Estruturas de Dados II

Decidibilidade e Eficiência

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Decidibilidade

- A decidibilidade é um conceito fundamental na Computação que se relaciona com a capacidade dos computadores de resolver problemas.
- Nos leva a perguntas sobre os limites da computação e o que é possível ou impossível de ser resolvido por algoritmos.
- Será que todo problema computacional pode ser resolvido por um algoritmo?

Problema da Parada

- Suponha que te entreguem um programa, denominado de **halts**, que dizem efetuar o seguinte: recebe como entrada qualquer programa P e uma entrada E para este programa e, de alguma forma, sempre determina corretamente se o programa P irá parar, ou não, com a entrada E. A entrada E pode ser vazia.
- Recebemos halts de alguém confiável, e que sabemos ser um excelente programador.
- O programa halts possui o seguinte funcionamento: ele recebe um programa, e uma entrada, e retorna **true** se o programa irá parar. Retorna **false** se o programa não irá parar.
- Se halts realmente funcionar, será um programa inestimável. Extremamente útil.
- Afinal, ele descobre todos os tipos de bugs que fazem programas executarem eternamente. Podemos executar cada subrotina como um programa para identificar o problema em questão.
- Parece razoável? É possível implementar tal algoritmo? Vamos tentar algo.

Problema da Parada

- Em dúvida se ele realmente faz o que promete, você traz o programa para o professor e eu decido fazer o seguinte.
- Vou usá-lo para escrever outro programa que chamarei de **testandoHalts**.

```
testandoHalts()
    if(halts(testandoHalts()))
      while(1){}
```

- testandoHalts é um programa muito interessante, alguém pode me explicar o seu funcionamento?
- Lembrem que o halts recebe um programa, e uma entrada, e retorna **true** se o programa irá parar ou retorna **false** se o programa não irá parar.

Problema da Parada

```
testandoHalts()
    if(halts(testandoHalts()))
      while(1){}
```

- Se halts decidir que o programa testandoHalts, sem entrada, pára, então ele irá retornar **true** e ocorrerá um laço infinito.
- Se halts decidir que ele não pára, então o programa testandoHalts pára.

Problema da Parada

```
testandoHalts()
    if(halts(testandoHalts()))
      while(1){}
```

- Ou seja, halts possuindo testandoHalts como entrada pára se, e somente se, ele não parar! O que é uma contradição.
- A suposição inicial de que halts é um programa possível de ser implementado é falsa. É um programa que não existe e não pode existir.
- Portanto, tal problema, de decidir se outro programa pára ou não, é **indecidível**.

Problema da Parada

- Problemas indecidíveis (ou insolúveis) são problemas para os quais não existem algoritmos.
- Ou seja, um problema indecidível é um problema de decisão para o qual se provou ser impossível construir um algoritmo que sempre leve a uma resposta correta de sim ou não.
- O problema da parada, é um exemplo: provamos que não existe um algoritmo que determine corretamente se um programa arbitrário eventualmente pára quando é executado.
- Historicamente, muitos problemas já foram demonstrados como indecidíveis.

Problema da Parada

- Por exemplo, na matemática temos o décimo problema de Hilbert, proposto em 1900.
- Trata-se do desafio de fornecer um algoritmo geral que, para qualquer e, possa decidir se a equação tem uma solução com todas as incógnitas tomando valores inteiros.
- Uma equação Diofantina é uma equação polinomial com coeficientes inteiros e um número finito de incógnitas.
- Passados 70 anos, chegou-se a uma conclusão negativa. Não foi fornecido um algoritmo, em vez disso, demonstrou-se que esse problema é indecidível.

Complexidade Computacional

- Aprendemos que existem problemas bem definidos que não podem ser resolvidos por algoritmos. São problemas **indecidíveis**.
- Portanto, podemos dividir os problemas entre os decidíveis e os indecidíveis.
- Entretanto, existem problemas que estão no conjunto de decidíveis que, apesar de serem solucionáveis por algoritmos, não podem ser resolvidos em nenhum sentido prático.
- Isto ocorre devido a excessivas exigências de tempo e/ou espaço.

Complexidade Computacional

- Suponha que precise escalonar a visita de um profissional a n escritórios regionais em São Paulo.
- Lhe pedem para criar um algoritmo que irá produzir o itinerário que **minimiza** a distância total percorrida.
- De início, já te enviam um mapa com 40 cidades do estado e as distâncias em quilômetros. Será a primeira instância que resolverá com seu algoritmo, assim que ele estiver pronto.

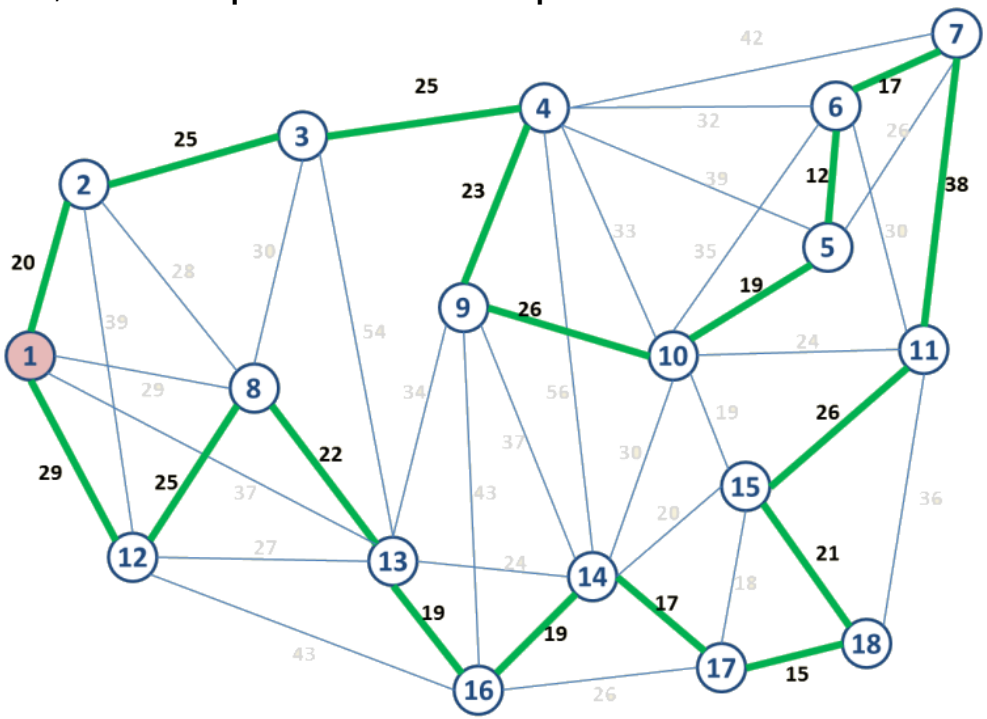

Figura: Exemplo de uma solução para uma instância com 18 cidades.

Complexidade Computacional

- Com certeza podemos usar um computador, um algoritmo, para resolver esse problema. Do ponto de vista teórico, é claramente um problema solúvel!
- E temos n = 40 cidades para visitar, o que é um número finito.
- Ou seja, parece ser um problema decidível.

Complexidade Computacional

- Podemos facilmente resolver esse problema com o seguinte algoritmo: examinamos todos os itinerários e selecionamos o mais curto.
- Chamamos isso de **abordagem de força-bruta**. Basta verificarmos todos os itinerários, é um algoritmo simples.
- Mas existe um desconforto em relação a esse algoritmo... parecem existir muitos itinerários para serem examinados.
- Apesar do número pequeno de cidades, afinal, são apenas 40 cidades.
- Se temos 40 cidades, teremos quantos itinerários distintos?

Complexidade Computacional

- Teremos 40! itinerários distintos.
- Mas tudo bem. Temos computadores muito poderosos.
- Suponha que nosso computador é muito rápido e examina 10^{15} itinerários por segundo (um quatrilhão!!).
- Logo resolveremos esse problema com essa máquina.
- Vamos efetuar os cálculos necessários para descobrir quanto tempo demorará.

Complexidade Computacional

- Primeiro, calculamos o valor de 40! (fatorial de 40):

$$40! = 40 \times 39 \times 38 \times \ldots \times 2 \times 1$$

- O valor aproximado é:

$$40! \approx 8,1591528 \times 10^{47}$$

- Então, dividimos este número por 10^{15} para encontrar quantos segundos são necessários para calcular todos os itinerários a essa taxa:

$$\begin{aligned} \text{Tempo (segundos)} &\approx \frac{8,1591528 \times 10^{47}}{10^{15}} \\ &\approx 8,1591528 \times 10^{32} \end{aligned}$$

Complexidade Computacional

- O tempo necessário para calcular todos os itinerários de 40 destinos, a uma taxa de um quatrilhão (10^{15}) de itinerários por segundo, é aproximadamente $8,1591528 \times 10^{32}$ segundos.
- Para ter uma ideia mais clara, podemos converter esse tempo em outras unidades de tempo:
  - Segundos em um minuto: 60
  - Segundos em uma hora: 3600
  - Segundos em um dia: 86400
  - Segundos em um ano (aproximadamente): $86400 \times 365,25 = 31.557.600$
- Vamos converter os segundos para anos para uma melhor compreensão. Ou seja: $\frac{8,1591528 \times 10^{32}}{31.557.600}$

Complexidade Computacional

- O resultado dessa divisão é: $2,58548 \times 10^{25}$.
- Portanto, o tempo necessário para calcular todos os itinerários de 40 destinos, a uma taxa de um quatrilhão (10^{15}) de itinerários por segundo, é de aproximadamente $2,58548 \times 10^{25}$ anos.
- Isso é um número extremamente grande, muito maior do que a idade atual do universo, que é de cerca de $13,8 \times 10^{9}$ anos.
- Portanto, mesmo a essa taxa muito rápida, levaria um tempo inimaginavelmente longo para calcular todos os itinerários possíveis para 40 destinos.

Complexidade Computacional

- Ou seja, um problema ser solúvel na teoria não implica que possa ser resolvido na prática.
- Temos um algoritmo de taxa de crescimento fatorial, onde com uma entrada suficientemente grande, torna-se irreal resolver o problema.
- Este é um problema clássico da computação para qual não conhecemos nenhum algoritmo **eficiente** para resolvê-lo.
- Mas o que é um algoritmo eficiente? Como definir eficiência?

Eficiência

- Entendemos que existem problemas que são indecidíveis e problemas que são decidíveis.
- Entre os problemas decidíveis, existem alguns que requerem tanta complexidade de tempo ou espaço que não são solucionáveis na prática.
- Pelo menos não de forma exata, ou seja, com algoritmos que garantam a melhor solução possível ou a solução correta.
- Mas entre os problemas decidíveis e tratáveis, podemos discutir o que constitui um algoritmo eficiente para um problema.

Eficiência

- Como podemos transformar a noção vaga de um algoritmo eficiente em algo mais concreto? Vamos tentar uma primeira definição.

Eficiência (1)

Um algoritmo é eficiente se, quando implementado, ele é executado rapidamente em instâncias reais de entrada.

- Parece uma boa definição, certo?

Eficiência

Eficiência (1)

Um algoritmo é eficiente se, quando implementado, ele é executado rapidamente em instâncias reais de entrada.

- O primeiro problema é que mesmo algoritmos ruins podem ser executados rapidamente quando aplicados a pequenos casos de teste em processadores extremamente rápidos.
- Outro problema é que bons algoritmos podem executar lentamente se mal codificados.

Eficiência

Eficiência (1)

Um algoritmo é eficiente se, quando implementado, ele é executado rapidamente em instâncias reais de entrada.

- Além disso, o que é uma "instância de entrada real"?
- Não conhecemos a gama completa de instâncias de entrada que serão encontradas na prática,
- E algumas instâncias de entrada podem ser muito mais difíceis do que outras.

Eficiência

Eficiência (1)

Um algoritmo é eficiente se, quando implementado, ele é executado rapidamente em instâncias reais de entrada.

- Finalmente, a definição proposta não considera quão bem, ou mal, um algoritmo pode escalar à medida que os tamanhos da instância do problema cresce.
- Podemos ter dois algoritmos diferentes que se comportam de maneira comparável para entradas de tamanho 100.
- Mas multiplique o tamanho da entrada por dez, e um ainda será executado rapidamente enquanto o outro consumirá uma enorme quantidade de tempo.
- Vimos isso com o cálculo do n-ésimo termo Fibonacci iterativamente e recursivamente.

Eficiência

- Precisamos de uma definição mais precisa, portanto, necessitaremos de uma visão mais matemática da situação.
- Para começar, vamos nos concentrar na análise do tempo de execução no pior caso.
- Entretanto, o foco de desempenho no pior caso inicialmente parece bastante rígido: e se um algoritmo se sair bem na maioria das instâncias e tiver apenas algumas entradas patológicas em que é muito lento?
- Isso certamente é um problema em alguns casos, mas, em geral, a análise do pior caso de um algoritmo tem se mostrado razoável em capturar sua eficiência na prática.

Eficiência

- Além disso, é difícil encontrar uma alternativa eficaz à análise do pior caso.
- A análise do caso médio pode ser difícil; como expressar a gama completa de instâncias de entrada que surgem na prática?
- Portanto a análise de casos médios arrisca nos dizer mais sobre os meios pelos quais as entradas aleatórias foram geradas do que sobre o próprio algoritmo.
- Mas qual é uma referência analítica razoável que possa nos dizer se um limite de tempo de execução é bom ou ruim?

Eficiência

- Vamos voltar ao problema do viajante. Podemos considerar uma comparação com a busca por força bruta sobre o espaço de busca de soluções possíveis.
- Essa abordagem é quase sempre muito lenta para ser útil, portanto, podemos utilizá-la para construir uma segunda definição de eficiência:

Eficiência (2)

Um algoritmo é eficiente se alcançar uma performance no pior caso qualitativamente melhor do que a busca por força bruta.

- Isso já é mais interessante, já que algoritmos que melhoram substancialmente em relação à busca por força bruta quase sempre contêm uma ideia valiosa que os faz funcionar.
- Eles nos dizem algo sobre a estrutura intrínseca do próprio problema.

Eficiência

Eficiência (2)

Um algoritmo é eficiente se alcançar uma performance no pior caso qualitativamente melhor do que a busca por força bruta.

- Mas é uma definição um tanto quanto vaga. Falta precisão.
- O que queremos dizer com "performance qualitativamente melhor"?
- **Entretanto, isso sugere um caminho**: podemos considerar o tempo de execução real dos algoritmos e tentar quantificar o que é um tempo de execução razoável.
- Mas precisaremos abrir um parêntese e revisar o conceito de polinômio antes de avançarmos.

Revisão – Polinômios

- Um polinômio é uma expressão matemática que consiste em uma soma de termos, cada um dos quais é o produto de uma constante (chamada de coeficiente) e uma variável elevada a uma potência não negativa. A forma geral de um polinômio em uma variável x é a seguinte:

$$P(x) = a_n \cdot x^n + a_{(n-1)} \cdot x^{(n-1)} + \ldots + a_2 \cdot x^2 + a_1 \cdot x^1 + a_0 \cdot x^0$$

- Neste polinômio, P(x) é a função polinomial, $a_n, a_{(n-1)}, \ldots, a_2, a_1, a_0$ são os coeficientes, x é a variável e n é um número inteiro não negativo.
- Os coeficientes podem ser números reais ou complexos, e o grau do polinômio é o valor mais alto entre as potências das variáveis com coeficientes não nulos.

Eficiência

- Portanto, a discussão deve ser em torno da **taxa de crescimento** de tempo dado um algoritmo.
- Na aula de recursão, observamos um algoritmo linear (iterativo) e um algoritmo exponencial (recursivo) para o cálculo do n-ésimo termo da sequência de Fibonacci.

Eficiência

- Na aula de fila de prioridade, observamos um algoritmo linear (implementação ingênua) e um algoritmo de complexidade logarítimica.
- Pudemos observar em um gráfico como o crescimento logarítimico é muito menor que o crescimento linear.

Eficiência

- Vimos que o algoritmo de força-bruta para encontrar todos os itinerários do viajante possuia um crescimento **fatorial**.
- E descobrimos o quanto grande é isso, comparando o tempo de execução do nosso algoritmo à idade do universo, com uma entrada de apenas 40 cidades.

Eficiência

- Quando as pessoas começaram a analisar algoritmos matematicamente, um consenso começou a surgir sobre como quantificar a noção de um tempo de execução "razoável".
- Observa-se que os espaços de busca para problemas **combinatórios** tendem a crescer exponencialmente com o tamanho da entrada.
- Portanto, queremos um algoritmo com uma propriedade melhor de escalonamento: quando o tamanho da entrada aumenta por um fator constante – ex: um fator de 2 –, o algoritmo deve apenas desacelerar por algum fator constante C.

Eficiência

- Podemos formular esse comportamento de escalonamento da seguinte forma: existem constantes absolutas c $>$ 0 e d $>$ 0 de modo que, **para toda instância de entrada de tamanho** n, seu tempo de execução seja limitado por $c \cdot n^d$ passos computacionais primitivos.
- Se este limite de tempo de execução for mantido, então dizemos que o algoritmo tem um **tempo de execução polinomial**, ou que é um algoritmo de tempo polinomial.
- Observe que qualquer limite de tempo polinomial tem a propriedade de escalonamento que estamos buscando.

Eficiência

- Se o tamanho da entrada aumenta de n para $2n$, o limite de tempo de execução aumenta de $c \cdot n^d$ para $c(2n)^d = c \cdot 2^d n^d$, o que representa uma desaceleração por um fator de 2^d.
- Uma vez que d é uma constante, também o é 2^d. Como seria de se esperar, polinômios de menor grau exibem um comportamento de escalonamento melhor do que polinômios de grau mais alto.
- A partir dessa noção e da intuição expressa anteriormente, podemos propor uma terceira definição de eficiência:

Eficiência (3)

Um algoritmo é eficiente se tiver um tempo de execução polinomial.

Eficiência

Eficiência (3)

Um algoritmo é eficiente se tiver um tempo de execução polinomial.

- Existem exceções?
- Existem algoritmos que não possuem tempo de execução polinomial e que consideraríamos "eficientes", fora dessa definição?
- Vocês sabem a resposta para essa pergunta.

Um Parêntese

- Mas em que medida algoritmos de complexidade polinomial captura a noção de "problemas solúveis na prática"?
- Se tivermos um algoritmo com exigências de tempo de n^{100}, ou $10^{100} \cdot n^2$, será que ele é viável na prática?
- A resposta a esses casos é que tais situações raramente ocorrem na prática.
- Algoritmos de complexidade polinomial geralmente possuem expoentes pequenos e coeficentes razoáveis.
- E algoritmos não-polinomiais geralmente possuem complexidade exponencial ou pior.

Um Parêntese

- Mas e um algoritmo com exigências de tempo de $\log_2 \log_2 n$?
- Este pode ser considerado solúvel na prática e seu crescimento não é limitado por um polinômio.
- Felizmente, nossa definição não exclui explicitamente a possibilidade de que existam algoritmos eficientes com tempos de execução não polinomiais.
- Ela não diz que um algoritmo é eficiente **somente** se tiver um tempo de execução polinomial.
- E de fato, algoritmos de tempo sublinear, como os que possuem complexidade logarítimica, podem ser considerados eficientes.

Eficiência

- Ou seja, esta é uma boa definição e é amplamente utilizada na Computação.
- Existem outros conceitos que poderiam aprender como: Classes P e NP, P versus NP, NP-Completude, Reduções, etc.
- Para finalizar, vamos olhar uma tabela comparando diferentes tempos de execução para que tenham uma noção prática de diferentes complexidades computacionais.

Eficiência

- Tempos de execução de diferentes algoritmos em entradas de tamanho crescente, para uma CPU executando um milhão de instruções de alto nível por segundo.
- Os casos em que o tempo de execução excede 10^{25} anos são registrados como "very long".

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long