Revisão para G1

Reforçar os conteúdos abordados até o momento para a prova.

Márcio Nicolau

2025-09-22

Table of contents

List of Figures

Introdução: Consolidando o Conhecimento

Esta aula especial de revisão apresenta questões discursivas e problemas de programação que cobrem os principais tópicos estudados ao longo do curso. O objetivo é consolidar o entendimento teórico e prático dos conceitos fundamentais de computabilidade e complexidade computacional.

As questões e problemas foram elaborados para integrar conhecimentos de diferentes aulas, promovendo uma compreensão holística da teoria da computação.

Questões Discursivas de Revisão

Questão 1: Classes de Complexidade e Redutibilidade

Contexto: Considere as classes de complexidade P, NP, coNP, PSPACE e EXP, bem como o conceito de redução polinomial ($\leq_p$).

Pergunta: Explique por que a existência de um problema NP-completo que pertence a P implicaria P = NP. Em sua resposta, discuta:

- O papel das reduções polinomiais na estrutura da classe NP
- Por que todos os problemas em NP poderiam ser resolvidos eficientemente
- As implicações práticas para a criptografia moderna se P = NP fosse verdade

> **Orientação**
>
> Sua resposta deve demonstrar compreensão profunda das relações entre classes de complexidade e o significado de NP-completude.

Questão 2: Estrutura e Componentes da Máquina de Turing

Contexto: A Máquina de Turing é o modelo computacional fundamental que define o que pode ser computado algoritmicamente.

Pergunta: Descreva os componentes essenciais de uma Máquina de Turing e explique como eles trabalham juntos para realizar computações. Em sua resposta:

- Liste e explique a função de cada componente (fita, cabeçote, conjunto de estados, função de transição)
- Descreva o processo de execução passo a passo de uma computação simples
- Explique por que a fita infinita é importante para o poder computacional da máquina

Questão 3: Máquinas de Turing e Church-Turing

Contexto: A Tese de Church-Turing estabelece que qualquer função efetivamente computável pode ser computada por uma Máquina de Turing.

Pergunta: Um estudante propõe um novo modelo de computação chamado "Máquina Quântica Hipotética" (MQH) que supostamente pode resolver o Problema da Parada. Explique:

- Por que isso violaria a Tese de Church-Turing
- Como você poderia usar a técnica de diagonalização para mostrar que mesmo a MQH teria limitações
- A diferença entre aumentar o poder computacional prático versus teórico

Questão 4: Enumerabilidade e Decidibilidade

Contexto: As classes R (decidível), RE (reconhecível) e co-RE formam uma hierarquia importante na teoria da computabilidade.

Pergunta: Considere uma linguagem L que é RE mas não R, como o Problema da Parada. Prove que o complemento de L (denotado co-L) não pode ser RE. Em sua prova:

- Use a relação entre decidibilidade e reconhecibilidade
- Explique o que aconteceria se tanto L quanto co-L fossem RE
- Forneça intuição sobre por que isso faz sentido computacionalmente

Questão 5: Aplicações Práticas da Teoria

Contexto: Embora a teoria da computação seja abstrata, ela tem implicações profundas para a prática da computação.

Pergunta: Discuta três situações práticas em desenvolvimento de software onde o conhecimento sobre indecidibilidade e complexidade computacional influencia decisões de design. Para cada situação:

- Identifique o problema teórico relacionado
- Explique como a limitação teórica se manifesta na prática
- Descreva estratégias práticas para contornar essas limitações

Problemas de Programação

Problema 1: Simulador de Enumeração Diagonal

Objetivo: Implementar um simulador que demonstra a técnica de diagonalização para provar que certas linguagens não são enumeráveis.

```
from typing import List, Set, Callable
import itertools

class DiagonalizacaoSimulator:
    """
    Simula a prova por diagonalização para linguagens sobre alfabeto binário.
    Demonstra a construção do elemento antagonista.
    """

    def __init__(self, alfabeto: List[str] = ['0', '1']):
        self.alfabeto = alfabeto
        self.strings = self._gerar_strings()

    def _gerar_strings(self):
        """Gera todas as strings em ordem canônica (por tamanho, depois lexicográfica)."""
        yield ''  # String vazia
        for length in itertools.count(1):
            for combo in itertools.product(self.alfabeto, repeat=length):
                yield ''.join(combo)

    def enumerar_strings(self, limite: int = 10) -> List[str]:
```

```
        """Retorna as primeiras 'limite' strings na enumeração."""
        gen = self._gerar_strings()
        return [next(gen) for _ in range(limite)]

    def simular_linguagens(self, linguagens: List[Set[str]], max_strings: int = 8):
        """
        Simula uma tabela de linguagens vs strings.
        Retorna a tabela e a linguagem diagonal construída.
        """
        strings = self.enumerar_strings(max_strings)

        # Criar tabela de pertinência
        tabela = []
        for i, lang in enumerate(linguagens[:max_strings]):
            linha = []
            for j, s in enumerate(strings):
                pertence = 1 if s in lang else 0
                linha.append(pertence)
            tabela.append(linha)

        # Construir linguagem diagonal (antagonista)
        linguagem_diagonal = set()
        for i in range(min(len(linguagens), max_strings)):
            if i < len(strings):
                # Inverte a diagonal: se Li contém si, LD não contém, e vice-versa
                if tabela[i][i] == 0:
                    linguagem_diagonal.add(strings[i])

        return tabela, linguagem_diagonal, strings

    def verificar_diferenca(self, linguagens: List[Set[str]],
                            linguagem_diagonal: Set[str],
                            strings: List[str]) -> List[int]:
        """
        Verifica em quais posições a linguagem diagonal difere de cada linguagem.
        Retorna os índices onde LD difere de cada Li.
        """
        diferencas = []
        for i, lang in enumerate(linguagens[:len(strings)]):
            if i < len(strings):
                s_i = strings[i]
                # LD difere de Li exatamente na string si
                if (s_i in lang) != (s_i in linguagem_diagonal):
                    diferencas.append(i)
        return diferencas
```

```
# TAREFA: Complete a implementação e teste o simulador
def main():
    """
    Tarefa: Implemente um exemplo completo que:
    1. Crie pelo menos 5 linguagens diferentes
    2. Execute a simulação de diagonalização
    3. Mostre que a linguagem diagonal é diferente de todas as linguagens originais
    4. Visualize a tabela de forma clara, destacando a diagonal
    """

    # Exemplo inicial (complete e expanda)
    sim = DiagonalizacaoSimulator()

    # Defina suas linguagens aqui
    L1 = {'', '0', '00'}  # Strings de apenas zeros
    L2 = {'1', '11', '111'}  # Strings de apenas uns
    # TODO: Adicione mais linguagens

    linguagens = [L1, L2]  # TODO: Adicione as linguagens criadas

    # Execute a simulação
    tabela, ld, strings = sim.simular_linguagens(linguagens)

    # TODO: Implemente visualização e verificação
    print("Implementação a ser completada...")

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

```
Implementação a ser completada...
```

> Objetivos de Aprendizagem
>
> Este problema ajuda a: - Entender concretamente como funciona a diagonalização - Visualizar a construção do elemento antagonista - Compreender por que o método garante que o elemento construído é diferente de todos na lista

Problema 2: Verificador de Redutibilidade e NP-Completude

Objetivo: Implementar um sistema que simula reduções entre problemas e verifica propriedades de NP-completude.

```
from typing import Dict, List, Tuple, Optional, Set
from dataclasses import dataclass
import networkx as nx
```

```python
import matplotlib.pyplot as plt

@dataclass
class Problema:
    """Representa um problema computacional."""
    nome: str
    classe: str  # 'P', 'NP', 'NP-completo', etc.
    descricao: str

@dataclass
class Reducao:
    """Representa uma redução polinomial entre dois problemas."""
    origem: str
    destino: str
    transformacao: Optional[Callable] = None
    tempo_polinomial: bool = True

class RedeReducoes:
    """
    Gerencia uma rede de problemas e suas reduções.
    Permite verificar propriedades como NP-completude.
    """

    def __init__(self):
        self.problemas: Dict[str, Problema] = {}
        self.reducoes: List[Reducao] = []
        self.grafo = nx.DiGraph()

    def adicionar_problema(self, problema: Problema):
        """Adiciona um problema ao sistema."""
        self.problemas[problema.nome] = problema
        self.grafo.add_node(problema.nome)

    def adicionar_reducao(self, reducao: Reducao):
        """Adiciona uma redução entre dois problemas."""
        if reducao.origem not in self.problemas or reducao.destino not in self.problemas:
            raise ValueError("Problemas devem ser adicionados antes das reduções")

        self.reducoes.append(reducao)
        self.grafo.add_edge(reducao.origem, reducao.destino)

    def existe_caminho_reducao(self, origem: str, destino: str) -> bool:
        """Verifica se existe uma cadeia de reduções de origem para destino."""
        return nx.has_path(self.grafo, origem, destino)
```

```python
    def verificar_np_completude(self, problema: str) -> Tuple[bool, str]:
        """
        Verifica se um problema é NP-completo baseado em:
        1. Está em NP
        2. Todo problema em NP reduz a ele (via transitividade)
        """
        if problema not in self.problemas:
            return False, "Problema não encontrado"

        p = self.problemas[problema]

        # Verificar se está em NP
        if p.classe not in ['NP', 'NP-completo']:
            return False, f"Problema está em {p.classe}, não em NP"

        # Verificar se existe algum NP-completo conhecido que reduz a ele
        np_completos_conhecidos = [nome for nome, prob in self.problemas.items()
                                   if prob.classe == 'NP-completo']

        if not np_completos_conhecidos:
            return False, "Nenhum problema NP-completo conhecido no sistema"

        # Se algum NP-completo reduz a este problema, ele também é NP-completo
        for npc in np_completos_conhecidos:
            if self.existe_caminho_reducao(npc, problema):
                return True, f"NP-completo via redução de {npc}"

        return False, "Não foi possível estabelecer NP-completude"

    def simular_reducao_3sat_clique(self, formula_cnf: List[List[str]]) -> Tuple[Set, int]:
        """
        Simula a redução clássica de 3-SAT para CLIQUE.

        Args:
            formula_cnf: Lista de cláusulas, cada cláusula é lista de literais

        Returns:
            Grafo como conjunto de arestas e valor k para o problema CLIQUE
        """
        vertices = []
        for i, clausula in enumerate(formula_cnf):
            for literal in clausula:
                vertices.append((i, literal))

        # Criar arestas: conectar vértices de cláusulas diferentes
```

```
        # que não são literais contraditórios
        arestas = set()
        for v1 in vertices:
            for v2 in vertices:
                if v1 != v2:
                    clausula1, lit1 = v1
                    clausula2, lit2 = v2

                    # Vértices de cláusulas diferentes
                    if clausula1 != clausula2:
                        # Literais não contraditórios
                        if not (lit1 == f"¬{lit2}" or lit2 == f"¬{lit1}"):
                            if lit1.replace('¬', '') != lit2.replace('¬', '') or lit1 == lit2:
                                arestas.add(tuple(sorted([str(v1), str(v2)])))

        k = len(formula_cnf)  # Precisamos de um clique de tamanho m (número de cláusulas)
        return arestas, k

    def visualizar_rede(self):
        """Visualiza a rede de reduções como um grafo direcionado."""
        plt.figure(figsize=(12, 8))

        # Definir cores baseadas nas classes
        color_map = {
            'P': 'lightgreen',
            'NP': 'lightblue',
            'NP-completo': 'lightcoral',
            'PSPACE': 'lightyellow'
        }

        node_colors = [color_map.get(self.problemas[node].classe, 'lightgray')
                       for node in self.grafo.nodes()]

        pos = nx.spring_layout(self.grafo, k=2, iterations=50)
        nx.draw(self.grafo, pos, node_color=node_colors, with_labels=True,
                node_size=2000, font_size=10, font_weight='bold',
                arrows=True, arrowsize=20, edge_color='gray')

        plt.title("Rede de Reduções entre Problemas")
        plt.axis('off')
        plt.tight_layout()
        return plt

# TAREFA: Complete a implementação
def main():
```

```
    """
    Tarefa: Implemente um exemplo completo que:
    1. Crie uma rede com pelo menos 7 problemas conhecidos (SAT, 3-SAT, CLIQUE, etc.)
    2. Adicione as reduções clássicas entre eles
    3. Verifique a NP-completude de diferentes problemas
    4. Simule pelo menos uma redução concreta (ex: 3-SAT para CLIQUE)
    5. Visualize a rede de reduções
    """

    rede = RedeReducoes()

    # Exemplo inicial (complete e expanda)
    sat = Problema("SAT", "NP-completo", "Satisfatibilidade booleana")
    three_sat = Problema("3-SAT", "NP-completo", "SAT com 3 literais por cláusula")
    clique = Problema("CLIQUE", "NP", "Encontrar clique de tamanho k")

    rede.adicionar_problema(sat)
    rede.adicionar_problema(three_sat)
    rede.adicionar_problema(clique)

    # Adicione reduções
    rede.adicionar_reducao(Reducao("SAT", "3-SAT"))
    rede.adicionar_reducao(Reducao("3-SAT", "CLIQUE"))

    # TODO: Adicione mais problemas e reduções
    # Sugestões: VERTEX-COVER, INDEPENDENT-SET, HAMILTONIAN-CYCLE, TSP, SUBSET-SUM

    # Teste a redução 3-SAT para CLIQUE
    formula = [['x', 'y', 'z'], ['¬x', 'y', '¬z'], ['x', '¬y', 'z']]
    arestas, k = rede.simular_reducao_3sat_clique(formula)

    print(f"Fórmula 3-SAT: {formula}")
    print(f"Grafo resultante tem {len(arestas)} arestas")
    print(f"Procurando clique de tamanho {k}")

    # TODO: Complete a implementação
    print("\nImplementação a ser completada...")

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

```
Fórmula 3-SAT: [['x', 'y', 'z'], ['¬x', 'y', '¬z'], ['x', '¬y', 'z']]
Grafo resultante tem 21 arestas
Procurando clique de tamanho 3
```

```
Implementação a ser completada...
```

Objetivos de Aprendizagem

Este problema ajuda a: - Compreender a estrutura das reduções polinomiais - Visualizar a rede de relações entre problemas NP-completos - Implementar uma redução clássica (3-SAT → CLIQUE) - Entender como a NP-completude se propaga via reduções

Verificação de Aprendizado

Questão de Síntese

Após completar as questões discursivas e os problemas de programação, reflita sobre a seguinte questão integradora:

Como a teoria da computação, desde os conceitos de enumerabilidade até as classes de complexidade, forma uma estrutura coerente para entender os limites fundamentais da computação? Em sua resposta, conecte:

1. O papel da diagonalização em estabelecer limites absolutos
2. A hierarquia de classes (R, RE, co-RE) e sua relação com decidibilidade
3. A transição de questões de computabilidade para questões de eficiência (P vs NP)
4. As implicações práticas desses resultados teóricos para a ciência da computação moderna

Reflexão Final

A teoria da computação não apenas nos diz o que é possível computar, mas também nos ensina a reconhecer e respeitar os limites fundamentais da computação. Essa compreensão é essencial para todo cientista da computação, pois informa tanto o design de algoritmos quanto a arquitetura de sistemas computacionais.

Recursos Adicionais

Para aprofundar seus estudos:

- Revisite as provas formais nas aulas anteriores
- Implemente variações dos problemas de programação propostos
- Explore as conexões entre diferentes técnicas de prova
- Investigue aplicações práticas dos conceitos teóricos em sua área de interesse

Referências

As questões e problemas desta revisão integram conceitos apresentados em:

- Sipser (2012) - Introduction to the Theory of Computation
- Aulas 1-9 do curso de Computabilidade e Complexidade
- Exercícios clássicos da literatura de teoria da computação

SIPSER, Michael. **Introdução à Teoria da Computação**. 3. ed. São Paulo, Brasil: Cengage Learning, 2012.