

软件工程实验报告

Lab4 编程实现、分析和测试
网络空间安全学院 信息安全专业
2112492 刘修铭 1027

1 编程实现

为了对后面的编程规范等留出优化空间，此处展示最初未进行优化的代码，但是保证结果的正确性。

本次实验使用 python 进行实现。

首先创建了一个电路门类，为其进行一定的初始化。

```
1 class Gate:
2     def __init__(self, func_str: str, input_lst: list):
3         self.func = func_str
4         self.input = input_lst
5         self.val = None
6         self.loop = False
```

接着按照题目要求，创建了一个 calculate 函数，用于完成门电路的计算。由于题目限制了门电路的种类，基于此插入了错误处理。

```
1 def calculate(self, input_lst):
2     try:
3         operations = {
4             "NOT": lambda x: int(not x[0]),
5             "AND": lambda x: int(all(x)),
6             "OR": lambda x: int(any(x)),
7             "XOR": lambda x: int(sum(x) % 2),
8             "NAND": lambda x: int(not all(x)),
9             "NOR": lambda x: int(not any(x))
10        }
11        self.val = operations[self.func](input_lst)
12    except Exception as e:
13        print(f"Error occurred while calculating {self.func}: {e}")
```

在本次中，逻辑电路可以看作是一个有向图，其中门（或逻辑门）表示图中的节点，门之间的连接表示有向边，输入端口表示图的起始节点，输出端口表示图的终止节点。在有向图中，如果存在循环，则一定存在一条路径从某个节点出发，经过若干条边回到该节点，这种路径称为循环路径。在逻辑电路中，循环依赖即表示存在循环路径。

通过在逻辑电路中进行深度优先搜索（DFS），可以在图中遍历节点，并在遍历过程中标记已经访问过的节点。如果在遍历过程中发现某个节点已经被标记为已访问过，则说明存在循环依赖，因为在有向图中，节点的访问顺序是由边的方向所决定的，如果已经访问过某个节点，则再次访问该节点必定会回到之前已访问过的节点，形成循环。

solve 函数对门进行深度优先搜索，并使用 loop 属性来标记已经访问过的节点。在遍历过程中，如果发现某个节点已经被访问过，则将该节点的 loop 属性设置为 `True`，表示存在循环依赖，直接返回 0，停止继续计算该节点的值。这样可以确保在计算逻辑电路的值时，不会陷入循环依赖导致的无限递归或死循环，保证了算法的正确性和鲁棒性。

```

1 def solve(self, gates, inputs, visited):
2     try:
3         if self.loop:
4             return 0
5         tmp_lst = []
6         if self.val is None:
7             for name in self.input:
8                 if name in visited:
9                     self.loop = True
10                    return 0
11                if name[0] == "I":
12                    tmp_lst.append(inputs[int(name[1:])])
13                elif name in gates:
14                    if gates[name].val is None:
15                        visited.add(name)
16                        gates[name].solve(gates, inputs, visited)
17                        visited.remove(name)
18                    if gates[name].val is None:
19                        self.loop = True
20                        return 0
21                    tmp_lst.append(gates[name].val)
22                self.calculate(tmp_lst)
23            return self.val
24        except Exception as e:
25            print(f"Error occurred while solving: {e}")
26            return None

```

然后编写了一个函数用于按照规定格式读入数据，并且设计数据结构进行存储。注意到，题目中提到“注意 O 序列不一定是递增的，即要求输出的器件可能以任意顺序出现。”此处对于读入的数据进行处理时，按照题目中的设定，将其输入格式均保存。此处采用 dict 数据类型保存门电路的输入等，采用键值对的形式进行处理，避免了下标越界等问题。

```

1 def get_data():
2     Q = int(input())
3     test_cases = []
4     for _ in range(Q):
5         gate_dict = {}
6         M, N = input().split()
7         M = int(M)
8         N = int(N)
9         for i in range(N):
10            inputs = input().split()
11            FUNC = inputs[0]
12            L = inputs[2:]
13            gate_dict[f"O{i + 1}"] = Gate(FUNC, L)
14        S = int(input())
15        input_list = []
16        for _ in range(S):
17            I = input().split()
18            inputs = {index + 1: int(val) for index, val in enumerate(I)}
19            input_list.append(inputs)
20        output_list = []
21        for _ in range(S):

```

```

22         S_inputs = input().split()
23         output_lst = [int(val) for val in S_inputs[1:]]
24         output_list.append(output_lst)
25         test_cases.append((gate_dict, input_list, output_list))
26     return test_cases

```

simulate_circuit 函数基于前面的输入调用函数进行直接计算。

1. 使用 `zip` 函数将 `input_list` 和 `output_list` 中的元素一一对应起来，形成一个迭代器，在每次迭代中得到一组输入值 `inputs` 和对应的期望输出值 `expected_output`。
2. 在每组输入值 `inputs` 上模拟逻辑电路的行为，通过 `gate_dict` 中的门逐一计算其输出值，并与期望输出值进行比较。
3. 在模拟逻辑电路的过程中，如果发现存在环，则输出 "LOOP" 并跳过当前组输入的模拟，进行下一组输入的模拟。
4. 如果不存在环，则将计算得到的输出值依次输出。

```

1 def simulate_circuit(gate_dict, input_list, output_list):
2     for inputs, expected_output in zip(input_list, output_list):
3         is_loop = False
4         for gate in gate_dict.values():
5             gate.clear()
6             for j in range(1, len(gate_dict) + 1):
7                 gate_dict[f"O{j}"].solve(gate_dict, inputs, set())
8                 if gate_dict[f"O{j}"].loop:
9                     is_loop = True
10                    break
11            if is_loop:
12                print("LOOP")
13                continue
14            for output_index in expected_output:
15                print(gate_dict[f"O{output_index}"].val, end=' ')
16            print("")

```

main 主函数中，则调用前面的函数进行综合计算。

运行给定测试样例，可以看到，输出的结果与预期输出相同，说明编程正确。

```

lxmliu2002@MacBook-Pro: ~/Desktop/Software_Engineering/lab4
ktop/Software_Engineering/lab4/code/Logical_Gates.py
1
3 5
XOR 2 I1 I2
XOR 2 O1 I3
AND 2 O1 I3
AND 2 I1 I2
OR 2 O3 O4
4
0 1 1
1 0 1
1 1 1
0 0 0
2 5 2
2 5 2
2 5 2
2 5 2
1 0
1 0
1 1
0 0

```

2 编程规范

参考的哪个规范，如何检查是否遵守编程规范的？

本次编程主要参考 [Google 开源项目风格指南](#)。

在进行编程规范检查时，使用 ChatGPT 进行检查，得到如下的输出结果。

ChatGPT

你的代码大体上符合谷歌编程规范，但还有一些地方可以改进：

1. **注释文档化**：添加函数和类的文档字符串，说明其作用、参数和返回值，以提高代码的可读性和可维护性。
2. **异常处理**：在 `simulate_circuit` 函数中，如果在解决逻辑电路时发生异常，只是简单地打印错误信息并继续处理下一组测试数据。这可能会掩盖程序中的潜在错误。建议将异常抛出，并在调用方 (`__main__`) 处捕获异常并进行处理，以便更好地追踪和调试错误。
3. **变量命名**：尽量避免使用单个字母作为变量名，因为它们通常缺乏描述性。尽可能使用具有描述性的名称，以提高代码的可读性。
4. **打印函数**：在代码中使用的 `print` 语句用于调试目的，但在生产代码中不建议这样做。如果需要记录日志或调试信息，请考虑使用 Python 的标准 `logging` 模块。
5. **函数和类之间的分隔线**：根据 PEP 8，最好在函数和类之间加上两个空行，以提高代码的可读性。
6. **异常类型**：在捕获异常时，最好明确指定要捕获的异常类型，而不是捕获所有异常。这样有助于更好地处理特定类型的异常，并且可以避免捕获意外的异常。
7. **变量命名风格**：变量命名使用的是 `snake_case`，这是符合 Python 的命名约定的。但是，类名使用的是 `PascalCase`，更符合常见的 Python 类命名约定。建议统一使用 `snake_case`。

针对其给出的改进说明，参考 [Google 开源项目风格指南](#)，将代码作如下修改。

```
1 class Gate:
2     """
3     Represents a logic gate.
4
5     Attributes:
6         func (str): The logic function of the gate.
7         inputs (list): The list of input ports.
8         val (int): The output value of the gate.
9         loop (bool): Flag indicating if there is a loop during solving.
10    """
11
12    def __init__(self, func_str: str, input_lst: list):
13        """
14        Initializes a Gate instance.
15
16        Args:
17            func_str (str): The logic function string.
18            input_lst (list): The list of input ports.
19        """
20        self.func = func_str
21        self.inputs = input_lst
22        self.val = None
23        self.loop = False
24
```

```

25     def calculate(self, input_lst):
26         """
27         Calculates the output value of the gate.
28
29         Args:
30             input_lst (list): The list of input values.
31         """
32         try:
33             operations = {
34                 "NOT": lambda x: int(not x[0]),
35                 "AND": lambda x: int(all(x)),
36                 "OR": lambda x: int(any(x)),
37                 "XOR": lambda x: int(sum(x) % 2),
38                 "NAND": lambda x: int(not all(x)),
39                 "NOR": lambda x: int(not any(x))
40             }
41             self.val = operations[self.func](input_lst)
42         except Exception as e:
43             raise RuntimeError(f"Error occurred while calculating {self.func}: {e}")
44
45     def solve(self, gates, inputs, visited):
46         """
47         Solves the gate.
48
49         Args:
50             gates (dict): Dictionary containing gate instances.
51             inputs (dict): Dictionary containing input values.
52             visited (set): Set containing visited gate names.
53         """
54         try:
55             if self.loop:
56                 return 0
57             tmp_lst = []
58             if self.val is None:
59                 for name in self.inputs:
60                     if name in visited:
61                         self.loop = True
62                         return 0
63                     if name[0] == "I":
64                         tmp_lst.append(inputs[int(name[1:])])
65                     elif name in gates:
66                         if gates[name].val is None:
67                             visited.add(name)
68                             gates[name].solve(gates, inputs, visited)
69                             visited.remove(name)
70                         if gates[name].val is None:
71                             self.loop = True
72                             return 0
73                         tmp_lst.append(gates[name].val)
74             self.calculate(tmp_lst)
75             return self.val
76         except Exception as e:

```

```

77         raise RuntimeError(f"Error occurred while solving: {e}")
78
79     def clear(self):
80         """Clears the output value of the gate."""
81         self.val = None
82
83     def get_data():
84         """
85         Retrieves test case data from input.
86
87         Returns:
88             list: List of test cases.
89         """
90         try:
91             Q = int(input())
92             test_cases = []
93             for _ in range(Q):
94                 gate_dict = {}
95                 M, N = input().split()
96                 M = int(M)
97                 N = int(N)
98                 for i in range(N):
99                     inputs = input().split()
100                     FUNC = inputs[0]
101                     L = inputs[2:]
102                     gate_dict[f"0{i + 1}"] = Gate(FUNC, L)
103                 S = int(input())
104                 input_list = []
105                 for _ in range(S):
106                     I = input().split()
107                     inputs = {index + 1: int(val) for index, val in enumerate(I)}
108                     input_list.append(inputs)
109                 output_list = []
110                 for _ in range(S):
111                     S_inputs = input().split()
112                     output_lst = [int(val) for val in S_inputs[1:]]
113                     output_list.append(output_lst)
114                 test_cases.append((gate_dict, input_list, output_list))
115             return test_cases
116         except Exception as e:
117             raise RuntimeError(f"Error occurred while getting data: {e}")
118
119     def simulate_circuit(gate_dict, input_list, output_list):
120         """
121         Simulates the logic circuit.
122
123         Args:
124             gate_dict (dict): Dictionary containing gate instances.
125             input_list (list): List of input values.
126             output_list (list): List of expected output values.
127         """
128         try:

```

```

129     for inputs, expected_output in zip(input_list, output_list):
130         is_loop = False
131         for gate in gate_dict.values():
132             gate.clear()
133             for j in range(1, len(gate_dict) + 1):
134                 gate_dict[f"O{j}"].solve(gate_dict, inputs, set())
135                 if gate_dict[f"O{j}"].loop:
136                     is_loop = True
137                     break
138             if is_loop:
139                 print("LOOP")
140                 continue
141             for output_index in expected_output:
142                 print(gate_dict[f"O{output_index}"].val, end=' ')
143             print("")
144     except Exception as e:
145         raise RuntimeError(f"Error occurred while simulating circuit: {e}")
146
147 if __name__ == "__main__":
148     try:
149         test_cases = get_data()
150         for gate_dict, input_list, output_list in test_cases:
151             simulate_circuit(gate_dict, input_list, output_list)
152     except Exception as e:
153         print(f"Error occurred: {e}")

```

使用 [pylint](#) 进行原始代码的静态分析，得到如下的结果。

```

lxmliu2002@MacBook-Pro ~/Desktop/Software_Engineering/lab4/code [ ] main ± pylint ./Logical_Gates.py
***** Module Logical_Gates
Logical_Gates.py:1:0: C0114: Missing module docstring (missing-module-docstring)
Logical_Gates.py:1:0: C0103: Module name "Logical_Gates" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:1:0: C0115: Missing class docstring (missing-class-docstring)
Logical_Gates.py:8:4: C0116: Missing function or method docstring (missing-function-docstring)
Logical_Gates.py:19:8: W0621: Redefining name 'e' from outer scope (line 101) (redefined-outer-name)
Logical_Gates.py:19:15: W0718: Catching too general exception Exception (broad-exception-caught)
Logical_Gates.py:22:4: C0116: Missing function or method docstring (missing-function-docstring)
Logical_Gates.py:45:8: W0621: Redefining name 'e' from outer scope (line 101) (redefined-outer-name)
Logical_Gates.py:45:15: W0718: Catching too general exception Exception (broad-exception-caught)
Logical_Gates.py:49:4: C0116: Missing function or method docstring (missing-function-docstring)
Logical_Gates.py:52:0: C0116: Missing function or method docstring (missing-function-docstring)
Logical_Gates.py:54:4: W0621: Redefining name 'test_cases' from outer scope (line 98) (redefined-outer-name)
Logical_Gates.py:56:8: W0621: Redefining name 'gate_dict' from outer scope (line 99) (redefined-outer-name)
Logical_Gates.py:66:8: W0621: Redefining name 'input_list' from outer scope (line 99) (redefined-outer-name)
Logical_Gates.py:71:8: W0621: Redefining name 'output_list' from outer scope (line 99) (redefined-outer-name)
Logical_Gates.py:53:4: C0103: Variable name "Q" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:57:8: C0103: Variable name "M" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:57:8: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:58:11: C0103: Variable name "M" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:59:8: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:62:12: C0103: Variable name "FUNC" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:63:12: C0103: Variable name "L" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:65:8: C0103: Variable name "S" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:68:12: C0103: Variable name "I" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:73:12: C0103: Variable name "S_inputs" doesn't conform to snake_case naming style (invalid-name)
Logical_Gates.py:79:0: C0116: Missing function or method docstring (missing-function-docstring)
Logical_Gates.py:79:21: W0621: Redefining name 'gate_dict' from outer scope (line 99) (redefined-outer-name)
Logical_Gates.py:79:32: W0621: Redefining name 'input_list' from outer scope (line 99) (redefined-outer-name)
Logical_Gates.py:79:44: W0621: Redefining name 'output_list' from outer scope (line 99) (redefined-outer-name)
Logical_Gates.py:101:11: W0718: Catching too general exception Exception (broad-exception-caught)

Your code has been rated at 6.63/10

```

针对于这些问题，将其进行一定的修改，得到如下修改后的代码。

```

1  """
2  Module: better_logical_gates.py
3  Description: Contains classes and functions for simulating logical gates
4  """

```

```

5
6 class Gate:
7     """
8     Represents a logic gate.
9
10    Attributes:
11        func (str): The logic function of the gate.
12        inputs (list): The list of input ports.
13        val (int): The output value of the gate.
14        loop (bool): Flag indicating if there is a loop during solving.
15    """
16
17    def __init__(self, func_str: str, input_lst: list):
18        """
19        Initializes a Gate instance.
20
21        Args:
22            func_str (str): The logic function string.
23            input_lst (list): The list of input ports.
24        """
25        self.func = func_str
26        self.inputs = input_lst
27        self.val = None
28        self.loop = False
29
30    def calculate(self, input_lst):
31        """
32        Calculates the output value of the gate.
33
34        Args:
35            input_lst (list): The list of input values.
36        """
37        operations = {
38            "NOT": lambda x: int(not x[0]),
39            "AND": lambda x: int(all(x)),
40            "OR": lambda x: int(any(x)),
41            "XOR": lambda x: int(sum(x) % 2),
42            "NAND": lambda x: int(not all(x)),
43            "NOR": lambda x: int(not any(x))
44        }
45        assert self.func in operations, f"Invalid logic function: {self.func}"
46        self.val = operations[self.func](input_lst)
47
48    def solve(self, gates, inputs, visited):
49        """
50        Solves the gate.
51
52        Args:
53            gates (dict): Dictionary containing gate instances.
54            inputs (dict): Dictionary containing input values.
55            visited (set): Set containing visited gate names.
56        """

```



```

57         if self.loop:
58             return 0
59         tmp_lst = []
60         if self.val is None:
61             for name in self.inputs:
62                 assert name[0] == "I" or name in gates, f"Invalid input: {name}"
63                 if name[0] == "I":
64                     assert int(name[1:]) in inputs, f"Input not provided: {name}"
65                     tmp_lst.append(inputs[int(name[1:])])
66                 else:
67                     if gates[name].val is None:
68                         assert name not in visited, f"Loop detected: {name}"
69                         visited.add(name)
70                         gates[name].solve(gates, inputs, visited)
71                         visited.remove(name)
72                         assert gates[name].val is not None, f"Gate not solved: {name}"
73                     tmp_lst.append(gates[name].val)
74             self.calculate(tmp_lst)
75         return self.val
76
77     def clear(self):
78         """Clears the output value of the gate."""
79         self.val = None
80
81
82 def get_data():
83     """
84     Retrieves test case data from input.
85
86     Returns:
87         list: List of test cases.
88     """
89     q = int(input())
90     test_cases = []
91     for _ in range(q):
92         gate_dict = {}
93         _, n = map(int, input().split())
94         for i in range(n):
95             inputs = input().split()
96             func = inputs[0]
97             l = inputs[2:]
98             gate_dict[f"O{i + 1}"] = Gate(func, l)
99     s = int(input())
100     input_list = []
101     for _ in range(s):
102         i = input().split()
103         inputs = {index + 1: int(val) for index, val in enumerate(i)}
104         input_list.append(inputs)
105     output_list = []
106     for _ in range(s):
107         s_inputs = input().split()
108         output_lst = [int(val) for val in s_inputs[1:]]

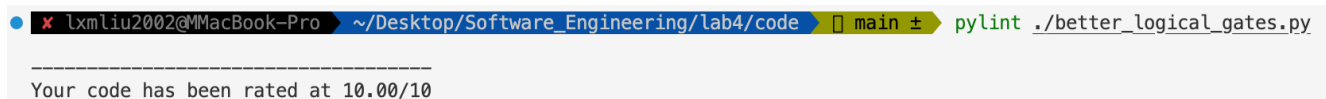
```

```

109         output_list.append(output_lst)
110         test_cases.append((gate_dict, input_list, output_list))
111     return test_cases
112
113
114 def simulate_circuit(gate_dict, input_list, output_list):
115     """
116     Simulates the logic circuit.
117
118     Args:
119         gate_dict (dict): Dictionary containing gate instances.
120         input_list (list): List of input values.
121         output_list (list): List of expected output values.
122     """
123     for inputs, expected_output in zip(input_list, output_list):
124         is_loop = False
125         for gate in gate_dict.values():
126             gate.clear()
127             for j in range(1, len(gate_dict) + 1):
128                 gate_dict[f"O{j}"].solve(gate_dict, inputs, set())
129                 if gate_dict[f"O{j}"].loop:
130                     is_loop = True
131                     break
132             if is_loop:
133                 print("LOOP")
134                 continue
135             for output_index in expected_output:
136                 print(gate_dict[f"O{output_index}"].val, end=' ')
137             print("")
138
139
140 if __name__ == "__main__":
141     cases = get_data()
142     for gate_dict, input_list, output_list in cases:
143         simulate_circuit(gate_dict, input_list, output_list)
144

```

对其进行检查，可以看到，所有的问题都已经解决。



```

lxmliu2002@MacBook-Pro ~/Desktop/Software_Engineering/lab4/code main ± pylint ./better_logical_gates.py
-----
Your code has been rated at 10.00/10

```

3 可扩展性

如何提高代码的可扩展性？

对于此编程任务，主要的扩展方向为门电路种类的增加。

本次实验主要有两种拓展思路。

首先，在原始代码中，门电路的实现部分主要在 `calculate` 函数中。即按照门电路要求添加对应的 FUNC 与其计算过程。因此，如果需要扩展，按照要扩展的门电路的要求，对其进行功能与名称的添加即可。

第二种思路则是对现有的代码进行修改，将逻辑门的 `calculate` 函数和 `solve` 函数作为接口，并为每种逻辑门创建一个独立的子类。这样做可以轻松地添加新的逻辑门类型，并且使得代码更易于理解和维护。在此给出一个代码示例，不做具体实现。

```
1  from abc import ABC, abstractmethod
2
3  class LogicGate(ABC):
4      """
5      Abstract base class for logic gates.
6      """
7      @abstractmethod
8      def calculate(self, input_lst):
9          """
10         Calculates the output value of the gate.
11
12         Args:
13             input_lst (list): The list of input values.
14         """
15         pass
16
17     @abstractmethod
18     def solve(self, gates, inputs, visited):
19         """
20         Solves the gate.
21
22         Args:
23             gates (dict): Dictionary containing gate instances.
24             inputs (dict): Dictionary containing input values.
25             visited (set): Set containing visited gate names.
26         """
27         pass
28
29     @abstractmethod
30     def clear(self):
31         """
32         Clears the output value of the gate.
33         """
34         pass
35
36 class NOTGate(LogicGate):
37     """
38     Represents a NOT logic gate.
39     """
40     def __init__(self, input_port):
41         self.input = input_port
42         self.val = None
43         self.loop = False
44
45     def calculate(self, input_lst):
```

```

46         assert len(input_lst) == 1, "NOT gate expects exactly 1 input"
47         self.val = int(not input_lst[0])
48
49     def solve(self, gates, inputs, visited):
50         if self.loop:
51             return 0
52         if self.val is None:
53             assert self.input in inputs, f"Input not provided: {self.input}"
54             self.calculate([inputs[self.input]])
55             return self.val
56
57     def clear(self):
58         self.val = None
59
60 # 添加其他逻辑门的类似实现...
61
62 def create_gate(func_str, input_lst):
63     """
64     Factory function to create logic gates based on function string.
65
66     Args:
67         func_str (str): The logic function string.
68         input_lst (list): The list of input ports.
69
70     Returns:
71         LogicGate: An instance of the appropriate logic gate class.
72     """
73     if func_str == "NOT":
74         assert len(input_lst) == 1, "NOT gate expects exactly 1 input"
75         return NOTGate(input_lst[0])
76 # 添加其他逻辑门类型的判断和实例化...
77

```

4 错误与异常处理

在代码中，主要使用 try-except 进行处理。

1. Gate 类的方法（`calculate`、`solve`、`clear`）：每个方法都包裹在 `try-except` 块中。如果在计算、求解或清除过程中发生异常（比如使用未定义的门电路，如 add），会引发一个 `RuntimeError`，并提供适当的错误消息。
2. `get_data` 函数：此函数从输入中获取测试用例数据。将主要逻辑包装在 `try-except` 块中。如果在检索过程中发生异常，会引发一个 `RuntimeError`，并提供错误的详细信息。
3. `simulate_circuit` 函数：此函数模拟逻辑电路。使用了 `try-except` 块，以捕获模拟过程中的任何异常，并引发 `RuntimeError`，提供相关的错误消息。

除此之外，在输入信息时，使用 `assert` 对输入进行检查，要求输入必须是 I 或 O 开头。

```

1 | assert all(val.startswith('O') or val.startswith('I') for val in L[2:]), "Invalid gate inputs"

```

5 算法复杂度

1. 初始化门电路时，对于每个逻辑门，初始化一个 Gate 实例，时间复杂度为 $O(N)$ ，其中 N 是逻辑门的数量。
2. 在 `solve` 方法中，算法通过深度优先搜索（DFS）的方式解决了逻辑门之间的依赖关系。每个逻辑门最多访问一次，而访问每个逻辑门需要的时间取决于其输入端口的数量。因此，解决整个电路的时间复杂度为 $O(N*M)$ ，其中 N 是逻辑门的数量， M 是平均每个门的输入端口数量。
3. 如果不存在环，就要计算每个端口的输出。对于每个输出端口，算法需要遍历一次逻辑门，查找其输出值，时间复杂度为 $O(N)$ 。

综上，整个算法的时间复杂度为 $O(N*(M+1))$ ，其中 N 是逻辑门的数量， M 是平均每个门的输入端口数量。空间复杂度主要取决于存储 Gate 实例和输入数据的空间，为 $O(N)$ 。

6 性能分析与代码优化

6.1 性能分析

此次编程的性能主要受如下几个因素影响：

1. **逻辑门数量 (N)**：逻辑门数量直接影响着算法的性能。随着逻辑门数量的增加，算法需要解决的依赖关系也会增加，从而增加算法的执行时间。
2. **逻辑门的输入端口数量 (M)**：每个逻辑门的输入端口数量也会影响算法的性能。如果每个逻辑门都有大量的输入端口，算法在解决依赖关系和计算输出值时会花费更多的时间。
3. **输入数据的规模**：输入数据的规模（例如测试用例的数量、每个测试用例的输入值数量）也会影响算法的性能。更大规模的输入数据需要更多的计算资源来处理。
4. **硬件设备性能**：算法的性能还受到执行环境的硬件设备性能的影响。更高性能的处理器和内存可以提高算法的执行效率。

在此，使用 `cProfile` 进行性能测试。其中

- `ncalls`：表示函数被调用的次数。
- `tottime`：表示函数在自身内部消耗的总时间（不包括调用其他函数的时间）。
- `percall`：表示每次函数调用的平均时间（`tottime / ncalls`）。
- `cumtime`：表示函数在包括调用其他函数的情况下消耗的总时间（即总运行时间）。
- `percall`：表示每次函数调用的平均时间（`cumtime / ncalls`）。

162 function calls in 0.000 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	profile_logical_gates.py:124(simulate_circuit)
20	0.000	0.000	0.000	0.000	profile_logical_gates.py:27(calculate)
8	0.000	0.000	0.000	0.000	profile_logical_gates.py:37(<lambda>)
4	0.000	0.000	0.000	0.000	profile_logical_gates.py:38(<lambda>)
8	0.000	0.000	0.000	0.000	profile_logical_gates.py:39(<lambda>)
20	0.000	0.000	0.000	0.000	profile_logical_gates.py:47(solve)
20	0.000	0.000	0.000	0.000	profile_logical_gates.py:81(clear)
8	0.000	0.000	0.000	0.000	{built-in method builtins.all}
4	0.000	0.000	0.000	0.000	{built-in method builtins.any}
4	0.000	0.000	0.000	0.000	{built-in method builtins.len}
12	0.000	0.000	0.000	0.000	{built-in method builtins.print}
8	0.000	0.000	0.000	0.000	{built-in method builtins.sum}
40	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
4	0.000	0.000	0.000	0.000	{method 'values' of 'dict' objects}

对于图中的结果，可以得到如下结论：

- `simulate_circuit` 函数总共被调用了 1 次，消耗了 0.000 秒。这是因为我们在 `main` 函数中直接调用了它，且程序运行时间很短。
- `calculate` 函数总共被调用了 20 次，消耗了 0.000 秒。这个函数用于计算逻辑门的输出值，由于逻辑简单，消耗的时间非常小。
- `solve` 函数总共被调用了 20 次，消耗了 0.000 秒。这个函数用于解决逻辑门之间的依赖关系，同样由于逻辑简单，消耗的时间很小。
- `clear` 函数总共被调用了 20 次，消耗了 0.000 秒。这个函数用于清除逻辑门的输出值，同样消耗的时间很小。
- 其他的函数调用和内置函数调用消耗的时间也都非常小。
- 此处使用的测试样例数据量较小，故而所有的数值几乎都为 0。

6.2 代码优化

在编写代码时，主要对门电路的实现进行了优化。最开始实现的时候，每个门电路定义了一个函数，在 `calculate` 函数中进行计算时，调用函数即可。为了避免多次的函数调用，简化代码逻辑，将各个计算函数优化到 `calculate` 函数中。

还有一些其他优化比如数据结构的选择，已经在前面提过，在此不再赘述。

在此，给出几个优化思路。

1. 优化循环结构，减少时间复杂度和内存占用
2. 细化错误处理，更好地检测代码 bug
3. 添加编码注释，使得他人能够更好地理解代码逻辑

7 单元测试

测试用例设计思路、测试覆盖指标、覆盖率、测试通过率

测试用例设计中，核心问题是需要考虑输入的情况是否将编程的情况全部“遍历”到，也就是类似于符号执行中，要执行到所有的路径。通用的思路为：

1. **边界情况测试**：测试输入值为边界情况时的行为，例如空输入、最小输入、最大输入等。
2. **功能测试**：测试代码的各种功能是否按预期工作。针对 `Gate` 类的各个方法，测试不同逻辑函数、不同输入情况下的计算结果是否正确。
3. **异常情况测试**：测试代码在异常情况下的行为是否符合预期。例如，当输入参数非法时，是否会抛出预期的异常。
4. **性能测试**：测试代码在大规模输入情况下的性能表现。虽然代码已经经过优化，但仍然需要确保在大规模输入情况下能够快速运行。
5. **集成测试**：在整个程序中，模拟整个逻辑电路的行为并测试其正确性。

基于此，给出如下的设计思路：

1. 对于 `Gate` 类的 `calculate` 方法：
 - 测试各种逻辑函数的计算结果是否正确。
 - 测试输入参数为边界情况时的计算结果是否正确。
2. 对于 `Gate` 类的 `solve` 方法：
 - 测试当门输入端口存在循环依赖时，是否能够正确检测到循环。
 - 测试当门输入端口存在非法输入时，是否能够正确处理异常情况。
3. 对于 `get_data` 函数：
 - 测试当输入数据格式不正确时，是否能够正确抛出异常。
 - 测试当输入数据为空时，是否能够正确处理。
4. 对于 `simulate_circuit` 函数：
 - 测试当逻辑电路存在循环时，是否能够正确检测到循环。
 - 测试当输入数据为空时，是否能够正确处理。

有如下的设计测试覆盖指标、覆盖率和测试通过率的实现方案。

1. **测试覆盖指标设计**：
 - **代码行覆盖率**：使用测试覆盖工具运行测试，并生成代码行覆盖率报告。
 - **分支覆盖率**：同样使用测试覆盖工具，但需要确保测试用例能够覆盖所有可能的分支。
 - **函数覆盖率**：同样使用测试覆盖工具，确保每个函数都至少被一个测试用例调用。

2. 覆盖率计算:

- 对于代码行覆盖率和分支覆盖率，通过测试覆盖工具生成的报告直接获取。
- 对于函数覆盖率，我们可以通过记录每个函数的调用情况来计算。

3. 测试通过率计算:

- 在运行测试时，记录成功通过的测试用例数量。
- 在测试结束后，计算测试通过率。

在此，基于上述方案，给出如下实现代码:

```
1  import coverage
2  class Gate:
3      """
4      Represents a logic gate.
5
6      Attributes:
7          func (str): The logic function of the gate.
8          inputs (list): The list of input ports.
9          val (int): The output value of the gate.
10         loop (bool): Flag indicating if there is a loop during solving.
11     """
12
13     def __init__(self, func_str: str, input_lst: list):
14         """
15         Initializes a Gate instance.
16
17         Args:
18             func_str (str): The logic function string.
19             input_lst (list): The list of input ports.
20         """
21         self.func = func_str
22         self.inputs = input_lst
23         self.val = None
24         self.loop = False
25
26     def calculate(self, input_lst):
27         """
28         Calculates the output value of the gate.
29
30         Args:
31             input_lst (list): The list of input values.
32         """
33         try:
34             operations = {
35                 "NOT": lambda x: int(not x[0]),
36                 "AND": lambda x: int(all(x)),
37                 "OR": lambda x: int(any(x)),
38                 "XOR": lambda x: int(sum(x) % 2),
39                 "NAND": lambda x: int(not all(x)),
```



```

40         "NOR": lambda x: int(not any(x))
41     }
42     self.val = operations[self.func](input_lst)
43 except Exception as e:
44     raise RuntimeError(f"Error occurred while calculating {self.func}: {e}")
45
46 def solve(self, gates, inputs, visited):
47     """
48     Solves the gate.
49
50     Args:
51         gates (dict): Dictionary containing gate instances.
52         inputs (dict): Dictionary containing input values.
53         visited (set): Set containing visited gate names.
54     """
55     try:
56         if self.loop:
57             return 0
58         tmp_lst = []
59         if self.val is None:
60             for name in self.inputs:
61                 if name in visited:
62                     self.loop = True
63                     return 0
64                 if name[0] == "I":
65                     tmp_lst.append(inputs[int(name[1:])])
66             elif name in gates:
67                 if gates[name].val is None:
68                     visited.add(name)
69                     gates[name].solve(gates, inputs, visited)
70                     visited.remove(name)
71                 if gates[name].val is None:
72                     self.loop = True
73                     return 0
74             tmp_lst.append(gates[name].val)
75         self.calculate(tmp_lst)
76         return self.val
77     except Exception as e:
78         raise RuntimeError(f"Error occurred while solving: {e}")
79
80 def clear(self):
81     """Clears the output value of the gate."""
82     self.val = None
83
84
85 def get_data():
86     """
87     Retrieves test case data from input.
88
89     Returns:
90         list: List of test cases.
91     """

```



```

143         print("LOOP")
144         break
145     for output_index in expected_output:
146         print(gate_dict[f"O{output_index}"].val, end=' ')
147         print("")
148 except Exception as e:
149     raise RuntimeError(f"Error occurred while simulating circuit: {e}")
150
151 def run_tests():
152     # 运行测试, 并生成覆盖率报告
153     cov = coverage.Coverage()
154     cov.start()
155     # 运行测试代码
156     try:
157         test_cases = get_data()
158         for gate_dict, input_list, output_list in test_cases:
159             simulate_circuit(gate_dict, input_list, output_list)
160     except Exception as e:
161         print(f"Error occurred: {e}")
162     finally:
163         cov.stop()
164         cov.save()
165
166 def calculate_coverage():
167     # 计算覆盖率
168     cov = coverage.Coverage()
169     cov.load()
170     cov.report()
171
172 def calculate_pass_rate():
173     # 计算测试通过率
174     total_tests = get_total_tests() # 获取总测试用例数量
175     passed_tests = get_passed_tests() # 获取通过的测试用例数量
176     pass_rate = passed_tests / total_tests
177     return pass_rate
178
179 def get_total_tests():
180     # 获取总测试用例数量
181     total_tests = 0
182     try:
183         test_cases = get_data()
184         for _, input_list, _ in test_cases:
185             total_tests += len(input_list)
186     except Exception as e:
187         print(f"Error occurred while getting total tests: {e}")
188     return total_tests
189
190 def get_passed_tests():
191     # 获取通过的测试用例数量
192     passed_tests = 0
193     try:
194         test_cases = get_data()

```

```

195         for gate_dict, input_list, output_list in test_cases:
196             passed_tests += count_passed_tests(gate_dict, input_list, output_list)
197     except Exception as e:
198         print(f"Error occurred while getting passed tests: {e}")
199     return passed_tests
200
201 def count_passed_tests(gate_dict, input_list, output_list):
202     # 计算通过的测试用例数量
203     passed_tests = 0
204     try:
205         for inputs, expected_output in zip(input_list, output_list):
206             actual_output = simulate_circuit(gate_dict, inputs)
207             if actual_output == expected_output:
208                 passed_tests += 1
209     except Exception as e:
210         print(f"Error occurred while counting passed tests: {e}")
211     return passed_tests
212
213 if __name__ == "__main__":
214     run_tests()
215     calculate_coverage()
216     pass_rate = calculate_pass_rate()
217     print(f"Test pass rate: {pass_rate}")

```

基于给定的样例进行测试，可以看到其覆盖率为 44%。

Name	Stmts	Miss	Cover
test.py	141	79	44%
TOTAL	141	79	44%

使用 `unittest` 工具对其进行测试，测试每个门电路以及各个函数的功能。

```

1 import unittest
2
3 class TestGate(unittest.TestCase):
4     def test_calculate(self):
5         # Test each logic operation
6         gate = Gate("NOT", [0])
7         gate.calculate([0])
8         self.assertEqual(gate.val, 1)
9
10        gate = Gate("AND", [1, 0, 1])
11        gate.calculate([1, 0, 1])
12        self.assertEqual(gate.val, 0)
13
14        gate = Gate("OR", [1, 0, 1])
15        gate.calculate([1, 0, 1])
16        self.assertEqual(gate.val, 1)
17
18        gate = Gate("XOR", [1, 0, 1])
19        gate.calculate([1, 0, 1])
20        self.assertEqual(gate.val, 0)
21

```

```

22     gate = Gate("NAND", [1, 0, 1])
23     gate.calculate([1, 0, 1])
24     self.assertEqual(gate.val, 1)
25
26     gate = Gate("NOR", [1, 0, 1])
27     gate.calculate([1, 0, 1])
28     self.assertEqual(gate.val, 0)
29
30     # Test error handling
31     gate = Gate("NOT", [0])
32     with self.assertRaises(RuntimeError):
33         gate.calculate([]) # Empty input list
34
35     def test_solve(self):
36         # Test solving logic gates
37         gate_dict = {
38             "01": Gate("AND", ["I1", "I2"]),
39             "02": Gate("OR", ["I1", "I3"])
40         }
41         inputs = {1: 1, 2: 0, 3: 1}
42         visited = set()
43         gate_dict["01"].solve(gate_dict, inputs, visited)
44         gate_dict["02"].solve(gate_dict, inputs, visited)
45         self.assertEqual(gate_dict["01"].val, 0)
46         self.assertEqual(gate_dict["02"].val, 1)
47
48         # Test handling loops
49         gate_dict = {
50             "01": Gate("AND", ["02"]),
51             "02": Gate("OR", ["01"])
52         }
53         inputs = {}
54         visited = set()
55         gate_dict["01"].solve(gate_dict, inputs, visited)
56         self.assertTrue(gate_dict["01"].loop)
57         self.assertEqual(gate_dict["01"].val, None)
58
59     def test_clear(self):
60         # Test clearing gate value
61         gate = Gate("AND", ["I1", "I2"])
62         gate.val = 1
63         gate.clear()
64         self.assertEqual(gate.val, None)
65
66     if __name__ == "__main__":
67         unittest.main()
68

```

运行代码，得到如下结果，说明单元测试通过率为 100%。

```

(base) lxm@lxmliu2002:~/SE$ /bin/python3 /home/lxm/SE/submit/unit_test.py
...
-----
Ran 3 tests in 0.000s

OK

```

8 实验收获

经过本次实验，对于个人编程习惯有了一个较好的改观，掌握了一些之前从未接触到的编程规范性问题。同时，也了解到许多编程中能够用到的小技巧，比如性能分析、单元测试的工具等。

总的来说，收获颇丰。

9 文件说明

本次实验中用到的所有代码均在 `code` 文件夹中。

- `better_logical_gates.py` 是经过规范性处理后的代码
- `logical_gates.py` 是原始的编程代码
- `profile_logical_gates.py` 是用于使用 `cProfile` 进行性能分析的代码
- `unit_test.py` 则是使用 `unittest` 进行单元测试的代码
- `dist` 文件夹中存放本次实验生成的可执行文件。如图可以看到其文件格式

```

(.venv) (base) lxm@lxmliu2002:~/SE/dist$ file ./better_logical_gates
./better_logical_gates: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=979c889cc9ca56a33ed67aeb4d2d05cb8db2df1b, for GNU/Linux 2.6.32, stripped
(.venv) (base) lxm@lxmliu2002:~/SE/dist$ file ./logical_gates
./logical_gates: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=979c889cc9ca56a33ed67aeb4d2d05cb8db2df1b, for GNU/Linux 2.6.32, stripped
(.venv) (base) lxm@lxmliu2002:~/SE/dist$ file ./profile_logical_gates
./profile_logical_gates: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=979c889cc9ca56a33ed67aeb4d2d05cb8db2df1b, for GNU/Linux 2.6.32, stripped
(.venv) (base) lxm@lxmliu2002:~/SE/dist$ file ./unit_test
./unit_test: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=979c889cc9ca56a33ed67aeb4d2d05cb8db2df1b, for GNU/Linux 2.6.32, stripped

```

- `better_logical_gates`、`logical_gates` 和 `profile_logical_gates` 需要参照给定输入要求，输入规定格式的数据
- `unit_test` 直接运行即可得到输出结果

```

1 | .
2 | └─ code
3 |   └─ better_logical_gates.py
4 |   └─ logical_gates.py
5 |   └─ profile_logical_gates.py
6 |   └─ unit_test.py
7 |   └─ dist
8 |       └─ better_logical_gates
9 |       └─ logical_gates
10 |      └─ profile_logical_gates
11 |      └─ unit_test
12 | └─ report.pdf

```