CI1055: Algoritmos e Estruturas de Dados I

Profs. Drs. Marcos Castilho, Bruno Müller Jr, Carmem Hara

Departamento de Informática/UFPR

13 de agosto de 2020

Resumo

Aplicações de vetores: permutações

Objetivos da aula

- Resolver problemas que envolvem o uso de vetores
- Discutir eficiência dos algoritmos

Motivação

- O aprendizado de vetores é fortalecido quando procuramos resolver problemas variados que podem ser implementados com seu uso
- O entendimento da diferença entre o índice do vetor e o conteúdo ao qual este índice aponta é importante
- Vamos apresentar um problema matemático conhecido como *permutação* e tentar resolvê-lo usando vetores
- Vamos também discutir a eficiência das diferentes soluções

Definição

- Uma permutação é uma função bijetora de um conjunto nele mesmo
- Intuitivamente, é uma maneira de reordenar os elementos do conjunto (que não tem elementos repetidos)

Exemplo

Seja o conjunto $\{1, 2, 3, 4, 5\}$:

Uma permutação pode ser esta:

$$P(1) = 4,$$
$$P(2) = 1,$$
$$P(3) = 5,$$
$$P(4) = 2,$$
$$P(5) = 3.$$

Esquematicamente:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

Outra pode ser esta:

$$P(1) = 2,$$
$$P(2) = 5,$$
$$P(3) = 1,$$
$$P(4) = 3,$$
$$P(5) = 2.$$

Esquematicamente:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 3 & 2 \end{pmatrix}$$

Número de diferentes permutações

- Existem $n!$ maneiras de se reordenar os n elementos de um conjunto
- Se $n = 5$ então existem 120 permutações

Representação computacional

- Pode-se usar um vetor para modelar uma permutação
- Seja V um vetor de n posições inteiras
- Cada posição é um valor (sem repetição) entre 1 e n.
- Por exemplo:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

1	2	3	4	5
4	1	5	2	3

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 3 & 2 \end{pmatrix}$$

1	2	3	4	5
2	5	1	3	2

Problemas com permutações

- Testar se uma representação corresponde a uma permutação
- Gerar aleatoriamente uma representação correspondente a uma permutação
- Determinar a ordem de uma permutação

Testar se uma representação é uma permutação

- Dado um vetor qualquer, ele pode ser a representação de alguma permutação do conjunto de números entre 1 e n ?
- Computacionalmente, deve-se testar de todos os números entre 1 e n estão presentes no vetor
- Isto é, não pode haver números repetidos nem faltar ninguém

Algoritmo 1

- Para todos os números entre 1 e n
- Testa se este número está presente no vetor
- Se todos estiverem, dado que são n números em um vetor de n elementos, é porque não houve repetição
- Logo, a permutação é válida

Implementação do algoritmo 1

```
1  function testa_permutacao (var v: vetor; n: integer): boolean;
2  var i, j: integer;
3      eh_permutacao: boolean;
4  begin
5      eh_permutacao:= true;
6      i:= 1;
7      while eh_permutacao and (i <= n) do
8      begin
9          j:= 1;                (* procura se i esta no vetor *)
10         while (j <= n) and (v[j] <> i) do
11             j:= j + 1;
12         if v[j] <> i then (* se nao achou nao eh permutacao *)
13             eh_permutacao:= false;
14         i:= i +1;
15     end;
16     testa_permutacao:= eh_permutacao;
17 end; (* testa_permutacao *)
```

Análise do algoritmo 1

- Melhor caso: ele procura o elemento 1 e não acha no término do laço interno. Executou n comparações
- Pior caso: ocorre quando o vetor representa uma permutação, neste caso o algoritmo executou n^2 comparações
- Pode ser mais eficiente?

Algoritmo 2

- Usa um vetor auxiliar, inicialmente zerado
- Percorre-se o vetor de entrada e para cada índice:
  - tenta inserir este elemento no vetor auxiliar
  - a inserção é válida se naquela posição do vetor auxiliar houver um zero
  - caso exista um valor diferente de zero, é porque existe um elemento repetido e a permutação não é válida
- Se conseguiu inserir todos os elementos, é porque a permutação é válida

Implementação do algoritmo 2

```
1  function testa_permutacao_v2 (var v: vetor; n: integer): boolean;
2  var i: integer;
3      aux: vetor;
4      eh_permutacao: boolean;
5  begin
6      zerar_vetor (aux,n);
7      eh_permutacao:= true;
8      i:= 1;
9      while eh_permutacao and (i <= n) do
10     begin
11         if (v[i] >= 1) and (v[i] <= n) and (aux[v[i]] = 0) then
12             aux[v[i]]:= v[i]
13         else
14             eh_permutacao:= false;
15         i:= i + 1;
16     end;
17     testa_permutacao_v2:= eh_permutacao;
18 end; (* testa_permutacao_v2 *)
```

Análise do algoritmo 2

- Melhor caso: executou n atribuições para zerar o vetor auxiliar. Em seguida, inseriu um elemento com sucesso. Logo depois, não conseguiu inserir o segundo. Custou portanto $n + 2$ operações
- Pior caso: ocorre quando o vetor representa uma permutação, neste caso o algoritmo executou n atribuições e n comparações, em um custo total de $n + n = 2n$ operações
- Isto é, o melhor caso é similar, mas o pior caso é bem mais eficiente

Gerando permutações válidas

- O próximo problema é gerar randomicamente um vetor que represente uma permutação válida
- Faremos uso da função `random` da linguagem *Pascal*

Algoritmo 1

- Gera um vetor aleatoriamente e testa usando a função anterior
- Repete este processo até gerar uma permutação válida

Implementação do algoritmo 1

```
1  procedure gerar_permutacao (var v: vetor; n: integer);
2  var i: integer;
3  begin
4      randomize;
5      repeat
6          for i:= 1 to n do
7              v[i]:= random (n) + 1; (* sorteia num entre 1 e n *)
8      until testa_permutacao_v2 (v, n);
9  end; (* gera_permutacao *)
```

Análise do algoritmo 1

- Algoritmos com componentes randômicos não são triviais de serem analisados analiticamente
  - Análise de Algoritmos!
- Por isso faremos uma análise empírica

Análise do algoritmo 1

- Muito lento quando n cresce
- É muito pouco provável que os vetores sejam gerados em repetição
- Experimentalmente, para valores acima de 13 ou 14 o tempo de relógio é inaceitável

Algoritmo 2

- Para cada índice do vetor, sortear o conteúdo
- Em seguida, verifica se algum elemento anteriormente atribuído é igual
- Caso seja, sorteia-se outro, até que a verificação retorne OK
- Isto garante que o vetor final produzido é válido

Implementação do algoritmo 2

```
1  procedure gerar_permutacao_v2 (var v: vetor; n: integer);
2  var i, j: integer;
3  begin
4      randomize;
5      v[1]:= random (n) + 1;
6      for i:= 2 to n do
7          repeat
8              v[i]:= random (n) + 1;  (* gera um numero entre 1 e n
                   *)
9              j:= 1; (* procura se o elemento ja existe no vetor *)
10             while (j < i) and (v[i] <> v[j]) do
11                 j:= j + 1;
12         until j = i;  (* descobre que o elemento eh novo *)
13 end; (* gera_permutacao_v2 *)
```

Análise do algoritmo 2

- Melhor que o anterior
- Executa na casa de 2 segundos para vetores de tamanho até 1000
- Entradas de temanho 30.000 demora cerca de 20 segundos
- Vetores maiores também demoram muito

Algoritmo 3

- Inicializa-se um vetor de forma ordenada
- Depois faz-se alterações aleatórias de seus elementos um número também aleatório de vezes

Implementação do algoritmo 3

```
1  procedure gerar_permutacao_v3 (var v: vetor; n: integer);
2  var i, j, k, aux, max: integer;
3  begin
4      for i:= 1 to n do
5          v[i] := i;
6
7      randomize;
8      max:= random (1000); (* troca dois elementos max vezes *)
9      for i:= 1 to max do
10     begin
11         j:= random (n) + 1;
12         k:= random (n) + 1;
13         aux:= v[j];
14         v[j]:= v[k];
15         v[k]:= aux;
16     end;
17 end; (* gera_permutacao_v3 *)
```

Análise do algoritmo 3

- Melhor que o anterior
- Produz corretamente vetores que representam permutações
- É muito rápido e mistura bem os números
- Foi testado com entradas da ordem de um milhão de elementos

Algoritmo 4

- Inicializar um vetor auxiliar com números sequenciais de 1 até n
- Em seguida se realizam n sorteios de índices do vetor auxiliar
- Para um índice j sorteado, copia-se $aux[j]$ no vetor permutação
- Em seguida diminui-se o tamanho do auxiliar, copiando o último sobre $v[j]$

Implementação do algoritmo 4

```
1  procedure gerar_permutacao_v4 (var v: vetor_i; n: longint);
2  var i, j, tam: longint;
3      aux: vetor_i;
4  begin
5      for i := 1 to n do
6          aux[i] := i;
7
8      randomize;
9      tam:= n;
10     for i := 1 to n do
11     begin
12         j := random(tam) + 1;
13         v[i] := aux[j];
14         aux[j] := aux[tam];
15         tam := tam - 1;
16     end;
17 end; (* gera_permutacao_v4 *)
```

Análise do algoritmo 4

- Executa em tempo linear!
- Produz corretamente vetores que representam permutações
- É muito rápido e mistura bem os números
- Foi testado com entradas da ordem de um milhão de elementos

Cálculo da ordem de uma permutação

O que é a ordem de uma permutação?

- Como $P(n)$ é uma permutação, então $P(P(n))$ também é
- Seja $P^2(n) = P(P(n))$
- Pode-se calcular, por exemplo, $P(P(1))$, isto é, $P^2(1)$

Exemplo

Considere a permutação:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

Então pode-se calcular:

- $P(P(1)) = P(4) = 2$.
- $P(P(2)) = P(1) = 4$.
- $P(P(3)) = P(5) = 3$.
- $P(P(4)) = P(2) = 1$.
- $P(P(5)) = P(3) = 5$.

Definição

$$\begin{cases} P^1(n) = P(n); \\ P^k(n) = P(P^{k-1}(n)), \quad k \geq 2. \end{cases}$$

Definição: permutação identidade

Quando $P(i) = i, \forall i$ a permutação recebe o nome de permutação identidade, ID.

$$ID = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Definição: ordem da permutação

Seja $P(n)$ uma permutação sobre um conjunto de n elementos. Então existe um número natural k tal que $P^k = ID$.

Este número natural é chamado da *ordem* da permutação.

Exemplo

Considere a permutação:

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

Podemos calcular para valores pequenos de k como é P^k

Exemplo

$$P^1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

Exemplo

$$P^2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 3 & 1 & 5 \end{pmatrix}$$

Exemplo

$$P^3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 4 & 3 \end{pmatrix}$$

Exemplo

$$P^4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 3 & 2 & 5 \end{pmatrix}$$

Exemplo

$$P^5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 1 & 3 \end{pmatrix}$$

Exemplo

$$P^6 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Isto é, a ordem de P é 6

O problema computacional

- Queremos um algoritmo para calcular a ordem de uma permutação qualquer
- Este problema caiu na maratona de programação da ACM
- O algoritmo que gera todas as k permutações é ineficiente, teria que simular todo o processo e a cada etapa testar se o resultado é a permutação ID
- Existe um método melhor

Ideia do algoritmo

- Cada elemento $P(i) = x$ do conjunto retorna à posição i ciclicamente de *cont* em *cont* permutações
- Ou seja, $P^{cont}(i) = x, P^{2 \times cont}(i) = x, \ldots$
- O mesmo ocorre para todos elementos do conjunto, mas cada um possui um ciclo (valor de *cont*) próprio

Exemplo

Considere a permutação:

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

- Para o índice 1, temos que $P^3(1) = 1$
- Isto quer dizer que para todo múltiplo de 3 (a cada 3 iterações) $P^{3k}(1) = 1$

Exemplo

- Para este exemplo, temos que:
  - $P^3(1) = 1$
  - $P^3(2) = 1$
  - $P^3(4) = 1$
  - $P^2(3) = 1$
  - $P^2(5) = 1$

Conclusão

- Pode-se concluir que a permutação *ID* ocorrerá na iteração que é o mínimo múltiplo comum (MMC) entre o número que provoca repetição entre todos os índices.
- Observação:
  - $MMC(x_1, x_2, \ldots, x_n) = MMC(x_1, MMC(x_2, \ldots, x_n))$

Algoritmo para o MMC

- Não existe algoritmo eficiente para cálculo do MMC
- Mas:
  - Existe para o MDC!

$$MDC(a, b) = \frac{a \times b}{MMC(a, b)}$$

Implementação deste algoritmo

```
1  function ordem_permutacao (var v: vetor_i; n: integer): int64;
2  var mmc, cont: int64;
3      p, i: integer;
4  begin
5      mmc := 1;
6      for i := 1 to n do
7      begin
8          cont := 1;
9          p := i;
10         while (v[p] <> i) do
11         begin
12             cont:= cont + 1;
13             p := v[p];
14         end;
15         mmc := mmc * cont div mdc(mmc, cont);
16     end;
17     ordem_permutacao:= mmc;
18 end;
```

Em resumo...

- Aprendemos a usar vetores como estrutura de dados para resolver um problema matemático
- Relembramos que existem algoritmos melhores do que outros, sob algum aspecto
- Fizemos um breve estudo sobre os algoritmos implementados
- Estudamos um problema de maratona de programação!

Fim do tópico

- este material está no livro no capítulo 9, seção 9.7.1

Licença

- Slides feitos em LaTeX usando beamer
- Licença

  Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.`http://creativecommons.org/licenses/by-nc-nd/2.5/br/`

  Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.`http://creativecommons.org/licenses/by-nc-nd/2.5/br/`