Projeto e Análise de Algoritmos II (MC558)
Reduções entre problemas

Prof. Dr. Ruben Interian

Resumo

Resumo

Objetivos

- Entender **o que é um problema**.
- Entender **como comparar dois problemas**.
- Entender **como resolver um problema usando outro**.

Resumo

O que é um problema

O que é um problema?

O que é um problema

O que é um problema?

Problema computacional

Um problema computacional é uma relação $P \subseteq \mathcal{E} \times \mathcal{S}$, onde

- $\mathcal{E}$ é o conjunto de entradas, e
- $\mathcal{S}$ é o conjunto de saídas.

O que é um algoritmo para um problema

Quando um algoritmo resolve um problema P?

O que é um algoritmo para um problema

Quando um algoritmo resolve um problema P?

Algoritmo

Um algoritmo A resolve um problema P se para toda entrada $E \in \mathcal{E}$, ele devolve uma saída $S \in \mathcal{S}$ tal que $(E, S) \in P$.

O que é um algoritmo para um problema

Quando um algoritmo resolve um problema P?

Algoritmo

Um algoritmo A resolve um problema P se para toda entrada $E \in \mathcal{E}$, ele devolve uma saída $S \in \mathcal{S}$ tal que $(E, S) \in P$.

- Com frequência a entrada é denotada por I ("Instância de P", ou input).
- $A(I)$ é a saída do algoritmo.
- O **tamanho** da entrada é n. Normalmente n é o números de bits de E.

Cota superior de um problema

Cota superior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $g(n)$ é uma cota superior de P, se existe **algum algoritmo** A que resolve P com complexidade $O(g(n))$.

Cota superior de um problema

Cota superior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $g(n)$ é uma cota superior de P, se existe **algum algoritmo** A que resolve P com complexidade $O(g(n))$.

Por exemplo, seja P o problema de ordenação.

- InsertionSort fornece uma cota superior quadrática (costuma-se dizer $O(n^2)$).

Cota superior de um problema

Cota superior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $g(n)$ é uma cota superior de P, se existe **algum algoritmo** A que resolve P com complexidade $O(g(n))$.

Por exemplo, seja P o problema de ordenação.

- InsertionSort fornece uma cota superior quadrática (costuma-se dizer $O(n^2)$).
- MergeSort fornece uma cota superior de $O(n \log n)$ (é uma cota "melhor").

Cota superior de um problema

Cota superior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $g(n)$ é uma cota superior de P, se existe **algum algoritmo** A que resolve P com complexidade $O(g(n))$.

Por exemplo, seja P o problema de ordenação.

- InsertionSort fornece uma cota superior quadrática (costuma-se dizer $O(n^2)$).
- MergeSort fornece uma cota superior de $O(n \log n)$ (é uma cota "melhor").

Queremos encontrar a melhor (menor) cota superior de um problema.

Cota inferior de um problema

Cota inferior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $f(n)$ é uma cota inferior de P, se **todo algoritmo** A que resolve P possui complexidade $\Omega(f(n))$.

Cota inferior de um problema

Cota inferior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $f(n)$ é uma cota inferior de P, se **todo algoritmo** A que resolve P possui complexidade $\Omega(f(n))$.

Isso é equivalente a dizer que não existe **nenhum algoritmo** cuja complexidade é assintoticamente menor do que $f(n)$ que resolva P: $\nexists A$ com complexidade $o(f(n))$.

Cota inferior de um problema

Cota inferior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $f(n)$ é uma cota inferior de P, se **todo algoritmo** A que resolve P possui complexidade $\Omega(f(n))$.

Isso é equivalente a dizer que não existe **nenhum algoritmo** cuja complexidade é assintoticamente menor do que $f(n)$ que resolva P: $\nexists A$ com complexidade $o(f(n))$.

Por exemplo, seja P o problema de ordenação.

- A cota inferior é $\Omega(n \log n)$. Não é um resultado simples de provar.

Cota inferior de um problema

Cota inferior (de complexidade) de um problema

Seja P um problema, e n o tamanho de uma instância de P. A função $f(n)$ é uma cota inferior de P, se **todo algoritmo** A que resolve P possui complexidade $\Omega(f(n))$.

Isso é equivalente a dizer que não existe **nenhum algoritmo** cuja complexidade é assintoticamente menor do que $f(n)$ que resolva P: $\nexists A$ com complexidade $o(f(n))$.

Por exemplo, seja P o problema de ordenação.

- A cota inferior é $\Omega(n \log n)$. Não é um resultado simples de provar.

Queremos encontrar a melhor (maior) cota inferior de um problema.

Algoritmo ótimo para um problema

Algoritmo ótimo

Um algoritmo A é ótimo para um problema P quando sua complexidade coincide com uma cota inferior de P.

Algoritmo ótimo para um problema

Algoritmo ótimo

Um algoritmo A é ótimo para um problema P quando sua complexidade coincide com uma cota inferior de P.

Ou seja, A é ótimo para P se:

1. A resolve P em tempo $O(f(n))$ $(\Rightarrow f(n)$ é uma **cota superior** de $P)$;
2. $f(n)$ é uma **cota inferior** de P.

Algoritmo ótimo para um problema

Algoritmo ótimo

Um algoritmo A é ótimo para um problema P quando sua complexidade coincide com uma cota inferior de P.

Ou seja, A é ótimo para P se:

1. A resolve P em tempo $O(f(n))$ $(\Rightarrow f(n)$ é uma **cota superior** de $P)$;
2. $f(n)$ é uma **cota inferior** de P.

Exemplo:

- **HeapSort** e **MergeSort** são ótimos para ordenação, pois eles têm complexidade $O(n \log n)$, e ordenação tem cota inferior $\Omega(n \log n)$. Não há muitos problemas para os quais as cotas superior e inferior coincidem.

Análise de um problema

Quando queremos **analisar um problema**, queremos determinar a complexidade de um algoritmo ótimo para esse problema.

Análise de um problema

Análise do algoritmo $\neq$ Análise do problema

- Análise do pior caso: avaliamos o tempo de execução mais longo para qualquer instância de tamanho n.
- Avaliamos o tempo de execução de pior caso do melhor algoritmo que pode existir para o problema.

Análise de um problema

Análise do algoritmo $\neq$ Análise do problema

- Análise do pior caso: avaliamos o tempo de execução mais longo para qualquer instância de tamanho n.
- Avaliamos o tempo de execução de pior caso do melhor algoritmo que pode existir para o problema.

Analisar um problema é difícil!

Análise de um problema

Análise do algoritmo $\neq$ Análise do problema

- Análise do pior caso: avaliamos o tempo de execução mais longo para qualquer instância de tamanho n.
- Avaliamos o tempo de execução de pior caso do melhor algoritmo que pode existir para o problema.

Analisar um problema é difícil!

Às vezes, já é suficiente saber se **existe** algum algoritmo eficiente!

Análise de problemas

Como comparar dois algoritmos para um problema?

- Comparar a complexidade de cada algoritmo.

Como comparar dois problemas A e B? (A é "mais fácil" do que B? ...)

Análise de problemas

Como comparar dois algoritmos para um problema?

- Comparar a complexidade de cada algoritmo.

Como comparar dois problemas A e B? (A é "mais fácil" do que B? ...)

- Em alguns casos, podemos comparar as cotas de cada problema.
  Exemplo: Achar o máximo é mais fácil que ordenar um vetor.
  - Máximo tem cota superior $O(n)$,
  - Ordenação tem cota inferior $\Omega(n \log n)$.

Análise de problemas

Como comparar dois algoritmos para um problema?

- Comparar a complexidade de cada algoritmo.

Como comparar dois problemas A e B? (A é "mais fácil" do que B? ...)

- Em alguns casos, podemos comparar as cotas de cada problema.
  Exemplo: Achar o máximo é mais fácil que ordenar um vetor.
  - Máximo tem cota superior $O(n)$,
  - Ordenação tem cota inferior $\Omega(n \log n)$.
- Esse caso é raro, pois conhecemos poucas cotas inferiores. Para comparar A e B, precisamos de uma nova ferramenta chamada redução de problemas.

Redução de problemas

Problema A:
- Instância: I_A
- Solução: S_A

Problema B:
- Instância: I_B
- Solução: S_B

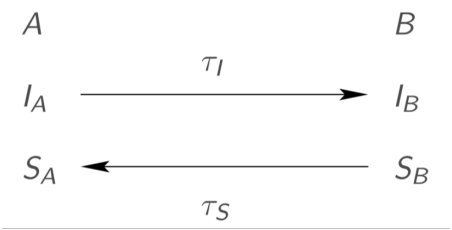
Redução

Uma redução do problema A ao problema B é um par de algoritmos τ_I e τ_S tais que:
- τ_I transforma uma instância I_A de A em uma instância I_B de B;
- τ_S transforma uma solução S_B de I_B em uma solução S_A de I_A.

Redução de problemas

- Se temos uma redução, e existe um algoritmo Alg_B para o problema B, então ...

Redução de problemas



Alg_A (I_A)

1: $I_B \leftarrow \tau_I(I_A)$
2: $S_B \leftarrow \mathrm{Alg}_B(I_B)$
3: $S_A \leftarrow \tau_S(S_B)$
devolva S_A

- Se temos uma redução, e existe um algoritmo Alg_B para o problema B, então **...**
- Existe um algoritmo Alg_A para o problema A (**!**).

Redução de problemas

$\text{Alg}_A\ (I_A)$

1: $I_B \leftarrow \tau_I(I_A)$
2: $S_B \leftarrow \text{Alg}_B(I_B)$
3: $S_A \leftarrow \tau_S(S_B)$
devolva S_A

- Se temos uma redução, e existe um algoritmo Alg_B para o problema B, então **...**
- Existe um algoritmo Alg_A para o problema A (**!**).
- A redução precisa ser **válida**: **para toda entrada** I_A de um problema A, a saída obtida precisa ser uma saída correta para esse problema:

$$(I_A, \tau_S(\text{Alg}_B(\tau_I(I_A)))) \in A.$$

Redução de problemas

$\text{Alg}_A\ (I_A)$

1: $I_B \leftarrow \tau_I(I_A)$
2: $S_B \leftarrow \text{Alg}_B(I_B)$
3: $S_A \leftarrow \tau_S(S_B)$
devolva S_A

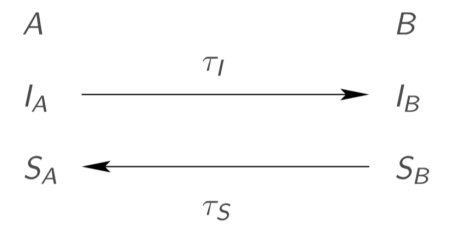
Se a redução é "rápida" (não é mais custosa do que qualquer Alg_B):

A **não é "mais difícil"** do que B.

=

B é **pelo menos tão difícil** quanto A.

Redução de problemas



$\text{Alg}_A\ (I_A)$

1: $I_B \leftarrow \tau_I(I_A)$
2: $S_B \leftarrow \text{Alg}_B(I_B)$
3: $S_A \leftarrow \tau_S(S_B)$
devolva S_A

Quando usar reduções?

1. Comparar dois problemas: A **não é "mais difícil"** do que B?

Redução de problemas

$\text{Alg}_A\ (I_A)$

1: $I_B \leftarrow \tau_I(I_A)$
2: $S_B \leftarrow \text{Alg}_B(I_B)$
3: $S_A \leftarrow \tau_S(S_B)$
devolva S_A

Quando usar reduções?

1. Comparar dois problemas: A **não é "mais difícil"** do que B?
2. Encontrar um **algoritmo** para resolver A (**cota superior**).

Redução de problemas

$\text{Alg}_A\ (I_A)$

1: $I_B \leftarrow \tau_I(I_A)$
2: $S_B \leftarrow \text{Alg}_B(I_B)$
3: $S_A \leftarrow \tau_S(S_B)$
devolva S_A

Quando usar reduções?

1. Comparar dois problemas: A **não é "mais difícil"** do que B?
2. Encontrar um **algoritmo** para resolver A (**cota superior**).
3. Encontrar uma **cota inferior** para B. – Veremos depois como fazer isto.

Redução de problemas

Um cientista da computação pergunta a um físico: "Como você faz para ferver água?" – "Muito simples", responde o físico. "É só ir na cozinha, encher a chaleira com água, acender o fogão, e colocar a água para ferver."

Redução de problemas

Um cientista da computação pergunta a um físico: "Como você faz para ferver água?" – "Muito simples", responde o físico. "É só ir na cozinha, encher a chaleira com água, acender o fogão, e colocar a água para ferver."

"Correto", diz o cientista da computação. "Agora, e se o gás estiver ligado? Como você ferve a água?" – "Não tem problema", responde o físico. "Basta ir na cozinha, encher a chaleira com água, e colocar ela no fogão."

Redução de problemas

Um cientista da computação pergunta a um físico: "Como você faz para ferver água?" – "Muito simples", responde o físico. "É só ir na cozinha, encher a chaleira com água, acender o fogão, e colocar a água para ferver."

"Correto", diz o cientista da computação. "Agora, e se o gás estiver ligado? Como você ferve a água?" – "Não tem problema", responde o físico. "Basta ir na cozinha, encher a chaleira com água, e colocar ela no fogão."

"É bem mais simples do que isso!", garante o cientista da computação. "Apenas desliga o gás, e então já sabemos como resolver o problema!"

Complexidade da redução

A complexidade $f(n)$ de uma redução do problema A ao problema B é a soma dos tempos das transformações τ_I e τ_S.

Um problema A é redutível a um problema B em tempo $f(n)$ se existe uma redução válida de A para B cuja complexidade é $O(f(n))$.

- **Notação**: $A \propto_{f(n)} B$ (quando queremos indicar a complexidade).
- **Notação**: $A \propto B$ (quando não queremos indicar a complexidade).

Complexidade da redução

Estamos interessados apenas em **reduções polinomiais**.

Um problema A é polinomialmente redutível a um problema B se existe uma redução válida de A para B cuja complexidade é polinomial.

- **Notação**: $A \propto_{\text{poli}} B$ (indicando que a redução é polinomial).

Complexidade da redução

Estamos interessados apenas em **reduções polinomiais**.

Um problema A é polinomialmente redutível a um problema B se existe uma redução válida de A para B cuja complexidade é polinomial.

- **Notação**: $A \propto_{\text{poli}} B$ (indicando que a redução é polinomial).
- Se B **pode** ser resolvido por um algoritmo polinomial, então A também **pode**.

Complexidade da redução

Estamos interessados apenas em **reduções polinomiais**.

Um problema A é polinomialmente redutível a um problema B se existe uma redução válida de A para B cuja complexidade é polinomial.

- **Notação**: $A \propto_{\text{poli}} B$ (indicando que a redução é polinomial).
- Se B **pode** ser resolvido por um algoritmo polinomial, então A também **pode**.
- **Mais do que isso**:
  Se A **não pode** ser resolvido em tempo polinomial, então B também **não pode**.

Redução de problemas: Exemplo 1

Problema do casamento cíclico de strings (CSM)

Entrada: um alfabeto Σ, e duas strings sobre Σ de tamanho n:

$$A = a_0 a_1 \dots a_{n-1} \text{ e } B = b_0 b_1 \dots b_{n-1}$$

Objetivo: decidir se B é um deslocamento cíclico de A.
Ou seja, determinar se existe um $k \in \{0, 1, \dots, n-1\}$ tal que $a_{(i+k) \mod n} = b_i$, $\forall i = 0, 1, \dots, n-1$.

Exemplo: para $A = acgtact$ e $B = gtactac$ $(n = 7)$, temos $k = 2$.

Como resolver CSM?

Redução de problemas: Exemplo 1

Problema do casamento de strings (SM, String Matching)

Entrada: um alfabeto Σ e duas strings sobre Σ:

$$A = a_0 a_1 \ldots a_{n-1} \text{ e } B = b_0 b_1 \ldots b_{m-1}, \text{ com } m \leq n.$$

Objetivo: decidir se B é subcadeia de A.
Ou seja, determinar o menor $k \in \{0, 1, \ldots, n-1\}$ tal que $a_{(i+k)} = b_i$, $\forall i = 0, 1, \ldots, m-1$, ou devolver $k = -1$ se B não é subcadeia de A.

Exemplo: para $A = acgttaccgtacccg$ e $B = tac$ ($n = 15$ e $m = 3$), temos $k = 4$.

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

- Se k é a solução de **SM** para $I_{SM} \Rightarrow k$ também é a solução de **CSM** para I_{CSM}.

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

- Se k é a solução de **SM** para $I_{SM} \Rightarrow k$ também é a solução de **CSM** para I_{CSM}.
- τ_S custa $O(1)$, e a redução custa $O(n)$.

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

- Se k é a solução de **SM** para $I_{SM} \Rightarrow k$ também é a solução de **CSM** para I_{CSM}.
- τ_S custa $O(1)$, e a redução custa $O(n)$.

Exemplo:

- $I_{CSM} = (acgtact, gtactac, 7)$

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

- Se k é a solução de **SM** para $I_{SM} \Rightarrow k$ também é a solução de **CSM** para I_{CSM}.
- τ_S custa $O(1)$, e a redução custa $O(n)$.

Exemplo:

- $I_{CSM} = (acgtact, gtactac, 7)$
- $I_{SM} = (acgtactacgtact, 14, gtactac, 7)$

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

- Se k é a solução de **SM** para $I_{SM} \Rightarrow k$ também é a solução de **CSM** para I_{CSM}.
- τ_S custa $O(1)$, e a redução custa $O(n)$.

Exemplo:

- $I_{CSM} = (acgtact, gtactac, 7)$
- $I_{SM} = (acgtactacgtact, 14, gtactac, 7)$
- $S_{SM} = \{2\} = S_{CSM}$

Redução de problemas: Exemplo 1

Redução: **CSM** $\propto_n$ **SM**.

- Instância de **CSM**: $I_{CSM} = (A, B, n)$.
- τ_I constrói a instância de **SM** em tempo $O(n)$:

$$I_{SM} = (A', 2n, B, n) \text{ , onde } A' = A + A \text{ (concatenação).}$$

- Se k é a solução de **SM** para $I_{SM} \Rightarrow k$ também é a solução de **CSM** para I_{CSM}.
- τ_S custa $O(1)$, e a redução custa $O(n)$.

Exemplo:

- $I_{CSM} = (acgtact, gtactac, 7)$
- $I_{SM} = (acgtactacgtact, 14, gtactac, 7)$
- $S_{SM} = \{2\} = S_{CSM}$

O problema **SM** pode ser resolvido **em tempo linear** ($O(n + m)$) pelo algoritmo de Knuth, Morris, Pratt (1977).

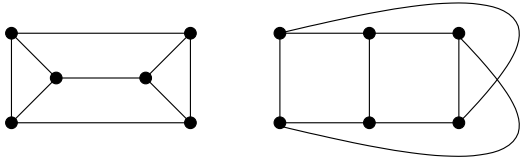
Redução de problemas: Exemplo 2

Problema da existência de triângulo (PET)

Entrada: grafo conexo $G = (V, E)$ com $|V| = n$ e $|E| = m$, representado por uma matriz de adjacência.

Objetivo: decidir se G contém um triângulo.

Exemplo:



SIM NAO

Redução de problemas: Exemplo 2

Problema da existência de triângulo (PET)

- Algoritmo trivial de complexidade $O(n^3)$: verificar todas as triplas de vértices.
- Existe um algoritmo $O(mn)$ que é melhor para grafos esparsos:
  para cada aresta (u, v) verificar se existe um vértice w adjacente a u e v.

Redução de problemas: Exemplo 2

Problema da Multiplicação de Matrizes Quadradas (MMQ)

Entrada: duas matrizes quadradas A e B de ordem n.

Objetivo: calcular o produto $A \times B$.

Redução de problemas: Exemplo 2

Ideia da redução **PET** $\propto$ **MMQ**:

- Seja A a matriz de adjacência de G. Se $A^2 = A \times A$, então $a_{ij}^2 = \sum_{k=1}^{n} a_{ik} a_{kj}$.

Redução de problemas: Exemplo 2

Ideia da redução **PET** $\propto$ **MMQ**:

- Seja A a matriz de adjacência de G. Se $A^2 = A \times A$, então $a_{ij}^2 = \sum_{k=1}^{n} a_{ik} a_{kj}$. Então:

$$a_{ij}^2 > 0 \Leftrightarrow \exists k \in \{1, \ldots, n\} \text{ tal que } a_{ik} = a_{kj} = 1.$$

Redução de problemas: Exemplo 2

Ideia da redução **PET** $\propto$ **MMQ**:

- Seja A a matriz de adjacência de G. Se $A^2 = A \times A$, então $a^2_{ij} = \sum_{k=1}^{n} a_{ik} a_{kj}$. Então:

$$a^2_{ij} > 0 \Leftrightarrow \exists k \in \{1, \ldots, n\} \text{ tal que } a_{ik} = a_{kj} = 1.$$

- Portanto, se $a_{ij} = 1$, então (i, j, k) corresponde a um triângulo para algum k, se e somente se $a^2_{ij} > 0$.

Redução de problemas: Exemplo 2

Redução: **PET** $\propto$ **MMQ**

- Instância de **PET**: $I_{PET} = A$.
- τ_I constrói a instância de **MMQ**: $I_{MMQ} = (A, A, n)$.
- τ_S constrói a resposta de **PET** pelo algoritmo abaixo, que recebe $A^2 = P$. Veja que τ_S custa $O(n^2)$.

$\tau_S\ (P)$

```
1: para cada i = 1 ate n faça
2:     para cada j = 1 ate n faça
3:         se a_ij = 1 e p_ij > 0 então
4:             devolva SIM
5: devolva NÃO
```

Redução de problemas: Exemplo 2

Observações: O algoritmo clássico para a **MMQ** possui complexidade $O(n^3)$.

Redução de problemas: Exemplo 2

Observações: O algoritmo clássico para a **MMQ** possui complexidade $O(n^3)$.
Mas, o problema **MMQ** pode ser resolvido em tempo:

- $O(n^{\log 7 = 2.807})$ pelo algoritmo de divisão e conquista de Strassen;

Redução de problemas: Exemplo 2

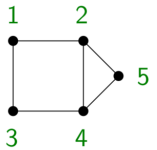
Observações: O algoritmo clássico para a **MMQ** possui complexidade $O(n^3)$.
Mas, o problema **MMQ** pode ser resolvido em tempo:

- $O(n^{\log 7 = 2.807})$ pelo algoritmo de divisão e conquista de Strassen;
- $O(n^{2.376})$ pelo algoritmo de Coppersmith e Winograd.

Redução de problemas: Exemplo 2

Observações: O algoritmo clássico para a **MMQ** possui complexidade $O(n^3)$.
Mas, o problema **MMQ** pode ser resolvido em tempo:

- $O(n^{\log 7 = 2.807})$ pelo algoritmo de divisão e conquista de Strassen;
- $O(n^{2.376})$ pelo algoritmo de Coppersmith e Winograd.



$A(G)$

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

$P = A \times A$

	1	2	3	4	5
1	2	0	0	2	1
2	0	3	2	1	1
3	0	2	2	0	1
4	2	1	0	3	1
5	1	1	1	1	2

Redução de problemas: Erros comuns

Erros comuns:

- Usar a redução na **ordem inversa**: em vez de fazer a redução $A \propto B$, fazer a redução $B \propto A$, e concluir que B é pelo menos tão difícil quanto A.

Redução de problemas: Erros comuns

Erros comuns:

- Usar a redução na **ordem inversa**: em vez de fazer a redução $A \propto B$, fazer a redução $B \propto A$, e concluir que B é pelo menos tão difícil quanto A.
- Achar que, na redução $A \propto B$, toda instância de B tem que ser mapeada (gerada) a partir de alguma instância de A.

  Geramos a instância de B que seja conveniente (útil) para resolver A!

Resumo

Síntese

- A redução de problemas é útil para criar **algoritmos mais eficientes** que possam resolver alguns problemas.
- Podemos usar a redução para comparar a **complexidade de problemas**.
- Podemos usar reduções para encontrar **cotas inferiores** de problemas (a ser estudado posteriormente).

Material bibliográfico e exercícios

U. Manber. Introduction to Algorithms. – **Cap. 10**

Exercícios: ver exercícios no final do Capítulo 10.

Dúvidas

Dúvidas?