EAD

EDUCAÇÃO A DISTÂNCIA

UNIDADE IV

Algoritmos e Estrutura de Dados em Python

Prof. MSc. Tarcísio Peres

Conteúdo da Unidade IV

- Algoritmos em Grafos.
- Caminhos mínimos: Dijkstra e Bellman-Ford.
- Grafos com árvores: algoritmos de Kruskal e Prim para árvores geradoras mínimas.
- Problemas de divisão e conquista: análise de exemplos como o problema da mochila.
- Programação dinâmica: introdução e resolução de problemas clássicos (como a subsequência comum mais longa).

Caminhos mínimos

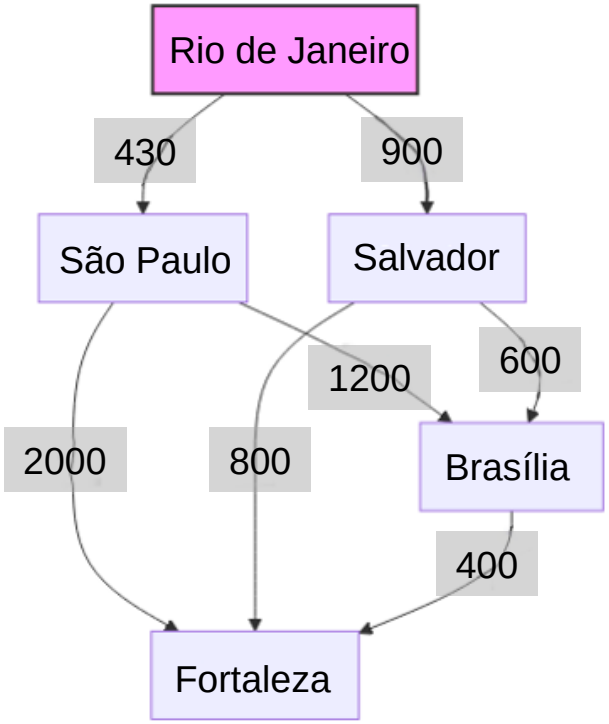
- A determinação de caminhos mínimos (single-source shortest path) em grafos constitui tema central da ciência da computação por viabilizar rotas com menor custo em mapas de tráfego rodoviário, redes de comunicação e fluxos de suprimentos.
- Em Python, duas abordagens clássicas atendem a tal demanda: 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

- 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.

Dijkstra

- 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.



Fonte: Autoria própria.

Dijkstra – Exemplo

- Imagine que você é um turista que começa sua jornada no RJ e quer visitar todas essas cidades, mas quer minimizar a distância de viagem. 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 1200 km.
- Fortaleza a 2000 km.
- O algoritmo escolhe Brasília, pois está mais próxima.

Dijkstra – Exemplo

Passo 3: Você chega em Brasília e verifica as opções de cidades próximas:

- Fortaleza a 400 km.
- O algoritmo escolhe Fortaleza, pois é a cidade mais próxima.

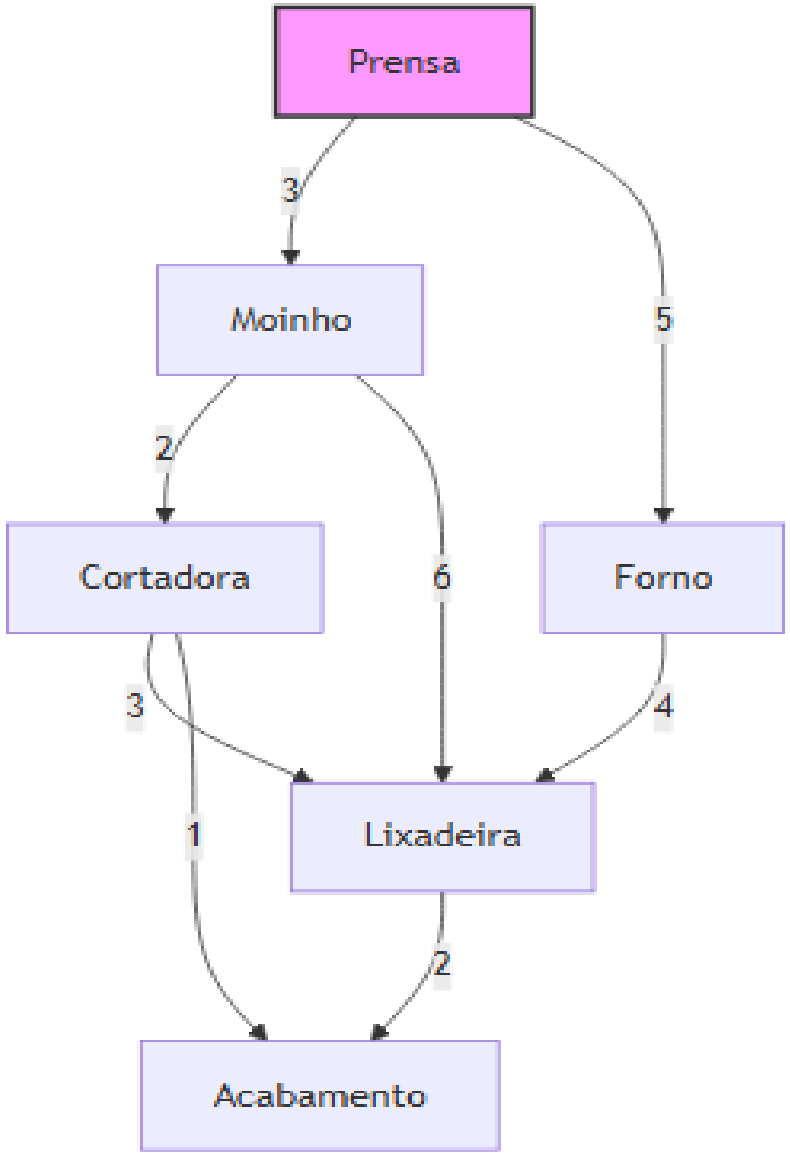
Passo 4: Agora que você está em Fortaleza, o único destino restante é Salvador, que está a 800 km de distância.

- 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.

Dijkstra – Exemplo fabril

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.



Fonte: Autoria própria.

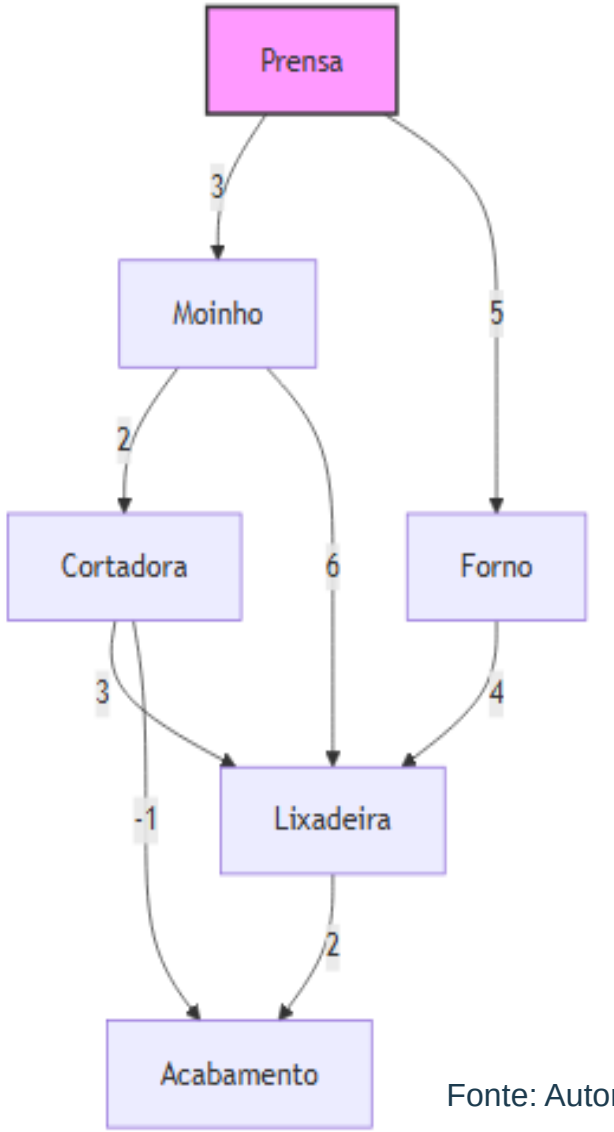
Adaptação – Exemplo fabril

- Vamos imaginar 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.

Adaptação – Exemplo fabril



Fonte: Autoria própria.

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 3 unidades.
- Enviar o material para o Forno a um custo de 5 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 2 unidades.
- Enviar para a Lixadeira a um custo de 6 unidades.
- O algoritmo escolhe a Cortadora, pois o custo é menor.

Bellman-Ford

Passo 3: O material chega na 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:

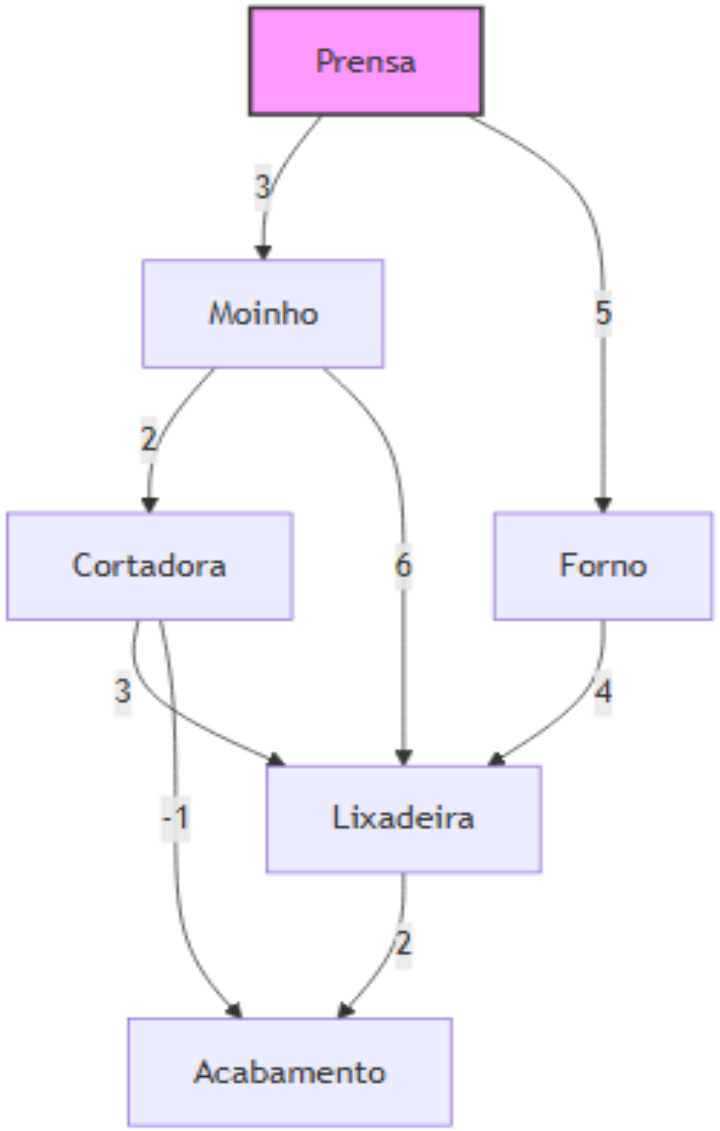
- Enviar para a Lixadeira a um custo de 3 unidades.
- Enviar para o Acabamento com um custo de -1 unidade.
- O algoritmo escolhe o Acabamento, aproveitando o custo negativo e, portanto, reduzindo o custo total.

Passo 4: O material chega diretamente ao Acabamento, finalizando o processo.

Bellman-Ford

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.



Fonte: Autoria própria.

Comparativo

- 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.

Implementação

- 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.

```
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)
```

Implementação

```
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))
```

Interatividade

Considerando os algoritmos de Dijkstra e Bellman-Ford para o cálculo de caminhos mínimos em grafos, analise as afirmativas a seguir e assinale a alternativa correta.

a) O algoritmo de Dijkstra pode ser utilizado normalmente em grafos que apresentam arestas com pesos negativos, sem risco de resultados incorretos.
b) Bellman-Ford possui a mesma complexidade temporal de Dijkstra, sendo preferível para grafos densos e sem restrições de peso nas arestas.
c) O algoritmo de Dijkstra é eficiente em grafos com pesos não negativos e não detecta ciclos negativos, enquanto Bellman-Ford pode lidar com arestas de peso negativo e é capaz de identificar ciclos de custo negativo.
d) Ambos os algoritmos exigem que o grafo seja acíclico para garantir a correta obtenção dos menores caminhos entre dois vértices.
e) Dijkstra é sempre mais eficiente e preciso que Bellman-Ford, independentemente do tipo de grafo ou dos pesos das arestas.

Resposta

Considerando os algoritmos de Dijkstra e Bellman-Ford para o cálculo de caminhos mínimos em grafos, analise as afirmativas a seguir e assinale a alternativa correta.

a) O algoritmo de Dijkstra pode ser utilizado normalmente em grafos que apresentam arestas com pesos negativos, sem risco de resultados incorretos.
b) Bellman-Ford possui a mesma complexidade temporal de Dijkstra, sendo preferível para grafos densos e sem restrições de peso nas arestas.
c) O algoritmo de Dijkstra é eficiente em grafos com pesos não negativos e não detecta ciclos negativos, enquanto Bellman-Ford pode lidar com arestas de peso negativo e é capaz de identificar ciclos de custo negativo.
d) Ambos os algoritmos exigem que o grafo seja acíclico para garantir a correta obtenção dos menores caminhos entre dois vértices.
e) Dijkstra é sempre mais eficiente e preciso que Bellman-Ford, independentemente do tipo de grafo ou dos pesos das arestas.

Definição e importância das árvores geradoras mínimas

- Árvores geradoras mínimas (MST, do inglês *Minimum Spanning Tree*) conectam todos os vértices de um grafo ponderado pelo menor custo possível, evitando a formação de ciclos.
- O objetivo é selecionar subconjunto de arestas que garanta conectividade global e minimize o custo total da rede.
- Essa abordagem é fundamental para problemas de infraestrutura, redes de computadores e economia de recursos.
- A árvore geradora mínima resulta em $V - 1$ arestas, onde V é o número total de vértices do grafo analisado.
    - Algoritmos clássicos como Kruskal e Prim são usados para obter essa solução de maneira eficiente e comprovada.
    - O resultado final atende à propriedade do corte: qualquer aresta de menor peso cruzando uma partição sem ciclo mantém a minimalidade.

Princípios do algoritmo de Kruskal

- Kruskal inicia com todos os vértices isolados, formando uma floresta na qual cada vértice é sua própria árvore.
- Ordena-se a lista de arestas pelo peso, do menor para o maior, garantindo prioridade às conexões mais baratas.
- Cada nova aresta é avaliada para inclusão, desde que não crie ciclo entre componentes já conectados.
- A verificação de ciclos utiliza estrutura união-busca (Union-Find) com compressão de caminho e união por tamanho.
    - O custo de identificar componentes cresce lentamente, devido à eficiência do Union-Find com função de Ackermann.
    - O processo termina quando todos os vértices formam um único componente conectado, ou seja, uma árvore.

Complexidade e vantagens do algoritmo de Kruskal

- O tempo total do algoritmo de Kruskal é dominado pela ordenação das arestas, resultando em complexidade $O(E \log E)$.
- Cada inclusão de aresta diminui o número de componentes, aproximando a floresta de uma árvore única.
- Union-Find opera em tempo quase constante, tornando eficiente a verificação de conexões entre vértices.
- Kruskal é especialmente eficiente em grafos esparsos, com número de arestas próximo ao número de vértices.
    - O algoritmo é naturalmente paralelizável, pois o processamento de diferentes blocos de arestas pode ocorrer simultaneamente.
    - É ideal em contextos nos quais as conexões físicas ou lógicas são adicionadas gradativamente por custos crescentes.

Definição do termo floresta e função de Ackermann

- “Floresta” representa um conjunto de componentes desconexos, cada um sendo uma árvore isolada no início do algoritmo.
- O termo descreve o estado inicial e intermediário em Kruskal, diferenciando-se da árvore única obtida ao final.
- Ao longo das iterações, árvores individuais vão sendo conectadas até restar apenas uma, a árvore geradora mínima.
- A função de Ackermann, citada no texto, expressa a eficiência teórica da compressão de caminho em Union-Find.
    - Seu crescimento extremamente rápido é utilizado na análise de algoritmos avançados de teoria da computação.
    - O uso dessa função demonstra que a identificação de componentes é praticamente constante mesmo para grandes grafos.

Princípios do algoritmo de Prim

- Prim parte de um único vértice e cresce continuamente o componente conectado ao restante do grafo.
- Utiliza uma fila de prioridade para selecionar, a cada etapa, a aresta de menor peso conectando novo vértice.
- Em cada iteração, um vértice externo é incorporado, expandindo a árvore sem formação de ciclos.
- As arestas incidentes ao vértice recém-incluído entram na fila se levarem a vértices ainda não visitados.
    - A implementação com heap binário (heapq) assegura inserção e remoção eficiente em $O(\log V)$ por operação.
    - O processo segue até que todos os vértices tenham sido adicionados ao componente em expansão.

Complexidade e aplicações do algoritmo de Prim

- O tempo de execução de Prim é $O(E \log V)$, tornando-o eficiente em grafos densos e com muitas arestas.
- Em grafos muito densos, substituir o heap por matriz de chaves pode simplificar para $O(V^2)$, útil em grafos completos.
- O algoritmo é incremental, pois depende da escolha local do mínimo a cada nova expansão de componente.
- Fila de prioridade bem implementada reduz latência na escolha da próxima aresta a ser incorporada.
    - Prim é preferido em aplicações nas quais o número de arestas excede significativamente o número de vértices.
    - Adequado para situações em que o armazenamento das arestas não é significativamente superior ao dos vértices.

Comparação entre Prim e Kruskal

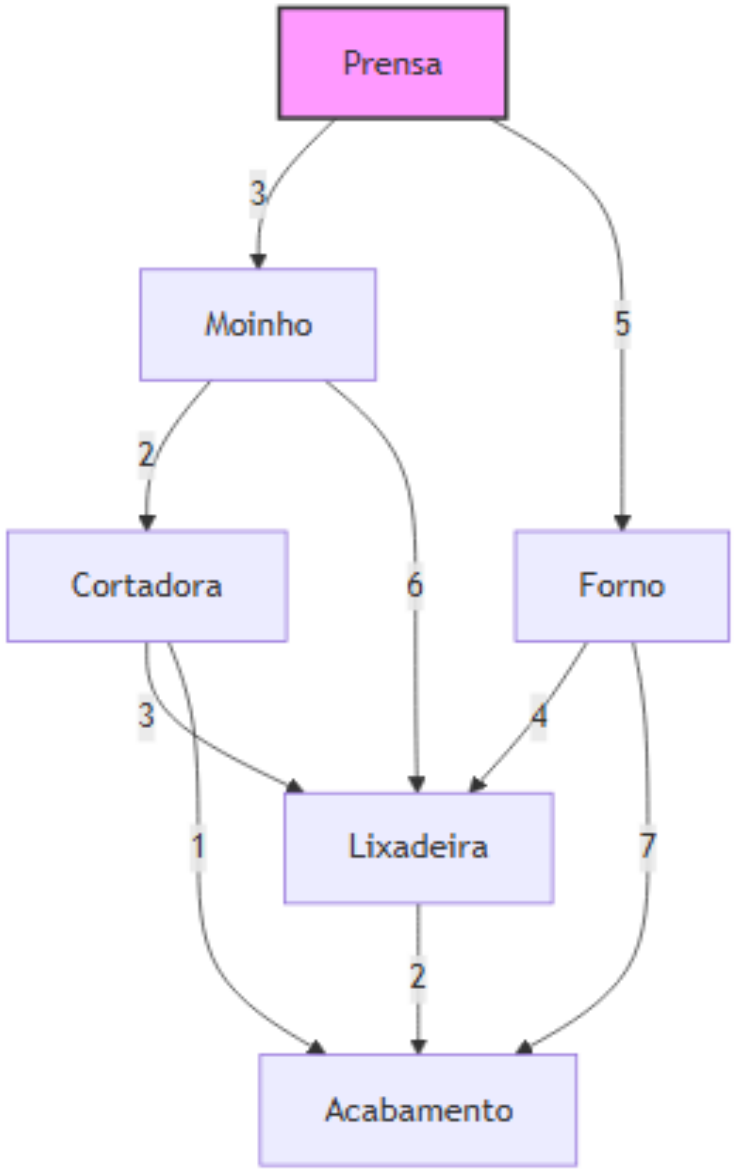
- Ambos os algoritmos produzem a mesma árvore geradora mínima em grafos sem pesos repetidos, garantindo unicidade da solução.
- A escolha entre eles depende do perfil do grafo: Kruskal destaca-se em grafos esparsos, Prim em densos.
- Kruskal beneficia-se da ordenação global e paralelização, facilitando uso em sistemas distribuídos.
- Prim se ajusta melhor a implementações que expandem um componente, como em roteadores de redes.
    - O volume de arestas e a estrutura de dados disponível podem influenciar na performance de cada algoritmo.
    - Fatores como facilidade de paralelização e tipo de armazenamento são determinantes na decisão do método.

Analogia industrial para Prim e Kruskal

- Usando o exemplo de uma fábrica, cada máquina representa um vértice e as conexões possíveis são arestas com custos variados.
- Prim começa em uma máquina e conecta, sempre, a opção mais barata com outra máquina ainda desconectada.
- O crescimento se dá de forma local, conectando progressivamente novas máquinas à rede já formada.
- Kruskal seleciona as conexões mais baratas em todo o grafo, independentemente de sua localização inicial.
    - O critério é sempre evitar ciclos, garantindo que a rede final seja acíclica e conectada ao menor custo.
    - As duas abordagens ilustram diferentes estratégias para resolver o mesmo problema de otimização de conexões.

Exemplo fabril

- O objetivo agora não é encontrar o caminho mais curto de uma máquina inicial (como em Dijkstra ou Bellman-Ford), mas sim conectar todas as máquinas da fábrica com tubos ou esteiras de transporte, gastando o menor custo possível, sem formar ciclos (loops).
- Algoritmo de Prim: "Expandir a partir de uma máquina“.
- Algoritmo de Kruskal: "Escolher as conexões mais baratas primeiro“.



Fonte: Autoria própria.

Exemplo fabril

Resultado Prim – 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.

Exemplo fabril

Resultado Kruskal – 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.

Aplicação prática em sistemas distribuídos

- Árvores geradoras mínimas são fundamentais para projetar redes eficientes em ambientes de microsserviços e nuvem.
- A topologia resultante afeta diretamente a latência global e o custo operacional de sistemas distribuídos.
- O objetivo é equilibrar redundância para resiliência e contenção de custos para otimizar a infraestrutura.
- A espinha dorsal da comunicação pode ser modelada por uma árvore mínima, removendo enlaces desnecessários.
    - A análise desse modelo permite identificar a base eficiente para posterior adição de caminhos redundantes.
    - Em sistemas de larga escala, a árvore mínima serve de ponto de partida para ajustes finos de topologia.

Estrutura das implementações em Python

- Python oferece recursos como heapq e dataclasses que simplificam a implementação dos algoritmos clássicos.
- O grafo é modelado como um dicionário de listas de adjacência e uma lista de arestas para rápida iteração.
- Arestas são objetos imutáveis, garantindo segurança na manipulação e no uso como chaves em dicionários.
- Union-Find é implementado com dicionários para pais e ranks, otimizando a detecção de ciclos em Kruskal.
    - Heap binário serve como fila de prioridade no algoritmo de Prim, assegurando seleções eficientes de arestas.
    - Funções auxiliares permitem carregar grafos de arquivos CSV, testar diferentes densidades e medir tempos de execução.

Eficiência e aplicação dos algoritmos em contextos reais

- Kruskal apresenta complexidade $O(E \log E)$, favorecendo cenários com muitos vértices e poucas arestas.
- Prim exibe desempenho $O(E \log V)$ com fila de prioridade, sendo vantajoso em grafos densos e completos.
- A modularidade e clareza do código facilitam ajustes, extensões e integração com sistemas maiores.
- A implementação serve como base para experimentos, validações conceituais e prototipagem de soluções reais.
    - A escolha do algoritmo deve considerar as características do grafo e o perfil de uso pretendido pela aplicação.
    - Dominar essas técnicas capacita profissionais a otimizar custos e desempenho em redes físicas e digitais.

Interatividade

Assinale a alternativa correta sobre a construção de árvores geradoras mínimas em grafos ponderados utilizando os algoritmos de Kruskal e Prim.

a) Kruskal constrói a árvore geradora mínima expandindo um componente único a partir de um vértice, usando fila de prioridade para escolher a aresta de menor peso.

b) No algoritmo de Prim, as arestas são inicialmente ordenadas por peso, e o algoritmo seleciona as conexões mais baratas, evitando ciclos com estrutura união-busca.

c) Kruskal é mais eficiente em grafos densos, enquanto Prim tem melhor desempenho em grafos esparsos, especialmente com uso de fila de Fibonacci.

d) Em Kruskal, a verificação de ciclos é feita com estrutura união-busca (Union-Find), que une componentes desconexos e mantém a identificação dos conjuntos com eficiência.

e) Ambos os algoritmos exigem ordenação das arestas por peso em cada iteração, resultando sempre em complexidade $O(V^2)$, independentemente do método.

Resposta

Assinale a alternativa correta sobre a construção de árvores geradoras mínimas em grafos ponderados utilizando os algoritmos de Kruskal e Prim.

a) Kruskal constrói a árvore geradora mínima expandindo um componente único a partir de um vértice, usando fila de prioridade para escolher a aresta de menor peso.

b) No algoritmo de Prim, as arestas são inicialmente ordenadas por peso, e o algoritmo seleciona as conexões mais baratas, evitando ciclos com estrutura união-busca.

c) Kruskal é mais eficiente em grafos densos, enquanto Prim tem melhor desempenho em grafos esparsos, especialmente com uso de fila de Fibonacci.

d) Em Kruskal, a verificação de ciclos é feita com estrutura união-busca (Union-Find), que une componentes desconexos e mantém a identificação dos conjuntos com eficiência.

e) Ambos os algoritmos exigem ordenação das arestas por peso em cada iteração, resultando sempre em complexidade $O(V^2)$, independentemente do método.

Introdução à Divisão e Conquista

- O paradigma de divisão e conquista busca decompor problemas complexos em partes menores que se assemelham à questão original, promovendo raciocínio recursivo estruturado e reutilizável.
- Cada subproblema criado por divisão recebe tratamento independente, com suas soluções parciais posteriormente reunidas para formar a resposta global desejada.
- O método é apropriado quando a estrutura dos subproblemas não difere substancialmente do problema maior, facilitando o uso de algoritmos recursivos.
- Um dos benefícios consiste na possibilidade de aplicar estratégias específicas a cada parte, resultando em maior flexibilidade do que abordagens monolíticas.
    - A combinação das respostas individuais requer critério cuidadoso, pois erros nessa etapa comprometem a solução integral.
    - A divisão e conquista pode ser observada em vários problemas clássicos de algoritmos, incluindo busca, ordenação e otimização.

O Problema da Mochila – Conceito

- O problema da mochila consiste em selecionar um subconjunto de objetos, cada um com peso e valor, sem ultrapassar um limite total de peso definido.
- O objetivo é maximizar o valor agregado dos itens escolhidos, respeitando a restrição imposta pela capacidade da mochila.
- Trata-se de um desafio clássico de otimização combinatória com aplicações que vão do transporte à alocação de recursos.
- A formulação permite interpretações variadas, adaptando-se a diferentes contextos, como duração, volume ou custo financeiro.
    - Frequentemente, as soluções exatas são inviáveis para grandes instâncias, justificando o uso de paradigmas como divisão e conquista.
    - A variante mais popular do problema é a chamada “mochila binária”, na qual cada item pode ser selecionado no máximo uma vez.

O Problema da Mochila

Fonte: Imagem produzida pelo autor com tecnologia Grok 3, uma ferramenta de IA desenvolvida pela xAI.

Divisão Inicial do Conjunto de Itens

- Uma aplicação direta de divisão e conquista divide o conjunto de n objetos em dois blocos de tamanho similar.
- Cada bloco é tratado isoladamente, facilitando o cálculo de soluções parciais sem interferência direta entre as partes.
- Esse fracionamento inicial pode ser feito de várias maneiras, desde divisões simétricas até critérios baseados em características dos itens.
- A divisão por metades permite balanceamento do esforço computacional e prepara o terreno para operações subsequentes eficientes.
    - O método se adapta tanto a implementações sequenciais quanto paralelas, dependendo do ambiente de execução.
    - A análise da estrutura dos blocos resultantes orienta o uso de algoritmos adequados para cada subinstância.

Cálculo por Força Bruta em Blocos

- Para cada bloco, todos os subconjuntos possíveis são examinados por força bruta, registrando para cada peso o valor máximo correspondente.
- Esse método resulta em duas listas, cada uma representando as possíveis somas de peso e valor da metade dos itens originais.
- A abordagem por força bruta é viável porque o espaço de busca em cada bloco é reduzido em relação ao problema completo.
- O registro sistemático de pares (peso, valor) evita recomputações e permite fusão eficiente das informações na etapa seguinte.
    - A análise exaustiva em blocos menores reduz o crescimento exponencial observado em abordagens ingênuas aplicadas ao conjunto inteiro.
    - A técnica destaca-se por permitir tratamento independente dos blocos, promovendo modularidade do processo algorítmico.

Analogia – Playlist e Problema da Mochila

- Planejar uma playlist para preencher exatamente uma hora, ou baixar músicas até ocupar quinhentos megabytes, equivale a resolver o problema da mochila.
- Cada música pode ser associada a um item com peso (duração ou tamanho) e valor (pontuação de preferência ou relevância).
- O desafio consiste em maximizar a soma das notas das faixas escolhidas sem ultrapassar o tempo ou espaço disponível.
- Essa modelagem facilita a aplicação de algoritmos de otimização já consolidados para personalização de experiências de usuário.
    - A correspondência entre elementos do problema clássico e o cenário de streaming é direta, possibilitando tradução imediata do raciocínio.
    - Plataformas de música utilizam variações desse modelo para sugerir e ajustar playlists em tempo real para milhões de usuários.

Analogia – Playlist e Problema da Mochila

Fonte: Imagem produzida pelo autor com tecnologia Grok 3, uma ferramenta de IA desenvolvida pela xAI.

As marcas, imagens e fotos aqui analisadas e citadas são meramente exemplificativas para o contexto da aula e pertencem ao Spotify, que é detentor dos direitos de propriedade industrial e intelectual.

Analogia – Playlist e Problema da Mochila

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

@dataclass
class Musica:
    id: str
    duracao: int          # segundos
    nota: int             # preferência 0-100
```

Analogia – Playlist e Problema da Mochila

```
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
```

Analogia – Playlist e Problema da Mochila

```
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]
```

Analogia – Playlist e Problema da Mochila

```
# 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
```

Aplicação Prática na Indústria de Streaming

- Grandes serviços de música como Spotify empregam divisão e conquista para decompor problemas globais em subproblemas administráveis.
- A fragmentação do catálogo em grupos temáticos ou intervalos de duração viabiliza cálculos em paralelo e respostas rápidas.
- A atualização dinâmica de playlists, necessária diante de novos lançamentos, é facilitada pela estrutura modular do algoritmo.
- O modelo permite manter recomendações personalizadas mesmo com catálogos que atingem dezenas de milhões de faixas.
    - Divisão em blocos menores acelera adaptações a preferências individuais sem reprocessar todo o universo de músicas.
    - A estratégia eleva a escalabilidade do serviço, essencial para operar em ambientes com alta concorrência de acessos.

Interatividade

Considerando a estratégia de divisão e conquista aplicada ao problema da mochila, assinale a alternativa que explique corretamente a principal vantagem do método de divisão em dois blocos e posterior fusão das soluções, quando comparado à busca exaustiva tradicional.

a) O método permite sempre ignorar os itens de menor valor, bastando analisar apenas subconjuntos com peso total igual à metade da capacidade da mochila.

b) A divisão em blocos reduz a complexidade do problema para O(n), tornando-o viável mesmo para centenas de itens e pequenas capacidades.

c) A abordagem de divisão em dois blocos possibilita gerar listas menores, eliminar redundâncias por peso e combinar soluções com busca binária, reduzindo significativamente o número de combinações analisadas.

d) A técnica elimina a necessidade de considerar a capacidade da mochila, pois apenas o valor total dos itens é relevante para a fusão dos blocos.

e) O método exige que todos os subproblemas sejam resolvidos sequencialmente, impossibilitando qualquer forma de paralelismo ou execução distribuída.

Resposta

Considerando a estratégia de divisão e conquista aplicada ao problema da mochila, assinale a alternativa que explique corretamente a principal vantagem do método de divisão em dois blocos e posterior fusão das soluções, quando comparado à busca exaustiva tradicional.

a) O método permite sempre ignorar os itens de menor valor, bastando analisar apenas subconjuntos com peso total igual à metade da capacidade da mochila.

b) A divisão em blocos reduz a complexidade do problema para O(n), tornando-o viável mesmo para centenas de itens e pequenas capacidades.

c) A abordagem de divisão em dois blocos possibilita gerar listas menores, eliminar redundâncias por peso e combinar soluções com busca binária, reduzindo significativamente o número de combinações analisadas.

d) A técnica elimina a necessidade de considerar a capacidade da mochila, pois apenas o valor total dos itens é relevante para a fusão dos blocos.

e) O método exige que todos os subproblemas sejam resolvidos sequencialmente, impossibilitando qualquer forma de paralelismo ou execução distribuída.

Fundamentos da Programação Dinâmica

- A programação dinâmica resolve problemas recursivos com subestruturas repetidas, armazenando respostas intermediárias para evitar cálculos redundantes e melhorar a eficiência computacional.
- Esse paradigma parte da decomposição do problema em subinstâncias de formato idêntico, permitindo que a solução global seja construída a partir das soluções ótimas das partes.
- A técnica identifica sobreposição de subproblemas: situações em que certas configurações retornam múltiplas vezes durante a recursão e, se memorizadas, economizam tempo.
- O uso de tabelas para guardar resultados reduz a complexidade de tempo de muitos problemas de exponencial para polinomial, tornando-os tratáveis na prática.
    - A economia de memória é viabilizada porque cada subinstância é resolvida uma única vez, em vez de repetir cálculos para configurações já analisadas.
    - Esse modelo é amplamente utilizado em contextos de otimização e busca de soluções exatas, especialmente onde há recombinação frequente de subproblemas.

Características Estruturais dos Problemas

- A programação dinâmica é adequada para problemas que podem ser quebrados em subinstâncias menores com o mesmo formato estrutural do original.
- A construção da solução global ocorre pela combinação ótima das respostas dos subproblemas resolvidos, mantendo dependência lógica entre etapas.
- O sucesso da técnica exige reconhecimento prévio de subestruturas comuns e definição clara das relações de dependência.
- Não basta quebrar o problema em partes; é fundamental garantir que cada subinstância contribua diretamente para a solução maior.
    - O paradigma é aplicável apenas quando o espaço de subproblemas possíveis é gerenciável, evitando explosão de memória.
    - O uso de tabelas ou dicionários de armazenamento depende do número de combinações relevantes a serem guardadas durante o processo.

A Subsequência Comum Mais Longa (LCS)

- O problema da subsequência comum mais longa busca a maior sequência ordenada de caracteres que aparece em duas cadeias, preservando a ordem relativa.
- A LCS ignora se os símbolos são contíguos, importando apenas que mantenham a sequência, o que permite flexibilidade na comparação.
- Uma abordagem recursiva pura testa todas as remoções possíveis de caracteres, resultando em crescimento exponencial do número de casos.
- Programação dinâmica reduz drasticamente o tempo, explorando apenas combinações essenciais e reutilizando respostas de prefixos analisados.
    - O algoritmo baseia-se em decidir, para cada par de prefixos das cadeias, se o caractere corrente é incluído ou se o melhor resultado é herdado de subproblemas.
    - A resposta é construída progressivamente, avançando nas cadeias, sem voltar a analisar sequências já comparadas anteriormente.

Construção da Tabela Dinâmica

- A solução da LCS utiliza uma tabela bidimensional, cujas dimensões são os tamanhos das cadeias comparadas, preenchida de maneira incremental.
- Cada célula da tabela corresponde à solução ótima para os prefixos terminados naquela posição nas duas cadeias.
- O preenchimento pode ser feito linha por linha ou coluna por coluna, com cada célula recebendo valores das células vizinhas já computadas.
- O algoritmo diferencia dois casos: caracteres iguais (incrementa a resposta anterior) e caracteres diferentes (escolhe o maior valor entre eliminar um ou outro).
    - O custo de tempo e memória da solução é O(n m), onde n e m são os comprimentos das cadeias, tornando a abordagem eficiente para sequências moderadas.
    - A tabela permite recuperar a própria subsequência comum, não apenas seu tamanho, por meio de percurso inverso a partir do resultado final.

Aplicações Práticas da Programação Dinâmica

- Programação dinâmica vai além do problema de LCS, sendo essencial em bioinformática para alinhamento de sequências de DNA e proteínas.
- O paradigma resolve o cálculo de rotas mínimas em grafos, exemplificado pelo algoritmo de Floyd-Warshall em redes com pesos não negativos.
- Problemas de corte de hastes metálicas na indústria utilizam programação dinâmica para maximizar valor a partir de combinações de cortes possíveis.
- Em cada aplicação, identificar padrões de sobreposição e formular a relação de recorrência entre subproblemas é etapa crítica.
    - Python, por sua clareza sintática, facilita a transcrição de recorrências matemáticas para código, garantindo compreensão e depuração.
    - A metodologia transforma problemas aparentemente intratáveis em soluções eficientes, ampliando o leque de desafios solucionáveis.

Vantagens e Limitações da Programação Dinâmica

- A principal vantagem reside na redução do tempo de execução, pois evita recomputar respostas de subinstâncias já resolvidas.
- O consumo de memória, embora menor que a busca exaustiva, ainda depende do número total de subproblemas distintos armazenados.
- A técnica não é aplicável quando não há subproblemas sobrepostos ou quando a decomposição não leva a instâncias de mesmo formato.
- Problemas que exigem apenas análise sequencial ou não possuem estrutura recursiva dificilmente se beneficiam de programação dinâmica.
    - Otimizações adicionais podem ser alcançadas armazenando apenas linhas ou colunas relevantes da tabela, reduzindo o espaço ocupado.
    - O paradigma demanda esforço inicial de modelagem, exigindo compreensão das dependências e da relação entre os subproblemas.

LCS em Redes Sociais – Afinidade de Emojis

- O estudo de afinidade entre usuários pode ser modelado pela LCS, aplicando-se às sequências de reações rápidas com emojis em stories do Instagram.
- O algoritmo busca a maior subsequência ordenada de emojis que aparece nas interações de ambos os perfis, ignorando outros símbolos.
- Um alto valor de LCS sugere sincronia e repertório emocional compartilhado entre os amigos, fortalecendo a percepção de afinidade.
- A análise considera apenas a ordem relativa dos emojis, não a data exata ou a presença de outros emojis intermediários.
    - Tal aplicação demonstra como um conceito algorítmico pode ser usado para quantificar aspectos sociais e de comportamento digital.
    - Técnicas formais, como programação dinâmica, iluminam padrões ocultos na interação cotidiana de usuários em ambientes virtuais.

LCS em Redes Sociais – Afinidade de Emojis

```
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'
```

LCS em Redes Sociais – Afinidade de Emojis

```
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
```

LCS em Redes Sociais – Afinidade de Emojis

```
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
```

LCS em Redes Sociais – Afinidade de Emojis

```
    # 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))
```

Interatividade

Considerando os princípios da programação dinâmica, qual das alternativas descreve corretamente o motivo pelo qual esse paradigma reduz a complexidade temporal de problemas como a subsequência comum mais longa (LCS)?

a) Porque permite quebrar qualquer problema em partes independentes, sem necessidade de armazenar resultados intermediários.

b) Porque evita recalcular soluções de subproblemas já resolvidos, armazenando os resultados em uma tabela para consultas futuras.

c) Porque realiza buscas exaustivas por todas as soluções possíveis, garantindo sempre o melhor resultado ao custo de memória elevado.

d) Porque utiliza exclusivamente abordagens iterativas, tornando recursão desnecessária para todos os problemas.

e) Porque emprega paralelismo automático em todas as etapas, independentemente do problema ou do ambiente de execução.

Resposta

Considerando os princípios da programação dinâmica, qual das alternativas descreve corretamente o motivo pelo qual esse paradigma reduz a complexidade temporal de problemas como a subsequência comum mais longa (LCS)?

a) Porque permite quebrar qualquer problema em partes independentes, sem necessidade de armazenar resultados intermediários.

b) Porque evita recalcular soluções de subproblemas já resolvidos, armazenando os resultados em uma tabela para consultas futuras.

c) Porque realiza buscas exaustivas por todas as soluções possíveis, garantindo sempre o melhor resultado ao custo de memória elevado.

d) Porque utiliza exclusivamente abordagens iterativas, tornando recursão desnecessária para todos os problemas.

e) Porque emprega paralelismo automático em todas as etapas, independentemente do problema ou do ambiente de execução.

Referências

- 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. O'Reilly Media, 2013.
- CORMEN, H. *et al*. *Algoritmos*: teoria e prática. 4. ed. LTC, 2024.
- FURTADO, A. L. *Python*: programação para leigos. Rio de Janeiro: Alta Books, 2021.
- KNUTH, D. *The art of computer programming*. 3. ed. Addison-Wesley Professional, v. 2, 1997.

Referências

- LUTZ, M. *Learning Python*. 5. ed. O'Reilly Media, 2013.
- NILO, L. E. *Introdução à programação com Python*. São Paulo: Novatec, 2019.
- SILVEIRA, B. R. *Desenvolvimento web com Python e Flask*. São Paulo: Casa do Código, 2021.
- VAN ROSSUM, G.; DRAKE, F. L. *Python reference manual*. PythonLabs, 2007.
- VIEIRA, A. *Introdução à ciência da computação com Python*. Rio de Janeiro: LTC, 2020.
- WING, J. M. Computational thinking. *Communications of the ACM*, v. 49, n. 3, p. 33-35, 2006.
- ZAVAGLIA, F. *Lógica de programação*: a construção de algoritmos e estruturas de dados. São Paulo: Érica, 2017.

ATÉ A PRÓXIMA!