



南開大學
Nankai University

网络空间安全学院
软件工程实验报告

软件编程实现、分析和测试

姓名：孙璐

学号：2112060

专业：信息安全

2024 年 5 月 19 日

目录

1 作业内容	2
1.1 作业要求	2
1.2 作业题目	2
1.2.1 题目描述	2
1.2.2 输入格式	2
1.2.3 输出格式	3
1.2.4 样例 1	4
2 Python 编程实现	5
2.1 代码实现	5
2.1.1 has_cycle	5
2.1.2 detect_cycle	7
2.1.3 主函数	9
2.1.4 测试	15
2.2 pylint 代码分析	16
2.3 profile 性能分析	22
2.4 pytest 单元测试	25
2.4.1 单元测试	25
2.4.2 pytest	25
2.4.3 pytest 规则	26
2.4.4 进行单元测试	26
2.4.5 测试覆盖指标与代码覆盖率	29
2.5 代码优化	31
2.5.1 代码分析	31
2.5.2 代码优化	32

1 作业内容

1.1 作业要求

基于飞书群附件一中的题目进行编程实现，并在此编程实现的基础上对代码进行单元测试、静态分析和代码的优化。

注：无编程语言要求，但注意选用合适的工具或框架进行代码分析和单元测试。如，Python 语言可以选用 pylint 代码分析，profile 性能分析、unittest/pytest 单元测试工具。

请在本次作业中充分考虑以下内容，并在作业中简要说明：

- a) 编程规范，参考的哪个规范，如何检查是否遵守编程规范的？
- b) 可扩展性，如何提高代码的可扩展性？
- c) 错误与异常处理
- d) 算法复杂度
- e) 性能分析与代码优化
- f) 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

1.2 作业题目

1.2.1 题目描述

本题中，你需要实现一个简单的数字逻辑电路模拟器。如果你已经有了此方面的基础，可以直接跳过本节。在阅读时，也可以参照前两个样例的图示和解释，这有助于你更好地理解数字逻辑电路的工作原理。

数字逻辑电路是用来传输数字信号（也就是二进制信号）的电路。一般来说，数字逻辑电路可以分为两大类，即组合逻辑（combinational logic）电路和时序逻辑（sequential logic）电路。在本题中，我们仅关注组合逻辑电路。这种电路仅由逻辑门（logical gate）构成。一个逻辑门可以理解为一个多输入单输出的函数，输入端连接至少一个信号，而后经过一定的逻辑运算输出一个信号。常见的逻辑门包括与（AND）、或（OR）、非（NOT）、异或（XOR）等，均与编程语言中的按位运算是对应的。

1.2.2 输入格式

从标准输入读入数据。

输入数据包括若干个独立的问题，第一行一个整数 Q ，满足 $1 \leq Q \leq Q_{max}$ 。接下来依次是这 Q 个问题的输入，你需要对每个问题进行处理，并且按照顺序输出对应的答案。每一个问题的输入在逻辑上可分为两部分。第一部分定义了整个电路的结构，第二部分定义了输入和输出的要求。实际上两部分之间没有分隔，顺序读入即可。

- 第一部分

第一行是两个空格分隔的整数 M, N ，分别表示了整个电路的输入和器件的数量，满足 $1 \leq N \leq N_{max}$ 并且 $0 \leq M \leq k_{max}N$ 。其中 k_{max} 和 N_{max} 都是与测试点编号有关的参数。

接下来 N 行，每行描述一个器件，编号从 1 开始递增，格式如下：

FUNC k L_1 L_2 ... L_k

其中 FUNC 代表具体的逻辑功能, k 表示输入的数量, 后面跟着该器件的 k 个输入端描述 L , 格式是以下二者之一:

- I_m : 表示第 m 个输入信号连接到此输入端, 保证 $1 \leq m \leq M$;
- O_n : 表示第 n 个器件的输出连接到此输入端, 保证 $1 \leq n \leq N$ 。

所有可能的 FUNC 和允许的输入端数量如下表所述:

FUNC	最少输入数量	最多输入数量	功能描述
NOT	1	1	非
AND	2	k_{max}	与
OR	2	k_{max}	或
XOR	2	k_{max}	异或
NAND	2	k_{max}	与非 (先全部与后取非)
NOR	2	k_{max}	或非 (先全部或后取非)

所有的器件均只有一个输出, 但这个输出信号可以被用作多个器件的输入。

• 第二部分

第一行是一个整数 S , 表示此电路需要运行 S 次。每次运行, 都会给定一组输入, 并检查部分器件的输出是否正确。 S 满足 $1 \leq S \leq S_{max}$, 其中 S_{max} 是一个与测试点编号有关的参数。

接下来的 S 行为输入描述, 每一行的格式如下:

$I_1 I_2 \dots I_M$

每行有 M 个可能为 0 或 1 的数字, 表示各个输入信号 (按编号排列) 的状态。

接下来的 S 行为输出描述, 每一行的格式如下:

$s_i O_1 O_2 \dots O_s$

第一个整数 $1 \leq s_i \leq N (1 \leq i \leq S)$ 表示需要输出的信号数量。后面共有 s_i 个在 1 到 N 之间的数字, 表示在对应的输入下, 组合逻辑完成计算后, 需要输出结果的器件编号。

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。

1.2.3 输出格式

输出到标准输出。

对于输入中的 Q 个问题, 你需要按照输入顺序输出每一个问题的答案:

如果你检测到电路中存在组合环路, 则请输出一行, 内容是 LOOP, 无需输出其他任何内容。

如果电路可以正常工作, 则请输出 S 行, 每一行包含 s_i 个用空格分隔的数字 (可能为 0 或 1), 依次表示“输出描述”中要求的各个器件的运算结果。

1.2.4 样例 1

- 样例 1 输入

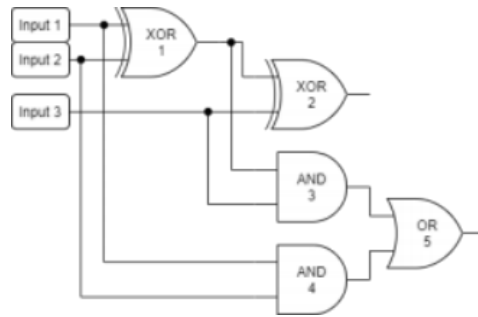
```
1
3 5
XOR 2 I1 I2
XOR 2 O1 I3
AND 2 O1 I3
AND 2 I1 I2
OR 2 O3 O4
4
0 1 1
1 0 1
1 1 1
0 0 0
2 5 2
2 5 2
2 5 2
2 5 2
```

- 样例输出

```
1 0
1 0
1 1
0 0
```

- 样例 1 解释

本样例只有一个问题，它定义的组合逻辑电路结构如下图所示。其功能是一位全加器，即将三个信号相加，得到一个两位二进制数。要求的器件 2 的输出是向更高位的进位信号，器件 5 的输出是本位的求和信号。



对于第一组输入 0 1 1，输出是 1 0；对于第二组输入 1 0 1，输出恰好依旧是 1 0（但电路内部状态不同）。

2 Python 编程实现

2.1 代码实现

2.1.1 has_cycle

函数 `has_cycle` 的目的是检测有向图中的环（循环），使用深度优先搜索（DFS）遍历图。它使用两个辅助列表 `visited` 和 `stack` 来帮助识别循环。

将当前节点 `v` 标记为已访问，以确保不会重复处理相同的节点，并标记在当前递归调用栈中。

遍历当前节点的所有邻居节点，递归调用 `has_cycle` 检查邻居节点是否已访问，如果递归调用返回 `True` 表示发现环，那么当前函数也返回 `True`。如果邻居节点已经在当前调用栈中，表示存在环，返回 `True`。

如果所有邻居节点都检查完毕且没有发现环，返回 `false` 表示不存在环。

```

def has_cycle(graph, v, visited, stack):
    visited[v] = True
    stack[v] = True

    for neighbor in graph.get(v, []):
        if not visited[neighbor]:
            if has_cycle(graph, neighbor, visited, stack):
                return True
        elif stack[neighbor]:
            return True

    stack[v] = False
    return False

```

```

1
2 def has_cycle(graph, v, visited, stack):
3     visited[v] = True
4     stack[v] = True
5
6     for neighbor in graph.get(v, []):
7         if not visited[neighbor]:
8             if has_cycle(graph, neighbor, visited, stack):
9                 return True
10        elif stack[neighbor]:

```

```
11         return True
12
13     stack[v] = False
14     return False
15
```

下面对当前 `has_cycle` 函数进行一些分析

- 复杂度分析

这个函数使用了深度优先搜索 DFS 遍历图,每个节点和边仅访问一次,因此时间复杂度为 $O(V+E)$ 。其中 V 是节点的数量, E 是边的数量。使用了 `visited` 数组和递归调用栈 `stack`, 大小都是 V , 因此空间复杂度为 $O(V)$ 。

- 编程规范

这里参考了下面两个编程规范

- PEP 8: Python 编码规范, 广泛使用和推荐的 Python 编码风格指南。
- 单一职责原则: 每个函数或方法应仅执行一项任务。

对具体的 `has_cycle` 函数, 是按如下方式遵循编程规范的:

- `has_cycle` 函数只执行一个任务, 即检测环路, 符合单一职责原则。
- 函数名 `has_cycle` 和变量名 `graph`, `v`, `visited`, `stack` 清晰易懂, 符合 PEP 8 的命名约定。函数参数传递局部变量, 未使用全局变量, 使函数更加独立和模块化。
- 每个语句独占一行, 没有长行或复杂的嵌套语句, 代码可读性强。保持了适当的空行, 逻辑分隔明确。
- 当前函数缺少注释, 虽然代码简单, 但在某些复杂的实现中应添加必要的注释来解释关键部分的逻辑。

- 可扩展性

如果想要提高该函数的可扩展性, 可以采用如下的方式

- 参数化和模块化:
将图表示方法参数化, 允许函数接受不同类型的图表示 (例如邻接矩阵、边列表等)。
- 添加配置选项:
允许配置递归深度或启用不同的图遍历算法 (例如广度优先搜索 BFS)。
- 使用面向对象编程:
将图和相关操作封装在类中, 使得操作图的功能可以通过继承和多态扩展。
- 分离关注点:
将图的构建和环检测逻辑分开, 使得每个模块都可以独立扩展和测试。

- 错误与异常处理

当前 `has_cycle` 函数没有错误和异常处理。如果想要进一步优化, 可以改进如下:

- 输入验证: 检查输入参数是否为合法图结构。
- 异常处理: 捕获递归深度超限的异常, 提供有用的错误信息。
- 性能分析和代码优化

对于非常大的图, 递归深度可能会达到 Python 的默认递归限制, 导致 `RecursionError`。

如果想要进一步优化。可以使用迭代替代递归, 以避免递归深度限制。并改进邻接表的数据结构, 如使用 `defaultdict` 提高访问速度。

2.1.2 detect_cycle

`detect_cycle` 是用来检测一个由元件和连接端口表示的图中是否存在环。首先构建图的邻接表表示, 然后使用 `has_cycle` 函数检查环的存在。

初始化一个空字典 `graph`, 表示设备之间的连接关系, 构建图的邻接表。

从 1-N 遍历循环每个元件, 并遍历获取该设备的每个输入端口。如果端口是连接到其他设备的输出 (端口名以 'O' 开头), 获取该输出连接的设备编号 `neighbor`, 如果 `neighbor` 不在 `graph` 中, 初始化其邻接列表为空, 并将当前设备 `i` 添加到 `neighbor` 的邻接列表中表示 `neighbor` 连接到 `i`。

初始化两个长度为 `N+1` 的布尔列表 `visited` 和 `stack`, 其中 `visited[i]` 用于记录元件 `i` 是否已访问, `stack[i]` 用于表示元件 `i` 是否在当前递归调用栈中。

循环遍历每个元件, 如果元件 `i` 未被访问, 调用 `has_cycle` 从元件 `i` 开始检测是否存在环, 如果 `has_cycle` 返回 `True`, 表示存在环, 返回 `True`。如果遍历所有元件后未检测到环, 返回 `false`。

```
def detect_cycle(devices, N):
    graph = {}

    for i in range(1, N + 1):
        _, inputs = devices[i]
        for port in inputs:
            if port[0] == 'O':
                neighbor = int(port[1:])
                if neighbor not in graph:
                    graph[neighbor] = []
                graph[neighbor].append(i)

    visited = [False] * (N + 1)
    stack = [False] * (N + 1)

    for i in range(1, N + 1):
        if not visited[i]:
            if has_cycle(graph, i, visited, stack):
                return True

    return False
```

```
1
2 def detect_cycle(devices, N):
3     graph = {}
4
5     for i in range(1, N + 1):
6         _, inputs = devices[i]
7         for port in inputs:
8             if port[0] == 'O':
9                 neighbor = int(port[1:])
10                if neighbor not in graph:
```



```
11         graph[neighbor] = []
12         graph[neighbor].append(i)
13
14     visited = [False] * (N + 1)
15     stack = [False] * (N + 1)
16
17     for i in range(1, N + 1):
18         if not visited[i]:
19             if has_cycle(graph, i, visited, stack):
20                 return True
21
22     return False
23
```

下面对当前 detect_cycle 函数进行一些分析

- 复杂度分析

这个函数使用了深度优先搜索 DFS 遍历图,每个节点和边仅访问一次,因此时间复杂度为 $O(V+E)$ 。其中 V 是节点的数量, E 是边的数量。使用了 visited 数组和递归调用栈 stack, 大小都是 V , 因此空间复杂度为 $O(V)$ 。

- 编程规范

这里参考了下面两个编程规范

- PEP 8: Python 编码规范, 广泛使用和推荐的 Python 编码风格指南。
- 单一职责原则: 每个函数或方法应仅执行一项任务。

对具体的 detect_cycle 函数, 是按如下方式遵循编程规范的:

- detect_cycle 函数只执行一个任务, 即检测环路, 符合单一职责原则。
- 函数名 detect_cycle 和变量名 graph, devices, visited, stack 清晰易懂, 符合 PEP 8 的命名约定。函数参数传递局部变量, 未使用全局变量, 使函数更加独立和模块化。
- 每个语句独占一行, 没有长行或复杂的嵌套语句, 代码可读性强。保持了适当的空行, 逻辑分隔明确。
- 当前函数缺少注释, 虽然代码简单, 但在某些复杂的实现中应添加必要的注释来解释关键部分的逻辑。

- 可扩展性

如果想要提高该函数的可扩展性, 可以采用如下的方式

- 参数化和模块化:
将图表示方法参数化, 允许函数接受不同类型的图表示 (例如邻接矩阵、边列表等)。
- 添加配置选项:
允许配置递归深度或启用不同的图遍历算法 (例如广度优先搜索 BFS)。

- 使用面向对象编程:
将图和相关操作封装在类中, 使得操作图的功能可以通过继承和多态扩展。
- 分离关注点:
将图的构建和环检测逻辑分开, 使得每个模块都可以独立扩展和测试。

- 错误与异常处理

当前 `has_cycle` 函数没有错误和异常处理。如果想要进一步优化, 可以改进如下:

- 输入验证: 检查输入参数是否为合法图结构。
- 异常处理: 捕获递归深度超限的异常, 提供有用的错误信息。

- 性能分析和代码优化

对于非常大的图, 递归深度可能会达到 Python 的默认递归限制, 导致 `RecursionError`。

如果想要进一步优化。可以使用迭代替代递归, 以避免递归深度限制。并改进邻接表的数据结构, 如使用 `defaultdict` 提高访问速度。

2.1.3 主函数

读取用户输入的问题数量 Q , 并将其转换为整数。遍历 Q 次获取问题, 获取每次电路的输入和器件的数量, 并将其转换成整数赋值给 M 和 N 。

初始化一个空字典 `device`, 存储器件信息。遍历器件编号范围 $1-N$, 获取用户输入第 i 个器件的信息, 并使用 `split` 分割成多个部分, 存储在 `device_info` 列表中。从器件信息中获取逻辑门类型, 存储在 `func` 变量中。从器件信息中获取端口数量和输入端口列表, 将第 i 个器件的信息存储在 `devices` 字典中, 其中键为器件编号 i , 值为一个元组, 包含逻辑门类型和输入端口列表。

使用 `detect_cycle` 检测当前是否存在环路, 如果存在环路, 则打印 `LOOP`, 并跳过当前循环。

读取用户输入的运行次数, 并将其转换成整数。初始化空列表用于存储所有运行次数的输入值。遍历运行次数 S 的范围, 获取 S 次运行电路的输入值。读取第 i 次运行的输入值, 添加到 `all_inputs` 列表中。初始化一个用于存储所有次数的需要的输出的器件编号及其对应的信号数量空列表 `all_output_devices`。遍历 S 次, 读取用户输入的需要输出的器件编号及其对应的信号数量使用 `split()` 方法将其分割成多个部分, 然后使用 `map()` 函数将每个部分转换为整数, 最后将其转换为列表形式, 并存储在 `output_devices` 变量中。将第 i 次运行的需要输出的器件编号及其对应的信号数量添加到 `all_output_devices` 列表中。

初始化一个空列表 `all_results` 用于存储所有运行次数的结果。使用 `zip()` 函数将 `all_inputs` 和 `all_output_devices` 同时迭代, 将每次迭代的输入值和需要输出的器件编号及其对应的信号数量存储在 `inputs` 和 `output_devices` 中。

通过内部循环, 对每个输入值和需要输出的器件编号及其对应的信号数量进行处理, 并得到相应的计算结果。对每个器件进行处理, 遍历器件的编号范围从 1 到 N 。从 `devices` 字典中获取第 i 个器件的逻辑门类型和输入端口列表。对每个输入端口进行处理, 根据端口类型 (是连接到输入信号还是其他器件的输出信号) 获取对应的值。根据逻辑门的功能, 计算器件的输出值, 并将结果存储在 `outputs` 字典中。最后, 根据需要输出的器件编号及其对应的信号数量, 将计算得到的结果存储在 `result` 列表中, 并将其添加到 `all_results` 列表中。最后将得到的计算结果输出。

```

if __name__ == "__main__":
    # 输入提示性语句
    print("请输入问题数量: ")

    # 读取问题数量
    Q = int(input())

    # 处理每个问题
    for i in range(1, Q+1):
        # 输入提示性语句
        print(f"请输入第{i}个问题的电路输入和器件的数量: ")

        # 读取输入和器件的数量
        M, N = map(int, input().split())

        # 初始化存储器件的字典
        devices = {}

        # 读取器件信息
        for J in range(1, N + 1):
            # 输入提示性语句
            print(f"请输入第{J}个器件的信息: ")

            # 读取器件信息
            device_info = input().split()
            func = device_info[0]
            k = int(device_info[1])
            inputs = device_info[2:]
            devices[J] = (func, inputs)

        # 检测是否存在环路
        if detect_cycle(devices, N):
            print("LOOP")
            continue

```

```

# 输入提示性语句
print("请输入运行次数: ")

# 读取运行次数
S = int(input())

# 存储所有运行次数的输入值
all_inputs = []
for p in range(1, S+1):
    # 输入提示性语句
    print(f"请输入第{p}次运行电路的输入值: ")

    # 读取电路的输入值
    inputs = list(map(int, input().split()))
    all_inputs.append(inputs)

# 读取所有运行次数的需要输出的器件编号及其对应的信号数量
all_output_devices = []
for z in range(1, S+1):
    # 输入提示性语句
    print(f"请输入第{z}次运行需要输出的器件编号及其对应的信号数量: ")
    output_devices = list(map(int, input().split()))
    all_output_devices.append(output_devices)

```

```

# 计算所有运行次数的结果
all_results = []
for inputs, output_devices in zip(all_inputs, all_output_devices):
    # 存储每个器件的输出结果
    outputs = {}
    for r in range(1, N + 1):
        func, in_ports = devices[r]
        in_values = []
        for port in in_ports:
            if port[0] == 'I': # 输入端连接的信号
                in_values.append(inputs[int(port[1:])] - 1)
            elif port[0] == 'O': # 输入端连接的器件的输出信号
                in_values.append(outputs[int(port[1:])])
        # 根据逻辑门的功能计算输出
        if func == 'AND':
            outputs[r] = int(all(in_values))
        elif func == 'OR':
            outputs[r] = int(any(in_values))
        elif func == 'XOR':
            outputs[r] = int(sum(in_values) % 2)
        elif func == 'NAND':
            outputs[r] = int(not all(in_values))
        elif func == 'NOR':
            outputs[r] = int(not any(in_values))
        elif func == 'NOT':
            outputs[r] = int(not in_values[0])

    # 存储当前运行次数的结果
    result = [outputs.get(dev, 0) for dev in output_devices[1:]] # 检查器件是否存在, 不存在则输出 0
    all_results.append(result)

# 输出所有运行次数的结果
for result in all_results:
    print(*result)

```

```

1
2 if __name__ == "__main__":
3     # 输入提示性语句
4     print(" 请输入问题数量: ")
5
6     # 读取问题数量
7     Q = int(input())
8
9     # 处理每个问题
10    for i in range(1, Q+1):
11        # 输入提示性语句
12        print(f" 请输入第{i}个问题的电路输入和器件的数量: ")
13
14        # 读取输入和器件的数量
15        M, N = map(int, input().split())
16
17        # 初始化存储器件的字典
18        devices = {}
19
20        # 读取器件信息
21        for J in range(1, N + 1):
22            # 输入提示性语句
23            print(f" 请输入第{J}个器件的信息: ")
24
25            # 读取器件信息

```

```
26     device_info = input().split()
27     func = device_info[0]
28     k = int(device_info[1])
29     inputs = device_info[2:]
30     devices[J] = (func, inputs)
31
32     # 检测是否存在环路
33     if detect_cycle(devices, N):
34         print("LOOP")
35         continue
36
37     # 输入提示性语句
38     print(" 请输入运行次数: ")
39
40     # 读取运行次数
41     S = int(input())
42
43     # 存储所有运行次数的输入值
44     all_inputs = []
45     for p in range(1,S+1):
46         # 输入提示性语句
47         print(f" 请输入第{p}次运行电路的输入值: ")
48
49         # 读取电路的输入值
50         inputs = list(map(int, input().split()))
51         all_inputs.append(inputs)
52
53     # 读取所有运行次数的需要输出的器件编号及其对应的信号数量
54     all_output_devices = []
55     for z in range(1,S+1):
56         # 输入提示性语句
57         print(f" 请输入第{z}次运行需要输出的器件编号及其对应的信号数量: ")
58         output_devices = list(map(int, input().split()))
59         all_output_devices.append(output_devices)
60
61     # 计算所有运行次数的结果
62     all_results = []
63     for inputs, output_devices in zip(all_inputs, all_output_devices):
64         # 存储每个器件的输出结果
65         outputs = {}
66         for r in range(1, N + 1):
67             func, in_ports = devices[r]
```

```

68         in_values = []
69         for port in in_ports:
70             if port[0] == 'I': # 输入端连接的信号
71                 in_values.append(inputs[int(port[1:]) - 1])
72             elif port[0] == 'O': # 输入端连接的器件的输出信号
73                 in_values.append(outputs[int(port[1:])])
74         # 根据逻辑门的功能计算输出
75         if func == 'AND':
76             outputs[r] = int(all(in_values))
77         elif func == 'OR':
78             outputs[r] = int(any(in_values))
79         elif func == 'XOR':
80             outputs[r] = int(sum(in_values) % 2)
81         elif func == 'NAND':
82             outputs[r] = int(not all(in_values))
83         elif func == 'NOR':
84             outputs[r] = int(not any(in_values))
85         elif func == 'NOT':
86             outputs[r] = int(not in_values[0])
87
88         # 存储当前运行次数的结果
89         # 检查器件是否存在, 不存在则输出 0
90         result = [outputs.get(dev, 0) for dev in output_devices[1:]]
91         all_results.append(result)
92
93         # 输出所有运行次数的结果
94         for result in all_results:
95             print(*result)
96

```

下面对当前主函数进行一些分析

- 复杂度分析

对于每个问题, 主函数包含了一个循环, 该循环的迭代次数取决于问题的数量 Q 。在每个问题的处理过程中, 还包含了对每个运行次数的循环, 其迭代次数取决于每个问题对应的运行次数 S 。在每个运行次数的处理过程中, 还包含了对每个器件的循环, 其迭代次数取决于器件的数量 N 。内部循环中的逻辑操作涉及了对每个器件的读取和处理, 以及对其输入值的计算和输出值的存储, 这些操作的复杂度取决于输入值的数量和每个器件所需的计算量。主函数的总时间复杂度可以表示为 $O(Q \times S \times N \times K)$, 其中 K 为每个器件所需的计算量。

- 编程规范

这里参考了下面两个编程规范

- PEP 8: Python 编码规范, 广泛使用和推荐的 Python 编码风格指南。

对具体的主函数, 是按如下方式遵循编程规范的:

- 大部分变量采用了清晰的命名方式, 如 Q、S、N、devices 等。这些命名能够清晰地表达变量的含义。
在循环中使用的迭代变量命名较为简洁, 例如使用了 i、J、p、z 等。这在单个循环中可能是足够的, 但如果有多重嵌套循环, 则可能会导致变量意义不清晰。
- 代码块采用了一致的缩进, 符合 Python 的缩进规范。这有助于代码的可读性和可维护性。字符串格式化使用了 f-string 的形式, 这是 Python3.6 及以上版本引入的新特性, 可以更清晰地插入变量, 提高了代码的可读性。
- 主函数中包含了一些注释, 用于解释代码的功能和输入提示信息。这有助于理解代码的目的和运行方式。
注释的语言简洁明了, 但可能还可以进一步详细说明一些复杂逻辑的实现方式, 以提高代码的可读性。

- 可扩展性

如果想要提高该函数的可扩展性, 可以采用如下的方式

- 参数化和模块化:
在函数或类的设计中, 尽量使用参数来传递数据和控制程序行为, 而不是硬编码在函数内部。这样可以使函数更加通用和灵活, 提高了可扩展性。
可以定义参数来控制处理问题的方式、处理输入值的类型、支持的逻辑门类型等, 从而使程序更加灵活。
将处理每个问题的逻辑封装成函数或类, 以实现模块化设计。这样可以使代码更易于理解、维护和扩展。
可以定义函数来处理不同类型的输入、计算逻辑门功能、输出结果等, 从而使得每个功能单元更加独立和可复用。

- 错误与异常处理

当前主函数没有错误和异常处理

- 用户输入验证不足:
当用户输入问题数量、输入和器件的数量、器件信息、运行次数、输入值等时, 没有进行有效的验证, 可能导致程序在接受到无效输入时产生错误。
缺乏对输入的类型、范围、格式等进行检查和处理, 可能会导致程序无法正确处理不合规范的输入。
- 图是否存在环路的检查:
在检测器件之间是否存在环路时, 使用了 has_cycle 函数, 但并未处理 has_cycle 函数本身可能引发的异常情况。
如果 has_cycle 函数出现异常, 可能会导致程序中断或输出错误的结果。

- 逻辑门功能处理:

目前的逻辑门功能处理部分没有对不合法的逻辑门功能进行检查, 如果用户提供了无法识别或不支持的逻辑门类型, 程序可能会出现错误。

- 结果输出处理:

目前的结果输出部分没有处理计算结果可能产生的异常情况, 如结果数量与预期不符或结果值超出范围等。如果计算结果存在异常, 可能会导致输出结果不准确或不完整。

当前主函数没有错误和异常处理。如果想要进一步优化, 可以改进如下:

- 输入验证:

在接受用户输入时, 应该进行有效的验证和处理, 以确保输入的合法性和正确性。可以通过条件判断、异常捕获等方式进行验证, 以避免程序因无效输入而产生错误。

- 异常处理机制:

在调用 `has_cycle` 函数时, 应该使用 `try-except` 语句捕获可能出现的异常, 并进行适当的处理或错误提示。这样可以增强程序的健壮性和鲁棒性, 防止异常情况导致程序崩溃或产生错误结果。

- 逻辑门功能检查:

在处理逻辑门功能时, 应该增加对逻辑门类型的合法性检查, 以确保只有合法的逻辑门类型被处理。可以使用条件判断或枚举类型等方式进行检查, 避免处理不支持的逻辑门类型而产生错误。

- 结果输出处理:

在输出结果之前, 应该对计算结果进行检查和处理, 以确保结果的正确性和完整性。可以检查结果数量是否与预期一致, 结果值是否在合理范围内, 从而避免输出错误的结果。

- 性能分析和代码优化

- 主函数中存在多层循环嵌套, 这可能会导致程序的性能下降。可以通过合并循环、减少循环次数等方式进行优化, 减少循环嵌套带来的性能损耗。

- 如果主函数中存在大量独立的计算任务, 可以考虑使用并行计算技术, 如多线程、多进程等, 将任务分配到多个处理器上并行执行, 以提高程序的运行速度。

2.1.4 测试

将附件样例中的输入作为输入, 验证代码的正确性, 结果如下图所示


```
请输入问题数量:
1
请输入输入和器件的数量:
3 5
请输入第1个器件的信息:
XOR 2 I1 I2
请输入第2个器件的信息:
XOR 2 O1 I3
请输入第3个器件的信息:
AND 2 O1 I3
请输入第4个器件的信息:
AND 2 I1 I2
请输入第5个器件的信息:
OR 2 O3 O4
请输入运行次数:
4
请输入第1次运行电路的输入值:
0 1 1
请输入第2次运行电路的输入值:
1 0 1
请输入第3次运行电路的输入值:
1 1 1
请输入第4次运行电路的输入值:
0 0 0
请输入第1次运行需要输出的器件编号及其对应的信号数量:
2 5 2
请输入第2次运行需要输出的器件编号及其对应的信号数量:
2 5 2
请输入第3次运行需要输出的器件编号及其对应的信号数量:
2 5 2
请输入第4次运行需要输出的器件编号及其对应的信号数量:
2 5 2
1 0
1 0
1 1
0 0
```

可以看到结果是正确的。

2.2 pylint 代码分析

Pylint 是一个 Python 代码分析工具，它分析 Python 代码中的错误，查找不符合代码风格标准（Pylint 默认使用的代码风格是 PEP 8，具体信息，请参阅参考资料）和有潜在问题的代码。

Pylint 是一个 Python 工具，除了平常代码分析工具的作用之外，它提供了更多的功能：如检查一行代码的长度，变量名是否符合命名标准，一个声明过的接口是否被真正实现等等。

如果运行两次 Pylint，它会同时显示出当前和上次的运行结果，从而可以看出代码质量是否得到了改进。

将文件从 jupyter 导出成.py 文件，在终端通过如下命令进行代码分析

```
pylint SE.py
```

结果如下图所示

```

PS C:\Users\LENOVO\Desktop> pylint SE.py
***** Module SE
SE.py:10:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:17:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:27:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:36:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:39:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:44:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:62:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:65:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:68:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:73:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:80:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:85:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:88:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:91:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:97:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:101:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:109:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:136:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:140:0: C0303: Trailing whitespace (trailing-whitespace)
SE.py:143:0: C0304: Final newline missing (missing-final-newline)
SE.py:1:0: C0114: Missing module docstring (missing-module-docstring)
SE.py:1:0: C0103: Module name "SE" doesn't conform to snake_case naming style (invalid-name)
SE.py:7:0: C0116: Missing function or method docstring (missing-function-docstring)
SE.py:7:21: C0103: Argument name "v" doesn't conform to snake_case naming style (invalid-name)
SE.py:25:0: C0116: Missing function or method docstring (missing-function-docstring)
SE.py:25:26: C0103: Argument name "N" doesn't conform to snake_case naming style (invalid-name)
SE.py:25:17: W0621: Redefining name 'devices' from outer scope (line 67) (redefined-outer-name)
SE.py:25:26: W0621: Redefining name 'N' from outer scope (line 64) (redefined-outer-name)
SE.py:28:8: W0621: Redefining name 'i' from outer scope (line 70) (redefined-outer-name)
SE.py:29:11: W0621: Redefining name 'inputs' from outer scope (line 78) (redefined-outer-name)
SE.py:30:12: W0621: Redefining name 'port' from outer scope (line 118) (redefined-outer-name)

Your code has been rated at 6.31/10 (previous run: 6.31/10, +0.00)

```

经过查询，上面的结果告诉了如下信息：

- C0303: Trailing whitespace (trailing-whitespace) 表示行尾存在空白字符
- C0114: Missing module docstring (missing-module-docstring) 表示缺少了模块的文档字符串。文档字符串是对函数或方法的简要描述，通常包括函数的功能、输入参数、返回值等信息。缺少文档字符串会降低代码的可读性和可维护性。
- C0116: Missing function or method docstring (missing-function-docstring) 表示缺少了函数或方法的文档字符串。
- C0103: Module name "SE" doesn't conform to snake_case naming style (invalid-name) C0103: Argument name "v" doesn't conform to snake_case naming style (invalid-name) 指出了模块名或相关变量名不符合蛇形命名风格。
- W0621: Redefining name 'devices' from outer scope 表示在函数内部重新定义了外部作用域中已存在的变量名。

针对代码分析提示的问题，进行如下修改：

- 针对 C0303 提示行尾存在空白字符的问题，将代码中的空行删除即可。虽然空行可以更清晰的显示代码的作用，但由于 PEP 8 编码规范，在 jupyter 编程过程中是会自动缩进的，导致出现行尾存在空白字符的问题。并且需要保证每一函数后面都有一个换行符
- 针对 C0114 和 C0116 提示的缺少了模块、函数或方法的文档字符串的问题，补充相应的文档字符串。

```
#!/usr/bin/env python
# coding: utf-8

"""
This module contains functions for detecting cycles in a circuit and
calculating its output for a given set of inputs.
"""

def has_cycle(graph, v, visited, stack):
    """
    Check if a graph has a cycle starting from vertex v.

    Args:
        graph (dict): The graph representation.
        v (int): The current vertex.
        visited (list): List to mark visited vertices.
        stack (list): List to track vertices in the current path.

    Returns:
        bool: True if a cycle is found, False otherwise.
    """
    visited[v] = True
    stack[v] = True
    for neighbor in graph.get(v, []):
        if not visited[neighbor]:
            if has_cycle(graph, neighbor, visited, stack):
                return True
        elif stack[neighbor]:
            return True
    stack[v] = False
    return False


def detect_cycle(devices, N):
    """
    Detect if there is a cycle in the circuit represented by devices.

    Args:
        devices (dict): Dictionary representing the circuit.
        n (int): Number of devices.

    Returns:
        bool: True if a cycle is detected, False otherwise.
    """
    graph = {}
    for i in range(1, N + 1):
        inputs = devices[i]
        for port in inputs:
            if port[0] == '0':
                neighbor = int(port[1:])
                if neighbor not in graph:
                    graph[neighbor] = []
                graph[neighbor].append(i)
    visited = [False] * (N + 1)
    stack = [False] * (N + 1)
    for i in range(1, N + 1):
        if not visited[i]:
            if has_cycle(graph, i, visited, stack):
                return True
    return False
```

- 针对 C0103 提示的模块名不符合蛇形命名风格，需要将参数名称改为符合 snake_case 命名风格的名称，如 v 改成 vertex，N 改成 num_devices 等，将命名从大写字母改成小写字母 + 下划线分隔的形式。
- 针对 W0621 提示的在函数内部重新定义了外部作用域中已存在的变量名的问题，需要确保内外层作用域中变量名称的唯一性。可以通过重命名内层作用域中的变量来避免这种重定义。通过修改一些参数或临时变量的名称解决这个问题。

将修改后的代码文件存储为 se_pylint.py，在终端通过如下命令进行代码分析

```
pylint se_pylint.py
```

结果如下：

```
PS C:\Users\LENOVO\Desktop> pylint se_pylint.py

-----
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
PS C:\Users\LENOVO\Desktop>
```

可以看到成功通过了 pylint 的代码分析。

下面附上修改后的代码：

```
1
2 #!/usr/bin/env python
3 # coding: utf-8
4
5 """
6 This module contains functions for detecting cycles in a circuit and
7 calculating its output for a given set of inputs.
8 """
9
10 def has_cycle(graph, vertex, visited, stack):
11     """
12     Check if a graph has a cycle starting from vertex v.
13
14     Args:
15         graph (dict): The graph representation.
16         vertex (int): The current vertex.
17         visited (list): List to mark visited vertices.
18         stack (list): List to track vertices in the current path.
19
20     Returns:
21         bool: True if a cycle is found, False otherwise.
22     """
23     visited[vertex] = True
24     stack[vertex] = True
25     for neighbor in graph.get(vertex, []):
26         if not visited[neighbor]:
27             if has_cycle(graph, neighbor, visited, stack):
28                 return True
29         elif stack[neighbor]:
30             return True
31     stack[vertex] = False
32     return False
33
34 def detect_cycle(circuit_devices, num_devices):
35     """
36     Detect if there is a cycle in the circuit represented by devices.
37
38     Args:
39         circuit_devices (dict): Dictionary representing the circuit.
40         num_devices (int): Number of devices.
41
```

```

42     Returns:
43         bool: True if a cycle is detected, False otherwise.
44     """
45     graph = {}
46     for device_id in range(1, num_devices + 1):
47         _, inputs_ports = circuit_devices[device_id]
48         for port in inputs_ports:
49             if port[0] == '0':
50                 neighbor = int(port[1:])
51                 if neighbor not in graph:
52                     graph[neighbor] = []
53                 graph[neighbor].append(device_id)
54     visited = [False] * (num_devices + 1)
55     stack = [False] * (num_devices + 1)
56     for device_id in range(1, num_devices + 1):
57         if not visited[device_id]:
58             if has_cycle(graph, device_id, visited, stack):
59                 return True
60     return False
61
62 if __name__ == "__main__":
63     # 输入提示性语句
64     print(" 请输入问题数量: ")
65     # 读取问题数量
66     Q = int(input())
67     # 处理每个问题
68     for _ in range(Q):
69         # 输入提示性语句
70         print(" 请输入输入和器件的数量: ")
71         # 读取输入和器件的数量
72         M, N = map(int, input().split())
73         # 初始化存储器件的字典
74         devices = {}
75         # 读取器件信息
76         for i in range(1, N + 1):
77             # 输入提示性语句
78             print(f" 请输入第{i}个器件的信息: ")
79             # 读取器件信息
80             device_info = input().split()
81             func = device_info[0]
82             k = int(device_info[1])
83             inputs = device_info[2:]

```

```

84         devices[i] = (func, inputs)
85     # 检测是否存在环路
86     if detect_cycle(devices, N):
87         print("LOOP")
88         continue
89     # 输入提示性语句
90     print(" 请输入运行次数: ")
91     # 读取运行次数
92     S = int(input())
93     # 存储所有运行次数的输入值
94     all_inputs = []
95     for i in range(1,S+1):
96         # 输入提示性语句
97         print(f" 请输入第{i}次运行电路的输入值: ")
98         # 读取电路的输入值
99         inputs = list(map(int, input().split()))
100        all_inputs.append(inputs)
101    # 读取所有运行次数的需要输出的器件编号及其对应的信号数量
102    all_output_devices = []
103    for i in range(1,S+1):
104        # 输入提示性语句
105        print(f" 请输入第{i}次运行需要输出的器件编号及其对应的信号数量: ")
106        output_devices = list(map(int, input().split()))
107        all_output_devices.append(output_devices)
108    # 计算所有运行次数的结果
109    all_results = []
110    for inputs, output_devices in zip(all_inputs, all_output_devices):
111        # 存储每个器件的输出结果
112        outputs = {}
113        for i in range(1, N + 1):
114            func, in_ports = devices[i]
115            in_values = []
116            for input_port in in_ports:
117                if input_port[0] == 'I': # 输入端连接的信号
118                    in_values.append(inputs[int(input_port[1:])] - 1))
119                elif input_port[0] == 'O': # 输入端连接的器件的输出信号
120                    in_values.append(outputs[int(input_port[1:])])
121            # 根据逻辑门的功能计算输出
122            if func == 'AND':
123                outputs[i] = int(all(in_values))
124            elif func == 'OR':
125                outputs[i] = int(any(in_values))

```

```

126         elif func == 'XOR':
127             outputs[i] = int(sum(in_values) % 2)
128         elif func == 'NAND':
129             outputs[i] = int(not all(in_values))
130         elif func == 'NOR':
131             outputs[i] = int(not any(in_values))
132         elif func == 'NOT':
133             outputs[i] = int(not in_values[0])
134         # 存储当前运行次数的结果
135         # 检查器件是否存在，不存在则输出 0
136         result = [outputs.get(dev, 0) for dev in output_devices[1:]]
137         all_results.append(result)
138         # 输出所有运行次数的结果
139         for result in all_results:
140             print(*result)
141
142

```

2.3 profile 性能分析

为了方便性能分析，需要将主要执行部分放到单独的 main 函数中，同时将提示性输出的中文语句注释掉。将修改后的文件命名为 se_profile1.py。

对 se_profile1.py 进行代码分析，可以看到出现了新的提示。

```

PS C:\Users\LENOVO\Desktop> pylint se_profile1.py
***** Module se_profile1
se_profile1.py:61:0: C0116: Missing function or method docstring (missing-function-docstring)
se_profile1.py:61:0: R0914: Too many local variables (19/15) (too-many-locals)
se_profile1.py:65:4: C0103: Variable name "Q" doesn't conform to snake_case naming style (invalid-name)
se_profile1.py:71:8: C0103: Variable name "M" doesn't conform to snake_case naming style (invalid-name)
se_profile1.py:71:11: C0103: Variable name "N" doesn't conform to snake_case naming style (invalid-name)
se_profile1.py:91:8: C0103: Variable name "S" doesn't conform to snake_case naming style (invalid-name)
se_profile1.py:61:0: R0912: Too many branches (17/12) (too-many-branches)
se_profile1.py:71:8: W0612: Unused variable 'M' (unused-variable)
se_profile1.py:81:12: W0612: Unused variable 'k' (unused-variable)

```

继续对 se_profile.py 文件进行相应修改，直到全部通过。

```

PS C:\Users\LENOVO\Desktop> pylint se_profile1.py
-----
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)

```

在终端输入如下命令进行性能分析

```
python -m cProfile -o C:\Users\LENOVO\Desktop\se_profile1.prof C:\Users\LENOVO\Desktop\se_profile1.py
```

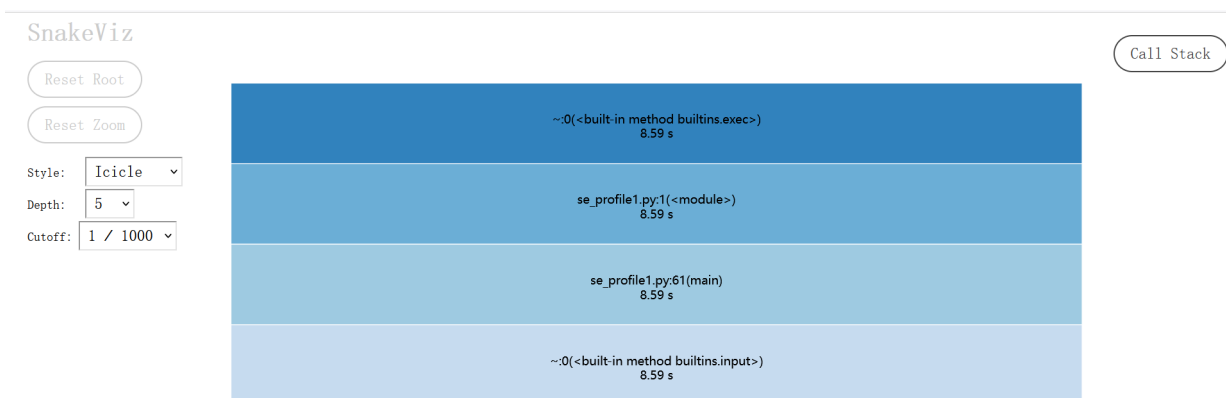
```
PS C:\Users\LENOVO\Desktop> python -m cProfile -o C:\Users\LENOVO\Desktop\se_profile1.prof C:\Users\LENOVO\Desktop\se_profile1.py
1
3 5
XOR 2 I1 I2
XOR 2 01 I3
AND 2 01 I3
AND 2 I1 I2
OR 2 03 04
4
0 1 1
1 0 1
1 1 1
0 0 0
2 5 2
2 5 2
2 5 2
2 5 2
1 0
1 0
1 1
0 0
PS C:\Users\LENOVO\Desktop>
```

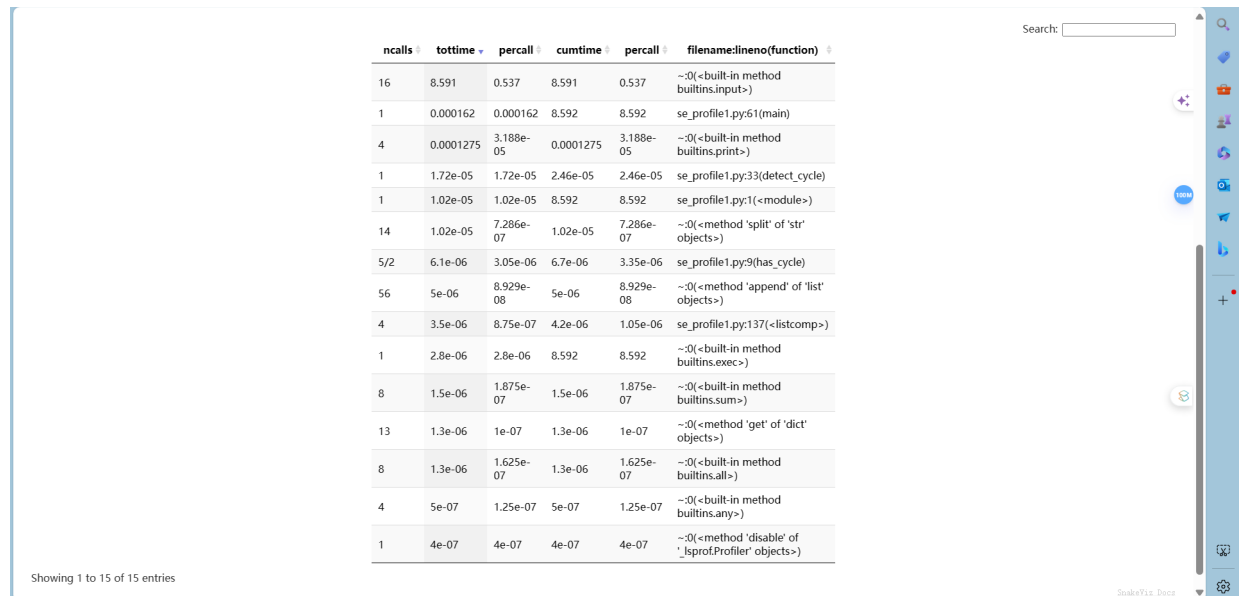
可以看到运行的结果是正确的。

安装 snakeviz，并使用如下命令可视化查看分析结果

```
snakeviz C:\Users\LENOVO\Desktop\se_profile1.prof
```

结果如下：





ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
16	8.591	0.537	8.591	0.537	~0(<built-in method builtins.input>)
1	0.000162	0.000162	8.592	8.592	se_profile1.py:61(main)
4	0.0001275	3.188e-05	0.0001275	3.188e-05	~0(<built-in method builtins.print>)
1	1.72e-05	1.72e-05	2.46e-05	2.46e-05	se_profile1.py:33(detect_cycle)
1	1.02e-05	1.02e-05	8.592	8.592	se_profile1.py:1(<module>)
14	1.02e-05	7.286e-07	1.02e-05	7.286e-07	~0(<method 'split' of 'str' objects>)
5/2	6.1e-06	3.05e-06	6.7e-06	3.35e-06	se_profile1.py:9(has_cycle)
56	5e-06	8.929e-08	5e-06	8.929e-08	~0(<method 'append' of 'list' objects>)
4	3.5e-06	8.75e-07	4.2e-06	1.05e-06	se_profile1.py:137(<listcomp>)
1	2.8e-06	2.8e-06	8.592	8.592	~0(<built-in method builtins.exec>)
8	1.5e-06	1.875e-07	1.5e-06	1.875e-07	~0(<built-in method builtins.sum>)
13	1.3e-06	1e-07	1.3e-06	1e-07	~0(<method 'get' of 'dict' objects>)
8	1.3e-06	1.625e-07	1.3e-06	1.625e-07	~0(<built-in method builtins.all>)
4	5e-07	1.25e-07	5e-07	1.25e-07	~0(<built-in method builtins.any>)
1	4e-07	4e-07	4e-07	4e-07	~0(<method 'disable' of '_lsprof.Profiler' objects>)

使用 pstats 对生成的 se_profile1.prof 文件进行查看

```
python -m pstats C:\Users\LENOVO\Desktop\se_profile1.prof
```

执行 sort cumtime 来按照累计时间对函数进行排序，然后再次执行 stats 命令来查看排序后的统计信息。

```
PS C:\Users\LENOVO> python -m pstats C:\Users\LENOVO\Desktop\se_profile1.prof
Welcome to the profile statistics browser.
C:\Users\LENOVO\Desktop\se_profile1.prof% sort cumtime
C:\Users\LENOVO\Desktop\se_profile1.prof% stats
Sat May 18 23:03:32 2024 C:\Users\LENOVO\Desktop\se_profile1.prof

137 function calls (134 primitive calls) in 8.592 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.000    0.000    8.592    8.592    {built-in method builtins.exec}
1      0.000    0.000    8.592    8.592    C:\Users\LENOVO\Desktop\se_profile1.py:1(<module>)
1      0.000    0.000    8.592    8.592    C:\Users\LENOVO\Desktop\se_profile1.py:61(main)
16     8.591    0.537    8.591    0.537    {built-in method builtins.input}
4      0.000    0.000    0.000    0.000    {built-in method builtins.print}
1      0.000    0.000    0.000    0.000    C:\Users\LENOVO\Desktop\se_profile1.py:33(detect_cycle)
14     0.000    0.000    0.000    0.000    {method 'split' of 'str' objects}
5/2    0.000    0.000    0.000    0.000    C:\Users\LENOVO\Desktop\se_profile1.py:9(has_cycle)
56     0.000    0.000    0.000    0.000    {method 'append' of 'list' objects}
4      0.000    0.000    0.000    0.000    C:\Users\LENOVO\Desktop\se_profile1.py:137(<listcomp>)
8      0.000    0.000    0.000    0.000    {built-in method builtins.sum}
13     0.000    0.000    0.000    0.000    {method 'get' of 'dict' objects}
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}
1      0.000    0.000    0.000    0.000    {method 'disable' of '_lsprof.Profiler' objects}

C:\Users\LENOVO\Desktop\se_profile1.prof%
```

上面的信息提供了关于程序运行期间函数调用的统计信息，其中包括

- 总体统计信息

程序共进行了 137 次函数调用（其中 134 次是原始调用，3 次是由原始调用衍生出来的）。程序的总运行时间为 8.592 秒。

- 函数统计信息

has_cycle 函数被调用了 5 次, 其中 2 次是由原始调用衍生出来的。总共消耗了 0.000 秒的时间。

detect_cycle 函数被调用了 1 次, 消耗了 0.000 秒的时间。

main 函数被调用了 1 次, 消耗了 8.592 秒的时间。

以及其他函数 (内置方法) 被调用的次数和消耗时间

根据提供的统计信息, 该程序的性能瓶颈主要在 main 函数中, 因为里面包含了很多次获取数据的 for 嵌套循环。但在本次的实验中, 这样的循环嵌套是必须的, 很难再优化, 除非可以修改程序数据的输入顺序。

2.4 pytest 单元测试

2.4.1 单元测试

单元测试是指在软件开发当中针对软件的最小单位 (函数、方法) 进行正确性的检查测试。

单元测试框架主要目的如下:

- 1) 测试发现: 从多个文件里面去找到我们的测试用例
- 2) 测试执行: 按照一定的顺序和规则去执行, 并生成结果
- 3) 测试判断: 通过断言判断预期结果和实际结果的差异
- 4) 测试报告: 统计测试进度、耗时、通过率、生成测试报告

2.4.2 pytest

pytest 是 Python 的一个单元测试框架, 与 python 自带的 unittest 测试框架类似;

pytest 比 unittest 框架使用起来更简洁, 效率更高, 而且特性比较多, 也就非常的灵活;

比如 pytest 常用的特性有:

- 对 case 可以进行设置跳过, 也可以进行标记 (比如失败等);
- 可以重复执行失败的 case;
- 可以兼容执行 unittest 编写的 case;
- 有很多第三方的插件, 比如报告 allure 等;

如:

- pytest
- pytest-html # (生成 html 格式的自动化测试报告)
- pytest-xdist # (测试用例分布式执行, 多 CPU 分发)
- pytest-ordering # (用于改变测试用例的执行顺序)
- pytest-rerunfailures # (用例失败后重跑)
- allure-pytest # (用于生成美观的测试报告)
- ...
- 可以与 Jenkins 集成;
- 和 unittest 一样支持参数化, 但 Pytest 方法更多, 更灵活;
- 可以和 selenium、requests、appium 结合实现 web 自动化、接口自动化、app 自动化等

2.4.3 pytest 规则

pytest 命名规则如下:

- 1) 模块名必须以 test_ 开头或者以 test_ 结尾
- 2) 测试类名必须以 Test 开头, 并且不能有 init 方法
- 3) 测试方法必须以 test 开头

2.4.4 进行单元测试

编写单元测试文件 se_pytest_test.py 如下:

```

1
2 import pytest
3 from io import StringIO
4 from unittest.mock import patch
5 from se_pytest_origin import detect_cycle, has_cycle, main
6
7 # 对 'detect_cycle' 函数的参数化测试用例
8 @pytest.mark.parametrize("circuit_devices, num_devices, expected", [
9     ({1: ('AND', ['I1', 'I2']), 2: ('OR', ['I3', 'O1']), 3: ('NOT', ['O1'])}, 3, False),
10    ({1: ('AND', ['I1', 'I2']), 2: ('OR', ['I3', 'O1']), 3: ('NOT', ['O2']), 4: ('NAND', ['O3',
11    ({1: ('AND', ['I1', 'I2']), 2: ('OR', ['I3', 'I4']), 3: ('XOR', ['O1', 'I5']), 4: ('NAND',
12    ({1: ('AND', ['I1', 'O5']), 2: ('OR', ['O1', 'I4']), 3: ('XOR', ['O2', 'I5']), 4: ('NAND',
13 ])
14 def test_detect_cycle(circuit_devices, num_devices, expected):
15     assert detect_cycle(circuit_devices, num_devices) == expected
16
17
18 # 对 'has_cycle' 函数的参数化测试用例
19 @pytest.mark.parametrize("graph, start_vertex, expected", [
20     ({1: [2], 2: [3], 3: [1]}, 1, True),
21     ({1: [2], 3: [4]}, 1, False),
22     ({1: [2], 2: [3], 3: [1], 4: [5]}, 1, True)
23 ])
24 def test_has_cycle(graph, start_vertex, expected):
25     visited = [False] * (max(graph.keys()) + 1)
26     stack = [False] * (max(graph.keys()) + 1)
27     assert has_cycle(graph, start_vertex, visited, stack) == expected
28
29 # 对 'main' 函数的参数化测试用例
30 @pytest.mark.parametrize("inputs, expected_output", [
31     ([
32         '2', '2 3', 'AND 2 I1 O3', 'OR 2 O1 I2', 'XOR 2 O2 I2', '3 5',
33         'XOR 2 I1 I2', 'XOR 2 O1 I3', 'AND 2 O1 I3', 'AND 2 I1 I2', 'OR 2 O3 O4',

```

```

34         '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'
35     ], ['LOOP', '1 0', '1 0', '1 1', '0 0']),
36     ([
37         '1', '3 3', 'AND 2 I1 I2', 'OR 2 I2 I3', 'XOR 2 I1 I3', '4',
38         '1 0 1', '0 0 1', '1 1 1', '1 1 0', '2 1 3', '2 3 1', '3 3 2 1', '1 3'
39     ], ['0 0', '1 0', '0 1 1', '1']),
40     ([
41         '1', '5 5',
42         'XOR 2 I1 I2', 'NAND 2 O1 I3', 'AND 2 O1 I4', 'NOT 1 I5', 'OR 2 O3 O4',
43         '3', '0 1 0 1 1', '0 0 1 1 1', '1 0 1 1 1', '2 5 2', '3 5 3 4', '4 5 4 1 3'
44     ], ['1 1', '0 0 0', '1 0 1 1']),
45 ])
46 @patch('builtins.input')
47 @patch('sys.stdout', new_callable=StringIO)
48 def test_main(mock_stdout, mock_input, inputs, expected_output):
49     mock_input.side_effect = inputs
50     main()
51     output = mock_stdout.getvalue().strip().split('\n')
52     assert output == expected_output
53
54 if __name__ == "__main__":
55     pytest.main()
56

```

使用如下命令进行单元测试

```
pytest se_pytest_test.py
```

测试用例设计思路如下:

- test_detect_cycle 函数:

对 detect_cycle 函数进行了参数化测试, 测试了不同的电路设备和设备数量组合下的环路检测情况。针对有环路和没有环路两种情况都进行了测试。

每个测试用例包括了输入的电路设备字典 circuit_devices、设备数量 num_devices 以及预期输出的环路检测结果 expected。

可能造成环路的情况包括但不限于以下几种情况:

- 反馈路径

在一个由多个逻辑门组成的电路中, 如果某个逻辑门的输出信号反馈到另一个逻辑门的输入端, 而後者的输出信号又反馈到前者的输入端, 就会产生环路。

- 逻辑回路

当一组器件的输出信号依赖于相互之间的输入信号时，如果存在某种组合使得输出信号又反过来影响输入信号，就会形成逻辑回路。

例如，当一个器件的输出信号作为另一个器件的输入信号时，而后者的输出信号又返回到前者的输入信号，形成了一个逻辑回路，这样就有可能形成一个环路。

– 错误的连接或设计

如错误地将某个器件的输出信号连接到自身的输入端，或者在设计电路时忽略了某些信号路径，从而造成了环路的形成。

针对以上 3 种可能造成环路的情况，在考虑单元测试样例中都有考虑到，详情可见代码。

- test_has_cycle 函数：

对 has_cycle 函数进行了参数化测试，测试了不同的图结构和起始顶点下的环路检测情况。针对有环图和无环图两种情况都进行了测试。

每个测试用例包括了输入的图结构 graph、起始顶点 start_vertex 以及预期输出的环路检测结果 expected。

- test_main 函数：

对 main 函数进行了参数化测试，测试了不同输入和预期输出下的整个主函数的行为。

每个测试用例包括了输入列表 inputs 和预期输出列表 expected_output。

在 main 函数单元测试样例的编写中，考虑到了有 1 个或多个问题的情况，也考虑到了电路可能存在环路需要输出 LOOP 的情况。针对每一次的电路运行，在单元测试样例的编写中，考虑到了每次电路输入信号可能数量不同，信号输出数量也可能不同，输出器件的顺序不一定是递增的，可能以任意顺序出现等不同的情况。单元测试样例详情可见代码。

在进行单元测试后，针对上面考虑到的不同情况，编写出了覆盖了考虑到的全部情况的 10 个单元测试样例，现有实现的代码全部通过测试，通过率 100%。

使用下列命令进行单元测试

```
pytest se_pytest_test.py
```

单元测试结果如下：

```
PS C:\Users\LENOVO\Desktop> pytest se_pytest_test.py
===== test session starts =====
platform win32 -- Python 3.11.4, pytest-7.4.0, pluggy-1.0.0
rootdir: C:\Users\LENOVO\Desktop
plugins: anyio-3.5.0
collected 10 items

se_pytest_test.py ..... [100%]

===== 10 passed in 0.03s =====
PS C:\Users\LENOVO\Desktop>
```

可以看到，代码的测试通过率达到 100%。

2.4.5 测试覆盖指标与代码覆盖率

测试覆盖指标是衡量测试代码覆盖率的指标，它表示了在执行测试时被覆盖到的代码行、分支、函数等的比例。通常包括语句覆盖、分支覆盖、函数覆盖等不同的覆盖指标，用于衡量测试对代码的覆盖程度。

覆盖率是测试覆盖指标的一种度量，表示了被测试覆盖的代码量占总代码量的比例。例如，语句覆盖率表示被执行的代码语句占总代码语句的比例，分支覆盖率表示被执行的分支占总分支数的比例，函数覆盖率表示被调用的函数占总函数数的比例。

使用下列命令查看测试的代码覆盖率

```
pytest -cov=. se_pytest_test.py
```

```
PS C:\Users\LENOVO\Desktop> pytest --cov=. se_pytest_test.py
===== test session starts =====
platform win32 -- Python 3.11.4, pytest-7.4.0, pluggy-1.0.0
rootdir: C:\Users\LENOVO\Desktop
plugins: anyio-3.5.0, cov-5.0.0
collected 10 items

se_pytest_test.py ..... [100%]

----- coverage: platform win32, python 3.11.4-final-0 -----
Name                               Stmts  Miss  Cover
-----
SE.py                               84      84     0%
se_profile1.py                      80      80     0%
se_pylint.py                        85      85     0%
se_pytest_origin.py                 80       2    98%
se_pytest_test.py                   22       1    95%
-----
TOTAL                               351     252    28%

===== 10 passed in 0.12s =====
PS C:\Users\LENOVO\Desktop>
```

可以看到，se_pytest_test.py 文件的测试语句覆盖率为 95%。这表示测试文件中的大部分代码行都被测试覆盖到了，只有 1 个代码语句行未被测试覆盖到。

使用上述命令生成了一个.coverage 文件，可使用下列命令进行查看。

```
coverage report -m
```

```
PS C:\Users\LENOVO\Desktop> coverage report -m
Name                               Stmts  Miss  Cover  Missing
-----
SE.py                               84      84     0%    7-143
se_profile1.py                      80      80     0%    4-144
se_pylint.py                        85      85     0%    4-138
se_pytest_origin.py                 80       2    98%    133, 144
se_pytest_test.py                   22       1    95%    54
-----
TOTAL                               351     252    28%
```

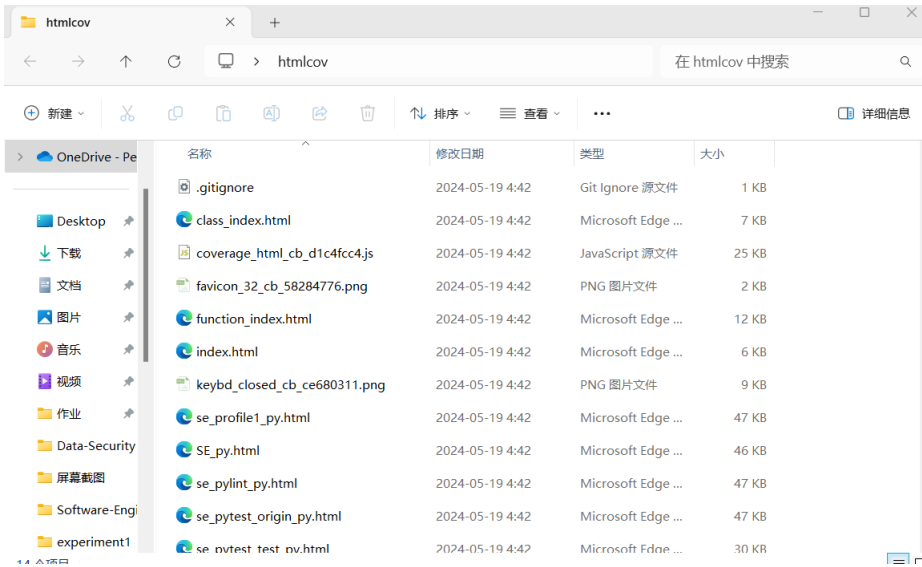
在图中可以看到具体文件中是哪行代码未被覆盖到。

使用下列命令，可以生成 HTML 格式的覆盖率报告，更直观的查看到覆盖情况。运行完这个命令后，coverage.py 会在当前目录下生成一个名为 htmlcov 的文件夹，其中包含了 HTML 格式的覆盖率

报告。可以在浏览器中打开 index.html 文件来查看报告。

coverage html

```
PS C:\Users\LENOVO\Desktop> coverage html
Wrote HTML report to htmlcov\index.html
PS C:\Users\LENOVO\Desktop>
```



点击 index.html 文件进行查看

Coverage report: 28%

Files

Functions

Classes

coverage.py v7.5.1, created at 2024-05-19 04:42 +0800

filter...

☐ hide covered

File	statements	missing	excluded	coverage
se_profile1.py	80	80	0	0%
se_pylint.py	85	85	0	0%
se_pytest_origin.py	80	2	0	98%
se_pytest_test.py	22	1	0	95%
SE.py	84	84	0	0%
Total	351	252	0	28%

coverage.py v7.5.1, created at 2024-05-19 04:42 +0800

Coverage report: 28%

Files Functions Classes

coverage.py v7.5.1, created at 2024-05-19 04:42 +0800

File	function	statements	missing	excluded	coverage
se_profile1.py	has_cycle	10	10	0	0%
se_profile1.py	detect_cycle	16	16	0	0%
se_profile1.py	main	48	48	0	0%
se_profile1.py	(no function)	6	6	0	0%
se_pylint.py	has_cycle	10	10	0	0%
se_pylint.py	detect_cycle	16	16	0	0%
se_pylint.py	(no function)	59	59	0	0%
se_pytest_origin.py	has_cycle	10	0	0	100%
se_pytest_origin.py	detect_cycle	16	0	0	100%
se_pytest_origin.py	main	48	1	0	98%
se_pytest_origin.py	(no function)	6	1	0	83%
se_pytest_test.py	test_detect_cycle	1	0	0	100%
se_pytest_test.py	test_has_cycle	3	0	0	100%
se_pytest_test.py	test_main	4	0	0	100%
se_pytest_test.py	(no function)	14	1	0	93%
SE.py	has_cycle	10	10	0	0%
SE.py	detect_cycle	16	16	0	0%
SE.py	(no function)	58	58	0	0%
Total		351	252	0	28%

Coverage report: 13%

Files Functions Classes

coverage.py v7.5.1, created at 2024-05-19 04:42 +0800

File	class	statements	missing	excluded	coverage
se_profile1.py	(no class)	6	6	0	0%
se_pylint.py	(no class)	59	59	0	0%
se_pytest_origin.py	(no class)	6	1	0	83%
se_pytest_test.py	(no class)	14	1	0	93%
SE.py	(no class)	58	58	0	0%
Total		143	125	0	13%

因为 coverage 是对当前文件目录下所有 .py 文件都进行了覆盖率测试,但其实只有 se_pytest_test.py 和 se_pytest_origin.py 两个文件在单元测试中被使用,其他文件并没有参与到单元测试中。TOTAL 统计了平均的覆盖其实在这里没有什么参考价值(当然如果把其他没用到的文件挪到其他位置就是另一说了)如果只看参与到单元测试的两个.py 文件,语句覆盖率都超过了 95%,也就是只有 1-2 个语句没有被覆盖到。函数覆盖率和分支覆盖率也都在 90% 以上,大部分函数的函数覆盖率可以达到 100%。

2.5 代码优化

2.5.1 代码分析

- 编程规范

该代码遵循了 Python 的 PEP 8 编程规范,包括了清晰且良好的命名规范(变量和函数名使用小写字母,单词间使用下划线分隔),函数和类的文档字符串,代码缩进、代码注释、缩进等。

在根据检查是否遵守编程规范的 pylint 静态分析后,已根据识别出的不规范的部分完成了全部的修改。

- 可扩展性

目前代码的可扩展性较好,因为使用了函数来模块化不同的功能,例如 has_cycle 函数用于检测环路, detect_cycle 函数用于检测整个电路是否存在环路, main 函数用于处理输入和输出。

提高代码可扩展性的一种方法是使用面向对象的编程方式。将功能封装到类中，每个类负责一个特定的功能模块，使得代码更加模块化和可维护。

引入配置文件或者参数化设置，使得代码适应不同的环境或需求。例如，将设备的类型和功能定义为配置参数，而不是硬编码在代码中。

使用适当的数据结构和算法，以应对可能的数据扩展和变化。例如，使用图数据结构来表示电路连接关系，而不是仅仅使用字典。

- 错误与异常处理

当前代码中并未包含显式的错误和异常处理，可以通过添加 try-except 块来捕获可能出现的异常，并对其进行处理或记录日志，以增强代码的鲁棒性。

对于输入数据的格式错误或无效数据，可以添加适当的验证和处理逻辑，以提供友好的错误提示。

- 算法复杂度

detect_cycle 函数中使用了深度优先搜索算法来检测图中是否存在环路。该算法的时间复杂度为 $O(V+E)$ ，其中 V 是顶点数， E 是边数。因此，算法的复杂度取决于电路的规模和连接关系。

- 性能分析与代码优化:

深度优先搜索算法的时间复杂度已经是较低的了，因此在性能上没有太多的优化空间。

当前代码可能存在性能瓶颈，是在处理大规模电路时。可以通过以下方式进行性能分析和优化：

- 使用合适的数据结构和算法来提高检测环路的效率。例如，可以考虑使用并查集等数据结构来优化图的表示和环路检测算法。
- 对循环中的关键部分进行性能测试和优化。特别是在计算器件的输出时，可能存在重复计算的情况，可以通过缓存计算结果来减少重复计算。
- 使用并行计算或异步编程技术来提高程序的并发性和响应性，特别是对于大规模输入数据的处理。

2.5.2 代码优化

根据上面的分析，下面主要需要处理上述代码的错误与异常处理部分和可扩展性。

根据作业要求中对不同输入取值范围和数量的制约关系，以及提升可扩展性的方法，为现有代码增加错误和异常处理如下：

```

1
2  """
3  Circuit Simulator Module
4
5  This module provides classes and functions to simulate logic circuits with various types of logic gates.
6  It supports AND, OR, XOR, NAND, NOR, and NOT gates. The main function processes user input to determine
7  the circuit and runs simulations based on the input values and output requirements.
8
9  Classes:
10     LogicGate - Base class for logic gates

```

```

11     ANDGate - AND logic gate
12     ORGate - OR logic gate
13     XORGate - XOR logic gate
14     NANDGate - NAND logic gate
15     NORGate - NOR logic gate
16     NOTGate - NOT logic gate
17     CircuitSimulator - Class for simulating logic circuits
18
19 Functions:
20     main - Main function to handle input and output
21     process_question - Process each question, read inputs, and run the simulation
22     add_devices_to_simulator - Add devices to the circuit simulator
23     read_all_inputs - Read input values for all runs
24     read_all_output_devices - Read output device configurations for all runs
25     """
26
27 class LogicGate:
28     """Base class for logic gates."""
29
30     def __init__(self, func, inputs):
31         """Initialize the gate with a function and input ports."""
32         self.func = func
33         self.inputs = inputs
34
35     def evaluate(self, inputs, outputs):
36         """Evaluate the gate's output based on inputs and previous outputs."""
37         raise NotImplementedError
38
39     def get_value(self, port, inputs, outputs):
40         """Get the value from an input or output port."""
41         if port[0] == 'I':
42             return inputs[int(port[1:]) - 1]
43         if port[0] == 'O':
44             return outputs[int(port[1:])]
45         raise ValueError(f"Invalid port: {port}")
46
47
48 class ANDGate(LogicGate):
49     """AND logic gate."""
50
51     def evaluate(self, inputs, outputs):
52         in_values = [self.get_value(port, inputs, outputs) for port in self.inputs]

```

```
53         return int(all(in_values))
54
55
56 class ORGate(LogicGate):
57     """OR logic gate."""
58
59     def evaluate(self, inputs, outputs):
60         in_values = [self.get_value(port, inputs, outputs) for port in self.inputs]
61         return int(any(in_values))
62
63
64 class XORGate(LogicGate):
65     """XOR logic gate."""
66
67     def evaluate(self, inputs, outputs):
68         in_values = [self.get_value(port, inputs, outputs) for port in self.inputs]
69         return int(sum(in_values) % 2)
70
71
72 class NANDGate(LogicGate):
73     """NAND logic gate."""
74
75     def evaluate(self, inputs, outputs):
76         in_values = [self.get_value(port, inputs, outputs) for port in self.inputs]
77         return int(not all(in_values))
78
79
80 class NORGate(LogicGate):
81     """NOR logic gate."""
82
83     def evaluate(self, inputs, outputs):
84         in_values = [self.get_value(port, inputs, outputs) for port in self.inputs]
85         return int(not any(in_values))
86
87
88 class NOTGate(LogicGate):
89     """NOT logic gate."""
90
91     def evaluate(self, inputs, outputs):
92         in_values = [self.get_value(port, inputs, outputs) for port in self.inputs]
93         return int(not in_values[0])
94
```

```
95
96 class CircuitSimulator:
97     """Class for simulating logic circuits."""
98
99     def __init__(self):
100         """Initialize the circuit simulator with supported gate classes."""
101         self.devices = {}
102         self.device_classes = {
103             'AND': ANDGate,
104             'OR': ORGate,
105             'XOR': XORGate,
106             'NAND': NANDGate,
107             'NOR': NORGate,
108             'NOT': NOTGate
109         }
110
111     def add_device(self, device_id, func, inputs):
112         """Add a logic device to the circuit."""
113         if func not in self.device_classes:
114             raise ValueError(f"Unsupported device function: {func}")
115         self.devices[device_id] = self.device_classes[func](func, inputs)
116
117     def detect_cycle(self):
118         """Detect if there is a cycle in the circuit."""
119         try:
120             graph = {device_id: [] for device_id in self.devices}
121             for device_id, device in self.devices.items():
122                 for port in device.inputs:
123                     if port[0] == '0':
124                         neighbor = int(port[1:])
125                         graph[neighbor].append(device_id)
126             visited = {device_id: False for device_id in self.devices}
127             stack = {device_id: False for device_id in self.devices}
128
129             for device_id in self.devices:
130                 if not visited[device_id]:
131                     if self._has_cycle(graph, device_id, visited, stack):
132                         return True
133             return False
134         except ValueError as val_err:
135             print(f"ValueError in detect_cycle: {val_err}")
136             return False
```

```
137     except KeyError as key_err:
138         print(f"KeyError in detect_cycle: {key_err}")
139         return False
140
141 def _has_cycle(self, graph, vertex, visited, stack):
142     """Helper method to detect a cycle starting from a given vertex."""
143     try:
144         visited[vertex] = True
145         stack[vertex] = True
146         for neighbor in graph[vertex]:
147             if not visited[neighbor]:
148                 if self._has_cycle(graph, neighbor, visited, stack):
149                     return True
150             elif stack[neighbor]:
151                 return True
152         stack[vertex] = False
153         return False
154     except ValueError as val_err:
155         print(f"ValueError in _has_cycle: {val_err}")
156         return False
157     except KeyError as key_err:
158         print(f"KeyError in _has_cycle: {key_err}")
159         return False
160
161 def run(self, all_inputs, all_output_devices):
162     """Run the simulation for a set of inputs and output devices."""
163     try:
164         all_results = []
165         for inputs, output_devices in zip(all_inputs, all_output_devices):
166             outputs = self._evaluate_circuit(inputs)
167             result = [outputs.get(dev, 0) for dev in output_devices[1:]]
168             all_results.append(result)
169         return all_results
170     except ValueError as val_err:
171         print(f"ValueError in run: {val_err}")
172         return []
173     except KeyError as key_err:
174         print(f"KeyError in run: {key_err}")
175         return []
176
177 def _evaluate_circuit(self, inputs):
178     """Evaluate the entire circuit for a given set of inputs."""
```

```

179         outputs = {}
180         for i in range(1, len(self.devices) + 1):
181             device = self.devices[i]
182             outputs[i] = device.evaluate(inputs, outputs)
183         return outputs
184
185
186 def main():
187     """Main function to handle input and output."""
188     try:
189         num_questions = int(input("Please enter the number of questions: "))
190         for _ in range(num_questions):
191             process_question()
192     except ValueError as val_err:
193         print(f"ValueError running the program: {val_err}")
194     except Exception as exc:
195         print(f"Error running the program: {exc}")
196
197
198 def process_question():
199     """Process each question, read inputs, and run the simulation."""
200     try:
201         num_inputs, num_devices = map(int, input("Please enter the number of inputs and devices: ").split())
202         if num_inputs <= 0 or num_devices <= 0:
203             raise ValueError(
204                 "The number of input signals must be non-negative and the number of devices must be non-negative."
205             )
206
207         simulator = CircuitSimulator()
208         add_devices_to_simulator(simulator, num_devices)
209
210         if simulator.detect_cycle():
211             print("LOOP")
212             return
213
214         num_runs = int(input("Please enter the number of runs: "))
215         if num_runs <= 0:
216             raise ValueError("The number of runs must be a positive integer.")
217
218         all_inputs = read_all_inputs(num_runs, num_inputs)
219         all_output_devices = read_all_output_devices(num_runs)
220

```

```

221         results = simulator.run(all_inputs, all_output_devices)
222         for result in results:
223             print(*result)
224     except ValueError as val_err:
225         print(f"Input error: {val_err}")
226     except KeyError as key_err:
227         print(f"KeyError processing the circuit: {key_err}")
228     except Exception as exc:
229         print(f"Error processing the circuit: {exc}")
230
231
232 def add_devices_to_simulator(simulator, num_devices):
233     """Add devices to the circuit simulator."""
234     for i in range(1, num_devices + 1):
235         device_info = input(f"Please enter the information for device {i}: ").split()
236         func = device_info[0]
237         num_inputs = int(device_info[1])
238         if num_inputs < 1:
239             raise ValueError(f"The number of inputs for device {i} must be at least 1.")
240         inputs = device_info[2:]
241         if len(inputs) != num_inputs:
242             raise ValueError(f"The number of inputs for device {i} does not match.")
243         simulator.add_device(i, func, inputs)
244
245
246 def read_all_inputs(num_runs, num_inputs):
247     """Read input values for all runs."""
248     all_inputs = []
249     for i in range(1, num_runs + 1):
250         inputs = list(map(int, input(f"Please enter the input values for run {i}: ").split()))
251         if len(inputs) != num_inputs:
252             raise ValueError(f"The number of input values for run {i} is incorrect.")
253         all_inputs.append(inputs)
254     return all_inputs
255
256
257 def read_all_output_devices(num_runs):
258     """Read output device configurations for all runs."""
259     all_output_devices = []
260     for i in range(1, num_runs + 1):
261         output_devices = list(map(int, input(f"Please enter the device numbers and corresponding
262         num_outputs = output_devices[0]

```

```

263         if len(output_devices[1:]) != num_outputs:
264             raise ValueError(f"The number of output devices for run {i} is incorrect.")
265         all_output_devices.append(output_devices)
266     return all_output_devices
267
268
269 if __name__ == "__main__":
270     main()
271
272

```

使用如下方式处理错误和异常：

- 在 `has_cycle` 和 `detect_cycle` 函数中，使用 `try-except` 块捕获和处理可能的异常。这些函数负责检测电路中的循环，并可能会遇到各种异常情况，例如访问不存在的键或类型错误。
- 在 `main` 函数的顶层，使用一个大的 `try-except` 块来捕获整个程序运行中的所有异常，确保任何未捕获的异常都会被打印出来，而不会导致程序崩溃
- 在每个输入步骤前，检查输入值的合法性，并在出现问题时抛出 `ValueError`。如检查电路输入和设备数量，输入的数量，运行次数，每次运行的输入值数量，每次运行的输出设备数量等。

采用了以下方式提高当前代码的可扩展性：

- 模块化代码：将代码分成多个函数，每个函数负责一个独立的功能。
- 使用类封装逻辑：将电路模拟器逻辑封装到类中，以便于扩展和维护。
- 添加更多的逻辑门支持：通过定义一个逻辑门的基类和具体的逻辑门子类，使得添加新的逻辑门类型变得更容易。
- 配置驱动：将输入格式解析、逻辑门定义等配置分离出来，便于修改和扩展。

通过输入样例 1 中的输入，得到了正确的输出。

```

PS C:\Users\LENOVO\Desktop> python se_pro.py
Please enter the number of questions: 1
Please enter the number of inputs and devices: 3 5
Please enter the information for device 1: XOR 2 I1 I2
Please enter the information for device 2: XOR 2 O1 I3
Please enter the information for device 3: AND 2 O1 I3
Please enter the information for device 4: AND 2 I1 I2
Please enter the information for device 5: OR 2 O3 O4
Please enter the number of runs: 4
Please enter the input values for run 1: 0 1 1
Please enter the input values for run 2: 1 0 1
Please enter the input values for run 3: 1 1 1
Please enter the input values for run 4: 0 0 0
Please enter the device numbers and corresponding signal quantities to output for run 1: 2 5 2
Please enter the device numbers and corresponding signal quantities to output for run 2: 2 5 2
Please enter the device numbers and corresponding signal quantities to output for run 3: 2 5 2
Please enter the device numbers and corresponding signal quantities to output for run 4: 2 5 2
1 0
1 0
1 1
0 0

```

pylint 静态分析结果如下：


```

PS C:\Users\LENOVO\Desktop> pylint se_pro.py
***** Module se_pro
se_pro.py:4:0: C0301: Line too long (104/100) (line-too-long)
se_pro.py:200:0: C0301: Line too long (108/100) (line-too-long)
se_pro.py:203:0: C0301: Line too long (120/100) (line-too-long)
se_pro.py:260:0: C0301: Line too long (150/100) (line-too-long)
se_pro.py:193:11: W0718: Catching too general exception Exception (broad-exception-caught)
se_pro.py:227:11: W0718: Catching too general exception Exception (broad-exception-caught)

-----
Your code has been rated at 9.64/10 (previous run: 9.46/10, +0.18)

```

profile 性能分析结果如下:

```

PS C:\Users\LENOVO\Desktop> python -m cProfile -o C:\Users\LENOVO\Desktop\se_pro_profile.prof C:\Users\LENOVO\Desktop\se_pro.py
Please enter the number of questions: 1
Please enter the number of inputs and devices: 3 5
Please enter the information for device 1: XOR 2 I1 I2
Please enter the information for device 2: XOR 2 O1 I3
Please enter the information for device 3: AND 2 O1 I3
Please enter the information for device 4: AND 2 I1 I2
Please enter the information for device 5: OR 2 O3 O4
Please enter the number of runs: 4
Please enter the input values for run 1: 0 1 1
Please enter the input values for run 2: 1 0 1
Please enter the input values for run 3: 1 1 1
Please enter the input values for run 4: 0 0 0
Please enter the device numbers and corresponding signal quantities to output for run 1: 2 5 2
Please enter the device numbers and corresponding signal quantities to output for run 2: 2 5 2
Please enter the device numbers and corresponding signal quantities to output for run 3: 2 5 2
Please enter the device numbers and corresponding signal quantities to output for run 4: 2 5 2
1 0
1 0
1 1
0 0
PS C:\Users\LENOVO\Desktop>

```

```

PS C:\Users\LENOVO\Desktop> python -m pstats
Welcome to the profile statistics browser.
%
PS C:\Users\LENOVO\Desktop> python -m pstats C:\Users\LENOVO\Desktop\se_pro_profile.prof
Welcome to the profile statistics browser.
C:\Users\LENOVO\Desktop\se_pro_profile.prof% sort cumtime
C:\Users\LENOVO\Desktop\se_pro_profile.prof% stats
Sun May 19 10:08:43 2024 C:\Users\LENOVO\Desktop\se_pro_profile.prof

229 function calls (226 primitive calls) in 3.392 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.000    0.000    3.392    3.392  {built-in method builtins.exec}
1      0.000    0.000    3.392    3.392  C:\Users\LENOVO\Desktop\se_pro.py:1(<module>)
1      0.000    0.000    3.392    3.392  C:\Users\LENOVO\Desktop\se_pro.py:185(main)
16     3.391    0.212    3.391    0.212  {built-in method builtins.input}
1      0.000    0.000    1.768    1.768  C:\Users\LENOVO\Desktop\se_pro.py:197(process_question)
1      0.001    0.001    1.765    1.765  C:\Users\LENOVO\Desktop\se_pro.py:256(read_all_output_devices)
1      0.000    0.000    0.001    0.001  C:\Users\LENOVO\Desktop\se_pro.py:221(add_devices_to_simulator)
1      0.000    0.000    0.001    0.001  C:\Users\LENOVO\Desktop\se_pro.py:245(read_all_inputs)
4      0.000    0.000    0.000    0.000  {built-in method builtins.print}
1      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:160(run)
4      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:176(_evaluate_circuit)
8      0.000    0.000    0.000    0.000  {built-in method builtins.__build_class__}
8      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:66(evaluate)
1      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:116(detect_cycle)
8      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:50(evaluate)
5      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:110(add_device)
40     0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:38(get_value)
8      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:67(<listcomp>)
8      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:51(<listcomp>)
4      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:58(evaluate)
14     0.000    0.000    0.000    0.000  {method 'split' of 'str' objects}
4      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:59(<listcomp>)
4      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:166(<listcomp>)
17     0.000    0.000    0.000    0.000  {built-in method builtins.len}
5/2    0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:140(_has_cycle)
1      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:98(__init__)
5      0.000    0.000    0.000    0.000  C:\Users\LENOVO\Desktop\se_pro.py:29(__init__)

```

可以看到性能有一定的提升。