实验报告

- 选题：PRNG (NIST SP 800-22)

实验环境

1. Windows 11
2. WSL(windows subsystem for linux): Ubuntu 22.04.3 LTS

总览

设计一个伪随机数发生器，并使其能够通过 NIST SP 800-22 (*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*) 。

由于设计过程中版本迭代过多，故只取最新版本解释其原理，改进过程可参考git仓库中的各个分枝和相关记录。

所附文件有：

- `PRNG/`：git仓库（分枝`alpha`上为最新版本的代码）
- `final-reports/`：测试结果
- `pics/`：本Markdown/PDF文件引用的图片

测试结果

NIST测试集

阅读NIST测试集的官方手册得知基本使用方法后，在知网查阅相关论文，发现绝大多数随机数发生器相关的论文在证明其发生器的随机性优良时，都采用NIST测试集，并使用默认测试参数，且测试数据规模为一千（样本数量）乘以一百万（单个样本中的比特数）。故这里我们也使用一样的测试参数（以下将该参数称为**NIST测试集的“通用参数”**）。

测试结果在`final-reports/NIST-alpha-1000-test-finalAnalysisReport.txt`，可见所有测试均已通过。

由于使用NIST测试集时的一些疑惑，以及查阅相关资料时了解到了Diehard(er)、TestU01等测试集，这里也使用了它们进行测试、分析。

Dieharder

经查阅资料，在最新的一些PRNG设计、测试中，Dieharder测试集使用率不高而且不被作为判断随机性的主要依据，猜测可能是该测试集对随机性的要求不够严格。实际测试中，也发现该测试集确实比较容易通过，故最终**不采用**。

TestU01

TestU01中含有非常多可选的单项测试，以及三个最常用的测试集：

- SmallCrush：很好通过，只要设计没有明显缺陷或BUG基本都能过。
- Crush：可以比较全面地衡量随机性。
- BigCrush：是目前最严格的测试集之一，其所需的数据量也比较大。而且由于TestU01测试时是实时请求随机数，同一个PRNG，更大的数据量也就意味着更长的运行时间。

在改进本PRNG（版本迭代）的过程中，使用TestU01和"通用参数"下的NIST测试集对各个版本进行了测试，发现很多版本虽然能通过NIST，却无法通过TestU01的Crush（BigCrush下failure当然更多）。此后改进到可通过Crush后，也试过几次将循环轮数减到很少，然后便无法通过Crush，但仍能通过NIST；或者将尚未写完的半成品算法直接跑NIST，但也能通过。可见NIST测试集也不是非常严格。故后续改进**主要依据Crush和BigCrush。**

最新版本的测试结果：

- Crush：见`final-reports/TestU01-Crush-alpha.txt`，文件末尾有"All tests were passed"。
- BigCrush：见`final-reports/Z-TestU01-BigCrush-alpha.txt`，文件末尾处可见只有两个failure，且为同一类（sknuth_Gap test），故共一类failure。该设计既然能通过Crush中的此类测试，说明在这方面的伪随机性并不是完全无效，应该只是强度不够。

PRNG的实现/原理

代码总览

如下是PRNG部分的全部代码，这里每次generator被调用都会返回一个`uint32_t`的输出，该输出即为伪随机数。

其中：`state_t`为`uint64_t`；`_VALUE`是`state`初值，可以随便选。

```
#define LSHIFT_64(x, n) (((x)<<(n))^((x)>>(64-(n))))
#define RSHIFT_64(x, n) (((x)>>(n))^((x)<<(64-(n))))
#define G(state) (((state)>>N) ^ ((state)&Y_MASK))

static inline void update_state(void);

static state_t state = _VALUE;

out_t generator(void)
{
    update_state();
    return G(state);
}

static inline void update_state(void)
{
    state_t register s = 0;
    uint8_t register i = 0;
    uint8_t register bit = 0;
    while (i < TWO_N)
    {
        s += RSHIFT_64(state, i);
        bit ^= state >> i++;
        if (bit&1)
        {
            s ^= LSHIFT_64(state, i);
            bit ^= state >> i++;
            bit ^= state >> i++;
        }
    }
    state = ~s;
    state ^= i + bit;
}
```

总体思路

PRNG可以看作一个状态机，给定状态机初态`state`，则PRNG的工作过程可以被描述为：

```
while(1):
    out = g(state)
    state = update(state)
    yield out
```

这里不在输出函数`g`上下功夫，其设计很简单，直接取`state`的高32位和低32位相异或。

设计的核心思想是更新时在状态`state`的每个比特间建立复杂的联系，达到输出（新状态）对输入（原状态）的高度敏感和二者的伪无关性。

此外，还应注意：

1. 保证基本的均匀性，即零和一出现的概率相等
2. 避免短周期或收敛

具体设计

代码的实现方法和原思路在结构上稍有不同，请参考代码阅读。

1. 取bit遍历state中的每个比特。将state记为比特的数组s[64]（s[0]为最低位），定义：$bit[n] = \Sigma_{k=0}^{n} s[k]$（这里Σ的运算为异或）
2. 取i初值为0，i每次根据bit[i]为1或0，进行+=1或+=3，直至i大于等于64。
3. 按上述方法取得一系列的i（i构成一个序列）
    - i：`s += RSHIFT_64(state, i);`
    - i序列的空隙中：`s ^= LSHIFT_64(state, i);`
4. 最后执行`state = ~s`，由于x取反等于-x-1，这里相当于混入了一个常量，这是为了避免状态收敛为全0。
5. `state ^= i + bit`：将未被使用的最后一轮i和bit直接混入state。

- 这种设计可以让i序列对state高度敏感，随着n增大，i_n的效果越来越好。
- 之所以选择+=1或+=3是因为这样可以令i序列长约为64/2，因为易知全取或全不取是无效，取一半时效果最好。
- **改进空间**：这个设计有一个缺点是丢失了早期设计中的对称性，后续可以考虑更改结构取回对称性，看看能否提高伪随机效果。

运行速度

一般的PRNG设计中，时间复杂度皆为$O(n)$。

本PRNG最初的设计中，state大小为256字节，虽然状态空间越大理想周期越长，设计上也会有很多优势，但**过大的状态空间**会直接导致运算时**访存增加**，大大**减慢运行速度**。该版本通过TestU01 Crush花了八九个小时，本打算再进行BigCrush的测试，但奈何运行耗时太长，故运行一段时间后终止了。

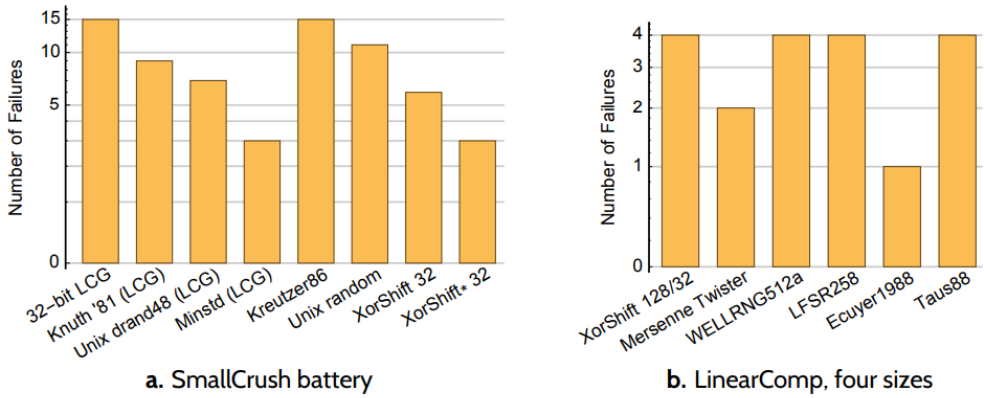
现版本可以在几毫秒内产生一百万比特，但运行Crush仍然需要一个多小时，BigCrush则需要十二个小时。

直到进行Crush和BigCrush测试时，才意识到速度的重要性。而且此后又查阅了一些已有的、较新较快的PRNG，例如Romu，已经把运算量压缩到了只剩几条指令。虽然在设计之初就已意识到PRNG的设计是一个工程问题，需要考虑到资源消耗，但未料到对运算量的控制已如此严格。这使得原本直接在比特间"建立复杂关系"的方法在速度上较劣势，除非采用巧妙的设计用原生指令一次性处理更多数据（**改进空间**）。

和其它PRNG的比较

- 主要参考 PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation
- SmallCrush

1. 很多常用PRNG甚至在SmallCrush时即出现了很多failure
2. 本程序failure：0



a. SmallCrush battery　　**b.** LinearComp, four sizes

Figure 1　Almost all generators in widespread use don't survive *very minimal* statistical scrutiny from TestU01 [29]. Even *one* failure indicates a problem.

- Crush + BigCrush

1. 一些更一般、更强的PRNG在Crush和BigCrush中的总failure数
2. 可见即使是一些精心设计的PRNG，其failure数也很多
3. 本程序最新版本总failure数：2

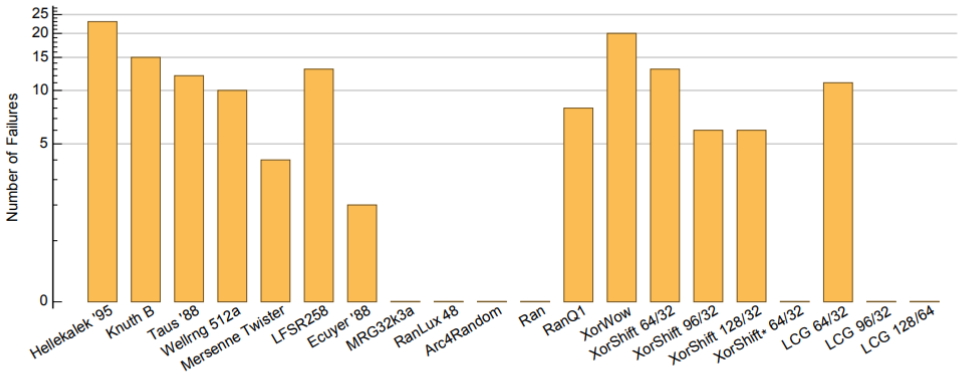


Figure 2　Combined test failures for Crush and BigCrush from TestU01 for a number of common generators. *Zero* failures are desired. Many well-respected generators fail. Despite their flaws, LCGs actually pass at larger bit sizes.

- 关于Next-bit test与CSPRNG
  - 一些算法，比如LCG，虽然在增大状态空间后能够通过很多随机性测试，但由于其内部结构过于简单，已经有已发表算法可以直接根据其过往输出预测将来输出，也就直接导致了无法通过Next-bit test；由于LCG采用线性同余变换，故由当前状态倒推过去状态为Trivial，所以不是CSPRNG
  - 本PRNG无证明保证前向/后向不可预测性，但该预测不是Trivial的
- 速度上，本程序运行速度一般（PCG论文里比较的是SmallCrush速度，这里没怎么测过SmallCrush所以没有数据，而且不同机器上运行，无法控制变量，即使测了也不能相比较，但通过指令数目和类型可以大致推测运行速度）
- 状态空间大小

1. 状态空间越小，短周期的设计缺陷就越容易暴露。于是就越需要算法充分利用状态空间，避免出现短环

2. 本程序状态空间大小：8-Byte（64-bit），尚未测试能否进一步压缩

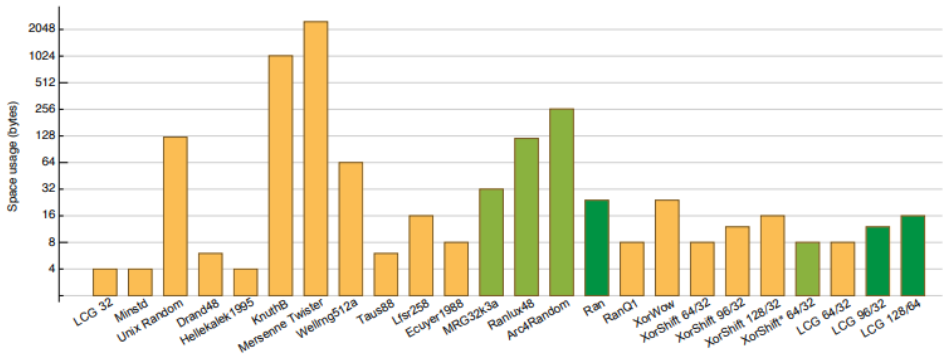


Figure 4 Space usage of different generators. As with Figure 3, the seven generators that pass statistic tests are shaded differently, but here the group is divided in two–four generators that pass TestU01 but have some disuniformity when used as 32-bit generators are shaded differently. Only a few of the generators that pass BigCrush are modestly sized.

3. 这里可见即使是理想的PRNG，也需要36-bit来通过BigCrush

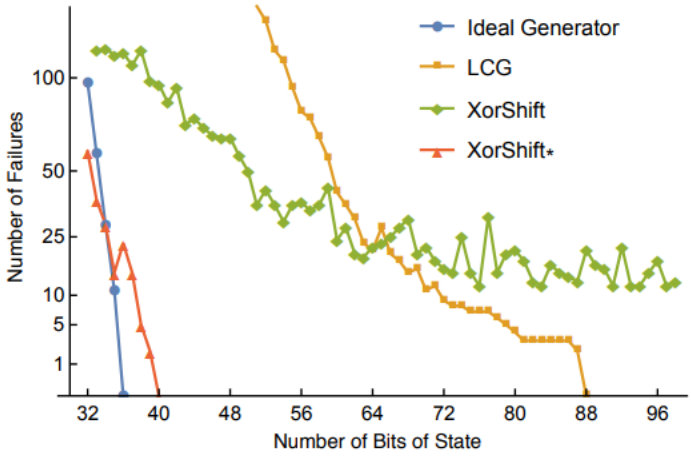


Figure 7 BigCrush + Crush test failures at different bit sizes (32-bit output).

总结

思而不学则殆，不能闭门造车