逻辑综合

1. 项目说明

1.1 项目介绍

该项目使用phyLS工具，通过Mockturtle库来管理和优化逻辑网络。通过提供电路等级优化balance与电路重写rewrite算法，来支持该框架通过对AIG的电路网络进行优化。优化主要考虑主要功能包括节点和信号的操作，旨在提高电路的“功耗、延迟、面积”方面的性能。

1.1.1 组内分工

- 姚晓辉：rewrite代码实现
- 徐一丁：balance代码实现
- 刘致依：算法原理分析，以及报告撰写

1.1.2 工具使用方法

以下流程在系统版本为Ubuntu 24.04 LTS的Linux系统下，编译器以及相应工具链的版本如下的环境下可以成功运行PhyLS工具和相应的balance和rewrite优化算法。

- g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
- gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
- clang version 14.0.0-1ubuntu1.1
- Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]

1.1.3 编译

- 在文件目录/phyLS/src/core下添加balance.hpp，rewrite.hpp文件
- 在/phyLS/build位置中运行
    - cmake ..
    - make
- 进入phyLS逻辑综合工具
    - ./bin/phyLS

1.1.4 运行

- 测试文件
    - read_aiger ../benchmarks/adder.aig
    - ps -a
- 两步优化
    - balance
    - rewrite

2. balance()设计说明

2.1 设计原理

2.1.1 k-可行割集

描述从逻辑网络中的一个给定节点（根节点）到其输入（通常是原始输入或其他逻辑门）的一个子集，这个子集被称为“切割”的叶子。

定义和用途

K切割是指一个节点的一个子图，这个子图包括：

- **根节点**：要进行切割的目标节点。
- **叶节点**：根节点的输入节点集合，这些节点直接或间接地提供输入到根节点。这个集合的大小被限制在K个或K个以下，这就是“K”在K切割中的含义。

这种切割的主要目的是识别出一个可以独立优化的网络部分，从而通过优化这一小部分来改善整个网络的性能（延迟、面积）。

考虑一个简单的逻辑表达式，根节点是一个大型的AND门，其输入是来自多个其他逻辑门的输出。

在之后的代码中，我们**遍历这个AND门节点执行K切割**，选择其最重要的几个输入作为叶节点，集中优化这些部分，而不是整个逻辑网络，如果当前处理的切割使得节点的AIG层级比已知的最佳切割还要小，则将其保存为最佳切割。最终选出最优的切割优化方式来作为本次更新的迭代结果。

2.1.2 AGI网络与SOP网络

AIG（And-Inverter Graph）

AIG使用“与”（AND）和反相器（非门），其优势包括：

- **简洁性**：AIG通过仅使用AND门和反相器来表示复杂的逻辑函数，这种简化使得逻辑操作和优化更加高效，也因此形式验证和逻辑合成中，AIG允许进行快速的布尔操作，尤其是等价检验和逻辑简化。

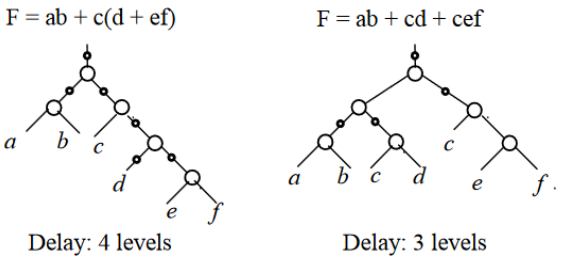
SOP（Sum of Products）

SOP通过“与”（AND）逻辑门的“和”（OR）来表达布尔函数，具体优势包括：

- **优化透明度**：在某些情况下，使用SOP形式可以更直观地看到优化的空间和途径，通过合并项来减少逻辑复杂度。
- **适用于小规模优化**：对于小规模的电路或者逻辑表达，SOP形式可以直接展示所有可能的简化和重构方式。

大型AIG的SOP平衡算法

因为AIG的逻辑简洁表示，大型网络通常选择AIG的表示方式。AIG形式网络输入后，本算法通过切割找到逻辑优化小窗口。然后对小型的电路转化为SOP网络，因为SOP表示可以优化AIG无法进行优化的小型电路（如下图所示，该逻辑无法动过AIG网络进行优化），对其进行除过And-balancing之外的SOP-balancing的优化。



2.1 设计思路

`run` 函数的执行逻辑可能包括以下几个主要步骤：

1. **初始化和准备工作**：
   - 清理或重置与AIG结构相关的状态或值，确保从节点目前的参数是更新后的目前状态开始处理。
2. **遍历处理**：
   - 通常会遍历AIG中的每个节点或特定的关键节点，并对每个节点执行一系列操作。
3. **节点处理**：
   - 对每个节点，可能需要计算某些特性，如K切割（K-feasible cuts），到确定能最有效表示节点功能的节点集合。
   - 对于每个节点或切割，执行进一步的逻辑优化处理，这可能包括生成和优化SOP表达式，或者直接在AIG上应用AND平衡技术。
4. **应用优化结果**：
   - 根据每个节点的处理结果，更新AIG的结构。添加、删除或重新连接节点，以实现更优的逻辑结构。
5. **最终更新**：
   - 在所有节点处理完毕后，执行一些最终的清理和更新操作，确保所有的优化更改都正确地反映在AIG结构上。

```
void run() {
    clear_values();  // 清理所有节点的临时值或标记
    foreach node in AIG {  // 遍历AIG的每个节点
        if (node is critical) {  // 只处理关键节点
            auto cuts = compute_cuts(node);  // 计算节点的K切割
            foreach cut in cuts {
                auto sop = generate_SOP(cut);  // 为每个切割生成SOP
                auto optimized_sop = optimize_SOP(sop);  // 优化SOP结构
                if (is_better(optimized_sop, node)) {  // 如果找到了更好的优化方案
                    update_AIG_structure(node, optimized_sop);  // 更新AIG结构
                }
            }
        }
    }
    finalize_updates();  // 执行最终的结构更新
}
```

2.2 主要算法

(1) **collect leaves of the AND tree**:

- **函数**: `collect_leaves`
- **描述**: 这个函数通常遍历节点的所有直接或间接输入（叶节点），收集那些对当前节点的逻辑实现至关重要的输入节点。它可能涉及到递归调用，尤其是在处理复杂的组合逻辑结构时。

```
template <class Ntk, class signal, class node>
void collect_leaves(Ntk& ntk, node const& n, std::vector<signal>&
leaves) {
    ntk.incr_trav_id();

    int ret = collect_leaves_rec<Ntk, signal, node>(ntk,
ntk.make_signal(n), leaves, true);

    /* check for constant false */
    if(ret < 0) {
        leaves.clear();
    }
}
```

(2) **recur over the leaves**:

- **函数**: `balance_rec`
- **描述**: 对每个叶节点递归执行平衡操作，这通常包括对每个节点调用同样的平衡算法，以确保整个逻辑树都得到适当处理。

```
template <class Ntk, class signal, class node>
signal balance_rec(Ntk& ntk, node const& n, uint32_t level,
std::vector<std::vector<signal>>& storage) {
    if(ntk.is_ci(n))
        return ntk.make_signal(n);

    /* node has been replaced in a previous recursion */
    if(ntk.is_dead(n) || ntk.value(n) > 0) {
        return ntk.make_signal(find_substituted_node<Ntk, signal, node>
(ntk, n));
    }

    if(level >= storage.size()) {
        storage.emplace_back(std::vector<signal>());
        storage.back().reserve(10);
    }

    /* collect leaves of the AND tree */
    collect_leaves<Ntk, signal, node>(ntk, n, storage[level]);

    if(storage[level].size() == 0) {
        ntk.substitute_node(n, ntk.get_constant(false));
        return ntk.get_constant(false);
    }

    /* recur over the leaves */
```

```
    for(auto& f : storage[level]) {
        signal new_signal = balance_rec<Ntk, signal, node>(ntk, ntk.get_node(f), level + 1, storage);
        f = new_signal ^ ntk.is_complemented(f);
    }

    assert(storage[level].size() > 1);

    /* sort by decreasing level */
    std::sort(storage[level].begin(), storage[level].end(), [&](auto const& a, auto const& b) {
        return ntk.level(ntk.get_node(a)) > ntk.level(ntk.get_node(b));
    });

    /* mark TFI cone of n */
    ntk.incr_trav_id();
    mark_tfi<Ntk, signal, node>(ntk, ntk.make_signal(n), true);

    /* generate the AND tree */
    while(storage[level].size() > 1) {
        /* explore multiple possibilities to find logic sharing */
        pick_nodes<Ntk, signal, node>(ntk, storage[level], find_left_most_at_level<Ntk, signal, node>(ntk, storage[level]));

        /* pop the two selected nodes to create the new AND gate */
        signal child1 = storage[level].back();
        storage[level].pop_back();
        signal child2 = storage[level].back();
        storage[level].pop_back();
        signal new_sig = ntk.create_and(child1, child2);

        /* update level for AND node */
        update_level<Ntk, signal, node>(ntk, ntk.get_node(new_sig));

        /* insert the new node back */
        insert_node_sorted<Ntk, signal, node>(ntk, storage[level], new_sig);
    }

    signal root = storage[level][0];

    /* replace if new */
    if(n != ntk.get_node(root)) {
        ntk.substitute_node(n, root);
    }

    /* remember the substitution and the new node as already balanced */
    ntk.set_value(n, ntk.node_to_index(ntk.get_node(root)));
    ntk.set_value(ntk.get_node(root), ntk.node_to_index(ntk.get_node(root)));

    /* clean leaves storage */
    storage[level].clear();

    return root;
}
```

(3) **sort by decreasing level**:

- **函数**：`insert_node_sorted`
- **描述**：根据节点的逻辑层级进行排序，通常是从最高到最低（或从最低到最高，取决于具体实现），以便优化处理顺序或数据结构的管理。

```
template <class Ntk, class signal, class node>
void insert_node_sorted(Ntk& ntk, std::vector<signal>& leaves, signal const& f) {
    node n = ntk.get_node(f);

    /* check uniqueness */
    for(auto const& s : leaves) {
        if(s == f)
            return;
    }

    leaves.push_back(f);
    for(size_t i = leaves.size() - 1; i > 0; --i) {
        auto& s2 = leaves[i - 1];

        if(ntk.level(ntk.get_node(s2)) < ntk.level(n)) {
            std::swap(s2, leaves[i]);
        }
        else {
            break;
        }
    }
}
```

(4)**mark TFI cone of n**:

- **函数**：ntk.incr_trav_id()
- **描述**：ntk库中函数，标记从输入到节点n的传递函数影响锥（TFI Cone），这涉及到识别和标记所有可以影响到节点n输出的节点。

(5)**generate the AND tree**:

- **函数**：`generate_and_tree`（被集成在balance_rec)中
- **描述**：基于收集和排序的叶节点生成新的AND树，这个步骤是重新构建优化后的逻辑结构的核心部分。

```
/* generate the AND tree */
    while(storage[level].size() > 1) {
        /* explore multiple possibilities to find logic sharing */
        pick_nodes<Ntk, signal, node>(ntk, storage[level], find_left_most_at_level<Ntk, signal, node>(ntk, storage[level]));

        /* pop the two selected nodes to create the new AND gate */
        signal child1 = storage[level].back();
        storage[level].pop_back();
        signal child2 = storage[level].back();
```

```
        storage[level].pop_back();
        signal new_sig = ntk.create_and(child1, child2);

        /* update level for AND node */
        update_level<Ntk, signal, node>(ntk, ntk.get_node(new_sig));

        /* insert the new node back */
        insert_node_sorted<Ntk, signal, node>(ntk, storage[level],
new_sig);
    }

    signal root = storage[level][0];
```

(6)**replace if new**:

- **函数**：`replace_node`（包含在balance_rec中）
- **描述**：如果新生成的AND树与原有结构不同，替换原有节点。这通常涉及到在逻辑网络中更新节点的连接和功能实现。

```
/* replace if new */
    if(n != ntk.get_node(root)) {
        ntk.substitute_node(n, root);
    }
```

(7)**remember the substitution and the new node as already balanced**:

- **函数**：`update_balanced_status`（包含在balance_rec中）
- **描述**：记录替换和新节点的状态，标记它们为已平衡，这有助于避免重复处理和优化性能。

```
/* remember the substitution and the new node as already balanced */
    ntk.set_value(n, ntk.node_to_index(ntk.get_node(root)));
    ntk.set_value(ntk.get_node(root), ntk.node_to_index(ntk.get_node(root)));

    /* clean leaves storage */
    storage[level].clear();

```

2.3 数据结构定义

在Mockturtle库中，

- **网络**（Network）是构成逻辑电路的基本数据结构，支持不同类型的逻辑门和连接。
- **节点**（Node）是网络中的一个元素，代表一个的操作点，可以是常量、主输入或逻辑门。
- **信号**（Signal）是节点的抽象表示，用于表示节点的输入和输出连接。可以视为指向一个节点的指针，或者是节点向其扇出的一个输出边。根据逻辑网络的类型，信号可能包含额外信息，如补码属性，在表示相应的AIG、MIG、XAG等多种网络类型的实现都依赖于信号的定义。

```
struct signal
  {
    signal() = default;

    signal( uint64_t index, uint64_t complement )
        : complement( complement ), index( index )
```

```
    {
    }

    explicit signal( uint64_t data )
        : data( data )
    {
    }

    signal( aig_storage::node_type::pointer_type const& p )
        : complement( p.weight ), index( p.index )
    {
    }

```

3. rewrite()设计说明

2.1 设计原理

3.1.1 NPN等价系统

两个布尔函数F和G如果可以通过对输入进行否定（N）、排列（P）和对输出进行否定（N），从而互相转换，那么这两个函数属于同一个NPN类。

3.1.2 二级子图与4可行切割

4-可行切割：

- 这指的是在处理逻辑电路优化时，选择的一个局部区域或子图包含的节点（或门）数量不超过4个。这样的切割能够保持处理的复杂性在可控范围内，同时允许足够的灵活性来进行有效的优化。
- 切割沿用'PI'和'PO'的相关限制。

二级子图：

- 通常指的是在逻辑网络中，直接与输出节点相连的仅包含两层逻辑门（如与门、或门等）的子图。优势包括逻辑简单，可以进行更粗略和更直接的

基于四可行切割的重写算法

对每个节点执行重写：

- 对于节点N的每一个4-可行切割C：
  - 计算以切割C的叶子为变量的布尔函数F。
  - 查找所有与布尔函数F属于同一个NPN类的预计算子图结构。
  - 对于每个可能的子图结构S：
    - 计算替换前后的节点差异，以确定是否替换能带来结构优化（如减少节点数量）。

3.2 设计思路

1. **确定函数功能**
   - `Rewriting` 函数输入三个参数：`network AIG` 是待优化的网络，`hash table PrecomputedStructures` 是LUT，包含预先计算最优化的4可行切割的结构，`bool UseZeroCost` 决定是否接受零成本的重写（不增加节点数）。

- 输出优化重写网络AIG

2. **外层循环**:
   - 遍历AIG中的每个节点N，按拓扑顺序处理。这保证了每个节点在其依赖节点之后处理，符合逻辑运算的顺序。
3. **内层循环**:
   - 对于节点N的每个4输入切割C，计算其布尔函数F。这里使用的切割可能是优化重点，以减少复杂度和提高重写的有效性。
4. **查找可能的结构**:
   - 使用布尔函数F作为键，从`PrecomputedStructures`哈希表中查找可能的重写结构。
5. **最佳重写结构选择**:
   - 初始化`BestS`（最佳结构）为NULL，`BestGain`（最大收益）为-1。
   - 遍历所有可能的结构S，计算用S替换N后的节点节省（`NodesSaved`）和节点增加（`NodesAdded`）。
   - 更新引用：先取消引用S，再引用N，确保AIG的一致性。
   - 如果此次替换的收益`Gain`大于0，或者在允许零成本的情况下收益为0，则考虑更新最佳替换结构。
6. **应用最佳结构**:
   - 如果找到了合适的替换结构`BestS`，则再次计算用`BestS`替换`N`后的节点节省和增加，以更新网络列表。
   - 使用断言验证更新后的`Gain`确实是最优的。

逻辑判断

- **节点保存和增加的计算**:
  - `DereferenceNode`函数和`ReferenceNode`函数可能涉及更新节点的引用计数，以计算删除和添加节点的数量。
- **收益计算**:
  - `Gain`是通过`NodesSaved - NodesAdded`计算得出，表示此次替换能节省多少节点。
- **最佳替换判断**:
  - 如果当前替换比之前的替换更优（即`Gain`更高），则更新最佳替换结构`BestS`和最大收益 `BestGain
-

3.3主要算法

1. 初始节点映射

- **目标**：设置算法所需的初始条件，包括加载AIG网络，初始化统计数据等。
- **节点映射**：建立一个从原网络节点到新网络节点的映射。
- **处理常量和输入**：映射网络中的常量节点和输入端口。
- 函数实现：
  - `initialize_network`：加载并初始化AIG网络，设置必要的数据结构。

```
/* initial node map */
node_map<signal, Ntk> old2new(ntk);
    Ntk res;
    old2new[ntk.get_constant(false)] = res.get_constant(false);
    if(ntk.get_node(ntk.get_constant(true)) !=
ntk.get_node(ntk.get_constant(false))) {
        old2new[ntk.get_constant(true)] = res.get_constant(true);
    }
    ntk.foreach_pi([&](auto const& n) {
        old2new[n] = res.create_pi();
    });

```

- `load_precomputed_structures`：加载所有预先计算的子图和NPN类，可能通过哈希表进行存储。

2. 剪切枚举和成本估计

- **剪切枚举**：枚举网络中所有节点的剪切。
- **初始化引用计数**：使用扇出大小初始化引用计数。
- **原始成本**：计算原网络的成本。

```
const auto cuts = cut_enumeration<Ntk, true>(ntk, cut_enumeration_ps);
initialize_values_with_fanout(ntk);
const auto orig_cost = costs<Ntk, NodeCostFn>(ntk);
```

3. 对每个节点执行重写

- **目标**：对每个节点尝试找到最优的子图替换，以减少总节点数并优化AIG的结构。
- 函数实现：
  - `enumerate_cuts(node)`：为给定节点枚举所有4-可行切割

    ```
    const auto cuts = cut_enumeration<Ntk, true>(ntk,
    cut_enumeration_ps);
    ```

  - `compute_boolean_function(cut)`：通过哈希表查找与布尔函数相对应的NPN类及其预计算子图,计算给定切割的布尔函数。

```
/* foreach cut */
            int32_t best_gain = -1;
            signal best_signal;
            for(auto& cut : cuts.cuts(ntk.node_to_index(n))) {
                /* skip small enough cuts */
                if(cut->size() == 1 || cut->size() < min_cand_cut_size)
                    continue;

                const auto tt = cuts.truth_table(*cut);
                assert(cut->size() == static_cast<unsigned>(tt.num_vars()));

                std::vector<signal> children(cut->size());
                auto ctr = 0u;
```

4. 评估和选择替换

- **目标**：评估不同替换的效果，选择最优替换以改进AIG结构。
- 函数实现：
  - 对于节点和其可能的子图替换，评估每个替换的节点节省和增加。
  - 从所有可能的替换中选择最佳的替换方案，基于最大的节点净增益。

```
 for(auto l : *cut) {
                    children[ctr++] = old2new[ntk.index_to_node(l)];
                }

                const auto on_signal = [&](auto const& f_new) {
                    auto value2 = recursive_ref<Ntk, NodeCostFn>(res, res.get_node(f_new));
                    recursive_deref<Ntk, NodeCostFn>(res, res.get_node(f_new));
                    int32_t gain = value - value2;

                    if(gain > 0 && gain > best_gain) {
                        best_gain = gain;
                        best_signal = f_new;
                    }

                    return true;
                };
    rewriting_fn(res, cuts.truth_table(*cut), children.begin(), children.end(), on_signal);
```

5. 应用替换并更新结构

- **目标**：应用选定的替换，更新AIG结构以及相关统计和引用计数。
- 函数实现：
  - 对选定节点应用最佳子图替换。
  - 更新引用计数，管理节点的添加和删除。

```
    if(best_gain == -1) {
        std::vector<signal> children(ntk.fanin_size(n));
        ntk.foreach_fanin(n, [&](auto const& f, auto i) {
            children[i] = ntk.is_complemented(f) ? res.create_not(old2new[f]) : old2new[f];
        });

        old2new[n] = res.clone_node(ntk, n, children);
    }
    else {
        old2new[n] = best_signal;
    }
```

6. 重复优化直至完成

- **目标**：根据设定的终止条件（如达到迭代次数或优化目标），重复执行上述步骤。
- 函数实现：
  - `repeat_optimization`控制算法重复执行，直到满足终止条件

```
/* nothing to optimize? */
        int32_t value = mffc_size<Ntk, NodeCostFn>(ntk, n);
        if(value == 1) {
            std::vector<signal> children(ntk.fanin_size(n));
            ntk.foreach_fanin(n, [&](auto const& f, auto i) {
                children[i] = ntk.is_complemented(f) ? res.create_not(old2new[f]) : old2new[f];
            });

            old2new[n] = res.clone_node(ntk, n, children);
        }
```

7. 输出和清理

- **目标**：输出优化后的AIG，并进行必要的清理操作。
- 函数实现：
  - 输出或保存优化后的AIG网络。
  - 释放使用的资源和内存。

```
/* create POs */
    ntk.foreach_po([&](auto const& f) {
        res.create_po(ntk.is_complemented(f) ? res.create_not(old2new[f]) : old2new[f]);
    });

    res = cleanup_dangling<Ntk>(res);

    /* new costs */
    if(costs<Ntk, NodeCostFn>(res) <= orig_cost) {
        ntk = res;
    }
```

4.测试结果

adder

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/adder.aig
phyLS> ps -a
AIG   i/o = 256/129   gates = 1020   level = 255
phyLS> balance
ntk   i/o = 256/129   gates = 1020   level = 255
phyLS> rewrite
ntk   i/o = 256/129   gates = 1020   level = 255
phyLS>  read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 639, area = 1849.00, delay = 204.90, power = 0.00
phyLS> quit
```

div

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/div.aig
phyLS> ps -a
AIG   i/o = 128/128   gates = 57247   level = 4372
phyLS> balance
ntk   i/o = 128/128   gates = 57156   level = 4372
phyLS> rewrite
ntk   i/o = 128/128   gates = 41460   level = 4403
phyLS> balance
ntk   i/o = 128/128   gates = 41455   level = 4373
phyLS> rewrite
ntk   i/o = 128/128   gates = 41402   level = 4373
phyLS>  read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 39620, area = 97439.00, delay = 3446.50, power = 0.00
phyLS> quit
```

log2

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/log2.aig
phyLS> ps -a
AIG   i/o = 32/32   gates = 32060   level = 444
phyLS> balance
ntk   i/o = 32/32   gates = 31886   level = 410
phyLS> rewrite
ntk   i/o = 32/32   gates = 29587   level = 397
phyLS>  read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 16045, area = 49424.00, delay = 276.70, power = 0.00
phyLS> quit
```

sin

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/sin.aig
phyLS> ps -a
AIG   i/o = 24/25   gates = 5416   level = 225
phyLS> balance
ntk   i/o = 24/25   gates = 5385   level = 186
phyLS> rewrite
ntk   i/o = 24/25   gates = 5201   level = 195
phyLS> balance
ntk   i/o = 24/25   gates = 5173   level = 184
phyLS> rewrite
ntk   i/o = 24/25   gates = 5157   level = 190
phyLS> balance
ntk   i/o = 24/25   gates = 5154   level = 184
phyLS>  read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 3359, area = 9654.00, delay = 134.10, power = 0.00
```

i2c

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/i2c.aig
phyLS> ps -a
AIG   i/o = 147/142   gates = 1342   level = 20
phyLS> balance
ntk   i/o = 147/142   gates = 1274   level = 16
phyLS> rewrite
ntk   i/o = 147/142   gates = 1251   level = 17
phyLS> balance
ntk   i/o = 147/142   gates = 1247   level = 16
phyLS> rewrite
ntk   i/o = 147/142   gates = 1243   level = 17
phyLS> balance
ntk   i/o = 147/142   gates = 1240   level = 16
phyLS>  read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 991, area = 2334.00, delay = 13.20, power = 0.00
```

multiplier

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/multiplier.aig
phyLS> ps -a
AIG   i/o = 128/128   gates = 27062   level = 274
phyLS> balance
ntk   i/o = 128/128   gates = 26953   level = 266
phyLS> rewrite
ntk   i/o = 128/128   gates = 24693   level = 264
phyLS> read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 11642, area = 40226.00, delay = 208.80, power = 0.00
phyLS> quit
```

hyp

```
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/hyp.aig
phyLS> ps -a
AIG   i/o = 256/128   gates = 214335   level = 24801
phyLS> read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 116061, area = 364855.00, delay = 16771.40, power = 0.0
0
phyLS> quit
nahida@nahida-virtual-machine:~/桌面/phyLS/build$ ./bin/phyLS
phyLS> read_aiger ../benchmarks/hyp.aig
phyLS> ps -a
AIG   i/o = 256/128   gates = 214335   level = 24801
phyLS> balance
ntk   i/o = 256/128   gates = 214335   level = 24801
phyLS> rewrite
ntk   i/o = 256/128   gates = 214007   level = 24905
phyLS> balance
ntk   i/o = 256/128   gates = 214007   level = 24801
phyLS> read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 116772, area = 365562.00, delay = 16771.50, power = 0.0
0
phyLS> quit
```

5. 总结

平衡算法：主要用于减少逻辑电路的级别（level）。通过平衡，电路的结构被调整为更加均衡，从而减少最大逻辑级别。例如，在处理 `i2c.aig` 时，级别从20降至16，显示出平衡操作有效地减少了电路的传播延迟。

重写算法：重写操作专注于减少门数，并可能进一步调整电路级别。例如，在 `sin.aig` 电路上，通过重写门数从5416降至5173，级别也有所改善。这表明重写算法在减少电路复杂性和提高效率方面具有显著效果。

rewrite和balance算法的应用显示出在不同类型的逻辑电路上，它们能有效地减少资源使用并优化电路性能，但是针对特殊电路比如adder（过小）或者hyp（过大）的电路没有表现出很好的性能。可能的原因有对于adder等加法器数据已经达到最优的基础模块，没有优化空间。而对于hyp可能是其平衡算法与重写算法两个算法方向相反，需要调整算法参数来引导优化方向。