

Unidade II

5 GRAFOS, FLUXO EM REDES E ANÁLISE AMORTIZADA

Neste tópico vamos resumir os conceitos fundamentais de teoria dos grafos e analisar os principais algoritmos de grafos e redes. Como não é fácil definir a função de complexidade desse importante algoritmo para analisar o fluxo máximo em redes, também apresentaremos a análise amortizada, outra ferramenta que determina a complexidade de tempo de um algoritmo.

5.1 Fundamentos da teoria dos grafos

Para efeitos deste livro, podemos definir grafo como um conjunto finito de elementos distintos e separados que se conectam. Essas conexões não serão necessariamente de todos os elementos entre si, mas é necessário que ao menos algumas conexões estejam presentes para um conjunto de elementos poder ser representado com um grafo (Lipschutz; Lipson, 2013, p. 166).

Assim, um grafo G será composto por:

- um conjunto V de vértices (também chamados de pontos ou nós) representando 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 .

Agora, algumas terminologias sobre grafos empregadas no desenvolvimento de algoritmos:

- **Grafo simples:** dois vértices são conectados por apenas uma aresta. Não há arestas paralelas nem outras que se iniciem e terminem no mesmo vértice.
- **Grafo conexo:** todos os vértices do grafo estão conectados por pelo menos uma aresta (se não houver ao menos uma aresta conectando um ou mais vértices, o grafo é dito desconexo).
- **Grafo ponderado:** cada aresta tem valor numérico associado que corresponde ao peso da aresta.
- **Grafo direcionado:** algumas (ou todas as) arestas só podem ser percorridas em um sentido. Esse gráfico costuma ser representado por setas que indicam o sentido das arestas direcionais.

Esses diferentes tipos de grafo podem ocorrer simultaneamente; por exemplo, um grafo pode ser ponderado e direcionado ao mesmo tempo. A figura 25 os ilustra, sendo os vértices representados por letras:

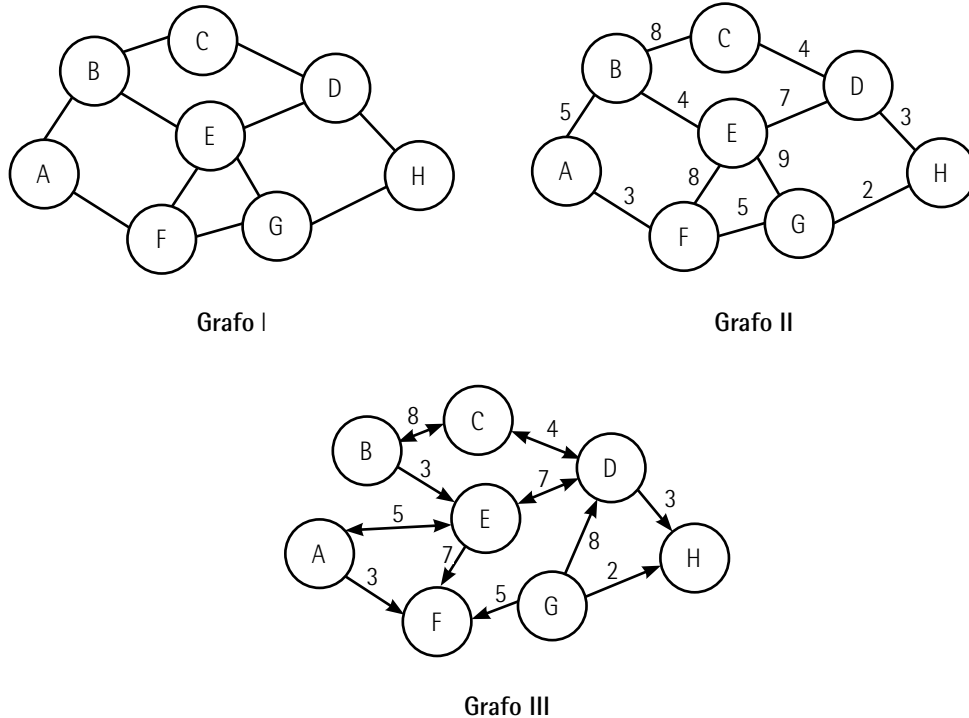


Figura 25 – Exemplos de grafo: grafo I – simples conexo; grafo II – ponderado; grafo III – direcionado ponderado

Computacionalmente, precisamos de outra forma de representar os grafos; por isso utilizamos a **matriz de adjacências**, uma matriz matemática na qual cada linha representa um dos vértices do grafo, e os valores são as conexões que esse vértice estabelece (Lipschutz; Lipson, 2013, p. 206):

- Para um grafo não ponderado, a matriz de adjacências conterá apenas 1 (se houver conexão entre dois nós) e 0 (se não houver conexão entre dois nós).
- Para um grafo ponderado, a matriz conterá o peso das arestas na posição correspondente aos vértices que se conectam e 0 (se não houver conexão entre dois vértices).
- Para um grafo direcionado, pode haver o caminho de um vértice V1 para um vértice V2, mas pode não haver o caminho oposto, de acordo com o sentido das arestas.

Dessa forma, para os grafos da figura 25, as matrizes correspondentes de adjacência são as apresentadas na figura 26:

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

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

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

Figura 26 – Matrizes de adjacências para os grafos da figura 25



Observação

A principal diagonal de uma matriz de adjacência em um grafo simples terá sempre valores 0 na diagonal principal, e a matriz será simétrica se o grafo não for orientado.

Porque em Python não temos matrizes como tipo de dado composto, utilizaremos uma lista de listas para representar a matriz de adjacências. Todos os algoritmos aqui apresentados trabalharão com grafos nessa forma (o código para o algoritmo de Dijkstra apresentado no tópico 3 utiliza a mesma representação).

Assim, novamente para os grafos do exemplo, teremos no Python:

Grafo I = [[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]]

Grafo II = [[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]]

Grafo III = [[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]]

Um grafo pode ser transformado em árvore, fazendo-se com que um vértice se torne sua raiz e as arestas se tornem seus galhos (ramificações), mas isso pode gerar alguns problemas. Assim, ao aplicar alguns dos algoritmos vistos no tópico 3, precisaremos de algumas adaptações para evitar isso.

5.2 Algoritmos sobre grafos

Vamos adaptar os algoritmos de busca que vimos no tópico 3. Em vez de receber uma árvore, aqui as funções vão executar o mesmo algoritmo, recebendo como parâmetro um grafo na forma de matriz de adjacências; porém um possível problema é criar uma árvore infinita, na qual um algoritmo comece a dar voltas ao longo da estrutura de dados sem nunca chegar a um resultado. Por exemplo, olhemos o grafo I das figuras anteriores: se o transformarmos numa árvore com o vértice A como raiz, seus nós-filhos serão os vértices B e F; no entanto, os filhos de B serão A, C e E. Assim, existe a chance – se o algoritmo não previr isso – de ficarmos presos em um trajeto A-B, por exemplo, indefinidamente (ou em processo de avançar e retroceder a cada iteração, tornando o algoritmo lento e sem rumo).

Para evitar isso, os algoritmos utilizados em árvores devem conter uma listagem dos nós já visitados. Ambos os algoritmos apresentados estão na forma mais completa, aceitando como entrada um grafo ponderado ou direcionado.

5.2.1 Busca em profundidade

A função em Python a seguir executa o algoritmo de busca em profundidade em um grafo, partindo de um vértice denominado **origem** até um vértice denominado **destino**. A lista **visitados** registra quais nós já foram visitados, assim não existirá um retrocesso no desenvolvimento em um mesmo caminho (todos os caminhos possíveis que não repetem vértices serão listados). A função é recursiva, e a cada execução ela reinicia as listas **visitados** e **caminhos** (que registram o caminho produzido em dada iteração).

```
def busca_profundidade(grafo, origem, destino, visitados=None, caminho=None):
    #Reinicia as listas ao início da iteração
    if visitados is None:
        visitados = [False] * len(grafo)
    if caminho is None:
        caminho = []
    visitados[origem] = True
    caminho.append(origem)
    #Executa a busca em profundidade
    if origem == destino:
        print("Caminho encontrado:", caminho)
    else:
        for i, adjacente in enumerate(grafo[origem]):
            if adjacente != 0 and not visitados[i]:
                busca_profundidade(grafo, i, destino, visitados, caminho)
        caminho.pop()
        visitados[origem] = False
```

5.2.2 Busca em largura

A adaptação do algoritmo de busca em largura é mais simples, porque basta registrar os vértices já visitados, e não é necessário recursão. Por outro lado, esse algoritmo exige maior movimentação ao longo da estrutura, pois não é uma árvore.

```
def busca_largura(grafo, origem, destino):
    #Reinicia as listas ao início da iteração
    fila = []
    visitados = [False] * len(grafo)
    parent = [-1] * len(grafo)
    fila.append(origem)
    visitados[origem] = True
    #Executa a busca em profundidade
    while fila:
        atual = fila.pop(0)
        if atual == destino:
            caminho = []
            while destino != -1:
                caminho.insert(0, destino)
                destino = parent[destino]
            return caminho
        for i, peso in enumerate(grafo[atual]):
            if peso >= 1 and not visitados[i]:
                fila.append(i)
```

```
visitados[i] = True
parent[i] = atual
#Retorna vazio se não encontrou um caminho
return None
```

5.2.3 Grafo euleriano

Um grafo é dito euleriano se for 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 o problema foi dada pelo matemático Leonhard Euler (1707-1783) diante da questão das sete pontes de Königsberg (cidade da Prússia que à época tinha sete pontes, e os habitantes se perguntavam se era possível cruzar todas elas na sequência, sem passar duas vezes pela mesma). A solução é simples: o grafo pode ter todas as arestas percorridas se tiver nenhum ou dois vértices de grau ímpar (o grau de um vértice é o número de arestas que se conectam a ele). No entanto, se o grafo tiver duas arestas com grau ímpar, o trajeto começará em um desses vértices e terminará no outro.

É simples verificar se um grafo é euleriano, embora a localização do trajeto não seja. A função em Python a seguir recebe um grafo na forma de uma matriz de adjacências e verifica se é euleriano (retornando verdadeiro ou falso):

```
def grafo_euleriano (grafo):
    no_impar = 0
    for i in grafo:
        if (len(i) - i.count(0)) % 2 != 0:
            no_impar += 1
    if no_impar == 0 or no_impar == 2:
        return True
    else:
        return False
```



Saiba mais

Para mais detalhes sobre as pontes de Königsberg, leia as páginas 160 e 161 deste livro:

LIPSCHUTZ, S.; LIPSON, M. *Matemática discreta*. Porto Alegre: Bookman, 2013.

O artigo a seguir também aprofunda o tema:

TAYLOR, P. What ever happened to those bridges? *Australian Mathematics Trust*, 2000. Disponível em: <https://tinyurl.com/m5r3c4jn>. Acesso em: 7 dez. 2023.

5.2.4 Gráfico hamiltoniano

Um grafo é denominado **hamiltoniano** se houver um trajeto que permita passar por todos os vértices do grafo, sem repetir nenhum. Se o trajeto existir e se iniciar e terminar no mesmo vértice, diz-se que existe um ciclo hamiltoniano no grafo. O nome deriva do matemático irlandês William Hamilton (1803-1865).



Observação

O fato de um grafo ser hamiltoniano ou não é um problema NP-completo e será tratado no tópico 8.

5.2.5 Algoritmo de Kruskal

Reduz um grafo conexo a uma **árvore geradora mínima** (embora seja comumente usado em grafos ponderados, aplica-se também em grafos sem peso nas arestas). Árvore geradora mínima é a que contém todos os vértices e apenas as arestas mínimas necessárias, de menor peso. O algoritmo de Kruskal é guloso (ver tópico 6), e seu nome vem do seu criador, o matemático americano Joseph Bernard Kruskal Junior (1928-2010).

Pode ser definido em duas etapas:

- selecionar a aresta de menor peso presente no grafo. Se ambos os vértices conectados por ela estiverem na mesma árvore (gerada pelas arestas que permaneceram das iterações anteriores), deve-se removê-la; caso contrário, a aresta permanecerá;
- repetir o processo em ordem crescente das arestas até não haver mais arestas disponíveis.

Exemplo de aplicação

A seguir, um exemplo do algoritmo de Kruskal no grafo II da figura 25:

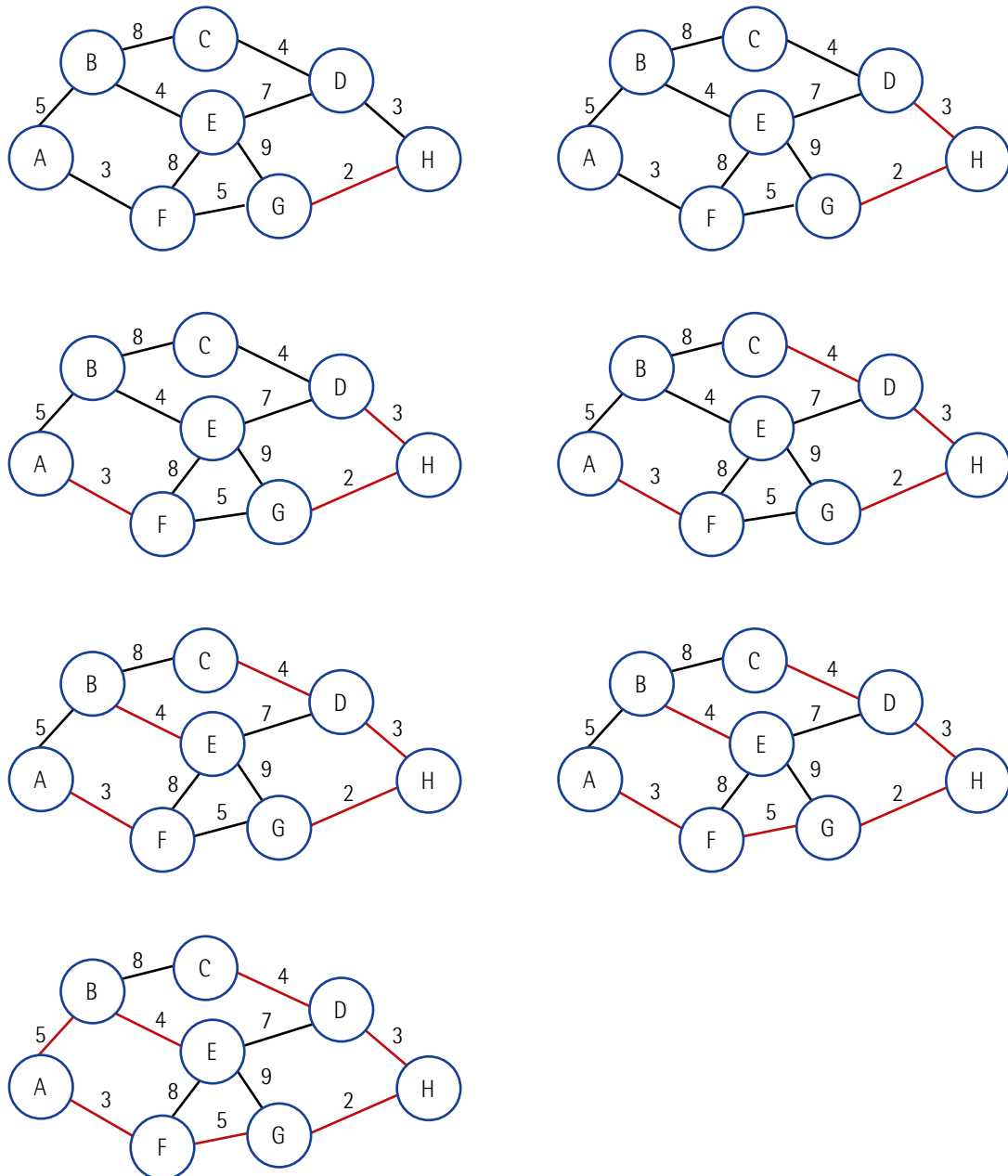


Figura 27

A partir do ponto mostrado na figura 26, todas as arestas seguintes estão conectadas a dois vértices que já têm pelo menos uma conexão cada; assim, todas as arestas restantes serão removidas. A árvore geradora mínima para esse grafo será:

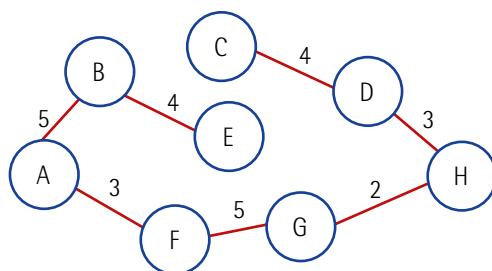


Figura 28

Um grafo reduzido a árvore geradora pode ser tratado como árvore onde qualquer vértice pode ser considerado raiz e, como já sabemos, a árvore resultante não será necessariamente binária ou balanceada.



Observação

À primeira vista, pode parecer que uma árvore geradora mínima represente um trajeto hamiltoniano, mas a árvore resultante pode ter ramificações que impeçam um trajeto por todos os vértices.

As funções em Python a seguir aplicam o algoritmo de Kruskal em um grafo na forma de uma matriz de adjacências, retornando uma nova matriz de adjacências da árvore geradora mínima correspondente:

```
#Função recursiva para verificar as conexões de um vértice
def encontrar(subconjuntos, i):
    if subconjuntos[i] == -1:
        return i
    return encontrar(subconjuntos, subconjuntos[i])

#Função para acrescentar à lista arvore_geradora as arestas que serão mantidas
def unir(subconjuntos, x, y):
    raiz_x = encontrar(subconjuntos, x)
    raiz_y = encontrar(subconjuntos, y)
    subconjuntos[raiz_x] = raiz_y

def kruskal(matriz_adjacencias):
    arestas = []
    vertices = len(matriz_adjacencias)
    for i in range(vertices):
        for j in range(i + 1, vertices):
            if matriz_adjacencias[i][j] != 0:
                arestas.append((i, j, matriz_adjacencias[i][j]))
    #Ordenação das arestas pelo seu peso
    arestas = sorted(arestas, key=lambda aresta: aresta[2])
    subconjuntos = [-1] * vertices
    #Criação da árvore, inicialmente vazia
    arvore_geradora = [[0] * vertices for _ in range(vertices)]
    arestas_adicionadas = 0
```

```
for aresta in arestas:
    x, y, peso = aresta
    raiz_x = encontrar(subconjuntos, x)
    raiz_y = encontrar(subconjuntos, y)
    if raiz_x != raiz_y:
        unir(subconjuntos, raiz_x, raiz_y)
        arvore_geradora[x][y] = peso
        arvore_geradora[y][x] = peso
        arestas_adicionadas += 1
    if arestas_adicionadas == vertices - 1:
        break
return arvore_geradora
```

5.3 Fluxo em redes

Podemos definir rede como um grafo no qual existem deslocamentos de entidades entre os vértices utilizando exclusivamente as arestas que as conectam como caminho. Considere uma rede D , representada por um grafo ponderado, no qual o peso da aresta E é um valor positivo c que significa a capacidade da aresta. Essa capacidade é o fluxo máximo dos elementos através desse caminho.



Observação

Fluxo pode ser definido como o valor de uma unidade em função do tempo. Por exemplo: bytes por segundo, litros por minuto, veículos por hora etc.

Um dos vértices do grafo é denominado **origem** (O), de onde parte o fluxo através da rede; um segundo nó é denominado **destino** (D), para onde os fluxos se destinam. O objetivo, nesse problema, é calcular o maior fluxo possível entre os vértices de origem e destino da rede. Um fator que surge desse contexto é a saturação de arestas e vértices: cada aresta comporta um fluxo máximo, indicado pelo seu peso (lembrando que é um grafo ponderado). E 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.

Dentre as muitas aplicações que envolvem 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;
- organização de eventos que envolvam o deslocamento de pessoas entre diferentes pontos (feiras, exposições etc.).

Por exemplo, na figura 29 temos O como a origem e D como o destino. O valor em cada aresta indica a capacidade daquela aresta, e não temos nenhuma indicação de fluxo ainda:

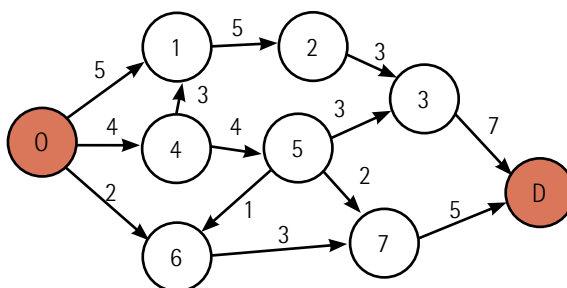


Figura 29 – Exemplo de rede

À primeira vista, pode parecer que o fluxo máximo possível seja 11, pois é a somatória dos fluxos máximos que podem sair de O, que é menor que a somatória máxima possível de chegar em D ($5 + 7 = 12$). Porém, será que todas as arestas ao longo dos diferentes trajetos (nos quais este fluxo vai se dividir) serão capazes de comportar esse fluxo? Por exemplo, analisemos o vértice 1: embora as arestas que chegam nele somem um fluxo total de 8 ($5 + 3$), dele pode sair um fluxo de apenas 5; assim, alguma das arestas dele precisará estar abaixo da sua capacidade máxima, pois, lembrando, o fluxo que chega em um vértice deve ser igual ao fluxo que sai dele.

Para resolver o problema do fluxo máximo em redes, utilizamos o **algoritmo de Ford-Fulkerson**, que recebe esse nome de seus criadores, os matemáticos americanos Lester Randolph Ford Junior 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. O valor será zero se o fluxo for igual à capacidade, e nesse caso dizemos que aquela aresta está saturada. Caso não haja nenhum fluxo pela aresta naquele momento, a capacidade residual é a própria capacidade da aresta.

O algoritmo de Ford-Fulkerson é iterativo e, a cada iteração, cumpre as seguintes etapas:

- 1) Identificar um caminho qualquer que saia da origem e chegue até o destino, preferencialmente (mas não obrigatoriamente) o que tiver as arestas com menor capacidade. Esse caminho não pode conter nenhuma aresta saturada.

- 2) Identificar a aresta de menor capacidade residual no caminho e subtrair o valor dessa capacidade residual de todas as arestas no trajeto selecionado. Deve-se então observar que aresta de menor capacidade ficará com capacidade residual igual a zero, a qual 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 subtraído).

- 4) Repetir o processo com outro trajeto dentro da rede enquanto houver caminhos entre a origem e o destino possíveis na rede.

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



Observação

A escolha dos trajetos na etapa 1 de cada iteração pode influenciar o número total de iterações, mas não afetará o resultado final do fluxo máximo da rede.

Exemplo de aplicação

Vamos calcular o fluxo máximo na rede da figura 26:

Iteração 1: selecionamos o caminho indicado em vermelho. A aresta com menor capacidade é 2. Este será o valor da iteração, com o valor que será reduzido de todas as arestas no caminho selecionado (a capacidade residual de cada aresta aparece em roxo):

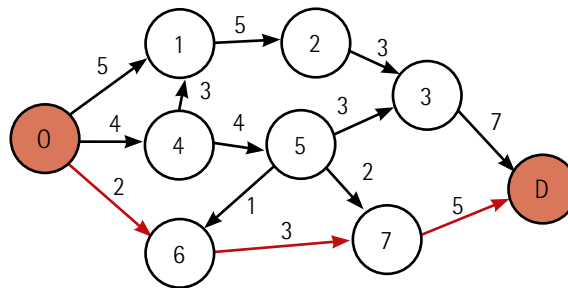


Figura 30

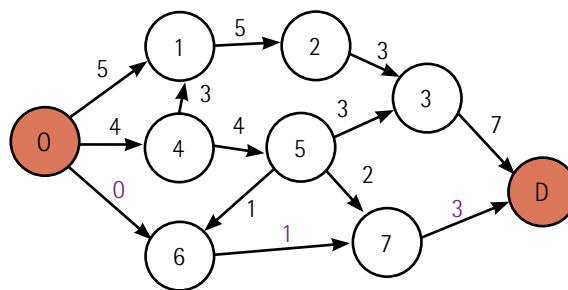


Figura 31

Valor da iteração 1: 2

Iteração 2: repetimos o processo em outro trajeto (arestas com capacidade residual igual a zero não estão mais disponíveis). O valor da segunda iteração é 1:

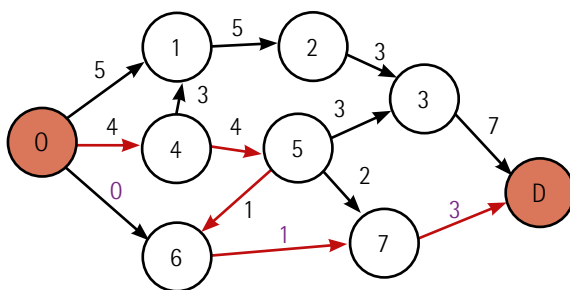


Figura 32

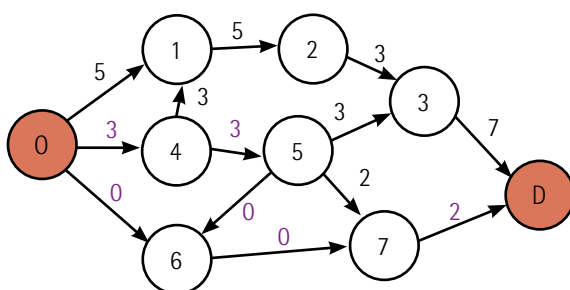


Figura 33

Valor da iteração 2: 1

Iteração 3: repetimos o processo em outro trajeto. O valor dessa iteração é 2. Podemos perceber que boa parte das arestas já está saturada:

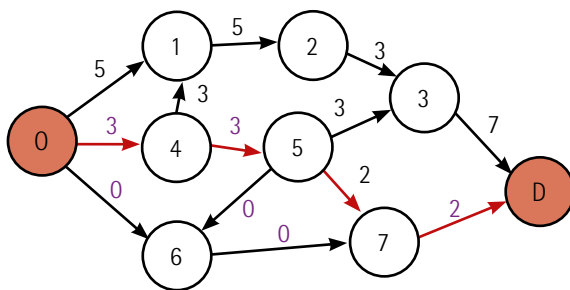


Figura 34

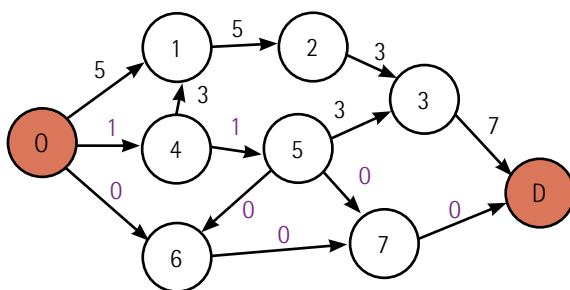


Figura 35

Valor da iteração 3: 2

Iteração 4: repetimos o processo em outro trajeto. O valor dessa iteração é 1:

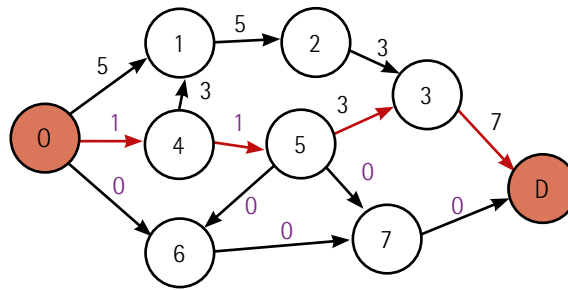


Figura 36

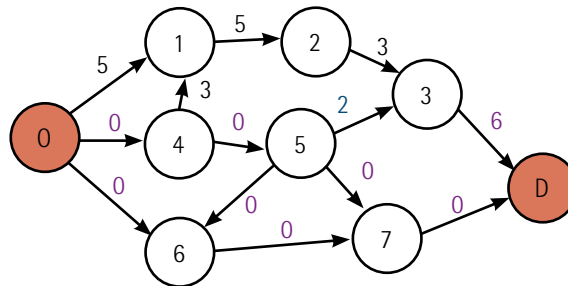


Figura 37

Valor da iteração 4: 1

Iteração 5: mais um trajeto selecionado. O valor dessa iteração é 3:

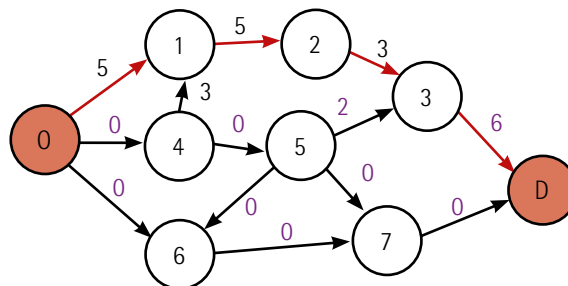


Figura 38

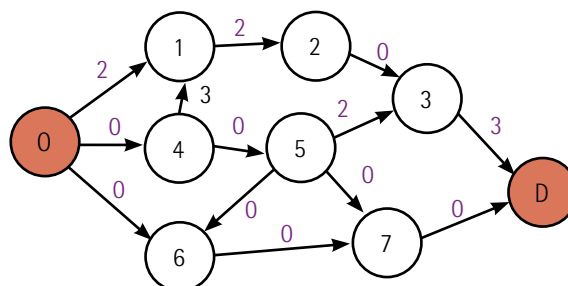


Figura 39

Valor da iteração 5: 3

Agora não conseguimos traçar mais nenhum trajeto entre a origem e o destino: qualquer outro trajeto passará por uma aresta saturada. Assim, o algoritmo se encerra após cinco iterações.

O fluxo máximo dessa rede será a soma dos valores de cada iteração: $2 + 1 + 2 + 1 + 3 = 9$. Atingido o fluxo máximo em uma rede, não estaremos necessariamente com todas as arestas saturadas: no exemplo, temos cinco arestas ainda com capacidade residual, e uma delas não está com fluxo nenhum (a aresta que liga os vértices 4 e 1).

A complexidade de tempo do algoritmo de Ford-Fulkerson é proporcional ao número de iterações, que por sua vez depende de quantos caminhos parciais diferentes podem ser traçados dentro da rede. Sendo M a maior capacidade na rede e V o número de vértices presentes (excluindo-se a origem e o destino), podemos facilmente calcular que o número máximo de iterações será $V \times M$.

Esse resultado sugere que teremos um comportamento polinomial como complexidade de tempo, mas podemos perceber que se, por exemplo, dobrarmos a capacidade M da maior aresta, dobraremos o número de iterações sem fazer qualquer alteração nos demais elementos da rede. Assim, uma pequena mudança na rede pode alterar drasticamente a execução do algoritmo sem alterar significativamente seu resultado. Na verdade, é um caso de tempo de execução estimado pela **análise amortizada**, ferramenta que apresentaremos a seguir.

O número de iterações no exemplo foi bem menor que o máximo calculado, $13 \times 7 = 91$. Isso se deve ao pequeno número de arestas presentes na rede, o que diminui o número de possíveis trajetos ao longo dela. O número máximo de iterações é atingido apenas se tivermos grafos completos (onde todos os vértices se conectam) e arestas com peso muito próximo entre si.

O programa em Python a seguir executa o algoritmo de Ford-Fulkerson em uma rede representada na forma de uma matriz de adjacências.

```
#Função de busca em largura adaptada
def busca_largura(grafo, origem, destino, parent):
    visitados = [False] * len(grafo)
    fila = [origem]
    visitados[origem] = True
    while fila:
        atual = fila.pop(0)
        for proximo, capacidade in enumerate(grafo[atual]):
            if not visitados[proximo] and capacidade > 0:
                fila.append(proximo)
                visitados[proximo] = True
                parent[proximo] = atual
    return visitados[destino]
```

```
#Função para o algoritmo de Ford-Fulkerson
def ford_fulkerson(grafo, origem, destino):
    parent = [-1] * len(grafo)
    fluxo_maximo = 0
    while busca_largura(grafo, origem, destino, parent):
        caminho_fluxo_minimo = float('inf')
        s = destino

        # Lista para armazenar o caminho dessa iteração
        caminho = []
        while s != origem:
            caminho.append(s)
            caminho_fluxo_minimo = min(caminho_fluxo_minimo, grafo[parent[s]][s])
            s = parent[s]
        caminho.append(origem)
        caminho.reverse()
        fluxo_maximo += caminho_fluxo_minimo
        v = destino

        # Exibir o caminho e o custo (fluxo) dessa iteração
        print(f'Caminho: {caminho}, Custo: {caminho_fluxo_minimo}')
        while v != origem:
            u = parent[v]
            grafo[u][v] -= caminho_fluxo_minimo
            grafo[v][u] += caminho_fluxo_minimo
            v = parent[v]
    return fluxo_maximo
```

5.4 Análise amortizada

A notação Grande-O para indicar a complexidade de um algoritmo pressupõe que a função matemática envolvida seja conhecida ou, pelo menos, fácil de obter a partir das instruções que o algoritmo executa em função da quantidade de entradas. Mas existem casos – como o do algoritmo que descrevemos no tópico anterior – onde não conseguimos determinar essa função. Para o algoritmo de Ford-Fulkerson não existe quantidade de entradas que possa ser variada.

Para calcular a complexidade de tempo de um algoritmo cuja função de complexidade ainda não sabemos, consideramos uma sequência de n execuções de uma dada operação desse algoritmo que atua sobre uma estrutura de dados E . Cada execução terá um custo de tempo que dependerá do estado atual da estrutura E e, portanto, das eventuais execuções anteriores. Teremos então operações caras (que consomem grande quantidade de tempo) e operações baratas (que consomem pequena quantidade de tempo). É razoável pressupor, na maioria dos casos, que cada operação cara será precedida (e seguida) por diversas operações baratas. A análise amortizada tenta estimar o consumo de tempo para essa 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 $c \times n$. Assim, assumimos que todas as operações têm 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 $c \times n$.

Exemplo de aplicação

Vamos aplicar essa análise ao algoritmo de Ford-Fulkerson. Supondo uma rede com capacidades das arestas como números naturais, teremos as seguintes questões:

- O maior caminho possível (operação mais cara do grafo) significa passar por todos os vértices da rede. Assim, o maior custo possível será o tempo de percorrer todos eles.
- No pior caso, o valor de cada iteração será 1. Assim, o número máximo de iterações será o valor da maior aresta M dividido por 1 (menor redução possível de cada aresta nesse caso), multiplicado pelo custo de percorrer o pior caminho (todos os vértices, V).

Assim, chegamos ao valor de tempo máximo de execução como o tempo de percorrer todos os nós do grafo (caminho hamiltoniano), multiplicado pelo valor da capacidade da maior aresta, ou seja, $M \times V$ (perceba que o valor de V aqui não significa mais o número de vértices, mas o custo de tempo para percorrer um caminho hamiltoniano no grafo).

Numa análise amortizada se calcula a média do tempo para executar um conjunto de operações num algoritmo. Assim, é possível demonstrar que o custo médio de operação é pequeno, mesmo que uma dada operação tenha um custo de tempo muito maior que as demais.



Observação

Análise amortizada é diferente de análise do caso médio; esta envolve determinar estatisticamente a situação mais comum de entradas no algoritmo e estimar o tempo de execução médio baseando-se nisso.

Expandindo o conceito de análise amortizada, vejamos a seguir as três técnicas mais utilizadas para determinar o custo de tempo de um algoritmo. Novamente, não estamos falando de uma função de complexidade, mas do tempo estimado para o pior caso (Cormem, 2013, p. 330):

- **Análise agregada:** nessa técnica um limite superior $T(n)$ é definido para executar uma sequência de n operações. O custo individual é o limite superior dividido pelo número de operações, gerando um custo médio $T(n)/n$, atribuído a todas as operações executadas pelo algoritmo, mesmo que sejam de tipos diferentes.
- **Método da contabilidade:** com essa técnica definimos um custo amortizado para cada operação, e operações diferentes terão custos diferentes. Nesse método algumas operações são "cobradas" a mais ou a menos do que realmente custam. O que é cobrado a mais para dada operação fica de "crédito" para "pagar" o custo de operações de maior custo. Assim, o custo amortizado de uma operação é seu custo real mais um crédito depositado para outras operações em um "banco", ou uma diferença "consumida" desse banco.

- **Método do potencial:** é semelhante ao método de contabilidade, pois também atribuímos um custo amortizado a cada operação. O custo a mais pago antecipadamente por uma dada operação é tratado como "energia potencial" ou apenas "potencial", que é associado ao algoritmo como um todo. Assim, o custo de toda operação individual será seu custo atribuído e a mudança no potencial que essa operação causa (na prática, temos um acréscimo do custo individual – se usarmos o potencial – ou uma redução – se a operação acrescentar ao potencial).

Exemplos de aplicação

Exercício 1. Vamos aplicar as diferentes metodologias de análise amortizada para a seguinte situação: considere uma pilha que, além da operação de inserção (push) e remoção (pop), tenha também uma terceira operação de remoção múltipla (multipop), onde serão removidos k elementos de uma só vez, deixando a pilha vazia caso k seja maior ou igual ao número de elementos presentes na pilha naquele momento.

Em uma análise superficial, como a operação de maior custo envolve, no pior caso, a totalidade dos n elementos presentes na pilha, podemos pensar que a complexidade de tempo desse algoritmo seja linear $O(n)$. Porém, ao aplicarmos a análise amortizada, obtemos o seguinte:

Análise agregada: a operação multipop terá um custo de execução maior, $O(n)$, que as demais operações, mas só poderá ser executada com esse custo após n inserções de custo constantes $O(1)$ terem sido realizadas. Assim, sendo $p(1)$ a operação push e $mp(n)$ a operação multipop, temos:

- Custo total = $p_1(1) + p_2(1) + \dots + p_n(1) + mp(n) = 2n$
- Custo por operação = $2n / (n+1) = 2$ (aproximadamente)

Assim, temos que o custo por operação na realidade continua constante, como em uma pilha normal.

Método da contabilidade: agora vamos atribuir um custo extra à operação de push: cada operação custará 2, para que cada operação de inserção acrescente "crédito ao banco". Se considerarmos um custo 0 para a operação multipop, isso significa que todo seu custo será pago pelo "crédito", na pior das hipóteses zerando-o todo:

- Custo total = $p_1(2) + p_2(2) + \dots + p_n(2) + mp(-n) = n$
- Custo por operação = $n / (n+1) = 2$ (aproximadamente)

Por esse método, temos novamente que o custo por operação da pilha é constante.

Método do potencial: novamente atribuímos os mesmos custos empregados no método da contabilidade. Porém, do custo 2 atribuído à operação push, um será o "custo real". Podemos notar que, a cada elemento inserido na pilha, o potencial acumulado nesse caso aumentará em 1:

- Custo total da inserção = $p_1(1) + p_2(1) + \dots + p_n(1) = n$
- Potencial acumulado = $p_1(1) + p_2(1) + \dots + p_n(1) = n$

Já a operação multipop, tendo custo 0, consumirá todo o potencial disponível. Assim, temos:

$$\begin{aligned}\text{Custo total} &= \text{custo real} + \text{acúmulo de potencial} + \text{consumo de potencial} = \\ &= n + n - n = n\end{aligned}$$

Novamente, um custo constante.

Assim, percebemos que a análise amortizada permite determinar um custo de tempo mais realista do que uma estimativa quando não houver função de complexidade de tempo bem determinada.

Exercício 2. (Enade 2017, adaptado) Busca primeiro em profundidade é um algoritmo de exploração sistemática em grafos cujas arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda tenha arestas inexploradas saindo dele. Exploradas todas as arestas de v , a busca regressa para explorar as arestas que deixam o vértice a partir do qual v foi descoberto. O processo continua até que todos os vértices acessíveis a partir do vértice inicial sejam explorados.

Fonte: CORMEN, T. H. et al. *Introduction to algorithms*. 3. ed. Cambridge, MA: The MIT Press, 2009.

Considere o grafo a seguir:

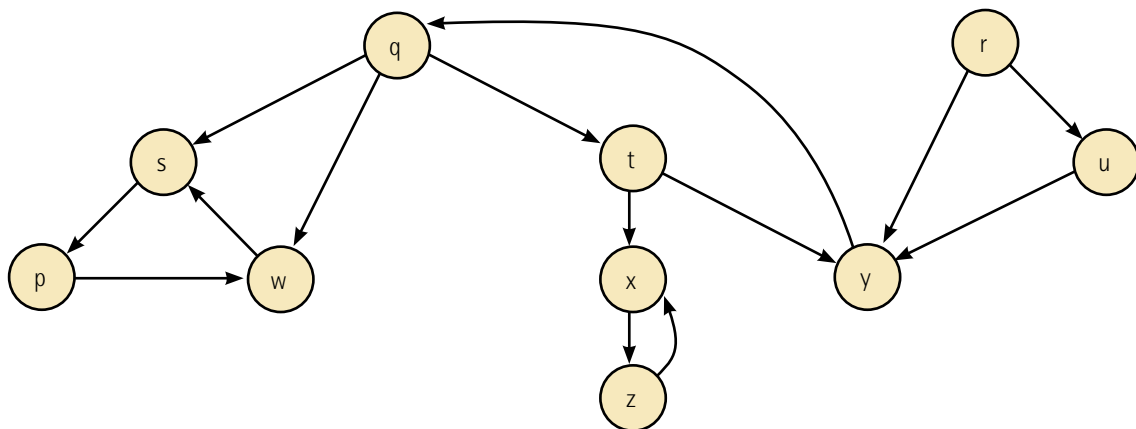


Figura 40

Com base nas informações apresentadas, faça o que se pede:

A) Mostre a sequência de vértices descobertos no grafo durante uma busca em profundidade com controle de estados repetidos; para isso, utilize o vértice r como inicial. Se um vértice explorado tiver mais de um vértice adjacente, utilize a ordem alfabética crescente como critério para priorizar a exploração.

Partindo do vértice r como raiz, temos duas ramificações possíveis: u e v. Pelo critério alfabético, seguimos para u, depois o único caminho possível será y e depois q.

A partir de q, temos a sequência s, p e w. Agora temos o primeiro ponto, onde não é possível seguir adiante. Assim, retrocede-se para q, de onde se segue para o último ramo t, x e z. O trajeto completo será:

r – u – y – q – s – p – w – t – y – z

B) Represente a matriz de adjacências desse grafo, podendo omitir os zeros nessa matriz.

Indicando os vértices em ordem alfabética e representando a matriz em forma de tabela para visualizá-la melhor, temos:

	p	q	r	s	t	u	w	x	y	z
p							1			
q				1	1		1			
r						1			1	
s	1									
t								1	1	
u									1	
w			1							
x										1
y		1								
z								1		

Como se trata de um grafo direcionado, lembremos que a matriz de adjacência não é simétrica.

Exercício 3. (Enade 2014, adaptado) Uma fazenda possui um único poço artesiano que deve abastecer n bebedouros para o gado. Deseja-se determinar um projeto de ligação para esses n+1 pontos através de encanamentos com a menor extensão total.

Um algoritmo proposto para resolver o problema segue estes passos:

1) Crie $n+1$ conjuntos unitários, cada um com um dos pontos a ser ligados entre si, e insira esses conjuntos em um conjunto C.

2) Crie um conjunto D com um registro para cada combinação possível de dois pontos distintos a ser ligados. Cada registro deve conter os campos c_i , c_j e d , em que c_i e c_j são os dois pontos a ser ligados e d é a distância entre eles.

3) Enquanto D não estiver vazio, faça o seguinte:

3.1) Remova o registro de D com o menor valor de distância d .

3.2) Se os valores c_i e c_j do registro removido pertencerem a conjuntos distintos de C:

3.2.1) Substitua esses dois conjuntos pela união entre eles.

3.2.2) Guarde o registro removido em conjunto-solução.

Com base no problema, o que podemos concluir sobre o algoritmo proposto?

Temos aqui um grafo onde os vértices são os bebedouros e o poço artesiano:

- O conjunto C é um conjunto composto por todos estes vértices.
- O conjunto D é composto pelas arestas do grafo, com seus respectivos pesos (no caso, a distância).

Dessa forma, a instrução 3 do algoritmo consiste em selecionar a aresta de menor peso e, caso ela conecte dois vértices que estejam em subgrafos distintos, será incluída como parte da solução do problema; do contrário, será desconsiderada.

Temos aqui uma aplicação prática do algoritmo de Kruskal para construir uma árvore geradora mínima. A complexidade de tempo desse algoritmo será o número de elementos de C multiplicado pelo número de elementos de D: pelo proposto na instrução 2 do algoritmo, todas as arestas possíveis do algoritmo estarão presentes em D, e assim partimos de um grafo completo. Portanto teremos uma complexidade de tempo polinomial $O(c \times d)$, sendo c e d os números de elementos dos conjuntos C e D, respectivamente.

Uma forma interessante de reduzir esse algoritmo seria aplicando o algoritmo de Dijkstra (abordado no tópico 3) a partir do poço artesiano, para reduzir os elementos de D (arestas), e só então aplicar o algoritmo proposto. Por ser um grafo completo, onde se conhece o peso de todas as arestas, uma solução mais elaborada – mas com resultado mais rápido – seria aplicar um método de busca heurística (assunto tratado no tópico 6).

6 ALGORITMOS GULOSOS

Algoritmos gulosos executam, a cada iteração, uma tomada de decisão tentando prever o melhor resultado para as próximas iterações.

6.1 Problemas de otimização

Em algoritmos, são muito comuns os problemas de otimização, nos quais buscamos obter o resultado máximo ou mínimo de algo em dada situação. O problema da multiplicação de matrizes (ver tópico 2) é um exemplo, assim como muitos problemas em encontrar trajetos em grafos (ver tópico 5).

Os chamados algoritmos gulosos realizam tomadas de decisão 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 considerar as ramificações decorrentes dessa decisão, nem registrar as operações anteriores para o caso de serem desfeitas ou reconsideradas (Dasgupta; Papadimitriou; Vazirani, 2009, p. 127).

6.2 Algoritmos gulosos

Devido às características apresentadas, é fácil criar e implementar algoritmos gulosos (que é a tradução mais empregada de *greedy algorithms*; uma tradução mais literal seria "algoritmos gananciosos"). Muitos métodos de busca utilizados em IA são algoritmos gulosos (trataremos desse assunto mais adiante). 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 (ver tópico 8).

Algoritmos gulosos possuem semelhanças com a programação dinâmica vista no tópico 2. Ambas as técnicas trabalham com problemas de otimização e 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. Em resumo, na programação dinâmica parte-se de subproblemas menores para maiores; e os algoritmos gulosos avançam de cima para baixo (top-down) visando reduzir o problema (pelo menos, esse é o objetivo).

Podemos, de modo geral, dividir um algoritmo guloso em cinco componentes:

- Um conjunto candidato de próximas etapas, a partir do qual é criada uma solução.
- Uma função de seleção, cujo objetivo é selecionar a melhor candidata a ser adicionada à solução.
- Uma segunda função, chamada de função de viabilidade, utilizada para definir se um candidato pode contribuir ou não para uma solução definitiva.

- Uma função objetivo, que atribui um valor a uma solução (parcial ou completa), de acordo com o objetivo do algoritmo.
- Uma função de solução, que indicará quando o algoritmo tiver encontrado uma solução completa para o problema.

Obviamente, nem todos os algoritmos gulosos apresentam essa divisão formal em etapas bem definidas. Na verdade, nos tópicos anteriores já vimos alguns algoritmos que podem ser considerados gulosos. Veja alguns exemplos:

- **Caminho de peso mínimo:** envolve identificar o menor trajeto em um grafo que possua arestas com pesos. Um exemplo é o algoritmo de Dijkstra (ver tópico 3).
- **Árvore geradora de peso mínimo em um grafo:** encontrar a árvore geradora mínima (que contenha todos os nós de um grafo) em um grafo com pesos nas arestas. Um exemplo é o algoritmo de Kruskal (ver tópico 5).
- **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 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.
- **Problema da mochila fracionária:** deseja-se preencher uma mochila que consiga carregar determinado peso com objetos com determinados pesos e valores diferentes. O objetivo é colocar e maximizar o valor dos objetos na mochila (esse problema será detalhado no tópico 8).

A principal característica de um algoritmo guloso é que ele encontra uma solução otimizada para um problema, mas não necessariamente a melhor solução possível (ótimo global). Isso se deve à característica de que, por serem algoritmos iterativos, eles tentam, a cada iteração, obter a melhor solução parcial possível, considerando a solução final. Assim, se houver mais de uma solução possível, pode ser que a primeira não seja a mais otimizada.



Observação

Uma solução não ótima significa uma solução para um problema que pode não ser a melhor possível, mas boa o bastante para ser utilizada e implementada em situações práticas.

Exemplo de aplicação

(Enade 2017, adaptado). Um país utiliza moedas de 1, 5, 10, 25 e 50 centavos. Um programador desenvolveu o método a seguir, que implementa a estratégia gulosa para o problema do troco mínimo. Esse método recebe como parâmetro um valor inteiro, em centavos, e retorna um array no qual cada posição indica a quantidade de moedas de cada valor:

```
public static int[] troco (int valor){
    int [] moedas = new int [5];
    moedas[4] = valor / 50;
    valor = valor % 50;
    moedas[3] = valor / 25;
    valor = valor % 25;
    moedas[2] = valor / 10;
    valor = valor % 10;
    moedas[1] = valor / 5;
    valor = valor % 5;
    moedas[0] = valor;
    return (moedas);
}
```

Considerando o método apresentado, avalie as asserções a seguir e a relação proposta entre elas:

I – O método guloso encontra o menor número de moedas para o valor de entrada, considerando as moedas do país.

porque

II – Métodos gulosos sempre encontram uma solução global ótima.

O algoritmo realmente é guloso: começamos reduzindo o valor ao máximo ao retirarmos dele primeiro as moedas de maior valor e maximizando essa redução a cada passagem. Por se tratar de um algoritmo pequeno, as iterações são executadas individualmente e não dentro de uma estrutura de repetição, mas isso não modifica a ideia central por trás do algoritmo. Assim, a afirmação I é **verdadeira**.

A afirmação II apresenta um erro conceitual acerca dos algoritmos gulosos: eles nem sempre encontram uma solução ótima global. Assim, a afirmação II é **falsa** e, consequentemente, não é uma justificativa válida para a primeira afirmação.

6.3 Métodos de busca

As buscas em largura e profundidade são baseadas no trajeto e na estrutura de dados. Elas não levam em consideração nenhum critério referente ao que está sendo buscado. Assim, muitas vezes essas buscas são denominadas **buscas cegas**. Por outro lado, existem métodos de busca que se utilizam de **heurísticas** para otimizar o processo de busca.



Lembrete

Na ciência da computação, heurística é um método de investigação baseado na aproximação progressiva da solução de um problema.

Cabe fazermos uma distinção entre os **algoritmos de busca** que vimos, que têm como objetivo identificar a existência ou não de um elemento em uma estrutura de dados, e **métodos de busca**. Um método de busca é uma metodologia de IA que utiliza 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 a um **estado final**, da forma mais otimizada possível (Russell; Norvig, 2022, p. 57).

Um exemplo seria o algoritmo de Dijkstra (ver tópico 3): cada vértice do grafo seria um estado possível do espaço de estados. Alguém percorrendo um grafo só poderia ir a um vértice de cada vez e, saindo de um vértice, só poderia se dirigir para outro vértice adjacente de cada vez. Um problema utilizando um método de busca seria encontrar o trajeto ótimo (com menor peso das arestas) entre dois vértices selecionados (que seriam o estado inicial e o estado final do sistema).

Um método de busca guloso conterà os elementos descritos no início deste tópico. Existem diversas heurísticas, muitas delas bastante sofisticadas para a execução desses métodos. Vamos abordar os casos mais simples:

- **Busca gulosa pela melhor escolha:** nesse método de busca, faz-se necessário o conhecimento de quão perto está determinado estado de um estado final desejado. A partir disso decide-se, entre os estados que podem ser atingidos a partir do estado atual, qual está mais próximo do estado final desejado.

Por exemplo, um algoritmo para resolver o cubo de Rubik (também conhecido no Brasil como cubo mágico): podemos atribuir um valor ao estado final "cubo resolvido", composto por uma somatória dos estados de cada uma das seis faces. Nos estados de "não resolução", cada face terá uma função atribuída a ela de acordo com a disposição dos quadrados que a compõem. Uma busca gulosa por melhor escolha decidirá o próximo movimento a ser realizado com base em qual deles geraria um conjunto de estados das faces mais próximo do estado final.



Figura 41 – Cubo de Rubik

Disponível em: <https://tinyurl.com/wnbtb399>. Acesso em: 6 dez. 2023.

Exemplo de aplicação

Um exemplo clássico e amplamente reproduzido é apresentado por Russell e Norvig no livro *Inteligência artificial: uma abordagem moderna* (2022, p. 58). Imagine que se deseje traçar uma rota entre as cidades de Arad e Bucareste, na Romênia. Sabe-se a distância por estrada entre diferentes cidades importantes na região, conforme mostrado na imagem a seguir:

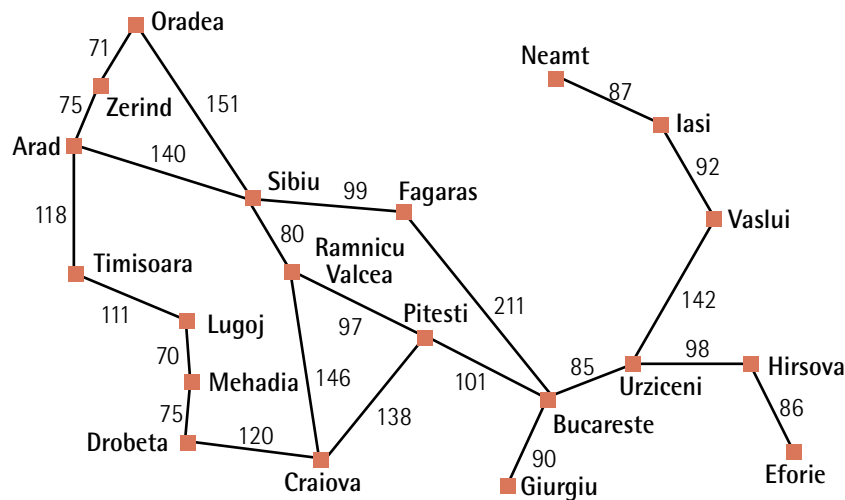


Figura 42

Fonte: Russell e Norvig (2022, p. 58).

Apenas com essa informação, seria uma resolução por meio do algoritmo de Dijkstra (ver tópico 3) para encontrar a menor distância entre os dois pontos. Porém, com a quantidade de cidades presentes no mapa, teríamos um algoritmo com vinte iterações (uma por vértice), com até quatro distâncias sendo calculadas e priorizadas em cada uma dessas iterações.

Agora, vamos acrescentar uma heurística neste problema: o conhecimento das distâncias em linha reta (não por estrada) de cada uma das cidades até Bucareste:

Tabela 4

Arad	366	Mehadia	241
Bucareste	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Ramnicu Valcea	193
Fagaras	176	Sibiu	253
Guirgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Fonte: Russell e Norvig (2022, p. 77).

Um método de busca guloso pela melhor escolha preferirá, a partir da cidade atual, mover-se para a cidade conectada de menor distância para Bucareste. Assim, partindo de Arad, temos três opções:

- **Sibiu:** 253 km de Bucareste.
- **Timisoara:** 329 km de Bucareste.
- **Zerind:** 374 km de Bucareste.

Um algoritmo guloso escolherá a cidade mais perto do destino final. Assim, o próximo estado será a cidade de Sibiu. A partir dessa cidade, temos as seguintes opções (já descartando a opção de voltar para Arad):

- **Fagaras:** 176 km de Bucareste.
- **Oradea:** 380 km de Bucareste.
- **Ramnicu Valcea:** 193 km de Bucareste.

Novamente pelo critério da menor distância linear, selecionamos Fagaras. Como de Fagaras temos apenas duas opções, retornar para Sibiu ou ir para Bucareste, que é o estado final, temos a seguinte rota:

Arad – Sibiu – Fagaras – Bucareste

Perceba que não houve necessidade aqui de explorar a totalidade do grafo, que seria o caso se aplicássemos o algoritmo de Dijkstra. Porém, embora mais rápido e fornecendo uma solução para o problema, **esta não é a solução ótima**: a rota **Arad – Sibiu – Ramnicu Valcea – Pitesti – Bucareste** é 32 km mais curta do que a rota obtida.

O exemplo ilustrou uma característica importante dos algoritmos gulosos: a solução encontrada nem sempre é a solução ótima. O código a seguir em Python implementa um algoritmo de busca gulosa por rota em um grafo, como no problema. Para evitar que o algoritmo ande em círculos, é implementado um backtracking: caso o vértice selecionado pelo algoritmo guloso leve a um beco sem saída (não existem arestas saindo dele) ou repetido, ele remove-o das estruturas a serem futuramente testadas, junto com suas ramificações, sem examiná-los.

```
def guloso(grafo, inicio, destino):  
    # Inicia a lista com os nós já visitados e a lista com o caminho  
    visitados = [False] * len(grafo)  
    caminho = [inicio]  
    while caminho[-1] != destino:  
        atual = caminho[-1]  
        vizinhos = [(vizinho, custo) for vizinho, custo in enumerate(grafo[atual])  
if not visitados[vizinho] and custo > 0]  
        if not vizinhos:
```

```
# Se não há vizinhos disponíveis, backtracking
caminho.pop()
visitados[atual] = False
else:
    # Escolhe o vizinho com menor custo
    vizinho_escolhido, custo_escolhido = min(vizinhos, key=lambda x: x[1])
    visitados[vizinho_escolhido] = True
    caminho.append(vizinho_escolhido)
return caminho
```

6.4 Código de Huffman

Este algoritmo tem uma história que merece ser contada. Em 1951, na disciplina de Teoria da Informação do Massachusetts Institute of Technology (MIT), o professor Robert M. Fano ofereceu duas opções a seus alunos: escrever um trabalho ou fazer um exame final. O tema do trabalho era encontrar a codificação binária mais eficiente. Um dos alunos, David A. Huffman (1925-1999), não conseguindo demonstrar que nenhum código era realmente a codificação mais eficiente, estava se preparando para realizar a prova, quando teve a ideia de criar um código que se baseava em uma árvore binária construída a partir das frequências relativas dos símbolos na mensagem que se desejava codificar.

Até então, o código binário mais eficiente era o código de Shannon-Fano, criado pelo seu professor em conjunto com Claude Shannon, e se baseava em uma metodologia bottom-up, ao contrário da abordagem top-down de codificação de Shannon-Fano.



Saiba mais

Claude Elwood Shannon (1916-2001), autor junto com Fano, da codificação anterior, foi matemático, engenheiro eletrônico e criptógrafo americano, conhecido como "o pai da teoria da informação".

O código de Huffman é uma das melhores técnicas de compressão de dados (Cormen, 2012, p. 314). 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; para efeitos deste livro-texto, vamos chamar de **símbolo** cada componente individual do texto). A ideia básica do algoritmo é muito simples: atribuem-se códigos com menores 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**.

Para entendermos o que é prefixação e por que ela não está presente no código de Huffman, vejamos um exemplo: vamos supor que desejemos codificar um texto composto pelos símbolos A, B, C e D. Arbitrariamente, vamos atribuir o seguinte código binário: A = 0, B = 1, C = 10 e D = 11. Esse código arbitrário tem um problema: por exemplo, a mensagem 1110 significa BBC, DBA, BDA ou DC? Assim, se utilizarmos esse código, não podemos agrupar diretamente os bits de forma sequencial; teremos que colocar algum tipo de **prefixação** para que se possa distinguir os caracteres individualmente.

Para o algoritmo de codificação de Huffman é necessário saber 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, mas isso não afeta a eficiência do código final).

5) O código de cada folha da árvore será a sequência dos caracteres das arestas da raiz até ela.

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 de aplicação

Vamos aplicar o código de Huffman, em seu no texto "Analisando seriamente os algoritmos". Em primeiro lugar, precisamos obter o número de ocorrências de cada símbolo:

Tabela 5

Símbolo	Ocorrências
A	5
O	4
S	4
Es (espaço)	4
I	3
N	3
E	3
L	2
M	2
R	2
D	1
G	1
T	1

Para o algoritmo de Huffman, podemos utilizar o número de ocorrências no lugar da frequência. Podemos ver que o símbolo mais frequente aparece seis vezes mais que os símbolos menos frequentes.

Vamos agora organizar isso na forma de um conjunto de árvores de um único nó (os valores entre parênteses indicam o número de ocorrências, para facilitar a visualização):

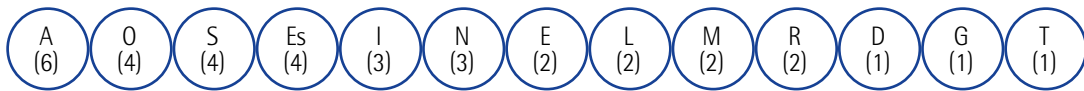


Figura 43

Os dois nós de menor número de ocorrência são agrupados (no caso de valores iguais, dois deles são escolhidos aleatoriamente). Utilizaremos um quadrado para ilustrar a raiz de uma árvore composta:

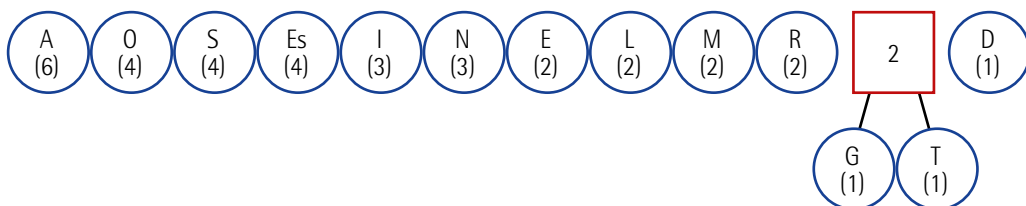


Figura 44

O processo se repete, sempre juntando as árvores que contenham menor valor na raiz:

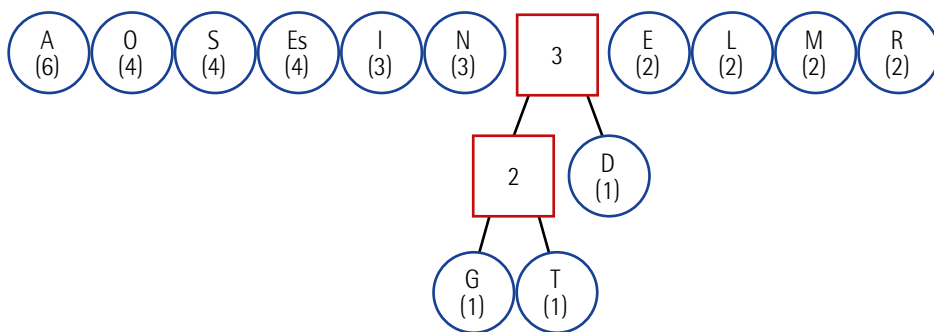


Figura 45

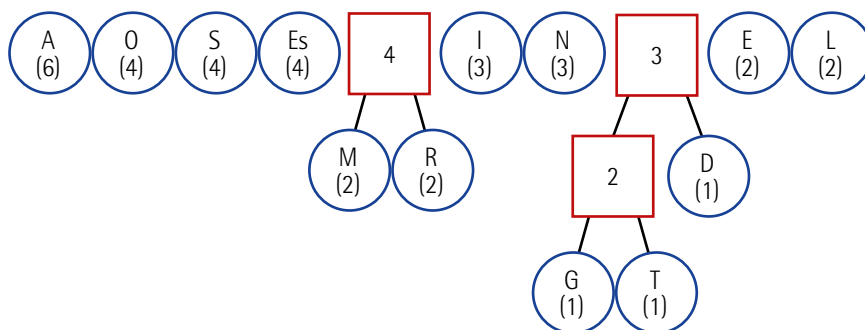


Figura 46

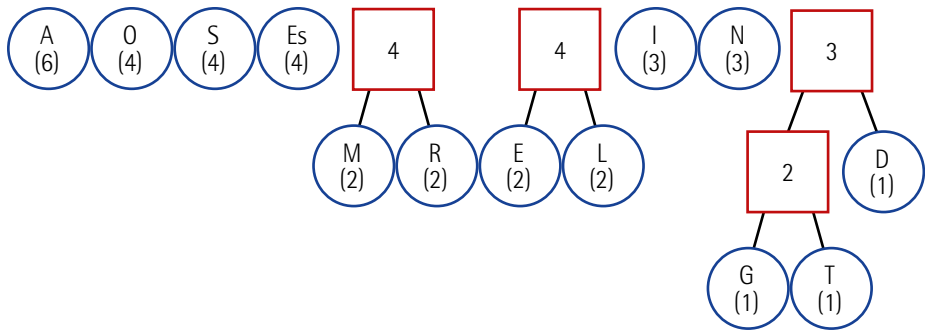


Figura 47

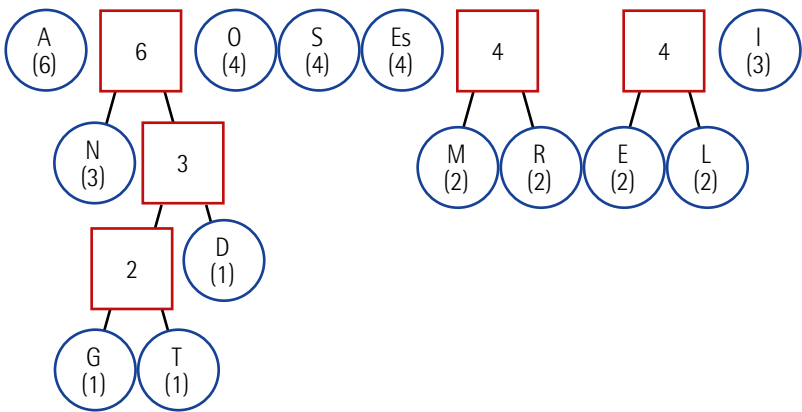


Figura 48

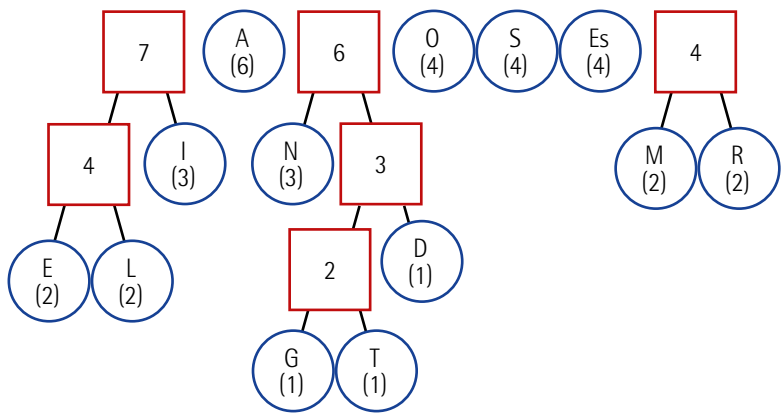


Figura 49

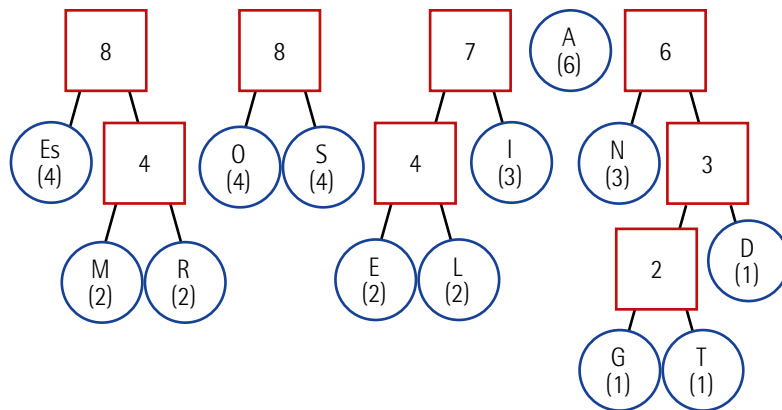


Figura 50

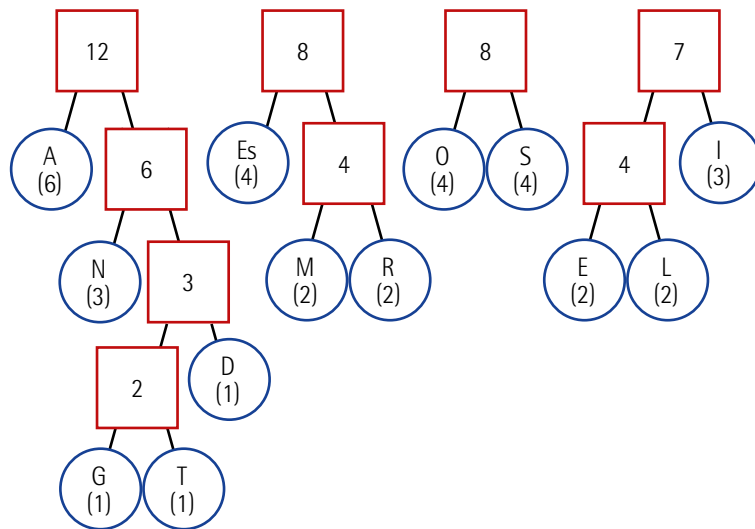


Figura 51

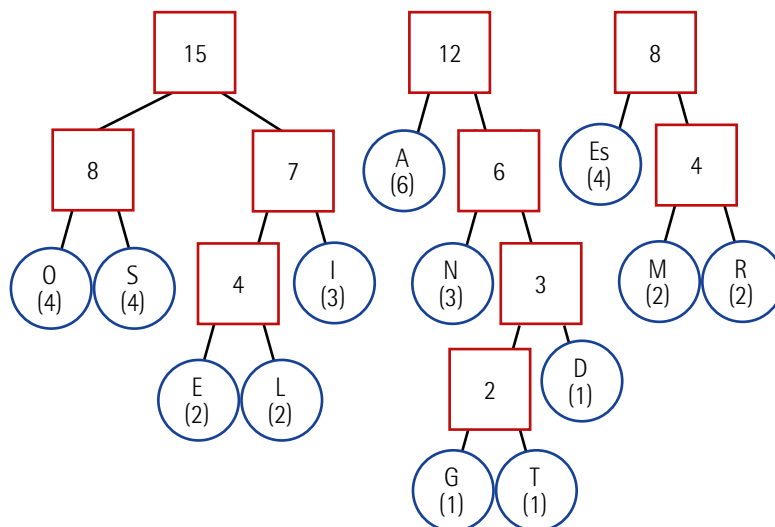


Figura 52

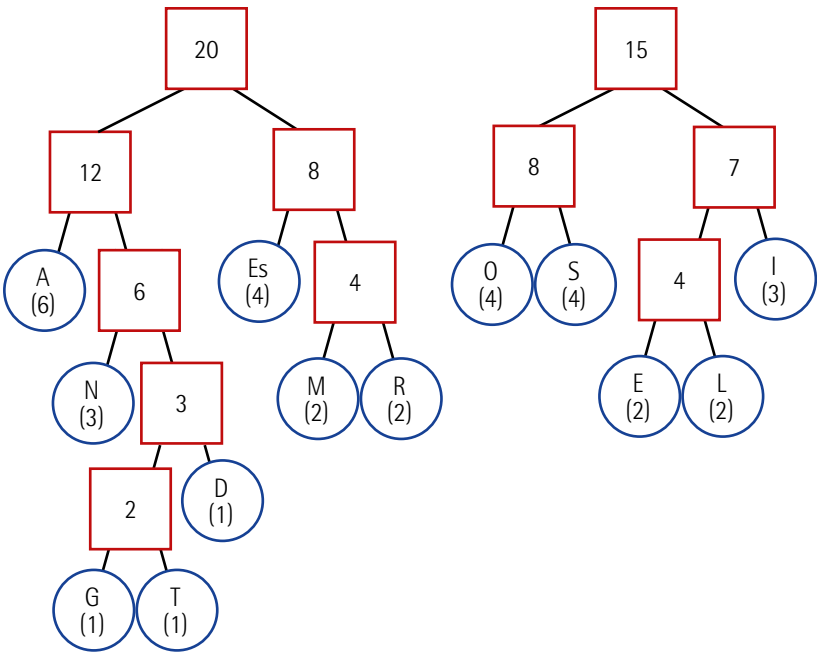


Figura 53

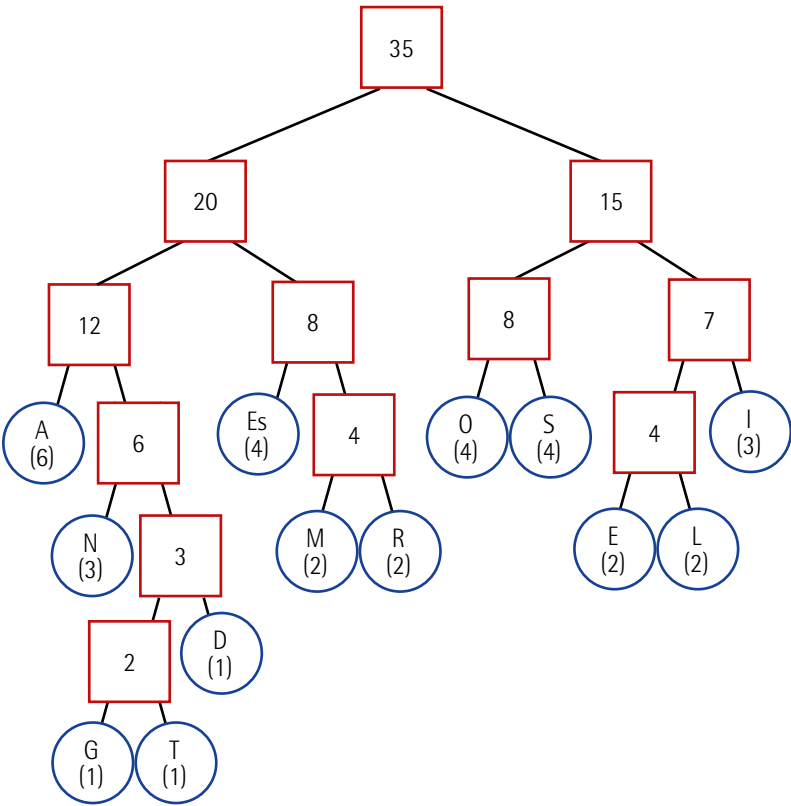


Figura 54

Construída a árvore, a última etapa é atribuir os valores aos ramos: 0 aos ramos da esquerda e 1 aos ramos da direita.

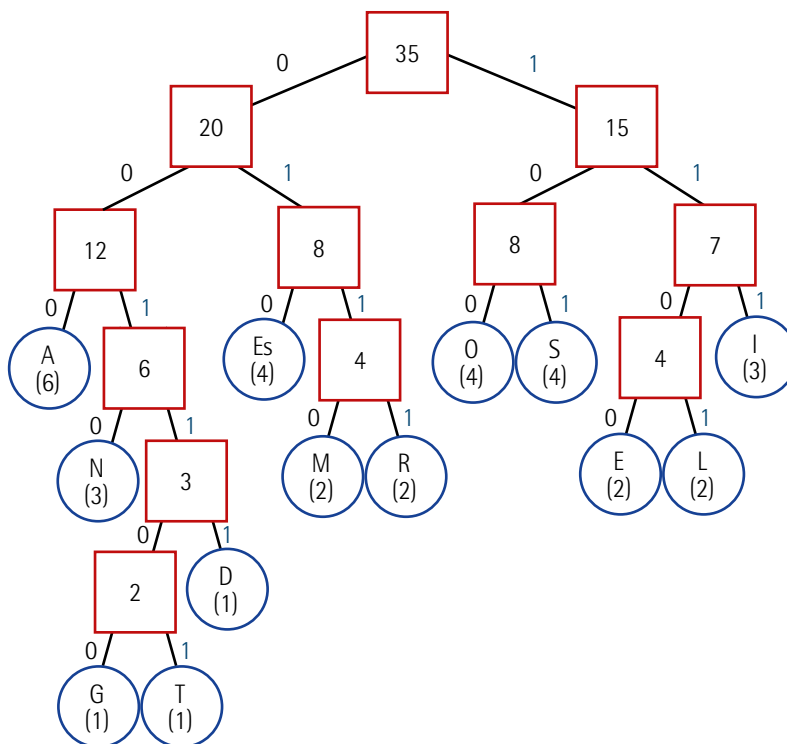


Figura 55

O código de cada símbolo será a sequência de bits que será percorrida partindo da raiz até chegar à folha correspondente:

Tabela 6

Símbolo	Ocorrências	Código de Huffman
A	5	000
O	4	100
S	4	101
Es (espaço)	4	010
I	3	111
N	3	0010
E	3	1100
L	2	1101
M	2	0110
R	2	0111
D	1	00111
G	1	001100
T	1	001101

É possível perceber que não existe necessidade de prefixação neste código: por exemplo, se pegarmos a sequência 01110000110100, a única decodificação possível seria a palavra RAMO, sem a possibilidade de ambiguidades.

A versão original desse algoritmo desenvolvida por Huffman tem uma complexidade de tempo $O(n \log n)$, onde n é o número de símbolos no texto. Atualmente, existem variações com complexidade de tempo $O(n)$, mas essas variações precisam que as frequências dos símbolos sejam conhecidas e estejam previamente ordenadas.

É possível perceber que a codificação de Huffman é única para cada texto, pois, como foi já dito, leva em conta as frequências dos símbolos que compõem o texto, algo único para cada um. Essa codificação é base para os algoritmos de compressão modernos: uma codificação binária 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 (por exemplo, os códigos ASCII ou Unicode).



Observação

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.

O programa em Python a seguir recebe um texto como entrada e devolve como saída um dicionário com o código de Huffman para cada caractere presente no texto. O programa se utiliza de métodos para a construção de heaps (ver tópico 3) contidos no módulo `heapq` para auxiliar na construção da árvore de Huffman.

```
import heapq
from collections import defaultdict

#Cálculo das frequências dos caracteres
#A lista é indexada a partir do caractere
def calcular_frequencias(frase):
    frequencias = defaultdict(int)
    for caractere in frase:
        frequencias[caractere] += 1
    return frequencias

#Construção da Árvore de Huffman
#Métodos para heaps são utilizados para otimizar o código
def construir_arvore(frequencias):
    heap = [[frequencia, [caractere, ""]] for caractere, frequencia in
    frequencias.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
```

```
for par in lo[1:]:
    par[1] = '0' + par[1]
for par in hi[1:]:
    par[1] = '1' + par[1]
heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
return heap[0][1:]
```

#Obtenção do código a partir da árvore

```
def obter_codigos(arvore_huffman):
    codigos = {caractere: codigo for caractere, codigo in arvore_huffman}
    return codigos
```

#Programa Principal

```
frase = input("Digite a frase: ")
frequencias = calcular_frequencias(frase)
arvore_huffman = construir_arvore(frequencias)
codigos_huffman = obter_codigos(arvore_huffman)
for i in codigos_huffman:
    print(i, ": ", codigos_huffman[i])
```



Lembrete

Dicionário é uma estrutura de dados composta do Python na qual elementos são indexados por meio de uma **chave** (como em um banco de dados), e não por posição.

6.5 Estatística por compressão

Um efeito colateral do código de Huffman é que é possível fazer algumas inferências estatísticas sobre a natureza dos dados que foram comprimidos por meio dessa codificação, apenas a partir do resultado desse processo de compressão. Vamos analisar um curioso exemplo extraído do livro *Algoritmos*, de Dasgupta, Papadimitriou e Vazirani (2009, p. 143):

A corrida de cavalos anual regional está trazendo três novos puros-sangues que nunca competiram um contra o outro. Animado, você estuda suas últimas 200 corridas e as resume como uma distribuição de probabilidades sobre quatro eventos: primeiro ("primeiro lugar"), segundo, terceiro e outros.

Evento	Aurora	Turbilhão	Fantasma
Primeiro	0,10	0,30	0,20
Segundo	0,15	0,05	0,30
Terceiro	0,70	0,25	0,30
Outros	0,05	0,40	0,20

Qual cavalo é o mais previsível? Uma abordagem quantitativa a esta questão é olhar a compressibilidade. Escreva o histórico de cada cavalo como uma string de 200 valores (primeiro, segundo, terceiro, outros).

Se considerarmos uma cadeia de comprimento 200, com os caracteres P, S, T e O representando os resultados, cada um ocorrendo com a frequência indicada, teremos os seguintes códigos de Huffman para cada um dos três cavalos:

- **Aurora**

Código de Huffman: P: 110, S: 10, T: 0, O: 111

- **Turbilhão**

Código de Huffman: P: 10, S: 110, T: 111, O: 0

- **Fantasma**

Código de Huffman: P: 10, S: 00, T: 01, O: 11

O total de caracteres em cadeia codificada será seu tamanho (200) multiplicado pela somatória dos produtos da frequência de cada caractere por seu número de bits correspondentes:

- **Aurora**

Tamanho da cadeia codificada = $200 \times (0,10 \times 3 + 0,15 \times 2 + 0,70 \times 1 + 0,05 \times 3) = 290$ bits

- **Turbilhão**

Tamanho da cadeia codificada = $200 \times (0,30 \times 2 + 0,05 \times 3 + 0,25 \times 3 + 0,40 \times 1) = 380$ bits

- **Fantasma**

Tamanho da cadeia codificada = $200 \times (0,20 \times 2 + 0,30 \times 2 + 0,30 \times 2 + 0,20 \times 2) = 400$ bits

A imprevisibilidade presente em uma distribuição de probabilidades, ou seja, sua aleatoriedade, pode ser avaliada por quanto é possível comprimir, via código de Huffman: quanto mais previsível, menor será a aleatoriedade e maior será a compressão possível (resultando em um menor número de bits). Assim, Aurora é o mais previsível dos cavalos, enquanto Fantasma é o menos previsível.

7 ALGORITMOS MULTITHREAD

Neste tópico, vamos tratar de algoritmos para processamento paralelo, nos quais diversas instruções que compõem um mesmo conjunto podem ser executadas de forma concorrente e simultânea.

7.1 Processamento paralelo e threads

Todos os algoritmos e programas apresentados até o momento neste livro-texto foram sequenciais. Isso significa que eles foram pensados para ter 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 (muitas vezes, nesses casos, em locais diferentes, fisicamente). Assim, acompanhando o desenvolvimento do hardware, surgiu o conceito de **processamento paralelo**.



Observação

Existe controvérsia sobre qual o primeiro chip com processamento paralelo para PCs. Muitos consideram que foi o Intel 80486, lançado em 1989, que integrava as unidades de processamento principal e aritmética.

O processamento paralelo levou ao surgimento do conceito de multithreading, ou seja, o processamento simultâneo de múltiplos threads. Um thread pode ser definido, de forma resumida, como 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.

Um processo é uma aplicação que está sendo executada pelo sistema. Cada processo tem, como padrão, sua própria alocação de recursos, mais especificamente de memória, que não pode ser compartilhada com outros processos. Já um thread, como foi dito, compartilha recursos, o que torna mais fácil a comunicação entre eles.

A forma mais comum de programar usando arquitetura paralela de computadores é por meio do thread estático. Isso significa que, mesmo em programa ou algoritmo sequencial, podemos construir funções (e classes e métodos) 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 (frequentemente chamado de destruição) dos threads é lento, muitas vezes os threads criados permanecem na memória; daí a denominação threads estáticos (Cormen, 2012, p. 560).



Observação

A criação e a destruição de threads são feitas 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 protocolos adequados de comunicação entre eles envolve acréscimo no trabalho e no código, para garantir o uso eficiente e correto dos recursos computacionais. Assim, muito mais utilizado é o conceito de thread dinâmico.

7.2 Multithread dinâmico

Um **multithread dinâmico** é um modelo cuja 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:** possibilita gerar um novo thread, de modo que o ativador dessa geração continue executando seu processamento enquanto o 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).

Para entender melhor esse conceito, vamos trazer de volta o problema para calcular os n primeiros valores da sequência de Fibonacci, conforme visto no tópico 2. Inicialmente, vamos considerar a mesma função recursiva:

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

Como vimos, essa função chamará a si mesma diversas vezes. Além disso, existe outro problema: ao iniciarmos o cálculo do número seguinte na sequência, ela ignora todos os esforços realizados nos cálculos anteriores, recomendo do zero o processo. A figura 56 ilustra quantas vezes a função será executada para calcular o sexto elemento da sequência:

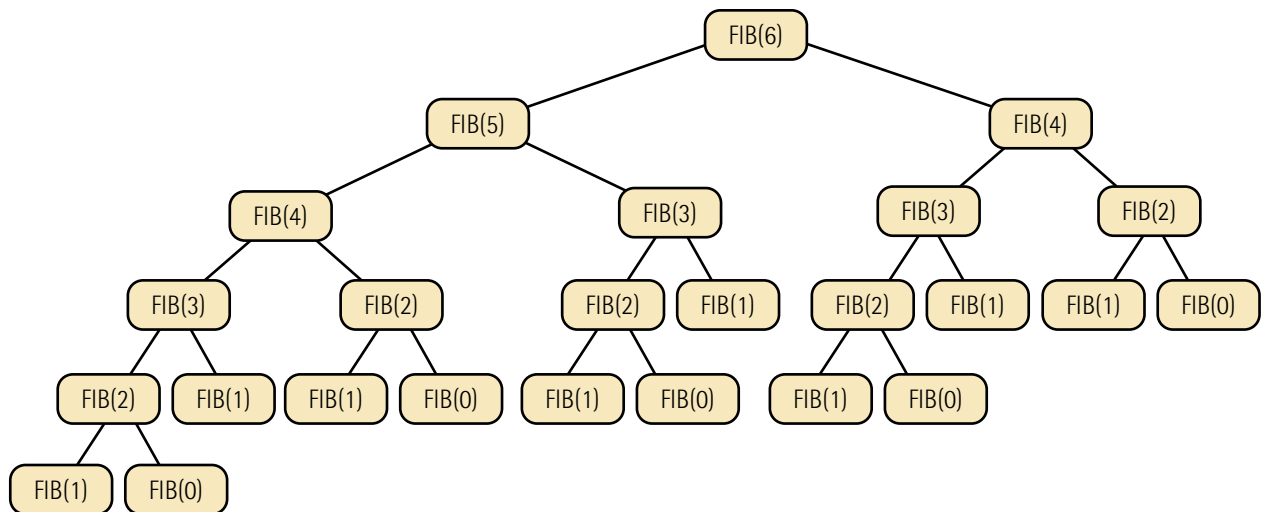


Figura 56 – Execuções da função para n = 6

Fonte: Cormen (2012, p. 562).

Agora vamos pensar em 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 fib_recursiva (5) e precisarmos, pela recursividade, de fib_recursiva (4) e fib_recursiva (3), estas já estarão calculadas e poderão ser empregadas. Assim, transformando nosso exemplo em um programa multithread:

```

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 )
    
```




Observação

A maioria das linguagens de programação modernas contém classes e módulos para a programação multithread. No Python, esses métodos estão contidos no módulo `threading`.

A classe **Thread** (do módulo `threading`) transforma a execução de uma função em um thread. No caso, trata-se da função `Fib2`. Perceba que os argumentos do thread não são os parâmetros da função: além de um valor numérico, o thread recebe como parâmetro uma lista, que é uma variável global. Essa classe representa um recurso comum a todas as linguagens que fazem uso de multithread, que é a criação do thread (muitas vezes esse recurso é identificado por `spawn`) (Multiprocessing..., 2008).

O método `start()` inicia a execução do thread. Assim, a função `Fib2` começará a ser executada em paralelo com o restante do programa. Por fim, o método `join()` bloqueia o programa principal até que o thread termine sua execução. Isso significa que o programa principal esperará que `thread1` conclua antes de continuar. Muitas vezes, esse método é denominado `sync`, pois faz a sincronização entre os diferentes threads e o tempo do programa principal que os contém.

Nesse exemplo, o paralelismo aninhado ocorre quando o thread é criado: como estamos lidando com uma função recursiva, o thread-pai (aquele que instancia a função e cria outro thread) não precisa esperar pelo término da execução do thread-filho para continuar sua execução. Cabe ao computador determinar o que é executado paralelamente e por quais processadores, conforme transcorre o programa como um todo. Não entraremos aqui neste livro-texto no assunto de arquitetura de computadores e processadores, que é o que define como é feito esse escalonamento dos diferentes threads (apenas vamos assumir que os computadores atuais são capazes de fazê-lo).

7.3 Desempenho do multithread

A eficiência de um algoritmo multithread pode ser medida de duas formas:

- **Trabalho computacional:** é o tempo total para executar a computação inteira. Como temos elementos sendo executados simultaneamente, o trabalho computacional na realidade é menor que a soma dos tempos individuais de execução dos threads.
- **Duração:** é o tempo mais longo consumido para executar um conjunto de threads, em qualquer ponto durante a execução do programa como um todo. No exemplo dado, seria o tempo de execução de todos os threads, pois, para o *n*-ésimo thread, todos deverão ser executados para obter o resultado.

O tempo de execução de uma computação multithread depende não apenas dessas duas medidas, mas também da alocação de processadores, memória e outros recursos. Assim, podemos indicar o tempo de execução de um programa cujo trabalho computacional é T e realizado por P processadores por TP (atente que isto não é um produto). Se considerarmos T_1 o trabalho de um único processador, temos que um computador paralelo pode realizar no máximo P unidades de trabalho. Dessa forma, para um tempo TP , o trabalho máximo possível é $P \cdot TP$. Dividindo por P obtemos a lei do trabalho: $TP \geq T_1/P$.

7.4 Escalonamento

Quando dois threads originários se alternam em relação ao uso de recursos, ocorre o que chamamos de troca de contexto. Nessa situação, os dados associados ao thread que será paralisado (variáveis, estruturas de dados etc.) precisam ser salvos. Essa troca de contexto pode ser **parcial**, quando dois threads originários de um mesmo processo, que já estavam em execução, se revezam no uso do processador, ou **completa**, quando um thread de outro processo se inicia e assume o uso do processador.

Quando temos vários threads operando em paralelo e há uma concorrência pelos recursos computacionais, torna-se necessário definir uma prioridade, ou uma liderança, nas alocações desses recursos.

Em um ambiente multithread, temos diversos threading competindo pelo processador ao mesmo tempo. Quando mais de um thread está pronto (preparado para ser executado) e temos apenas um processador disponível, o sistema computacional deverá decidir qual deles executará primeiro. Essa parte do sistema que faz a escolha é denominada **escalonador**.

Uma solução proposta por Dijkstra (o mesmo autor do algoritmo do caminho em grafos, citado muitas vezes ao longo deste livro-texto) em 1965 é a ideia do semáforo (Tanenbaum; Woodhull, 2008, p. 87). Em cima dessa variável, são executadas duas operações, apresentadas a seguir de forma muito simplificada:

- **Down:** significa que temos um thread em andamento, ou talvez uma fila de threads aguardando para ser executados. Nesse caso, o novo thread é posto em espera e a variável associada ao semáforo é decrementada.
- **Up:** a variável é incrementada e, caso haja threads parados resultantes de uma operação down anterior, um deles é selecionado para ser executado.

Essa solução gerencia o acesso ao processador, mas ainda permanece uma questão: quando temos mais de um thread em espera, como definir qual terá acesso ao processador?

Isso se torna necessário também quando o sistema está sendo iniciado ou o thread que estava liderando anteriormente não consegue mais se comunicar com os demais, seja por uma falha, por ter sido interrompido ou por ter ocorrido uma troca de contexto total. Na realidade, esse é um problema envolvendo sistemas distribuídos, uma vez que ocorre nas alocações de processamentos, e não no desenvolvimento dos códigos multithread em si. Portanto, não abordaremos esse assunto aqui.

7.4.1 Implementação de multithreads

O desenvolvimento de aplicações multithread não é simples, pois exige que a comunicação e o compartilhamento de recursos entre os diversos threads sejam feitos de forma sincronizada, para evitar problemas de inconsistências e deadlock (Tanenbaum o define como um conjunto de processos que "estará em situação de deadlock se todo processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer").

Dependendo da implementação, a definição do número de threads pode ser dinâmica ou estática. Quando a criação e/ou eliminação dos threads é dinâmica, estes são criados ou eliminados conforme a demanda da aplicação, oferecendo grande flexibilidade. Já nos ambientes estáticos, o número de threads é definido na criação do processo em que a aplicação será executada, limitando a saída do sistema.

Como foi visto, aplicações que são implementadas em multithreading devem então ser divididas em unidades independentes de execução. Assim, se uma aplicação envolve diversas unidades independentes que podem ser executadas de forma simultânea ou concorrente, podemos ter um ganho significativo de desempenho, mesmo em um ambiente no qual haja um único processador (lembrando que os processadores atuais já comportam processamento paralelo). Notadamente, as aplicações envolvendo sistemas distribuídos, compartilhando recursos de base de dados e hardware em geral, são as que mais se beneficiam desse modelo. No entanto, há a exigência de que a comunicação e o compartilhamento de recursos entre os threads sejam feitos de forma sincronizada, para evitar problemas. Os dois principais problemas que podem acontecer neste ambiente são:

- **Inconsistências:** um thread A que precisa da resposta de um thread B para finalizar é executado antes dele. Assim, temos um problema de ordem de execução, o que, embora a maioria dos sistemas tenha como contornar, afeta o desempenho do algoritmo como um todo.
- **Deadlock:** é quando "todo processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer" (Tanenbaum; Woodhull, 2008, p. 89). Ou seja, todos os threads estão esperando um thread ser executado e nenhum está realmente sendo executado, paralisando o processamento e a execução do programa.

As soluções para esses problemas, novamente, estão vinculadas às características de processamento e sistema operacional utilizadas no sistema, fugindo ao escopo deste livro-texto. Mas, em situações usuais, o programador pode fazer uso de algoritmos multithreading sem precisar se preocupar com esses aspectos.

Exemplo de aplicação

(Enade 2017, adaptado) Um programador inexperiente está desenvolvendo um sistema multithread que possui duas estruturas de dados diferentes, E1 e E2, as quais armazenam valores inteiros. O acesso concorrente a essas estruturas é controlado por semáforos. Durante sua execução, o sistema dispara os threads T1 e T2 simultaneamente. A tabela a seguir possibilita uma visão em linhas gerais dos algoritmos desses threads.

Tabela 7

T1	T2
Aloca E1	Aloca E2
Calcula a média M1 dos valores de E1	Calcula a soma S1 de todos os valores de E2
Aloca E2	Aloca E1
Calcula a média M2 dos valores de E2	Calcula a soma S2 de todos os valores de E1
Calcula $M3 = M1 + M2$	Calcula $S3 = S1 - S2 $
Soma M3 em todos os valores de E2	Subtrai S2 de todos os valores de E1
Libera E1	Libera E2
Libera E2	Libera E1

O que podemos afirmar sobre a possibilidade de deadlock durante a execução deste programa?

Vamos analisar os threads individualmente:

- T1 acessa primeiro E1 e na sequência E2, só liberando ambas as estruturas de dados após sua conclusão.
- T2, de forma semelhante, acessa primeiro E2 e na sequência E1, só liberando ambas após sua conclusão.

Assim, existe a possibilidade de T1, tendo terminado o processamento de E1, tentar acessar E2, que ainda estaria alocado para T2; e T2, tendo terminado o processamento de E2, queira acessar E1, que ainda está alocada para T1. Ou seja, cada thread estará esperando o outro terminar sua execução para terminar sua própria execução, em um impasse. Nesse cenário, temos uma situação de deadlock.

Uma forma de evitar essa situação é observar o uso que cada thread faz das duas estruturas de dados. Ambos os threads acessam e não modificam essas estruturas, apenas extraem seus dados. E o processamento de cada uma dessas estruturas, em ambos, é independente do processamento da outra estrutura de dados.

Assim, se fizermos os threads acessarem E1 e E2 na mesma ordem (os dois acessando E1 primeiro e depois E2, ou o oposto), os semáforos garantirão o gerenciamento adequado das alocações das bases de dados e o deadlock não ocorrerá mais.

8 PROBLEMAS NP

Nesse último tópico, vamos explorar os limites dos algoritmos, analisando problemas para os quais não sabemos se existem soluções próprias adequadas. Porém, muitos precisam ser solucionados computacionalmente, o que nos leva a abordagens alternativas para suas resoluções.

8.1 Tipos de problema

Ao longo deste livro-texto, foram feitos diversos comentários sobre as diferentes naturezas de problemas que resolvemos computacionalmente. De forma simplificada, podemos classificá-los em três grandes tipos (ainda que muitos problemas se enquadrem em mais de um):

- **Problemas de otimização:** podemos dizer simplificadamente que o objetivo é encontrar o máximo ou mínimo de algo. Entre os exemplos desses problemas que tratamos aqui, temos o MCOP (tópico 2), o algoritmo de Dijkstra (tópico 3) e o algoritmo de Ford-Fulkerson (tópico 5).
- **Problemas de busca:** queremos verificar a existência e localizar algo que satisfaça determinadas condições. Algoritmos de busca em largura e profundidade (vistos para árvores no tópico 3 e de forma mais ampla para grafos no tópico 5) e algoritmos de busca de subcadeias em uma cadeia maior (tópico 4) são alguns exemplos desse tipo de problema.
- **Problemas de tomada de decisão:** envolvem a escolha de uma entre duas opções: executar ou não uma instrução; seguir ou não por uma determinada aresta em um grafo. Na verdade, esses problemas são formas mais simples de problemas de busca. Como exemplo, temos a seleção dos símbolos para construir o código de Huffman (tópico 6), a escolha dos diferentes trajetos no algoritmo de Ford-Fulkerson e a organização dos elementos em um heap (tópico 3).

Problemas de otimização usualmente possuem algoritmos conhecidos e específicos para sua resolução. Por outro lado, problemas de busca podem ter uma solução mais complexa.

Veja-se o seguinte 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? (Cormen, 2013, p. 153). Nenhum algoritmo que vimos neste livro-texto resolve o problema. Podemos pensar na aplicação sucessiva do algoritmo de Dijkstra a partir do ponto atual para definir o próximo ponto, mas não temos garantia de que a somatória dos resultados individuais das diferentes aplicações trará o melhor resultado no conjunto. A aplicação de um algoritmo guloso (tópico 6) parece uma boa alternativa, mas estes algoritmos se baseiam na melhor decisão no momento, não levando o conjunto em consideração.

Uma última alternativa seria uma solução de força bruta: calcular todas as possíveis rotas e escolher a melhor. Porém, esse algoritmo tem um problema: existe um número de $12!$ rotas possíveis, o que significa que são mais de 470 milhões de possibilidades. Ou seja, temos um algoritmo de complexidade de tempo $O(n!)$, a pior possível.

8.2 Classes P e NP

Para podermos trabalhar adequadamente com problemas dessa natureza, vamos definir algumas terminologias antes de abordar como solucioná-los:

- **P**: designa todos os problemas que podem ser resolvidos com complexidade de tempo no máximo polinomial.
- **NP**: uma vez que tenhamos uma proposta de solução para ela, podemos verificar se 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*).

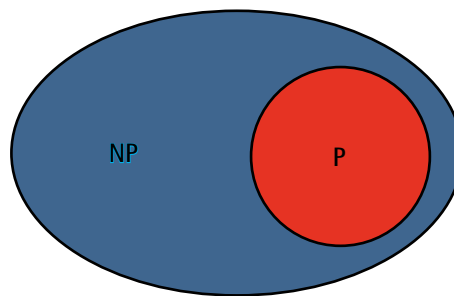
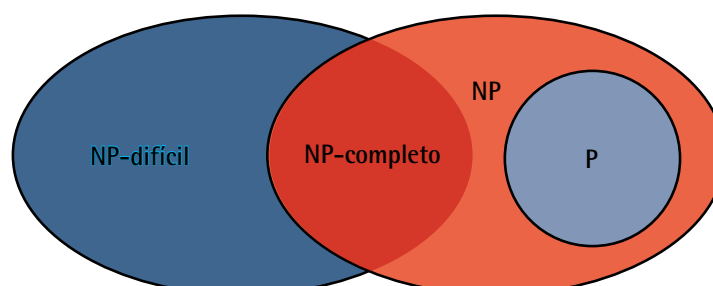


Figura 57 – Todo problema P é NP

- **NP-difícil**: caso exista um algoritmo de tempo polinomial para resolvê-lo, poderemos converter todos os problemas em NP para esse problema, de modo a poder resolver todos os problemas NP em tempo polinomial.
- **NP-completo**: um problema que é NP-difícil, e também é um problema NP.

A questão "Todos NP são P?", resumida na forma " $P = NP$?", é uma das grandes questões em aberto da matemática, fazendo parte do conjunto, com outras questões em aberto, das chamadas Sete Questões do Milênio, propostas pelo Instituto Clay de Matemática em 2000 (apenas uma dessas questões foi resolvida até hoje, em 2010). Para clarificar esta questão, a figura 58 ilustra as duas possibilidades de resposta para a pergunta, expandindo o diagrama da figura 57:

Se $P \neq NP$:



Se $P = NP$:

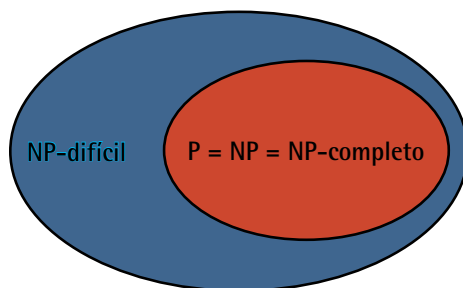


Figura 58 – As duas respostas possíveis para a pergunta " $P = NP$ "



Observação

Matemáticos especulam que a sobreposição dos conjuntos P e NP s possa ser de alguma forma diferente da apresentada. No entanto, esta mostrada na figura é a mais aceita atualmente.



Saiba mais

Há um prêmio oferecido de um milhão de dólares para a resolução de cada um dos Sete Problemas do Milênio. Conheça os outros problemas (em inglês):

THE MILLENNIUM Prize problems. *Claymath*, 1º mar. 2006. Disponível em: <https://tinyurl.com/3992fwkw>. Acesso em: 8 dez. 2023.

Para entendermos melhor o conceito de um problema NP , vamos dar um exemplo: considere um conjunto de números inteiros, positivos e negativos, com n elementos. O problema é: existe algum subconjunto não vazio deste conjunto cuja soma dos seus elementos resulte em zero? Temos claramente um problema de busca; para responder afirmativamente a essa pergunta, temos apenas que encontrar um subconjunto que satisfaça essa condição. Para encontrá-lo, temos que testar todos os subconjuntos possíveis: como vimos no tópico 1, existem 2^n subconjuntos, assim apenas o processo de obter os subconjuntos envolve uma complexidade de tempo que é exponencial (maior que polinomial). Testar uma parte dos conjuntos não resolverá o problema: se nenhum conjunto testado tiver a soma dos seus elementos igual a zero, isso não nos permite afirmar que o subconjunto desejado não exista (ele pode ser um dos que não foram testados).

Por outro lado, para verificar se um conjunto é uma resposta, o processo é bem simples, com uma complexidade de tempo linear igual ao número de elementos do subconjunto. Assim, claramente temos

um problema NP, que não pode ser resolvido em tempo de execução polinomial, mas que pode ter uma proposta de solução testada em tempo polinomial.

Agora, vamos supor que alguém encontre um algoritmo para a resolução desse problema que possa ser resolvido com complexidade de tempo polinomial. Como o que faz com que esse problema tenha tempo de execução exponencial é a geração de todos os subconjuntos, essa solução deveria conseguir gerar todos os subconjuntos em tempo polinomial. Assim, uma solução polinomial para este problema faria com que **todos** os problemas envolvendo subconjuntos se tornassem também problemas com tempo de execução polinomial. Portanto, esse é um problema NP-difícil.

Por fim, esse problema é NP e NP-difícil, é automaticamente um problema.

O estudo dessas classes de problemas se deve ao conjunto de perguntas que devemos fazer quando buscamos um algoritmo para implementar computacionalmente visando resolver um problema:

- 1) Existe um algoritmo conhecido específico para resolução desse problema?
- 2) Se existir, qual é a complexidade de tempo desse algoritmo?
- 3) Novamente, se o algoritmo existe, a complexidade de tempo é uma função aceitável?
- 4) Se a complexidade de tempo não for aceitável, existem soluções para implementar uma versão computacional com uma função melhor (mais rápida)?

A resposta para a primeira pergunta é: nem sempre. Existem problemas para os quais não se conhecem algoritmos específicos para sua resolução, restando como alternativa o uso de algoritmos "genéricos" (métodos de busca, algoritmos de força bruta etc.). Do ponto de vista matemático, existe uma grande diferença entre "não existir" e "não conhecer": na matemática, a não existência precisa ser provada. Se não é possível provar que algo não existe, isso significa que uma solução pode existir, mas ainda não é conhecida (ninguém descobriu ou criou o algoritmo em questão até o momento).

As perguntas 2 e 3 se relacionam: se existe o algoritmo, as funções de complexidade dele permitem que ele seja implementado computacionalmente de forma a gerar uma resposta em tempo aceitável? O termo "tempo aceitável" é relativo: em pesquisa científica, um algoritmo que leve alguns dias para ser executado é aceitável; na área de indústria e comércio, não (voltando ao exemplo da transportadora, não se pode esperar alguns dias, nem mesmo horas, para calcular a rota de uma única van).

Por fim, tudo nos leva à última pergunta: se não existe um algoritmo com um tempo de execução computacional aceitável (e o problema precisa ser resolvido), quais são as alternativas? Muitas envolvem reduzir o problema, aceitar uma solução não ótima como resposta ou utilizar um algoritmo que tenha alguma margem de erro (no qual a maioria das soluções está correta, mas há a pequena possibilidade de uma resposta inadequada em alguns casos) (Dasgupta; Papadimitriou; Vazirani, 2009, p. 183).

A seguir, alguns exemplos conhecidos de problemas NP-completos e como podemos obter soluções. Diversas abordagens tratadas nos tópicos anteriores serão revisitadas com um enfoque levemente modificado para que consigamos chegar a um algoritmo computacionalmente implementável.

Exemplo de aplicação

(Enade 2014, adaptado). Um cientista afirma ter encontrado uma redução polinomial de um problema NP-completo para um problema pertencente à classe P. Considerando que essa afirmação tem implicações importantes no que diz respeito à complexidade computacional, avalie as seguintes asserções e a relação proposta entre elas:

I – A descoberta do cientista implica que $P = NP$.

porque

II – A descoberta do cientista implica a existência de algoritmos polinomiais para todos os problemas NP-completos.

Como vimos, um problema NP completo tem duas condições: ser NP e NP-difícil. Se existe uma solução polinomial para um problema NP, isso significa que $P = NP$, respondendo, desse modo, a uma das Perguntas do Milênio (e o cientista em questão ganharia o prêmio do Instituto Clay de Matemática). Para tornar mais claro, reveja o diagrama da figura 58.

Assim, as duas afirmações estão corretas, e a afirmação II é uma justificativa correta da afirmação I.

8.3 Exemplos de problemas NP-completos

Após apresentar toda a questão matemática referente aos problemas NP-completos, os professores Dasgupta, Papadimitriou e Vazirani, em seu famoso livro sobre algoritmos, colocaram da seguinte maneira a busca por soluções:

Se tiver sorte, seu problema estará entre aquela meia dúzia de problemas sobre grafos com pesos (caminho mínimo, árvore espalhada mínima, fluxo máximo etc.) que resolvemos neste livro. Mesmo se esse for o caso, reconhecer um tal problema em seu habitat natural – obscurecido pela realidade e contexto – requer prática e habilidade. É mais provável que você tenha de reduzir seu problema a um destes afortunados – ou resolvê-lo usando programação dinâmica ou programação linear.

Mas há uma chance de que nada disso aconteça. O mundo dos problemas de busca é uma paisagem negra. Existem uns poucos pontos de luz – ideias algorítmicas brilhantes –, cada um iluminando uma pequena área em seu redor (os problemas que se reduzem a ele; duas dessas áreas,

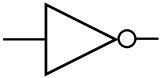
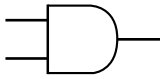
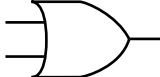
programação dinâmica e linear, são, de fato, bastante grandes). Mas a vasta extensão restante é escura como breu: NP-completa. O que você deve fazer? (2009, p. 282).

É exatamente o que faremos agora: apresentar alguns problemas NP-completos selecionados e mostrar, com as técnicas algorítmicas já estudadas, as soluções conhecidas e eventuais reduções dos problemas para permitir uma implementação computacional.

8.3.1 SAT e Circuit-SAT

Estes são os problemas mais importantes entre os NP-completos, e os primeiros demonstrados como tal. 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 (o termo SAT vem da abreviação de satisfazível) (Cormen, 2012, p. 780)?

A)

NOT (negação)		$\neg A$
AND (e)		$A \wedge B$
OR (ou)		$A \vee B$

B)

A	B	NOT A	NOT B	A AND B	A OR B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Figura 59 – Elementos de circuitos lógicos envolvidos no problema de Circuit-SAT

Embora possa parecer um problema simples, a verificação consome um tempo de execução polinomial: no pior caso, as combinações de entradas têm que ser testadas uma a uma até que uma satisfaça o circuito, sem sabermos quando isso ocorrerá. Por exemplo, o circuito lógico da figura 60 é satisfazível?

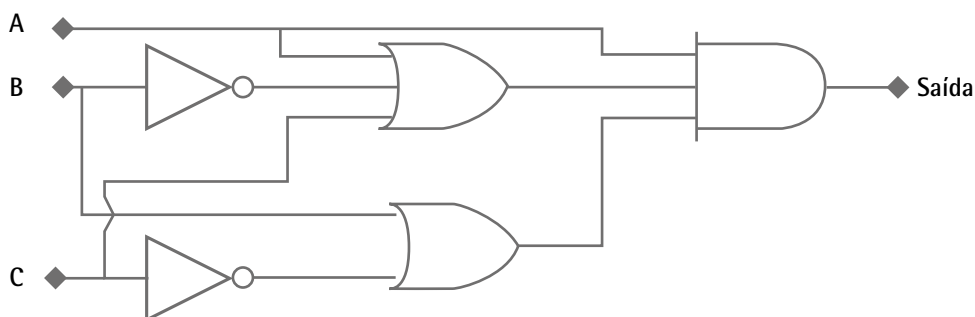


Figura 60 – Esse circuito lógico é satisfazível?

Se não sabemos uma solução, devemos testar todas as combinações de valores lógicos de A, B e C até encontrarmos uma que faça com que a saída do circuito seja igual a 1. Por outro lado, se tivermos uma candidata a solução, por exemplo $A = 1$, $B = 0$ e $C = 0$, podemos testar se esta satisfaz o circuito com extrema rapidez (se sim, essa solução satisfaz o circuito).

O problema pode parecer pequeno se considerarmos poucas entradas: com apenas três entradas no exemplo, temos apenas 2^3 combinações, e testar todas consumirá uma fração de segundo em computador comum. Porém, se tivermos 10 entradas, teremos 1024 possíveis soluções (2^{10}); para 40 entradas, o número de combinações passa de um trilhão. Ou seja, o teste individual é um problema de complexidade de tempo exponencial.

Outra versão para esse problema é o SAT, no qual o circuito lógico é expresso na forma de cláusulas lógicas. Assim, a expressão do circuito apresentado na figura 60 ficaria:

$$\text{Saída} = (A \vee \neg B \vee C) \wedge (B \vee \neg C) \wedge A$$



Lembrete

Em lógica matemática, cláusula é um conjunto de proposições, ou suas negações, conectadas por uma conjunção (E) ou disjunção (OU).

Em 1971, Steven Cook provou que o problema SAT é um problema NP-completo. Em 1984, Leonid Levin expandiu essa demonstração, no que hoje é 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. Daí a afirmação que fizemos anteriormente neste tópico: se existe 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, ou seja, $P = NP$.

Se as cláusulas lógicas envolvidas em um problema SAT tiverem apenas duas proposições cada, independentemente do número de proposições envolvidas, temos o problema denominado 2SAT. Este problema pode ser resolvido em tempo polinomial, não sendo um problema NP-completo (Dasgupta; Papadimitriou; Vazirani, 2009, p. 262). Porém, se as cláusulas contiverem três elementos cada, temos o problema 3SAT, que é NP-completo.

Opções eficientes, mas ainda não polinomiais, para resolver esses problemas são o algoritmo DPLL (criado em 1962 por Martin Davis, Hilary Putnam, George Logemann e Donald W. Loveland; o nome deriva das iniciais do sobrenome de seus criadores) e o algoritmo Chaff (uma variação do DPLL). Eles na verdade são um conjunto de algoritmos cada, que se baseiam na ramificação e propagação do valor lógico de uma das proposições e do uso de backtracking; assim, existem vários algoritmos sendo executados em conjunto, um para cada uma das proposições envolvidas. Assim, eles se beneficiam de um ambiente multithreading.



Lembrete

Backtracking é um conceito empregado em algoritmos no qual múltiplas soluções podem ser eliminadas sem ser previamente examinadas em detalhe.

8.3.2 Problema da soma dos subconjuntos

Este problema foi usado neste tópico como exemplo de um problema NP-completo. Vamos agora expandi-lo: 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 s ? 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 seja igual?

Como foi dito, esse é um problema de tempo de resolução exponencial. Muitas abordagens para tentar otimizar o tempo de resolução desse problema envolvem dividir o conjunto numérico, quando possível, em dois, um contendo os números positivos e outro os números negativos. Então, por meio de backtracking (dependendo do valor de s) soluções podem ser removidas e uma abordagem envolvendo programação dinâmica pode ser empregada. Para conjuntos pequenos, existem algoritmos que podem ter seu tempo de execução aproximado para uma função polinomial. No entanto, para problemas com conjuntos muito grandes, este continua sendo um problema NP-completo (Cormen, 2012, p. 799).

8.3.3 Problema da mochila com repetição

Esse problema na verdade é uma variação mais elaborada do problema da soma dos subconjuntos. O problema da mochila (muitas vezes apresentado por seu nome em inglês, knapsack problem) é de otimização. Seu enunciado pode ser definido da seguinte forma: 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 obter o maior valor possível dentro da mochila? Esse problema também é algumas vezes denominado problema do ladrão, assumindo que a mochila é de um ladrão que invadiu uma casa e os objetos a colocar foram furtados por ele.

Outra versão do mesmo problema tem uma abordagem mais moderna e vinculada à computação: dado um grande conjunto de arquivos de diferentes tamanhos, de que forma eles podem ser copiados para um conjunto de discos removíveis (com capacidade limitada cada um deles), de modo a precisarmos da menor quantidade de discos possível?



Figura 61 – Problema da mochila com repetição

Disponível em: <https://tinyurl.com/yu4at8dm>. Acesso em: 6 dez. 2023.

Esse problema é mais complicado que o da soma dos subconjuntos por dois motivos: primeiro, não existe a possibilidade de haver números negativos envolvidos; segundo, existem duas variáveis envolvidas, o peso e o valor, fazendo com que seja um problema de tempo de execução exponencial se adotarmos uma abordagem de força bruta. Esse problema serviu como base do primeiro algoritmo criptográfico de chaves assimétricas.

Muitas das abordagens atuais para a resolução desse problema envolvem algoritmos gulosos com backtracking e algoritmos genéticos.

8.3.4 Problema do caixeiro-viajante

Existem registros de problemas semelhantes a esse datados do século 19, e o enunciado que empregamos atualmente já foi proposto na década de 1930: "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?". O nome deriva dos caixeiros-viajantes, vendedores ambulantes que percorriam diversas cidades com seus produtos.

Ele é um problema NP-difícil, cuja natureza se baseia na necessidade dos vendedores em realizar entregas em diversos locais (as cidades) percorrendo o menor caminho possível, reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível. A figura 62 ilustra o problema mostrando as rotas entre diferentes cidades e uma possível solução:

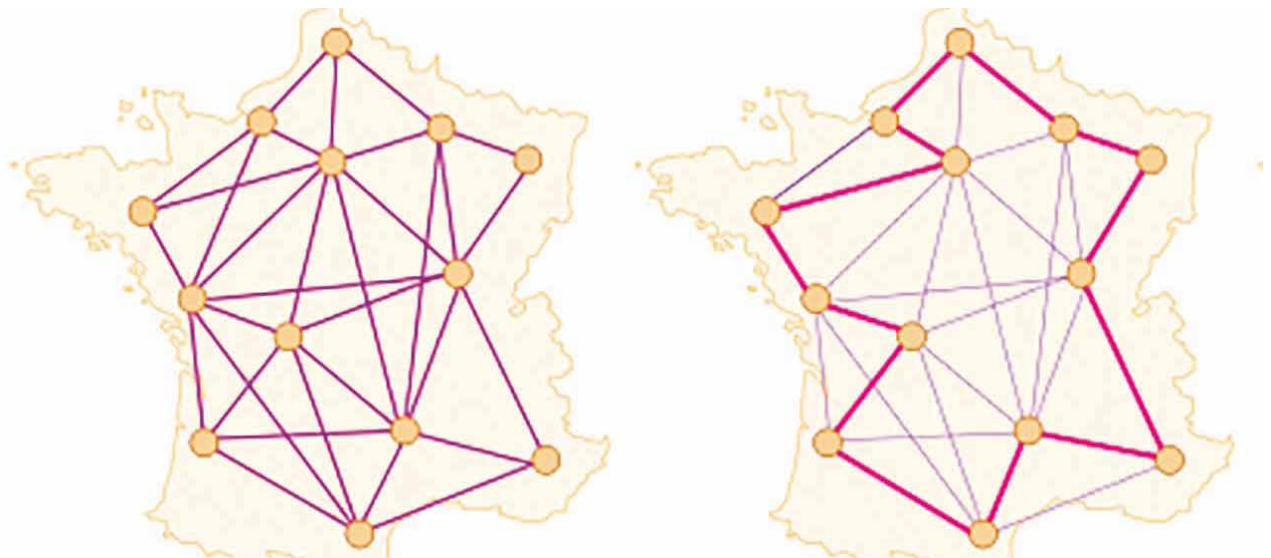


Figura 62 – Problema do caixeiro-viajante

Adaptada de: <https://tinyurl.com/yc3m6zxd>. Acesso em: 6 dez. 2023.

Sob uma ótica mais computacional, trata-se de um problema de trajeto em um grafo. Esse problema implica a resolução prévia de outro: existe uma rota possível que passe por todos os vértices do grafo, sem repetição, iniciando e terminando no mesmo vértice? Ou seja, é possível um trajeto hamiltoniano (ver tópico 5) nesse grafo? Por exemplo, o mapa da Romênia apresentado no exemplo de algoritmos gulosos no tópico 6 (figura 42) não permite, com apenas aquelas estradas, um trajeto hamiltoniano.

O problema de definir se um grafo possui ou não ao menos um trajeto hamiltoniano é por si só também um problema NP-completo. Ou seja, temos que resolver um problema NP-completo para resolver outro problema NP-completo. Alguns grafos são sabidamente hamiltonianos, assim como outros sabidamente não o são. Por exemplo, todo grafo completo (nos quais cada vértice tem pelo menos uma aresta conectada a cada um dos outros vértices) é hamiltoniano. Esse conhecimento pode, dependendo do grafo analisado, reduzir o problema (Cormen, 2012, p. 794).

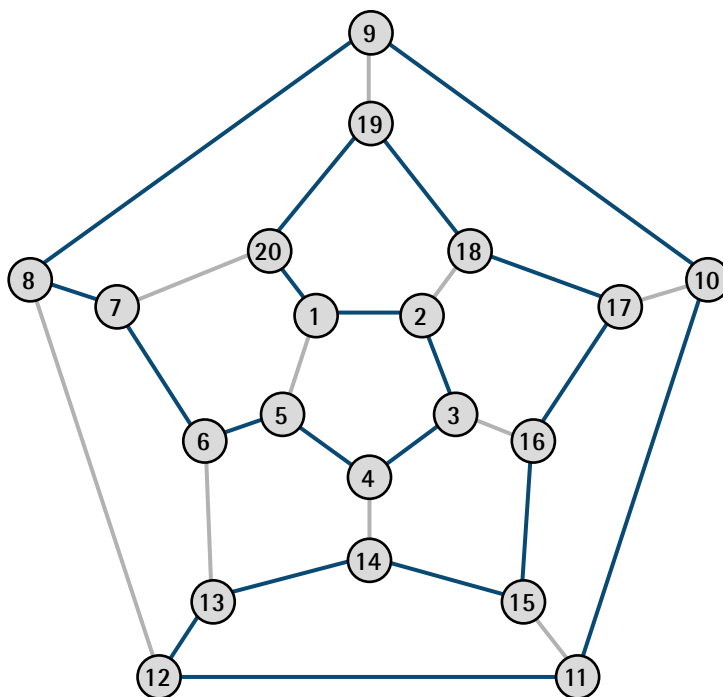


Figura 63 – Exemplo de grafo hamiltoniano

Disponível em: <https://tinyurl.com/ypzje4pn>. Acesso em: 6 dez. 2023.

O problema do caixeiro-viajante, sendo sabido que o grafo em questão possui um trajeto hamiltoniano, tem uma complexidade de tempo fatorial (a pior de todas, conforme vimos no tópico 1). Muitas abordagens para resolver esse problema envolvem a redução do grafo a uma árvore geradora mínima (via algoritmo de Kruskal ou algum outro) e a aplicação subsequente de algoritmos gulosos (Cormen, 2012, p. 809).

Uma abordagem interessante é o chamado algoritmo da otimização da colônia de formigas (ACO, do inglês *ant colony optimization algorithm*), criado pelo cientista italiano Marco Dorigo (1961-). Esse algoritmo se utiliza de uma heurística probabilística para encontrar caminhos em grafos. Basicamente consiste em definir um caminho inicial aleatório. Outras iterações seguirão os caminhos anteriores, os caminhos mais percorridos serão reforçados, e os caminhos menos percorridos acabarão apagados. O nome desse algoritmo deriva do comportamento de formigas percorrendo uma trilha e deixando um rastro de feromônios. Caminhos com mais rastro serão percorridos por mais formigas, e os com menos rastro serão apagados (Goldbarg; Goldbarg, 2012, p. 555).

Existem muitas variações desse problema, com diferentes heurísticas aplicadas nas suas resoluções. Talvez seja um dos problemas NP-completos mais estudados, devido à sua aplicabilidade em diferentes áreas, tais como logística, roteamento de redes de computadores, telecomunicações em geral etc.

Exemplo de aplicação

(Enade 2014, adaptado). Considere o processo de fabricação de um produto siderúrgico que necessita passar por n tratamentos térmicos e químicos para ficar pronto. Cada uma das n etapas de tratamento é cumprida uma única vez na mesma caldeira. Além do custo próprio de cada etapa do tratamento, existe o custo de se passar de uma etapa para outra, uma vez que, dependendo da sequência escolhida, pode ser necessário alterar a temperatura da caldeira e limpá-la para evitar a reação entre os produtos químicos utilizados. Assuma que o processo de fabricação inicia e termina com a caldeira limpa. Deseja-se projetar um algoritmo para iniciar a sequência de tratamentos que possibilite fabricar o produto com o menor custo total. De que tipo de algoritmo estamos falando?

Apesar da estranheza, do ponto de vista industrial, do problema proposto, computacionalmente podemos pensar nele como um grafo, no qual cada processo (p) é um vértice e cada custo de transição entre processos (c) é uma aresta. A figura a seguir ilustra isto para três processos:

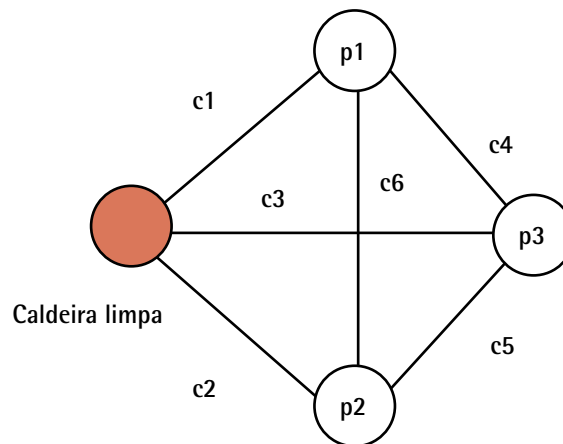


Figura 64

Ou seja, o problema consiste em iniciar no estado de caldeira limpa, passar por todos os outros estados e retornar ao estado inicial, percorrendo o trajeto de menor custo total. Em resumo, trata-se de uma versão do problema do caixeiro-viajante.

Uma consideração especial é que, como o enunciado da questão dá a entender que pode se passar de qualquer processo para qualquer outro processo, temos um grafo completo. Em um grafo completo, sempre é possível um trajeto hamiltoniano, ou seja, o "pré-problema" de verificar se existe uma rota que não precisa ser resolvida, pois existirão diversos trajetos possíveis.

8.4 Considerações finais

Vimos ao longo deste livro-texto que, para dados problemas, existem diversas soluções que podem ser implementadas computacionalmente, com diferentes complexidades de tempo e espaço. Assim, devemos sempre buscar a solução que melhor se adeque à situação que estamos resolvendo, levando em conta diversos aspectos:

- A natureza de problema que precisa ser resolvido.
- A linguagem de programação utilizada (as linguagens mais modernas oferecem uma ampla gama de recursos em bibliotecas e módulos). Vários dos exemplos apresentados neste livro-texto fazem uso de módulos específicos do Python para aproveitar as ferramentas dessa linguagem.
- A disponibilidade de recursos computacionais: capacidade de memória e processamento e, principalmente, de tempo de execução adequado para um algoritmo.
- O conhecimento de lógica de programação e algoritmos, uma vez que para muitos problemas já existem algoritmos consagrados e não precisamos ficar "reinventado a roda".

Porém, também vimos que existem problemas para os quais não se conhece (perceba que evitamos utilizar a expressão "não existe") um algoritmo ideal, que são os problemas NP. Para esses casos, no entanto, diferentes abordagens podem ser empregadas para se conseguir uma solução (nem sempre perfeita, mas boa o bastante para ser aceita). Como não temos como saber qual algoritmo é o melhor para resolver um problema NP-completo específico, adotamos diferentes abordagens para obter uma solução:

- **Busca exaustiva:** tentar todas as soluções possíveis para um problema, em uma abordagem também chamada de computação de força bruta. Muitas vezes utiliza-se o backtracking para eliminar parte das candidatas a solução. É um método que sempre nos trará uma resposta, se esta existir, mas possui complexidade de tempo extremamente elevada. No entanto, considerando os computadores atuais, é viável para problemas pequenos, em problemas cujas soluções possíveis a serem testadas somam apenas algumas centenas, por exemplo.
- **Algoritmos gulosos:** tendem a encontrar uma solução ótima (mas nem sempre a melhor) para um problema. Muitos métodos convergentes utilizados em cálculo numérico e outros empregados em IA (por exemplo, algoritmos genéticos) podem ser considerados, sob certos aspectos, gulosos.
- **Busca heurística:** quanto mais conhecemos a natureza de um problema, mais podemos utilizar esse conhecimento na construção de um algoritmo para resolvê-lo. O conceito de heurística envolve justamente utilizar o máximo de conhecimento disponível para testar e descartar possíveis soluções de modo a otimizar a resolução do problema. Muitas das ferramentas algorítmicas mais modernas da IA, como redes neurais e métodos de aprendizado de máquina, fazem uso de heurísticas.

- **Algoritmos genéticos:** essa abordagem foi proposta por John Henry Holland (1929-2015) em 1975. Consiste em criar um conjunto de soluções (denominado população), inicialmente aleatória, para o problema. Uma função de seleção é aplicada a essas soluções, sendo as melhores combinadas entre si (cruzamento) e aplicando-se modificações aleatórias em algumas delas (mutações) para gerar uma nova população. Esse processo é repetido iterativamente, e cada nova população terá soluções mais adequadas, convergindo para uma solução definitiva. Um algoritmo genético é uma abordagem de alto custo computacional, tanto de tempo quanto de espaço, mas é uma alternativa para a resolução de muitos problemas NP-completos.
- **Computação quântica:** baseia-se no conceito da superposição dos elétrons. De forma simplificada, um elétron pode estar em um conjunto de estados simultaneamente: se transformarmos isso em informação, temos os bits quânticos (qubits), que não estão restritos a apenas um de dois valores binários cada vez (Dasgupta; Papadimitriou; Vazirani, 2009, p. 310). Algoritmos quânticos fazendo uso dessas propriedades são radicalmente diferentes dos algoritmos computacionais convencionais, atingindo velocidades que começam a apenas ser conhecidas. Algumas pessoas alertam para a possibilidade de um "apocalipse quântico" no futuro: a computação quântica seria tão veloz que seria capaz de quebrar toda e qualquer medida de segurança computacional de hoje. Estudos para estabelecer novas ferramentas de segurança da informação à prova de computação quântica estão sendo realizados. Mas é claro que para implementar a computação quântica precisamos de computadores quânticos, que ainda estão muito longe de estar amplamente disponíveis.

Com essas considerações, finalizamos mais uma jornada pelo mundo dos algoritmos. Esperamos que novos conceitos tenham sido aprendidos e que os algoritmos e ideias aqui apresentadas se tornem úteis na vida do aluno.



Resumo

Grafos são uma forma de modelar conjuntos de dados em que temos diversos elementos que se relacionam entre si de forma não sequencial e não linear. Muitas situações reais podem ser modeladas assim: rotas, redes, fluxos, dados (ordenados ou não).

Entre os algoritmos sobre grafos, os de busca estão entre os mais importantes: eles têm como finalidade localizar a existência ou não de um elemento em uma estrutura de dados organizada na forma de um grafo. Outro tipo de busca é por rota: verificar se existe ou não uma rota entre dois vértices do grafo, e qual seria a melhor rota entre dois vértices.

Uma rede é um grafo no qual temos um ponto de partida denominado origem e um ponto de destino, e um fluxo (pessoas, dados etc.) se desloca entre estes dois pontos. O fluxo máximo de uma rede é o limite de fluxo que o grafo comporta, considerando as capacidades dos nós intermediários entre a origem e o destino.

Um algoritmo para determinar o fluxo máximo de uma rede é o de Ford-Fulkerson. Esse algoritmo, assim como vários outros, não pode ter seu tempo de execução avaliado adequadamente por meio de uma redução assintótica a uma função de complexidade de tempo. Assim, para esses casos, utilizamos a análise amortizada, que consiste em dividir o problema em um conjunto de operações e estimar um tempo médio, por meio de amortização, para cada uma delas. O tempo total de execução do algoritmo será a somatória dos tempos de todas as operações.

Algoritmos gulosos são uma abordagem na qual existe uma tomada de decisão em busca de uma resposta, e essa tomada de decisão se baseia em uma estimativa de quão perto se chegaria da resposta desejada com a decisão escolhida. O código de Huffman gera uma codificação binária eficiente de tamanho variável para um conjunto de dados e tem exemplos de algoritmos gulosos.

Algoritmos multithread são algoritmos nos quais o programa pode ser construído de modo que diferentes instruções do programa possam ser executadas simultaneamente, paralelamente, beneficiando-se das arquiteturas dos processadores e computadores modernos.

Por fim, problemas-NP são problemas que não podem ser resolvidos em um tempo de execução polinomial ou melhor, mas que podem ter

suas soluções testadas nesse tempo. Esses problemas são os limites do conhecimento da matemática algorítmica atual. Entre as abordagens possíveis para resolução desses problemas em um tempo computacional razoável estão os algoritmos gulosos, algoritmos com backtracking, buscas heurísticas, entre outras opções.



Exercícios

Questão 1 (Fapec 2022, adaptada). A figura a seguir ilustra um grafo. Assinale a alternativa que representa corretamente o grafo como uma matriz de adjacência.

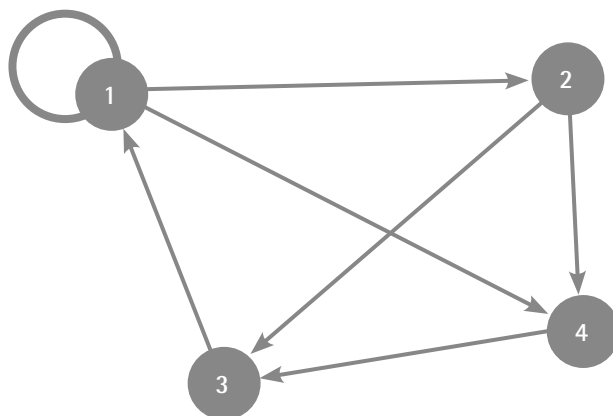


Figura 65

A)

	1	2	3	4
1	1	1	0	1
2	0	0	1	1
3	1	0	0	0
4	0	0	1	0

B)

	1	2	3	4
1	0	0	1	0
2	1	1	0	0
3	0	1	1	1
4	1	1	0	1

C)

	1	2	3	4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

D)

	1	2	3	4
1	-1	1	0	1
2	0	0	1	1
3	1	0	0	0
4	0	0	1	0

E)

	1	2	3	4
1	1	1	1	1
2	0	0	1	1
3	1	0	0	0
4	1	1	0	0

Resposta correta: alternativa A.

Análise da questão

Uma matriz de adjacências representa um grafo. Cada linha representa um dos vértices do grafo, e os valores de seus elementos representam as conexões que esse vértice estabelece com outro. Para um grafo não ponderado, como o encontrado no enunciado da questão, a matriz de adjacências conterá apenas elementos 1 (se houver conexão entre dois nós) e 0 (se não houver conexão entre dois nós). Logo, não é possível ter um elemento -1, como na resposta da alternativa D.

Analisando o nó 1 do grafo, vemos que ele se conecta a ele mesmo, ao nó 2 e ao nó 4. Logo, a primeira linha da matriz de adjacências é preenchida conforme exposto a seguir.

	1	2	3	4
1	1	1	0	1
2				
3				
4				

Analisando o nó 2, vemos que ele se conecta ao nó 3 e ao nó 4. Logo, a segunda linha é preenchida do modo a seguir.

	1	2	3	4
1	1	1	0	1
2	0	0	1	1
3				
4				

Analisando o nó 3, vemos que ele se conecta apenas ao nó 1, resultando na linha a seguir.

	1	2	3	4
1	1	1	0	1
2	0	0	1	1
3	1	0	0	0
4				

Finalmente, vemos que o nó 4 se conecta apenas ao nó 3, resultando nesta matriz de adjacências:

	1	2	3	4
1	1	1	0	1
2	0	0	1	1
3	1	0	0	0
4	0	0	1	0

Questão 2 (Instituto AOCF 2019, adaptada). Na área computacional, existem tanto algoritmos sequenciais quanto paralelos, conhecidos como multithreads.

Em relação aos algoritmos multithread, avalie as afirmativas.

I – São algoritmos com uma fila prioritária para ser executados paralelamente em computadores monoprocessados.

II – Podem constar em computadores multiprocessados que permitem a execução concorrente de diversas instruções.

III – Diversas linguagens de programação modernas contêm classes e módulos para a programação multithread.

É correto o que se afirma em:

A) I, apenas.

B) III, apenas.

C) I e II, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa D.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: a execução paralela geralmente envolve a distribuição de tarefas em diferentes threads, para que elas sejam executadas simultaneamente. Em computadores monoprocessados, nos quais há apenas um núcleo de processamento, threads não podem ser executados verdadeiramente em paralelo, já que não há modo de executar threads distintos ao mesmo tempo. Nesse caso, podem ser executados alternadamente, por meio da execução multitarefa.

II – Afirmativa correta.

Justificativa: em computadores multiprocessados, diversos threads podem ser executados em paralelo, aproveitando os múltiplos núcleos de processamento.

III – Afirmativa correta.

Justificativa: muitas linguagens de programação modernas, como a linguagem Python, oferecem suporte para programação multithread. Essas linguagens fornecem bibliotecas, classes e módulos que facilitam a criação e a coordenação de threads.

REFERÊNCIAS

Textuais

ARITHMETIC, population and energy – a talk by Al Bartlett. *Albartlett.org*, 11 jul. 2015. Disponível em: <http://tinyurl.com/5aw63kjb>. Acesso em: 13 dez. 2023.

BRASIL, M. L. R. F.; BALTHAZAR, J. M.; GÓIS, W. *Métodos numéricos e computacionais na prática de engenharias e ciências*. São Paulo: Blucher, 2015. Disponível em: <https://tinyurl.com/s6rubyny>. Acesso em: 7 dez. 2023.

BUILT-IN functions. *Python*, 2 out. 2008. Disponível em: <https://tinyurl.com/274wsdsd>. Acesso em: 7 dez. 2023.

CHAPRA, S. C.; CANALE, R. P. *Métodos numéricos para engenharia*. Porto Alegre: AMGH, 2016. Disponível em: <https://tinyurl.com/2wnxspf8>. Acesso em: 7 dez. 2023.

CORMEN, T. *Algoritmos: teoria e prática*. Rio de Janeiro: LTC, 2012. Disponível em: <https://tinyurl.com/2am6f3uu>. Acesso em: 7 dez. 2023.

CORMEN, T. *Desmistificando algoritmos*. Rio de Janeiro: Campus, 2013. Disponível em: <https://tinyurl.com/2p92nj4b>. Acesso em: 7 dez. 2023.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. *Algoritmos*. Porto Alegre: McGraw-Hill, 2009. Disponível em: <https://tinyurl.com/bdh9xza7>. Acesso em: 7 dez. 2023.

FEOFILOFF, P. *Algoritmos em linguagem C*. Rio de Janeiro: Elsevier, 2009.

FEOFILOFF, P. Programação dinâmica. *IME-USP*, 3 jul. 2020. Disponível em: <https://tinyurl.com/2b4bwktv>. Acesso em: 7 dez. 2023.

GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação*. Rio de Janeiro: LTC, 2016. Disponível em: <https://tinyurl.com/473hwnuj>. Acesso em: 7 dez. 2023.

GOLDBARG, M.; GOLDBARG, E. *Grafos: conceitos, algoritmos e aplicações*. São Paulo: Campus, 2012. Disponível em: <https://tinyurl.com/mw47wzsc>. Acesso em: 7 dez. 2023.

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados e algoritmos em Java*. Porto Alegre: Bookman, 2013. Disponível em: <https://tinyurl.com/2eba5xvv>. Acesso em: 7 dez. 2023.

LIPSCHUTZ, S.; LIPSON, M. *Matemática discreta*. Porto Alegre: Bookman, 2013. Disponível em: <https://tinyurl.com/yaabs7t9>. Acesso em: 7 dez. 2023.

MENEZES, P. B. *Linguagens formais e autômatos*. Porto Alegre: Grupo A, 2011. v. 3. Disponível em: <https://tinyurl.com/yt7hd9em>. Acesso em: 7 dez. 2023.



Handwriting practice lines consisting of 30 horizontal blue lines. Each line is preceded by a small blue square marker. The lines are evenly spaced and extend across the width of the page.



Informações:
www.sepi.unip.br ou 0800 010 9000