人工智能导论大作业一：算式谜

自 64　赵文亮　2016011452

2018 年 12 月 17 日

1　引言

算式迷（Cryptarithmetic）是一种用字母代替数字的数学游戏：给定一个算式，其中的操作数或结果中的全部或部分数字用字母来替代，目标是求解出每个字母对应的数字。图 1 中展示了两个算式谜的示例。这两个算式谜中单词本身的含义也满足算式，这种算式谜被称作双重算式谜。

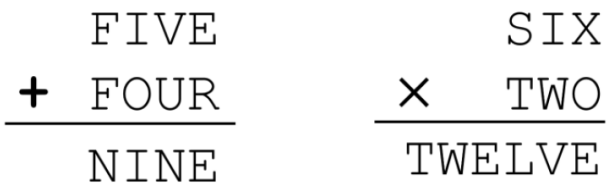


图 1: 算式谜示例

通常来说，算式谜满足以下条件：

- 每个算式中，每个字母代表 $0 \sim 9$ 中的一个数字；
- 不同字母代表不同的数字。

本文将使用搜索算法求解算式谜，并给出求解这种问题的统一框架。本文后续内容的结构如下：第 2 节中提出了两种模型（附加法和交换法）；第 3 节介绍了每种模型下不同搜索算法的实现方法和程序的软件架构；第 4 节中介绍了程序的编译方式和操作方法；第 5 中将给出本节中两个算式谜的求解结果，并通过求解多种类型的算式谜来分析比较本文提出的各种算法的效率；最后对本次大作业的实现过程进行总结。

2　模型建立

本节将建立本文算法的两个模型，两个模型的按照后继状态产生方式分为附加法和交换法。

2.1　符号约定

设输入算式中未知字符有 n 个，记作 $\boldsymbol{X} = \{X_1, X_2, \ldots, X_n\}$。由算式谜的特点，可知 $X_i \in \{1, 2, \ldots, 9\}, \forall i$，且 X_i 互不相同。设算式中的 操作符[1]为 *Operator*，操作数为 *Operands*， 算式结果为 *Answer*。例如，在图 1 中的第一个算式中，*Operator* 为加法，*Operands* 为“FIVE”和“FOUR”，*Answer* 为“NINE”。

[1]由于本文算法建立了求解该类问题的统一框架，具体的操作符并不影响算法的实现。本文程序中展示了四种操作符（加、减、乘、除）的求解，事实上还可以不断扩展下去。

设数字序列 $\boldsymbol{D}=\{D_1,D_2\ldots,D_n\}$ 为未知字符对应的数字，也就是我们的求解目标。设把操作数或结果中每一个字母替换成数字序列 $\boldsymbol{D}$ 中对应数字的代入操作为 $Substitute$。则对于最终的结果来说，必有

$$Operator(Substitute(Operands,\boldsymbol{D}))=Substitute(Answer,\boldsymbol{D}) \tag{1}$$

2.2 附加法

我们的求解目标为数字序列 $\boldsymbol{D}$，而由问题特点可知，$\boldsymbol{D}$ 的所有可能情况为 A_{10}^n 种，其中 A 表示排列数。当未知字符数目最多，即 $n=10$ 时，至多有 10 种情况。附加法的基本思路是从一个空序列出发，每次附加一个新数字，最终得到长度为 n 的数字序列，详细介绍如下：定义状态为 S 为当前层得到的字符序列，即 $\boldsymbol{S}_l=\{D_1,D_2,\ldots,D_l\}$，其中最初的空序列为 $\boldsymbol{S}_0=\varnothing$。附加法的状态转移通过在当前状态 $\boldsymbol{S}_l$ 后附加一个新数字 $\boldsymbol{D}_{l+1}$ 实现，即得到 $\boldsymbol{S}_{l+1}=\{D_1,D_2,\ldots,D_{l+1}\}$。可以将上述过程表达为以下递推关系：

$$\begin{cases}\boldsymbol{S}_0=\varnothing\\ \boldsymbol{S}_{l+1}=\{\boldsymbol{S}_l,D_{l+1}\}\quad 0\le l\le n-1\end{cases} \tag{2}$$

问题的关键在于如何选择下一次附加的数字 D_{l+1}。在满足 D_{l+1} 与 S_l 中的每个数字不同的情况下，可以有两种算法产生下一个状态：

- 按照一定的顺序（例如 $0\sim9$）遍历所有可能的 D_{l+1}，得到所有可能的下一状态 $\boldsymbol{S}_{l+1}$ 并依次扩展；
- 对所有可能的 D_{l+1}，得到对应的下一状态 S_{l+1}，使用某个启发函数计算每个 $\boldsymbol{S}_{l+1}$ 的代价，按照代价从小到大的顺序依次扩展。

上述两种思路对应的即为深度优先搜索和贪婪优先搜索，具体的算法实现将在第 3 节给出。

2.3 交换法

将输入字符序列 $\boldsymbol{X}$ 使用空字符将其补齐为 10 个，得到 $\boldsymbol{T}_0=\{X_1,X_2,\ldots,X_n,N_{n+1},\ldots,N_{10}\}$，其中 $N_i=null,n+1\le i\le 10$。例如，当 $n=5$ 时，各个字符对应的数字如表 1a 所示。从初始状态 T_0 出发，每次选择 $1\le i<j\le 10$，交换第 i 和 j 个字符，就会得到一组新的对应关系。表 1b 展示了从初始状态出发，使用 $i=1,j=8$ 时交换的结果。

表 1: 交换法状态定义示意（$n=5$）

字符	X_1	X_2	X_3	X_4	X_5	N_6	N_7	N_8	N_9	N_{10}
数字	0	1	2	3	4	5	6	7	8	9

(a) 初始状态

字符	N_8	X_2	X_3	X_4	X_5	N_6	N_7	X_1	N_9	N_{10}
数字	0	1	2	3	4	5	6	7	8	9

(b) 交换后 $(i=1,j=8)$

不难证明，对于每一个初始状态，至多经过 n 次交换就可以达到任何目标状态，故在搜索时可以将深度限制在 n 以内。这种定义状态的方式的好处在于，每一个状态都可能对应着原问题的一个解，所以可以将多种传统的搜索算法应用在上面。第 3 节中将使用深度优先、广度优先、贪婪最佳优先的搜索算法来实现。交换法的缺点在于，每次扩展节点的分支因子至多为 $C_{10}^2=45$，在某些情况下搜索时间较长。

3 算法实现

3.1 搜索算法详解

3.1.1 附加法

深度优先搜索 使用递归算法实现，主要流程如算法 1 所示。

算法 1: 深度优先搜索（附加法）

```
bool DFSAppend (S_l) /* DFS append recursive function */
    if n == l then
        if Operator(Substitute(Operands, S_l)) == Substitute(Answer, S_l) then
            AddResult(S_l) return true;
        else
            return false;
        end
    end
    foreach available D_{l+1} do
        S_{l+1} ← {S_l, D_{l+1}};
        if DFSAppend(S_{l+1}) then
            return true;
        end
    end
    return false;
end
```

最佳贪婪优先搜索 整体框架于深度优先搜索类似，不同的是扩展节点的顺序通过计算启发函数来确定。由于 $\boldsymbol{S}_l$ 中元素的个数仅为 l，为了定义启发函数，可以将其补零，得到

$$\hat{\boldsymbol{S}}_l = \{D_1, D_2, \ldots, D_l, Z_{l+1}, \ldots, Z_n\} \tag{3}$$

其中 $Z_i \equiv 0, l+1 \leq i \leq n$。设启发函数为：

$$h_1(\boldsymbol{S}_l) = |Operator(Substitute(Operands, \hat{\boldsymbol{S}}_l)) - Substitute(Answer, \hat{\boldsymbol{S}}_l)| \tag{4}$$

上述方法得到的 h_1 具有一定的启发性。一方面，当 S_l 为原问题的一个解时，必有 $\hat{S}_l = 0$；另一方面，h_1 可以在一定程度上刻画当前 S_l 与实际解的接近程度。贪婪最佳优先搜索的算法如算法 2 所示。此处有几点需要说明：

- 由于附加法定义的状态和状态转移方式保证了永远不会访问到重复节点（事实上为树搜索），故不需要 CloseList 来保存访问过的节点；
- 由于 $l < n$ 时的 $\boldsymbol{S}_l$ 只是中间状态而不可能是最终目标，在选择下一节点时应该以当前节点为基础扩展下去，而不是像图搜索的贪婪最佳优先算法一样从 OpenList 中选择最小代价的节点。此处的贪婪最佳优先搜索事实上是贪婪局部搜索。

算法 2: 贪婪最佳优先搜索（附加法）

```
bool GreedyAppend (S_l) /* Greedy append recursive function */
    if n == l then
        if h_1(S_l) == 0 then
            AddResult(S_l) return true;
        else
            return false;
        end
    end
    foreach available D_{l+1} do
        S_{l+1} ← {S_l, D_{l+1}};
        NextStateList.Add(S_{l+1});
        LossList.Add(h_1((S_{l+1}));
    end
    SortedLossList ← LossList.Sort;
    SortedIndex ← SortedLossList.getSortedIndex;
    foreach i in SortedIndex do
        if GreedyAppend (NextStateList[i]) then
            return true;
        end
    end
    return false;
end
```

3.1.2 交换法

由于交换法定义的每一个状态 $\boldsymbol{T}_i$ 都可能对应着一个原问题的解，在交换法的定义下，原问题是一个图搜索算法。在图搜索中，如果启发函数是一致的，则 A^* 算法是最优的。然而在本问题中，A^* 算法并不适用，原因有以下几点：

- 本问题中，并不关心经过多少步搜索才能找到解，所以不需要路径代价函数 $g(n)$；
- 图搜索 A^* 算法中的启发函数是一致的情况下，才能保证 A^* 算法是最优的。与路径规划类型的问题不同，在本问题中，我们事先根本无从得知目标节点是什么，所以也很难设计出一个合适的启发函数。

基于以上几点考虑，本问题中采用宽度优先搜索、深度优先搜索和贪婪最佳优先搜索。其中贪婪最佳优先搜索仍然是由局部贪婪算法，我们并不能保证它是最优的。对于每一个状态 $\boldsymbol{T}$，都可以将其转换为一个解 $\boldsymbol{D}$。例如，表 1b 中的状态对应的解为 $\boldsymbol{D} = \{7, 1, 2, 3, 4\}$。将这个转换过程记作函数 $Convert$，即 $\boldsymbol{D} = Convert(\boldsymbol{T})$。这样，计算启发函数的公式就可以写为：

$$h_2(\boldsymbol{T}) = Operator(Substitute(Operands, Convert(\boldsymbol{T}))) - Substitute(Answer, Convert(\boldsymbol{T})) \tag{5}$$

宽度优先搜索 算法实现的详细流程如算法 3 所示。

算法 3: 宽度搜索（交换法）

```
bool BFSSwap (T_0) /* BFS swap function                                    */
    BFS_Queue.Enqueue(T_0);
    ClossList.Add(T_0);
    if h_2(T_0) == 0 then
        return true;
    end
    while !BFS_Queue.Empty() do
        T_current ← BFS_Queue.Dequeue();
        foreach available 1 ≤ i < j ≤ 10 do
            T_next ← Swap(T_current, i, j);
            if !ClossList.Contains(T_next) then
                ClossList.Add(T_next);
                if h_2(T_next) == 0 then
                    return true;
                end
                BFS_Queue.Enqueue(T_next);
            end
        end
    end
    return false;
end
```

深度优先搜索 使用递归方法实现，具体参见算法 4 。需要指出的是，在调用递归算法前应该判断初始状态是否为原问题的解，并将初始状态加入到 CloseList 中。

算法 4: 深度优先搜索（交换法）

```
bool DFSSwap (T_current) /* DFS swap recursive function                     */
    foreach available 1 ≤ i < j ≤ 10 do
        T_next ← Swap(T_current, i, j);
        if !ClossList.Contains(T_next) then
            ClossList.Add(T_next);
            if h_2(T_next) == 0 then
                return true;
            end
            DFSSwap (T_next);
        end
    end
    return false;
end
```

贪婪最佳优先搜索 基于启发函数 h_2 选择局部最优解进行下一次的节点扩展，如算法 5 所示。

算法 5: 贪婪最佳优先搜索（交换法）

```
bool GreedySwap (T_current) /* Greedy swap recursive function */
    foreach available 1 ≤ i < j ≤ 10 do
        T_next ← Swap(T_current, i, j);
        if !ClossList.Contains(T_next) then
            ClossList.Add(T_next);
            if h2(T_next) == 0 then
                return true;
            end
            NextStateList.Add(T_next);
            LossList.Add(h2((T_next));
        end
    end
    SortedLossList ← LossList.Sort;
    SortedIndex ← SortedLossList.getSortedIndex;
    foreach i in SortedIndex do
        if GreedySwap (NextStateList[i]) then
            return true;
        end
    end
    return false;
end
```

3.2 软件架构

本次大作业使用 C# 语言完成，编程时充分体现了面向对象的思想。求解器的基类为 Solver，按照状态转移方式划分的附加法和交换法的求解器继承自 Solver 类，它们下面再根据搜索算法分为不同的子类。五种求解算法对应的类分别为 DFSAppendSolver、GreedyAppendSolver、BFSSwapSolver、DFSSwapSolver、GreedySwapSolver。和求解方法相关的类的继承关系如图 2 所示。

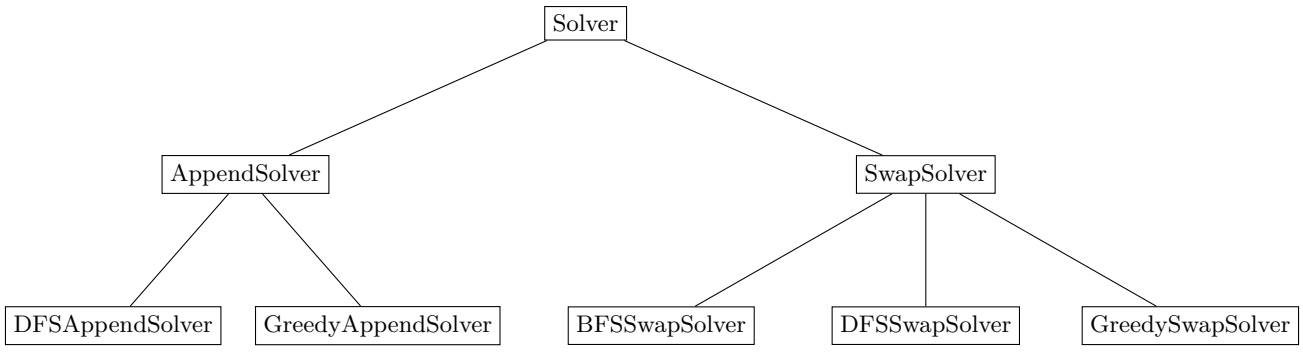


图 2: 各种求解方法类继承关系

这里需要详细介绍以下程序中求解算法的实现。首先，为了防止求解线程阻塞 UI 线程，所有的求解过程都在一个新的线程中完成。对于附加法的两种算法，直接通过调用对应的求解类中的方法即可求解。而对于交

换法中的三种算法，在实践中发现，由于分支因子过大，在某些情况下它们在短时间内找不到解。另一方面，在无解的情况下，它们会搜索很长时间才能够退出。因此，在调用交换法中的算法求解时，我为交换法求解器 SwapSolver 定义了最大求解步数（本程序中取 10^6 步），同时在另一个线程中使用基于附加法的贪婪优先算法（GreedyAppendSolver）进行求解。当 GreedyAppendSolver 发现无解时，立即抛出异常，显示无解信息；当 SwapSolver 求解超过最大步数时，也抛出异常，显示超时信息，并询问是否显示附加法求解的结果。

4 程序编译及运行方式

4.1 编译和运行环境

本程序使用 Microsoft .NET Framework 4.6.1 编译和运行。

4.2 运行方式

4.2.1 本地运行

Windows 系统双击 bin 目录下的 Cryptarithmetic.exe 即可运行。

4.2.2 在线安装（推荐）

访问网址http://ca.johnwilliams.online/download/index.html，点击安装按钮下载 setup.exe，双击即可安装。这种安装方式检测并安装系统必备组件。

4.3 操作方式

程序的初始界面如图 3a 所示。用户可以根据需要将调整界面大小，界面布局会自适应调整。程序界面分为菜单栏、工具栏、主界面。每个部分主要功能如下

- 菜单栏：包括状态转移方式和搜索算法的下拉菜单，可以自行修改；
- 工具栏：包括开始求解、查看问题、增加操作数、编辑栏、帮助按钮；
- 主界面：包括加法、减法、乘法、除法四个标签页，显示题目或答案。

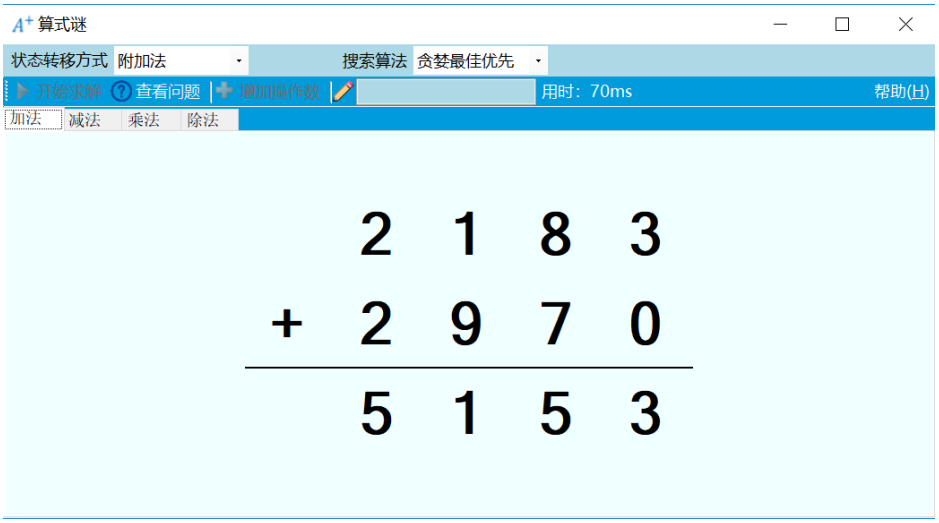
详细的操作指南如下：

界面模式 主界面分为问题模式和解答模式。问题模式下显示原始算式，解答模式下显示求解后的算式。例如，图 3a 为问题模式，图 3b 为解答模式。在问题模式下，查看问题按钮被禁用；在解答模式下，开始求解按钮以及和修改操作数相关的功能被禁用。

切换算式类型 点击对应的标签页（加、减、乘、除）进行切换。

切换算法 在菜单栏的下拉框中选择状态转移方式和求解算法（详见第 2 节和第 3 节）。

(a) 初始界面



(b) 求解结果

图 3: 程序运行截图

编辑操作数 在问题模式下可以对操作数进行编辑，包括操作数的增加、删除和修改。

- 增加操作数：点击工具栏中的*增加操作数*按钮，这时光标焦点移动到修改框，填写完毕后按下回车即可增加操作数；
- 修改操作数：鼠标移动到算式上时，对应的区域会变色，此时双击鼠标左键，工具栏的修改框中会显示该字符所在的操作数，修改完毕后按下回车；
- 删除操作数：右键双击待删除操作数的任意一个字符，或在修改该操作数时将其置为空。

开始求解 点击*开始求解*按钮启动求解过程。如果算式无解或输入不正确，程序会弹出相应的信息提示框。如果算式有解，则会切换到解答模式并显示求解结果，菜单栏会显示求解时间。

查看帮助 点击工具栏的*帮助*菜单，随时查看操作指南和在线文档。

5 结果分析

本节将使用多组输入，比较不同算法之间的效率，测试程序的通用性。表 2 中给出了四种不同类型算式（加、减、乘、除）的结果。其中加法和乘法即为第 1 中的两个算式谜。

表 2: 不同算法求解效率比较

求解算法		求解结果	求解时间
附加法	深度优先	$1254 + 1980 = 3234$	268ms
	贪婪最佳优先	$2183 + 2987 = 5153$	56ms
交换法	深度优先	$1687 + 1950 = 3637$	4983ms
	宽度优先	\	超时
	贪婪最佳优先	$1562 + 1970 = 3532$	3972ms

(a) FIVE + FOUR = NINE

求解算法		求解结果	求解时间
附加法	深度优先	$0264 - 132 = 132$	2ms
	贪婪最佳优先	$1468 - 734 = 734$	13ms
交换法	深度优先	$0714 - 357 = 357$	177ms
	宽度优先	$0714 - 357 = 357$	219ms
	贪婪最佳优先	$0918 - 459 = 459$	68ms

(b) FOUR − TWO = TWO

求解算法		求解结果	求解时间
附加法	深度优先	$923 \times 067 = 061841$	11417ms
	贪婪最佳优先	$986 \times 345 = 340170$	24ms
交换法	深度优先	\	超时
	宽度优先	\	超时
	贪婪最佳优先	$962 \times 057 = 054834$	52ms

(c) SIX × TWO = TWELVE

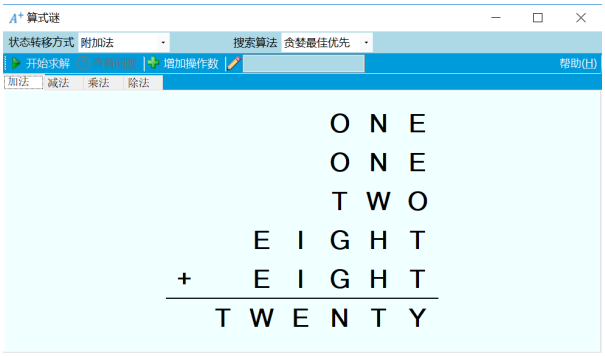
求解算法		求解结果	求解时间
附加法	深度优先	$026516 \div 947 = 028$	112ms
	贪婪最佳优先	$026516 \div 947 = 028$	200ms
交换法	深度优先	$210370 \div 965 = 218$	822ms
	宽度优先	$210370 \div 965 = 218$	1300ms
	贪婪最佳优先	\	超时

(d) TWELVE ÷ SIX = TWO

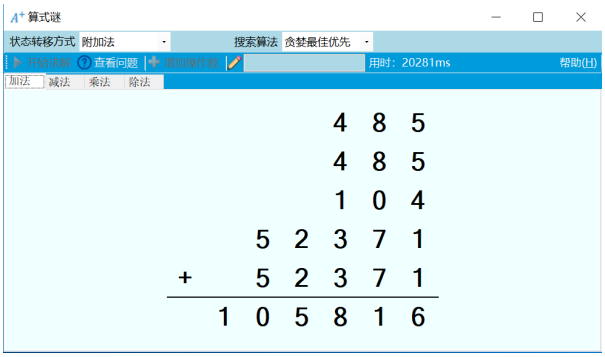
从中可以总结出以下结论：

1. 从状态转移方法上来看，附加法能保证每次都能较快求解，而交换法在某些情况下会发生超时。这是由于交换法的分支因子较大，在算法设计中已经考虑到；

2. 从求解算法上来看，平均意义上来讲贪婪最佳优先算法的求解效率较高，这也体现了启发函数的作用；



(a) 输入算式



(b) 求解结果

图 4: ONE + ONE + TWO + EIGHT + EIGHT = TWENTY 求解示例

3. 贪婪最佳优先并不是在所有情况下都是最优的，有时会劣于深度或广度优先。这是由于该问题本身很难找到一个一致的启发函数，并不能保证贪婪最佳优先算法的最优性。

借下来我们验证以下算法的通用性，给定输入：ONE + ONE + TWO + EIGHT + EIGHT = TWENTY，图 4 中展示了输入算式和求解结果。本次求解耗时 20281ms。从中可见，算法可以适应多个操作数的情况，通用性较强。

6 总结

本次作业从选题到算法设计，再到代码编写，我都在不断地挑战自己，整个过程中也由很多收获。选题时，我也在两个题目之间犹豫过。迷宫的问题比较简单，能够很容易设计出代价函数和启发函数。但是我觉得算式谜的问题既有趣又有挑战性，所以最终我决定选择算式谜问题来求解。由于算式谜问题的特殊性，我们并不能提前指导目标节点的位置，从而难以设计启发函数。我只能使用当前代入结果和真实结果的差值来近似，最终也得到了较好的效果。代码和 GUI 编写上，我没有使用熟悉的 C++ 和 Qt 库，而是尝试使用 C# 和 Winform。在我之前的经历中，我发现学习一门语言最好的方法就是用这门语言完成一个项目，于是我就想通过这个机会学习一下 C#。C# 可以说是一个完全面向对象的语言，许多功能都有封装好的方法来调用。

在完成报告的过程中，我锻炼了自己的表达能力。一个算法在自己的脑海中或是在代码中实现起来可能并不复杂，但是要想落在纸面上，最好还是要使用规范的数学表达。曾经听一位老师说过，论文中复杂的概念尽量少写文字，多用公式、表格和插图来说明。这次报告中我也尽量遵循这一原则。

此外，我尝试了将程序发布在我自己的服务器上。这一过程中我又锻炼了计网课上的搭建网站的相关内容。总的来说，本次大作业真的是收获满满。