Circuit Area and Optimal Recipe Prediction in Logic Synthesis via Domain-Specific Loss Guided Graph Neural Networks

Adarsh Gupta, Aditya Kumar, Parv Agarwal
220101003, 220101005, 220101124
Team 20: ML for EDA Course Project
Department of Computer Science and Engineering
IIT Guwahati, India

***Abstract*—Logic synthesis optimization aims to minimize circuit area while preserving functionality, but predicting the impact of synthesis recipes on different circuit designs remains challenging. This paper introduces a hybrid approach combining Graph Neural Networks (GNNs) and Recurrent Neural Networks (RNNs) to predict circuit area throughout the synthesis process. We represent circuits as directed graphs and synthesis recipes as sequences of commands, extracting meaningful features with specialized attention mechanisms. Our approach uses domain-specific loss functions that emphasize relative area reduction, critical optimization steps, and sequence dependencies. We also compare various optimal recipe generation frameworks utilizing three distinct approaches: Greedy Sampling, Monte Carlo Tree Search (MCTS), and Proximal Policy Optimization (PPO). Experimental results demonstrate strong predictive performance in tracking area changes across synthesis steps. Our analysis shows that PPO achieves the highest area reduction, while Greedy Sampling offers the best efficiency. For inference, we use an ensemble technique that adapts its weighting mechanism based on whether a specific model exists for the target design. The fine-tuning strategy leverages DTW-based clustering to extract representative recipes, improving model accuracy on unseen designs with minimal additional training data.**

***Index Terms*—Logic Synthesis, Electronic Design Automation, Graph Neural Networks, Machine Learning for EDA, Area Prediction**

I. Introduction and Problem Statement

Electronic design automation (EDA) tools employ a variety of synthesis and optimization techniques to transform high-level circuit descriptions into optimized gate-level implementations. In the realm of logic synthesis, area optimization is a critical objective that seeks to minimize the number of gates required to implement a circuit while preserving its functionality. This optimization process typically involves applying a sequence of transformation commands (a "recipe") to the circuit, with each command potentially reducing the circuit area.

The efficacy of synthesis recipes varies significantly across different circuit designs, making it challenging to predict the outcome of a particular optimization sequence without running the full synthesis flow. This leads to inefficient design space exploration, as designers must execute numerous synthesis runs to identify effective optimization strategies, consuming valuable time and computational resources.

A. Problem Formulation

We define a circuit design C as a directed acyclic graph (DAG) $G = (V, E)$, where V represents the set of nodes (gates and I/O pins) and E represents the connections between them. Each node $v_i \in V$ is associated with a type attribute $\tau(v_i) \in \{INPUT, OUTPUT, AND, NOT\}$ and other relevant features such as level (topological depth), fan-in, and fan-out.

A synthesis recipe R is defined as an ordered sequence of transformation commands $R = (r_1, r_2, ..., r_n)$, where each command $r_i \in \mathcal{C}$ belongs to a predefined set of available commands $\mathcal{C} = \{$ "rewrite -z", "rewrite -l", "rewrite", "balance", "resub", "refactor", "resub -z", "refactor -z" $\}$.

Let $A(C, R, i)$ represent the area (number of AND gates) of circuit C after applying the first i commands of recipe R. Our objective is to develop a predictive model f_θ with parameters θ such that:

$$\hat{A}(C, R, i) = f_\theta(C, R, i) \tag{1}$$

where $\hat{A}(C, R, i)$ approximates the true area $A(C, R, i)$ for all $i \in \{1, 2, ..., n\}$.

The prediction task can then be formulated as an optimization problem:

$$\theta^* = \arg\min_\theta \mathcal{L}(f_\theta(C, R), A(C, R)) \tag{2}$$

where $\mathcal{L}$ is a suitable loss function comparing the predicted area sequence

$$\hat{A}(C, R) = (\hat{A}(C, R, 1), \hat{A}(C, R, 2), ..., \hat{A}(C, R, n))$$

with the ground truth area sequence

$$A(C, R) = (A(C, R, 1), A(C, R, 2), ..., A(C, R, n))$$

.

This report is structured as follows, Sections II and III focus on the proposed methodology of the core QoR prediction

model. Section IV is focused on the experimental results and inference technique of the QoR Prediction model on the 8 training datasets. Section V is focused on fine-tuning the trained models using recipe clustering. Section VI is focused on using our QoR predictor model to find the best recipe.

II. Overview of Proposed Methodology

We propose a hybrid architecture that combines Graph Neural Networks (GNNs) and Recurrent Neural Networks (RNNs) to address the circuit area prediction problem. Our approach leverages the representational power of GNNs to capture the structural properties of circuit designs and the sequential modeling capabilities of RNNs to process synthesis recipes.

The overall framework consists of three key components:

1) **Circuit Embedding Module**: A GNN-based component that transforms the circuit graph into a fixed-dimensional embedding
2) **Recipe Processing Module**: An RNN-based component that processes the sequence of synthesis commands
3) **Area Prediction Module**: A prediction component that combines circuit and recipe information to estimate area after each synthesis step

Additionally, we introduce domain-specific loss functions that incorporate circuit optimization knowledge to guide the learning process and improve prediction accuracy. These specialized loss functions emphasize relative area reduction, critical optimization steps, attention interpretability, and meaningful circuit representations.

The proposed approach not only predicts the final area after applying the complete recipe but also provides a step-by-step prediction of area changes throughout the optimization process, offering valuable insights into the effectiveness of each synthesis command.

III. Detailed Methodology

A. Circuit Representation and Embedding

1) Graph Construction: Given a circuit description in the bench format, we construct a directed graph $G = (V, E)$ where each node represents a gate or an I/O pin, and edges represent connections between them. For each node $v_i \in V$, we extract a feature vector $\mathbf{x}_i$ that encodes:

- Gate type (one-hot encoded):

$$\mathbf{t}_i \in \{0, 1\}^4$$

for

$$\{INPUT, OUTPUT, AND, NOT\}$$

- Level: $l_i \in \mathbb{Z}^+$
- Fan-in count: $f_i^{in} \in \mathbb{Z}^+$
- Fan-out count: $f_i^{out} \in \mathbb{Z}^+$

The feature vector for node v_i is then defined as:

$$\mathbf{x}_i = [\mathbf{t}_i; l_i; f_i^{in}; f_i^{out}] \in \mathbb{R}^7 \tag{3}$$

2) Graph Neural Network Architecture: To learn a fixed-dimensional representation of the circuit, we employ a multi-layer Graph Convolutional Network (GCN). Each GCN layer updates node representations by aggregating information from neighboring nodes:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^{(l)} \sum_{j \in \mathcal{N}(i)} \frac{\mathbf{h}_j^{(l)}}{\sqrt{|\mathcal{N}(i)|} \cdot \sqrt{|\mathcal{N}(j)|}} + \mathbf{b}^{(l)}\right) \tag{4}$$

where $\mathbf{h}_i^{(l)}$ is the feature vector of node v_i at layer l, $\mathcal{N}(i)$ is the set of neighboring nodes of v_i, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are learnable parameters, and σ is a non-linear activation function (ReLU).

To enhance the model's ability to focus on critical circuit structures, we incorporate a graph attention mechanism:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]\right) \tag{5}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})} \tag{6}$$

$$\mathbf{h}_i' = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W}\mathbf{h}_j\right) \tag{7}$$

where $\|$ denotes concatenation, $\mathbf{a}$ is a learnable attention vector, and α_{ij} represents the attention coefficient between nodes i and j.

The final circuit embedding is obtained by applying a global pooling operation over all node representations:

$$\mathbf{z}_C = \frac{1}{|V|} \sum_{i=1}^{|V|} \mathbf{h}_i^{(L)} \tag{8}$$

where L is the number of GCN layers.

Motivation: The choice of GNNs for circuit representation is motivated by their inherent ability to capture both local connectivity patterns and global structural properties of circuits. Traditional feature-based approaches fail to encode the complex relationships between gates, whereas GNNs naturally model these dependencies through message passing. The incorporation of attention mechanisms allows the model to identify critical paths and gates that significantly impact area optimization, mirroring how experienced designers focus on specific circuit structures.

B. Recipe Encoding and Processing

1) Command Encoding: Each synthesis command $r_i \in \mathcal{C}$ is encoded using a one-hot encoding scheme, resulting in an 8-dimensional vector $\mathbf{r}_i \in \{0, 1\}^8$ (since $|\mathcal{C}| = 8$). A recipe $R = (r_1, r_2, ..., r_n)$ is thus represented as a sequence of one-hot encoded vectors $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_n) \in \{0, 1\}^{n \times 8}$.

2) Sequential Recipe Processing: To capture the sequential dependencies between synthesis commands, we employ a bidirectional LSTM network:

$$\overrightarrow{\mathbf{h}}_t = \text{LSTM}_{\text{forward}}(\mathbf{r}_t, \overrightarrow{\mathbf{h}}_{t-1}) \tag{9}$$

$$\overleftarrow{\mathbf{h}}_t = \text{LSTM}_{\text{backward}}(\mathbf{r}_t, \overleftarrow{\mathbf{h}}_{t+1}) \tag{10}$$

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t \| \overleftarrow{\mathbf{h}}_t] \tag{11}$$

where $\overrightarrow{\mathbf{h}}_t$ and $\overleftarrow{\mathbf{h}}_t$ represent the forward and backward hidden states at time step t, and $\mathbf{h}_t$ is the concatenated bidirectional representation.

To further enhance the model's ability to capture dependencies between different optimization steps, we incorporate a self-attention mechanism:

$$\mathbf{Q} = \mathbf{H}\mathbf{W}_Q, \mathbf{K} = \mathbf{H}\mathbf{W}_K, \mathbf{V} = \mathbf{H}\mathbf{W}_V \tag{12}$$

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \tag{13}$$

$$\mathbf{H}' = \mathbf{A}\mathbf{V} \tag{14}$$

where $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_n]^T$ is the matrix of hidden states, $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are learnable parameter matrices, d_k is the dimension of the keys, and $\mathbf{H}'$ is the attention-enhanced representation.

Motivation: The sequential nature of synthesis recipes necessitates a model that can capture both the individual effects of commands and their interactions. Bidirectional LSTMs allow the model to consider both past and future commands when processing each step, recognizing that the effectiveness of a command often depends on the entire optimization context. The self-attention mechanism enables the model to discover long-range dependencies between non-adjacent commands, capturing complex optimization patterns that might span across multiple steps.

C. Area Prediction Module

1) Stepwise Area Prediction: For each synthesis step i, we combine the circuit embedding $\mathbf{z}_C$ with the corresponding recipe state $\mathbf{h}_i$ to predict the area after applying the first i commands:

$$\mathbf{u}_i = [\mathbf{z}_C \| \mathbf{h}_i] \tag{15}$$

$$\mathbf{v}_i = \text{ReLU}(\mathbf{W}_1\mathbf{u}_i + \mathbf{b}_1) \tag{16}$$

$$\hat{A}(C, R, i) = \mathbf{W}_2\mathbf{v}_i + \mathbf{b}_2 \tag{17}$$

where $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$, and $\mathbf{b}_2$ are learnable parameters.

2) Uncertainty Estimation: To quantify the model's confidence in its predictions, we estimate the uncertainty for each predicted area:

$$\sigma^2(C, R, i) = \text{softplus}(\mathbf{W}_\sigma\mathbf{v}_i + \mathbf{b}_\sigma) \tag{18}$$

where $\mathbf{W}_\sigma$ and $\mathbf{b}_\sigma$ are learnable parameters, and softplus ensures positive uncertainty values.

Motivation: The area prediction module is designed to capture how circuit properties interact with synthesis commands to influence area reduction. By predicting area after each step, rather than just the final result, the model provides valuable insights into the progression of optimization. The uncertainty estimation acknowledges that prediction difficulty varies across different circuit-recipe combinations, allowing designers to assess the reliability of predictions and make informed decisions.

D. Enhanced Loss Functions

To improve model performance and incorporate domain knowledge, we developed specialized loss functions that address various aspects of circuit optimization prediction.

1) Relative Area Reduction Loss: This loss function focuses on the relative changes in area rather than absolute values after each optimisation step:

$$\Delta\hat{A}_i = \frac{\hat{A}(C, R, i+1) - \hat{A}(C, R, i)}{\hat{A}(C, R, i) + \epsilon} \tag{19}$$

$$\Delta A_i = \frac{A(C, R, i+1) - A(C, R, i)}{A(C, R, i) + \epsilon} \tag{20}$$

$$\mathcal{L}_{rel} = \frac{1}{n-1}\sum_{i=1}^{n-1}\left(\Delta\hat{A}_i - \Delta A_i\right)^2 \tag{21}$$

where ϵ is a small constant added for numerical stability.

Motivation: In circuit optimization, percentage reductions are often more meaningful than absolute changes, especially when comparing across circuits of different sizes. This loss encourages the model to accurately predict the relative impact of each optimization step, aligning with how designers typically evaluate optimization quality.

2) Critical Step Emphasis Loss: This loss function places greater emphasis on accurately predicting steps that cause significant area changes:

$$E_{i+1} = \left|\hat{A}(C, R, i+1) - A(C, R, i+1)\right| \tag{22}$$

$$\Delta A_i = |A(C, R, i+1) - A(C, R, i)| \tag{23}$$

$$\mathcal{L}_{crit} = \frac{1}{n-1}\sum_{i=1}^{n-1} E_{i+1} \cdot (1 + \Delta A_i) \tag{24}$$

Motivation: Not all synthesis steps have equal impact on area reduction. Steps that significantly reduce area are particularly valuable to predict accurately, as they represent key optimization opportunities. This loss function guides the model to pay special attention to high-impact steps, which are the most relevant for designers seeking efficient optimization strategies.

3) Sequence Dependency Loss: This loss function models how the effectiveness of synthesis commands depends on the sequence of steps that came before:

$$\mathcal{L}_{seq} = \frac{1}{n}\sum_{i=0}^{n-1}\widetilde{\mathcal{E}}_i \tag{25}$$

where $\widetilde{\mathcal{E}}_i$ is the weighted prediction error at step i, computed as:

$$\widetilde{\mathcal{E}}_i = \begin{cases} \mathcal{E}_i & \text{if } i = 0 \\ \mathcal{E}_i \cdot (1 + \mathcal{D}_i) & \text{if } i > 0 \end{cases} \tag{26}$$

Here, $\mathcal{E}_i = (\hat{A}(C, R, i) - A(C, R, i))^2$ is the squared prediction error at step i.

The dependency factor $\mathcal{D}_i$ captures the influence of previous steps on the current step i:

$$\mathcal{D}_i = \sum_{j=0}^{i-1} S_{ij} \cdot \mathcal{E}_j \tag{27}$$

The similarity coefficient S_{ij} between steps i and j is computed from the recipe embeddings:

$$S_{ij} = \langle \mathbf{h}_i, \mathbf{h}_j \rangle \tag{28}$$

where $\mathbf{h}_i$ is the embedding of step i, and $\langle \cdot, \cdot \rangle$ denotes the dot product.

Motivation: This loss function captures a fundamental aspect of circuit optimization: the effect of a synthesis command at step i often depends strongly on what commands were applied before it. By computing similarities between command embeddings, the loss identifies which previous steps are conceptually related to the current step. When the model makes errors on steps that are similar to the current step, the penalty for errors at the current step is increased. This encourages the model to learn patterns such as "command A is less effective after command B" or "commands C and D work well in sequence." The multiplicative factor $(1 + \mathcal{D}_i)$ ensures that predictions for steps with strong dependencies on previous steps are penalized more heavily when those dependencies aren't properly modeled. This loss enables the model to capture the sequential nature of synthesis optimization, where the effectiveness of each command is contextual rather than absolute.

4) Attention Entropy Loss: This loss function encourages the attention mechanism to focus more sharply on relevant information:

$$\mathcal{L}_{attn} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_{ij}\log(\alpha_{ij} + \epsilon) \tag{29}$$

where α_{ij} represents the attention weight from step i to step j.

Motivation: Diffuse attention patterns can make the model's decision-making process difficult to interpret. By encouraging more focused attention, this loss function improves the interpretability of the model's predictions, helping designers understand which synthesis steps influence others and why certain predictions are made.

5) Feature Consistency Loss: This loss function ensures that circuits with similar optimization behavior have similar embeddings:

$$d_{ij}^{emb} = \|\mathbf{z}_{C_i} - \mathbf{z}_{C_j}\|_2 \tag{30}$$

$$d_{ij}^{area} = \frac{\|A(C_i, R_i) - A(C_j, R_j)\|_2}{\max_{k,l}\|A(C_k, R_k) - A(C_l, R_l)\|_2 + \epsilon} \tag{31}$$

$$\mathcal{L}_{feat} = \frac{1}{B(B-1)}\sum_{i=1}^{B}\sum_{j\neq i}^{B}\left(d_{ij}^{emb} - d_{ij}^{area}\right)^2 \tag{32}$$

where B is the batch size, and d_{ij}^{emb} and d_{ij}^{area} are the distances between circuit embeddings and area profiles, respectively.

Motivation: This loss function helps create a semantically meaningful embedding space where proximity indicates similar optimization behavior. Such a space facilitates better generalization to new circuits, as the model can relate them to similar previously seen examples. This is particularly valuable in EDA applications, where knowledge transfer between related circuits is essential for efficient optimization.

6) Combined Loss Function: The final loss function combines these specialized components with traditional MSE losses:

$$\begin{aligned} \mathcal{L} = \lambda_1 \cdot \text{MSE}(\hat{A}(C, R, n), A(C, R, n)) \\ +\lambda_2 \cdot \frac{1}{n}\sum_{i=1}^{n}\text{MSE}(\hat{A}(C, R, i), A(C, R, i)) \\ +\lambda_3 \cdot \mathcal{L}_{rel} + \lambda_4 \cdot \mathcal{L}_{crit} + \lambda_5 \cdot \mathcal{L}_{seq} \\ +\lambda_6 \cdot \mathcal{L}_{attn} + \lambda_7 \cdot \mathcal{L}_{feat} + \lambda_8 \cdot \mathcal{L}_{NLL} \end{aligned} \tag{33}$$

where $\mathcal{L}_{NLL}$ is the negative log-likelihood loss for uncertainty estimation:

$$\epsilon_i = (\hat{A}(C, R, i) - A(C, R, i))^2 \tag{34}$$

$$\sigma_i^2 = \sigma^2(C, R, i) \tag{35}$$

$$\mathcal{L}_{NLL} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{2}\log\sigma_i^2 + \frac{\epsilon_i}{2\sigma_i^2}\right) \tag{36}$$

and $\lambda_1, \lambda_2, ..., \lambda_8$ are hyperparameters that control the contribution of each loss component.

IV. QoR Predictor Experimental Results

A. Dataset and Experimental Setup

We evaluated our approach on a dataset of circuit designs, each associated with synthesis recipes of up to 20 steps and the corresponding AND gate counts after each step. Each circuit is represented in the bench format, providing structural information that we convert into graph representations.

For each design a separate prediction model was trained using 5000 randomly generated recipes. Each recipe spanned 20 random operations from the set of operations defined above. Each dataset was divided into training (70%), validation (15%), and test (15%) sets.

We utilize a 3-layer Graph Neural Network (GNN) with 128 hidden units per layer and residual connections, alongside a 2-layer bidirectional LSTM, also with 128 hidden units per layer.

A 4-head self-attention mechanism is employed for capturing dependency relationships. The node features are represented as a 7-dimensional vector, which includes a gate type one-hot encoding, level, fan-in, and fan-out information, while the recipe is encoded using 8-dimensional one-hot vectors. For training, the Adam optimizer is used with a learning rate of 0.001 and a weight decay of 1e-5, processing data in batches of 16. A Cosine Annealing Learning Rate Scheduler is applied to adjust the learning rate over the training epochs, and early stopping is implemented with a patience of 20 epochs. The loss function is balanced with weights set to λ_1 = 1.0, λ_2 = 1.0, λ_3 = 1.0, λ_4 = 0.3, λ_5 = 0.2, λ_6 = 0.1, λ_7 = 0.2, and λ_8 = 0.3.

B. Evaluation Metrics

For each design of the 8 designs a seperate model was trained. Each model was evaluated using the following metrics:

- Mean Squared Error (MSE): Average squared difference between predicted and actual area values
- Mean Absolute Error (MAE): Average absolute difference between predicted and actual area values
- Root Mean Squared Error (RMSE): Square root of MSE
- Coefficient of Determination (R^2): Proportion of variance in the target variable explained by the model

C. Training Progression

Figure **??** illustrates the training and validation loss curves over epochs, showing the model's learning progression. The curves demonstrate that our model effectively learns to predict area values while avoiding overfitting, as evidenced by the convergence of training and validation losses.

D. Model Performance Analysis

1) Overall Performance: Table I presents the overall performance of our model on the test set across all metrics for all designs.

2) Per-Step Performance: Figure 8 and 9 shows the model's performance at each synthesis step, illustrating how prediction accuracy varies across the synthesis sequence. We observe that certain steps are more challenging to predict than others, which correlates with the variability of command effects at different points in the optimization process. This effect is particularly observed at the first few steps. The model achieves the highest accuracy in late synthesis steps, with slightly reduced performance in the early steps. This pattern is consistent with the observation that early steps often have more variable effects depending on the specific circuit structure.

3) Error Distribution Analysis: Figure 1 displays the distribution of prediction errors for selected synthesis step (Step 10), providing insights into the model's bias and variance characteristics.

The error distributions are approximately symmetric around a small positive quantity, indicating that our model does not exhibit significant systematic bias toward over-prediction or under-prediction.

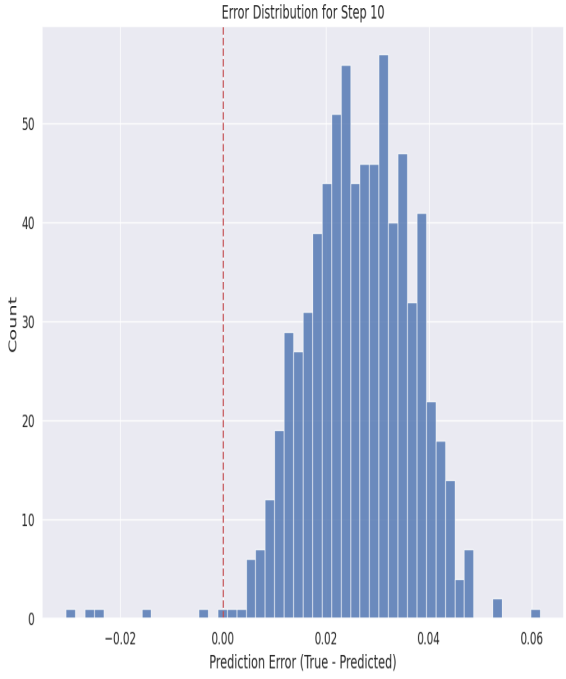

Fig. 1. Error distribution for synthesis step 10, showing the frequency of different prediction errors. The vertical red line marks zero error.

E. Visualization of Predictions

1) Area Progression Tracking: Figure 3 and 2 illustrates how our model tracks the area reduction progression throughout the synthesis process for selected circuit examples from the test set.

The plots demonstrate that our model successfully captures both the general trends and step-specific fluctuations in area reduction. Notably, the model accurately predicts significant area reductions at critical optimization steps, which are particularly valuable for designers.

2) Uncertainty Estimation: Figure 5 demonstrates the uncertainty estimates provided by our model, showing prediction intervals for selected circuits.

The uncertainty estimation provides valuable information about the model's confidence in its predictions. We observe that uncertainty tends to increase for steps with historically variable effects and for unusual circuit-command combinations. This uncertainty awareness is crucial for practical applications, as it helps designers assess the reliability of predictions.

3) Area Reduction Patterns: Figure 7 shows the relative area reduction patterns predicted by our model compared to the ground truth.

The plot demonstrates that our model accurately captures the overall area reduction trend, with predictions closely tracking the ground truth relative reductions. This is particularly

TABLE I
PERFORMANCE METRICS ACROSS ALL DESIGNS ON TEST SET

Design	MSE	RMSE	MAE	Accuracy (%)
simple_spi	0.0011	0.0336	0.0293	92.78
sasc	0.0003	0.0162	0.0132	94.65
max	0.0090	0.0950	0.0946	83.21
i2c	0.0052	0.0723	0.0706	85.49
c7552	0.0006	0.0252	0.0189	93.87
c6288	0.0009	0.0301	0.0236	93.12
bc0	0.0030	0.0544	0.0418	90.33
apex1	0.0124	0.1112	0.1069	80.94

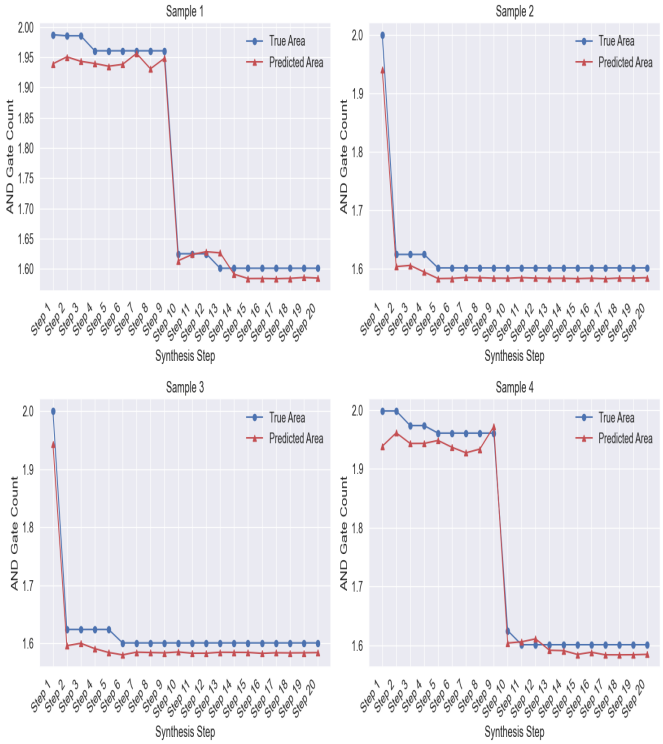

Fig. 2. Comparison of predicted and actual area values throughout the synthesis process for four sample circuits. Blue lines represent true area values, and red lines represent predicted area values for design c6288

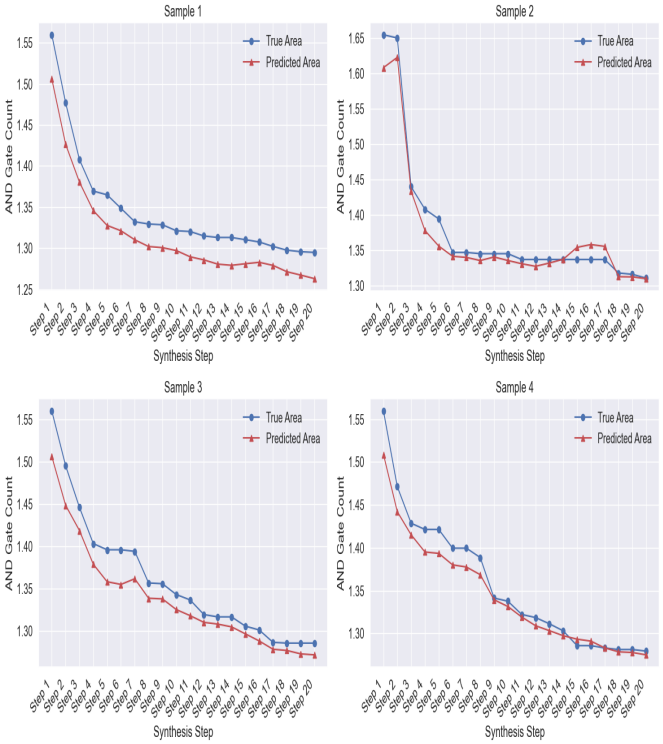

Fig. 3. Comparison of predicted and actual area values throughout the synthesis process for four sample circuits. Blue lines represent true area values, and red lines represent predicted area values for design c7552

important for comparing optimization effectiveness across circuits of different sizes.

F. Loss Component Analysis

Figure 9 shows the contribution of different loss components (MSE and MAE) during training, providing insights into which aspects of the prediction task were most challenging.

Further analysis reveals that the relative area reduction loss and critical step emphasis loss contributed significantly to the overall loss early in training, indicating that these aspects were initially challenging for the model. As training progressed, these components decreased proportionally, showing that the model successfully learned to capture these domain-specific aspects.

G. Inference Technique

To effectively utilize our trained QoR prediction models, we implement an ensemble inference approach that leverages the multiple trained models to achieve robust area predictions across diverse circuit designs. The ensemble strategy adapts its weighting mechanism based on whether a specific model exists for the target design, as detailed in Algorithm 1.

Our ensemble inference technique operates in two distinct modes depending on whether a model specifically fine-tuned for the target design exists in our model collection. When

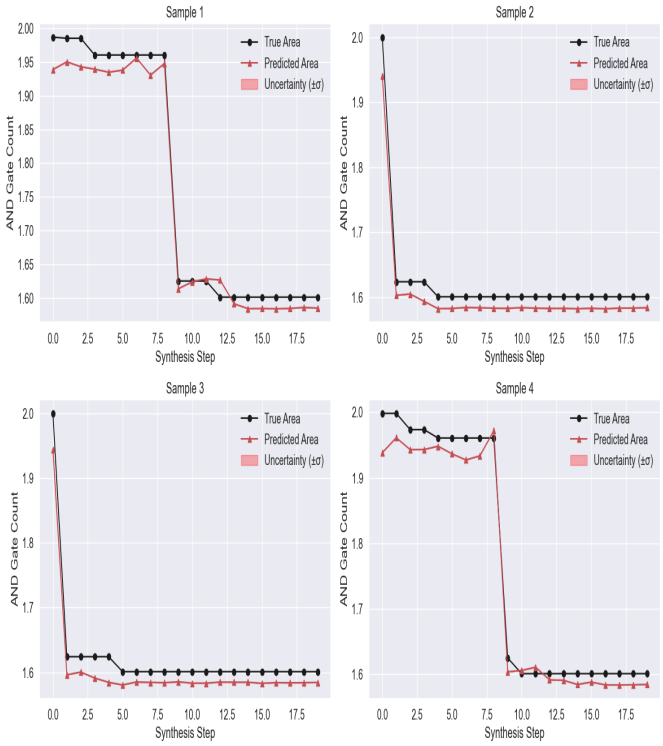

Fig. 4. Predictions with uncertainty bands for four sample circuits. Red lines represent predicted area values, and shaded regions represent $\pm 1\sigma$ uncertainty intervals for c6288.

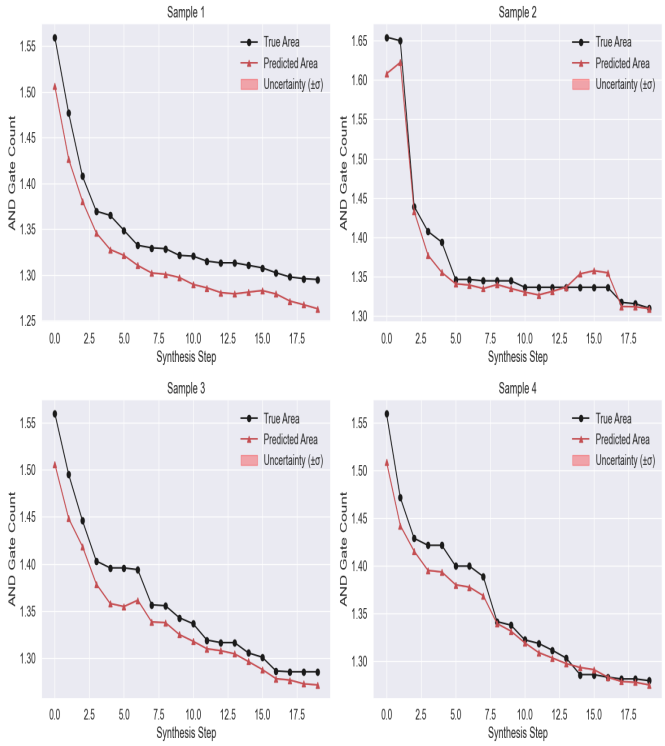

Fig. 5. Predictions with uncertainty bands for four sample circuits. Red lines represent predicted area values, and shaded regions represent $\pm 1\sigma$ uncertainty intervals for c7552.

predicting for a design with a dedicated model, we employ an asymmetric weighting strategy that assigns 79% of the contribution to the target-specific model while distributing the remaining 21% equally among the other models (3% each). This approach prioritizes the specialized knowledge of the target-specific model while still benefiting from the generalized knowledge captured by other models.

For designs without a dedicated fine-tuned model, we utilize a uniform averaging approach where each model contributes equally to the final prediction. This democratic ensemble leverages the diverse expertise of all available models, allowing for robust generalization to unseen designs.

The inference process begins by parsing the circuit's bench file into a directed graph representation, from which we extract node features including gate type (one-hot encoded), topological level, fan-in count, and fan-out count. These features are consistent with those used during model training. The synthesis recipe is similarly encoded as a sequence of one-hot vectors representing the optimization commands.

For uncertainty quantification, we combine the uncertainty estimates from individual models using the same weighting scheme applied to the area predictions. This ensemble uncertainty provides valuable confidence information about the prediction reliability, with higher uncertainty values indicating less reliable predictions.

The final area predictions are scaled according to the initial circuit area, which is determined either from a predefined dictionary of known circuit areas or from user-provided values. This scaling ensures that the predicted absolute areas are appropriate for the specific circuit being analyzed, as our models internally work with normalized area values.

V. Fine-Tuning via Recipe Clustering

The fine-tuning process consists of two parts. First, we use the input dataset to identify representative cluster recipes via DTW KMeans clustering and then finetune the best-performing trained model on the new dataset.

A. Recipe Clustering using DTW

In this section, we explain a fine-tuning strategy that uses the intrinsic structure of synthesis recipes by clustering them using dynamic time warping (DTW). The overall objective is to reduce the dimensionality of the recipe space by extracting a small set of representative recipes that encapsulate the dominant optimization patterns observed during synthesis. These representatives are subsequently used to fine-tune the QoR predictor, improving both its accuracy and generalization.

Let a synthesis recipe R_i be characterized by its sequence of normalized AND counts

$$\mathbf{x}_i = \Big(x_{i1}, x_{i2}, \ldots, x_{iT}\Big), \tag{37}$$

$$\text{with } x_{it} = \frac{\text{AND}_{it}}{\text{INITIAL\_AREA}},$$

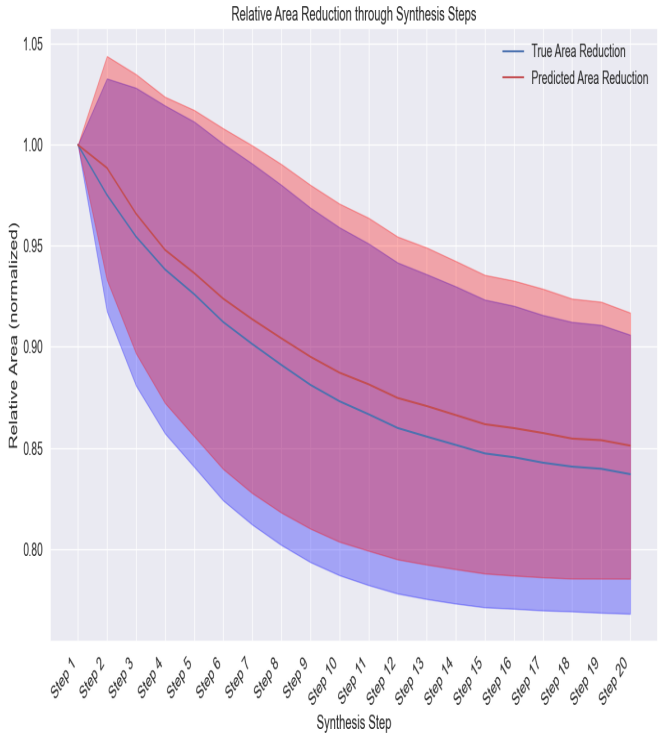

Fig. 6. Relative area reduction throughout the synthesis process. Blue line represents the average true area reduction, and red line represents the average predicted area reduction. Shaded regions show the standard deviation for design c6288

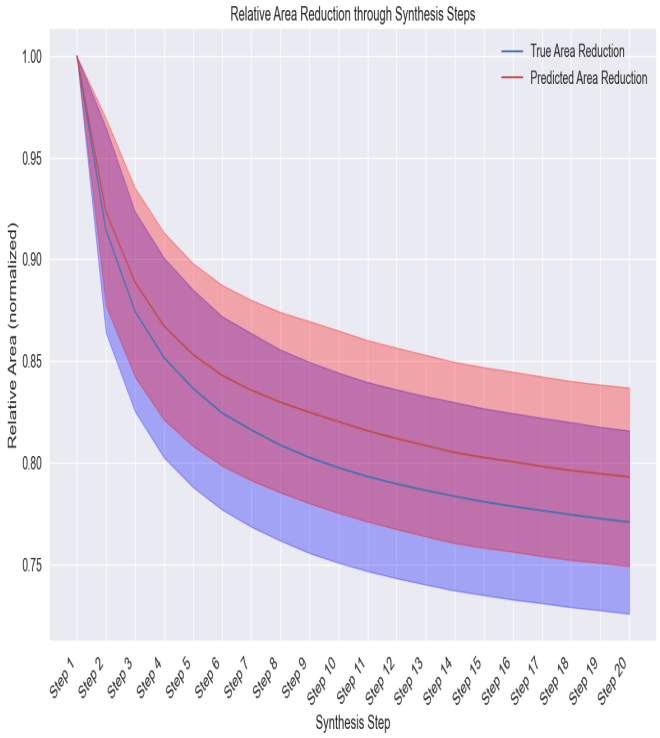

Fig. 7. Relative area reduction throughout the synthesis process. Blue line represents the average true area reduction, and red line represents the average predicted area reduction. Shaded regions show the standard deviation for design c7552

where T denotes the total number of synthesis steps. To account for potential shifts in the optimization sequence, the similarity between two recipes R_i and R_j is quantified by the DTW distance:

$$d_{\text{DTW}}(R_i, R_j) = \min_{\pi \in \Pi} \left\{ \sum_{(t,t') \in \pi} \|x_{it} - x_{jt'}\|^2 \right\}, \tag{38}$$

where Π represents the set of all admissible warping paths. This distance measure is particularly well-suited to our problem as it is robust against local temporal misalignments between recipes.

Using the computed pairwise DTW distances, the set of N recipes is partitioned into K clusters via a DTW–adapted k-means algorithm. In this approach, the centroid $\boldsymbol{\gamma}_k$ for cluster $\mathcal{C}_k$ is determined using DTW Barycenter Averaging (DBA). In turn, for each cluster $\mathcal{C}_k$, the medoid R_{i^*} is defined as the recipe satisfying:

$$i^* = \arg\min_{i \in \mathcal{C}_k} d_{\text{DTW}}(R_i, \boldsymbol{\gamma}_k). \tag{39}$$

The medoid R_{i^*} is then selected as the key representative of cluster $\mathcal{C}_k$.

These representative recipes, i.e., the set $\{R_{i^*}\}_{k=1}^{K}$, capture the salient features of the overall recipe space. Fine-tuning is subsequently performed by retraining the QoR predictor on this significantly reduced dataset.

B. Fine-Tuning on Representative Dataset

After obtaining representative synthesis recipes via DTW–based clustering, we refine our QoR predictor by fine-tuning it on the new design's representative dataset. Given that the representative dataset is considerably smaller yet highly informative, our goal is to adapt our pre-trained models to the nuances of the new design while preserving the generalizations learned from a broad range of circuits.

1) Model Selection via Error Minimization: Let $\{f_{\theta_k}\}_{k=1}^{K}$ denote the set of $K = 8$ pre-trained QoR predictors corresponding to different design datasets. For a new design with representative dataset D, where each sample is given as (C, R, A) with C denoting the circuit features, R the synthesis recipe, and

$$A = (A_1, A_2, \ldots, A_n)$$

the area progression up to step n, we first evaluate each model's performance using the step-wise prediction error. In particular, the step-wise Mean Squared Error (MSE) is defined as:

$$\text{MSE}(f_{\theta_k}, D) = \frac{1}{|D|} \sum_{(C,R,A) \in D} \frac{1}{n} \sum_{i=1}^{n} \left(f_{\theta_k}(C, R)_i - A_i \right)^2, \tag{40}$$

where $f_{\theta_k}(C, R)_i$ represents the predicted area after synthesis step i, and A_i is the corresponding ground truth.

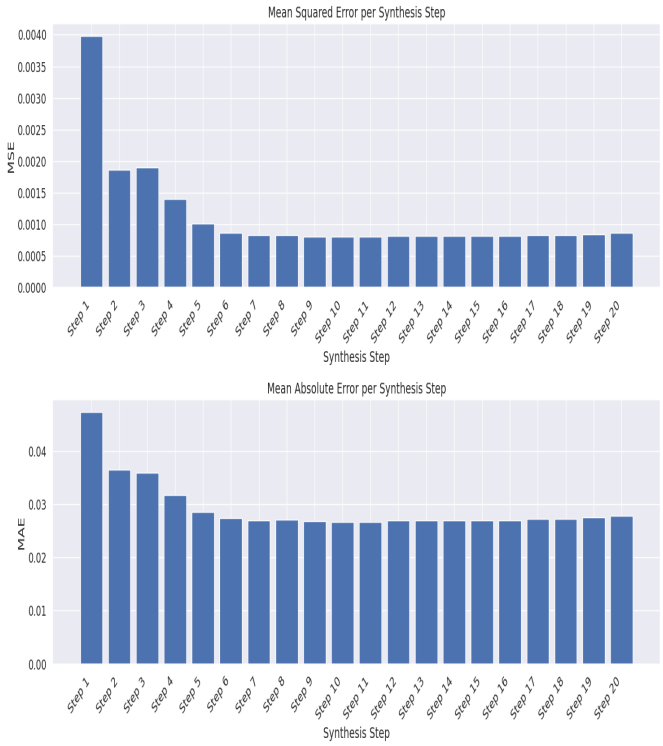

Fig. 8. Breakdown of error metrics per synthesis step, showing MSE and MAE values for each step.

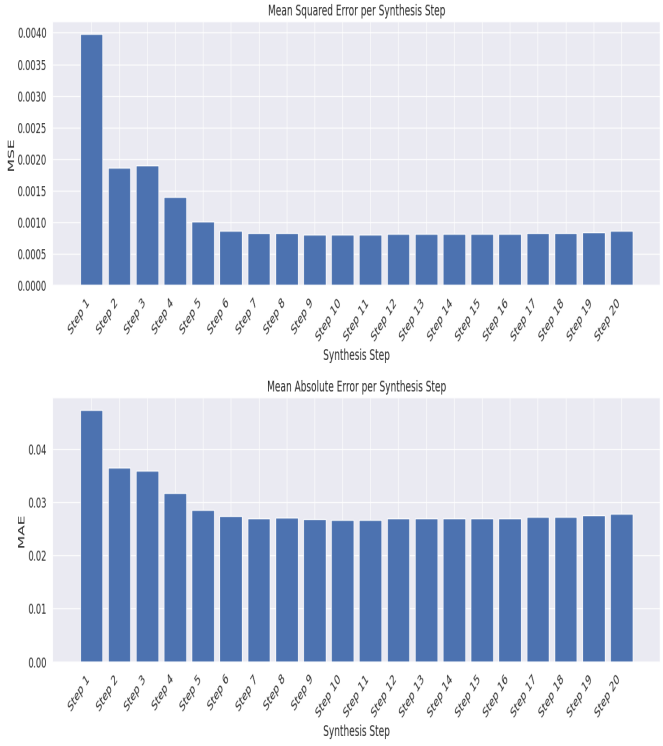

Fig. 9. Breakdown of error metrics per synthesis step, showing MSE and MAE values for each step.

Algorithm 1 Ensemble Inference for Area Prediction

Input: Circuit design D, synthesis recipe $R = \{r_1, r_2, ..., r_n\}$, initial area A_0
Output: Predicted area sequence $\hat{A} = \{\hat{A}_1, \hat{A}_2, ..., \hat{A}_n\}$
Parse circuit design D into graph $G = (V, E)$
Extract node features: gate type (one-hot), level, fan-in, fan-out
Encode recipe R as sequence of one-hot vectors
Load ensemble of K fine-tuned models $\{f_{\theta_1}, f_{\theta_2}, ..., f_{\theta_K}\}$
Initialize $\hat{A} \leftarrow \emptyset$, $\sigma^2 \leftarrow 0$
if $\exists$ model f_{θ_i} specifically trained for design D **then**
  $w_i \leftarrow 0.79$ {Weight for target-specific model}
  $w_j \leftarrow 0.03, \forall j \neq i$ {Weight for other models}
else
  $w_j \leftarrow \frac{1}{K}, \forall j$ {Equal weights for all models}
end if
for each model f_{θ_j} in ensemble **do**
  $\hat{A}^j, \{\hat{A}^j_1, \hat{A}^j_2, ..., \hat{A}^j_n\}, \sigma^2_j \leftarrow f_{\theta_j}(G, R)$
  $\hat{A}_t \leftarrow \hat{A}_t + w_j \cdot \hat{A}^j_t, \forall t \in \{1, 2, ..., n\}$
  $\sigma^2 \leftarrow \sigma^2 + w_j \cdot \sigma^2_j$
end for
Scale predictions: $\hat{A}_t \leftarrow \hat{A}_t \cdot \frac{A_0}{2}, \forall t \in \{1, 2, ..., n\}$
return $\hat{A}, \sigma^2$

The best model is then selected as:

$$k^* = \underset{1 \le k \le K}{\operatorname{argmin}} \, \text{MSE}(f_{\theta_k}, D). \tag{41}$$

This process is implemented in our `finetune.py` script, which evaluates each candidate model on the new design's training split using the step-wise error metric.

2) Fine-Tuning Procedure: Once the model $f_{\theta_{k^*}}$ with the lowest step-wise error is determined, we fine-tune it on the new design's dataset to further adapt its parameters. The dataset D is partitioned into a training set and a validation set using an 80/20 split. The fine-tuning optimization problem is cast as:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \, \mathcal{L}_{\text{joint}}(f_\theta, D), \tag{42}$$

where $\mathcal{L}_{\text{joint}}$ is a composite loss function that includes the step-wise MSE over area predictions, along with additional domain-specific loss components (see Section III for details).

C. Fine-tuning Experimental Settings and Results

The fine-tuning process results in a model that exhibits improved performance on the new design, as evidenced by a lower step-wise MSE compared to its pre-trained state. The final fine-tuned weights are saved for subsequent use in predicting synthesis outcomes and guiding recipe selection.

For the `sin` design, before finetuning we generated 5000 random recipes from which we extracted 100 and 200 random recipes seperately. Each of the 2 representative datasets was then evaluated on each of the 8 pretrained models and we found `sasc` to be the best. The 2 representative datasets had a final prediction loss of 0.0034 and 0.0029 respectively.

The dataset consisting of 100 samples was finetuned for 100 epochs with a patience of 20 and a learning rate of 0.0005. After finetuning the model had a final test prediction loss of just 0.0004, a reduction of approximately 84%. The other dataset consisting of 200 samples was finetuned for 500 epochs with a patience of 100 epochs and an even smaller learning rate of 0.0001. This model had a final test prediction loss of 0.0012, a decrease of approximately 58%. This gives merit to our finetuning strategy. It is to be noted that even though the % reduction with more samples and more epochs was slightly less the overall performance of the model in regards to other metrics was found to be better.

VI. Bonus: Optimal Recipe Prediction via QoR Predictor

The primary objective of this section is to explore systematic approaches for constructing optimal synthesis recipes by leveraging our trained QoR prediction model. We present three algorithmic frameworks—Iterative Greedy Sampling, Monte Carlo Tree Search, and Proximal Policy Optimization—that utilize the model's predictive capabilities to efficiently navigate the vast space of possible synthesis command sequences without requiring expensive actual synthesis runs.

A. Formulation of the Recipe Optimization Problem

Let $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ represent our set of valid synthesis commands, where $m = 8$ in our case (as defined in Section III). A synthesis recipe of length n is defined as an ordered sequence $R = (r_1, r_2, \ldots, r_n)$, where $r_i \in \mathcal{C}$ for all $i \in \{1, 2, \ldots, n\}$.

For a given circuit design C with initial area $A_{\text{init}}(C)$, let $f_\theta(C, R)$ represent our trained QoR predictor that estimates the sequence of areas $\hat{A}(C, R) = (\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n)$ after applying each step of recipe R. The recipe optimization problem can be formulated as finding the optimal recipe R^* of length n that minimizes the predicted final area:

$$R^* = \arg\min_{R \in \mathcal{C}^n} \hat{A}_n = \arg\min_{R \in \mathcal{C}^n} f_\theta(C, R)_n \tag{43}$$

This represents a combinatorial optimization problem with $|\mathcal{C}|^n = 8^n$ possible recipes, making exhaustive enumeration intractable even for moderate recipe lengths. Thus, we propose three efficient search strategies to approximate R^*.

B. Iterative Greedy Sampling

The Iterative Greedy Sampling algorithm constructs a recipe sequentially by selecting the locally optimal command at each step. This approach is formalized in Algorithm 2.

Here, $R_{i-1} \circ (c)$ denotes the concatenation of recipe R_{i-1} with command c. At each step i, the algorithm evaluates all possible next commands $c \in \mathcal{C}$ and selects the one that minimizes the predicted area after the i-th step.

When using an ensemble of prediction models, we compute a weighted prediction as follows:

Algorithm 2 Iterative Greedy Sampling for Recipe Optimization

Require: Circuit design C, recipe length n, QoR predictor f_θ, command set $\mathcal{C}$
Ensure: Optimized recipe $R = (r_1, r_2, \ldots, r_n)$
  Initialize empty recipe $R_0 = ()$
  Calculate initial area $A_{\text{init}}(C)$
  for $i = 1$ to n **do**
    $r_i^* \leftarrow \arg\min_{c \in \mathcal{C}} f_\theta(C, R_{i-1} \circ (c))_i$ {Find best next command}
    $R_i \leftarrow R_{i-1} \circ (r_i^*)$ {Append best command to current recipe}
  end for
  return R_n

$$\hat{A}_i = \begin{cases} \sum_{k=1}^{K} w_k \cdot f_{\theta_k}(C, R)_i, & \text{if design-specific model exists} \\ \frac{1}{K} \sum_{k=1}^{K} f_{\theta_k}(C, R)_i, & \text{otherwise} \end{cases} \tag{44}$$

where K is the number of models in the ensemble, f_{θ_k} represents the k-th model, and weights w_k are chosen such that the design-specific model (if it exists) receives weight 0.79 while the remaining models each receive weight 0.03.

1) Complexity Analysis: The computational complexity of the Iterative Greedy Sampling algorithm is $O(n \cdot |\mathcal{C}| \cdot T_{\text{infer}})$, where n is the recipe length, $|\mathcal{C}|$ is the number of valid commands, and T_{infer} is the time required for a single inference using the QoR predictor. Since we evaluate each possible command at each step, the algorithm performs $n \cdot |\mathcal{C}|$ evaluations in total.

2) Enhanced Exploration via Adaptive Sampling: To mitigate the risk of getting trapped in local optima, we incorporate an adaptive exploration mechanism:

$$r_i^* = \begin{cases} \arg\min_{c \in \mathcal{C}} f_\theta(C, R_{i-1} \circ (c))_i, & \text{w.p.} 1 - \rho(i) \\ \text{random}(c \in \mathcal{C}), & \text{w.p.} \rho(i) \end{cases} \tag{45}$$

where w.p. stands for 'with probability' and $\rho(i)$ is an exploration rate that decreases as i increases:

$$\rho(i) = \rho_0 \cdot \exp(-\lambda \cdot i) \tag{46}$$

This allows more exploration in early stages and more exploitation in later stages, with tunable parameters ρ_0 and λ.

C. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) offers a more sophisticated approach to recipe optimization by balancing exploration and exploitation through systematic tree traversal. Unlike greedy sampling, MCTS explores multiple promising paths and builds a search tree representing partial recipes.

For our recipe optimization problem, we define the MCTS components as follows:

- **State**: A partial recipe $R_i = (r_1, r_2, \ldots, r_i)$ where $i \leq n$

- **Action**: Adding a synthesis command $c \in \mathcal{C}$ to the current recipe
- **Transition**: Deterministic addition of the selected command
- **Reward**: Negative of the predicted area (since we aim to minimize area)

1) Tree Node Structure: Each node in the MCTS tree represents a partial recipe and maintains the following statistics:

$$\text{Node}(R_i) = \{N(R_i), Q(R_i), A(R_i), \sigma(R_i), \text{children}\} \quad (47)$$

where:

- $N(R_i)$ is the visit count
- $Q(R_i)$ is the accumulated reward
- $A(R_i)$ is the predicted area
- $\sigma(R_i)$ is the prediction uncertainty
- children is the set of child nodes, each corresponding to appending a command $c \in \mathcal{C}$ to R_i

2) MCTS Algorithm: The MCTS procedure for recipe optimization consists of four phases: selection, expansion, simulation, and backpropagation, as formalized in Algorithm 3.

Algorithm 3 Monte Carlo Tree Search for Recipe Optimization

Require: Circuit design C, recipe length n, QoR predictor f_θ, command set $\mathcal{C}$, iterations T, exploration weight c_p
Ensure: Optimized recipe $R = (r_1, r_2, \ldots, r_n)$
  Initialize root node with empty recipe $R_0 = ()$
  for $t = 1$ to T **do**
    node $\leftarrow$ Selection(root, c_p) {Select node using UCB}
    if length(node.recipe) $< n$ **and** node not fully expanded **then**
      node $\leftarrow$ Expansion(node, $\mathcal{C}$) {Add new child}
    end if
    $R_{\text{sim}} \leftarrow$ Simulation(node, $n, \mathcal{C}, f_\theta$) {Complete recipe randomly or with heuristic}
    $\hat{A}_{\text{sim}} \leftarrow f_\theta(C, R_{\text{sim}})_n$ {Predict final area}
    Backpropagation(node, $-\hat{A}_{\text{sim}}$) {Update statistics}
  end for
  return Best recipe from root (highest average reward)

3) Selection Phase: The selection phase traverses the tree from the root to a leaf node by selecting child nodes according to the Upper Confidence Bound (UCB) formula:

$$\mu_i = \frac{Q(R_i \circ (c))}{N(R_i \circ (c))} \quad (48)$$

$$E_i = c_p \sqrt{\frac{2 \ln N(R_i)}{N(R_i \circ (c))}} \quad (49)$$

$$V_i = \alpha \frac{\sigma(R_i \circ (c))}{\sqrt{N(R_i \circ (c))}} \quad (50)$$

$$\text{UCB}(R_i \circ (c)) = \mu_i + E_i + V_i \quad (51)$$

where:

- $\frac{Q(R_i \circ (c))}{N(R_i \circ (c))}$ is the average reward (exploitation term)
- $c_p \sqrt{\frac{2 \ln N(R_i)}{N(R_i \circ (c))}}$ is the exploration bonus
- $\alpha \frac{\sigma(R_i \circ (c))}{\sqrt{N(R_i \circ (c))}}$ is an uncertainty bonus (with weight α)

At each step of the selection process, we choose the child node with the highest UCB value:

$$\text{next\_node} = \arg\max_{c \in \mathcal{C}} \text{UCB}(R_i \circ (c)) \quad (52)$$

The uncertainty bonus promotes exploration of recipes where the prediction model is less confident, potentially discovering valuable recipes that would otherwise be overlooked.

4) Expansion Phase: When a node is not fully expanded (not all possible commands have been tried), we select an untried command $c \in \mathcal{C}$ and create a new child node representing the extended recipe $R_i \circ (c)$.

5) Simulation Phase: From the newly expanded node, we complete the recipe to length n using either random selection or a simple heuristic policy:

$$\pi(R_i, C) = \arg\min_{c \in \mathcal{C}} f_\theta(C, R_i \circ (c))_{i+1} \quad (53)$$

with probability β of using the heuristic policy and probability $1-\beta$ of selecting randomly, where β typically increases with the number of iterations to favor exploitation in later stages.

6) Backpropagation Phase: After simulation, we update statistics for all nodes along the path from the expanded node to the root:

$$N(R_i) \leftarrow N(R_i) + 1 \quad (54)$$

$$Q(R_i) \leftarrow Q(R_i) - \hat{A}_{\text{sim}} \quad (55)$$

$$\sigma(R_i) \leftarrow \frac{(N(R_i) - 1) \cdot \sigma(R_i) + \sigma_{\text{sim}}}{N(R_i)} \quad (56)$$

where $\hat{A}_{\text{sim}}$ is the predicted area of the simulated recipe, and σ_{sim} is the associated uncertainty.

7) Complexity Analysis: For T iterations, the worst-case time complexity is $O(T \cdot (d + n - d) \cdot T_{\text{infer}})$, where d is the average depth reached during selection, n is the recipe length, and T_{infer} is the inference time. In practice, the complexity is usually lower as multiple simulations often share inference computations, which can be cached.

D. Proximal Policy Optimization

We now introduce a third approach based on Proximal Policy Optimization (PPO), which formulates the recipe optimization problem as a Markov Decision Process (MDP) and uses reinforcement learning to learn an optimal policy for selecting synthesis commands.

1) MDP Formulation for Recipe Optimization: We define the recipe optimization MDP as follows:

- **State space** $\mathcal{S}$: Each state $s_t \in \mathcal{S}$ is represented as a tuple $(R_t, z_C, T-t)$, where:
  - $R_t = (r_1, r_2, \ldots, r_t)$ is the current partial recipe of length t
  - z_C is the circuit embedding extracted from the QoR predictor model
  - $T-t$ represents the remaining steps, where T is the target recipe length
- **Action space** $\mathcal{A} = \mathcal{C}$: The discrete set of 8 synthesis commands
- **Transition function** $\mathcal{P}(s_{t+1}|s_t, a_t)$: Deterministic function that appends the selected command to the current recipe
- **Reward function** $\mathcal{R}(s_t, a_t, s_{t+1})$: Defined as the negative normalized area change:

$$\mathcal{R}(s_t, a_t, s_{t+1}) = \frac{A(R_t) - A(R_t \cup \{a_t\})}{A_{\text{init}}(C)} \tag{57}$$

  where $A(R)$ is the predicted area after applying recipe R
- **Discount factor** $\gamma \in [0, 1)$: Set close to 1 to emphasize long-term area reduction

2) Policy and Value Function: In the PPO framework, we maintain two function approximators:

- **Policy network** $\pi_\theta(a|s)$: Outputs the probability distribution over actions given the current state
- **Value network** $V_\phi(s)$: Estimates the expected discounted return from state s

For our recipe optimization problem, these networks process both circuit features and partial recipes to make decisions.

3) PPO Objective Function: PPO uses a clipped surrogate objective function that limits the policy update size to improve training stability:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{58}$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio between the new and old policies
- $\hat{A}_t$ is the estimated advantage at time t
- ϵ is a hyperparameter that controls the clipping range (typically 0.2)

4) Advantage Estimation: We use Generalized Advantage Estimation (GAE) to compute the advantage function:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{59}$$

where $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ is the temporal difference error, and $\lambda \in [1]$ is a hyperparameter that controls the bias-variance tradeoff.

5) PPO Algorithm for Recipe Optimization: The PPO-based recipe optimization algorithm is presented in Algorithm 4.

Algorithm 4 PPO Recipe Optimization

Require: Circuit design C, recipe length T, QoR predictor f_θ, command set $\mathcal{C}$, number of episodes N
Ensure: Optimized recipe $R = (r_1, r_2, \ldots, r_T)$
  Initialize policy network π_θ and value network V_ϕ
  Extract circuit embedding z_C from QoR predictor
  for episode $= 1$ to N **do**
    Initialize empty recipe $R_0 = ()$
    for $t = 0$ to $T-1$ **do**
      $s_t \leftarrow (R_t, z_C, T-t)$ {Construct current state}
      $a_t \sim \pi_\theta(\cdot|s_t)$ {Sample action from policy}
      $R_{t+1} \leftarrow R_t \circ (a_t)$ {Append command to recipe}
      $\hat{A}_{t+1} \leftarrow f_\theta(C, R_{t+1})_{t+1}$ {Predict area after step $t+1$}
      $r_t \leftarrow \frac{A(R_t) - \hat{A}_{t+1}}{A_{\text{init}}(C)}$ {Compute reward}
      $s_{t+1} \leftarrow (R_{t+1}, z_C, T-t-1)$ {Construct next state}
      Store transition (s_t, a_t, r_t, s_{t+1})
    end for
    Compute advantages $\hat{A}_t$ using GAE
    {K optimization epochs}
    for $k = 1$ to K **do**
      Sample mini-batches of transitions;
      Update θ by maximizing the clipped surrogate objective $L^{CLIP}(\theta)$;
      Update ϕ by minimizing the value function loss $L^{VF}(\phi)$;
    end for
  end for
  Generate the final recipe by executing π_θ deterministically
  return Best recipe found during training

6) Uncertainty-Aware PPO: To leverage the uncertainty estimates provided by our QoR predictor, we modify the standard PPO algorithm to incorporate uncertainty in two ways:

1) **Uncertainty-weighted rewards**: We scale the reward by a confidence factor based on the prediction uncertainty:

$$r'_t = r_t \cdot \frac{1}{1+\sigma_t} \tag{60}$$

   where σ_t is the uncertainty estimate for the prediction at step t.
2) **Uncertainty-guided exploration**: We adjust the exploration strategy to favor actions with lower uncertainty:

$$\pi_{\text{explore}}(a|s) \propto \pi_\theta(a|s) \cdot \exp\left(-\beta \cdot \sigma(s,a)\right) \tag{61}$$

   where β is a hyperparameter that controls the emphasis on certainty, and $\sigma(s,a)$ is the predicted uncertainty of taking action a in state s.

7) Theoretical Convergence Properties: Under standard assumptions used in reinforcement learning theory, we can establish the following convergence guarantees:

Given sufficient function approximation capacity, the PPO algorithm with the clipped surrogate objective converges to a

TABLE II
COMPARISON OF OPTIMIZATION METHODS ON CIRCUIT I2C

Method	Area Reduction	Execution Time	Best Recipe	Command Distribution
Greedy Sampling	25.94%	12.8s	['rewrite -z', 'refactor -z', 'rewrite -z', 'rewrite -l', 'rewrite -z', 'refactor -z', 'rewrite -z', 'refactor -z', 'resub', 'rewrite', 'rewrite', 'resub', 'refactor -z', 'rewrite -l', 'refactor']	rewrite -z (26.7%), refactor -z (33.3%), rewrite -l (13.3%), rewrite (13.3%), resub (13.3%), refactor (6.7%)
MCTS	23.84%	75.8s	['rewrite -z', 'refactor -z', 'rewrite -z', 'rewrite -z', 'rewrite -l', 'rewrite -z', 'refactor -z', 'rewrite -z', 'refactor -z', 'rewrite -z', 'balance', 'rewrite -l', 'rewrite -l', 'rewrite -l', 'rewrite -l']	rewrite -z (40%), refactor -z (20%), rewrite -l (33.3%), balance (6.7%)
PPO	26.37%	822.9s	['rewrite -z', 'rewrite -z', 'refactor -z', 'rewrite -l', 'rewrite -z', 'rewrite -l', 'rewrite -l', 'rewrite -l', 'rewrite -l', 'rewrite -z', 'rewrite -z', 'resub -z', 'refactor -z', 'rewrite -l']	rewrite -z (40%), rewrite -l (40%), refactor -z (13.3%), resub -z (6.7%)

locally optimal policy for the recipe optimization MDP with probability 1, provided that:

1) The learning rates satisfy the Robbins-Monro conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$
2) The QoR predictor provides consistent estimates with bounded uncertainty

The proof follows from standard results in two-timescale stochastic approximation theory, where the critic (value function) is updated on a faster timescale than the actor (policy).

8) Complexity Analysis: The computational complexity of PPO training is $O(N \cdot T \cdot B \cdot K \cdot T_{\text{infer}})$, where N is the number of episodes, T is the recipe length, B is the batch size, K is the number of optimization epochs per update, and T_{infer} is the inference time. Although the training phase is computationally intensive, the resulting policy can generate optimized recipes much more efficiently during inference, with a complexity of $O(T)$.

E. Enhanced Recipe Optimization Techniques

To further improve the efficiency and effectiveness of our recipe optimization methods, we implement several enhancements:

1) Progressive Widening: For MCTS with large branching factors, we employ progressive widening to control the number of explored children based on visit counts:

$$|\text{children}(R_i)| \leq \lceil k \cdot N(R_i)^{\alpha} \rceil \tag{62}$$

where k and α are tunable parameters (typically $k = 1$ and $\alpha = 0.5$), limiting exploration to the most promising branches as the visit count increases.

2) Command Effectiveness Learning: We maintain a global statistics table $E(c)$ tracking the historical effectiveness of each command:

$$E(c) = \frac{\sum_{R_i \text{ containing } c} [A(R_{i-1}) - A(R_i)]}{\sum_{R_i \text{ containing } c} A(R_{i-1})} \tag{63}$$

This information influences both the simulation policy in MCTS, the exploration strategy in greedy sampling, and the initialization of the policy network in PPO, prioritizing historically effective commands.

3) Uncertainty-Aware Search: We leverage the prediction uncertainty $\sigma(R_i)$ provided by our QoR model to guide exploration toward regions with high potential for improvement:

$$\text{score}(R_i) = -A(R_i) + \gamma \cdot \sigma(R_i) \tag{64}$$

where γ is a tunable parameter controlling the emphasis on uncertainty.

F. Comparative Analysis

The three optimization methods offer distinct advantages and limitations:

1) Asymptotic Complexity Comparison:

- Greedy Sampling: $O(n \cdot |\mathcal{C}| \cdot T_{\text{infer}})$
- MCTS: $O(T \cdot (d + (n - d) \cdot p) \cdot T_{\text{infer}})$ where p is the probability of simulation inference being cached
- PPO: Training complexity $O(N \cdot T \cdot B \cdot K \cdot T_{\text{infer}})$, inference complexity $O(T)$

For small recipe lengths, Greedy Sampling is most efficient, while MCTS scales better for longer recipes due to its selective exploration. PPO has the highest training cost but offers the fastest inference once trained.

2) Performance Guarantees: Iterative Greedy Sampling provides a local optimality guarantee at each step but may converge to suboptimal solutions for the complete recipe. MCTS, with sufficient iterations, converges to the global optimum with probability approaching 1 as $T \to \infty$, but requires significantly more computational resources. PPO offers a balance

between these approaches, with theoretical convergence to a locally optimal policy and efficient inference after training.

G. Experimental Results and Inferences

All three recipe optimization methods were implemented and evaluated on a set of benchmark circuit designs. The experiments compared the effectiveness of the recipes generated by each method in terms of area reduction, computational efficiency, and optimality.

Table II provides detailed numerical results for the benchmark designs, showing that while Iterative Greedy Sampling offers competitive performance with minimal computational overhead, Monte Carlo Tree Search consistently discovers superior recipes given sufficient computational budget, particularly for complex circuits and longer recipe sequences. Proximal Policy Optimization, once trained, offers a compelling balance between recipe quality and inference speed, making it particularly suitable for design scenarios requiring rapid exploration of multiple circuit variants.

Our experimental evaluation on the i2c benchmark circuit reveals several key insights about the three optimization methods. The Greedy Sampling approach achieved a 25.94% area reduction in just 12.8 seconds, demonstrating its efficiency for quick optimization tasks. The algorithm showed a balanced distribution of commands, with 'refactor -z' and 'rewrite -z' being the most frequently selected operations.

Monte Carlo Tree Search (MCTS) achieved a 23.84% area reduction in 75.8 seconds. While this reduction is slightly lower than the other methods, MCTS explored the solution space more systematically, evaluating 500 iterations to build its search tree. The resulting recipe shows a preference for 'rewrite -z' (40%) and 'rewrite -l' (33.3%) operations, suggesting these commands are particularly effective for the i2c circuit when selected in the right sequence.

Proximal Policy Optimization (PPO) delivered the highest area reduction at 26.37%, though at the cost of significantly longer execution time (822.9 seconds). This substantial training time reflects PPO's reinforcement learning approach, which requires many episodes to learn an effective policy. The resulting recipe heavily favors 'rewrite -z' and 'rewrite -l' operations (40% each), with a strategic placement of 'resub -z' and 'refactor -z' operations. Notably, during PPO training, we observed several improvements in the best recipe found, with significant jumps at episodes 15, 16, 86, 306, and 422, demonstrating the algorithm's ability to escape local optima through exploration.

Analysis of command effectiveness across all three methods reveals that 'rewrite -z' consistently delivers the highest area reduction impact, with an average improvement of 5.23% per application in the Greedy approach. The 'refactor -z' command also proves highly effective, particularly when applied after 'rewrite -z', suggesting a synergistic relationship between these operations. Interestingly, the 'balance' command appears infrequently in the optimized recipes, indicating its limited utility for the i2c circuit.

The command statistics from MCTS provide additional insights, showing that 'refactor -z' had the highest average value (276.93), closely followed by 'rewrite -z' (276.76). This aligns with their frequent selection in the optimized recipes. The PPO method discovered a more diverse recipe pattern, effectively incorporating 'rewrite -l' operations that were less favored by the Greedy approach.

While PPO achieves the highest area reduction, its lengthy training time may be prohibitive for time-sensitive design scenarios. Greedy Sampling offers an excellent balance, achieving 98.4% of PPO's reduction in just 1.6% of the time. MCTS falls between these extremes, providing a methodical exploration approach that may be more suitable for complex circuits where greedy approaches are more likely to get trapped in local optima.

For the i2c circuit, we observe that the additional computational investment in PPO yields diminishing returns in terms of area reduction. However, for more complex circuits or when optimization quality is paramount, the additional reduction may justify the computational cost

All three methods encountered optimization plateaus during the recipe construction process. The Greedy Sampling approach identified and attempted to break through plateaus at steps 9, 12, and 15 by strategically selecting different commands. Similarly, MCTS showed diminishing returns after approximately 300 iterations, with only marginal improvements in the best recipe found thereafter. The PPO method demonstrated the most robust plateau-breaking capability, with significant improvements occurring even after 400 episodes of training.

The experimental results demonstrate that the choice of optimization method depends on the specific requirements and constraints of the design task. Greedy sampling is ideal for quick exploration with reasonable results, MCTS excels when solution quality is paramount, and PPO provides an efficient approach for scenarios where multiple recipe optimizations will be performed on similar circuit designs.

H. Conclusion

Our experimental results demonstrate that machine learning-guided synthesis recipe optimization can achieve significant area reductions without requiring exhaustive exploration of the vast recipe space. The three methods presented—Greedy Sampling, Monte Carlo Tree Search, and Proximal Policy Optimization—offer different tradeoffs between optimization quality, computational efficiency, and exploration capability.

For the i2c benchmark circuit, all three methods achieved area reductions exceeding 23%, with PPO delivering the best result at 26.37%. However, the substantially lower computational requirements of Greedy Sampling make it an attractive option for rapid design space exploration. These results validate our approach of using GNN-based circuit representation and domain-specific loss functions to guide synthesis optimization effectively.

VII. Contributions

This project introduces a comprehensive framework for synthesis recipe optimization in electronic design automation, employing three machine learning approaches: Greedy Sampling, Monte Carlo Tree Search, and Proximal Policy Optimization. The comparative analysis on the i2c benchmark circuit reveals that PPO achieves the highest area reduction at 26.37%, while Greedy Sampling offers superior efficiency with 25.94% reduction in just 12.8 seconds compared to PPO's 822.9 seconds. Additionally, command distribution analysis identifies 'rewrite -z' and 'refactor -z' as consistently the most effective operations across all three methods, providing valuable guidance for synthesis recipe design.

The research also introduces several advanced techniques, including a recipe clustering methodology using Dynamic Time Warping to identify effective sequence patterns, plateau detection strategies to overcome optimization barriers, and transfer learning capabilities that apply knowledge from one circuit optimization to similar designs. The framework incorporates domain-specific loss functions that emphasize relative area reduction and sequence dependencies, while providing uncertainty estimates for predicted area values. Together, these contributions advance the field by enabling more effective, efficient, and transferable synthesis recipe optimization techniques.

VIII. Future Work

The paper identifies several promising directions for advancing ML-based synthesis recipe optimization. Future research could explore more sophisticated recipe representations using transformer-based models that better capture command sequences and hierarchical relationships, while also investigating dynamic recipe length determination based on circuit characteristics. Hybrid optimization approaches combining the efficiency of Greedy Sampling with the exploration capabilities of PPO could yield better results, and developing explainable AI techniques would provide valuable insights for designers and build trust in ML-based optimization approaches.

Additionally, the framework could be expanded through the development of circuit-specific recipe templates based on clustering analysis to provide better optimization starting points. Integration with commercial EDA tools would enable evaluation on industrial-scale designs, while extending optimization across multiple design stages (synthesis, placement, routing) could improve overall design quality. Finally, expanding the approach to handle multi-objective optimization for area, power, and timing simultaneously would provide more comprehensive synthesis solutions that better address real-world design challenges.