P versus NP

O problema "**P versus NP**" é o principal problema aberto da Ciência da Computação. Possui também enorme relevância em campos que vão desde a Engenharia até a criptografia aplicada aos serviços militares e às transações comerciais e financeiras via Internet.

Problemas do Prémio Millennium

- P versus NP
- Conjectura de Hodge
- Conjectura de Poincaré (solução)
- Hipótese de Riemann
- Existência de Yang-Mills e intervalo de massa
- Existência e suavidade de Navier-Stokes
- Conjectura de Birch e Swinnerton-Dyer

Definição do problema

De modo simplificado, o problema pergunta se existem problemas matemáticos cuja resposta pode ser verificada em tempo polinomial, que não possam ser resolvidos (diretamente, sem se ter um candidato à solução) em tempo polinomial. Ilustrando: se alguém lhe disser que o número 13.717.421 pode ser escrito como o produto de dois outros inteiros, você provavelmente demorará para provar isso; contudo, se lhe assoprarem que ele é o produto de 3.607 por 3.803, você seria capaz de muito rapidamente verificar tal fato.

O problema "P versus NP" parte da constatação que são muito frequentes as situações em que parece ser muito mais rápido verificar solução do que achar um processo de resolução, e então pergunta: isso sempre ocorre, ou simplesmente ainda não descobrimos um modo de resolvê-los rapidamente?

Para uma discussão mais desenvolvida, mas ainda evitando tecnicalidades, sobre o Problema "P versus NP", veja outro problema clássico: o problema do caixeiro viajante.

Sua importância resume-se em duas capacidades:

- de modo simples e concreto, exemplifica a enorme velocidade de crescimento da fatorial;
- prova-se que muitos problemas combinatórios envolvem tantas alternativas de solução quanto este problema, de modo que ele é uma espécie de métrica com a qual medimos a complexidade computacional dos problemas combinatórios ocorrendo em engenharia e no trabalho científico.

Contexto

A relação entre as classes de complexidade P e NP é estudada na teoria da complexidade computacional, parte da teoria da computação que lida com os recursos necessários computacionais para se resolver um determinado problema. Os recursos mais comuns são o tempo (quantos passos é preciso para se resolver um problema) e espaço (a quantidade de memória necessária para se resolver um problema). Em tal análise, um modelo de computador cujo tempo deve ser analisado é necessário. Geralmente, esses

modelos assumem que o computador é determinístico (dado o estado atual do computador e todas as entradas, há apenas uma ação possível que o computador pode realizar) e sequencial (executa ações uma após a outra).

Nesta teoria, a classe P consiste em todos os problemas de decisão (definidos abaixo) que podem ser resolvidos por uma máquina determinística sequencial em uma quantidade de tempo que é polinomial para o tamanho da entrada; a classe NP consiste em todos os problemas de decisão cujas soluções positivas podem ser verificadas em tempo polinomial dada a informação correta, ou de forma análoga, cujas soluções podem ser encontradas em tempo polinomial em uma máquina não determinística. Claramente, P ⊆ NP. Sem dúvida a maior questão sem respostas na ciência da computação teórica se diz respeito à relação entre essas duas classes: **P é igual a NP?**

NP-completo

Para atacar a questão P = NP, o conceito de NP-completude é muito útil. Problemas NP-completos são um conjunto de problemas que podem ser reduzidos em tempo polinomial a partir de qualquer outro problema NP, mas cuja solução ainda pode ser verificada em tempo polinomial. Informalmente, um problema NP-completo é pelo menos tão difícil quanto qualquer outro problema NP. Problemas NP-difíceis são aqueles ao menos tão difíceis quanto problemas NP-completos, ou seja, todos os problemas NP podem ser reduzidos (em tempo polinomial) para NP-difíceis. Problemas NP-difíceis não precisam estar em NP, ou seja, eles não precisam ter soluções verificáveis em tempo polinomial.

Por exemplo, a versão do problema de decisão do caixeiro viajante é NP-completo, portanto, qualquer instância de qualquer problema em NP pode ser transformada mecanicamente em uma instância do problema do caixeiro viajante, em tempo polinomial. O problema do caixeiro viajante é um dos muitos problemas NP-completos. Se qualquer problema NP-completo está em P, então sabe-se que P=NP. Infelizmente, muitos problemas importantes foram mostrados serem NP-completos, e até 2018 nenhum algoritmo rápido para qualquer um deles foi descoberto.

Só com base na definição não fica claro que problemas NP-completos existem. Um trivial e artificial problema NP-completo pode ser formulado como: dada uma descrição de uma máquina de Turing M garantida para parar em tempo polinomial, existe um polinômio de entrada de tamanho M que será aceito? Este problema está em NP porque (dando uma entrada) é simples de se verificar se M aceita ou não a entrada através de uma simulação de M; é NP-completo porque o verificador para qualquer instância particular de um problema em NP pode ser codificado como uma máquina M em tempo polinomial que toma a solução para ser verificada como entrada. Então, a questão é saber se existe ou não uma instância determinada por uma entrada válida.

O primeiro problema provado ser NP-completo foi o Problema de Satisfatibilidade Booleana (SAT). Ele passou a ser conhecido como o teorema de Cook-Levin; ele prova que satisfatibilidade, que é NP-completo, contém detalhes técnicos sobre máquinas de Turing e mostra como esses detalhes se relacionam com a definição de NP. No entanto, após este problema ser provado ser NP-completo, pela prova da redução previa-se de maneira simples que vários outros problemas estariam nessa classe. Assim, uma vasta classe de problemas aparentemente não relacionados seriam todos redutíveis uns aos outro, e seriam de alguma forma “o mesmo problema”.

Problemas mais difíceis

Embora não se saiba se P ≠ NP, problemas fora de P são conhecidos. Uma série de problemas sucintos (problemas que não operam na entrada normal, mas na descrição computacional da entrada) são conhecidos por serem EXPTIME-completo. Por causa disso, pode ser mostrado que problemas P EXPTIME estão fora de P, e assim exigem um tempo maior do que um tempo polinomial. De fato, pelo teorema da Hierarquia de Tempo, eles não podem ser resolvidos em tempo significativamente menor do que exponencial. Exemplos incluem a busca de uma estratégia perfeita para o xadrez (um N X N) e alguns outros jogos de tabuleiro.

O problema de decidir a verdade de uma declaração na aritmética de Presburger requer ainda mais tempo. Fischer e Rabin, em 1874 provaram que todo algoritmo que decide a verdade de declarações na aritmética de Presburger tem um tempo de execução de, pelo menos, 2^2cn menor que uma constante c, sendo n o comprimento da declaração. Assim, o problema é conhecido por precisar de mais tempo do que uma execução exponencial. Ainda mais difícil são os problemas indecidíveis, tais como o Problema da Parada. Eles não podem ser completamente resolvidos por nenhum algoritmo, no sentido de que para qualquer algoritmo, há pelo menos uma entrada para a qual o algoritmo não irá produzir a resposta certa; ele tenderá a produzir uma resposta errada, terminar sem dar nenhuma resposta conclusiva ou caso isso não ocorra, produzir qualquer resposta.

Problemas em NP não conhecidos em P ou NP-completo

Latner mostrou que se P ≠ NP então existem problemas em NP que não estão nem em P nem em NP-completo. Esses problemas são chamados Problemas intermediários da NP. O problema do isomorfismo gráfico, o problema do Logaritmo Discreto e o problema da Fatoração de Números Inteiros são exemplos de problemas que acredita-se que sejam NP-intermediários. Eles são alguns dos muito poucos problemas de NP que não se sabe se são em P ou se são NP-completo.

O problema do isomorfismo gráfico é o problema computacional que determina se dois gráficos finitos são isomórficos. Um importante problema não resolvido na teoria da complexidade é se o problema do isomorfismo gráfico está em P, NP-completo ou NP-intermediário. A resposta é desconhecida, mas acredita-se que, ao menos, o problema é não é NP-completo. Se o isomorfismo gráfico é NP-completo, a hierarquia do tempo polinomial entra em colapso com o segundo nível. Uma vez que acredita-se amplamente que a hierarquia polinomial não entra em colapso com qualquer nível finito, acredita-se que o isomorfismo gráfico não é NP-completo. O melhor algoritmo para este problema, graças a Laszlo Babai e Eugene Luks tem tempo de execução 20(√(n log n)) para gráfico com n vértices.

O problema da fatoração de inteiros é o problema computacional de determinar a fatoração de primos de um inteiro dado. Estabelecido como um problema de decisão, é o problema de decidir se a entrada tem um fator menor que k. Nenhum algoritmo de fatoração inteiro eficiente é conhecido, e este fato forma a base de vários sistemas criptográficos modernos, como o algoritmo RSA. O problema da fatoração do inteiro está na NP e na co-NP (ou até na UP e na co-UP). Se o problema é NP-completa, a hierarquia de tempo polinomial vai entrar em colapso com seu primeiro nível. (isto é, NP = co-NP). O melhor algoritmo conhecido para a fatoração de inteiro é a General Number Field Sieve (GNFS), que leva o

tempo esperado $\exp\left(\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)(\ln n)^{\frac{1}{3}}(\ln\ln n)^{\frac{2}{3}}\right)=L_n\left[\frac{1}{3},\sqrt[3]{\frac{64}{9}}\right]$ para o fator de um n-bit inteiro. No entanto, o melhor algoritmo quântico conhecido por este problema, o algoritmo de Shor, é executado em tempo polinomial. Infelizmente, este fato não diz muito sobre onde o problema está em relação a classes de complexidade não-quânticas.

P significa “fácil”?

Toda a discussão acima tem dito que P significa "fácil" e "não em P" significa "difícil", uma hipótese conhecida como tese de Cobham. É uma hipótese comum e razoavelmente exata em teoria da complexidade, porém tem algumas ressalvas.

Primeiro, nem sempre é verdade na prática. Um algoritmo polinomial teórico pode ter muitos fatores constantes ou expoentes, tornando-o, assim, impraticável. Por outro lado, até mesmo se um problema se mostrar NP-completo, e mesmo se P ≠ NP, ainda pode haver abordagens eficazes para combater o problema na prática. Existem algoritmos para muitos problemas NP-completos, como o Problema da Mochila, o Problema do Caixeiro Viajante e o Problema de Satisfatibilidade Booleana, que podem resolver com otimização muitas instâncias do mundo real, em tempo razoável. A complexidade empírica da média de casos (tempo versus tamanho do problema) de tais algoritmos pode ser surpreendentemente baixa. Um exemplo famoso é o algoritmo simplex na programação linear, que funciona surpreendentemente bem na prática, apesar de ter pior caso de complexidade de tempo exponencial que anda junto com os mais conhecidos algoritmos de tempo polinomial.

Segundo, existem tipos de cálculos que não são compatíveis ao modelo de máquina de Turing em que P e NP são definidos, assim como computação quântica e algoritmos randomizados.

Razões para acreditar que P ≠ NP

De acordo com uma pesquisa,[1][2] muitos cientistas da computação acreditam que P ≠ NP. A principal razão para essa crença é que, após décadas estudando estes problemas, ninguém foi capaz de encontrar um algoritmo de tempo polinomial para qualquer um dos mais de 3000 problemas NP-completos conhecidos. Esses algoritmos foram procurados muito antes do conceito de NP-completude ter sido definido (21 problemas NP-completos de Karp, entre os primeiros encontrados, eram todos problemas bem conhecidos existentes no momento em que foram mostrados para ser NP-completo). Além disso, o resultado P = NP implicaria em muitos outros resultados surpreendentes que estão acredita-se serem falsos atualmente, como NP = co-NP e P = PH.

Também é intuitivamente argumentado que a existência de problemas que são difíceis de resolver, mas para os quais as soluções são fáceis de verificar, combinam com a experiência do mundo real.

Se P = NP, então o mundo seria um lugar profundamente diferente do que supomos que ele seja. Não haveria nenhum valor especial em "saltos criativos," nenhuma lacuna fundamental entre resolver um problema e reconhecer a solução, uma vez que ele é encontrado. Todos que pudessem apreciar uma sinfonia seriam Mozart; todos que pudessem seguir um argumento passo a passo seriam Gauss ... - Scott Aaronson, MIT

Por outro lado, alguns pesquisadores acreditam que há excesso de confiança em acreditar que P ≠ NP e que os investigadores devem explorar provas de que P = NP também. Por exemplo, essas declarações foram feitas em 2002:

O principal argumento em favor da P ≠ NP é a total carência de progresso fundamental na área de busca exaustiva. Isto é, na minha opinião, um argumento muito fraco. O espaço de algoritmos é muito grande e nós estamos apenas no início de sua exploração. [. . .] A resolução do Último Teorema de Fermat mostra também que as questões muito simples podem ser resolvidos apenas por teorias muito profundas. -Moshe Y. Vardi, Rice University

Estar ligado a uma especulação não é um bom guia para o planejamento de pesquisa. Um deve sempre tentar ambas as direções de todos os problemas. O preconceito tem levado matemáticos famosos a falhar ao resolver problemas conhecidos cuja solução era oposta às suas expectativas, apesar de terem sido desenvolvidos todos os métodos necessários. -Anil Nerode, Cornell University

Consequências da resolução do problema

Uma das razões de o problema atrair tanta atenção são as consequências da resposta. Qualquer direção de resolução iria avançar a teoria enormemente, e talvez tenha enormes consequências práticas também.

P = NP

A prova de que P = NP poderia ter consequências práticas magníficas, se a prova levasse a métodos eficientes para resolver alguns dos importantes problemas na NP. Também é possível que uma prova não levaria diretamente para métodos eficientes, talvez se a prova não for construtiva, ou se o tamanho do polinômio delimitador for grande demais para ser eficiente na prática. As consequências, tanto positivas como negativas, originam-se desde que vários problemas NP-completos são fundamentais em muitos campos. Criptografia, por exemplo, depende da dificuldade de certos problemas. Uma solução construtiva e eficiente para um problema NP-completo, tais como 3-SAT, iria quebrar a maioria dos sistemas de encriptação (???) existentes, incluindo criptografia de chave pública, uma base para muitas aplicações de segurança modernos, como transações econômicas seguras através da Internet, e codificações simétricas, como AES ou 3DES, usadas para a criptografia de dados de comunicações. Estes teriam de ser modificados ou substituídos por soluções de informação teoricamente seguras.

Por outro lado, há enormes consequências positivas que poderiam acompanhar a representação tratável de muitos problemas atualmente matematicamente intratáveis. Por exemplo, muitos problemas em pesquisas de operações são NP-completo, tais como alguns tipos de programação inteira, e o problema do CAIXEIRO VIAJANTE, para citar dois dos exemplos mais famosos. Soluções eficazes para esses problemas teriam enormes implicações para logísticas. Muitos outros problemas importantes, como alguns problemas na previsão da estrutura de proteínas, também são NP-completos; se estes problemas fossem eficiente resolvíveis, poderiam estimular avanços consideráveis na biologia.

Mas tais mudanças podem enfraquecer significativamente se comparadas à revolução que um método eficiente para a resolução de problemas NP-completos poderia causar na própria matemática. De acordo com Stephen Cook, ... seria transformar a matemática, deixando que um computador encontre uma prova

formal de qualquer teorema que tem uma prova de um tamanho razoável, uma vez que as provas formais podem ser facilmente reconhecidas em tempo polinomial. Problemas-exemplo podem muito bem incluir todos os problemas-prêmio CMI.

Pesquisadores matemáticos gastam suas carreiras tentando provar teoremas, e algumas provas têm levado décadas ou mesmo séculos para serem decifradas - por exemplo, o Último Teorema de Fermat levou mais de três séculos para ser provado. Um método que fosse garantido para encontrar provas de teoremas deveria existir de um tamanho "razoável" iria essencialmente acabar com essa luta.

P ≠ NP

A prova que demonstrasse que P ≠ NP não traria os mesmos benefícios práticos computacionais da prova de que P = NP, mas, no entanto, representaria um avanço muito significativo na teoria da complexidade computacional e forneceria orientações para pesquisas futuras. Permitiria mostrar de uma maneira formal de que muitos problemas comuns não podem ser resolvidos de forma eficiente, de modo que a atenção de pesquisadores poderia ser focada em soluções parciais ou soluções para outros problemas. Devido à crença generalizada em P ≠ NP, grande parte desse foco de pesquisa já foi realizada.

P ≠ NP também ainda deixa aberta a complexidade média-caso de problemas difíceis em NP. Por exemplo, é possível que SAT requeira tempo exponencial no pior caso, mas que quase todas as instâncias selecionadas aleatoriamente de que são resolução eficiente. Russell Impagliazzo descreveu cinco hipotéticos "mundos" que poderiam resultar das diversas soluções possíveis para a questão caso a complexidade da média. Estes vão desde "Algorithmica", onde P = NP e problemas como SAT pode ser resolvido de forma eficiente em todas as instâncias, para "Cryptomania", onde P ≠ NP e gerando casos de problemas de difícil fora P é fácil, com três possibilidades intermediárias refletindo diferentes distribuições possíveis de dificuldade mais casos de problemas NP-difíceis. O "mundo" onde P ≠ NP, mas todos os problemas em NP são tratáveis no caso médio é chamado de "Heuristica" no papel. A Princeton University oficina em 2009 estudou a situação dos cinco mundos.

Algoritmo de tempo polinomial

Nenhum algoritmo para qualquer problema NP-completo é conhecido para ser executado em tempo polinomial. No entanto, existem algoritmos para problemas NP-completo com a propriedade que se P = NP, então o algoritmo é executado dentro do tempo polinomial (embora com constantes enormes, tornando o algoritmo impraticável). O algoritmo a seguir, devido à Levin, é um exemplo. Corretamente aceita a linguagem NP-completo SUBSET-SUM, e executado em tempo polinomial se e somente se P = NP.

// Algoritmo que aceita a linguagem NP-completa SUBSET-SUM // // Esse é um algoritmo de tempo polinomial se e somente se P = NP. // // “Tempo-polinomial" significa retornar sim dentro do tempo // polinomial quando a resposta deveria ser “sim”, e executar // infinitamente quando for “não”. // // Entrada: S = a conjunto finite de inteiros // Saída: "sim" se qualquer se qualquer subconjunto de S acrescenta-se a 0 // Executa infinitamente quando a saída for não, por outro lado. // Nota: "program number P" é um programa obtido por // escrever o inteiro P em binário, então // considerar que a string de bits pode ser um // programa. Todo programa possível pode ser // gerado desta maneira, embora a maioria

não faz nada // por erro de sintaxe. // //Para N = 1...infinito // Para P = 1...N // Executar o program number P por N etapas com entrada S // Se a saída do programa for uma lista distinta de inteiros // E os inteiros for todos dentro de S // E a soma dos inteiros a 0 // Então // Saída "sim" e para

Se, e somente se, P = NP, então este é um algoritmo de tempo polinomial aceitar a linguagem NP-completa. “Aceitar” significa retornar respostas “sim” em tempo polinomial, mas é permitido executar pra sempre quando a resposta é “não”.

Este algoritmo é impraticável, mesmo se P = NP. Se o menor programa que pode resolver o SUBSET-SUM dentro do tempo polinomial é b bits de comprimento, o algoritmo acima irá tentar no mínimo 2b−1 outros programas primeiro.

Formulando o problema do caixeiro viajante

Suponha que um caixeiro viajante tenha de visitar *n* cidades diferentes, iniciando e encerrando sua viagem na primeira cidade. Suponha, também, que não importa a ordem com que as cidades são visitadas e que de cada uma delas pode-se ir diretamente a qualquer outra. O problema do caixeiro viajante consiste em descobrir a rota que torna mínima a viagem total.

Exemplificando o caso *n*= 4

Se tivermos quatro cidades A, B, C e D, uma rota que o caixeiro deve considerar poderia ser: saia de A e daí vá para B, dessa vá para C, e daí vá para D e então volte a A. Quais são as outras possibilidades? É muito fácil ver que existem seis rotas possíveis:

AB**C**DA

ABD**C**A

A**C**BDA

A**C**DBA (possibilidade de ser certa)

ADB**C**A

AD**C**BA

Complexidade computacional do problema

O problema do caixeiro é um clássico exemplo de problema de otimização combinatória. A primeira coisa que podemos pensar para resolver esse tipo de problema é reduzi-lo a um problema de enumeração: achamos todas as rotas possíveis e, usando um computador, calculamos o comprimento de cada uma delas e então vemos qual a menor. É claro que se acharmos todas as rotas estaremos contando-as, daí podermos dizer que estamos reduzindo o problema de otimização a um de enumeração.

Para acharmos o número *R(n)* de rotas para o caso de *n* cidades, basta fazer um raciocínio combinatório simples e clássico. Por exemplo, no caso de *n =4* cidades, a primeira e última posição são fixas, de modo que elas não afetam o cálculo; na segunda posição podemos colocar qualquer uma das 3 cidades restantes

B, C e D, e uma vez escolhida uma delas, podemos colocar qualquer uma das 2 restantes na terceira posição; na quarta posição não teríamos nenhuma escolha, pois sobrou apenas uma cidade; consequentemente, o número de rotas é $3 \times 2 \times 1 = 6$, resultado que tínhamos obtido antes contando diretamente a lista de rotas acima.

De modo semelhante, para o caso de n cidades, como a primeira é fixa, o leitor não terá nenhuma dificuldade em ver que o número total de escolhas que podemos fazer é $(n-1) \times (n-2) \times ... \times 2 \times 1$. De modo que, usando a notação de fatorial: $R(n) = (n - 1)!$.

Assim que nossa estratégia reducionista consiste em gerar cada uma dessas $R(n) = (n - 1)!$ rotas, calcular o comprimento total das viagens de cada rota e ver qual delas tem o menor comprimento total (esse método é chamado no jargão matemático de força bruta e ignorância[3]). Trabalho fácil para o computador, diria alguém. Bem, talvez não. Vejamos o porquê.

Suponhamos que temos um computador muito veloz, capaz de fazer 1 bilhão de adições por segundo. Isso parece uma velocidade imensa, capaz de tudo. Com efeito, no caso de 20 cidades, o computador precisa apenas de 19 adições para dizer qual o comprimento de uma rota e então será capaz de calcular $10^9 / 19 = 53$ milhões de rotas por segundo. Contudo, essa imensa velocidade é um nada frente à imensidão do número 19! de rotas que precisará examinar. Com efeito, o valor de 19! é 121.645.100.408.832.000 (ou , aproximadamente, $1{,}2 \times 10^{17}$ em notação científica). Consequentemente, ele precisará de $1{,}2 \times 10^{17} / (53 \times 10^6) = 2{,}3 \times 10^9$ segundos para completar sua tarefa, o que equivale a cerca de 73 anos. O problema é que a quantidade $(n - 1)!$ cresce com uma velocidade alarmante, sendo que muito rapidamente o computador torna-se incapaz de executar o que lhe pedimos. Constate isso mais claramente na tabela a seguir:

n	**rotas por segundo**	***(n - 1)!***	**cálculo total**
5	250 milhões	24	insignificante
10	110 milhões	362.880	0.003 seg
15	71 milhões	87 bilhões	20 minutos
20	53 milhões	1,2 × 10^17	73 anos
25	42 milhões	6,2 × 10^23	470 milhões de anos

Observe que o aumento no valor do n não provoca uma grande diminuição na velocidade com que o computador calcula o tempo de cada rota (ela diminui apenas de um sexto ao n aumentar de 5 para 25), mas provoca um imensamente grande aumento no tempo total de cálculo. Em outras palavras: a inviabilidade computacional é devida à presença da fatorial na medida do esforço computacional do método da redução. Com efeito, se essa complexidade fosse expressa em termos de um polinômio em n o nosso computador seria perfeitamente capaz de suportar o aumento do n. Confira isso na seguinte tabela que corresponde a um esforço computacional polinomial $R(n) = n^5$:

n	**rotas por segundo**	***n^5***	**cálculo total**
5	250 milhões	3.125	insignificante
10	110 milhões	100.000	insignificante

15	71 milhões	759.375	0,01 seg
20	53 milhões	3.200.000	0,06 seg
25	42 milhões	9.765.625	0,23 seg

Então o método reducionista não é prático (a não ser para o caso de muito poucas cidades), mas será que não se pode inventar algum método prático (por exemplo, envolvendo esforço polinomial na variável número de cidades) para resolver o problema do caixeiro viajante? Bem, apesar de inúmeros esforços, ainda não foi achado tal método e começa-se a achar que o mesmo não existe.

Conclusão

A existência ou não de um método polinomial para resolver o problema do caixeiro viajante é um dos grandes problemas em aberto da Matemática na medida em que Stephen Cook (1971) e Richard Karp (1972) mostraram que uma grande quantidade de problemas importantes (como é o caso de muitos tipos de problemas de otimização combinatória, o caso do problema da decifração de senhas criptografadas com processos modernos como o DES, etc.) podem ser reduzidos, em tempo polinomial, ao problema do caixeiro.

Consequentemente, se descobrirmos como resolver o problema do caixeiro em tempo polinomial, ficaremos sendo capazes de resolver também, em tempo polinomial, uma grande quantidade de outros problemas matemáticos importantes; por outro lado, se um dia alguém provar que é impossível resolver o problema do caixeiro em tempo polinomial no número de cidades, também se terá estabelecido que uma grande quantidade de problemas importantes não tem solução prática. Por isso, já foi até oferecido um Prêmio de um milhão de dólares a quem conseguir resolver este problema matemático.

Costuma-se resumir essas propriedades do problema do caixeiro viajante dizendo que ele pertence à categoria dos problemas NP-completo.

Código em Python para Simulação da Estratégia Reducionista

```
# Programa: Simulação da Estratégia Reducionista no Problema do Caixeiro Viajante

# Autor: Odin Oliveira

# Revisor:

# Data:06/02/2023

# Objetivo:

# Simulação de tempo que o computador que está executando o programa

# levaria para calcular algumas rotas para o caixeiro viajante sem

# utilizar Otimização Combinatória

# Calcula o tempo que o processador da máquina leva para fazer 10 milhões

# de adições, cada edição é o calculo de um caminho (ligação entre duas

# cidade)


import time
```

```python

tempo = time.time()
i = sum(range(1, 10000000))
tempo = time.time() - tempo
print("Tempo para processar 10 milhões de adições (segundos): ", tempo)
adicoes = 10000000 / tempo
print("Adições por segundo: ", int(adicoes))

cidades = [5, 10, 15, 20, 25]
for contador in cidades:
    print("\nCidades", contador)
    rotas_seg = (adicoes / (contador - 1))
    print("Rotas por segundo: ", int(rotas_seg))
    rotas = 1
    for i in range(2, contador):
        rotas *= i
    print("Rotas possíveis: ", rotas)
    tempo_calculo = rotas / rotas_seg
    if tempo_calculo < 1:
        print("Tempo para cálculo: ", int(tempo_calculo * 1000), " milisegundos")
    elif tempo_calculo < 60:
        print("Tempo para cálculo: ", int(tempo_calculo), " segundos")
    elif tempo_calculo < 3600:
        print("Tempo para cálculo: ", int(tempo_calculo / 60), " minutos")
    elif tempo_calculo < 86400:
        print("Tempo para cálculo: ", int(tempo_calculo / 3600), " horas")
    else:
        print("Tempo para cálculo: ", int(tempo_calculo / 86400), " dias")
```

Bibliografia

- Fraenkel, A. S.; Lichtenstein, D.. Computing a Perfect Strategy for n*n Chess Requires Time Exponential in N.. doi:10.1007/3-540-10843-2+23. (https://web.archive.org/web/20170425031512/http://www.pubzone.org/dblp/conf/icalp/FraenkelL81)
- Garey, Michael (1979). Computers and Intractability. San Francisco: W.H. Freeman. ISBN 0716710455.
- Goldreich, Oded (2010). P, Np, and Np-Completeness. Cambridge: Cambridge University Press. ISBN 9780521122542.
- Immerman, N. (1983). Languages which capture complexity classes. pp. 347. doi:10.1145/800061.808765. (http://dl.acm.org/citation.cfm?doid=800061.808765)
- Cormen, Thomas (2001). Introduction to Algorithms. Cambridge: en:Special:BookSources/0262032937 MIT Press. ISBN 0262032937.]
- Papadimitriou, Christos (1994). Computational Complexity. Boston: Addison-Wesley. ISBN 0201530821. (http://enwiki/Special:BookSources/0201530821)
- Fortnow, L. (2009). "The status of the P versus NP problem". Communications of the ACM 52 (9): 78. doi:10.1145/1562164.1562186. (http://dl.acm.org/citation.cfm?doid=1562164.1562186)

Ver também

- Estrutura de dados árvore ordenada - DAGs, árvores binárias e outras formas especiais de grafos.
- Teoria da computação
- Grafo

- Lista de tópicos da teoria dos grafos
- Construção de grafos
- Redes Complexas
- Problemas do Prémio Millenium

Referências

1. William I. Gasarch (junho de 2002). «The **P**=?**NP** poll.» (http://www.cs.umd.edu/~gasarch/papers/poll.pdf) (PDF). *SIGACT News*. **33** (2): 34–47. CiteSeerX 10.1.1.172.1005 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.172.1005). doi:10.1145/564585.564599 (https://dx.doi.org/10.1145%2F564585.564599)
2. Rosenberger, Jack (maio de 2012). «**P** vs. **NP** poll results» (http://mags.acm.org/communications/201205?pg=12). *Communications of the ACM*. **55** (5): 10
3. Rob Womersley, *Parabola Volume 37, Issue 2 (2001)*m *The Travelling Salesman Problem and Computational Complexity* [1] (http://www.parabola.unsw.edu.au/vol37_no2/node1.html) Arquivado em (https://web.archive.org/web/20110410005720/http://www.parabola.unsw.edu.au/vol37_no2/node1.html) 10 de abril de 2011, no Wayback Machine. [em linha]

Ligações externas

- Folha: Solução para maior problema da computação está provavelmente errada (http://www1.folha.uol.com.br/ciencia/782833-solucao-para-maior-problema-da-computacao-esta-provavelmente-errada.shtml)
- Lista de artigos com esforços para provar igualdade ou não de P e NP (http://www.win.tue.nl/~gwoegi/P-versus-NP.htm)

Obtida de "https://pt.wikipedia.org/w/index.php?title=P_versus_NP&oldid=69467921"