Exploring Logic Optimizations with Reinforcement Learning and Graph Convolutional Network

Keren Zhu
ECE Department, UT Austin
Austin, Texas, USA
keren.zhu@utexas.edu

Mingjie Liu
ECE Department, UT Austin
Austin, Texas, USA
jay_liu@utexas.edu

Hao Chen
ECE Department, UT Austin
Austin, Texas, USA
haoc@utexas.edu

Zheng Zhao
Synopsys
Mountain View, California, USA
zhengz@synopsys.com

David Z. Pan
ECE Department, UT Austin
Austin, Texas, USA
dpan@ece.utexas.edu

ABSTRACT

Logic synthesis for combinational circuits is to find the minimum equivalent representation for Boolean logic functions. A well-adopted logic synthesis paradigm represents the Boolean logic with standardized logic networks, such as and-inverter graphs (AIG), and performs logic minimization operations over the graph iteratively. Although the research for different logic representation and operations is fruitful, the sequence of using the operations are often determined by heuristics. We propose a Markov decision process (MDP) formulation of the logic synthesis problem and a reinforcement learning (RL) algorithm incorporating with graph convolutional network to explore the solution search space. The experimental results show that the proposed method outperforms the well-known logic synthesis heuristics with the same sequence length and action space.

CCS CONCEPTS

• **Hardware** → **Combinational synthesis**; **Circuit optimization**.

KEYWORDS

logic synthesis; reinforcement learning; graph neural network

ACM Reference Format:
Keren Zhu, Mingjie Liu, Hao Chen, Zheng Zhao, and David Z. Pan. 2020. Exploring Logic Optimizations with Reinforcement Learning and Graph Convolutional Network . In *2020 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD '20), November 16–20, 2020, Virtual Event, Iceland.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3380446.3430622

MLCAD '20, November 16–20, 2020, Virtual Event, Iceland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7519-1/20/11…$15.00
https://doi.org/10.1145/3380446.3430622

1 INTRODUCTION

Logic synthesis is the transitional step between abstract logic and physical implementation for very large-scale integrated (VLSI) circuits design. Figure 1 shows a typical VLSI design flow. A hardware system is first described in hardware descriptive language, from very abstract high-level synthesis language (HLS) to relatively concrete register-transfer level (RTL) language. For the hardware to be physically implemented, the hardware system needs to be translated from functional descriptive language and mapped into logic gate-level descriptions. Logic synthesis is the step in this transformation that happens, where the logic for the hardware needs to be translated into physical logic devices (such as NAND, NOR, XOR, INV).

To improved hardware performance and cost, optimizations are taken in the logic synthesis steps to reduce the number of mapped devices and decrease the hardware's latency for increased speed. Since there are multiple equivalent ways to implement the same functionality with different logical gates, there is often an ample space for logic synthesis to optimize. Usually, a large portion of the optimization efforts is on the logic function level, which is usually represented as logic graphs. A number of operations, e.g. *balance*, *rewrite* and *refactor*, on the logic graph is able to find an alternative representation without altering the logic function. Those operations would be beneficial to the design scale if they can reduce the number of nodes or depth of the logic graphs. After transferring the abstract logic to a detailed logic graph, the graph is mapped to the actual gates or devices in a specific technology, e.g., field-programmable gate array (FPGA) or VLSI technology. The logic gates are then physically placed and connected to generate a physical layout for manufacturing VLSI chips or a bitstream for programming FPGAs.

ABC [6][1] is a well adopted framework to perform logic synthesis and technology mapping in academia. As shown in Figure. 1, the ABC is composed of logic synthesis, technology mapping, and verification. Because of the variety of technology and the more "black box" nature, the logic synthesis community has focused on the logic synthesis algorithm alone. In other words, the literature often targets to optimize the statistic of the logic graph, such as the number of nodes and deepest logic level, instead of full power, area, timing after technology mapping.

[1] https://github.com/berkeley-abc/abc

Although the research for different logic representation and operations is fruitful, the sequence of using the operations are often determined by heuristic, such as *resyn2* in ABC flow. Several recent studies have proposed to explore the logic synthesis sequences automatically. The work [3] formulates the logic synthesis flow problem into the Markov decision process (MDP) and propose a reinforcement learning (RL) algorithm with functional policy approximation via Graph Neural Network (GNN) to explore the synthesis flow. However, the algorithm's scalability is doubtful, and the experiments shown in the work are limited. Therefore the effectiveness of the algorithm from [3] is not known. On the other hand, to solve the scalability of using GNN, Yu et al. [13] propose to use a convolution neural network (CNN) to predict whether a synthesis flow sequence is right or not. They randomly generate the sequences and prune them with the trained CNN classifier. There are also work on exploring the hyperparameters for high-level synthesis flow [11] and learning a compact circuit representation for high dimensional boolean logic [2].

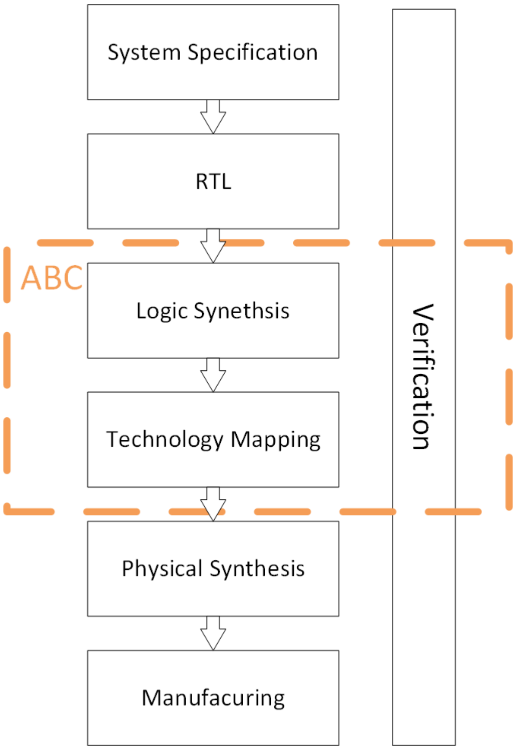

Figure 1: The flow of integrated circuit design with ABC.

This paper proposes a reinforcement learning (RL) algorithm to explore the search space for an effective logic synthesis sequence. We aim to push the synthesis results using the same action space of state-of-the-art heuristic *resyn2* to better results in terms of the number of nodes and logic depth. Our main contributions are summarized as follows:

- We propose to formulate the logic synthesis process as Markov Decision Process (MDP).
- We propose using a policy gradient algorithm to explore the search space of logic synthesis sequences effectively.
- We propose to use a graph convolutional network to aid the state representation in reinforcement learning.
- Experimental results demonstrate that our proposed framework outperforms the state-of-the-art heuristic with the same action space.

The source codes for this work have been released on Github[2].

2 PRELIMINARIES

In this section, we introduce the preliminaries for AIG network in Section 2.1, logic synthesis in Section 2.2 and Markov Decision Process in Section 2.3.

2.1 Logic Networks

In hardware designs, netlists are used to represent the implementation of logic circuits. A logic network is essentially a graph abstraction with a combinational logic function. Each node in the directed acyclic graph represents a primary logic gate, and the connection represents downstream logic paths. In the case of and-inverter graph (AIG) representations, the logic is only decomposed into using only *AND* and *NOT* gates. Figure 2.1 gives the example of a simple decoder logic AIG representation. AIG network can represent any combinational logic function and is equipped with effective logic operations for optimizations. AIG network is widely adopted in the ABC framework.

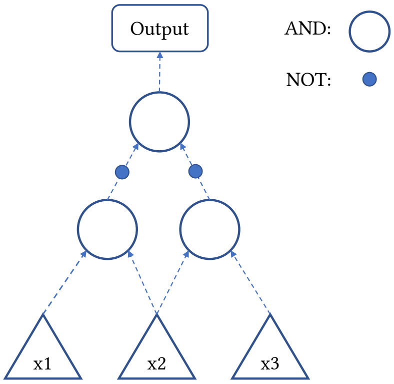

Figure 2: Example of a AIG logic $\neg(\mathbf{x1} \wedge \mathbf{x2} \vee \mathbf{x2} \wedge \mathbf{x3})$

2.2 Logic Synthesis

Logic synthesis transforms abstract hardware descriptive language into a gate-level netlist while optimizing the VLSI implementations' area, delay, and power. Widely-adopted logic synthesis frameworks [6, 8] achieves this by:

- Technology independent logic representations, such as AIGs.
- Fast optimization techniques based on graph representations for reduced logic.

[2] https://github.com/krzhu/abcRL

- Technology specific mapping representations.

The optimization process is at the core of logic synthesis algorithms, and various techniques have been proposed for reduced logic implementations, such as logic sharing and reuse. However, most of these optimizations need to be scalable for large graphs and fast. Thus these algorithms rely on suboptimal graph heuristics for fast and local optimizations. Generally, for complete logic design, these optimization techniques are applied iteratively until no improvements could be made. The sequence of different synthesis procedures could significantly impact the result. Commonly used performance evaluations are the number of logic nodes and the depth of the logic graph. A smaller number of logic nodes would result in fewer logic gates used, and a shallow depth would improve the logic circuit's speed.

2.3 Markov Decision Process

Markov decision process is widely used to formulate the discrete-time stochastic control process. In MDP formulation, the control processes are described with states, actions, state transitions, and rewards. Specially, an agent and the environment interacts with each other in sequences of discrete time steps, $t = 0, 1, 2, 3, \ldots$. In each time step t, the environment is in state $S_t \in \mathcal{S}$ and is able to take an action $A_t \in \mathcal{A}(s_t)$. Furthermore, after the agent takes action A_t, the $\mathcal{P}$ and $\mathcal{R}$ describe the probability of the next state it enters and the reward it gets as Equation 1.

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \qquad (1)$$

MDP is usually assuming Markov property so that the probability distribution of the future states depends only upon the current state. In other words, describing the MDP formulation process depends purely on the present state, and the past state sequence does not affect the environment.

3 ALGORITHM

We present our main algorithm in this section. Specially, we present our MDP formulation in Section 3.1, the RL algorithm in Section 3.2, graph convolutional network (GCN) in Section 3.3 and the used neural network architecture in Section 3.4.

3.1 MDP Formulation

In this section, we formulate the logic synthesis sequence into an MDP. As introduced previously in Section 2, logic synthesis optimization is naturally a sequential process, where the sequence of optimization actions would result in different performance outcomes. Furthermore, given an AIG graph, the outcome of an operation on the graph is deterministic, which simplifies the common probabilistic setting in typical MDP formulations. However, representing the non-Euclidean graph structure of the state is challenging. To tackle the problem's unique properties, we formulate our MDP problem as introduced in the rest of the section.

The work of [5] formulate synthesis as a Markov Chain Monte Carlo optimization, where it employs the Metropolis-Hastings Algorithm to determine the acceptance of different moves. However, the Markovian property would not hold if the transformation states are not involved since the outcome of moves would be dependant on its history. Or in other words:

$$Pr(a_{t+1} = a'|A_1 = a_1, \cdots, A_t = a_t) \neq Pr(a_{t+1} = a'|A_t = a_t) \quad (2)$$

The work of [3] refines the problem formulation as an MDP, which involves the state as the current logic graph. It decouples the reliance of future moves on its history by assuming that the logic graph states captures all the containing information:

$$Pr(a_{t+1} = a'|S_1 = s_1, \cdots, S_t = s_t) = Pr(a_{t+1} = a'|S_t = s_t) \quad (3)$$

We adopt a similar MDP formulation. Compared with the work in [3] where the state is represented using the entire logic graph. However, this limits the size of circuits at a scale of fewer than 100 nodes [3]. We extend previous works by exploring logical graph state representations that could be scalable to typical benchmark designs in our work.

3.1.1 State Space. In the paper, we propose to use the following state representation.

- The current number of nodes and logic depth.
- The number of nodes and logic depth before the last action.
- The one-hot vector for the last action.
- The sum of the one-hot vectors of the last three actions. Normalized.
- A scalar representing the current step. Normalized by 18, the empirical expected length of sequence.
- The AIG graph.

The above states would be compacted into a single vector representation. We integrate the graph statistics, operation history information, as well as the AIG graph itself together. The graph statistics and operation history information are concatenated into a vector. On the other hand, the AIG graph is handled separately, and we use a graph neural network to handle it, as explained in Section 3.3. We use the one-hot vector of node type as node features. The types are (1) constant one, (2) primary output, (3) primary input, (4) no inverter, (5) one inverter, and (6) two inverters.

3.1.2 Action Space. We represent the action space as discrete actions, as described in Tab. 1. For fair comparisons, in our project, we restrict the action space into the same as well-known heuristic *resyn2*. Table 1 shows the operations used in *resyn2*. Each operation may change the logic graph, and the result is deterministic.

Table 1: Action space $\mathcal{A}$

Abbreviation	Command
b	Balance: Balance the current network
rw	Rewrite: Performing rewriting of the AIG
rf	Refactor: Performing refactoring of the AIG
rwz	Rewrite with zero-cost replacements
rfz	Refactor with zero-cost replacements

3.1.3 Rewards. One naive approach to define the reward is directly using the gain in the number of nodes. However, since ABC will automatically discard the downgrades, such formulation causes the reward to be positive and results in a lousy convergence issue. To avoid that, we subtract the gain by a baseline. The baseline is gained from the heuristic *resyn2*. We run *resyn2* twice, which is in total a sequence of 20 operations, and get the reductions in the number of nodes or logic depth. We divide the gain by 20 to get the average improvement of each operation. In summary, the reward is defined in Equation 4.

$$\begin{aligned} r_t^1 &= num\_nodes_t - num\_nodes_{t+1} - baseline \\ r_t^2 &= depth - depth_{t+1} - baseline \end{aligned} \tag{4}$$

The two settings of reward formulations are used separately in the experiments. We normalize the number of nodes by the initial number from the input.

3.2 Reinforcement Learning Algorithm

In this work, we use a Monte Carlo policy gradient reinforcement learning (RL) algorithm, REINFORCE, as shown in Algorithm 1 [10].

REINFORCE algorithm is a policy gradient method. It learns the value for actions and selects the actions based on the estimated action values using a differentiable policy $\pi(a|s,\theta)$. $\pi(a|s,\theta)$ approximates the value of each action given the current state s. Action values are then used to construct a probability distribution of actions using softmax function. We sample an action to be taken from this distribution. The stochastic process allows the RL agent to explore action space.

After the current episode terminates, the REINFORCE algorithm accumulates the rewards in a Monte Carlo manner and further trains the policy. It accumulates the whole episode's rewards with a discount factor γ and inspects the RL agent's actions (Algorithm 1 line 5). It increased the action value approximation when the action resulted in positive rewards and vice versa. In other words, the $\pi(a|s,\theta)$ learns from experience and improves the accuracy of action values over time. We use $\gamma = 0.9$ in the experiments.

We also adopts a state-value function approximation $v(s,w)$ as baseline in REINFORCE algorithm [9]. The state value approximation serves as an estimation of the expected reward at each step. When updating the policy, this estimation of state value is subtracted from the actual reward (Algorithm 1 line 6). This approach helps reduce the variance in the learning process as it provides more consistent feedback to $\pi(a|s,\theta)$.

Algorithm 1 REINFORCE algorithm with Baseline

```
Input: A differentiable policy parameterization $\pi(a|s,\theta)$.
Input: A differentiable state-value function parameterization
    $v(s,w)$
Output: Updated $\pi(a|s,\theta), v(s,w)$
 1: Init. policy parameter $\theta$
 2: while Until convergence do
 3:     Generate an episode following $\pi(\cdot|\cdot,\theta)$
 4:     for each step of the episode $t = 0, 1, ..., T-1$ do
 5:         $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
 6:         $\delta \leftarrow G - v(S_t, w)$
 7:         $w \leftarrow w + Adam(\delta \nabla_w \ln v(S_t, w))$
 8:         $\theta \leftarrow \theta + Adam(\gamma^t \delta \nabla_\theta \ln \pi(A_t|S_t,\theta))$
 9:     end for
10: end while
```

3.3 Graph Convolutional Network

Graph neural networks (GNNs) have gained increasing attention in the design automation community because circuits can be naturally modeled as graphs. Since AIG logic could also be converted into a graph representation, we leverage graph convolutional networks to extract the current state's features. The initial node embedding for each graph node is a concatenated vector, containing information about the logic node type and PI/PO type. The node embeddings are then passed through two consecutive layers of graph convolution. Each graph convolution layer aggregates the local neighbor feature for each node based on graph connectivity,

$$h_i^{(k)} = \sigma\left(\sum_{j \in N(i)} \frac{1}{c_{ij}} h_u^{(k-1)} W^{(k-1)} + b^{(k-1)}\right). \tag{5}$$

The new node embedding would then be input for the next graph convolution layer. The embedding for the entire graph is thus calculated as the mean of all node embeddings at the final graph convolution layer,

$$h_{AIG} = \frac{1}{|V|} \sum_{i \in V} h_i^{(k)}. \tag{6}$$

3.4 Neural Network Architecture

We implement the policy and state-value function approximations using neural networks. Figure 3 shows the network architectures. We separate the state in the vector and the graph in Figure 3.

For the policy parameterization $\pi(a|s,\theta)$, we use four layers of graph convolutional layers to extract information from the AIG graph and three fully-connected layers to approximate the action values. The state-value parameterization, $v(s,w)$, on the other hand, only uses the vector part of the state representation in this work. The intuition is that we are motivated to use the state value to update the policy. Therefore it might be beneficial to trade off accuracy for lower variance.

The input vector is the state representation with a dimension of 10. We apply two fully connected hidden layers with 32 hidden neurons. The policy network outputs each action's probability, while the value network outputs a single scalar as the baseline state value. Empirically, the graph statistics and history operations in the vector part can provide credible information for the state value estimation.

The networks are differentiated using the back-propagation algorithm. We use Adam as the optimization function [4]. In the experiments, the learning rate is chosen to be 8×10^{-4} for $\pi(a|s,\theta)$ and 3×10^{-3} for $v(s,w)$. The exponential decay rates for the moment estimates β_1 and β_2 are chosen to be 0.9 and 0.999, respectively.

4 EXPERIMENTAL RESULTS

In this section, we evaluate our proposed algorithm on various benchmarks.

4.1 Experiment Setup

We set up the experiment in a Linux workstation. We implement the interface to ABC using C++ and the RL algorithm in Python with Pytorch [7] machine learning library. The experiments were conducted on a Linux workstation with an 8-core Intel 3.0 GHz

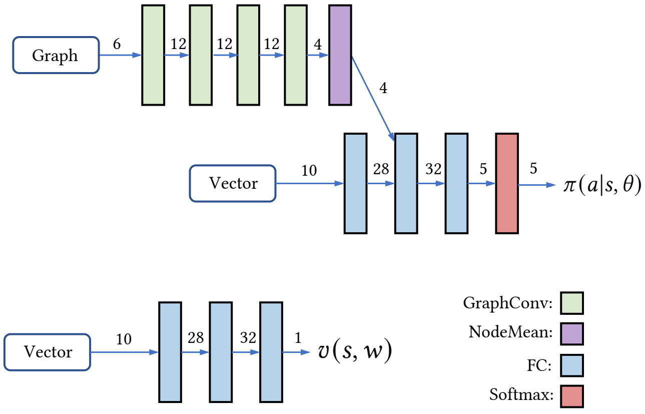

Figure 3: The neural network architectures. Nonlinear activation layers are omitted.

CPU with 64 GB memory. We trained the neural network with CPU, considering the network size is relatively small.

4.2 Evaluation of the RL Agent Effectiveness

We first evaluate the effectiveness of the RL agent. We perform 100 runs of our RL agent on the benchmark *i10* from [6]. For each run, we re-initialize every parameter and run 200 episodes. Each episode is of length 20.

We compare the results with two runs of *resyn2*. The action space and the length of the sequence are identical for the RL agent and the baseline.

Fig. 4 shows the mean and standard deviation of the improvement over the baseline. The black line represents the reduction of the number of nodes of two runs of *resyn2*. The blue line represents the average improvement in each step in 75 experiments. The shadowed region denotes the standard deviation of the reduction in the experiments. The y-axis is the ratio of further improvement over the baseline; the higher, the better. After roughly 50 iterations, the average performance of the RL agent has exceeded the state-of-the-art *resyn2*.

Fig. 5 shows the best and worst results of each step in the experiments. The worst results are closer to the mean in the later steps. This observation demonstrates that our RL agent is getting more and more robust in a short period.

4.3 Evaluation of the Performance

To evaluate our RL algorithm's performance, we arbitrary choose eight relatively large combinational circuit benchmarks from [1, 6, 12]. For each benchmark, we perform ten complete runs of our RL algorithm. There are 200 episodes in each run. Each episode is of 20 logic synthesis operations. At the end of each run, we inferences ten sequences and pick the best of them. We average the results collected in the runs. We present the RL algorithm results in two rewards settings: r_t^1 and r_t^2 as defined in Section 3.1.3. r_t^1 is focusing on optimizing the number of nodes in AIG graph, while r_t^2 is on optimizing the logic depth. The results for the two settings are denoted as **RL-1** and **RL-2**, respectively. We compare the average performance of the RL algorithm with three baselines:

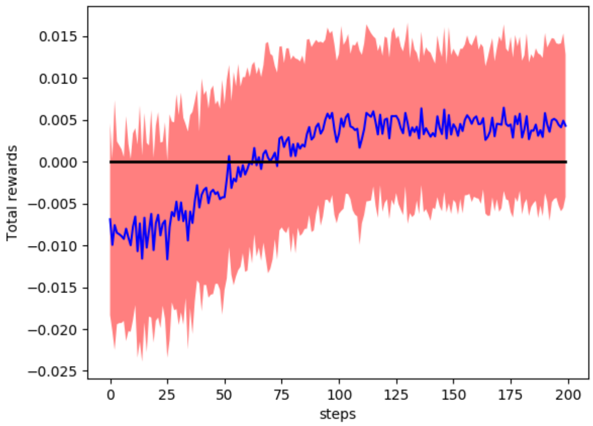

Figure 4: The improvement over two runs of resyn2. Shadow denotes the standard deviation.

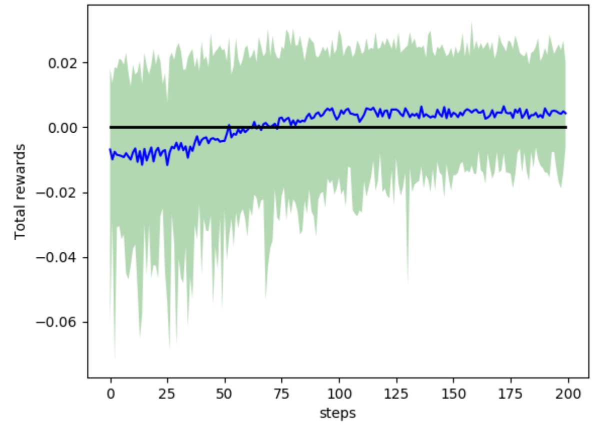

Figure 5: The improvement over two runs of resyn2. Shadow denotes the minimum and maximum in the experiments.

- One execution of *resyn2*. Denotes as **resyn2-1**.
- Two executions of *resyn2*. The sequence length of this baseline is the same as our RL setting. Denotes as **resyn2-2**.
- The converged results of iteratively executing *resyn2*. We repeatedly execute *resyn2* until the results becomes unchanged for at least 5 execution. Denotes as **resyn2-∞**.

Tab. 2 shows the results on ten different benchmarks. In general, our proposed algorithm not only beat the performance of **resyn2-2**, but also on average outperform the **resyn2-∞**. Considering the **resyn2-∞** in general has a much longer sequence length, the experimental results suggest the RL agent can find a more useful sequence then fixed heuristics. We believe the RL agent can explore

Table 2: Performance results: Number of nodes and logic depth in different optimization settings.

Benchmark	**Initial**		**resyn2-1**		**resyn2-2**		**resyn2-*n***			**RL-1**		**RL-2**	
	#Nodes	Depth	#Nodes	Depth	#Nodes	Depth	#Nodes	Depth	#resyn2	#Nodes	Depth	#Nodes	Depth
i10	2675	50	1829	32	1804	32	1789	32	6	1730.2	40.3	1839.4	31.9
c1355	504	25	390	16	390	16	390	16	1	386.2	17.6	390.0	16.0
c7552	2093	29	1469	26	1416	26	1398	26	7	1395.4	27.4	1460.8	22.1
c6288	2337	120	1870	89	1870	89	1870	89	1	1870.0	88.0	1882.0	88.0
c5315	1780	37	1306	28	1295	26	1294	26	3	1337.4	27.2	1364.7	25.4
dalu	1371	35	1106	31	1103	31	1103	31	2	1039.8	33.2	1095.6	30.0
k2	1998	23	1234	13	1186	13	1145	13	11	1128.4	19.8	1187.5	13.0
mainpla	5346	38	3678	26	3583	26	3504	25	7	3438.4	25.0	3504.0	25.5
apex1	2665	27	1999	17	1966	17	1941	17	7	1921.6	19.2	2004.7	17.0
bc0	1592	31	933	17	899	17	875	17	8	819.4	18.6	851.7	17.5
Ratio	1.000	1.000	0.730	0.706	0.717	0.702	0.707	0.699	-	0.698	0.757	0.720	0.687

a more extensive search space and escape from the local minima trapping the *resyn2*. On the other hand, the flexible reward settings also allow the RL agent to optimize for different objectives, as shown in the experimental results.

5 CONCLUSION

We present the RL for logic synthesis operation sequence exploration. We propose a MDP formulation of the logic synthesis problem. Our experimental results show our RL agent can outperform the state-of-the-art *resyn2* heuristic using the same action space.

ACKNOWLEDGEMENT

The authors would like to thank Cunxi Yu from University of Utah for helpful discussions.

REFERENCES

[1] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS*, May 1989.

[2] P.-W. Chen, Y.-C. Huang, C.-L. Lee, and J.-H. R. Jiang. Circuit learning for logic regression on high dimensional boolean space. 2020.

[3] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli. Deep learning for logic optimization algorithms. In *ISCAS*, 2018.

[4] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[5] G. Liu and Z. Zhang. A parallelized iterative improvement approach to area optimization for lut-based technology mapping. In *FPGA*, 2017.

[6] A. Mishchenko. Abc: A system for sequential synthesis and verification.

[7] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *Conference on Neural Information Processing Systems (NIPS)*, 2017.

[8] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarù, G. D. Micheli, and M. Soeken. Scalable generic logic synthesis: One approach to rule them all. In *DAC*, 2019.

[9] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 2st edition, 2018.

[10] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Conference on Neural Information Processing Systems (NIPS)*, 1999.

[11] Z. Wang and B. C. Schafer. Machine leaming to set meta-heuristic specific parameters for high-level synthesis design space exploration. In *DAC*, 2020.

[12] S. Yang. Logic synthesis and optimization benchmarks, 1989.

[13] C. Yu, H. Xiao, and G. De Micheli. Developing synthesis flows without human knowledge. In *DAC*, 2018.