Trabalho 7 TC 2023-2

Aluno: Heitor Freitas Ferreira [11921BCC026]

Ps.: Ao exportar para pdf, os simbolos LaTeX não são renderizados corretamente. Por isso, estou enviando o arquivo .md também.

Prova de graduação em Ciência da Computação – 06/11/2018

1

No caso das linguagens regulares, elas podem ser reconhecidas por máquinas de Turing determinísticas simples conhecidas como autômatos finitos. Um autômato finito é uma máquina de estados que percorre um conjunto finito de estados de acordo com a entrada recebida, e aceita ou rejeita uma cadeia de entrada com base nesse percurso.

2

Se existe uma máquina de Turing que decide uma linguagem, podemos criar um procedimento efetivo para enumerar as palavras dessa linguagem em ordem lexicográfica crescente. Vamos supor que temos uma máquina de Turing (M) que decide a linguagem (L). Aqui está o procedimento para enumerar as palavras de (L) em ordem lexicográfica crescente:

1. Inicialize uma variável (n) para representar o tamanho da palavra a ser gerada. Comece com (n = 0).
2. Para cada palavra de comprimento (n), faça o seguinte:
   a. Gere todas as palavras de comprimento (n) de acordo com a ordem lexicográfica.
   b. Execute a máquina de Turing (M) em cada uma dessas palavras.
   c. Se (M) aceitar a palavra, imprima a palavra.
3. Incremente (n) para o próximo tamanho de palavra e repita o passo 2.

Este procedimento efetivo é baseado na existência da máquina de Turing que decide a linguagem (L). Ele gera e testa todas as palavras possíveis de (L) em ordem lexicográfica crescente. Quando uma palavra é aceita pela máquina de Turing, ela é impressa, garantindo que todas as palavras aceitas pela linguagem sejam enumeradas na ordem correta.

No entanto, é importante notar que, dependendo da linguagem e da máquina de Turing que a decide, esse procedimento pode ser impraticável em termos de tempo e espaço. Além disso, para linguagens infinitas, como as linguagens regulares, esse procedimento efetivo nunca terminará de enumerar todas as palavras da linguagem, embora possa gerar uma quantidade arbitrária de palavras pertencentes à linguagem.

3

"As linguagens reconhecidas por um procedimento efetivo são as linguagens computáveis, ou seja, aquelas que podem ser decididas por uma máquina de Turing ou qualquer modelo computacional equivalente, como máquinas de registro, sistemas de Post, máquinas de células, entre outros."

4

Uma linguagem que não pertence à classe de decidibilidade RE (Recursivamente Enumerável) é uma linguagem que não pode ser reconhecida por uma máquina de Turing em um tempo finito. Em termos formais, uma linguagem (L) não pertence à classe RE se não existe uma máquina de Turing que, quando dada uma entrada (w), pare de aceitar se (w) pertence a (L) e nunca aceite se (w) não pertence a (L).

Um exemplo específico de uma linguagem que não pertence à classe RE é a linguagem (L_{\text{halt}}), que consiste de todas as descrições de máquinas de Turing (M) e entradas (w) tais que (M) pára de executar em (w). Em outras palavras, (L_{\text{halt}}) contém todas as instâncias em que uma máquina de Turing para de computar sobre uma entrada específica. Esta linguagem é conhecida como o problema da parada e é bem conhecida por ser não-recursivamente enumerável.

Formalmente, (L_{\text{halt}} = { \langle M, w \rangle \mid \text{M pára em } w })

Aqui, (\langle M, w \rangle) é a codificação de uma máquina de Turing (M) e uma entrada (w) em uma representação adequada para uma máquina de Turing. A linguagem (L_{\text{halt}}) é não-recursivamente enumerável porque não existe um procedimento efetivo para determinar se uma máquina de Turing pára em uma entrada dada. Isso faz com que (L_{\text{halt}}) não pertença à classe RE.

5

Para provar que a linguagem (H = { \langle M, w \rangle \mid M \text{ para em } w }) não é decidível, usaremos a técnica da redução da linguagem universal (LU = { \langle M, w \rangle \mid M \text{ aceita } w }), que já foi estabelecido como não decidível.

A ideia é mostrar que se (H) fosse decidível, então (LU) também seria decidível, o que é uma contradição, já que sabemos que (LU) não é decidível.

Suponha por absurdo que (H) seja decidível por uma máquina de Turing (D). Então, podemos construir uma máquina de Turing (E) que decide (LU), usando (D) como um subcomponente. Aqui está a descrição de (E):

1. Recebe uma entrada (\langle M, w \rangle).
2. Constrói uma nova máquina de Turing (M') que:
    - Simula (M) em (w).
    - Se (M) aceita (w), (M') entra em um loop infinito (não para).
    - Caso contrário, (M') para.
3. Executa a máquina de Turing (D) na entrada (\langle M', w \rangle).
4. Se (D) aceita (\langle M', w \rangle), então (E) rejeita (\langle M, w \rangle).
5. Se (D) rejeita (\langle M', w \rangle), então (E) aceita (\langle M, w \rangle).

Agora, vamos analisar as possibilidades:

- Se (M) aceita (w), então (M') não para em (w), portanto (\langle M', w \rangle) não pertence a (H). Isso significa que (D) deve rejeitar (\langle M', w \rangle), e consequentemente, (E) deve aceitar (\langle M, w \rangle).
- Se (M) não aceita (w), então (M') para em (w), portanto (\langle M', w \rangle) pertence a (H). Isso significa que (D) deve aceitar (\langle M', w \rangle), e consequentemente, (E) deve rejeitar (\langle M, w \rangle).

Então, (E) decide (LU), o que contradiz a hipótese de que (LU) não é decidível. Portanto, concluímos que (H) não pode ser decidível, já que isso levaria a uma contradição com a não decidibilidade de (LU).

6

Vamos descrever a Máquina de Turing determinística de fita infinita dos 2 lados que aceita palavras com um número igual de A's e B's em qualquer ordem, com (\Sigma = {A, B}) e (\Gamma = {A, B, X, #}).

A máquina terá os seguintes estados:

- (q_0): estado inicial onde a leitura começa.
- (q_1): estado para marcar o primeiro símbolo como A.
- (q_2): estado para marcar o primeiro símbolo como B.
- (q_3): estado para procurar a próxima letra A ou B.
- (q_4): estado para marcar a próxima letra A como X se a primeira letra for A.
- (q_5): estado para marcar a próxima letra B como X se a primeira letra for B.
- (q_6): estado para verificar se a palavra é aceita ou não.

As transições serão as seguintes:

1. No estado (q_0), ao ler A, vá para (q_1) e substitua A por X.
2. No estado (q_0), ao ler B, vá para (q_2) e substitua B por X.
3. No estado (q_1), ao ler A, volte para (q_0) e substitua A por X.
4. No estado (q_1), ao ler B, vá para (q_3) e substitua B por X.
5. No estado (q_2), ao ler A, vá para (q_3) e substitua A por X.
6. No estado (q_2), ao ler B, volte para (q_0) e substitua B por X.
7. No estado (q_3), ao ler A, vá para (q_4) e substitua A por X.
8. No estado (q_3), ao ler B, vá para (q_5) e substitua B por X.
9. No estado (q_4), ao ler A, volte para (q_3) e substitua A por X.
10. No estado (q_4), ao ler B, volte para (q_3) e substitua B por X.
11. No estado (q_5), ao ler A, volte para (q_3) e substitua A por X.
12. No estado (q_5), ao ler B, volte para (q_3) e substitua B por X.
13. No estado (q_3), ao ler X, vá para (q_6).
14. No estado (q_6), ao ler X, volte para (q_6) até encontrar #.

A sequência de configurações para a palavra de entrada ABBA é a seguinte:

1. #ABBA# (estado (q_0))
2. #XBBA# (estado (q_1))
3. #XXBA# (estado (q_3))
4. #XXXA# (estado (q_4))
5. #XXXA# (estado (q_3))
6. #XXX#A# (estado (q_4))
7. #XXX#X# (estado (q_6))

Neste ponto, a máquina para na configuração final e aceita a palavra ABBA, pois o número de A's e B's é o mesmo. Note que a representação da fita é simplificada aqui, com "#" marcando os extremos da fita e "X" marcando os símbolos substituídos durante o processamento.

Prova de graduação em Ciência da Computação – 11/12/2018

1

A classe de problemas práticos encontrados em Ciência da Computação que é formalizada por máquinas de Turing não determinísticas de complexidade polinomial em tempo é a classe NP (Non-deterministic Polynomial time).

Essa classe engloba problemas para os quais podemos verificar a solução de maneira eficiente (ou seja, em tempo polinomial), mas não necessariamente podemos encontrar a solução de maneira eficiente. Em outras palavras, se alguém nos der uma solução para um problema em NP, podemos verificar se essa solução está correta em um tempo que é uma função polinomial do tamanho da entrada.

O impacto de representar problemas dessa classe por uma máquina de Turing determinística seria significativo em termos de complexidade. Se considerarmos que a classe NP é uma classe de problemas para os quais podemos verificar soluções em tempo polinomial, mas não podemos necessariamente encontrar soluções em tempo polinomial, isso implica que, para cada problema em NP, não existe uma máquina de Turing determinística que possa encontrar soluções em tempo polinomial.

Isso significa que, se os problemas em NP fossem representados por uma máquina de Turing determinística, a complexidade desses problemas passaria de polinomial para exponencial (ou possivelmente até mesmo mais alta). Isso ocorre porque, em geral, encontrar soluções para problemas em NP requer uma busca exaustiva por todas as possíveis soluções, o que levaria a um aumento significativo no tempo de execução se fosse feito de maneira determinística.

Em resumo, o impacto em termos de complexidade seria bastante negativo se os problemas em NP fossem representados por uma máquina de Turing determinística, pois isso resultaria em um aumento drástico no tempo de execução para resolver esses problemas.

2

A classe NP-Hard é uma classe de problemas de decisão na teoria da complexidade computacional que é pelo menos tão difícil quanto os problemas mais difíceis em NP. Formalmente, um problema de decisão (L) é NP-Hard se, para todo problema (L') em NP, existe uma redução polinomial de (L') para (L).

Em termos mais simples, um problema (L) é NP-Hard se a solução para qualquer problema em NP pode ser transformada em uma instância de (L) em tempo polinomial. No entanto, um problema NP-Hard não precisa ser em si mesmo um problema em NP.

Por exemplo, o problema do caixeiro viajante (TSP) é um problema NP-Hard. Isso significa que, mesmo que não esteja diretamente na classe NP (não se pode verificar rapidamente uma solução proposta), qualquer problema em NP pode ser reduzido ao problema do caixeiro viajante em tempo polinomial. Isso mostra a dificuldade do TSP e sua relação com problemas em NP.

Portanto, a classe NP-Hard é uma classe de problemas que representa problemas tão difíceis quanto os mais difíceis em NP, mas que não necessariamente pertencem à classe NP.

3

A classe NP-Completa é uma classe especial de problemas na teoria da complexidade computacional que contém os problemas mais difíceis em NP e ao mesmo tempo é NP-Hard. Em outras palavras, um problema (L) é NP-Completo se ele satisfaz duas condições:

1. (L) é em NP, ou seja, uma solução para (L) pode ser verificada em tempo polinomial.
2. Qualquer problema (L') em NP pode ser reduzido polinomialmente a (L), ou seja, existe uma redução polinomial de (L') para (L).

Em resumo, um problema é NP-Completo se for pelo menos tão difícil quanto os problemas mais difíceis em NP e ao mesmo tempo pertencer à classe NP. A importância dos problemas NP-Completos reside no fato de que se um único problema NP-Completo puder ser resolvido em tempo polinomial, então todos os problemas em NP também podem ser resolvidos em tempo polinomial, tornando-se P = NP.

Um exemplo clássico de um problema NP-Completo é o problema do circuito hamiltoniano (HC). Este problema pergunta se existe um ciclo que visite cada vértice de um grafo exatamente uma vez. O HC é em NP porque podemos verificar rapidamente se um ciclo proposto é de fato um circuito hamiltoniano. Além disso, é NP-Hard, o que significa que qualquer problema em NP pode ser reduzido a ele em tempo polinomial.

Portanto, a classe NP-Completa desempenha um papel fundamental na teoria da computação, pois representa a fronteira entre os problemas que podem ser resolvidos de maneira eficiente (em NP) e aqueles que são teoricamente difíceis de resolver (NP-Hard).

4

Duas classes de equivalência polinomial importantes na teoria da complexidade computacional são as classes P e NP. Abaixo, vou fornecer a definição de cada uma dessas classes e explicar por que são consideradas classes de equivalência polinomial.

1. Classe P: A classe P é composta por problemas de decisão que podem ser resolvidos em tempo polinomial por uma máquina de Turing determinística. Formalmente, um problema (L) está em P se existe uma máquina de Turing determinística que decide (L) em tempo (O(n^k)), onde (n) é o tamanho da entrada e (k) é uma constante.
   Exemplo de problema em P: Verificar se um número é primo. Embora o algoritmo de verificação de primalidade possa ser complexo, sua complexidade é polinomial no tamanho da entrada, tornando-o um problema em P.

2. Classe NP: A classe NP é composta por problemas de decisão para os quais uma solução proposta pode ser verificada em tempo polinomial por uma máquina de Turing não determinística. Formalmente, um problema (L) está em NP se existir uma máquina de Turing não determinística que, dada uma solução proposta para (L), possa verificar se a solução é correta em tempo (O(n^k)), onde (n) é o tamanho da entrada e (k) é uma constante.

Exemplo de problema em NP: O problema do caixeiro viajante (TSP). Embora encontrar a solução para o TSP possa ser difícil, podemos verificar rapidamente se uma solução proposta visita todos os pontos uma vez e retorna ao ponto de partida, o que torna o problema um membro de NP.

Ambas as classes, P e NP, são consideradas classes de equivalência polinomial porque problemas em P podem ser reduzidos a problemas em NP em tempo polinomial e vice-versa. Isso significa que, se pudermos resolver um problema em NP em tempo polinomial, podemos resolver todos os problemas em P em tempo polinomial e vice-versa, estabelecendo uma equivalência entre essas classes em termos de complexidade computacional.

5

Para mostrar que as linguagens regulares pertencem à classe P, precisamos demonstrar que podemos decidir qualquer linguagem regular em tempo polinomial. Uma linguagem é considerada regular se pode ser reconhecida por um autômato finito determinístico (DFA) ou não determinístico (NFA).

Vamos primeiro considerar o caso de um autômato finito determinístico (DFA). Um DFA é uma máquina de Turing determinística que possui um número finito de estados e transições bem definidas para cada símbolo do alfabeto de entrada. Dado um DFA que reconhece uma linguagem regular, podemos construir uma máquina de Turing determinística que simula o DFA em tempo polinomial.

A máquina de Turing determinística que simula um DFA opera da seguinte maneira:

1. Recebe a entrada na fita.
2. Inicia no estado inicial do DFA.
3. Para cada símbolo de entrada, transita para o próximo estado conforme as transições do DFA.
4. Se a sequência de entrada for completamente processada e o estado atual da máquina corresponder a um estado final do DFA, aceite a cadeia; caso contrário, rejeite a cadeia.

Como o número de estados de um DFA é finito e fixo, a máquina de Turing determinística que simula o DFA opera em tempo polinomial, pois o número de passos necessários para processar a entrada é limitado pelo número de estados do DFA.

Além disso, podemos generalizar esse argumento para autômatos finitos não determinísticos (NFA). Embora um NFA possa ter múltiplos caminhos de computação para uma mesma entrada, podemos converter um NFA em um DFA equivalente usando o algoritmo de subconjuntos, o que nos permite aplicar o mesmo raciocínio para a classe P.

Portanto, as linguagens regulares pertencem à classe P, pois podem ser decididas em tempo polinomial por uma máquina de Turing determinística que simula um DFA ou um NFA equivalente.

6

As classes de equivalência polinomial, como P e NP, têm um significado prático muito importante na teoria da complexidade computacional e na prática da resolução de problemas computacionais. Aqui está o significado dessas classes na prática:

1. **Classe P (Problemas Polinomiais):**
    - **Eficiência em tempo:** Problemas em P são aqueles para os quais existem algoritmos eficientes que podem ser executados em tempo polinomial em relação ao tamanho da entrada. Isso significa que, na prática, esses problemas podem ser resolvidos de forma rápida, mesmo para entradas de tamanho considerável.
    - **Aplicações práticas:** Muitos problemas do mundo real que podem ser formulados como problemas de decisão (ou seja, que possuem uma resposta "sim" ou "não") estão em P. Por exemplo, verificar se um número é primo, ordenar uma lista de números, encontrar o caminho mais curto em um grafo com pesos não negativos (como no algoritmo de Dijkstra) são problemas em P.
2. **Classe NP (Problemas Não Determinísticos Polinomiais):**

- **Verificação de soluções:** Problemas em NP são aqueles para os quais, se uma solução proposta é dada, podemos verificar rapidamente se essa solução está correta ou não em tempo polinomial. Embora encontrar a solução possa ser difícil, verificar se uma solução proposta é correta é fácil.
- **Desafio prático:** Muitos problemas de otimização e de busca de soluções estão em NP. Por exemplo, o problema do caixeiro viajante (TSP), o problema da mochila (knapsack problem), e problemas relacionados a grafos e programação inteira são exemplos de problemas em NP.
- **Criptografia e segurança:** A classe NP também é relevante em criptografia. Problemas como o problema do logaritmo discreto e o problema do logaritmo discreto em curvas elípticas são NP-completos, o que significa que, se alguém encontrasse uma solução eficiente para esses problemas, isso teria implicações significativas na criptografia.

3. **Classes NP-Completa e NP-Hard:**

- **Complexidade comparativa:** As classes NP-Completa e NP-Hard são importantes porque representam problemas que são tão difíceis quanto os problemas mais difíceis em NP. Resolver um único problema NP-Completo em tempo polinomial provaria que P = NP, o que teria enormes implicações em muitas áreas da computação, incluindo criptografia, otimização e inteligência artificial.
- **Limitações práticas:** A existência de problemas NP-Completo e NP-Hard também coloca limitações práticas sobre o que podemos resolver eficientemente em tempo polinomial. Se esses problemas não podem ser resolvidos de maneira eficiente, isso nos lembra que há limites para a capacidade computacional, e que alguns problemas podem ser intrinsecamente difíceis de resolver.

7

Para mostrar que o problema do circuito mais longo é NP-Completo (NPC), primeiro precisamos mostrar que ele está em NP (problemas que podem ser verificados em tempo polinomial) e então demonstrar que é NP-Difícil, ou seja, que é pelo menos tão difícil quanto o problema do circuito Hamiltoniano, que é NPC.

1. **Está em NP:**

- Dado um circuito fechado (C) no grafo (G) e um número (J), podemos verificar em tempo polinomial se (C) é válido (não passa duas vezes pelo mesmo vértice) e se a soma dos comprimentos dos arcos é maior ou igual a (J). Basta percorrer (C) e verificar se cada vértice é visitado no máximo uma vez e se a soma dos comprimentos dos arcos é maior ou igual a (J).
- Portanto, o problema do circuito mais longo está em NP.

2. **É NP-Difícil:**

- Vamos reduzir o problema do circuito Hamiltoniano (HC) ao problema do circuito mais longo.
- Dado um grafo (G) para o problema HC, criamos um grafo (G') para o problema do circuito mais longo da seguinte maneira:
  - Para cada vértice (v) em (G), duplicamos (v) em (G') para formar (v_1) e (v_2).
  - Para cada arco ((u, v)) em (G) com comprimento (l(u, v)), criamos dois arcos em (G'): ((u_1, v_1)) com comprimento (l(u, v)) e ((u_2, v_2)) com comprimento (0).
  - Para cada vértice (v) em (G), adicionamos um arco ((v_1, v_2)) em (G') com comprimento (0).

- Agora, podemos mostrar que (G) tem um circuito Hamiltoniano se e somente se (G') tem um circuito fechado com soma dos comprimentos dos arcos maior ou igual a (0).
  - Se (G) tem um circuito Hamiltoniano, então em (G') podemos percorrer (v_1) para (v_2) para cada vértice (v) no circuito Hamiltoniano, resultando em um circuito fechado em (G') com soma dos comprimentos dos arcos maior ou igual a (0).
  - Se (G') tem um circuito fechado com soma dos comprimentos dos arcos maior ou igual a (0), então em (G) podemos percorrer cada vértice (v) de (v_1) para (v_2) no circuito fechado em (G'), formando um circuito Hamiltoniano em (G).
- Como o problema do circuito Hamiltoniano é NPC, a redução acima mostra que o problema do circuito mais longo é NP-Difícil.

Portanto, como o problema do circuito mais longo está em NP e é NP-Difícil (reduzível ao problema do circuito Hamiltoniano), podemos concluir que ele é NP-Completo.

8

As classes de equivalência polinomial, como P, NP e suas subclasses, têm um significado muito prático na teoria da complexidade computacional e na prática da resolução de problemas computacionais. Aqui estão alguns significados importantes dessas classes na prática:

1. **P (Problemas Polinomiais):**
   - **Eficiência em Tempo:** Problemas em P são aqueles para os quais existem algoritmos eficientes que podem ser executados em tempo polinomial em relação ao tamanho da entrada. Isso significa que, na prática, esses problemas podem ser resolvidos de forma rápida mesmo para entradas de tamanho considerável.
   - **Aplicações Práticas:** Muitos problemas do mundo real que podem ser formulados como problemas de decisão estão em P. Por exemplo, verificar se um número é primo, ordenar uma lista de números e encontrar o caminho mais curto em um grafo (algoritmo de Dijkstra) são problemas em P.
   - **Complexidade Gerenciável:** O fato de um problema estar em P significa que, na prática, podemos lidar com instâncias razoavelmente grandes do problema sem problemas de escalabilidade, o que é crucial em muitos campos da computação aplicada.
2. **NP (Problemas Não-Determinísticos Polinomiais):**
   - **Verificação de Soluções:** Problemas em NP são aqueles para os quais, se uma solução proposta é dada, podemos verificar rapidamente se essa solução está correta ou não em tempo polinomial. Embora encontrar a solução possa ser difícil, verificar se uma solução proposta é correta é fácil.
   - **Desafios Práticos:** Muitos problemas de otimização e busca de soluções estão em NP. Por exemplo, o problema do caixeiro viajante (TSP), o problema da mochila (knapsack problem), e problemas relacionados a grafos e programação inteira são exemplos de problemas em NP.
   - **Criptografia e Segurança:** A classe NP também é relevante em criptografia. Problemas como o problema do logaritmo discreto e o problema do logaritmo discreto em curvas elípticas são NP-completos, o que significa que, se alguém encontrasse uma solução eficiente para esses problemas, isso teria implicações significativas na criptografia.
3. **Classes NP-Completa e NP-Hard:**

- **Complexidade Comparativa:** As classes NP-Completa e NP-Hard são importantes porque representam problemas que são tão difíceis quanto os problemas mais difíceis em NP. Resolver um único problema NP-Completo em tempo polinomial provaria que P = NP, o que teria enormes implicações em muitas áreas da computação, incluindo criptografia, otimização e inteligência artificial.
- **Limitações Práticas:** A existência de problemas NP-Completo e NP-Hard também coloca limitações práticas sobre o que podemos resolver eficientemente em tempo polinomial. Se esses problemas não podem ser resolvidos de maneira eficiente, isso nos lembra que há limites para a capacidade computacional, e que alguns problemas podem ser intrinsecamente difíceis de resolver.

Em resumo, as classes de equivalência polinomial nos ajudam a entender a complexidade dos problemas computacionais e a avaliar quais problemas podem ser resolvidos de maneira eficiente na prática. Elas também desempenham um papel fundamental na pesquisa teórica em ciência da computação e em áreas aplicadas, como otimização, criptografia e inteligência artificial.

9

Para provar que existe uma transformação polinomial entre uma linguagem (L_1) em P e uma linguagem (L_2) em NP, devemos demonstrar que podemos transformar uma instância de (L_1) em uma instância de (L_2) em tempo polinomial.

Seja (L_1) uma linguagem em P, o que significa que existe um algoritmo polinomial (A) que decide (L_1). E seja (L_2) uma linguagem em NP, o que significa que existe um verificador polinomial (V) que verifica se uma solução proposta para (L_2) é correta ou não em tempo polinomial.

Vamos definir a transformação polinomial (f) que mapeia instâncias de (L_1) para instâncias de (L_2). Dada uma instância (x) de (L_1), a transformação (f(x)) deve produzir uma instância (y) de (L_2) de forma que:

1. Se (x) pertence a (L_1), então (f(x)) pertence a (L_2).
2. Se (x) não pertence a (L_1), então (f(x)) não pertence a (L_2).

A transformação (f) deve ser computável em tempo polinomial, ou seja, deve haver um algoritmo polinomial que possa calcular (f(x)) para qualquer (x).

A seguir, apresento um exemplo de como essa transformação pode ser realizada:

Seja (L_1) a linguagem de números pares e (L_2) a linguagem de números primos. Podemos definir a transformação (f) da seguinte maneira:

- Para uma instância (x) de (L_1), (f(x)) é o número (2x).
- Para uma instância (y) de (L_2), (f^{-1}(y)) é (\frac{y}{2}).

Essa transformação é polinomial, pois multiplicar por 2 ou dividir por 2 é uma operação que pode ser feita em tempo constante.

Assim, mostramos que é possível realizar uma transformação polinomial entre uma linguagem em P e uma linguagem em NP. Isso demonstra que (L_1) é redutível em tempo polinomial a (L_2), ou seja, (L_1) é equivalente a (L_2) em termos de complexidade computacional.

10

Aqui estão as classes de problemas ordenadas/relacionadas usando o símbolo "⊆":

1. P ⊆ NP
    - A classe P está contida na classe NP porque qualquer problema em P pode ser resolvido em tempo polinomial por uma máquina de Turing não determinística.
2. NP ⊆ PSPACE
    - A classe NP está contida na classe PSPACE porque uma máquina de Turing não determinística pode ser simulada por uma máquina de Turing determinística que utiliza espaço polinomial.
3. P ⊆ PSPACE
    - A classe P está contida na classe PSPACE porque qualquer problema resolvido em tempo polinomial também pode ser resolvido utilizando espaço polinomial.
4. PSPACE ⊆ EXPTIME
    - A classe PSPACE está contida na classe EXPTIME porque qualquer problema resolvido utilizando espaço polinomial pode ser resolvido em tempo exponencial.
5. NP ⊆ EXPTIME
    - A classe NP está contida na classe EXPTIME porque qualquer problema resolvido em tempo polinomial por uma máquina de Turing não determinística pode ser resolvido em tempo exponencial.
6. P ⊆ EXPSPACE
    - A classe P está contida na classe EXPSPACE porque qualquer problema resolvido em tempo polinomial também pode ser resolvido utilizando espaço exponencial.
7. PSPACE ⊆ EXPSPACE
    - A classe PSPACE está contida na classe EXPSPACE porque qualquer problema resolvido utilizando espaço polinomial também pode ser resolvido utilizando espaço exponencial.

Assim, a ordem das classes de problemas em termos de inclusão é: P ⊆ NP ⊆ PSPACE ⊆ EXPTIME ⊆ EXPSPACE. Note que EXPSPACE é uma superclasse tanto de PSPACE quanto de EXPTIME.

11

A formalização genérica da noção de problema na teoria da computação é frequentemente feita através do conceito de linguagem. Uma linguagem em teoria da computação é um conjunto de cadeias de caracteres (ou palavras) sobre um alfabeto finito. Cada cadeia de caracteres representa uma instância do problema.

A razão pela qual as linguagens são usadas para formalizar problemas na teoria da computação é que muitos problemas computacionais podem ser formulados como perguntas sobre conjuntos de cadeias de caracteres. Por exemplo:

1. **Problema da Pertinência:** Dado um conjunto de cadeias de caracteres (L) e uma cadeia de caracteres (w), o problema é determinar se (w) pertence a (L).

2. **Problema da Decisão:** Dado um conjunto de cadeias de caracteres (L) e uma cadeia de caracteres (w), o problema é determinar se (w) satisfaz uma certa propriedade ou critério especificado por (L).

3. **Problema de Otimização:** Dado um conjunto de cadeias de caracteres (L) e um critério de otimização (C), o problema é encontrar uma cadeia de caracteres (w) em (L) que otimize (C(w)).

Ao formalizar um problema como uma linguagem, podemos usar ferramentas matemáticas como teoria dos conjuntos, lógica e álgebra para analisar e classificar a complexidade do problema. Por exemplo, podemos aplicar autômatos, máquinas de Turing e modelos de computação para estudar a decidibilidade, a complexidade em tempo e espaço, e a relação entre diferentes problemas.

Além disso, a formalização por meio de linguagens permite uma abordagem abstrata e geral para representar uma ampla variedade de problemas, independentemente de sua natureza específica (decisão, otimização, etc.), facilitando a compreensão e a comparação entre diferentes problemas e algoritmos.

12

Um conjunto infinito enumerável é um conjunto que possui uma correspondência um a um com os números naturais, ou seja, um conjunto cujos elementos podem ser colocados em uma sequência infinita onde cada elemento tem uma posição única na sequência.

Formalmente, um conjunto (A) é infinito enumerável se existir uma função bijetora (f: \mathbb{N} \to A) que mapeia cada número natural para um elemento único de (A), e vice-versa, de modo que cada elemento de (A) tenha uma posição única na sequência infinita.

Em termos mais simples, um conjunto infinito enumerável é aquele para o qual podemos listar todos os seus elementos em uma sequência infinita, de modo que não haja elementos faltando ou repetidos na sequência, e cada elemento tenha uma posição específica.

Exemplos de conjuntos infinitos enumeráveis incluem o conjunto dos números naturais (\mathbb{N}), o conjunto dos números inteiros (\mathbb{Z}), o conjunto dos números racionais (\mathbb{Q}), o conjunto dos números racionais positivos (\mathbb{Q}^+), e o conjunto dos números racionais não negativos (\mathbb{Q}^{\geq 0}). Todos esses conjuntos têm uma correspondência um a um com os números naturais e, portanto, são infinitos enumeráveis.

13

A diferença fundamental entre as classes das linguagens recursivas (R) e das linguagens enumeráveis recursivamente (RE) está na capacidade de decisão das máquinas de Turing correspondentes a cada classe.

1. **Linguagens Recursivas (R):**

    - Uma linguagem está na classe R se existe uma máquina de Turing que, ao receber uma entrada, para em um estado de aceitação se a entrada pertence à linguagem e para em um estado de rejeição se a entrada não pertence à linguagem.
    - O funcionamento de uma máquina de Turing correspondente à classe R é determinístico, ou seja, para cada configuração da máquina, há uma transição única determinada pelas regras da máquina.

2. **Linguagens Enumeráveis Recursivamente (RE):**
    - Uma linguagem está na classe RE se existe uma máquina de Turing que, ao receber uma entrada, para em um estado de aceitação se a entrada pertence à linguagem e pode continuar indefinidamente (não para ou entra em loop) se a entrada não pertence à linguagem.
    - O funcionamento de uma máquina de Turing correspondente à classe RE é não determinístico, ou seja, para cada configuração da máquina, pode haver múltiplas transições possíveis e a máquina pode escolher qualquer uma delas.

Para ilustrar essas diferenças, vamos considerar exemplos de máquinas de Turing correspondentes às classes R e RE:

1. **Máquina de Turing para uma Linguagem Recursiva (R):**
    - Suponha que temos uma linguagem que aceita palavras que são palíndromos. Uma máquina de Turing para essa linguagem na classe R funcionaria da seguinte forma:
        - Ao receber uma entrada, a máquina de Turing verifica se a palavra é um palíndromo.
        - Se a palavra for um palíndromo, a máquina para em um estado de aceitação.
        - Se a palavra não for um palíndromo, a máquina para em um estado de rejeição.
2. **Máquina de Turing para uma Linguagem Enumerável Recursivamente (RE):**
    - Suponha que temos uma linguagem que aceita palavras que são múltiplos de 3. Uma máquina de Turing para essa linguagem na classe RE funcionaria da seguinte forma:
        - Ao receber uma entrada, a máquina de Turing verifica se a palavra é um número múltiplo de 3.
        - Se a palavra for um número múltiplo de 3, a máquina para em um estado de aceitação.
        - Se a palavra não for um número múltiplo de 3, a máquina continua operando indefinidamente (não para) porque não pode decidir se a palavra é aceita ou não.

Portanto, a diferença fundamental entre R e RE está na capacidade de decisão das máquinas de Turing correspondentes. As máquinas de Turing para linguagens recursivas podem decidir em um número finito de passos se uma palavra pertence à linguagem ou não, enquanto as máquinas de Turing para linguagens enumeráveis recursivamente podem continuar operando indefinidamente se não conseguirem decidir em um número finito de passos.

14

No contexto da tese de Turing, o termo "mais poderosa" refere-se a uma máquina hipotética que pode resolver problemas que estão além da capacidade da máquina de Turing clássica. Essa máquina "mais poderosa" seria capaz de realizar cálculos ou processamentos que não podem ser realizados por uma máquina de Turing, mesmo em teoria.

A tese de Turing não pode ser formalmente provada porque é uma proposição teórica sobre o poder computacional máximo possível. A tese de Turing afirma que qualquer problema que possa ser resolvido por um algoritmo efetivo (ou seja, um problema decidível) pode ser resolvido por uma máquina de Turing.

A ideia por trás da refutação da tese de Turing envolveria o desenvolvimento de uma máquina hipotética que pudesse resolver problemas que estão além da capacidade da máquina de Turing clássica. Essa

máquina "mais poderosa" teria capacidades computacionais superiores às da máquina de Turing e poderia resolver problemas considerados atualmente como não computáveis.