Random Iterator

Otávio Vasques

Março 2016

1 Uniformidade

A proposta desse relatório é atestar a uniformidade da distribuição das permutações de uma sequência de N números dados M pontos gerados pelo iterador da classe `RandomIterator.java`. Pode-se utilizar dois métodos:

1. Método visual
2. χ^2

1.1 Método Visual

Podemos fazer um mosaico(Figure 1) de histogramas variando os dois parâmetros da simulação N e T.

Vê-se que conforme se aumenta o valor de T a distribuição vai se adequando ao padrão uniforme.

1.2 χ^2

Utilizando o método do χ^2 devemos ajustar sobre os histogramas anteriores a distribuição abaixo:

$$y = a = \bar{x}$$

Usando o teste de χ^2 espera-se

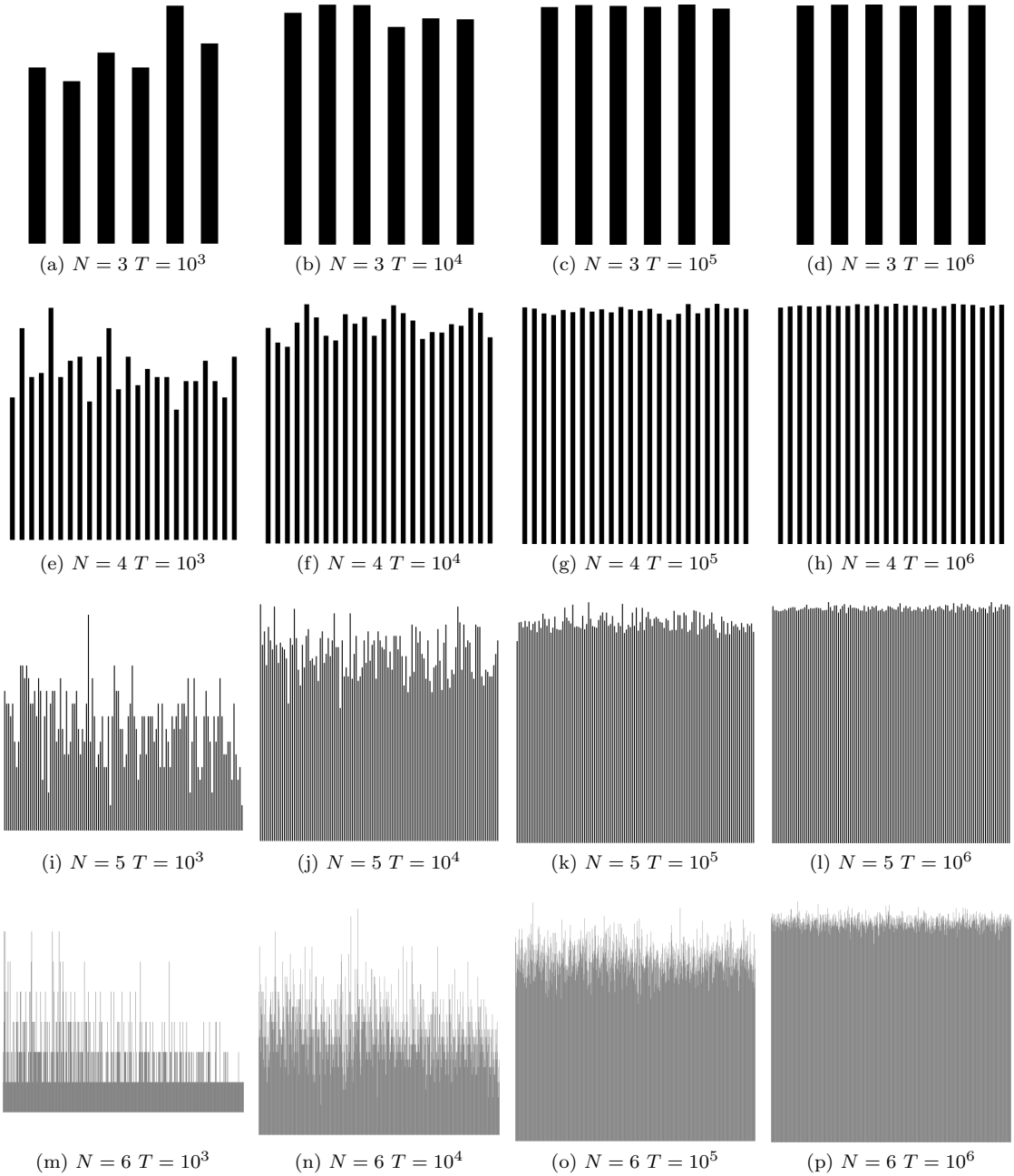
$$\chi^2 = T - 1$$

Podemos, então, fazer a mesma tabela porém para os χ^2 (Table 1):

A partir dessa tabela podemos fazer um gráfico de χ^2 reduzido em função de T (Figure 2):

2 Algoritmo

O iterador abaixo tem como objetivo visitar todos os elementos em uma ordem aleatória sem repetições, os pontos fundamentais para seu funcionamento são a não repetição dos elementos e a equiprobabilidade de se gerar qualquer uma das



(a) $N = 3$ $T = 10^3$ (b) $N = 3$ $T = 10^4$ (c) $N = 3$ $T = 10^5$ (d) $N = 3$ $T = 10^6$

(e) $N = 4$ $T = 10^3$ (f) $N = 4$ $T = 10^4$ (g) $N = 4$ $T = 10^5$ (h) $N = 4$ $T = 10^6$

(i) $N = 5$ $T = 10^3$ (j) $N = 5$ $T = 10^4$ (k) $N = 5$ $T = 10^5$ (l) $N = 5$ $T = 10^6$

(m) $N = 6$ $T = 10^3$ (n) $N = 6$ $T = 10^4$ (o) $N = 6$ $T = 10^5$ (p) $N = 6$ $T = 10^6$

Figure 1: Histogramas das permutações para diferentes valores de N e T

Table 1: Tabela de χ^2

N \ T	10^3	10^4	10^5	10^6
3	1179.33	12343.3	229717	944961
4	893.333	7717.33	126599	622401
5	1088.67	9876.67	75052.7	800927
6	525.885	10313.1	105659	935983

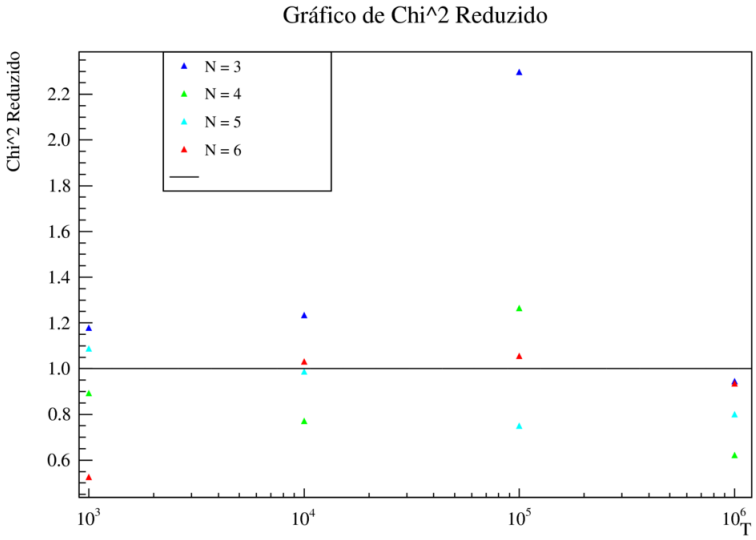


Figure 2: Gráfico de χ^2 Reduzido

$N!$ permutações. O algoritmo se baseia na natureza randômica de `RandomQueue`, sem a necessidade de uma ordem específica pode-se manipular os elementos de qualquer forma para produzir o resultado desejado.

```
private class RandomIterator implements Iterator<Item> {
    private int i = k;

    public boolean hasNext() {
        return i > 0;
    }

    public Item next() {
        Random rdm = new Random();
        int position = rdm.nextInt(i);
        Item temp = content[position];
        content[position] = content[i-1];
```

```
            content[i-1] = temp;
            return content[--i];
        }

         public void remove() {}
    }
```

Em cada vez que `next()` é chamado sorteia-se um uma posição qualquer entre $[0, i[$ e retorna-se esse valor. Em cada chamada troca-se este elemento com o último e decrementa-se o valor de i, impedindo que ele seja sorteado novamente. Supondo que a função `nextInt()` possui um comportamento uniforme a cada etapa os i elementos restantes possuem sempre a mesma probabilidade de serem sorteados. Vemos abaixo um exemplo de execução para $N = 5$.

```
chamadas de next() |   i   |      content    | position | return
-------------------+-------+-----------------+----------+--------
        0          |   5   | [1, 2, 3, 4, 5] |   null   |  null
        1          |   4   | [1, 2, 3, 5, 4] |    4     |   4  -+
        2          |   3   | [1, 5, 3, 2, 4] |    2     |   2   | s
        3          |   2   | [3, 5, 1, 2, 4] |    1     |   1   | e
        4          |   1   | [5, 3, 1, 2, 4] |    1     |   3   | q
        5          |   0   | [5, 3, 1, 2, 4] |    1     |   5  -+
```

Caso `RandomQueue` não pudesse ter a ordem alterada consumiria-se N de espaço extra, um novo vetor seria construído para rastrear os índices ou os elementos que já foram sorteados.