Aluno: Jogno Vezú Júnior

Matrícula: 1815080035

Simulação e Análise de Desempenho

Gerador de números pseudoaleatórios

O gerador de números aleatórios *Xorshift* produz uma sequência de $2^{32} - 1$ de *x* inteiros, ou uma sequência de $2^{64} - 1$ pares *(x, y)*, ou uma sequência de $2^{64} - 1$ triplos *(x, y, z)*, e assim por diante, por meio do uso repetido de uma construção simples de computador: a operação "exclusive-or" (xor) de uma palavra de computador com uma versão deslocada de si mesma. Em C, a operação básica é: y^ *(y << a)* para deslocamentos à esquerda, *y^ (y >> a)* para deslocamentos à direita.

Combinar tais operações *Xorshift* para vários deslocamentos e argumentos fornece RNGs extremamente rápidos e simples que parecem ter um bom desempenho em testes de aleatoriedade. Para dar uma ideia do poder e da eficácia das operações *Xorshift*, aqui está a parte essencial de um procedimento em C que, com apenas três operações *Xorshift* por chamada, fornecerá $2^{128} - 1$inteiros aleatórios de 32 bits, dado quatro *seeds* aleatórios *x, y, z, w*:

```
tmp = (x ^ (x << 15));
x = y;
y = z;
z = w;
return w = (w ^ (w >> 21)) ^ (tmp ^ (tmp >> 4));

```

Tal procedimento é muito rápido, tipicamente acima de 200 milhões por segundo, e os inteiros aleatórios resultantes passam em todos os testes de aleatoriedade aos quais foram submetidos.

Um modelo matemático para a maioria dos RNGs pode ser colocado na seguinte forma: Temos um conjunto de sementes Z composto por m-tuplas $(x_1, x_2, \ldots, x_m)$ e uma função bijetora $f(\,)$ em Z. Mais comumente, Z é apenas um conjunto de inteiros, mas para melhores RNGs, pode ser um conjunto de pares, triplos, etc. Se z é escolhido uniformemente e aleatoriamente de Z, então a saída do RNG é a sequência $f(z), f^2(z), f^3(z), \ldots$, onde $f^2(z)$ significa $f\big(f(z)\big)$, etc. Porque f é 1-1 sobre Z, a variável aleatória $f(z)$ é ela mesma uniforme sobre Z, assim como $f^2(z)$; na verdade, cada elemento da sequência $f(z), f^2(z), \ldots$ é uniformemente distribuído sobre o conjunto de sementes Z, mas eles não são independentes.

Para o RNGs *Xorshift*, o conjunto de sementes Z é o conjunto de vetores binários $1 \times n$, $\beta = (b_1, b_2, \ldots, b_n)$ excluindo o vetor zero. Normalmente, n será 32, 64, 96, etc., de modo que seus elementos possam ser formados ao juntar palavras de 32 bits do computador. Os elementos dos vetores β em Z estão no campo $\{0, 1\}$, de modo que a adição de vetores binários pode ser implementada pela operação XOR das partes de 32 bits constituintes. Para nosso RNG *Xorshift*, precisamos de uma função invertível sobre Z e, para isso, usamos uma transformação linear sobre o espaço vetorial binário, caracterizada por uma matriz binária não singular $n \times n$. Se β é uma escolha aleatória uniforme (o seed) de Z, então cada membro da sequência $\beta T, \beta T^2, \beta T^3, \ldots$, também é uniformemente distribuído sobre Z, então temos uma sequência de elementos ID, Identically Distributed (Identicamente Distribuídos), uniformes de Z, mas eles não são IID, ou seja, Independent Identically Distributed (distribuídos de forma independente e identicamente). Mas verifica-se aqui, como para muitos RNGs, que funções dos elementos ID muitas vezes têm distribuições muito próximas das mesmas funções dos elementos de uma sequência IID. Essa é a propriedade notável de certas escolhas de funções $f(\,)$ sobre conjuntos de sementes Z que justifica sua utilidade em computadores nos últimos cinquenta anos.

Algoritmo

Para a implementação do Xorshift, usamos como base o script fornecido no material disponibilizado no Classroom: Numerical Recipes: The Art of Scientific Computing.

Inicialmente, foi somente definido a Estrutura Xorshift e o seu construtor, recebendo um seed (semente) como parâmetro.

```c
typedef struct {
    uint32_t state;
} Xorshift;

void xorshift_init(Xorshift *x, uint32_t seed) {
    x->state = seed;
}
```

Com isso, foi implementado o algoritmo *Xorshift* em si, para gerar o próximo número pseudoaleatório

```c
uint32_t xorshift_next(Xorshift *x) {
    uint32_t x = x->state;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    x->state = x;
    return x;
}
```

Em seguida foi implementada a função *xorshift_next_event* para que o número gerado permaneça entre 0 e 9. Para armazenar os valores gerados, usamos a função *generate_xorshift_numbers*;

```c
uint32_t xorshift_next_event(Xorshift *x) {
    return xorshift_next(x) % 10;
}

void generate_xorshift_numbers(uint32_t seed, int count, int *numbers) {
    Xorshift x;
    xorshift_init(&x, seed);
    for (int i = 0; i < count; i++) {
        numbers[i] = xorshift_next_event(&x);
    }
}
```

Para realizar a comparação necessária para avaliação do algoritmo, a função *generate_rand_numbers* foi criada para gerar os números através da função *rand()*, gerando os valores também entre 0 e 9. Além disso também foi criada a função *time_function* para medir o tempo de execução dos geradores;

```c
void generate_rand_numbers(int count, int *numbers) {
    for (int i = 0; i < count; i++) {
        numbers[i] = rand() % 10;
    }
}

double time_function(void (*func)(int, int*), int count, int *numbers) {
    clock_t start = clock();
    func(count, numbers);
    clock_t end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC;
}
```

Para realizar o teste do chi-quadrado e verificar a uniformidade dos números gerados. O teste compara as frequências observadas (*observed*) com as frequências esperadas teóricas, assumindo uma distribuição de 10% para cada dígito de 0 a 9. O resultado é o valor que quantifica o quão bem a distribuição observada se ajusta à distribuição esperada.

```c
double chi_squared_test(int *numbers, int count) {
    int observed[10] = {0};
    double expected = count / 10.0;
    double chi_squared = 0.0;

    for (int i = 0; i < count; i++) {
        observed[numbers[i]]++;
    }

    for (int i = 0; i < 10; i++) {
        double diff = observed[i] - expected;
        chi_squared += diff * diff / expected;
    }

    return chi_squared;
}
```

Resultado

Analisando os resultados obtidos para os diferentes tamanhos de amostra (100.000, 1.000.000, 10.000.000 e 100.000.000) usando os métodos `rand()` e Xorshift para geração de números pseudoaleatórios, temos:

```
Resultados para 100000 gerações:
Tempo para rand(): 0.001834 segundos
Teste do chi-quadrado para rand(): 10.156600
Tempo para Xorshift: 0.000001 segundos
Teste do chi-quadrado para Xorshift: 10.156600

Resultados para 1000000 gerações:
Tempo para rand(): 0.019002 segundos
Teste do chi-quadrado para rand(): 5.273520
Tempo para Xorshift: 0.000001 segundos
Teste do chi-quadrado para Xorshift: 5.273520

Resultados para 10000000 gerações:
Tempo para rand(): 0.190637 segundos
Teste do chi-quadrado para rand(): 4.211126
Tempo para Xorshift: 0.000001 segundos
Teste do chi-quadrado para Xorshift: 4.211126

Resultados para 100000000 gerações:
Tempo para rand(): 1.827503 segundos
Teste do chi-quadrado para rand(): 12.032469
Tempo para Xorshift: 0.000001 segundos
Teste do chi-quadrado para Xorshift: 12.032469
```

Tempo de Execução

Os tempos para o método *rand()* aumentam conforme o tamanho da amostra aumenta. Isso ocorre porque ele geralmente é implementado de maneira menos eficiente e pode envolver operações mais complexas para garantir aleatoriedade e uniformidade.

Os tempos para o *Xorshift* são extremamente baixos e praticamente constante (`0.000001` segundos em todas as medições). Isso ocorre devido à simplicidade do algoritmo, que envolve apenas operações de bit shifting e XOR.

Teste do Chi-Quadrado

Observamos valores semelhantes para o teste do chi-quadrado entre `rand()` e Xorshift em todas as amostras. Isso indica que ambos os métodos geram números pseudoaleatórios com uma distribuição uniforme satisfatória para os propósitos práticos considerados.

Análise Comparativa

Eficiência: O *Xorshift* é significativamente mais rápido que *rand()* em todas as amostras testadas. Isso torna o *Xorshift* uma escolha mais eficiente em aplicações onde a geração de números aleatórios é intensiva e o desempenho é crucial.

Qualidade Estatística: Ambos os métodos passam no teste do chi-quadrado, indicando que ambos são adequados para aplicações que exigem uniformidade na distribuição dos números gerados.