# CAD Contest Problem A: Reinforcement Logic Optimization for a General Cost Function

112-2 IEDA Final Project

Chun-Kai Yang
*Dept. of Electrical Engineering*
B10901027
b10901027@ntu.edu.tw

Cheng-En Tsai
*Dept. of Electrical Engineering*
B10901098
b10901098@ntu.edu.tw

Ting-Hsuan Tien
*Dept. of Electrical Engineering*
B10901106
b10901106@g.ntu.edu.tw

***Abstract*—With the advancement of technology, the traditional optimization methods are no longer sufficient to address diverse optimization scenarios. Therefore, a more comprehensive approach is necessary, one that can be applied to a variety of cost criteria. To achieve this, we pre-process our library during mapping to effectively handle different synthesis cost criteria. This report also outlines our circuit optimization methods and compares the performance of Reinforcement Learning against other baselines, such as Greedy and Simulated Annealing, under conditions where the cost function is unknown.**

## I. Introduction

Established optimization metrics such as PPA (power, performance, area) are increasingly inadequate for addressing the diverse and complex optimization scenarios that arise in modern digital design stages. As systems grow in complexity and sophistication, relying solely on PPA metrics fails to capture the breadth of requirements needed for effective optimization and variant purposes. This has generated an urgent requirement for a more flexible and inclusive approach to optimization. To model this situation, we will receive a cost function estimator (a black box whose workings and criteria are unknown), and our task is to develop a program that performs optimizations to minimize the cost.

In this program, we first parse the provided cell library (.json) and create a netlist (.v) containing only single primitive gates. Using the cost returned from the cost estimator, we then build an optimized library (.lib). To optimize the circuit with Yosys and Yosys-ABC commands, we transform the original netlist (.v) into an AIG file. Finally, we implement a Reinforcement Learning algorithm to determine the optimal commands and compare its performance against other baseline algorithms, such as Greedy and Simulated Annealing.

The rest of the report is organized as follows: Section II discusses related work that informed the development of our RL algorithm. Section III presents a description and all requirements regarding the environment and I/O format. Section IV provides a detailed explanation of our RL method, while Section V describes other baseline methods. We present and discuss our results in Section VI. Finally, Section VII offers our conclusions and suggests potential future improvements.

## II. Related Work

**Deep Reinforcement Learning for Logic Synthesis [1]**

Recent advancements in logic synthesis have leveraged machine learning techniques to enhance the optimization process. One notable work in this area is the Deep Reinforcement Learning for Logic Synthesis (DRiLLS) framework. DRiLLS focuses on optimizing the delay and area of a given design (.blif) using reinforcement learning. By applying deep reinforcement learning algorithms, DRiLLS can learn effective optimization strategies from the synthesis process itself.

The framework demonstrates significant improvements in circuit optimization, reducing both the area and delay of synthesized circuits compared to traditional heuristic-based methods. DRiLLS highlights the potential of reinforcement learning to automate and enhance logic synthesis, paving the way for more efficient and effective design automation tools.

This work is particularly relevant to our research, as it underscores the importance of exploring advanced machine learning techniques for optimizing digital circuit design. Our approach similarly aims to utilize these techniques to improve the performance and efficiency of circuit synthesis processes.

However, there are several aspects that can be improved. For example, the action space chosen in the DRiLLS framework is quite limited, and there are many other effective commands for AIG optimization. Additionally, the state vector representing the AIG in DRiLLS is not optimal, as it contains too little information to fully represent an AIG graph, leading to significant information loss.

We have taken the structure of this work as a reference and implemented several enhancements to address these issues in our netlist optimization problem. Our approach aims to achieve more comprehensive and effective optimization results.

## III. Problem Formulation

### A. Problem Description

The task involves developing a program that optimizes a flattened Verilog netlist using a designated cell library and minimizes costs based on a provided estimator. The program should generate an optimized netlist that maintains functional equivalence while minimizing the cost.

### B. Program Requirements

- **Operating Environment**: The program must be compatible with Linux.
- **Time Constraints**: Each optimization session is limited to a maximum of 3 hours.
- **Parallel Processing**: The use of multiple threads or processes for computation is prohibited.
- **Command Line Interface**: The program should be executable via command line, accepting the following arguments:
  - `-netlist <netlist_path/name.v>`: Specifies the path to the flattened Verilog netlist.
  - `-library <lib_path/name.lib>`: Identifies the cell library detailing the specifications of each library cell.
  - `-cost_function <cost_func_path/name>`: Indicates the executable file serving as the cost function estimator.
  - `-output <output_path/name.v>`: Specifies the destination path for the optimized netlist.

## IV. Method

This section provides an explanation of the program's functionality. Subsection IV-A presents the overall structure and workflow of our work. Subsection IV-B details the process of generating a suitable library (.lib) for mapping AIG to netlist from the given library (.json). Subsection IV-C describes the conversion between AIG and netlist using the Yosys/Yosys-ABC tool. Subsection IV-D explains how we calculate the cost of the AIG using the provided cost estimator. Finally, Subsection IV-E focuses on the core of our work, which involves iteratively optimizing the AIG using Reinforcement Learning based on the circuit's cost.

### A. Framework Overview

Fig.1 is the flow chart of our program.

Before starting the optimization process, we first generate an optimized library (.lib). Based on the gates listed in the original given library, we build a few primitive gate netlists and send them to the black-box cost estimator. With the returned costs, we rank gates of the same type and complete the optimized library (.lib), which will be used later during mapping.

Next, we convert the given unprocessed netlist (.v) into an AIG file using the `subprocess` package in Python to execute `yosys` and `yosys-abc` binary files to utilize the commands. An optimization algorithm, such as reinforcement learning (RL) or another baseline method, is then applied to this AIG circuit. The optimized AIG circuit is mapped back

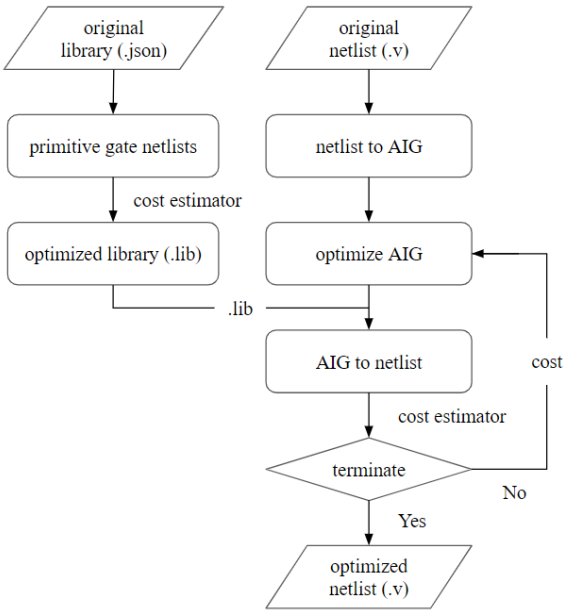


Fig. 1. Flow Chart

to the primitive gates netlist using the optimized library just built. The netlist is sent to the cost estimator and determined whether the process must terminate based on the optimization algorithm. If not, the cost and AIG circuit are updated, and the algorithm continues running until the optimized netlist with the minimum cost is acheived.

### B. Library Preprocessing

In this problem, we are provided with a library (.json) containing primitive gates, each having several different gate names. For example, the AND gate has multiple variants named `and_1` to `and_8`. Each gate name is associated with several numerical values representing specific costs for the circuits. For instance, `and_1` has associated data values from `data_1_f` to `data_7_f`.

As mentioned in our framework, we optimize the circuit in AIG format instead of Verilog. Therefore, we need to prepare a library to efficiently map our optimized AIG to a netlist (.v) with the lowest cost. To obtain such a library, we follow these steps for each gate type: we wrap all the gate names into independent netlists (.v) containing only primary I/O and the specific gate name, send each netlist (.v) to the given cost estimator (.exe) to determine the cost associated with each gate name, and then choose the gate name with the lowest cost to represent the corresponding gate type. For example, in our case, we use `and_6` to represent the AND gate. Thus, `and_6` will appear in the subsequent mapping whenever an AND gate is required.

After acquiring the best gate name for each gate type, we parse the set of optimized gates and the associated costs into a library (.lib) that can be successfully read by the ABC [2]/Yosys-ABC [3] tool. This allows us to directly use the mapping commands in Yosys/Yosys-ABC to map the AIG

(.aig) to a netlist (.v) with our optimized library (.lib). The algorithm for library generation is shown in Algorithm 1. With this step, we successfully reduce the cost of the primitive gates, which further reduces the cost of the entire netlist circuits after applying our subsequent optimization methods.

**Algorithm 1** Library Generation Algorithm

```
Input: Library (.json) with primitive gates
Output: Optimized library (.lib)
Parse the library (.json) into an improved structure (.json)
for each gate type in the library do
    for each gate name within the gate type do
        Generate a netlist (.v) for the gate name
        Use the cost_estimator (.exe) to obtain the cost
    end for
    Select the gate name with the lowest cost
    Update the gate type with the selected gate name
    Record the associated cost for library
end for
Parse the set of optimized pairs (gate_name, cost) into a library (.lib)
```

### C. Circuit Conversion

In this section, we implement the conversion between AIG (.aig) and netlist (.v) using the well-developed tool Yosys/Yosys-ABC. Since Yosys is an extended version of the ABC tool, we chose Yosys/Yosys-ABC as the optimization tool for our work. Yosys-ABC is essentially the ABC tool integrated within Yosys.

*1) Netlist to AIG:* Converting from netlist (.v) to AIG (.aig) is straightforward and can be done using the following terminal commands in Yosys:

```
read_verilog {verilog_file};
aigmap;
write_aiger {aig_file};
```

The generated `aig_file` can then be read in for subsequent optimization. Meanwhile, since the cost_estimator recognizes the module name, we store the module name of the netlist (.v) for use in our optimized netlist (.v) later.

*2) AIG to Netlist:* Conversely, converting from AIG (.aig) to netlist (.v) is more complex, as it involves technology mapping with a specified library. Fortunately, Yosys provides ABC commands that support mapping with an input library (.lib). To obtain the mapped netlist, execute the following command:

```
abc -exe {abc_binary} -liberty {lib_file}
```

Next, we need to write the netlist (.v) into a format that is supported by the given cost_estimator:

```
clean;
rename -top {module_name[1:]};
write_verilog -noattr {netlist_file}
```

The `clean` command effectively reduces wire assignments in the netlist (.v) by directly connecting the output of one gate to the input of the next gate. This is necessary because wire assignments cannot exist in the Verilog file for the cost_estimator. The `rename -top {module_name[1:]}` command replaces the module name generated by the mapping command with the module name of the original netlist (.v), which was previously stored during the *Netlist to AIG* process.

However, wire assignments may still exist in the Verilog file if there are wires connecting primary inputs and primary outputs, which cannot be eliminated by the `clean` command. In such cases, we replace these wires with a BUF gate from our optimized library, directly placing it between the primary inputs and outputs. After these steps, we obtain a valid netlist (.v) corresponding to our optimized AIG (.aig) for the cost_estimator.

### D. Cost Function Estimator

In this section, we will explain how we utilize the provided black-box cost_estimator binary file. The command for executing the cost_estimator includes inputs for the library (.json), netlist (.v), and output (.out) file.

To acquire the cost of a netlist(.v), we use the `check_output` function provided by the `subprocess` module in Python to execute the command on the cost_estimator binary file. We then decode the output to extract the cost. This cost information is subsequently used for optimization and library preparation.

### E. AIG Optimization

As described in the title, we employ reinforcement learning (RL) to address this problem. Due to the complexity of the AIG and the competition constraints—specifically, the limitation of using only one CPU core and a time limit of three hours—it is challenging to use a computationally feasible method to fully describe the state of the AIG. Consequently, the environment is only partially observable to our agent, making this a partially observable Markov decision process (POMDP)[4].

A POMDP is a framework for decision-making where the agent does not have complete information about the environment. It is defined by a 7-tuple $(S, A, R, T, \Omega, P, \gamma)$:

- $S$ represents the set of environment hidden states that the agent can't observe
- $A$ represents the set of actions, i.e., action space.
- $R$ is the reward function, which gives the immediate reward received after transitioning from one state to another due to an action
- $T$ is the transition function, describing the probability of moving from one state to another given a specific action
- $\Omega$ is the set of observations. In the following context, we will use "state space" to refer to it for simplicity.
- $P$ is the observation probability function, defining the probability of observing a particular observation given the state

- $\gamma$ is the discount factor, which represents the importance of future rewards

Since the AIG optimization problem can be further viewed as determining the best optimization action sequence to reach the lowest cost, our goal is to derive a policy $\pi^*$ that maximizes the probability of the correct state-action pair.

For computational feasibility, we need to define a fixed-dimension state space and action space, which will be detailed later. To ensure the agent continues to improve and does not get stuck in suboptimal solutions, we designed the reward function to be one-tenth of the cost difference each time, with a larger reward given when the cost surpasses the previous best. Our reward function can be written as:

$$R = \begin{cases} \Delta\text{cost} & \text{, if cost surpasses the best} \\ \frac{\Delta\text{cost}}{10} & \text{, otherwise} \end{cases} \tag{1}$$

*1) State Space:* As shown in Fig.2, we use commands in Yosys-ABC to get a summary of the current AIG graph as our state. We first use `print_stats` (abbr. `ps`) to extract vital status, such as the number of primary I/O, latency, and the number of `and` gates and layers. Then, we use `print_supp` to extract the support of every primary output. The support here means how many primary inputs are needed for this output.

```
abc 02> ps
../data/aigers/netlist       : i/o =   14/    8  lat =    0  and =     72  lev = 12
abc 02> print_supp
Structural support info:
    0                  po0 :  Cone =    13.  Supp =     8. (PIs =     8. FFs =     0.)
    1                  po1 :  Cone =    34.  Supp =    12. (PIs =    12. FFs =     0.)
    2                  po2 :  Cone =    23.  Supp =    10. (PIs =    10. FFs =     0.)
    3                  po3 :  Cone =    34.  Supp =    12. (PIs =    12. FFs =     0.)
    4                  po4 :  Cone =    64.  Supp =    14. (PIs =    14. FFs =     0.)
    5                  po5 :  Cone =    44.  Supp =    14. (PIs =    14. FFs =     0.)
    6                  po6 :  Cone =    43.  Supp =    13. (PIs =    13. FFs =     0.)
    7                  po7 :  Cone =    19.  Supp =    10. (PIs =    10. FFs =     0.)
```

Fig. 2. Example result of running ABC command to extract state.

We concatenate these into a vector to become our final state. Note that since primary input and output are fixed in a task, our state space is thus a fixed dimension continuous space with integer type.

*2) Action Space:* We choose 27 commands in `abc.rc`, including original commands of ABC, combinations of them, and do nothing, as our one step. To reduce the time of I/O, while maintaining the proper complexity of action space, we take 3 consecutive steps at a time before we get a reward from the environment. Thus, our action space is a "MultiDiscrete" with 3 dimensions of 27 values. The total combinations of our actions are

$$||A|| = 27^3 = 19683$$

### F. Advantage Actor Critique(A2C)

Among various RL methods, the Advantage Actor-Critic (A2C) algorithm (synchronous version of A3C [5]) is known for its efficiency and stability by combining value-based and policy-based approaches.

The A2C algorithm leverages the actor-critic architecture, where the actor learns a policy while the critic evaluates this policy.

- **Actor** modeled a policy (action probabilities for a given state, $\pi(a|s)$).
- **Critic** estimate the value function, the expected discounted return from a state.

The advantage function, defined as:

$$A(s,a) = Q(s,a) - V(s),$$

measures the relative quality of actions by comparing the action-value function $Q(s,a)$ with the value function $V(s)$.

The actor updates its policy to increase the probability of actions with positive advantages using the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) A(s,a)],$$

where $\theta$ represents the policy parameters.

The critic updates the value function by minimizing the mean squared error between predicted Q-values and actual returns:

$$L(\phi) = \mathbb{E}[(R_t - Q_\phi(s_t))^2],$$

where $\phi$ denotes the value function parameters, and $R_t$ is the actual return from state $s_t$.

Besides efficiency and stability, the computational resources and the properties of our state and action space are also reasons for us to choose A2C. While another state-of-the-art algorithm, PPO [6], shows promising results and is compatible with our space properties, it is known to be computationally intensive and sensitive to hyperparameters.

## V. Baselines

In addition to the Reinforcement Learning method for AIG optimization mentioned in the previous section, we have also developed three different optimization algorithms as baselines: greedy, simulated annealing, and fast simulated annealing[7].

### A. Greedy Algorithm

This greedy approach ensures that each step aims to reduce the cost, iterating until no further improvements can be made.

As shown in Algorithm 2, the best cost in previous iterations *prev_cost* is set to infinity and assigns the current design file to the given AIG file at first. In each iteration of the while loop, the algorithm initializes the minimmum cost for current command *best_cost* to the *prev_cost*. For each command in the provided list of commands, it applies the command to the AIG file to attempt an improvement. If the AIG is changed, it converts the improved AIG to a netlist and estimates the cost by black-box cost estimator. If the new cost is lower than the best cost found so far, it updates *best_cost* and saves the best AIG file. After all commands have been applied, if the best cost found in the current iteration is lower than the previous cost, the algorithm updates the current design file and *prev_cost*. If no improvement is found, the algorithm terminates. Finally, it returns the optimized AIG file and the best cost achieved.

**Algorithm 2** Greedy Optimization

**Input:** AIG file, commands
**Output:** Optimized AIG file, best cost
Initialize *prev_cost* to $\infty$
Initialize current design file to AIG file
**while** True **do**
    Initialize *best_cost* to *prev_cost*
    **for** each command in commands **do**
        Apply command to improve AIG
        **if** AIG is changed **then**
            Convert improved AIG to netlist
            Estimate cost of improved AIG
            **if** cost $<$ *best_cost* **then**
                Update *best_cost*
                Save the best AIG file
            **end if**
        **end if**
    **end for**
    **if** *best_cost* $<$ *prev_cost* **then**
        Update current design file and *prev_cost*
    **else**
        Break
    **end if**
**end while**
Return current design file and *best_cost*

**Algorithm 3** Simulated Annealing Optimization

**Input:** AIG file, commands, initial temperature, cooling rate
**Output:** Optimized AIG file, best cost
Initialize temperature with the input temperature
Set initial design file to AIG file and estimate initial cost
**while** temperature $>$ 0.1 **do**
    **for** trial = 1 to 100 **do**
        Randomly select a set of commands
        Improve AIG and estimate cost
        **if** cost is lower than previous cost **then**
            Accept the new design
            Update the design file and cost
        **else**
            Accept the design with probability $e^{-\Delta/T}$
        **end if**
        **if** cost $<$ best cost **then**
            Save the best design
        **end if**
        **if** number of accepted optimizations $= 10$ **then**
            Break
        **end if**
    **end for**
    Cool down the system
    Update temperature
**end while**
Return best cost and corresponding design file

### *B. Simulated Annealing (SA)*

The algorithm for simulated annealing applied to AIG optimization is shown in Algorithm 3.

In this algorithm, we recursively attempt to optimize the AIG until the temperature drops below a defined threshold. For each iteration, we randomly select a set of optimization commands from the action space, apply these commands to the current AIG, and convert it to a netlist to determine its cost using the cost_estimator. If the cost improves, we accept the new AIG and proceed to the next iteration. If the cost does not improve, there is still a chance to accept the new AIG with a probability of $e^{-\Delta/T}$. This acceptance criterion is a crucial aspect of simulated annealing, as it allows for occasional uphill moves, providing an opportunity to escape local minima and potentially find a better global minimum, thereby addressing the main drawback of the greedy algorithm.

### *C. Fast Simulated Annealing (FSA)*

[7] utilize the Fast Simulated Annealing (FSA) algorithm and show good results in the field of floorplanning. The algorithm for fast simulated annealing applied to AIG optimization is shown in Algorithm 4. Similar to simulated annealing, FSA uses temperature to control the probability of accepting a rising step. However, it defines temperature as:

$$T_n = \begin{cases} \frac{\Delta_{up}}{\ln P} & n = 1 \\ \frac{T_1 < \Delta_{cost} >}{nc} & 2 \le n \le k \\ \frac{T_1 < \Delta_{cost} >}{n} & k \le n \end{cases} \tag{2}$$

where, $n$ is the number of iterations, $\Delta_{up}$ is the average up-hill cost, $P$ is the initial probability of accepting an up-hill step, $\Delta_{cost}$ is the average cost change for the current temperature, and c, k are hyperparameters.

We would like it to explore the solution space at the first iteration. $P$ is set close to 1, so it performs a random search to find a good solution. The second stage performs the pseudo-greedy search until the *k*th iteration. Here, we set $c = 100$, $k = 7$. After $k$ iterations, $T$ jumps up to further improve the solution.

## VI. Results

The results are shown in Table I below. There are 48 test cases, encompassing 6 different netlist designs and 8 different cost estimators provided by the 2024 IC/CAD contest. We compare the performance of various optimization algorithms, including Reinforcement Learning and baseline algorithms: Greedy, Simulated Annealing, and Fast Simulated Annealing. The cost values are bolded if they represent the minimum cost in each test case.

The Greedy method, while efficient and yields acceptable results, demonstrates significant limitations in finding long-horizon action sequences to escape sub-optimal points. Conversely, the SA and FSA methods, which incorporate probabilistic techniques to escape local minima, exhibit a more balanced performance across all cost estimators, though their results still vary in comparison to the RL method evaluated.

TABLE I
COMPARISON OF OPTIMIZATION METHODS ACROSS DIFFERENT DESIGNS AND COST ESTIMATORS

| **design1.v** | **cost_estimator 1** | **cost_estimator 2** | **cost_estimator 3** | **cost_estimator 4** | **cost_estimator 5** | **cost_estimator 6** | **cost_estimator 7** | **cost_estimator 8** |
|---|---|---|---|---|---|---|---|---|
| **Greedy** | 2.4966 | **1.0000** | **1.0000** | 1.8319 | **1.0066** | **0.0000** | **20018002.2824** | 1.2557 |
| **SA** | 2.4368 | **1.0000** | **1.0000** | 1.8001 | 1.0067 | **0.0000** | 20018002.4148 | 1.2478 |
| **FSA** | 2.4680 | **1.0000** | **1.0000** | 1.8153 | 1.0067 | **0.0000** | **20018002.2824** | 1.2520 |
| **RL** | **2.4277** | **1.0000** | **1.0000** | **1.7880** | **1.0066** | **0.0000** | 20016002.4636 | **1.2384** |
| **design2.v** | **cost_estimator 1** | **cost_estimator 2** | **cost_estimator 3** | **cost_estimator 4** | **cost_estimator 5** | **cost_estimator 6** | **cost_estimator 7** | **cost_estimator 8** |
| **Greedy** | 40.8392 | 511.4755 | 3.3226 | 24.4458 | 1.1160 | 12.0000 | 20196048.2383 | 4.2530 |
| **SA** | 41.6155 | 498.8681 | 4.9847 | **23.9611** | 1.1212 | **6.0000** | **20160051.5121** | 4.3094 |
| **FSA** | 43.5569 | 821.6993 | 5.0252 | 25.3386 | 1.1225 | 9.0000 | 20168048.9632 | 4.3323 |
| **RL** | **40.0006** | **228.5182** | **3.3306** | 24.0189 | **1.1103** | **6.0000** | 20164050.3260 | **4.2240** |
| **design3.v** | **cost_estimator 1** | **cost_estimator 2** | **cost_estimator 3** | **cost_estimator 4** | **cost_estimator 5** | **cost_estimator 6** | **cost_estimator 7** | **cost_estimator 8** |
| **Greedy** | 53.0899 | **1.0000** | **1.0000** | **13.9047** | 1.1351 | 273.0000 | 20048449.8874 | 3.0060 |
| **SA** | 53.5426 | **1.0000** | **1.0000** | 14.0095 | 1.1424 | 228.0000 | **60.9022** | 3.0131 |
| **FSA** | 54.0533 | **1.0000** | **1.0000** | 14.0851 | **1.1389** | 210.0000 | 61.6071 | 3.0131 |
| **RL** | **52.6597** | **1.0000** | **1.0000** | 13.9424 | **1.1389** | **174.0000** | 62.4987 | **2.9883** |
| **design4.v** | **cost_estimator 1** | **cost_estimator 2** | **cost_estimator 3** | **cost_estimator 4** | **cost_estimator 5** | **cost_estimator 6** | **cost_estimator 7** | **cost_estimator 8** |
| **Greedy** | 143.7540 | **1.0000** | **1.0000** | 26.5101 | 1.2001 | **0.0000** | 148.2734 | 4.0501 |
| **SA** | **137.0197** | **1.0000** | **1.0000** | 25.7560 | **1.1974** | **0.0000** | **145.4593** | 3.9968 |
| **FSA** | 138.7082 | **1.0000** | **1.0000** | 25.9582 | 1.2038 | **0.0000** | 152.1720 | 4.0075 |
| **RL** | 140.2703 | **1.0000** | **1.0000** | **25.4062** | 1.1981 | **0.0000** | 147.5512 | **3.9607** |
| **design5.v** | **cost_estimator 1** | **cost_estimator 2** | **cost_estimator 3** | **cost_estimator 4** | **cost_estimator 5** | **cost_estimator 6** | **cost_estimator 7** | **cost_estimator 8** |
| **Greedy** | 455.0677 | **1.0000** | **1.0000** | **57.0821** | 2.1644 | 753.0000 | 383.8225 | **5.7745** |
| **SA** | 455.7214 | **1.0000** | **1.0000** | 57.1714 | **2.1638** | **648.0000** | 383.5749 | 5.7911 |
| **FSA** | 456.8446 | **1.0000** | **1.0000** | 57.2526 | 2.1868 | 753.0000 | 386.1975 | 5.7913 |
| **RL** | **450.7899** | **1.0000** | **1.0000** | **57.0821** | 2.3174 | 879.0000 | **380.4570** | 5.7773 |
| **design6.v** | **cost_estimator 1** | **cost_estimator 2** | **cost_estimator 3** | **cost_estimator 4** | **cost_estimator 5** | **cost_estimator 6** | **cost_estimator 7** | **cost_estimator 8** |
| **Greedy** | 948.3810 | **1.0000** | **1.0000** | 92.7069 | 3.5991 | 5313.0000 | 809.4337 | **7.5672** |
| **SA** | 938.5901 | **1.0000** | **1.0000** | 93.3088 | 3.6362 | 5280.0000 | 809.2110 | 7.6128 |
| **FSA** | 951.5275 | **1.0000** | **1.0000** | 93.9176 | 3.6200 | 5184.0000 | 810.5592 | 7.6128 |
| **RL** | **935.5502** | **1.0000** | **1.0000** | **91.9176** | **3.5476** | **5052.0000** | **803.8760** | 7.5961 |

Our main method RL stands out as the most consistently effective optimization method in this comparative study. RL outperforms other methods across a diverse array of cost estimators, underscoring its robust adaptability and comprehensive problem-solving capabilities.

During the experimental process, we observed that assigning a high learning rate to the RL agent can be problematic. The intrinsic characteristics of the AIG optimization problem make the agent prone to getting stuck in suboptimal solutions, resulting in excessive interactions with these suboptimal states. Even with the implementation of early stopping to terminate the episode prematurely, the agent quickly learns and becomes restricted, continuously producing the same actions. Therefore, we assigned a lower learning rate to the agent and utilized early stopping mechanisms as much as possible to prevent it from being trapped in suboptimal states for extended periods.

We found that this approach effectively resolved the issue, allowing the agent ample opportunities for random searching and exploration during the initial stages. This broader exploration helps in learning a more comprehensive value function. Simultaneously, maintaining a moderate learning rate ensures that after adequate exploration, the agent can gradually learn the optimal action sequence, reducing the proportion of useless attempts. This balanced strategy enhances the agent's performance by optimizing the exploration-exploitation trade-off, leading to more efficient and effective learning outcomes.

## VII. FUTURE WORK

### A. Technology Mapping and Library Generation

In our approach, we generate an optimized library (.lib) containing the best pairs of gate names and their respective costs. We then map the optimized AIG (.aig) to a netlist (.v) using the costs of the gates specified in our library.

There are two potential factors to consider. The first is the Library Generation process. We select the gate name with the lowest cost for each gate type and discard all others. This method significantly reduces the flexibility in mapping, as a gate name with a higher individual cost might lead to a lower overall cost for the entire netlist when evaluated by the black-box cost_estimator.

The second factor is the technology mapping process. We employ a command in Yosys to map the AIG (.aig) to a netlist (.v) using the input library. However, mapping is also a critical step for reducing the cost of circuits. For instance, the choice between using a NAND gate and an AND gate combined with a NOT gate can result in different costs. Therefore, this process should also incorporate efficient optimization algorithms, not solely AIG optimization.

### B. Netlist (.v) Optimization

Although Yosys/Yosys-ABC offers many useful optimization commands for AIG, the conversion between AIG (.aig) and Netlist (.v) can be quite redundant, potentially impacting

**Algorithm 4** Fast Simulated Annealing Optimization

**Input:** AIG file, commands, initial acceptance probability $P$, constant $c$, iterations $k$
**Output:** Optimized AIG file, best cost
Initialize iteration count $i = 1$
Estimate initial cost
Collect rollout costs to calculate $< \Delta_{up} >$ and $< \Delta_{cost} >$
Set initial temperature $T_1 = \frac{\Delta_{\text{avg}}}{\ln(P)}$
**while** True **do**
    **for** trial = 1 to 100 **do**
        Randomly select a set of commands
        Improve AIG and estimate cost
        **if** cost is lower than previous cost **then**
            Accept the new design
            Update the design file and cost
        **else**
            Accept the design with probability $e^{-\Delta/T}$
        **end if**
        **if** cost is lower than best cost **then**
            Save the best design
        **end if**
    **end for**
    Calculate new temperature $T_i$ using:

$$T_n = \begin{cases} \frac{\Delta_{up}}{\ln P} & n = 1 \\ \frac{T_1 < \Delta_{cost} >}{nc} & 2 \leq n \leq k \\ \frac{T_1 < \Delta_{cost} >}{n} & k \leq n \end{cases}$$

    Update temperature
**end while**
Return best cost and corresponding design file

the optimized cost significantly. Therefore, it is promising to develop algorithms that directly optimize the netlist using the given library or to find other representations that facilitate both optimization and conversion more efficiently.

## VIII. Job Division

### A. Chun-Kai Yang, B10901027

- Research on the problem and design the overall framework of the program
- Action space selection and state extraction
- Library generation
- AIG Optimization: RL, FSA
- Organizing the entire code structure

### B. Cheng-En Tsai, B10901098

- Research on the problem and design the overall framework of the program
- Research on the use of Yosys/Yosys-ABC
- Library generation
- Circuit conversion
- AIG Optimization: SA, FSA

### C. Ting-Hsuan Tien, B10901106

- Research on the problem and design the overall framework of the program
- Research on the use of Yosys/Yosys-ABC
- Circuit conversion
- AIG Optimization: Greedy

## References


[1] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "Drills: Deep reinforcement learning for logic synthesis," 2019.

[2] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, 2010.

[3] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," 2013.

[4] Åström, Karl Johan, "Optimal Control of Markov Processes with Incomplete State Information I," 1965.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[7] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on fast simulated annealing," 2005.