Unidade IV

7 ALGORITMOS EM GRAFOS

Em um grafo simples, os vértices representam objetos ou estados, enquanto arestas simbolizam interações, dependências, rotas ou fluxos de informação. Ao atribuir pesos às arestas, adiciona-se dimensão quantitativa apta a expressar custos, distâncias ou capacidades, permitindo tratamento de problemas de otimização. Desde sistemas de navegação até redes neurais profundas, a eficiência do software depende da escolha apropriada de algoritmos que percorrem, medem ou transformam essas topologias.

A exploração inicial costuma recorrer às buscas em largura e profundidade. A busca em largura parte de um vértice fonte e visita vizinhos em camadas crescentes, garantindo descoberta do caminho mínimo em número de saltos quando o grafo não apresenta pesos. O emprego de fila permite que a travessia mantenha ordem de expansão por distância. A busca em profundidade, por sua vez, avança por um caminho até o fim para depois retroceder, procedimento natural para identificar componentes conectados, detectar ciclos ou gerar ordens topológicas em grafos acíclicos direcionados. Ambas consomem tempo proporcional à soma de vértices e arestas, além de memória que se restringe a estruturas de marcação e, no caso da busca em largura, a fila de fronteira.

Quando as arestas carregam pesos não negativos, surge o problema de caminhos mínimos ponderados, resolvido com eficiência pela técnica de Dijkstra. Esse procedimento mantém conjunto de vértices com distância definitiva e utiliza fila de prioridade para selecionar, a cada passo, o vértice com menor custo provisório, relaxando arestas adjacentes. A complexidade resulta de extrações e diminuições de chave, operações que, com heap binário, conferem tempo O((V+E) log V). Na presença de pesos negativos, a abordagem de Dijkstra deixa de oferecer garantias e cede lugar ao algoritmo de Bellman-Ford, que executa relaxamento de todas as arestas repetidas vezes até não ocorrerem melhorias. Ainda que seu consumo temporal O(V E) se mostre superior em magnitude, Bellman-Ford detecta ciclos de peso negativo, condição crítica em aplicações financeiras ou de planejamento energético, nas quais ciclos lucrativos ou perdas irreversíveis exigem tratamento explícito.

Lembrete

Como visto no quarto exercício do tópico 3, V ("vértices") é comumente usado para denotar o número de vértices de um grafo e E ("edges", arestas) é a convenção para o número de arestas. Essa notação é amplamente adotada para facilitar a comunicação e a análise de complexidade.

A construção de árvores geradoras mínimas introduz outro conjunto fundamental de algoritmos. Kruskal escolhe arestas em ordem crescente de peso enquanto mantém florestas disjuntas, unindo componentes

até que todos os vértices se encontrem conectados. A utilização de união-busca com compressão de caminho garante quase linearidade na prática. Prim, em contraposição, expande um único componente, adicionando sempre a aresta de menor custo que conecta o conjunto já escolhido a um vértice externo, estratégia que se beneficia de fila de prioridade semelhante à de Dijkstra. Áreas de engenharia civil, redes de distribuição elétrica e design de circuitos integram tais técnicas em sistemas de decisão que minimizam consumo de recursos.

A robustez dos algoritmos em grafos provém da versatilidade na modelagem de problemas reais, unindo notação concisa e fundamentos combinatórios sólidos. A seleção cuidadosa entre busca em largura, busca em profundidade, Dijkstra, Bellman-Ford, Kruskal, Prim ou estratégias de fluxo define a viabilidade e a escalabilidade de soluções que dependem da análise de caminhos, conexões ou capacidades. Desenvolvedores e pesquisadores encontram, em bibliotecas amplamente adotadas como NetworkX ou Graph-tool, implementações maduras que incorporam essas ideias, porém a compreensão conceitual permanece indispensável quando o intuito envolve adaptar critérios de peso, refinar políticas de fila ou paralelizar etapas críticas. O conhecimento dos algoritmos em grafos habilita profissionais a transformar bases de dados relacionais, malhas de transporte, arquiteturas de risco e ecossistemas de informação em estruturas navegáveis, fornecendo respostas confiáveis em prazos bem definidos e com consumo controlado de recursos.

7.1 Caminhos mínimos: Dijkstra e Bellman-Ford

A determinação de caminhos mínimos em grafos constitui tema central da ciência da computação por viabilizar rotas com menor custo em mapas de tráfego rodoviário, redes de comunicação e fluxos de suprimentos. Em Python, duas abordagens clássicas atendem a tal demanda: o algoritmo de Dijkstra e o de Bellman-Ford. Ambos partem de um vértice de origem e atribuem a cada vértice adjacente um valor que representa a distância mais curta já conhecida, atualizando essas estimativas conforme as arestas são examinadas. Apesar de perseguirem o mesmo objetivo, as estratégias divergem na maneira de selecionar o próximo vértice a processar, na forma de atualizar distâncias e nas hipóteses sobre os pesos das arestas.

Dijkstra admite que todos os pesos sejam não negativos e emprega uma fila de prioridade para escolher em cada iteração o vértice cuja distância provisória já atingiu o menor valor entre os que permanecem por visitar. Ao retirar esse vértice da fila, a distância associada torna-se definitiva, pois nenhum caminho alternativo poderá reduzi-la, dado que todas as rotas futuras acrescentarão custo positivo. A estrutura de dados típica para expressar a fila de prioridade em Python é o heap binário oferecido pelo módulo `heapq`. A cada extração, o algoritmo percorre as arestas que partem do vértice recém-fixado, recalculando o custo até os seus vizinhos mediante a operação denominada relaxamento. Se a soma da distância definitiva com o peso da aresta reduzir a estimativa anterior do vizinho, essa estimativa é substituída e o par atualizado retorna à fila.

A complexidade global depende da eficiência da fila, situando-se em $O((V + E) \log V)$ quando o heap binário gerencia tanto extrações quanto inserções resultantes dos relaxamentos. Em muitos problemas práticos, tal desempenho apresenta-se excelente, sobretudo em grafos esparsos, nos quais o número de arestas cresce linearmente em relação aos vértices.

O funcionamento do algoritmo é bem simples. Considere a figura a seguir como algumas cidades e as distâncias (em quilômetros) entre elas.

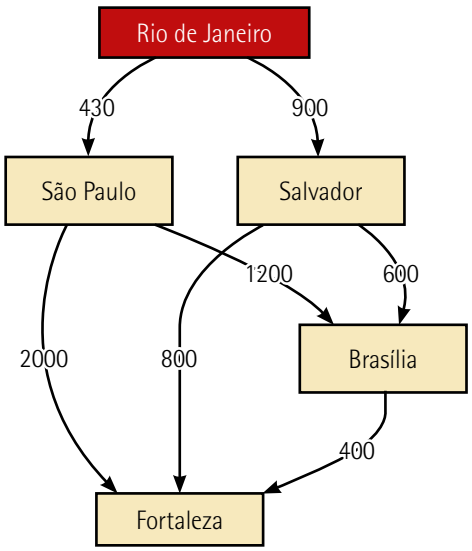

Figura 18 – Cidades e distâncias em quilômetros: grafo ponderado

Imagine que você é um turista que começa sua jornada no Rio de Janeiro e quer visitar todas essas cidades, mas quer minimizar a distância percorrida. O algoritmo de Dijkstra é como um assistente de viagem que ajuda a escolher as rotas mais curtas.

- **Passo 1**: você começa no Rio de Janeiro e verifica as distâncias até as cidades mais próximas. As opções são:
  - São Paulo a 430 km.
  - Salvador a 900 km.
  - O algoritmo escolhe São Paulo, pois está mais próxima.
- **Passo 2**: agora você está em São Paulo e precisa ir para a próxima cidade. As opções são:
  - Brasília a 1.200 km.
  - Fortaleza a 2.000 km.
  - O algoritmo escolhe Brasília, pois está mais próxima.
- **Passo 3**: você chega a Brasília e verifica as opções de cidades próximas:
  - Fortaleza a 400 km.
  - O algoritmo escolhe Fortaleza, pois é a cidade mais próxima.

- **Passo 4**: agora que você está em Fortaleza, o único destino restante é Salvador, que está a 800 km de distância.

Ao final do percurso, o algoritmo de Dijkstra terá identificado o caminho mais curto entre as cidades, minimizando a distância total percorrida. O ponto fundamental é que ele sempre escolhe o caminho mais curto disponível a cada etapa, sem precisar verificar todas as possibilidades de antemão.

Vamos fazer outro exemplo: imagine que você está gerenciando uma fábrica que produz peças metálicas e a produção passa por várias etapas de processamento, cada uma realizada por um equipamento específico. O objetivo é minimizar o custo total, levando em consideração o tempo de uso, consumo de energia ou desgaste de cada equipamento. Segue a lista de equipamentos da fábrica:

- **Prensa**: o material começa a ser moldado.
- **Moinho**: o material é triturado ou processado em menor escala.
- **Forno**: o material é aquecido.
- **Cortadora**: o material é cortado em formas específicas.
- **Lixadeira**: o material é polido ou suavizado.
- **Acabamento**: etapa final de polimento e acabamento da peça.

Essa lista pode ser representada como um grafo, conforme a figura a seguir:

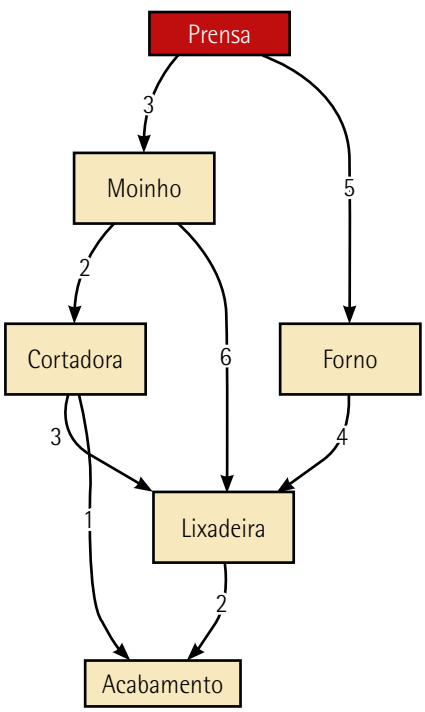

Figura 19 – Equipamentos e custos: grafo ponderado

- **Passo 1**: o processo de produção começa na prensa. O operador tem duas opções:
  - Enviar o material para o moinho a um custo de três unidades.
  - Enviar o material para o forno a um custo de cinco unidades.
    - O algoritmo de Dijkstra escolhe o moinho, pois o custo é mais baixo.
- **Passo 2**: agora, o material está no moinho. As opções de transição são:
  - Enviar para a cortadora a um custo de duas unidades.
  - Enviar para a lixadeira a um custo de seis unidades.
    - O algoritmo escolhe a cortadora, pois o custo é menor.
- **Passo 3**: o material chega à cortadora. As opções de transição são:
  - Enviar para a lixadeira a um custo de três unidades.
  - Enviar para o acabamento a um custo de uma unidade.
    - O algoritmo escolhe o acabamento, pois o custo é mais baixo.
- **Passo 4**: o material agora segue diretamente para o acabamento, finalizando o processo.

O algoritmo de Dijkstra identificou a sequência mais eficiente de equipamentos para processar o material: prensa → moinho → cortadora → acabamento, com o menor custo total de 3 + 2 + 1 = 6 unidades. Em uma fábrica real, o algoritmo de Dijkstra poderia ser utilizado para otimizar o uso dos equipamentos, levando em conta o custo operacional, tempo de inatividade, consumo de energia ou até mesmo o desgaste de cada máquina. Isso ajudaria a garantir que o processo de produção fosse o mais eficiente possível, economizando recursos e reduzindo custos.

Agora, imaginemos um cenário em que o custo de utilizar determinados equipamentos na fábrica varia ao longo do tempo. Por exemplo, durante certos períodos do dia, o custo operacional das máquinas pode ser menor devido a descontos, manutenção programada ou outros fatores. Isso cria arestas com custos negativos temporários:

- Durante certos períodos do dia, a cortadora pode ser usada com desconto devido à manutenção preventiva, o que cria um custo negativo ao transitar para ela.
- Por outro lado, o uso do forno pode ter custos mais elevados devido à sobrecarga de demanda.

A figura a seguir mostra o grafo desse cenário.

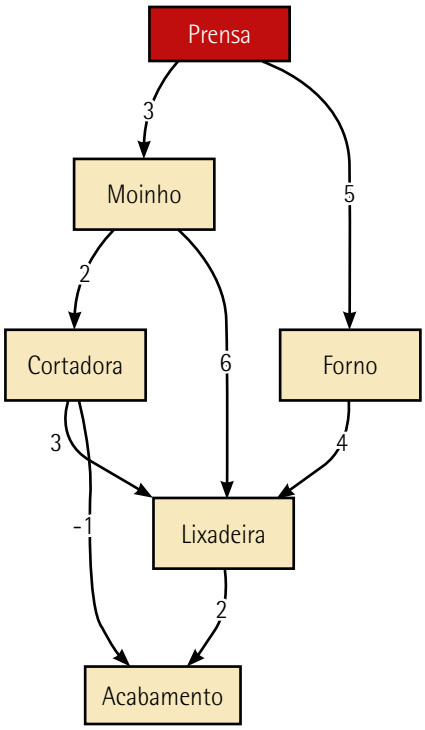

Figura 20 – Equipamentos e custos: grafo ponderado. Note uma pequena alteração: custo negativo da cortadora para o acabamento

Usando o algoritmo de Bellman-Ford:

- **Passo 1**: o processo de produção começa na prensa. O operador tem duas opções:
  - Enviar o material para o moinho a um custo de três unidades.
  - Enviar o material para o forno a um custo de cinco unidades.
    - O algoritmo de Bellman-Ford escolhe o moinho, pois o custo é mais baixo.
- **Passo 2**: agora, o material está no moinho. As opções de transição são:
  - Enviar para a cortadora a um custo de duas unidades.
  - Enviar para a lixadeira a um custo de seis unidades.
    - O algoritmo escolhe a cortadora, pois o custo é menor.
- **Passo 3**: o material chega à cortadora. Aqui, o algoritmo observa que a transição para o acabamento tem um custo negativo de -1 unidade devido a um desconto especial. As opções de transição são:

– Enviar para a lixadeira a um custo de três unidades.

– Enviar para o acabamento com um custo de -1 unidade.

- O algoritmo escolhe o acabamento, aproveitando o custo negativo e, portanto, reduzindo o custo total.

- **Passo 4**: o material chega diretamente ao acabamento, finalizando o processo.

No final do cálculo, o algoritmo de Bellman-Ford identificou a sequência mais eficiente, considerando os custos negativos e os ajustes no custo das máquinas: prensa → moinho → cortadora → acabamento, com um custo total de 3 + 2 + (-1) = 4 unidades, que é mais eficiente devido ao uso do desconto na cortadora.

Consequentemente, em cenários nos quais podem existir pesos negativos, Dijkstra perde a garantia de correção, pois a definição de um caminho definitivo não impede que um custo menor surja depois pela introdução de uma aresta negativa. Nessa circunstância, a escolha recai sobre Bellman-Ford, cuja lógica baseia-se em varrer todas as arestas repetidas vezes. A cada passagem, o algoritmo examina uma aresta e verifica se a distância até o vértice de destino pode ser melhorada pela rota que passa pela aresta de origem; quando a soma apresenta valor inferior à estimativa corrente, realiza-se o relaxamento. Se o grafo contém V vértices, repetir o processo V – 1 vezes assegura que qualquer caminho com até V – 1 arestas terá sido considerado, contemplando todos os caminhos simples possíveis.

Uma passagem adicional detecta a presença de ciclos de peso negativo, pois qualquer relaxamento ainda possível após V – 1 iterações indica a existência de ciclo no qual o custo total diminui indefinidamente. Em Python, listas de arestas e laços aninhados fornecem implementação direta, porém a complexidade temporal O(V E) demonstra-se mais elevada que a de Dijkstra. Essa característica torna Bellman-Ford apropriado apenas quando a presença potencial de pesos negativos justifica o custo adicional ou quando o número de arestas é baixo o suficiente para manter o tempo de execução aceitável.

A distinção conceitual entre os dois algoritmos sugere critérios de escolha em aplicações reais desenvolvidas em Python. Caso as arestas apresentem custos não negativos, Dijkstra oferece resposta rápida e escalável, aproveitando a eficiência de `heapq` para gerir prioridades. Se o modelo de dados admite valores negativos – por exemplo, créditos em transações financeiras ou ganhos de energia recuperada – Bellman-Ford assegura resultado correto e ainda permite verificar se tal característica compromete a integridade do sistema, pela detecção de ciclos que reduzem o custo total indefinidamente. Portanto, a compreensão dos requisitos específicos do grafo, aliada ao conhecimento das hipóteses de cada algoritmo, orienta a seleção acertada da técnica, culminando em rotas mínimas calculadas com precisão e desempenho adequado ao contexto.

Os avanços recentes em veículos autônomos e logística inteligente transformaram a maneira como mercadorias e passageiros se deslocam entre pontos geográficos. Frotas de robôs de entrega, caminhões sem motorista e automóveis de passeio equipados com sistemas de navegação inteligente já percorrem estradas e ruas de grandes centros urbanos, guiados por algoritmos que analisam em tempo real condições de tráfego, regulamentações de trânsito e disponibilidade de infraestrutura de recarga. Nesse contexto, o

planejamento de rotas deixou de ser mera busca do caminho mais curto: ele passou a incorporar múltiplos objetivos, como a minimização do tempo de viagem, do consumo de energia e da emissão de carbono, além de respeitar janelas de entrega e restrições legais de circulação.

A pressão por eficiência acontece em um panorama logístico marcado pela popularização do comércio eletrônico e pela expectativa de entregas no mesmo dia. Transportadoras que operam redes globais precisam decidir, minuto a minuto, como alocar veículos, recarregar baterias e sequenciar coletas, reduzindo atrasos sem ultrapassar limites operacionais. Veículos elétricos, por sua própria natureza, amplificam o desafio: se observarem rotas subótimas, poderão chegar ao destino com baixa autonomia ou demandar paradas imprevistas que comprometem o cronograma. A integração de heurísticas de caminho mínimo, alimentadas por dados de topografia e padrões de tráfego, tornou-se, portanto, requisito básico de qualquer sistema de navegação voltado a frotas autônomas.

Esse cenário atribui especial importância a algoritmos de grafos capazes de calcular caminhos mínimos ponderados por custo composto. Dijkstra oferece solução eficiente quando todos os pesos são não negativos, característica de métricas de tempo e energia. Bellman-Ford, embora mais custoso, estende o alcance a grafos que contêm pesos negativos, situação encontrada ao modelar recuperações de energia em descidas ou descontos de pedágio que reduzem o custo final.

Exemplo de aplicação

Exemplo 1 – Veículos autônomos e logística

Uma empresa de logística urbana opera dez veículos elétricos em uma malha de 15 pontos de interesse que incluem armazéns, lockers de retirada e estações de recarga rápida. Cada trecho de rua entre nós do grafo possui dois pesos: tempo estimado de percurso em segundos, derivado de dados históricos de tráfego, e energia consumida em quilowatt-hora, obtida por modelos que levam em conta inclinação do terreno e perfil de aceleração.

A política operacional determina que, ao receber um pedido de coleta com destino a um locker específico, o sistema escolha a rota que minimize primeiro o tempo total; em caso de empate, selecione o trajeto que consome menos energia.

A mesma política exige que, para rotas longas com queda acumulada de altitude, seja permitido representar trechos com energia negativa, simulando regeneração da bateria. O algoritmo deve detectar esse caso e, se houver ciclo de custo negativo no critério de energia, rejeitar a rota como inviável por risco de estimativa irrealista.

Essas regras direcionam a utilização de Dijkstra na maioria das consultas e de Bellman-Ford, com verificação de ciclo, quando a presença de pesos negativos for detectada.

O primeiro exercício desta unidade apresenta duas implementações em Python: uma versão de Dijkstra com fila de prioridade baseada em `heapq` e uma versão de Bellman-Ford que relaxa arestas

repetidamente. Embora bibliotecas como `NetworkX` forneçam versões otimizadas, a escrita manual deixa explícitas a escolha da estrutura de dados e a contagem de relaxamentos, aspectos fundamentais para calibrar desempenho em ambientes embarcados. A fila de prioridade de Dijkstra armazena tuplas cujo primeiro elemento é o custo acumulado em tempo; o custo em energia acompanha, permitindo o segundo critério de desempate sem comprometer a propriedade de extração no heap. O algoritmo encerra-se ao alcançar o destino, economizando processamento em comparação com a expansão completa da árvore de caminhos mínimos.

Bellman-Ford percorre todas as arestas durante |V|1 iterações, detectando redução de custo na passagem seguinte para sinalizar ciclo negativo. Para adaptar-se ao requisito de rejeitar rotas irrealistas, a implementação mantém dois vetores de custo: um para tempo e outro para energia; somente quando a nova aresta proporciona tempo não superior e energia inferior é que o relaxamento ocorre, garantindo prioridade de tempo. Após a fase principal, percorre as arestas mais uma vez; se encontrar redução no vetor de energia, lança exceção indicando presença de ciclo negativo nesse critério. A decisão de lançar erro em vez de corrigir o grafo obriga o usuário a revisar os dados de entrada, refletindo procedimentos de validação usados em pipelines de mapas.

Saiba mais

A obra a seguir aborda aplicações de Dijkstra e Bellman-Ford em redes de comunicação (roteamento em protocolos de internet), planejamento de rotas em logística e modelagem de sistemas com pesos negativos, como em análise de fluxo de redes. Ela também conecta os algoritmos a problemas reais, como otimização em sistemas de transporte e redes sociais.

KLEINBERG, J.; TARDOS, É. *Algorithm design*. Boston: Addison-Wesley Professional, 2005.

O código utiliza `dataclasses` para representar arestas e nós, aproveitando tipagem estática para evitar erros de parâmetro. A criação de um grafo leve, baseado em dicionários, favorece clareza sem incorrer em sobrecarga de orientação a objetos profundas. Além disso, a função de geração aleatória de malhas permite simular milhares de testes e estimar latência média dos algoritmos, abordagem comum quando equipes de navegação precisam dimensionar infraestrutura antes da implantação no veículo. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Dict, List, Tuple
import heapq, random

@dataclass
class Aresta:
    destino: str
    tempo: float      # segundos
    energia: float    # kWh (pode ser negativa em descida)
```

```
class Grafo:
    def __init__(self):
        self.adj: Dict[str, List[Aresta]] = {}

    def adicionar_aresta(self, origem: str, destino: str, tempo: float, energia:
float):
        self.adj.setdefault(origem, []).append(Aresta(destino, tempo, energia))

    # ---------- Dijkstra ----------
    def dijkstra(self, origem: str, alvo: str) -> Tuple[float, float, List[str]]:
        fila: List[Tuple[float, float, str]] = [(0.0, 0.0, origem)]
        visitado: Dict[str, Tuple[float, float]] = {origem: (0.0, 0.0)}
        ante: Dict[str, str] = {}

        while fila:
            tempo_atual, energia_atual, no = heapq.heappop(fila)
            if no == alvo:
                caminho = [alvo]
                while caminho[-1] != origem:
                    caminho.append(ante[caminho[-1]])
                return tempo_atual, energia_atual, list(reversed(caminho))
            for ar in self.adj.get(no, []):
                t = tempo_atual + ar.tempo
                e = energia_atual + ar.energia
                if ar.destino not in visitado or t < visitado[ar.destino][0]
or (
                        t == visitado[ar.destino][0] and e < visitado[ar.
destino][1]):
                    visitado[ar.destino] = (t, e)
                    ante[ar.destino] = no
                    heapq.heappush(fila, (t, e, ar.destino))
      raise ValueError("Alvo inalcançável")

    # ---------- BellmanFord ----------
    def bellman_ford(self, origem: str, alvo: str) -> Tuple[float, float,
List[str]]:
        # coleta todos os vértices, inclusive os que só aparecem como destino
        vertices = set(self.adj.keys())
        for edges in self.adj.values():
            for ar in edges:
                vertices.add(ar.destino)

        dist_tempo: Dict[str, float] = {v: float("inf") for v in vertices}
        dist_energia: Dict[str, float] = {v: float("inf") for v in vertices}
        ante: Dict[str, str] = {}
        dist_tempo[origem] = 0.0
        dist_energia[origem] = 0.0

        for _ in range(len(vertices) - 1):
            atualizado = False
            for u in vertices:
                for ar in self.adj.get(u, []):
                    if dist_tempo[u] == float("inf"):
                        continue
                    novo_t = dist_tempo[u] + ar.tempo
                    novo_e = dist_energia[u] + ar.energia
```

```
                        if novo_t < dist_tempo[ar.destino] or (
                            novo_t == dist_tempo[ar.destino] and novo_e < dist_
energia[ar.destino]):
                            dist_tempo[ar.destino] = novo_t
                            dist_energia[ar.destino] = novo_e
                            ante[ar.destino] = u
                            atualizado = True
            if not atualizado:
                break

        # verificação de ciclo de energia negativa
        for u in vertices:
            for ar in self.adj.get(u, []):
                if dist_tempo[u] == float("inf"):
                    continue
                if (dist_tempo[u] + ar.tempo == dist_tempo[ar.destino] and
                        dist_energia[u] + ar.energia < dist_energia[ar.destino]):
                    raise ValueError("Ciclo de energia negativa detectado")
            if dist_tempo[alvo] == float("inf"):
                raise ValueError("Alvo inalcançável")

        caminho = [alvo]
        while caminho[-1] != origem:
            caminho.append(ante[caminho[-1]])
        return dist_tempo[alvo], dist_energia[alvo], list(reversed(caminho))

# ---------- Gerador de malha ----------
def gerar_malha(g: Grafo, n: int, densidade: float):
    random.seed(7)
    pontos = [f"P{i}" for i in range(n)]
    for u in pontos:
        for v in pontos:
            if u != v and random.random() < densidade:
                tempo = random.uniform(50, 400)
                energia = random.uniform(0.2, 2.0)
                if random.random() < 0.05: # descida
                    energia *= -0.3
                g.adicionar_aresta(u, v, tempo, energia)

# ---------- Demonstração ----------
def principal():
    grafo = Grafo()
    gerar_malha(grafo, 15, 0.25)
    origem, destino = "P0", "P10"
    tem_negativo = any(ar.energia < 0 for edges in grafo.adj.values() for ar in
edges)

    try:
        if tem_negativo:
            tempo, energia, caminho = grafo.bellman_ford(origem, destino)
            print("Algoritmo BellmanFord aplicado devido a pesos negativos.")
        else:
            tempo, energia, caminho = grafo.dijkstra(origem, destino)
            print("Algoritmo Dijkstra aplicado (pesos não negativos).")
        print("Caminho:", " → ".join(caminho))
        print(f"Tempo total: {tempo:.1f} s, Energia: {energia:.2f} kWh")
```

```
    except ValueError as e:
      print("Falha no cálculo de rota:", e)

if __name__ == "__main__":
        principal()
```

O quadro 23 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 23 – Funções usadas no código-fonte do primeiro exercício prático

Função ou construção	Explicação de uso
heapq.heappush(heap, tupla)	Insere uma tupla no heap, em que os elementos são automaticamente ordenados com base no primeiro valor (tempo, no caso). Isso garante extração eficiente do próximo vértice a ser processado
heapq.heappop(heap)	Remove e retorna a tupla de menor valor do heap. Garante que sempre se processe o vértice com menor tempo acumulado no Dijkstra
defaultdict-like uso de dict. setdefault()	O método `self.adj.setdefault(origem, []).append(...)` adiciona uma aresta à lista de adjacência, inicializando a lista caso ainda não exista para aquele vértice
float('inf')	Representa o infinito positivo. Usado como valor inicial nas distâncias dos vértices, permitindo comparações corretas nas atualizações
any(condição for item in iterável)	Retorna `True` se alguma condição for verdadeira no iterável. Aqui, detecta se há alguma aresta com energia negativa
break em laço externo	Usado no Bellman-Ford para interromper as iterações quando nenhuma atualização de caminho é feita, o que indica convergência
raise ValueError("mensagem")	Lança explicitamente um erro com uma mensagem. No código, é usado para sinalizar que o alvo é inalcançável ou que há ciclo negativo de energia
list(reversed(lista))	Inverte uma lista, retornando uma nova. Usado para reconstruir o caminho do destino até a origem após o término da busca

As bibliotecas `dataclasses`, `typing`, `heapq` e `random` são usadas para definir estruturas de dados, gerenciar tipos, implementar filas de prioridade e gerar valores aleatórios, respectivamente. A classe `Aresta`, definida com o decorador `@dataclass`, representa uma aresta no grafo, armazenando o nó de destino (destino), o tempo necessário para atravessá-la (tempo) e o consumo ou ganho de energia (energia), que pode ser negativo em casos de descida.

A classe `Grafo` é o núcleo da implementação. Ela utiliza um dicionário `adj` para representar a lista de adjacência, na qual cada chave é um nó (string) e o valor é uma lista de arestas saindo desse nó. O método `adicionar_aresta` insere uma nova aresta no grafo, criando uma entrada no dicionário para o nó de origem, se necessário, e adicionando a aresta à lista correspondente. Essa estrutura permite representar grafos dirigidos com pesos duplos (tempo e energia).

O algoritmo de Dijkstra, implementado no método `dijkstra`, encontra o caminho de menor tempo entre um nó de origem e um alvo, considerando também a energia como critério secundário de desempate. Ele utiliza uma fila de prioridade (implementada com `heapq`) para explorar os nós, armazenando tuplas com o tempo acumulado, a energia acumulada e o nó atual. Um dicionário `visitado` registra o menor tempo e energia para alcançar cada nó, enquanto outro dicionário, `ante`, rastreia o nó predecessor no caminho. O algoritmo extrai o nó com menor tempo da fila, verifica se é o alvo e, se não, explora suas

arestas. Para cada aresta, calcula o novo tempo e energia acumulados e atualiza o caminho se o novo tempo for menor ou se o tempo for igual, mas a energia for menor. Se o alvo não for alcançado, uma exceção é levantada. O algoritmo assume que os pesos (tempo e energia) são não negativos, o que é garantido pela escolha do algoritmo no código principal.

O algoritmo de Bellman-Ford, implementado no método `bellman_ford`, é usado quando há pesos negativos (energia negativa em descidas). Ele suporta grafos com arestas de peso negativo e detecta ciclos de energia negativa. Primeiro, coleta todos os vértices do grafo, incluindo aqueles que aparecem apenas como destinos. Dois dicionários, `dist_tempo` e `dist_energia`, armazenam os menores tempos e energias para alcançar cada nó, inicializados com infinito, exceto para o nó de origem, que começa com zero. Um dicionário `ante` rastreia os predecessores. O algoritmo itera até `n-1` vezes (n é o número de vértices), relaxando todas as arestas: para cada aresta, atualiza o tempo e a energia do nó de destino se o novo tempo for menor ou se o tempo for igual, mas a energia for menor. Uma otimização interrompe o loop se nenhuma atualização ocorrer. Após o relaxamento, verifica se há ciclos de energia negativa, examinando se alguma aresta pode reduzir ainda mais a energia sem alterar o tempo. Se o alvo for inalcançável, uma exceção é levantada. O caminho é reconstruído a partir do dicionário `ante`.

A função `gerar_malha` cria um grafo aleatório com n nós, nomeados como `P0`, `P1`,`...`, `Pn-1`. A densidade do grafo é controlada pelo parâmetro densidade, que determina a probabilidade de existir uma aresta entre dois nós distintos. Para cada aresta criada, o tempo é gerado aleatoriamente entre 50 e 400 segundos, e a energia, entre 0,2 e 2,0 kWh. Com 5% de chance, a energia é multiplicada por `-0`, `3`, simulando uma descida que gera energia. A semente aleatória é fixada `random.seed(7)` para reprodutibilidade.

A função `principal` demonstra o uso do código. Cria um grafo com 15 nós e densidade `0`, `25`, seleciona `P0` como origem e `P10` como destino, e verifica se há arestas com energia negativa. Se houver, usa Bellman-Ford; caso contrário, usa Dijkstra. O resultado (tempo, energia e caminho) é impresso ou uma mensagem de erro é exibida se o cálculo falhar. A escolha entre os algoritmos reflete suas propriedades: Dijkstra é mais eficiente para pesos não negativos, enquanto Bellman-Ford lida com pesos negativos e detecta ciclos negativos.

O código é robusto, com tratamento de erros para alvos inalcançáveis e ciclos de energia negativa, e utiliza boas práticas, como tipagem explícita e modularização. A implementação é eficiente para grafos de tamanho moderado, embora a geração de malhas densas possa aumentar o custo computacional. A flexibilidade para lidar com dois pesos (tempo e energia) torna o código aplicável a problemas reais, como roteamento em redes de transporte com consumo ou regeneração de energia.

A seguir, constam sugestões de melhoria para que você implemente no futuro.

- Uma ampliação natural envolveria acrescentar restrições de recarga, inserindo nós que representem estações e penalidades de tempo para reabastecimento.

- Outra evolução consistiria em empregar algoritmo A* com heurística de distância euclidiana, melhorando tempo de execução em grafos extensos.

- Em cenários de malha dinâmica, seria útil incorporar atualizações de peso em tempo real e reexecutar Dijkstra com fila Fibonacci ou técnicas de incremental search para ajustar rotas sem reiniciar o cálculo.

- Por fim, exportar o grafo em formato GeoJSON e integrar visualização em mapas web permitiria analisar o comportamento dos trajetos em ambiente espacial, aproximando o exercício de painéis operacionais usados por frotas autônomas comerciais.

7.2 Grafos com árvores: algoritmos de Kruskal e Prim para árvores geradoras mínimas

A construção de árvores geradoras mínimas em grafos ponderados estabelece um procedimento que seleciona subconjunto de arestas com custo total mínimo sem violar a conectividade global do conjunto de vértices. Sob essa perspectiva, os algoritmos de Kruskal e Prim emergem como estratégias consagradas, cada qual explorando princípios distintos de seleção progressiva ao mesmo objetivo.

Kruskal inicia com floresta formada por vértices isolados, ordena todas as arestas pelo peso e percorre essa lista crescente, acrescentando sucessivamente arestas que unem componentes desconexos. A decisão de inclusão baseia-se na verificação de que a nova aresta não introduz ciclo, operação implementada com estrutura união-busca equipada com compressão de caminho e união por tamanho, capazes de manter a identificação de componentes em tempo quase constante. Em consequência, a execução percorre E arestas e realiza buscas em subárvore de complexidade praticamente inversa do crescimento de Ackermann, conduzindo a custo final O(E log E) devido à ordenação inicial. A cada inclusão, o número de componentes diminui, até que apenas um componente permaneça, no qual o conjunto resultante apresenta exatamente V – 1 arestas e custo mínimo garantido pela propriedade do corte: quando o conjunto de arestas já escolhidas não contém ciclo, qualquer aresta de menor peso que atravesse partição entre componentes preserva a condição de minimalidade.

Observação

O termo "floresta" aparece porque o algoritmo de Kruskal, embora usado para encontrar uma árvore geradora mínima (um tipo de árvore), começa com uma floresta, que é um grafo formado por várias componentes desconexas, cada uma sendo uma árvore (inicialmente, cada vértice é uma árvore isolada). Em outras palavras, o termo é usado para descrever o estado inicial e intermediário do algoritmo, enquanto o resultado final é uma árvore. A confusão surge porque floresta é um conceito mais geral, que inclui várias árvores e Kruskal manipula essas árvores até formar uma única.

Já Ackermann é uma função matemática recursiva que cresce extremamente rápido, usada em teoria da computação para ilustrar funções recursivas que não são primitivamente recursivas.

Prim adota perspectiva alternativa; em vez de unir componentes dispersas, expande um componente único começando por vértice arbitrário. A fronteira inicial contém todas as arestas que partem desse vértice, armazenadas em fila de prioridade. Em cada iteração extrai-se a aresta de menor peso que conecta o componente já formado a um vértice externo. Tal vértice é incorporado e as arestas incidentes a partir dele ingressam na fila se conduzirem a vértices ainda não visitados. A fila de prioridade implementada com heap binário, provido pelo módulo `heapq` em Python, assegura remoções do mínimo e inserções em tempo O(log V). Como cada vértice ingressa na árvore exatamente uma vez e cada aresta é considerada no máximo quando seu vértice final permanece fora do componente, o custo total situa-se em O(E log V). Quando o grafo é denso, substituir o heap por matriz de chaves e buscar o mínimo em varredura linear conduz a tempo $O(V^2)$, conferindo vantagem a Prim sobre Kruskal na ausência de estrutura de dados esparsa.

Embora ambos produzam árvore geradora mínima idêntica para grafos sem pesos repetidos, fatores como densidade do grafo, volume das arestas e facilidade de paralelização influenciam a escolha prática. Kruskal, ao ordenar arestas previamente, paraleliza de forma natural em ambientes que dividem blocos de arestas, ao passo que Prim apresenta comportamento incremental sensível à heurística de adiamento de arestas internas. Em sistemas distribuídos que manipulam grafos gigantescos, Kruskal costuma associar-se ao paradigma MapReduce, enquanto Prim atende roteadores embarcados nos quais o armazenamento das arestas excede pouco o número de vértices. A compreensão detalhada desses algoritmos fornece ao projetista de sistemas capacidade de adaptar a seleção ao perfil específico de entrada, garantindo que a rede resultante minimize custos de infraestrutura sem renunciar ao requisito fundamental de conectividade global.

Para diferenciar Prim e Kruskal usando a mesma analogia da fábrica que produz peças metálicas, adaptaremos a ideia. Aqui, o objetivo não é encontrar o caminho mais curto de uma máquina inicial (como em Dijkstra ou Bellman-Ford), mas conectar todas as máquinas da fábrica com tubos ou esteiras de transporte, gastando o menor custo possível, sem formar ciclos (loops). Esses tubos/esteiras representam a árvore geradora mínima, que é o que Prim e Kruskal calculam. Usaremos o mesmo conjunto de equipamentos (prensa, moinho, forno, cortadora, lixadeira e acabamento) e supor que cada conexão entre máquinas tem um custo (unidades monetárias, tempo ou energia).

Exemplo de aplicação

Conectar todas as máquinas da fábrica

Imagine que você é o gerente da fábrica e precisa instalar tubos ou esteiras para transportar materiais entre todas as máquinas (prensa, moinho, forno, cortadora, lixadeira e acabamento), garantindo que:

- Todas as máquinas estejam conectadas (o material pode chegar a qualquer máquina).
- O custo total de instalação seja o menor possível.
- Não haja ciclos (não queremos tubos desnecessários que formem loops, pois isso aumenta o custo sem benefício).

Na figura a seguir, o grafo ponderado (como na figura 20) mostra os custos das conexões entre as máquinas:

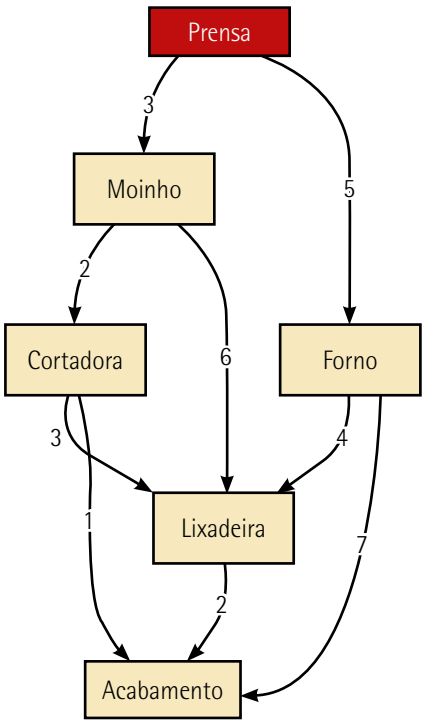

Figura 21 – Equipamentos e custos de conexão: grafo ponderado

Agora, mostraremos como Prim e Kruskal abordam esse problema de forma diferente.

Algoritmo de Prim: "expandir a partir de uma máquina"

Como funciona: você começa em uma máquina qualquer (digamos, a prensa) e instala a conexão mais barata para outra máquina que ainda não está conectada. Depois, olha todas as conexões possíveis a partir das máquinas já conectadas e escolhe a mais barata para uma nova máquina. Você continua assim, crescendo a rede de tubos até todas as máquinas estarem conectadas.

- **Passo 1**: comece na prensa. As opções são:
  - Prensa → moinho: 3 unidades
  - Prensa → forno: 5 unidades
    - Prim escolhe prensa → moinho (custo 3, mais barato)
- **Passo 2**: agora, as máquinas conectadas são prensa e moinho. Olhe as conexões disponíveis a partir delas:
  - De prensa: forno (5)

– De moinho: cortadora (2), lixadeira (6)

- Prim escolhe moinho → cortadora (custo 2, mais barato)

- **Passo 3**: máquinas conectadas: prensa, moinho e cortadora. Conexões disponíveis:

  – De prensa: forno (5)

  – De moinho: lixadeira (6)

  – De cortadora: lixadeira (3), acabamento (1)

  – Prim escolhe cortadora → acabamento (custo 1, mais barato)

- **Passo 4**: máquinas conectadas: prensa, moinho, cortadora e acabamento. Conexões disponíveis:

  – De prensa: forno (5)

  – De moinho: lixadeira (6)

  – De cortadora: lixadeira (3)

  – De acabamento: lixadeira (2)

  – Prim escolhe acabamento → lixadeira (custo 2, mais barato)

- **Passo 5**: máquinas conectadas: prensa, moinho, cortadora, acabamento e lixadeira. Conexões disponíveis:

  – De prensa: forno (5)

  – Prim escolhe prensa → forno (custo 5)

- Resultado: todas as máquinas estão conectadas. As conexões escolhidas são:

  – Prensa → moinho (3)

  – Moinho → cortadora (2)

  – Cortadora → acabamento (1)

  – Acabamento → lixadeira (2)

  – Prensa → forno (5)

– Custo total: 3 + 2 + 1 + 2 + 5 = 13 unidades

- Característica principal: Prim cresce a rede a partir de uma máquina inicial, sempre escolhendo a conexão mais barata para uma máquina ainda não conectada, como se estivesse expandindo um sistema de tubos localmente.

Algoritmo de Kruskal: "escolher as conexões mais baratas primeiro"

Como funciona: você olha todas as conexões possíveis entre as máquinas e começa instalando a mais barata, não importa onde ela esteja. Depois, escolhe a próxima mais barata, mas apenas a instala se ela não criar um ciclo (um loop entre máquinas já conectadas). Continua assim até todas as máquinas estarem conectadas.

- **Passo 1**: liste todas as conexões em ordem crescente de custo:

  – Cortadora → acabamento: 1

  – Moinho → cortadora: 2

  – Acabamento → lixadeira: 2

  – Prensa → moinho: 3

  – Cortadora → lixadeira: 3

  – Forno → lixadeira: 4

  – Prensa → forno: 5

  – Moinho → lixadeira: 6

  – Forno → acabamento: 7

  - Kruskal escolhe cortadora → acabamento (custo 1, mais barato)

- **Passo 2**: próxima conexão mais barata: moinho → cortadora (custo 2). Não cria ciclo (moinho não está conectado à cortadora ou ao acabamento ainda). Escolhida.

- **Passo 3**: próxima: acabamento → lixadeira (custo 2). Não cria ciclo (lixadeira não está conectada ainda). Escolhida.

- **Passo 4**: próxima: prensa → moinho (custo 3). Não cria ciclo (prensa não está conectada). Escolhida.

- **Passo 5**: próxima: cortadora → lixadeira (custo 3). Não escolhida, pois cortadora e lixadeira já estão conectadas (via acabamento) e isso criaria um ciclo.

- **Passo 6**: próxima: forno → lixadeira (custo 4). Não cria ciclo (forno não está conectado). Escolhida.

- Resultado: todas as máquinas estão conectadas. As conexões escolhidas são:

  – Cortadora → acabamento (1)

  – Moinho → cortadora (2)

  – Acabamento → lixadeira (2)

  – Prensa → moinho (3)

  – Forno → lixadeira (4)

  – Custo total: 1 + 2 + 2 + 3 + 4 = 12 unidades

- Característica principal: Kruskal escolhe as conexões mais baratas de todo o grafo, independentemente de onde estão, mas verifica se elas conectam máquinas que ainda não estão ligadas, evitando ciclos.

Para visualizar melhor esses conceitos, faremos um exercício prático. A arquitetura de sistemas distribuídos tornou-se a espinha dorsal de aplicações em nuvem, plataformas de streaming e serviços bancários digitais, pois permite escalar componentes de forma independente, tolerar falhas e entregar atualizações contínuas. Com o avanço do modelo de microsserviços, cada funcionalidade é encapsulada em um serviço leve que interage com dezenas de outros por meio de chamadas de rede. Essa granularidade traz flexibilidade, mas expõe a aplicação a uma nova ordem de complexidade: o tempo de resposta global passa a depender da topologia de conexões e dos custos de comunicação entre contêineres, pods ou nós físicos espalhados por data centers distintos. Em ambientes nos quais centenas de microsserviços competem por links de rede limitados, otimizar o encadeamento dessas relações torna-se requisito para garantir latência previsível.

Ao considerar que cada serviço estabelece canais de RPC (Remote Procedure Call), filas de mensagens ou conexões a bancos de dados, surge naturalmente um grafo no qual os nós representam os microsserviços e as arestas simbolizam canais de comunicação com pesos ligados à latência média, vazão ou custo de transferência de dados. A construção de uma malha eficiente deve equilibrar redundância, necessária para resiliência, e contenção de custos, indispensável para manter contas de nuvem sob controle. Árvores geradoras mínimas oferecem um ponto de partida matematicamente sólido: elas unem todos os nós do grafo com o menor peso total possível, removendo enlaces supérfluos que somente aumentariam o gasto sem reduzir distância significativa. Embora sistemas reais exijam múltiplos caminhos para tolerar falhas, partir de uma árvore mínima ajuda arquitetos a identificar a espinha dorsal essencial e acrescentar redundância de forma informada.

Em um cenário em que instâncias são continuamente criadas e destruídas por orquestradores como Kubernetes, algoritmos de construção de árvore geradora – notadamente Kruskal e Prim – precisam de versões rápidas e fáceis de prototipar. Python, graças à sua sintaxe expressiva e às bibliotecas nativas de filas de prioridade e estruturas de conjuntos, oferece terreno fértil para modelar essas topologias, validar hipóteses de ligação e comparar custos antes que a equipe de infraestrutura aplique as mudanças em ambiente de produção.

Exemplo de aplicação

Exemplo 2 – Sistemas distribuídos

Considere uma plataforma de comércio eletrônico que executa quarenta microsserviços distribuídos entre três regiões de nuvem. Cada serviço possui dependências explícitas definidas por sua função de negócio; por exemplo, o serviço de carrinho precisa consultar "catálogo" e "preços", enquanto "pagamentos" interage com "fraude" e "contabilidade".

Análises de telemetria revelaram que a soma dos tempos de ida-e-volta entre pares de serviços consome parcela excessiva da latência percebida pelo usuário. A equipe de operações estipula, então, que o conjunto de links diretos deverá ser revisto: quer-se manter um caminho de comunicação entre todos os serviços com custo total mínimo medido em milissegundos médios.

Após aplicar essa redução, engenheiros acrescentarão rotas extras apenas onde a disponibilidade histórica justificar. O problema prático resume-se a receber o grafo ponderado de latências, computar uma árvore geradora mínima e exibir o custo somado, permitindo comparação com a malha original.

A solução em Python emprega duas técnicas padrão da literatura de grafos. Kruskal percorre todas as arestas em ordem crescente de peso e une componentes desconexos por meio de uma estrutura de união-busca (disjoint set), garantindo que nenhum ciclo seja formado; a implementação recorre a um dicionário de pais e alturas para realizar união por ranking, aproximando o custo amortizado de quase constante. Prim, por sua vez, parte de um nó arbitrário e expande uma fronteira de vértices visitados, escolhendo sempre a aresta mais barata que conecta o conjunto interno ao externo; o módulo heapq provê a fila de prioridade necessária para retirar o próximo enlace mínimo em O(log m). Embora ambos algoritmos produzam árvore de mesmo custo, Kruskal costuma destacar-se em grafos esparsos, enquanto Prim exibe desempenho superior em grafos densos se aliado a uma fila de Fibonacci. A versão aqui apresentada prioriza simplicidade e ilustra que, para um conjunto de quarenta nós, as diferenças são irrelevantes para validação conceitual.

Os dados de entrada são descritos em um arquivo CSV com colunas `origem`, `destino` e `latencia_ms`, representando as latências médias medidas entre cada par de microsserviços. A classe `GrafoServicos` carrega esse material em um dicionário que associa cada vértice à lista de suas arestas. As duas funções de cálculo de árvore geradora retornam o conjunto de arestas pertencentes à solução e ao custo total, possibilitando que o time visualize a economia potencial. `Dataclasses` modelam tanto o vértice quanto a aresta, conferindo tipagem explícita sem sacrificar legibilidade.

Para enfatizar a aplicação prática, o script inclui um gerador aleatório de malhas que usa parâmetros de densidade para simular diferentes cenários de comunicação, além de pequenos trechos de código que mostram como medir tempo de execução. Desse modo, arquitetos podem ajustar a densidade e avaliar rapidamente o impacto no custo total da árvore mínima, elaborando argumentos quantificados em reuniões de design. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
import csv, random, time
from dataclasses import dataclass
from typing import Dict, List, Tuple, Set
import heapq

@dataclass(frozen=True)
class Aresta:
    origem: str
    destino: str
    peso: float

class GrafoServicos:
    def __init__(self):
        self.adj: Dict[str, List[Aresta]] = {}
        self.arestas: List[Aresta] = []

    def adicionar_aresta(self, origem: str, destino: str, peso: float):
        ar = Aresta(origem, destino, peso)
        self.arestas.append(ar)
        self.adj.setdefault(origem, []).append(ar)
        self.adj.setdefault(destino, []).append(Aresta(destino, origem, peso))

    def carregar_csv(self, caminho: str):
        with open(caminho, newline="") as arq:
          leitor = csv.DictReader(arq)
          for linha in leitor:
              self.adicionar_aresta(linha["origem"], linha["destino"],
float(linha["latencia_ms"]))

    # ---------- Kruskal ----------
    def arvore_minima_kruskal(self) -> Tuple[List[Aresta], float]:
        parent: Dict[str, str] = {}
        rank: Dict[str, int] = {}
        def achar(u):
            while parent[u] != u:
                parent[u] = parent[parent[u]]
                u = parent[u]
            return u
        def unir(u, v):
            ru, rv = achar(u), achar(v)
            if ru == rv:
                return False
            if rank[ru] < rank[rv]:
                parent[ru] = rv
            elif rank[ru] > rank[rv]:
                parent[rv] = ru
            else:
```

```
                    parent[rv] = ru
                    rank[ru] += 1
               return True
          for no in self.adj:
               parent[no] = no
               rank[no] = 0
          arestas_ord = sorted(self.arestas, key=lambda a: a.peso)
          resultado: List[Aresta] = []
          custo = 0.0
          for ar in arestas_ord:
               if unir(ar.origem, ar.destino):
                    resultado.append(ar)
                    custo += ar.peso
          return resultado, custo

     # ---------- Prim ----------
     def arvore_minima_prim(self, inicio: str) -> Tuple[List[Aresta], float]:
          visitado: Set[str] = set([inicio])
          heap: List[Tuple[float, str, str]] = []
          resultado: List[Aresta] = []
          custo = 0.0
          for ar in self.adj[inicio]:
               heapq.heappush(heap, (ar.peso, ar.origem, ar.destino))
          while heap and len(visitado) < len(self.adj):
               peso, u, v = heapq.heappop(heap)
               if v in visitado:
                    continue
               visitado.add(v)
               resultado.append(Aresta(u, v, peso))
               custo += peso
               for ar in self.adj[v]:
                    if ar.destino not in visitado:
                         heapq.heappush(heap, (ar.peso, ar.origem, ar.destino))
          return resultado, custo

# --------- Gerador de grafo simulado ----------
def gerar_grafo(n_servicos: int, densidade: float) -> GrafoServicos:
     random.seed(11)
     nomes = [f"S{i:02d}" for i in range(n_servicos)]
     g = GrafoServicos()
     for i, u in enumerate(nomes):
          for v in nomes[i+1:]:
              if random.random() < densidade:
                  lat = random.uniform(1.0, 20.0)
                  g.adicionar_aresta(u, v, lat)
     return g

# --------- Demonstração ----------
def principal():
     grafo = gerar_grafo(40, 0.35)
     t0 = time.perf_counter()
     arv1, custo1 = grafo.arvore_minima_kruskal()
     kr = time.perf_counter() - t0
     t1 = time.perf_counter()
     arv2, custo2 = grafo.arvore_minima_prim("S00")
     pr = time.perf_counter() - t1
```

```
    print(f"Árvore mínima por Kruskal: custo {custo1:.2f} ms, gerada em
{kr*1000:.1f} ms")
    print(f"Árvore mínima por Prim: custo {custo2:.2f} ms, gerada em
{pr*1000:.1f} ms")
    print("\nPrimeiras dez arestas da solução de Kruskal:")
    for ar in arv1[:10]:
        print(f"{ar.origem} ↔ {ar.destino}: {ar.peso:.2f} ms")

if __name__ == "__main__":
        principal()
```

O quadro 24 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 24 – Funções usadas no código-fonte do segundo exercício prático

Função ou construção	Explicação de uso
@dataclass(frozen=True)	Cria uma classe imutável cujas instâncias podem ser usadas como chaves em conjuntos ou dicionários. Garante integridade dos dados da aresta
heapq.heappush(heap, tupla)	Insere uma tupla no heap, mantendo-o ordenado. No algoritmo de Prim, é usado para priorizar arestas com menor peso
heapq.heappop(heap)	Remove e retorna a tupla de menor valor (menor peso da aresta) do heap. Permite expandir o grafo com a próxima aresta mais barata
set([elemento])	Cria um conjunto com elementos únicos. No código, é usado para rastrear os vértices já incluídos na árvore mínima
dicionários de união e busca (parent, rank)	Técnica conhecida como Union-Find com compressão de caminho e união por rank. Essencial para detectar ciclos de forma eficiente no Kruskal
lambda a: a.peso	Função anônima usada como chave para ordenar as arestas pelo atributo peso
sorted(lista, key=...)	Ordena a lista de arestas com base na latência. Necessário para que o Kruskal processe as arestas do menor para o maior peso
DictReader (csv.DictReader)	Embora presente, o carregamento de CSV é opcional e segue padrão já tratado antes: lê linhas como dicionários para extrair colunas por nome

No código-fonte, os módulos `csv`, `random` e `time` são usados para leitura de arquivos, geração de números aleatórios e medição de tempo, respectivamente. O módulo `dataclasses` simplifica a criação de classes imutáveis, enquanto `typing` fornece tipos como `Dict`, `List`, `Tuple` e `Set` para anotações precisas. Por fim, `heapq` é utilizado para implementar uma fila de prioridade no algoritmo de Prim.

A classe `Aresta`, definida com o decorador `@dataclass(frozen=True)`, representa uma aresta imutável do grafo, com três atributos: `origem` (nó inicial), `destino` (nó final) e `peso` (latência em milissegundos, representada como `float`). A imutabilidade garante que objetos dessa classe não sejam modificados após a criação, promovendo segurança em operações subsequentes.

A classe `GrafoServicos` é o núcleo da implementação, modelando um grafo não direcionado com representação por lista de adjacência. Seu construtor inicializa dois atributos: `adj`, um dicionário que mapeia cada nó a uma lista de arestas incidentes, e `arestas`, uma lista que armazena todas as arestas do grafo. O método `adicionar_aresta` insere uma aresta bidirecional no grafo, criando um objeto `Aresta` e atualizando tanto o dicionário `adj` quanto a lista arestas. Para garantir a bidirecionalidade,

uma aresta inversa é adicionada ao nó destino, refletindo a natureza não direcionada do grafo. O método `carregar_csv` permite a leitura de dados de um arquivo CSV, esperando colunas nomeadas origem, destino e `latencia_ms`. Cada linha do arquivo é processada para adicionar uma aresta ao grafo, com a latência convertida para float.

O algoritmo de Kruskal, implementado no método `arvore_minima_kruskal`, encontra a árvore geradora mínima (AGM) ordenando todas as arestas por peso e selecionando-as de forma a evitar ciclos. Para isso, utiliza a estrutura `Union-Find` com otimizações de compressão de caminho e união por rank. A função interna `achar` localiza a raiz de um nó, aplicando compressão de caminho para melhorar a eficiência. A função `unir` junta dois conjuntos, utilizando o rank para manter a árvore balanceada. As arestas são processadas em ordem crescente de peso e, se a união dos nós de uma aresta for bem-sucedida (ou seja, não formar um ciclo), a aresta é adicionada à AGM e seu peso é somado ao custo total. O método retorna uma tupla contendo a lista de arestas da AGM e o custo total.

Observação

O Union-Find é uma estrutura de dados utilizada para gerenciar conjuntos disjuntos de forma eficiente, com operações para unir conjuntos e verificar se dois elementos pertencem ao mesmo conjunto. No contexto do algoritmo de Kruskal, descrito no texto, ele é usado para detectar e evitar ciclos ao construir a árvore geradora mínima (AGM).

O algoritmo de Prim, implementado em `arvore_minima_prim`, também calcula a AGM, mas utiliza uma abordagem baseada em crescimento a partir de um nó inicial. Ele mantém um conjunto de nós visitados e uma fila de prioridade (implementada com `heapq`) para selecionar a aresta de menor peso que conecta um nó visitado a um não visitado. A cada iteração, a aresta de menor peso é retirada da fila e, se seu nó destino ainda não foi visitado, ela é adicionada à AGM, o nó destino é marcado como visitado e as arestas incidentes a esse nó (que levam a nós não visitados) são inseridas na fila. O processo continua até que todos os nós sejam visitados ou a fila esteja vazia. Assim como no método de Kruskal, o resultado é uma tupla com a lista de arestas e o custo total.

A função `gerar_grafo` cria um grafo simulado com um número especificado de serviços (nós) e uma densidade que controla a probabilidade de existência de arestas entre pares de nós. Os nós são nomeados como `S00`, `S01` etc., e as arestas recebem pesos aleatórios entre 1.0 e 20.0 milissegundos. A semente fixa `random.seed(11)` garante resultados reprodutíveis. Essa função é útil para testes, simulando redes de serviços com latências variadas.

A função principal demonstra o uso da implementação, gerando um grafo com 40 nós e densidade de 0.35. Ela executa os algoritmos de Kruskal e Prim, mede o tempo de execução de cada um com `time.perf_counter` e exibe os resultados, incluindo o custo total das AGMs e o tempo de processamento em milissegundos. Além disso, imprime as primeiras dez arestas da solução de Kruskal, mostrando os nós conectados e suas respectivas latências.

O código é executado diretamente se o módulo for o `principal` (`if __name__ == "__main__"`), chamando a função `principal`. Sua estrutura é robusta, com tipagem explícita e uso de estruturas de dados eficientes, como dicionários para listas de adjacência e filas de prioridade para o algoritmo de Prim. A implementação dos algoritmos é otimizada, com complexidades temporais de O(E log E) para Kruskal (devido à ordenação das arestas) e O(E log V) para Prim (devido às operações na fila de prioridade), em que E é o número de arestas e V é o número de vértices. A modularidade e a clareza do código facilitam sua manutenção e extensão para outros contextos de grafos. A seguir, constam sugestões de melhoria para que você implemente no futuro.

- Uma ampliação natural consistiria em atribuir segundo peso de custo, como tráfego mensal em gigabytes, e aplicar algoritmo de árvore geradora mínima multiobjetivo com otimização por escalarização ou método de Pareto, refletindo cobranças diferenciadas de provedores de nuvem em horário de pico.

- Outra evolução poderia incorporar remoção incremental de nós, simulando falhas de zona de disponibilidade, e medir a robustez da espinha dorsal resultante.

- Para ambientes extremamente grandes, substituir a fila de prioridade de Prim por heap de Fibonacci ou pairing heap reduziria a complexidade assintótica, enquanto a off-loading da ordenação de Kruskal para `numpy.argsort` aceleraria o pré-processamento em grafos de milhares de arestas.

- Finalmente, exportar a árvore mínima para formato DOT e renderizar em Graphviz facilitaria discussões de arquitetura com times multidisciplinares, elevando o exercício a ferramenta de design colaborativo.

8 APLICAÇÕES PRÁTICAS E PROBLEMAS CLÁSSICOS

A aprendizagem de algoritmos em Python conduz à percepção de que padronizar raciocínios em torno de paradigmas fortalece tanto a clareza do código quanto a eficiência de execução. A estratégia de divisão e conquista ilustra esse princípio ao fragmentar o problema em subpartes independentes, resolver cada fragmento de forma recursiva e, por fim, recombinar as respostas.

O caso da mochila binária destaca essa metodologia quando se trata de selecionar subconjunto ótimo de itens em capacidade limitada. Ao dividir o conjunto de objetos em duas metades, calcula-se o ganho máximo de cada subconjunto independente e, na etapa de fusão, examina-se a possibilidade de combinar lucros sem exceder o peso permitido. Embora o custo total permaneça exponencial, esse arranjo divide o espaço de busca e facilita paralelização em Python por meio de multiprocessamento ou bibliotecas de computação distribuída. Além disso, o fluxo de chamadas recursivas torna-se claro mediante funções puras, pois cada ramo recebe subconjunto próprio de itens e retorna resultado isolado.

Esse mesmo raciocínio se aplica a algoritmos clássicos de ordenação como Merge Sort, reforçando a ideia de que a decomposição estruturada permite aproveitar caches de processador e reduzir latência de memória, aspectos relevantes em ambientes que tratam lotes extensos de dados financeiros ou logs de telemetria.

Lembrete

Vimos, em detalhes, o algoritmo Merge Sort no tópico 4.

Quando um problema exibe subestruturas que se repetem, a divisão ingênua conduz a recomputação redundante. Nesse cenário emerge a programação dinâmica, técnica que armazena soluções intermediárias para que cada subproblema seja tratado apenas uma vez. Na subsequência comum mais longa, procura-se o maior arranjo de símbolos que aparece, na mesma ordem, em duas cadeias. O procedimento matricial preenche tabela bidimensional em que cada posição contém comprimento do melhor alinhamento prefixo a prefixo. A transição depende de verificar se o símbolo atual coincide: quando aparece correspondência, soma-se um à célula diagonal anterior; do contrário, utiliza-se o valor máximo entre célula superior e célula esquerda. Python favorece tal abordagem mediante listas aninhadas ou arrays da biblioteca NumPy e o ritmo previsível de acesso sequencial garante bom aproveitamento de cache mesmo em instâncias com dezenas de milhares de caracteres, situação típica em bioinformática quando se comparam genomas ou em processamento de linguagem ao alinhar versões de documentos.

A grande força da programação dinâmica reside em transformar problemas de natureza combinatória, originalmente exponenciais, em rotinas polinomiais. O próprio caso da mochila, ao ser reorientado para esse paradigma, ganha algoritmo que percorre matriz de dimensões capacidade × número de itens. Durante o preenchimento, cada decisão compara incluir ou excluir o item atual, reutilizando valores calculados anteriormente e produzindo complexidade $O(n\,C)$, viável em muitas aplicações empresariais nas quais capacidades não ultrapassam alguns milhares. Esse raciocínio propaga-se a desafios como alinhamento de sequências múltiplas, cálculo de caminhos mínimos em grafos com pesos não negativos ou determinação de corte de hastes de aço que maximiza receita, todos beneficiados por cache de subsoluções.

Figura 22 – Representação artística do problema da mochila: Como guardar diversos itens? Imagem produzida pelo autor com a ferramenta de inteligência artificial generativa Grok 3

Em termos práticos, profissionais encontram na divisão e na conquista solução apropriada quando subproblemas permanecem independentes e fusão apresenta custo relativamente baixo, enquanto recorrem à programação dinâmica quando a sobreposição de subestruturas exige memorização para evitar redundância. Python, devido ao suporte a geradores, compreensões de listas e dicionários, fornece ferramentas diretas para concretizar ambos os paradigmas, seja através de recursão explícita, seja empregando iteração controlada que preenche tabelas de maneira ascendente. A escolha consciente entre dividir para conquistar ou armazenar subsoluções define a viabilidade de aplicações que precisam entregar resultados com prazos rígidos, como recomendadores em tempo real, planejamento de produção industrial ou análise de grandes cadeias genéticas.

8.1 Problemas de divisão e conquista: análise de exemplos como o problema da mochila

A estratégia de divisão e conquista estabelece um percurso lógico no qual um problema complexo é fracionado em partes menores, cada uma tratada independentemente, e, por fim, as respostas parciais são combinadas para formar a solução global.

Tal paradigma mostra-se particularmente útil quando cada subinstância mantém estrutura semelhante à da questão original, permitindo reutilizar o mesmo raciocínio recursivo. O problema da mochila, ainda que tradicionalmente associado à programação dinâmica, fornece exemplo esclarecedor do modo como a divisão e a conquista oferecem ganhos práticos em circunstâncias específicas, especialmente quando o número de itens permanece moderado e a capacidade da mochila assume valores elevados.

Figura 23 – Representação artística do problema da mochila: arrajo de itens. Imagem produzida pelo autor com a ferramenta de inteligência artificial generativa Grok 3

Considera-se uma coleção de objetos, cada qual caracterizado por peso e valor, uma mochila com limite de carga. O objetivo consiste em escolher subconjunto cujo valor total seja máximo sem exceder o peso disponível. Uma aplicação direta de divisão e conquista parte da observação de que o conjunto inicial de n objetos pode ser dividido em dois blocos de tamanhos semelhantes.

Para cada bloco calculam-se, por força bruta, todos os subconjuntos possíveis, armazenando para cada peso atingido o valor máximo correspondente. Esse exame gera duas listas, uma para cada metade. No passo subsequente ordena-se uma das listas por peso e, para valores repetidos, mantém-se apenas a melhor configuração, processo que elimina redundância. A fusão ocorre combinando, por busca binária, cada entrada da primeira lista com a melhor opção compatível na segunda, de forma a não ultrapassar a capacidade. A complexidade resulta em $O(2^{n/2})$ para gerar os subconjuntos de cada metade e $O(2^{n/2} \log 2^{n/2})$ para a fusão, desempenho que supera a tentativa exaustiva $O(2^n)$ quando n encontra-se na faixa de cinquenta a sessenta elementos.

Observação

No contexto do texto, o termo força bruta refere-se a uma abordagem algorítmica que resolve um problema explorando todas as possibilidades ou combinações possíveis de forma exaustiva, sem otimizações sofisticadas. Especificamente, na frase mencionada, está relacionado ao cálculo de todos os subconjuntos possíveis de um bloco de itens no problema da mochila binária.

Além da redução de tempo, o esquema ilustrado demonstra como a divisão inicial viabiliza paralelismo natural. Cada processador, núcleo ou execução distribuída em cluster pode calcular autonomamente o mapa de pesos e valores da sua metade, necessitando apenas compartilhar as listas finais para produção da combinação ótima. Em Python, tal independência harmoniza-se com bibliotecas de multiprocessamento, que evitam travamentos de memória, pois cada processo trabalha com subconjunto contido de dados. A fusão, realizada após a sincronização, beneficia-se de estruturas ordenadas, como arrays NumPy ou listas comuns após aplicação de função sort, resultando em busca altamente eficiente pelos pares compatíveis.

O paradigma ainda apoia variações nas quais a divisão não ocorre somente em dois blocos. Sistemas de recomendação que precisam avaliar combinações de produtos, por exemplo, podem fracionar o inventário em diversos lotes, calcular localmente as melhores composições e, em seguida, aplicar etapa de mesclagem incremental, privilegiando sempre a menor lista disponível para reduzir custo de ordenação. A ideia central permanece: substitui-se a busca em espaço exponencial completo pela análise de subespaços menores, seguida de reconciliação cuidadosa das informações.

A natureza recursiva da divisão e conquista demanda atenção à profundidade de pilha em Python, já que chamadas excessivamente profundas podem exceder o limite padrão do interpretador. Contudo, como o algoritmo da mochila usando divisão ao meio produz árvore de recursão com altura logarítmica em relação ao número de itens, tal obstáculo raramente se materializa na prática. Ainda assim, programadores prudentes recorrem a estratégias iterativas quando a profundidade previsível se aproxima de milhares de níveis.

A compreensão desse método oferece lição valiosa: a escolha do paradigma não depende somente da teoria associada ao problema, tampouco da solução clássica mais citada, mas dos parâmetros reais de entrada. Quando a capacidade da mochila é grande e o total de itens pequeno, a divisão e a conquista com fusão de metades revela desempenho convidativo, produzindo resultado exato sem

custos de memória quadráticos característicos da programação dinâmica matricial. A análise criteriosa dos limites de cada instância, somada à versatilidade sintática de Python, capacita o desenvolvedor a selecionar a abordagem cujo equilíbrio entre velocidade, memorização e paralelismo melhor se alinha à demanda do projeto.

Vamos fazer um exercício prático. Ao planejar uma playlist para uma sessão de treino de exatamente uma hora ou para preencher um limite de quinhentos megabytes quando se baixa música para ouvir offline, surge o mesmo tipo de dilema do problema da mochila. Em ambiente de streaming, cada faixa corresponde a um item cujo peso pode ser interpretado como a duração em segundos ou o tamanho em megabytes. O valor atribuído a cada faixa pode espelhar diversos critérios: preferência do usuário, popularidade, grau de descoberta ou mesmo um score interno que o serviço de recomendação usa para medir adequação ao contexto.

Figura 24 – Representação artística do problema da playlist: é o mesmo problema da mochila.
Imagem produzida pelo autor com a ferramenta de inteligência artificial generativa Grok 3

Com essa analogia, a construção de playlists passa a ser encarada como tarefa de otimização. Se o ouvinte deseja preencher exatamente 60 minutos com músicas que possuam maior pontuação de relevância, a plataforma precisa selecionar faixas cujas somas de duração respeitem o limite, entregando, ao mesmo tempo, o maior valor agregado possível. Do ponto de vista operacional, a situação se repete na funcionalidade de download: o aplicativo, ao perceber que o usuário possui apenas quinhentos megabytes disponíveis no aparelho, deve escolher o subconjunto de canções que maximize satisfação sem ultrapassar o espaço em disco. Em ambos os casos, o algoritmo da mochila oferece o arcabouço matemático capaz de fornecer resposta ótima – ou, quando imprescindível, aproximada – em prazos aceitáveis para uma experiência em tempo real.

A relevância dessa modelagem extrapola o entretenimento. Grandes serviços como o Spotify investem em técnicas de divisão e conquista para decompor o problema global em subproblemas menores, permitindo paralelizar cálculos e ajustar playlists à medida que novas canções entram no catálogo. A divisão do universo de músicas em grupos temáticos ou de durações similares reduz a complexidade e viabiliza atualizações frequentes, requisito imprescindível em catálogos que ultrapassam setenta milhões de faixas.

Exemplo de aplicação

Exemplo 1 – Criação de playlist com restrição

Um usuário define que deseja ouvir música durante exatamente uma hora enquanto pratica corrida; além disso atribui, em seu histórico, notas de preferência de zero a cem para cada faixa sugerida pelo algoritmo de recomendação. O sistema de streaming recebe as durações e os valores dessas músicas e precisa devolver uma playlist cujo tempo total se aproxime ao máximo de 3.600 segundos, sem superá-los, e cujo somatório de notas seja o maior possível.

O serviço não pode simplesmente pegar as canções mais longas, porque talvez tenham pontuação baixa, nem pode escolher as de pontuação altíssima se isso ocasionar sobra de tempo significativo. A regra de negócio específica determina ainda que, para playlist motivacional, nenhuma canção pode ter duração inferior a noventa segundos, pois cortes bruscos interferem na cadência do treino.

Essas restrições guiam a rotina de cálculo dinâmico que selecionará a melhor combinação de faixas.

O código explora programação dinâmica, técnica que armazena resultados intermediários de subproblemas para evitar recomputações. Em Python, dicionários e listas facilitam a construção de matrizes de estado, nas quais as colunas representam capacidades parciais – aqui, segundos de duração – e as linhas correspondem às faixas consideradas até o momento. O módulo `dataclasses` simplifica a representação de cada música, associando duração e valor a um identificador legível. A escolha de lista bidimensional pode, à primeira vista, levantar preocupações de memória; entretanto, ao se guardar apenas a linha corrente e a anterior, economiza-se espaço, prática amplamente adotada em bibliotecas de otimização empregadas na indústria.

Para reconstruir a playlist final, o algoritmo percorre a matriz de trás para frente, descobrindo quais faixas contribuíram para o valor ótimo. Esse passo exige acesso aos estados salvos, razão pela qual se mantém não apenas o valor acumulado, mas um campo de retorno rápido que indica se o item atual foi incluído. Ao final, basta ordenar a seleção de faixas na ordem original ou reorganizá-las conforme critérios de transição suave de BPM (Beats Per Minute) se a especificação o exigir. A implementação utiliza ainda funções geradoras para permitir que o chamador transforme o resultado em streaming sem bloquear a interface, alinhando-se às boas práticas de aplicações assíncronas comuns em microsserviços. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from dataclasses import dataclass
from typing import List, Tuple

@dataclass
class Musica:
    id: str
    duracao: int     # segundos
    nota: int        # preferência 0100
```

```
def mochila_playlist(musicas: List[Musica], limite_seg: int) -> Tuple[int,
List[Musica]]:
    n = len(musicas)
    # matriz de valores: duas linhas para economia de memória
    dp = [[0]*(limite_seg+1) for _ in range(2)]
    # matriz booleana que marca inclusão da música na solução
    escolha = [[False]*(limite_seg+1) for _ in range(n)]

    for i in range(n):
        atual = i % 2
        prev = (i-1) % 2
        dur = musicas[i].duracao
        val = musicas[i].nota
        for s in range(limite_seg + 1):
            if dur <= s:
                incluir = val + dp[prev][s - dur]
                nao_incluir = dp[prev][s]
                if incluir > nao_incluir:
                  dp[atual][s] = incluir
                  escolha[i][s] = True
                else:
                  dp[atual][s] = nao_incluir
            else:
                dp[atual][s] = dp[prev][s]

    # reconstrução do conjunto de músicas
    s = limite_seg
    selecionadas: List[Musica] = []
    for i in range(n-1, -1, -1):
        if escolha[i][s]:
            selecionadas.append(musicas[i])
            s -= musicas[i].duracao
    selecionadas.reverse()
    return dp[(n-1)%2][limite_seg], selecionadas

# -------- Demonstração --------
def principal():
    catalogo = [
      Musica("Faixa1", 210, 85),
      Musica("Faixa2", 180, 92),
      Musica("Faixa3", 200, 75),
      Musica("Faixa4", 240, 60),
      Musica("Faixa5", 300, 95),
      Musica("Faixa6", 150, 80),
      Musica("Faixa7", 190, 70),
      Musica("Faixa8", 230, 88),
      Musica("Faixa9", 260, 77),
      Musica("Faixa10", 215, 90)
    ]
    valor, playlist = mochila_playlist(catalogo, 3600)
    tempo_total = sum(m.duracao for m in playlist)
    print(f"Nota agregada: {valor}")
    print(f"Tempo total: {tempo_total//60} min {tempo_total%60} s")
    print("\nPlaylist gerada:")
    for m in playlist:
      print(f"{m.id} - {m.duracao//60}:{m.duracao%60:02d} | nota {m.nota}")
```

```
if __name__ == "__main__":
        principal()
```

O quadro 25 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 25 – Funções usadas no código-fonte do primeiro exercício prático

Função ou construção	Explicação de uso
@dataclass	Cria uma classe Musica com os campos id, duracao e nota, e fornece automaticamente métodos como `__init__`, `__repr__` e `__eq__`
dp = [[0]*(limite+1) for _ in range(2)]	Cria uma matriz 2x(limite+1) para armazenar os valores máximos calculados, com alternância entre linhas para economizar memória (programação dinâmica com otimização de espaço)
escolha = [[False]*(limite+1) for _ in range(n)]	Matriz booleana que rastreia quais músicas foram incluídas na solução final, permitindo reconstrução posterior da playlist
if dur <= s:	Verifica se a música atual cabe no tempo disponível. Essa é a condição essencial do problema da mochila
dp[atual][s] = max(...) (aqui feito com lógica explícita)	Escolhe entre incluir ou não a música com base em qual das duas opções leva à maior nota agregada
playlist.reverse()	Após reconstruir a solução de trás para frente, inverte a lista para apresentá-la na ordem original das escolhas
sum(m.duracao for m in playlist)	Soma a duração total das músicas selecionadas
f-strings com formatação de tempo	`f"{m.duracao//60}:{m.duracao%60:02d}"` formata minutos e segundos com dois dígitos, garantindo alinhamento visual limpo da saída

O código Python apresentado implementa uma solução para o problema da mochila (knapsack problem) adaptada à seleção de músicas para compor uma playlist com duração limitada, maximizando a soma das notas de preferência das faixas selecionadas. Utilizando uma abordagem de programação dinâmica, o código organiza a lógica de forma eficiente, empregando estruturas de dados otimizadas e boas práticas de tipagem e modularidade.

A biblioteca `dataclasses` é usada para criar a classe `Musica`, que representa uma faixa com três atributos: `id` (string identificadora), `duracao` (segundos) e `nota` (valor de preferência entre 0 e 100). A biblioteca `typing` fornece suporte para a anotação de tipos, como `List` e `Tuple`, garantindo maior clareza e robustez ao código.

A função principal, `mochila_playlist`, recebe uma lista de objetos `Musica` e um limite de duração em segundos (`limite_seg`), retornando uma tupla com a soma máxima das notas das músicas selecionadas e a lista das músicas escolhidas. A implementação utiliza programação dinâmica para resolver o problema, mas otimiza o uso de memória ao empregar uma matriz bidimensional `dp` com apenas duas linhas, alternando entre elas para armazenar os estados atual e anterior. Essa técnica reduz a complexidade espacial de O(n × limite_seg) para O(limite_seg), em que n é o número de músicas. A matriz `dp` armazena, para cada capacidade de tempo s (de 0 a limite_seg), o valor máximo de notas que pode ser obtido considerando as músicas até o índice atual.

Além disso, uma matriz booleana escolha registra quais músicas foram incluídas na solução ótima para cada capacidade de tempo. Durante o preenchimento da matriz dp, o algoritmo itera sobre cada

música e cada capacidade de tempo possível. Para uma música de índice i com duração dur e nota val, o algoritmo compara duas opções: incluir a música (se a duração não exceder a capacidade restante) ou não a incluir. A inclusão adiciona a nota da música ao valor armazenado para a capacidade restante (s – dur), enquanto a não inclusão mantém o valor da iteração anterior. A decisão que maximiza o valor é registrada em dp, e a matriz escolha marca se a música foi incluída.

Após preencher a matriz, o algoritmo reconstrói a lista de músicas selecionadas percorrendo a matriz `escolha` de trás para frente, começando da capacidade total (`limite_seg`). Se uma música foi marcada como incluída, ela é adicionada à lista e a capacidade restante é reduzida pela duração da música. A lista resultante é invertida para manter a ordem original das músicas selecionadas. A função retorna a soma máxima das notas (obtida em `dp`) e a lista de músicas selecionadas.

A função `principal` demonstra o uso do algoritmo. Ela cria uma lista de dez objetos `Musica` com diferentes identificadores, durações e notas, representando um catálogo de faixas. A função chama `mochila_playlist` com um limite de `3600` segundos (1 hora) e exibe os resultados: a soma total das notas, o tempo total da playlist (convertido para minutos e segundos) e a lista de músicas selecionadas, com suas durações formatadas no padrão `minutos:segundos` e suas respectivas notas.

O código é eficiente, com complexidade temporal de O(n × limite_seg) e complexidade espacial de O(limite_seg) para a matriz dp, além de O(n × limite_seg) para a matriz escolha. A modularidade, o uso de tipagem explícita e a otimização de memória refletem boas práticas de programação, tornando o código robusto e adequado para cenários nos quais a seleção de playlists precisa balancear qualidade (notas) e restrições de tempo.

A seguir, constam sugestões de melhoria para que você implemente no futuro.

- A versão atual considera nota de preferência como valor escalar, mas poderia combinar múltiplos critérios – skip rate, similaridade de BPM, diversidade de artistas – mediante ponderação linear ou técnicas de otimização multiobjetivo.

- Outra extensão plausível envolveria a troca do algoritmo exato por heurísticas gulosas ou algoritmos genéticos quando o número de faixas ultrapassa dezenas de milhares, evitando latência em dispositivos móveis.

- Também seria possível integrar a solução com um motor de recomendação por colaboração filtrada, recalculando notas de valor em tempo real segundo o humor do usuário ou as condições climáticas detectadas.

- Finalmente, transformar a função geradora de playlist em corrotina compatível com asyncio permitiria que a interface exibisse resultados parciais à medida que eles fossem encontrados, melhorando a sensação de responsividade em redes instáveis.

8.2 Programação dinâmica: introdução e resolução de problemas clássicos (como a subsequência comum mais longa)

A programação dinâmica apresenta-se como uma técnica que transforma problemas recursivos com sobreposição de subestruturas em algoritmos eficientes por meio do armazenamento de resultados intermediários. A ideia central parte da identificação de duas propriedades: o problema original pode ser quebrado em subproblemas menores que mantêm mesmo formato e a solução global deriva da combinação ótima das soluções dessas partes. Quando o programador reconhece que certas subinstâncias surgirão repetidamente ao longo da recursão, aparece a oportunidade de evitar recalculações, guardando cada resposta em tabela que será consultada sempre que a mesma configuração reaparecer. Esse conceito reduz o tempo de execução de exponencial para polinomial em diversos casos, mantendo consumo de memória controlado, pois cada subproblema é resolvido uma única vez.

A subsequência comum mais longa ilustra de forma clara a utilidade desse paradigma. Diante de duas cadeias de caracteres, procura-se a maior sequência de símbolos que aparece nas duas cadeias na mesma ordem relativa, embora não necessariamente contígua. Um procedimento ingenuamente recursivo testaria todas as combinações possíveis de remoção de caracteres, gerando quantidade exponencial de chamadas.

A programação dinâmica revela que a resposta para prefixos de comprimento i e j depende apenas de duas situações: se o caractere corrente nas duas cadeias coincide, então basta acrescentar um à solução encontrada para os prefixos imediatamente anteriores; caso contrário, escolhe-se a maior entre as soluções obtidas ao descartar o último caractere de uma cadeia ou da outra. Após reconhecer esse relacionamento, preenche-se uma tabela bidimensional cujas dimensões correspondem aos tamanhos das cadeias, avançando linha a linha ou coluna a coluna. Cada célula recebe valor proveniente de operações constantes sobre células vizinhas já calculadas, resultando em complexidade de tempo O(n m) e consumo de memória proporcional à área da tabela, no qual n e m são os comprimentos das cadeias.

Observação

Do ponto de vista didático, a construção ascendente da tabela (versão iterativa) costuma ser mais intuitiva para iniciantes, pois a ordem de preenchimento espelha diretamente as dependências entre subproblemas. Já a versão descendente (memorização) mantém estrutura recursiva original, recorrendo a dicionário ou lista para guardar resultados conforme a execução avança. Em Python, ambas abordagens tiram proveito de características da linguagem: a flexibilidade das listas aninhadas favorece a implementação bottom-up, enquanto a facilidade de definir funções internas ou uso de decoradores facilita o cache automático dos valores em uma versão top-down. A escolha entre essas duas formas depende da clareza buscada pelo autor do código e das restrições de memória do ambiente, pois a variante descendente permite, em algumas situações, limitar armazenamento a subinstâncias realmente visitadas.

A aplicação de programação dinâmica ultrapassa o exemplo da subsequência comum mais longa e cobre um espectro que inclui alinhamento de cadeias de DNA em bioinformática, cálculo de rotas mínimas em grafos com pesos não negativos via algoritmo de Floyd-Warshall e otimização de corte de hastes metálicas em processos industriais. Em cada um desses cenários, o analista precisa identificar o padrão de sobreposição e projetar uma expressão de recorrência que represente a dependência das partes menores, tarefa que exige entendimento profundo do modelo do problema. Uma vez formulada a recorrência, a transcrição para código Python torna-se direta, aproveitando a clareza semântica da linguagem para manter o algoritmo legível e a facilidade de depuração durante a verificação dos resultados. Por meio dessa técnica, problemas antes considerados intratáveis tornam-se viáveis em tempo polinomial, demonstrando a importância de reconhecer subestruturas repetidas e de armazenar suas respostas em tabelas que garantem eficiência tanto temporal quanto espacial.

Saiba mais

O livro a seguir é específico para programação dinâmica e detalha problemas como mochila e subsequência comum mais longa, com soluções em Python. Ensina a construir tabelas iterativas e otimizar acessos à memória, conectando-se aos exemplos de bioinformática e alinhamento de sequências do texto. Sendo excelente para prática intensiva e aplicações reais.

KAMAL, M.; RAWAT, K. *Dynamic programming for coding interviews*. Chetpet: Notion Press, 2017.

Novamente, faremos um exercício prático. As redes sociais adicionaram camadas de linguagem que transcendem o texto e os emojis transformaram-se em sinais de empatia digital. No universo do Instagram, o recurso "Reações rápidas" permite que o usuário responda a um story com um único clique em um emoji predefinido. Diferentemente de comentários longos, essas reações funcionam como microexpressões que, quando observadas em sequência, podem revelar padrões de sincronia ou afinidade entre dois perfis. Se dois amigos costumam reagir aos stories de seus círculos usando a mesma ordem de emojis – riso, coração, aplauso –, cresce a impressão de que pensam parecido e compartilham repertório afetivo.

Essa hipótese, divertida e quase antropológica, pode ser mapeada para o problema clássico da subsequência comum mais longa (LCS). Como vimos, a LCS mede quão parecidas são duas sequências, preservando a ordem de ocorrência, mas sem exigir contiguidade. Aplicada às reações de stories, a métrica responde à pergunta: qual é o maior conjunto de emojis, na ordem original, que aparece em ambos os históricos de reação? Quanto maior esse conjunto, maior é o indício de que os amigos não apenas gostam das mesmas coisas, mas respondem de maneira sincronizada ao conteúdo, reforçando a tese de amigos do peito.

O charme dessa abordagem reside no fato de que o LCS ignora os intervalos entre reações: não importa se um deles reagiu em dias diferentes ou se adicionou emojis extras, basta que a sequência relativa se

mantenha. Dessa forma, a análise reflete afinidade de padrão e não apenas coincidência pontual. Ao traduzir um fenômeno social em algoritmo de programação dinâmica, o exercício demonstra como técnicas formais podem iluminar hábitos aparentemente triviais da vida online.

Exemplo de aplicação

Exemplo 2 – Afinidade via emojis

Dois usuários, Ana e Beto, usam o Instagram diariamente. O aplicativo interno da startup que os monitora registrou, durante um período de trinta stories, as reações rápidas de cada um, codificando 😂 como "R", 😍 como "C" (coração) e 👋 como "A".

Para estabelecer um índice de amizade, a empresa define a seguinte regra: se a LCS entre as duas sequências de reações possuir tamanho superior a quinze, Ana e Beto entram no cluster "superamigos" e liberam um badge especial.

Caso a LCS fique entre dez e quinze, recebem o selo "amigos em ascensão".

Abaixo de dez, nenhum selo é atribuído.

O sistema precisa ler as sequências, calcular a LCS e, com base no resultado, imprimir o relacionamento detectado. Como os dados são coletados em dispositivos móveis, o algoritmo deve ocupar pouca memória e responder em tempo real, garantindo que o selo apareça no perfil assim que o trigésimo story expira.

A resolução emprega programação dinâmica, técnica que quebra um problema global em subproblemas de fronteira compartilhada, armazenando resultados intermediários para evitar recomputação. Em Python, a construção de uma grade bidimensional de inteiros é simples graças à compreensão de listas, mas o consumo de memória cresce com o produto dos tamanhos das sequências. Para reduzir esse impacto, a implementação armazena apenas a linha corrente e a anterior da matriz `dp`, estratégia clássica que reduz o espaço de $O(n \times m)$ para $O(2 \times m)$, em que m é o comprimento da segunda sequência.

A reconstrução da LCS exige atenção, pois a otimização de memória impede guardar toda a matriz necessária para retroceder. O código contorna esse obstáculo preservando, durante o preenchimento, uma estrutura adicional que grava pontos de salto quando ocorre igualdade de caracteres. Essa técnica, embora ocupe memória proporcional ao resultado em vez da matriz completa, satisfaz o requisito de economia e segue práticas adotadas em sistemas de recomendação que operam sobre logs comprimidos.

Para manter a clareza do domínio, o script utiliza `dataclass Perfil` para empacotar nome do usuário e sequência de reações; isso exemplifica como equipes de engenharia modelam entidades de negócio e ainda se beneficiam de comparações estruturais automáticas. Além disso, a tipagem opcional from __future__ import annotations torna o código pronto para análise com `MyPy`, garantindo que a solução possa migrar para bases de produção em que verificação estática virou padrão. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from dataclasses import dataclass
from typing import List, Tuple

@dataclass
class Perfil:
    nome: str
    reacoes: str       # sequência como string de códigos 'R','C','A'

def lcs_quase_linear(a: str, b: str) -> Tuple[int, str]:
    if len(a) < len(b): # garante que b seja a menor para poupar memória
        a, b = b, a
    m, n = len(a), len(b)
    dp = [[0]*(n+1) for _ in range(2)]
    caminho: List[List[int]] = [[0]*(n+1) for _ in range(m+1)]

    for i in range(1, m+1):
        linha_atual = i % 2
        linha_ant = (i-1) % 2
        for j in range(1, n+1):
            if a[i-1] == b[j-1]:
                dp[linha_atual][j] = dp[linha_ant][j-1] + 1
                caminho[i][j] = 1 # diagonal
            else:
                if dp[linha_ant][j] >= dp[linha_atual][j-1]:
                    dp[linha_atual][j] = dp[linha_ant][j]
                    caminho[i][j] = 2 # cima
                else:
                    dp[linha_atual][j] = dp[linha_atual][j-1]
                    caminho[i][j] = 3 # esquerda

    # reconstrução da subsequência
    lcs_lista: List[str] = []
    i, j = m, n
    while i > 0 and j > 0:
        if caminho[i][j] == 1:
          lcs_lista.append(a[i-1])
          i -= 1
          j -= 1
        elif caminho[i][j] == 2:
          i -= 1
        else:
          j -= 1
    return dp[m % 2][n], "".join(reversed(lcs_lista))

def classificar_amistade(tamanho_lcs: int) -> str:
    if tamanho_lcs > 15:
        return "super amigos"
    if tamanho_lcs >= 10:
        return "amigos em ascensão"
    return "apenas conhecidos"

# -------- Demonstração --------
def principal():
    ana = Perfil("Ana", "RCAARCRACRACARCRAA")
    beto = Perfil("Beto", "RRACRACRCRAARCRACA")
```

```
    tam, seq = lcs_quase_linear(ana.reacoes, beto.reacoes)
    grau = classificar_amistade(tam)
    print(f"A subsequência comum mais longa tem {tam} emojis.")
    print(f"Sequência compartilhada: {seq}")
    print(f"Diagnóstico de amizade: {grau}")

if __name__ == "__main__":
        principal()
```

O quadro 26 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 26 – Funções usadas no código-fonte do segundo exercício prático

Função ou construção	Explicação de uso
@dataclass	Define a estrutura `Perfil` com campos nome e reacoes. A anotação automatiza métodos como `__init__` e `repr`, simplificando a manipulação dos dados
dp = [[0]*(n+1) for _ in range(2)]	Cria uma matriz de duas linhas (espaço otimizado) para armazenar os valores intermediários da LCS. Usa alternância entre linhas para economizar memória
caminho = [[0]*(n+1) for _ in range(m+1)]	Armazena a direção das decisões tomadas na programação dinâmica (diagonal, cima ou esquerda), permitindo reconstrução posterior da sequência
if a[i-1] == b[j-1]:...	Compara os caracteres das sequências. Se iguais, avança na diagonal, o que indica um caractere da LCS
dp[linha_atual][j] = max(...)	Aplica a regra da programação dinâmica: se os caracteres não coincidem, escolhe o valor máximo entre cima ou esquerda
reconstrução da LCS com while i > 0 and j > 0	Percorre a matriz caminho de volta à origem para formar a subsequência comum mais longa
"".join(reversed(lista))	Reverte e junta a lista de caracteres da LCS, pois ela foi construída de trás para frente
tamanho_lcs > 15 / >= 10	Classifica o grau de amizade com base no comprimento da LCS. Um exemplo prático de uso de thresholds em categorização

No início, a função `principal` cria dois objetos da classe `Perfil`, representando `Ana` e `Beto`, cada um com um nome e uma string de reações composta por caracteres R, C e A. Essas strings, passadas como argumentos, são `RCAARCRACRACARCRAA` para `Ana` e `RRACRACRCRAARCRACA` para `Beto`. A classe `Perfil`, definida como um `dataclass`, organiza esses dados de forma estruturada, garantindo que cada perfil tenha um `nome` (string) e uma sequência de `reações` (string).

A função `lcs_quase_linear` é chamada com as sequências de reações de `Ana` e `Beto` como entrada. Essa função implementa uma versão otimizada do algoritmo de programação dinâmica para calcular a LCS. Para economizar memória, ela verifica qual das duas strings é menor e, se necessário, troca-as, garantindo que a string menor seja processada como segunda entrada. A função utiliza uma matriz `dp` com apenas duas linhas, alternando entre elas para armazenar os comprimentos das subsequências comuns em cada iteração, o que reduz o consumo de memória em comparação com uma matriz completa de tamanho m x n, em que m e n são os comprimentos das strings.

Durante a execução, a função constrói uma matriz auxiliar chamada `caminho`, que registra as decisões tomadas para reconstruir a subsequência comum. Para cada par de caracteres das duas strings, a função verifica se eles são iguais. Se forem, o comprimento da subsequência aumenta em 1 e a direção "`diagonal`" é marcada em `caminho`. Caso contrário, a função escolhe o maior valor entre a célula

acima (ignorando o caractere da primeira string) e a célula à esquerda (ignorando o caractere da segunda string), registrando a direção correspondente ("`cima`" ou "`esquerda`").

Após calcular o comprimento da LCS, a função reconstrói a subsequência comum, percorrendo a matriz `caminho` a partir da posição final (m, n). Ela segue as direções armazenadas, adicionando caracteres à lista `lcs_lista` quando encontra uma direção "`diagonal`" (indicando que os caracteres correspondentes são iguais). A lista é revertida e unida em uma string, retornando tanto o comprimento da LCS quanto a própria subsequência.

O comprimento da LCS retornado por `lcs_quase_linear` é passado para a função `classificar_amistade`, que categoriza a amizade com base em critérios predefinidos. Se o comprimento for maior que `15`, os perfis são classificados como "`super amigos`"; se for maior ou igual a `10`, são "`amigos em ascensão`"; caso contrário, são "`apenas conhecidos`". Essa classificação reflete o grau de similaridade entre as sequências de reações.

Na função `principal`, os resultados são exibidos em três linhas: a primeira informa o comprimento da LCS, a segunda mostra a sequência compartilhada e a terceira apresenta o diagnóstico de amizade. No caso de `Ana` e `Beto`, a execução do código calcula a LCS entre as strings fornecidas, determina seu comprimento, reconstrói a sequência comum e classifica a amizade com base nesse valor, exibindo os resultados de forma clara e concisa.

A seguir, constam sugestões de melhoria para que você implemente no futuro.

- O exercício pode evoluir para comparar não apenas emojis, mas o timing das reações, criando subsequências de pares (emoji, carimbo de hora) e aplicando variações de LCS que toleram diferença de até trinta segundos entre eventos.

- Outra linha de extensão seria ponderar cada emoji por peso emocional, atribuindo valores mais altos para ❤️ ou 🔥 e transformar o problema em "LCS com peso", que requer adaptação da matriz de programação dinâmica.

- Para escalar a solução a milhões de usuários, seria útil investigar algoritmos de LCS aproximado baseados em sketches ou hashing sensível a ordem, reduzindo processamento em grafos densos de amizade.

- Finalmente, integrar a análise em tempo real usando corrotinas asyncio permitiria que a recomendação de stickers e filtros se adaptasse on-the-fly quando a LCS entre pares superasse certos limiares durante eventos ao vivo.

Resumo

Nesta unidade, descrevemos inicialmente grafos simples, nos quais vértices representam entidades e arestas relações; ao receberem pesos, tais arestas passam a acomodar custos, distâncias ou capacidades, permitindo formular problemas de otimização. A exploração estrutural começou com a busca em largura e em profundidade: a primeira visita vértices em camadas, garantindo caminho mínimo em número de saltos quando não há pesos, enquanto a segunda avança recursivamente por um ramo até o fim antes de retroceder, revelando componentes conectados e ciclos.

Em seguida, estudamos o problema de caminhos mínimos ponderados. O algoritmo de Dijkstra fornece solução rápida quando todos os pesos são não negativos, utilizando fila de prioridade implementada com heap binário para selecionar o vértice de menor custo provisório e relaxar arestas adjacentes, perfazendo complexidade O((V + E) log V). Caso existam pesos negativos, Dijkstra perde correção, motivo pelo qual se adota Bellman-Ford, que relaxa todas as arestas repetidamente até V - 1 vezes, detectando ciclos de peso negativo em passagem extra e apresentando custo O(V E).

Abordamos ainda a construção de árvores geradoras mínimas. Kruskal ordena todas as arestas por peso e utiliza a estrutura de união-busca para unir componentes desconexos sem formar ciclos, atingindo complexidade O(E log E). Prim parte de um vértice arbitrário, mantém uma fronteira de arestas em fila de prioridade e incorpora, a cada passo, a aresta mais barata que liga o conjunto visitado a um vértice externo, obtendo custo O(E log V).

Apresentamos também a divisão e a conquista com programação dinâmica. O primeiro fragmenta problemas independentes, resolve-os recursivamente e funde resultados, ilustrando o processo com o problema da mochila dividido em duas metades, técnica que permite paralelismo natural e supera força bruta para até sessenta itens. A programação dinâmica, por sua vez, armazena subsoluções para eliminar recomputação, como na subsequência comum mais longa ou na mochila matricial, levando problemas originalmente exponenciais a tempo polinomial. O exemplo da geração de playlists de uma hora demonstra como restrições de duração ou espaço de armazenamento convertem-se diretamente no modelo da mochila; a implementação em Python otimiza memória mantendo apenas duas linhas da matriz e reconstrói a seleção final percorrendo marcações de inclusão.

Por fim, conclui-se que a compreensão conceitual dos algoritmos em grafos e dos paradigmas de divisão e conquista e programação dinâmica

sustenta decisões de projeto que influenciam viabilidade e escalabilidade de sistemas, desde navegação veicular até distribuição de microsserviços e curadoria de conteúdo. Como vimos, Python oferece bibliotecas maduras que encapsulam essas técnicas.

Exercícios

Questão 1. (Cespe-Cebraspe 2018, adaptada) Considere o grafo disposto a seguir.

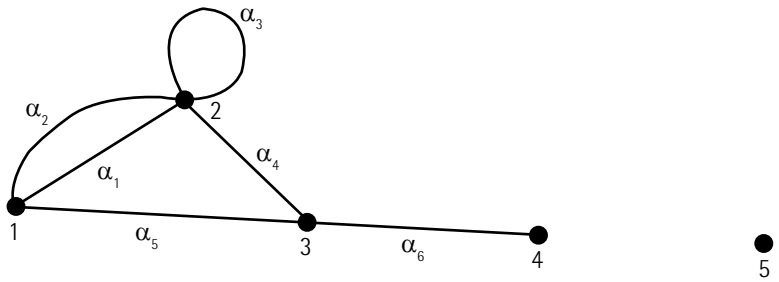

Figura 25

A respeito do grafo apresentado, avalie as afirmativas.

I – Os nós 1 e 4 são adjacentes.

II – O nó 5 é adjacente a si mesmo.

III – Os nós 2 e 3 têm grau 3.

IV – O grafo não pode ser classificado como conexo.

É correto o que se afirma em:

A) I, apenas.

B) III, apenas.

C) IV, apenas.

D) I, II e IV, apenas.

E) I, II, III e IV.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: dois nós em um grafo são ditos adjacentes se ambos são extremidades de algum arco. No grafo da questão, os nós 1 e 4 não são conectados diretamente por apenas um arco, por mais que existam caminhos entre eles. Desse modo, eles não são considerados adjacentes.

II – Afirmativa incorreta.

Justificativa: um nó que é terminal de um laço é dito ser adjacente a si próprio. No nó 5, não há qualquer laço, já que ele não é conectado a ele mesmo por uma aresta. Desse modo, não é possível dizer que o nó 5 é adjacente a si mesmo.

III – Afirmativa incorreta.

Justificativa: o grau de um nó é o número de arcos que terminam naquele nó. O nó 3 tem grau 3, já que 3 arcos chegam a ele. O nó 2 tem grau 4, já que 4 arcos chegam a ele.

IV – Afirmativa correta.

Justificativa: um grafo é classificado como conexo se existe um caminho de qualquer nó para qualquer outro. Em outras palavras, em um grafo conexo, existe pelo menos um caminho entre cada par de nós do grafo. No grafo da questão, o nó 5 não está conectado a nenhum outro. Isso faz com que o grafo seja classificado como desconexo.

Questão 2. (UFMG 2019, adaptada) O famoso algoritmo de Dijkstra soluciona um problema de grafos direcionados e não direcionados com uma certa complexidade. A respeito desse algoritmo, avalie as afirmativas.

I – Trata-se de um algoritmo que resolve o problema de encontrar o caminho com menor número de arestas entre um vértice e outro.

II – Somente pode ser utilizado em grafos ponderados cujas arestas não são negativas.

III – Tem complexidade O(V!), em que V é o número de vértices no grafo.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa B.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: o algoritmo de Dijkstra é um algoritmo de caminho mínimo de fonte única, que resolve o problema de encontrar o caminho mais curto em um grafo ponderado com arestas não negativas. O objetivo do algoritmo é determinar o caminho mais curto a partir de um nó de origem para todos os outros vértices do grafo, levando em consideração os pesos das arestas. Logo, seu objetivo não é diminuir o número de arestas ao longo do caminho, mas encontrar o caminho cujo somatório dos pesos é menor.

II – Afirmativa correta.

Justificativa: o algoritmo de Dijkstra trabalha com grafos ponderados, direcionados ou não, que não têm arestas negativas.

III – Afirmativa incorreta.

Justificativa: a complexidade do algoritmo de Dijkstra depende da implementação, mas nunca é O(V!). Considerando que E é o número de arestas e V é o número de nós no grafo, a complexidade situa-se em O((V + E) log V),quando o heap binário gerencia tanto extrações quanto inserções resultantes dos relaxamentos.

REFERÊNCIAS

ADEL'SON-VEL'SKII, G. M.; LANDIS, E. M. An algorithm for the organization of information. *Soviet Math*, v. 3, p. 1259-1263, 1962.

AHO, A. V.; HOPCROFT, J.; ULLMAN, J. *Estruturas de dados e algoritmos*. Porto Alegre: Bookman, 2002.

BEAZLEY, D.; JONES, B. K. *Python cookbook*: recipes for mastering Python 3. 3. ed. [S.l.]: O'Reilly Media, 2013.

BOOCH, G. *Object-oriented analysis and design with applications*. 2. ed. Redwood City: Benjamin/Cummings, 1994.

CORMEN, T. H. *et al. Algoritmos*: teoria e prática. 4. ed. Rio de Janeiro: LTC, 2024.

GARG, H. K. *Hands-on bitcoin programming with Python*: build powerful online payment centric applications with Python. Birmingham: Packt Publishing, 2018.

GÉRON, A. *Mãos à obra*: aprendizado de máquina com Scikit-Learn, Keras & TensorFlow – conceitos, ferramentas e técnicas para a construção de sistemas inteligentes. 2. ed. Rio de Janeiro: Alta Books, 2021.

GOODRICH, M. T.; TAMASSIA, R.; GOLDWASSER, M. *Data structures and algorithms in Python*. New Jersey: Wiley2021.

GRUS, J. *Data Science do zero*: noções fundamentais com Python. 2. ed. Rio de Janeiro: Alta Books, 2021.

HETLAND, M. L. *Python algorithms*: mastering basic algorithms in the Python language. 2. ed. New York: Apress, 2014.

KAMAL, M.; RAWAT, K. *Dynamic programming for coding interviews*. Chetpet: Notion Press, 2017.

KLEINBERG, J.; TARDOS, É. *Algorithm design*. Boston: Addison-Wesley Professional, 2005.

KNUTH, D. *The art of computer programming*. 3. ed. California: Addison-Wesley Professional, v. 2, 1997.

KOK, A. S. *Hands-on blockchain for Python developers*: gain blockchain programming skills to build decentralized applications using Python. Birmingham: Packt Publishing, 2019.

LUTZ, M. *Learning Python*. 5. ed. Sebastopol: O'Reilly Media, 2013.

MCKINNEY, W. *Python para análise de dados*: tratamento de dados com pandas, NumPy & Jupyter. 3. ed. São Paulo: Novatec, 2023.

MILLER, B. N.; RANUM, L. *Problem solving with algorithms and data structures using Python*. [S.l.]: Franklin, Beedle & Associates, 2013.

PATTERSON, D.; HENNESSY, J. *Computer organization and design risc-V edition*: the hardware software interface. San Francisco: Morgan Kaufmann Publishers, 2020.

PHILLIPS, D. *Python 3 object-oriented programming*. Birmingham: Packt Publishing, 2010.

RAMALHO, L. *Python fluente*: programação clara, concisa e eficiente. São Paulo: Novatec, 2015.

RAMALHO, L. *Fluent Python*. 2. ed. Sebastopol: O'Reilly Media, 2022.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Reading: Addison-Wesley Professional, 2011.

SWEIGART, A. *Automatize tarefas maçantes com Python*: programação prática para verdadeiros iniciantes. São Paulo: Novatec, 2015.

VAN FLYMEN, D. *Learn blockchain by building one*: a concise path to understanding cryptocurrencies. Berkeley: Apress, 2020.

VAN ROSSUM, G.; DRAKE, F. L. *Python language reference manual*. Virgínia: PythonLabs, 2007.

WENGROW, J. *A common-sense guide to data structures and algorithms in Python*. Vol. 1. Raleigh: Pragmatic Bookshelf, 2024.

WING, J. M. Computational thinking. *Communications of the ACM*, v. 49, n. 3, p. 33-35, 2006.

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