朱奕新
19307090029

Lab 2 实验报告

数据结构
2021 年 9 月 25 日

1 算法实现

1. 核心算法实现如下。（C++ 和 python 都实现了两个算法）
2. 请您查看”Readme.pdf” 了解更多关于本工程的信息，包括如何运行这些程序，以及其他重要的细节.
3. 请您进入相应语言的文件夹查看”Matrix.cpp” 和”main.py” 获得完整的代码，并了解代码中的其他功能和函数.
4. 请您查看 C++ 相关文件夹下的”generator.py” 了解随机测试数据是如何生成的.python 的随机测试数据则由主函数生成。
5. 所有代码均已经过测试，测试环境写在”Readme.pdf” 中。

算法 1: Ordinary Multi(C++ 实现)

```cpp
Matrix Matrix::operator*(const Matrix &ma) const
{
        Matrix ans;
        if (n != ma.m)
        {
                cout << "Matrix * invalid!" << endl;
                return ans;
        }
        ans.m = m;
        ans.n = ma.n;
        for (int i = 0; i < m; ++i)
                for (int j = 0; j < ma.n; ++j)
                        ans.con[i][j] = 0;
        for (int i = 0; i < m; ++i)
                for (int j = 0; j < ma.n; ++j)
                        for (int k = 0; k < n; ++k)
                                ans.con[i][j] += (con[i][k] * ma.con[k][j]);
        return ans;
}
```

算法 2: Ordinary Multi(Python 实现)

```python
def ordinaryMulti(x, y):
```

```
        ans = np.zeros((x.shape[0], y.shape[1]))
        if x.shape[1] != y.shape[0]:
                print("OrdinaryMulti error!\n")
                return ans
        for i in range(x.shape[0]):
                for j in range(y.shape[1]):
                        for k in range(x.shape[1]):
                                ans[i][j] += (x[i][k] * y[k][j])
        return ans
```

算法 3: Strassen Multi(C++ 实现)

```
Matrix strassenMulti(const Matrix &x, const Matrix &y)
{
        Matrix ans;
        ans.m = x.m;
        ans.n = y.n;
        if (x.n != y.m)
        {
                cout << "Matrix strenssen invalid!" << endl;
                return ans;
        }
        int all = 1;
        int p = max(x.n, x.m);
        int q = max(y.n, y.m);
        int w = max(p, q);
        while (all < w)
        {
                all = all * 2;
        }
        if (all == 1)
        {
                ans = x * y;
                return ans;
        }
        int mid = all / 2;
        Matrix a, b, c, d, e, f, g, h;
        a.copyFrom(x, 0, mid, 0, mid);
        b.copyFrom(x, 0, mid, mid, all);
        c.copyFrom(x, mid, all, 0, mid);
```

```
29          d.copyFrom(x, mid, all, mid, all);
30          e.copyFrom(y, 0, mid, 0, mid);
31          f.copyFrom(y, 0, mid, mid, all);
32          g.copyFrom(y, mid, all, 0, mid);
33          h.copyFrom(y, mid, all, mid, all);
34          Matrix P1 = strassenMulti(a, (f - h));
35          Matrix P2 = strassenMulti((a + b), h);
36          Matrix P3 = strassenMulti((c + d), e);
37          Matrix P4 = strassenMulti(d, (g - e));
38          Matrix P5 = strassenMulti((a + d), (e + h));
39          Matrix P6 = strassenMulti((b - d), (g + h));
40          Matrix P7 = strassenMulti((a - c), (e + f));
41          Matrix r = P5 + P4 - P2 + P6;
42          Matrix s = P1 + P2;
43          Matrix t = P3 + P4;
44          Matrix u = P5 + P1 - P3 - P7;
45          ans.copyTo(r, 0, mid, 0, mid);
46          ans.copyTo(s, 0, mid, mid, all);
47          ans.copyTo(t, mid, all, 0, mid);
48          ans.copyTo(u, mid, all, mid, all);
49          return ans;
50  }
```

算法 4: Strassen Multi(Python 实现)

```
1   def strassenMulti(x, y):
2           all = 1
3           p = max(x.shape[0], x.shape[1])
4           q = max(y.shape[0], y.shape[1])
5           w = max(p, q)
6           while(all < w):
7                   all = all*2
8           ans = np.zeros((all, all))
9           if x.shape[1] != y.shape[0]:
10                  print("OrdinaryMulti error!\n")
11          if all == 1:
12                  ans[0][0] = x[0][0] * y[0][0]
13          else:
14                  mid = all // 2
15                  xx = np.zeros((all, all))
```

```
16                  copy(xx, x)
17                  yy = np.zeros((all, all))
18                  copy(yy, y)
19                  a = xx[0:mid, 0:mid]
20                  b = xx[0:mid, mid:all]
21                  c = xx[mid:all, 0:mid]
22                  d = xx[mid:all, mid:all]
23                  e = yy[0:mid, 0:mid]
24                  f = yy[0:mid, mid:all]
25                  g = yy[mid:all, 0:mid]
26                  h = yy[mid:all, mid:all]
27                  P1 = strassenMulti(a, (f - h))
28                  P2 = strassenMulti((a + b), h)
29                  P3 = strassenMulti((c + d), e)
30                  P4 = strassenMulti(d, (g - e))
31                  P5 = strassenMulti((a + d), (e + h))
32                  P6 = strassenMulti((b - d), (g + h))
33                  P7 = strassenMulti((a - c), (e + f))
34                  r = P5 + P4 - P2 + P6
35                  s = P1 + P2
36                  t = P3 + P4
37                  u = P5 + P1 - P3 - P7
38                  ans = vstack((hstack((r, s)), hstack((t, u))))
39                  ans = ans[0:x.shape[0], 0:y.shape[1]]
40          return ans
```

2 实验情况

1. 本次实验分两组测试数据点进行数据。第一组数据的 N 均为 2 的幂次，从 1 测到 128，以此看出程序运行时间大的变化趋势。第二组数据的 N 测试了从 1 到 64 的全部整数，精细地反映出运行时间随 N 增大变化的情况。

2. 表中所有的运行时间数据，均为三次运行程序取平均值。时间测量方法请见实际代码。

3. 因为时间所限，此处只列出了 Python 实现的两种矩阵乘法的运行时间，以及根据实验数据绘制的统计图。但 C++ 代码也实现了测试程序运行时间等相关功能，不难据此得到相应实验数据，若您感兴趣请参见实际代码。

N	Ordinary Multi	Strassen Multi
1	997800	0
2	997000	997400
4	997700	1994300
8	3989400	6981300
16	12965400	32912500
32	24942600	123658500
64	183508700	812825400
128	1459180200	5644895500

表 1：Python 不同算法的运行时间粗略（纳秒）

N	Ordinary Multi	Strassen Multi
1	0	997300
2	0	999600
3	997500	1992500
4	1000100	1991800
5	997400	5985300
6	2991800	5982600
7	1995900	5982200
8	2992800	6980700
9	2992700	21764300
10	2985800	20943400
11	3020000	21941100
12	3990300	20943100
13	3991200	19944400
14	4986900	21941300
15	7979400	21940700
16	8976000	22940300
17	9972400	121673600
18	10970500	122679700

19	12957200	133643200
20	12964600	121674800
21	15826700	123668500
22	14959600	125665300
23	16953700	125910600
24	20936900	162565500
25	35905200	126660000
26	21941500	126663000
27	30914700	126663200
28	24931800	127657700
29	32912600	131305500
30	51862500	130648600
31	34906900	155583200
32	23935900	118684300
33	27924100	844956800
34	30729600	791077000
35	34399700	782890600
36	39892200	801854900
37	37902000	782900600
38	49867600	786893600
39	47872500	793894700
40	50862700	785896800
41	52858100	787247900
42	54716100	932500800
43	56848000	783919400
44	73785900	781923000
45	65811000	797333600
46	68792000	782134300
47	77775200	779918200
48	81775600	789906000
49	88287300	784899900

50	90102500	788879000
51	94714500	782902300
52	99733900	986360100
53	104740800	788868800
54	112491400	792875600
55	135481500	778915900
56	122673400	787890100
57	126661100	964051100
58	173962800	787892800
59	166553800	928771800
60	149597500	787891100
61	156299500	784078000
62	163669000	785312600
63	176522400	785915200
64	185706100	825810100

表 2：Python 不同算法的运行时间精细（纳秒）

请在下一页查看使用 Python 绘制的运行时间随 N 变化曲线图。图中附上了理论复杂度的函数曲线来逼近实际曲线。

首先是对应表 1 的较为粗略的散点测试结果图。

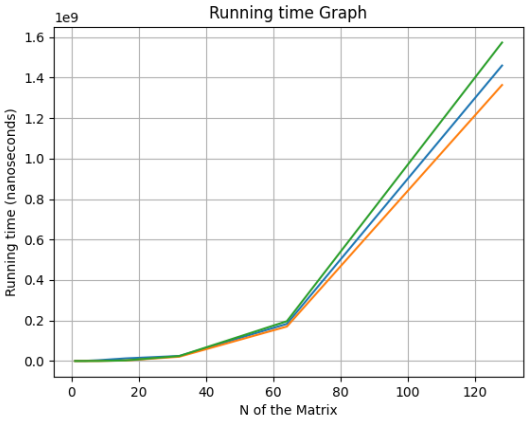

图 1: Ordinary Multi 粗略散点图

蓝色为实际实验数据，橙色为 $f(x) = 650x^3$, 绿色为 $f(x) = 750x^3$. 事实上 $f(x) = 700x^3$ 吻合较好。

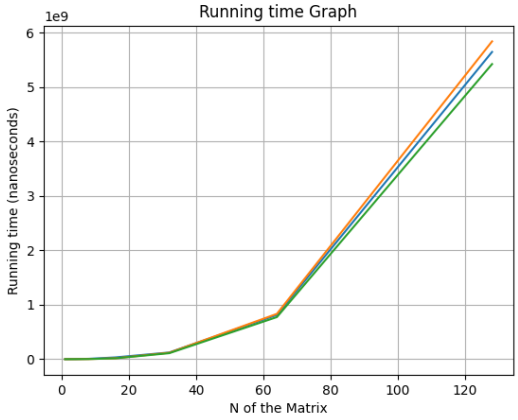

图 2: Strassen Multi 粗略散点图

蓝色为实际实验数据，绿色为 $f(x) = 6500x^{2.81}$, 橙色为 $f(x) = 7000x^{2.81}$.

接着是对应表 2 的较为精细的测试结果图。

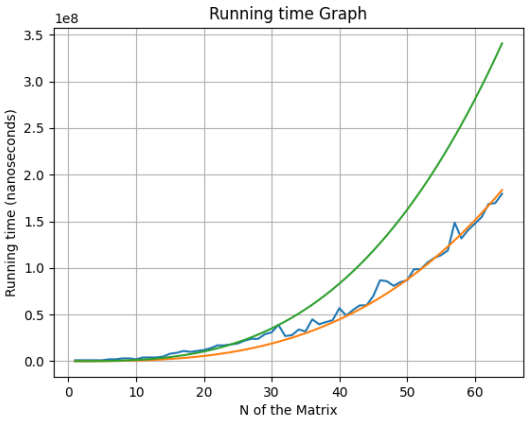

图 3: Ordinary Multi 精细点图

蓝色为实际实验数据，橙色为 $f(x) = 700x^3$, 绿色为 $f(x) = 1300x^3$. 橙色与蓝色后半吻合较好，绿色与蓝色前半段吻合较好。

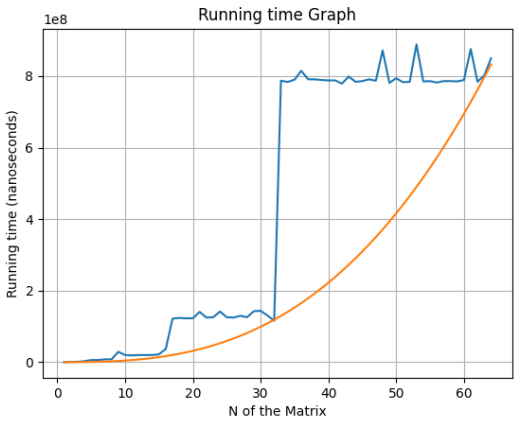

图 4: Strassen Multi 精细点图

蓝色为实际实验数据，橙色为 $f(x) = 7000x^{2.81}$。因为递归层数问题，实际数据呈现突变情况，但橙色可以大体反映趋势，并与粗略点图逼近结果接近。

3 理论分析和实际讨论

此处的分析讨论基于本实验中的 Python 和 C++ 代码。

3.1 一般算法的复杂度理论分析

让答案矩阵每一格置零需要花费 N^2 的复杂度。
而要计算每一格的新的值又需要花费 N^3 的复杂度。
这样一共是 $N^3 + N^2$, 即 $\Theta(N^3)$ 的复杂度。

3.2 Strassen 算法的复杂度理论分析

先来看递归的第一层，划分需要花费 N^2 的复杂度，把内容都保存下来。接着进入下一层递归，一共要进行 7 次递归的乘法，每一次的规模都是原来的一半。还要进行 18 次加减操作，赋值给 11 个小矩阵。这些一共是 $7.25N^2$ 的复杂度。拼接另外还需要花费 N^2 的复杂度。
所以可以列出算式 $T(n) = 7T(n/2) + 9.25N^2$
于是运用主定理的第一种情况估计复杂度,省去指数较小的无穷小项,复杂度应为 $\Theta(N^{\log_2 7})$ 即约 $\Theta(N^{2.81})$，比一般算法更优。

3.3 一般算法的复杂度实际情况

实验结果显示，一般算法的运行时间随 N 变化曲线图较为平滑且与复杂度为 N^3 的曲线吻合较好，说明实验与理论较为符合。
可能的多余开销：Python 代码中在生成零矩阵时使用了库函数，但总共只需调用一次。实验证明对结果影响较小。

3.4 Strassen 算法的复杂度实际情况

在 Python 实践中，虽然划分、拼接等操作都较简单实现，但实际上使用的是 numpy 库中预先定义的 matrix 类，实际上在较深递归中构造和析构了大量对象。这些操作耗费了大量时间，甚至有可能在内部改变了时间复杂度。
而在 C++ 的实现中使用的是自己定义的 Matrix 类，在比较深的递归中必然产生众多临时对象，而这些都又要求多次调用构造、析构函数，因此耗费了大量时间。

另外，为了完成 Strassen 算法，切割出的小矩阵需要补为方阵并补为相同大小，因此耗费了更多时间。

3.5 两种矩阵乘法算法的比较

根据理论分析，Srassen 算法的复杂度比一般算法更优，在 N 比较大时应比一般算法更快。但这与实验结果不符合。实际上随着 N 的扩大，Stressen 算法所花时间大大增加，且远远甩开了一般算法。原因推测如上一小节所述。