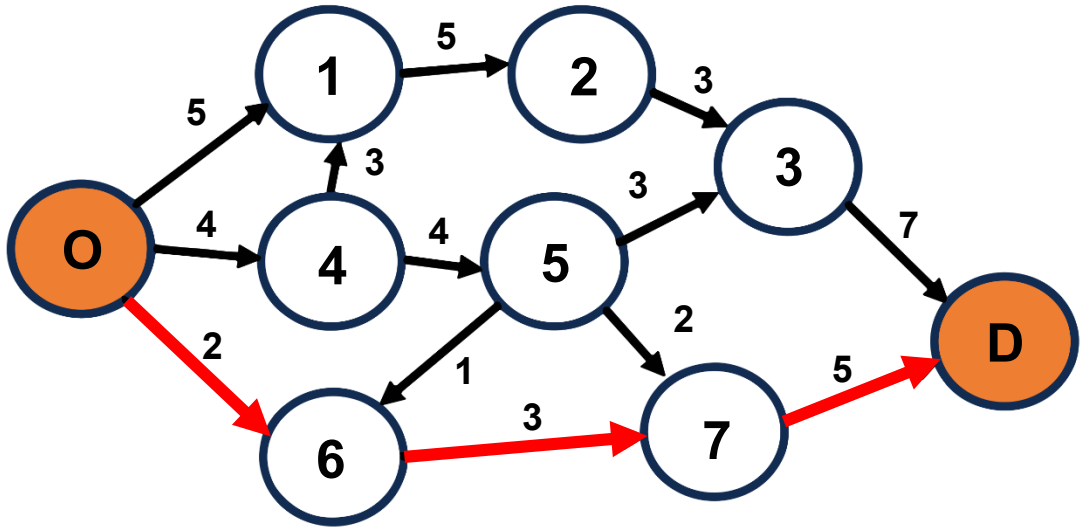
Fluxo em redes

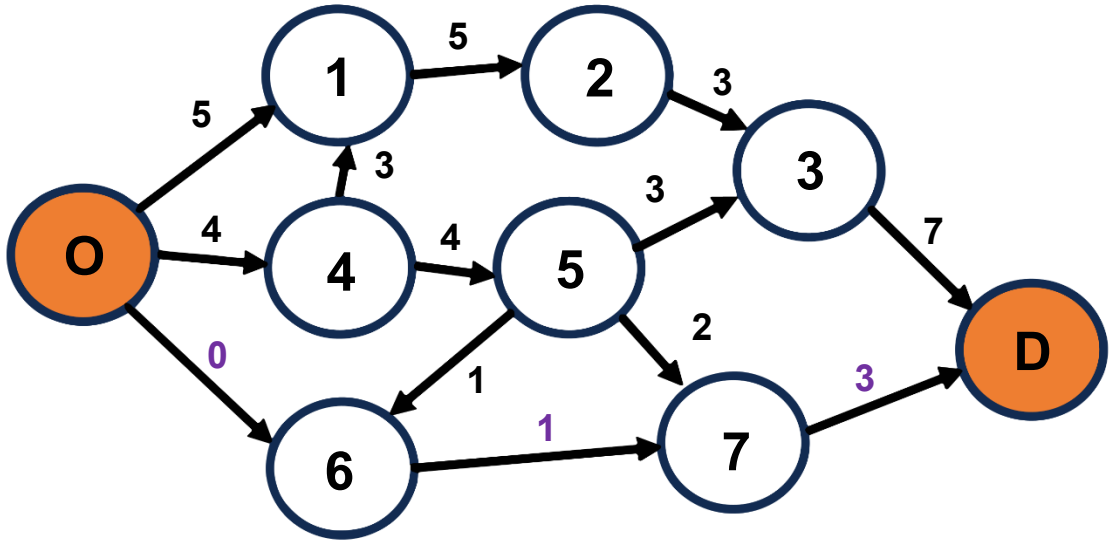
O algoritmo de Ford-Fulkerson é um algoritmo iterativo que executa a cada iteração as seguintes etapas:

1. Identificar um caminho qualquer que sai da origem e chegue até o destino, preferencialmente o que possuir as arestas com menor capacidade. Esse caminho não pode conter nenhuma aresta saturada.
2. Identificar a aresta de menor capacidade residual nesse caminho e subtrair esse valor de todas as arestas no trajeto selecionado. Observamos que a aresta de menor capacidade ficará saturada e não poderá ser utilizada em mais nenhum trajeto.
3. O valor atribuído a essa iteração será a capacidade original da aresta de menor capacidade (o valor que foi subtraído).
4. Repetir o processo com outro trajeto dentro da rede, enquanto houver caminhos possíveis entre a origem e o destino na rede.

Fluxo em redes

- Quando não for possível traçar mais nenhum caminho entre a origem e o destino, somamos os valores de cada iteração realizada. Essa somatória será o fluxo máximo da rede.





  - Valor da iteração 1: 2
  - O número de iterações dependerá do rede e dos caminhos escolhidos em cada uma delas.

Fluxo em redes

- A complexidade de tempo do algoritmo de Ford-Fulkerson é proporcional ao número de iterações.
- Sendo M a maior capacidade presente na rede e V o número de vértices presentes na rede (excluindo-se a Origem e o Destino), podemos facilmente calcular que o número máximo de iterações será $V \cdot M$.
- Esse resultado sugere que teremos um comportamento polinomial como complexidade de tempo, mas podemos perceber que se, por exemplo, simplesmente dobrarmos a capacidade M da maior aresta, dobraremos o número de iterações.
  - Assim, uma pequena mudança na rede pode alterar drasticamente a execução do algoritmo, sem alterar significativamente seu resultado.
  - A complexidade desse algoritmo pode ser calculada por meio da Análise Amortizada, da qual falaremos mais adiante.

Interatividade

Uma rede possui 10 vértices, incluindo a origem e o destino, conectados entre si por um total de 25 arestas. Qual o número máximo de iterações que podem ser necessárias para obtermos o fluxo máximo nessa rede pelo algoritmo de Ford-Fulkerson?

a) 100.
b) 125.
c) 150.
d) 200.
e) 250.

Resposta

Uma rede possui 10 vértices, incluindo a origem e o destino, conectados entre si por um total de 25 arestas. Qual o número máximo de iterações que podem ser necessárias para obtermos o fluxo máximo nessa rede pelo algoritmo de Ford-Fulkerson?

a) 100.
b) 125.
c) 150.
d) 200.
e) 250.

Algoritmos gulosos

- Um tipo de problema muito comum em algoritmos são os problemas de otimização.
- Nesses problemas, buscamos obter o resultado máximo ou mínimo de algo em dada situação.
- O problema das multiplicações de matrizes (MCOP) é um exemplo, assim como o algoritmo de Ford-Fulkerson.
- Os chamados algoritmos gulosos realizam tomadas de decisões baseados apenas na informação disponível no momento, sem considerar os efeitos futuros de tais decisões no progresso da execução do algoritmo.
    - Dessa forma, não há necessidade de se considerar as ramificações decorrentes dessa decisão, nem a necessidade de registros das operações anteriores.

Algoritmos gulosos

- Devido às características apresentadas, algoritmos gulosos (que é a tradução mais empregada de greedy algorithms; uma tradução melhor seria “algoritmos gananciosos”) são fáceis de serem criados e implementados.
- Muitos métodos de busca utilizados em Inteligência Artificial são algoritmos gulosos.
- Algoritmos gulosos também são uma abordagem para resolver problemas cujo tempo de execução é muito elevado ou para o qual não se conhece um algoritmo ótimo para sua resolução.
- Algoritmos gulosos possuem semelhanças com a Programação Dinâmica: ambas as técnicas trabalham com uma sequência de passos, muitas vezes recursiva.
    - A diferença principal ocorre em como os passos a serem tomados são definidos: na Programação Dinâmica, a escolha das próximas etapas depende das soluções dos subproblemas anteriores, enquanto nos algoritmos gulosos a escolha da próxima etapa é feita apenas com base no que parecer ser mais promissor naquele momento.

Algoritmos gulosos

Alguns exemplos de algoritmos gulosos:

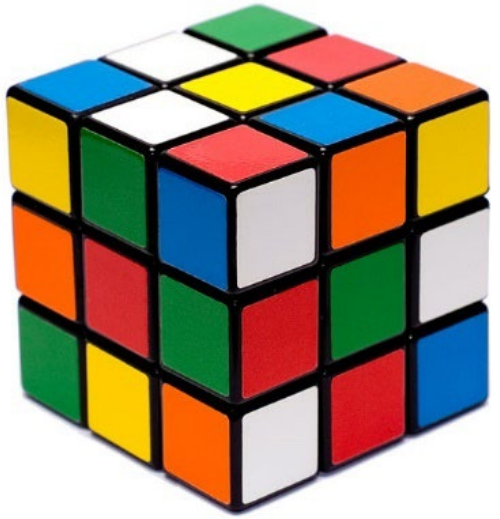
- Caminho de peso mínimo: envolve identificar o menor trajeto em um grafo que possua arestas com pesos.
- Algoritmo de Kruskal.
- Intervalos disjuntos: matematicamente falando, dois intervalos são ditos disjuntos quando não há sobreposição entre eles. Em situações práticas, um local só pode estar disponível para um determinado evento, ou uma pessoa só pode estar em um dado local em um dado momento. Esse tipo de problema ocorre na alocação de espaços de eventos e feiras, reservas de hotéis, alocação de professores para aulas etc.

Algoritmos gulosos

- A principal característica de um algoritmo guloso é que ele encontra uma solução otimizada para um problema, mas não necessariamente sempre a melhor solução possível (ótimo global).
- Assim, se houver mais de uma solução possível, pode ser que a primeira solução encontrada não seja necessariamente a solução mais otimizada.
- Algoritmos gulosos são amplamente empregados em métodos de busca no campo da Inteligência Artificial.

Algoritmos gulosos

- Um método de busca é uma metodologia que faz utilização de um espaço de estados.
- Um espaço de estados é um conjunto finito de estados em que um sistema pode estar em dado momento.
- Em um método de busca, parte-se de um estado inicial do sistema e tem-se como objetivo chegar em um estado final, da forma mais otimizada possível.
- No método de busca gulosa pela melhor escolha, faz-se necessário o conhecimento de quão perto um determinado estado está de um estado final desejado.
- A partir desse conhecimento, decide-se, dentre os estados que podem ser atingidos a partir do estado atual, qual está mais próximo do estado final desejado.



Fonte: https://commons.wikimedia.org/wiki/File:Rubiks_cube_by_keqs.jpg

Código de Huffman

- Para o algoritmo de codificação de Huffman é necessário saber quais os símbolos presentes no texto e a frequência com a qual eles ocorrem.

A estrutura desse algoritmo guloso se baseia na construção de uma árvore binária completa:

1. Inicialmente, cada símbolo é representado por uma árvore com apenas um nó (apenas a raiz), com sua respectiva frequência associada a eles.
2. As duas menores frequências são agrupadas em uma árvore binária, cuja raiz é a somatória das frequências. Essa árvore substitui os símbolos que a compõem.
3. Repete-se a etapa 2 até que todos os símbolos estejam agrupados em uma única árvore binária, formando a chamada árvore de Huffman.
4. Por fim, associa-se um valor binário (0 ou 1) a cada ramo (usualmente 0 à esquerda e 1 à direita).
5. O código de cada folha da árvore será a sequência dos caracteres das arestas da raiz até ela.

Código de Huffman

- Esse algoritmo permite a codificação binária mais eficiente para uma mensagem ou um texto.
- Esse código se baseia em uma árvore binária construída a partir das frequências relativas dos símbolos na mensagem que se desejava codificar.
- O código de Huffman é uma das melhores técnicas de compressão de dados.
- Esse algoritmo se utiliza de códigos de tamanho variável para representar os símbolos do texto que será codificado (que podem ser caracteres em sequência, ou cadeias de caracteres, ou realmente um texto).
    - A ideia básica do algoritmo é muito simples: atribui-se códigos com quantidades de bits menores para os símbolos mais frequentes no texto e códigos mais longos para os mais incomuns.
    - Além disso, o algoritmo de Huffman gera um código livre de prefixação.

Código de Huffman

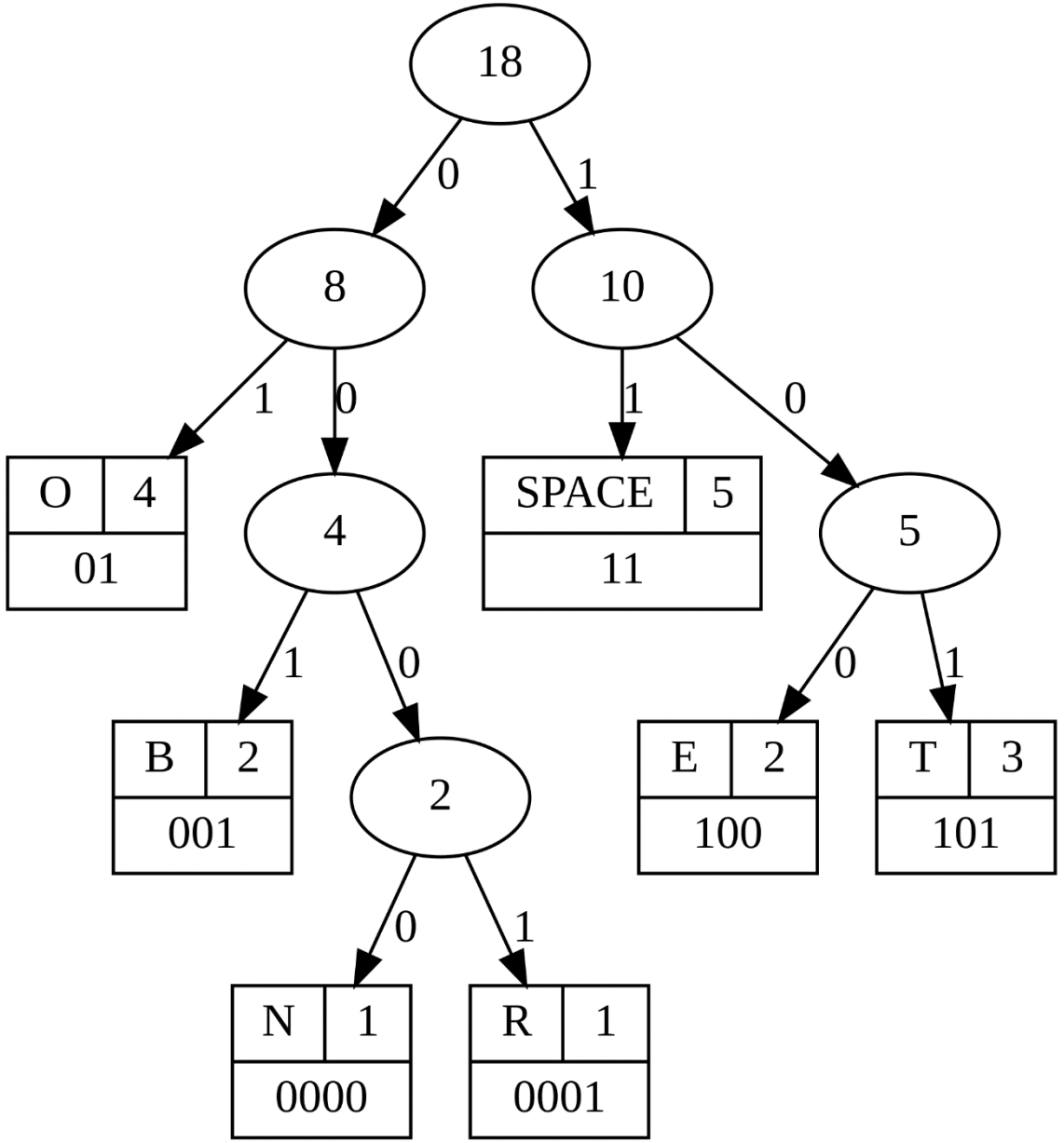
- Nesse algoritmo, os símbolos mais frequentes estarão mais perto da raiz da árvore e consequentemente terão códigos com menor quantidade de bits.
- De forma semelhante, os símbolos de menor frequência terão códigos com maior quantidade de bits.

Exemplo: considere a famosa frase "*TO BE OR NOT BE*". As ocorrências de cada símbolo são:

Símbolo	Ocorrências
O	5
Espaço	5
T	3
B	2
E	2
N	1
R	1

Código de Huffman

Símbolo	Ocorrências
Espaço	5
O	4
T	3
B	2
E	2
N	1
R	1



Fonte: https://commons.wikimedia.org/wiki/File:Huffman_(To_be_or_not_to_be).svg

Código de Huffman

- A versão original desse algoritmo possui uma complexidade de tempo O(n log n), em que n é o número de símbolos no texto.
- Se a frequência dos símbolos já for conhecida, a complexidade de tempo cai para O(n).
- É possível perceber que a codificação de Huffman é única para cada texto, pois, como já foi dito, leva em conta as frequências dos símbolos que compõem o texto, que é algo único para cada um.
- Essa codificação é base para os algoritmos de compressão modernos: uma codificação pelo código de Huffman reduz, em média, um texto para 30% do tamanho que se obteria com uma codificação com um número fixo de bits para cada símbolo.
    - A frequência dos símbolos afeta o resultado: quanto maiores as diferenças entre as quantidades de ocorrências dos símbolos, maior a diferença de bits entre símbolos mais e menos frequentes.

Interatividade

Em quais tipos de problemas computacionais são mais aplicáveis os algoritmos gulosos?

a) Problemas de ordenação.
b) Problemas de tomada de decisão.
c) Problemas de codificação.
d) Problemas de busca de dados.
e) Problemas de otimização.

Resposta

Em quais tipos de problemas computacionais são mais aplicáveis os algoritmos gulosos?

a) Problemas de ordenação.
b) Problemas de tomada de decisão.
c) Problemas de codificação.
d) Problemas de busca de dados.
e) Problemas de otimização.

Análise amortizada

- O uso da notação Grande-O para indicar a complexidade de um algoritmo pressupõe que a função matemática envolvida seja conhecida, ou, pelo menos, fácil de ser obtida a partir das instruções que o algoritmo executa.
- Mas existem casos em que não conseguimos determinar essa função.
- Para calcularmos a complexidade de tempo de um algoritmo para o qual não sabemos a função de complexidade, consideremos uma sequência de n execuções de uma dada operação desse algoritmo que atua sobre uma estrutura de dados E.
- Cada uma das execuções da operação terá um determinado custo de tempo, o qual dependerá do estado atual da estrutura E e, portanto, das eventuais execuções anteriores dessa operação.
    - Teremos, então, operações caras (que consomem grande quantidade de tempo) e operações baratas (que consomem uma pequena quantidade de tempo).

Análise amortizada

- É razoável pressupor, na maioria dos casos, que cada operação cara será precedida (e seguida) por diversas operações baratas.
- A análise amortizada tenta fazer estimativas de consumo de tempo para esse tipo de situação.
- Cada operação tem seu custo de tempo e podemos novamente supor que exista um número c tal que o custo das n primeiras operações seja necessariamente menor que $c \cdot n$.
- Assim, assumimos que todas as operações possuem um custo menor que c.
  - Na verdade, isso é uma média: operações baratas consumirão menos tempo que c, enquanto operações caras custarão mais, mas essa diferença será amortizada (daí o nome dessa análise) pelo tempo que "sobrou" das execuções baratas.
  - Assim, podemos definir o tempo geral de execução do algoritmo como sendo $c \cdot n$.

Análise amortizada

- Na análise amortizada, é calculada a média do tempo para executar um conjunto de operações em um algoritmo.
- A análise amortizada é diferente da análise do caso médio. Essa segunda envolve determinar estatisticamente qual a situação mais comum de entradas no algoritmo e estimar o tempo de execução médio baseada nisso.
- A análise amortizada não é uma função de complexidade, mas sim do tempo estimado para o pior caso do algoritmo.
- Por exemplo, a estimativa do número de iterações do algoritmo de Ford-Fulkerson é obtido pela análise amortizada.
    - O número de iterações $V \cdot M$ é o pior caso possível. Na prática, a grande maioria das execuções desse algoritmo irá realizar uma quantidade muito menor de iterações.

Processamento paralelo e threads

- Todos os algoritmos e programas apresentados até o momento foram sequenciais.
- Isso significa que eles foram pensados para terem uma instrução executada de cada, em uma ordem específica.
- Porém, nas últimas décadas, os computadores passaram a ser multiprocessadores, com mais de uma unidade de processamento.
- Além disso, temos a expansão dos sistemas computacionais distribuídos, que também contam com diversas unidades de processamento separadas.
- Dessa forma, acompanhando o desenvolvimento do hardware, surgiu o conceito de processamento paralelo.

Processamento paralelo e threads

- O processamento paralelo levou ao surgimento do conceito de multithreading, ou seja, o processamento simultâneo de múltiplos threads.
- Um thread pode ser definido como sendo uma unidade de execução individual que tem origem em conjunto maior, tal como um programa ou aplicativo.
- Esse modelo de execução assume que diferentes threads podem ser criados e executados de forma individual, compartilhando recurso de hardware.
- A forma mais comum de programarmos fazendo uso da arquitetura paralela de computadores é por meio do thread estático.

Processamento paralelo e threads

- Isso significa que podemos construir funções que atuam de forma independente, mas compartilhando recursos de memória.
- Um thread é carregado para execução pelo processador e encerrado quando concluído ou quando a execução de outro thread se faz necessária.
- Porém, como o processo de criação e encerramento (muitas vezes chamado de "destruição") dos threads é um processo lento, muitas vezes os threads criados permanecem na memória, daí a denominação threads estáticos.
- A criação e destruição de threads é feita pelo processador, não pelo programa ou pelo algoritmo, de acordo com as demandas de recursos (usualmente memória) ao longo do processamento.
    - O uso de threads estáticos é difícil, uma vez que gerenciar os protocolos adequados de comunicação entre eles envolve acréscimo no trabalho e no código.

Multithread dinâmico

- Um multithread dinâmico é um modelo em que a plataforma na qual os threads são executados gerencia automaticamente o balanceamento de recursos, sem que os programadores precisem se preocupar em como isso será feito.

Nesse modelo temos duas funcionalidades:

- Paralelismo aninhado: isso possibilita que um novo thread seja gerado, possibilitando que o ativador dessa geração continue executando seu processamento enquanto esse novo thread está sendo executado em paralelo.
    - Laços paralelos: sob muitos aspectos, é um laço de repetição normal, porém com a possibilidade de que as múltiplas iterações sejam executadas paralelamente (e não apenas sequencialmente).

Multithread dinâmico

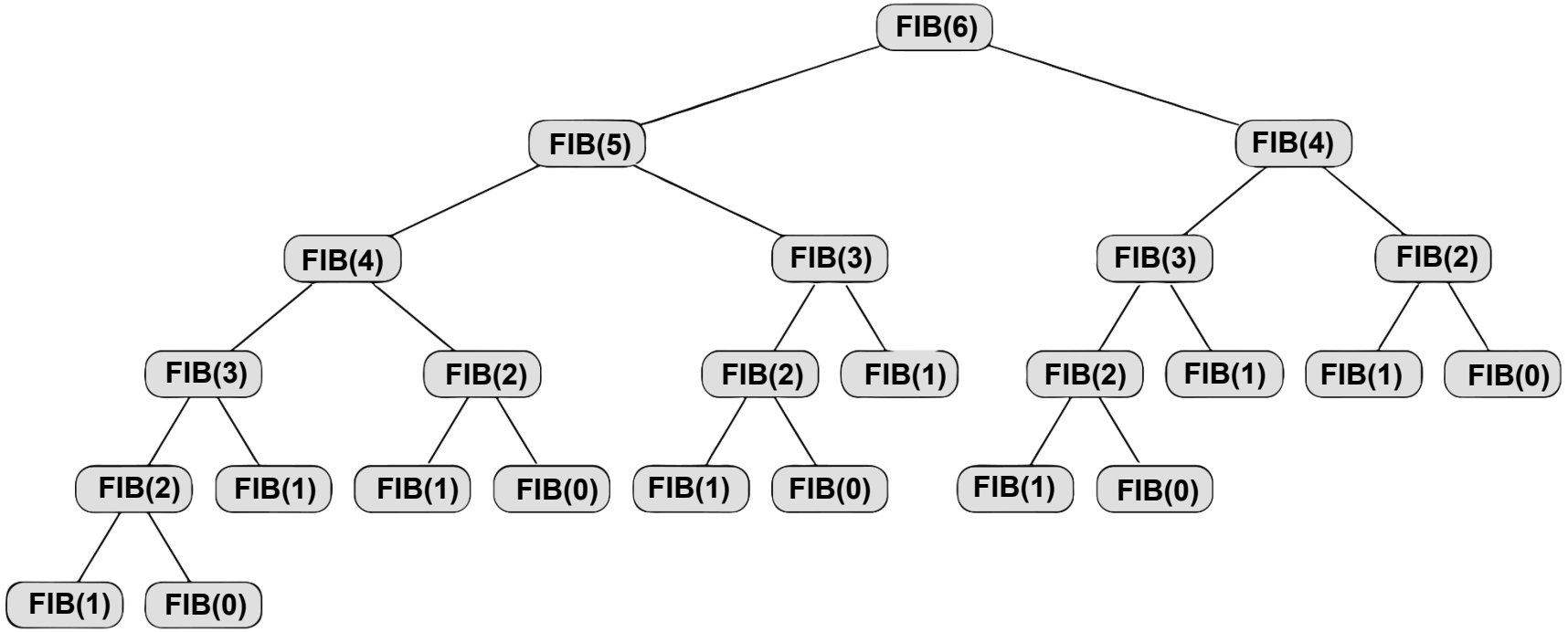
Exemplo: uma função recursiva para calcular o n-ésimo número da sequência de Fibonacci (1, 1, 2, 3, 5, 8, 13, ...):

```
def Fib (n):
    if n <=1:
        return n
    else:
        return Fib  (n-1) +  Fib (n-2)
```

- Temos um problema aqui: ao iniciarmos o cálculo do número seguinte na sequência, ela ignora todo os esforços realizados nos cálculos anteriores, recomeçando do zero o processo.

Multithread dinâmico

Para calcularmos, por exemplo Fib(6), alguns valores terão que ser calculados diversas vezes:



Fonte: adaptado de: Cormen, 2012, p 562.

Multithread dinâmico

- Uma outra abordagem: a execução da função se dá na forma de um thread, ficando seu resultado, para um dado valor, armazenado na memória para as execuções posteriores.
- Por exemplo, quando fizermos Fib2 (5) e precisarmos, pela recursividade, de Fib2 (4) e Fib2 (3), elas já estarão calculadas e poderão ser empregadas.

Assim, transformando nosso exemplo em um programa multihread:

Multithread dinâmico

```
import threading
def Fib2 (n, resultado):
    fib=[1,1]
    for i in range(2, n):
      fib.append(fib[-1] + fib[-2])
    resultado.extend(fib)
n = int(input("Digite o número de elementos da sequência de Fibonacci: "))
resultado=[]
# Criar um thread e executar para calcular a sequência de Fibonacci
thread1 = threading.Thread(target = Fib2, args = (n, resultado))
thread1.start()
thread1.join()
# Exibir o resultado
print("Sequência de Fibonacci com {n} elementos:", resultado)
```

Multithread dinâmico

- Assim, um algoritmo multithread acaba por ser muito mais veloz do que um programa recursivo simples.
- A maioria das linguagens de programação mais modernas (Python, Java etc.) possuem módulos, classes e métodos para permitir o desenvolvimento de programas multithread.
- Porém, o uso desse tipo de algoritmo torna a execução (e por consequência, o tempo gasto) do algoritmo mais dependente do ambiente em que está sendo realizado.

Interatividade

Em que caso aplica-se a análise amortizada em relação ao tempo de execução de um algoritmo?

a) Quando sua complexidade de tempo é muito ruim, para melhorar a eficiência do algoritmo.
b) Para compararmos dois algoritmos distintos que podem ser empregados para resolver um mesmo problema.
c) Quando o algoritmo é empregado em problemas envolvendo grafos.
d) Quando uma função de complexidade de tempo não pode ser obtida diretamente a partir do número de entradas do algoritmo.
e) Em qualquer situação na qual um algoritmo possuirá ao menos um trecho iterativo ou recursivo.

Resposta

Em que caso aplica-se a análise amortizada em relação ao tempo de execução de um algoritmo?

a) Quando sua complexidade de tempo é muito ruim, para melhorar a eficiência do algoritmo.
b) Para compararmos dois algoritmos distintos que podem ser empregados para resolver um mesmo problema.
c) Quando o algoritmo é empregado em problemas envolvendo grafos.
d) Quando uma função de complexidade de tempo não pode ser obtida diretamente a partir do número de entradas do algoritmo.
e) Em qualquer situação na qual um algoritmo possuirá ao menos um trecho iterativo ou recursivo.

Problemas NP

De uma forma simplificada, podemos classificar os problemas computacionais em três grandes grupos:

1. Problemas de otimização.
2. Problemas de busca: nesses problemas, queremos verificar a existência e localizar algo que satisfaça determinadas condições.
3. Problemas de tomada de decisão: envolvem a escolha de uma dentre duas alternativas: executar ou não uma instrução, seguir ou não por uma determinada aresta em um grafo.

Problemas NP

- Problemas de busca podem ter uma solução mais complexa.
- Exemplo: uma van de entregas sai da transportadora para realizar doze entregas. Qual a melhor ordem (que percorra menor distância ou gaste menos tempo) para realizar essas entregas e retornar para a transportadora?
- A aplicação de um algoritmo guloso parece uma boa alternativa, mas esses algoritmos se baseiam na melhor decisão no momento, não levando em consideração o conjunto de decisões.
- Uma última alternativa seria uma solução de força bruta: calcular todas as possíveis rotas e escolher a melhor. Porém, esse algoritmo tem um problema: existem 12! rotas possíveis (mais de 470 milhões).

Problemas NP

Para entendermos a matemática envolvida, vamos utilizar dois conjuntos:

- P: designa todos os problemas que podem ser resolvidos com complexidade de tempo no máximo polinomial.
- NP: um problema que, uma vez que tenhamos uma proposta de solução para ela, possamos verificar se essa solução é válida em tempo de execução polinomial.

- Todo problema P é NP, mas não se sabe se todo problema NP é P.
- O termo NP significa “tempo polinomial não determinístico” (em inglês, “nondeterministic polynomial time”).

Problemas NP

- A questão “Todos NP são P?”, resumido na forma “P = NP?” é uma das grandes questões em aberto da matemática.
- Exemplo: consideremos um conjunto de números inteiros, positivos e negativos, com n elementos. Existe algum subconjunto não vazio desse conjunto cuja soma dos seus elementos resulte em zero?
- Existem 2^N subconjuntos possíveis, assim, obter e testar os subconjuntos envolve uma complexidade de tempo que é exponencial (maior que polinomial).
- Porém, verificar se um determinado conjunto é uma resposta é bem simples, com uma complexidade de tempo linear igual ao número de elementos do subconjunto.

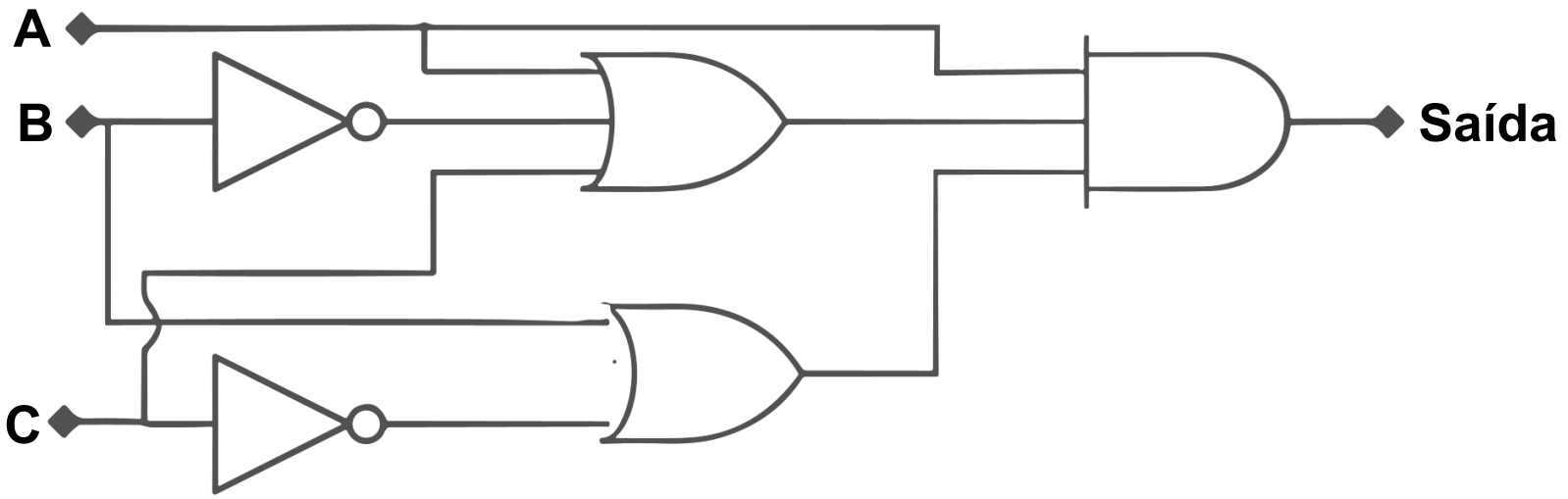
Problemas NP

- NP-difícil: um problema que, caso exista um algoritmo de tempo polinomial para resolvê-lo, poderemos converter todos os problemas em NP para esse problema, de forma a poder resolver todos os problemas NP em tempo polinomial.
- NP-completo: um problema que é NP-difícil e também é um problema NP.
- Em 1971, Steven Cook provou que o problema SAT é um problema NP-completo. Em 1984, Leonid Levin expandiu essa demonstração, criando o chamado Teorema de Cook-Levin.
- Uma das consequências desse teorema é demonstrar que todos os problemas NP-completos podem ser reduzidos a um problema SAT.
    - Assim, se existir uma solução em tempo polinomial para um problema NP-completo, existe uma solução em tempo polinomial para o problema SAT.
    - E, se existe uma solução polinomial para o problema SAT, já que ele é transformável em qualquer problema NP-completo, existe solução em tempo polinomial PARA TODOS os problemas NP-completos.

Problema SAT e Circuit-SAT

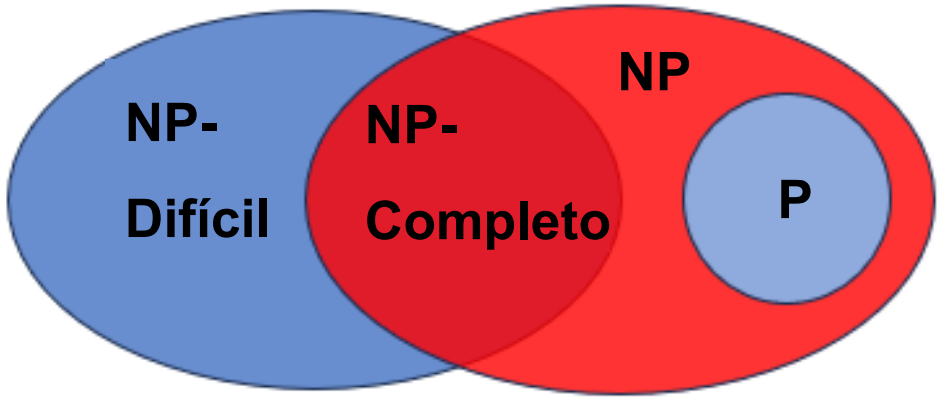
- Esses são alguns dos problemas mais importantes dentre os problemas NP-completos.
- O enunciado de problema Circuit-SAT pode ser descrito como “Dado um circuito lógico composto por portas lógicas NOT, AND e OR, com N entradas e uma única saída, ele é satisfazível?”

Ou seja, existe pelo menos uma combinação de valores lógicos das entradas que faz com que a saída do circuito seja igual a 1?

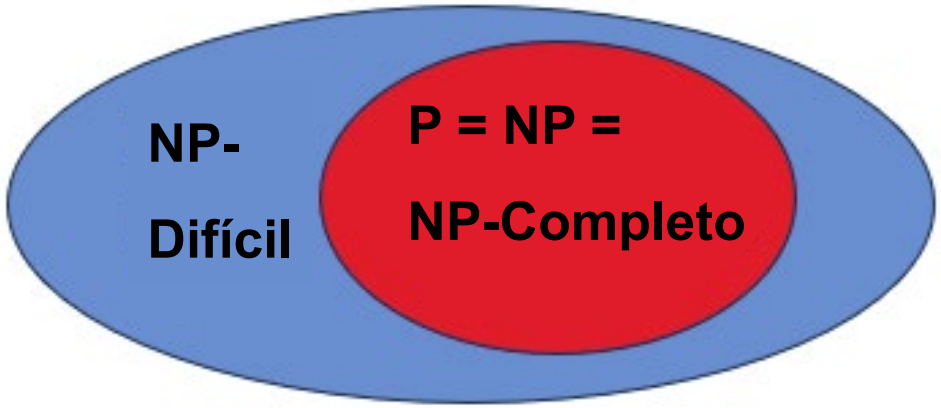


Fonte: livro-texto

P = NP?



P ≠ NP



P = NP

Fonte: livro-texto

Alguns outros problemas NP-Completos

- Problema da soma dos subconjuntos

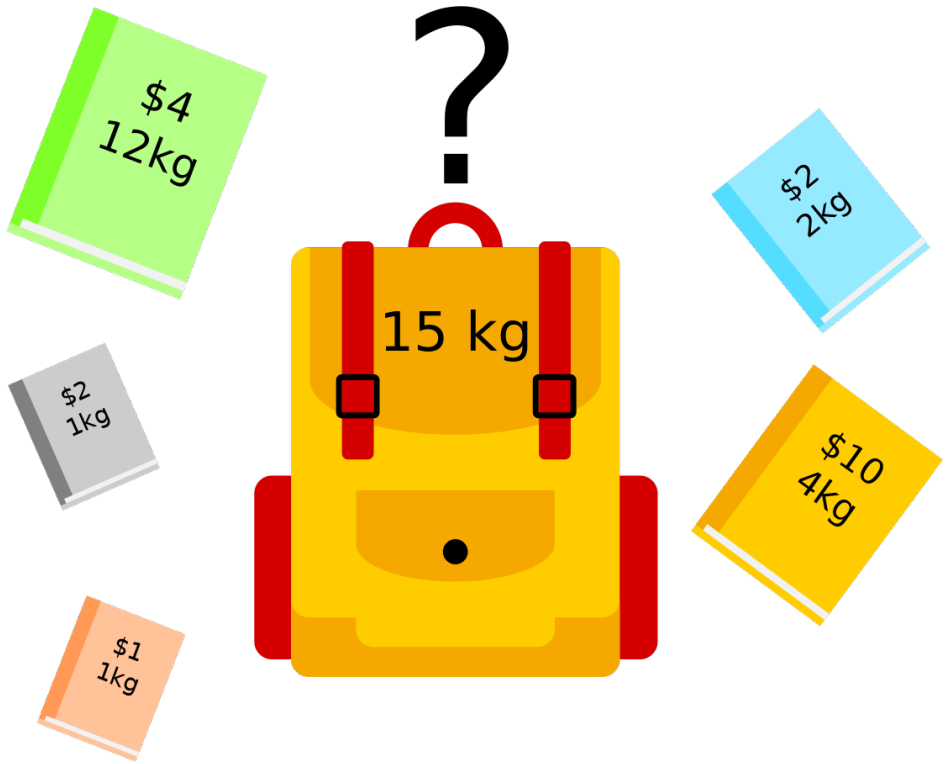
Dado um conjunto finito de números inteiros, positivos e negativos, com n elementos, existe algum subconjunto não vazio desse conjunto cuja soma dos seus elementos resulte em um determinado valor específico?

Uma variação é o problema da partição: como dividir um conjunto em dois subconjuntos (não necessariamente com a mesma quantidade de elementos) de forma que a soma dos elementos de cada um dos dois deles seja igual?

Alguns outros problemas NP-Completos

- Problema da mochila com repetição
- Esse problema é uma variação mais elaborada do problema da soma dos subconjuntos.

Dado um conjunto de diferentes objetos, cada um com um peso e um valor associado (podendo haver mais de um objeto do mesmo tipo), como podemos preencher uma mochila com determinada capacidade máxima de peso de forma a obtermos o maior valor possível dentro da mochila?



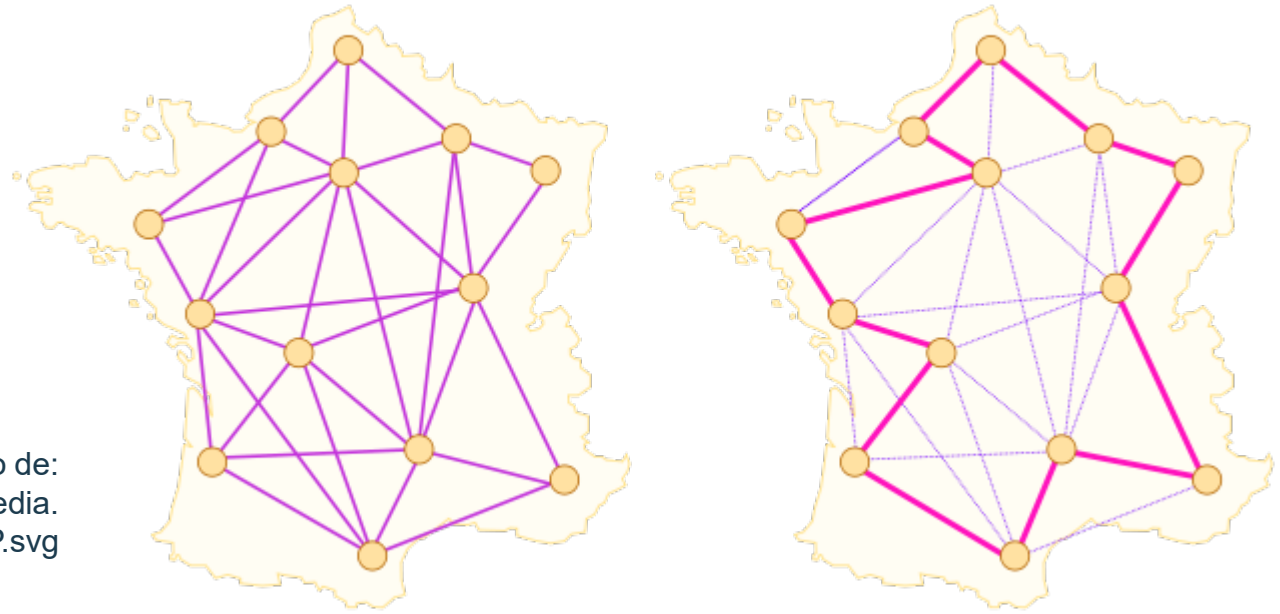
Fonte: https://commons.wikimedia.org/wiki/File:Knapsack_Problem_Illustration.svg

Alguns outros problemas NP-Completos

- Problema do caixeiro viajante

Como percorrer uma série de cidades, visitando cada uma delas uma única vez e retornando para a cidade de origem, de forma a otimizar o trajeto (percorrendo a menor distância) entre elas?

- É um problema de logística que se baseia na necessidade de realizar entregas em diversos locais, percorrendo o menor caminho possível, reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível.



Fonte: adaptado de: https://commons.wikimedia.org/wiki/File:Aco_TSP.svg

Como resolver?

Sabendo que existem aqueles problemas para os quais não se conhece (percebam que não usamos a expressão "não existe") um algoritmo ideal, como podemos resolvê-los?

Algumas abordagens:

- Busca exaustiva
- Algoritmos gulosos
- Busca heurística
- Algoritmos genéticos
- Computação quântica

Interatividade

O que significa dizer que um problema é NP?

a) Ele não possui um algoritmo que o resolva com complexidade de tempo polinomial.
b) Ele possui um algoritmo que o resolva com complexidade de tempo polinomial.
c) Ele não pode ter uma proposta de solução testada em tempo polinomial.
d) Ele pode ter uma proposta de solução testada em tempo polinomial.
e) Não se conhece um algoritmo específico para sua resolução.

Resposta

O que significa dizer que um problema é NP?

a) Ele não possui um algoritmo que o resolva com complexidade de tempo polinomial.
b) Ele possui um algoritmo que o resolva com complexidade de tempo polinomial.
c) Ele não pode ter uma proposta de solução testada em tempo polinomial.
d) Ele pode ter uma proposta de solução testada em tempo polinomial.
e) Não se conhece um algoritmo específico para sua resolução.

Referências

- DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. *Algoritmos*. Porto Alegre: McGraw-Hill, 2009.
- CORMEN, T. *Algoritmos*: teoria e prática. Rio de Janeiro: LTC, 2012.
- CORMEN, T. *Desmistificando algoritmos*. Rio de Janeiro: Campus, 2013.

ATÉ A PRÓXIMA!