An Investigation into Reinforcement Learning in FPGA Placement Optimization

by

Ruichen Chen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

Abstract

With the increasing complexity and capacity of modern Field-Programmable Gate Arrays (FPGAs), there is a growing demand for efficient FPGA computer-aided design (CAD) tools, particularly at the placement stage. While some previous works, such as *RLPlace*, have explored the efficacy of single-state Reinforcement Learning (RL) to optimize FPGA placement by framing it as a multi-armed bandit (MAB) problem, numerous AI techniques remain unexplored due to the outstanding engineering challenges of integrating them into the FPGA CAD flow, which is implemented typically using C++. In this thesis, we propose VPR-Gym, a Python environment built on OpenAI Gym that allows seamless integration with various machine learning libraries, including PyTorch, TensorFlow, and Nevergrad, while enabling the comparison between different AI techniques for FPGA placement optimization. To determine the optimal RL algorithm for FPGA placement, we perform regret analysis and non-stationary analysis of MAB algorithms used in FPGA placement optimization. Moreover, we introduce a learning objective that reformulates the MAB problem as an optimization problem, thereby expanding the range of AI techniques that can be investigated beyond those for MAB problems. To investigate the effectiveness of different algorithms in FPGA placement and thus showcase the capabilities of our VPR-Gym platform, we conduct experiments that compare the performance of various MAB algorithms and evolution strategy (ES) algorithms. Our findings demonstrate that the ES approaches exhibit superior performance over the existing MAB approaches, highlighting the effectiveness of VPR-Gym in facilitating AI research to enhance FPGA placement.

Preface

A portion of content in this thesis, including Chapters 3, 4, 5 and 6, has been accepted for presentation at the International Conference on Field-Programmable Logic and Applications (FPL) in Sweden, at September 2023.

Acknowledgements

I would like to thank Dr. Niu for helping me through my research. His supervision helped me greatly overcome the difficulties encountered during my research. Furthermore, his guidance offered me lasting insights on how to become a good researcher.

I would also like to thank my family, who offered me support through these years. They have offered me encouragement during these times, allowing me to pursue my research with full effort.

Table of Contents

List of Tables

List of Figures

List of Symbols

Latin

A, a	Action
N	Number of action
P	Probability
Q	Action value
R, r	Reward
T, t	Rounds
V	Control vector
Z	Gumbel distribution

Greek

α	Confidence parameter
ϵ	Exploration rate
γ	Discount Factor
μ	Mean values associated with the reward distributions
ρ	Regret
τ	Temperature parameter

Abbreviations

ASIC Application-specific integrated circuits.

BE Boltzmann Exploration.

BGE Boltzmann–Gumbel Exploration.

CAD Computer-aided design.

CLI Command line interface.

CMA Covariance Matrix Adaptation.

CPD Critical path delay.

DQN Deep Q-Network.

DSP Digital signal processing.

DTS Discounted Thompson Sampling.

ES Evolution strategy.

FPGA Field-programmable gate array.

IC Integrated Circuit.

IO Input/Output.

LAB Logic array block.

MAB Multi-armed bandit.

PLL phase-locked loop.

QoR Quality of result.

RL Reinforcement Learning.

SA Simulated Annealing.

TBPSA Test-Based Population Size Adaptation.

UCB Upper Bound Confidence.

VPR Versatile Place-and-Route.

VTR Verilog-to-Routing.

WL Wirelength.

Chapter 1

Introduction

1.1 Motivation

The Field-programmable gate array (FPGA) is a type of Integrated Circuit (IC) which can be configured after manufacturing. Although FPGA's performance is not as efficient and cost-effective as application-specific integrated circuits (ASIC), FPGA offers flexibility, allowing users to tailor their specific application after the manufacturing of the chip. The flexibility and high performance make FPGA a popular choice in many industries. For example, Microsoft has deployed FPGA hardware in Bing and other data centers since 2014 [1, 2].

The growth of FPGA technology follows the exponential forecast of Moore's law [3], turning the FPGA into an intricate device with great capacity and architectural complexity. Nowadays, the most powerful FPGA, AMD's Versal VP1902 Premium Adaptive SoC, has 18.5 million logical cells [4], increased from 8.5 million logical cells available per device in 2019. The increasing capacity and architectural complexity of modern FPGAs have led to a growing demand for efficient FPGA computer-aided design (CAD) tools. Modern FPGA CAD flow can be divided into multiple steps [5], as shown in Fig. 1.1. The introduction of each step is listed as follows:

- RTL Synthesis: RTL (Register Transfer Level synthesis) synthesis is the process of translating a high-level hardware description of a digital design, typically written in a hardware description language (HDL) such as VHDL or Verilog,

into a gate-level representation.

- Logic Synthesis: The logic synthesis stage focuses on optimizing clusters of logic gates and registers in an effort to reduce the cost of the circuit.
- Technology Mapping: Technology mapping transforms the circuit from a network of generic logic elements gates into a network of the logic blocks available in the target FPGA.
- Packing: Packing, or clustering, involves organizing technology-mapped circuit elements into logic blocks comprised of LUTs and flip-flops.
- Placement: The result of technology mapping and packing is a network of logic blocks, which is located on the target two-dimensional FPGA device by the placement step.
- Routing: Routing is to form the desired electrical connections between the logic blocks in the post-placement design.
- Physical Synthesis: Physical synthesis provides feedback containing physical information after placement (or routing) to earlier phases of the flow, to allow better optimizations to be applied based on more accurate information.

Among the various stages in the CAD flow, placement is the most time-consuming, while its outcome significantly impacts the runtime and quality of result (QoR) of the subsequent routing stage [6]. Moreover, if a placement result requires an excessive amount of wiring, the routing step will fail or be highly limited by the available routing resources of the FPGA. Therefore, optimizing the placement stage is vital for enhancing the overall efficiency of traditional FPGA CAD tools.

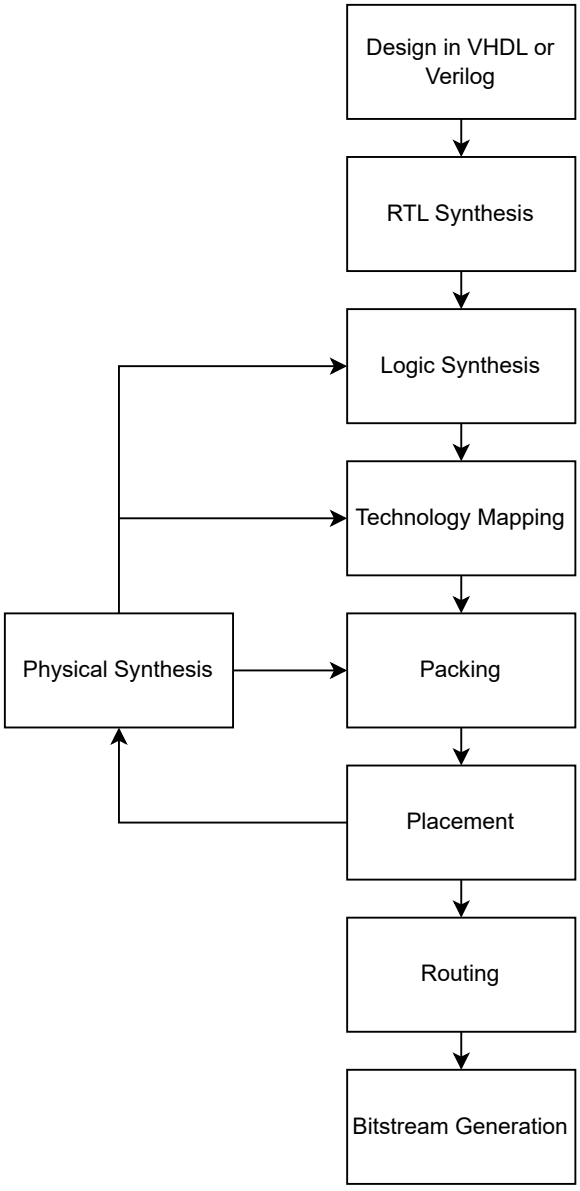


Figure 1.1: Modern FPGA CAD flow.

1.2 FPGA Placement

The placement stage aims to determine which logical cell should implement the logical components required by the input circuit design, in order to minimize the required wiring and critical path delay (CPD). Because the FPGA placement is an NP-hard problem, most algorithms are heuristic to find a sufficient result within a reasonable runtime [7]. Traditionally, there are three categories of placement algorithms [7]:

Simulated Annealing, Partition-based algorithms, and Analytical placement methods.

Simulated annealing mimics the annealing process used to gradually cool a sample of molten metal in order to produce high-quality metalwork. Starting from an initial placement configuration, simulated annealing introduces moves to the configuration by swapping the positions of two blocks or moving a block to an unoccupied position. A cost function is calculated afterward to determine whether to accept or reject the moves. A hill-climbing mechanism allows the simulated annealing algorithm to accept less desirable moves, in order to escape local optima. SA can guarantee high-quality results, while it suffers from the long runtime. Versatile Place-and-Route (VPR) [8], an open-source FPGA CAD tool, uses simulated annealing to perform FPGA placement. Recently, Some researchers have reformulated SA into a Reinforcement Learning (RL) problem [9–11] (or specifically the multi-armed bandit (MAB) problem), allowing the simulated annealing algorithms to generate "smart" moves by choosing directed moves [12] or block types rather than randomly swapping blocks, which can significantly improve the runtime without sacrificing QoR.

Partition-based algorithms use a divide-and-conquer paradigm by recursively partitioning a circuit design into two sub-circuits, which induces ever-smaller placement problems. [13–15] are all fast placement methods for FPGAs, while using different methodologies to partition the circuit and minimize the sub-circuits delay. Because partition-based algorithms solve the problem locally, they produce solutions with short runtime but suffer from low QoR.

Analytical placement methods define the placement goals as a mathematical optimization problem based on design objectives and constraints then solve it to find an optimal or near-optimal placement arrangement. Commonly, the analytical goal is to optimize wirelength and increase routability, as done by Gplace [16] and elfPlace[17]. FPGA CAD tools, like [18], incorporate timing performance goals as well.

1.3 Contributions

We investigate the RL algorithms of the VPR. We provide a theoretical analysis and comparison between multiple RL agent algorithms. From the theoretical perspective, we find that the Upper Bound Confidence (UCB) family algorithm and Boltzmann–Gumbel Exploration (BGE) are better than the VPR's current RL algorithm for simulated annealing placement in FPGAs.

Meanwhile, we discovered an issue that can't be solved by the current VPR algorithm. The current algorithm doesn't consider the FPGA placement problem's *non-stationary*. In the context of RL, *non-stationary* refers to a situation where the optimal actions or policies change over time, which means that the algorithm needs to continually adapt its policy to maximize its performance in the evolving environment. However, the current VPR algorithm that assumes stationary environments may struggle to adapt to changing conditions. Based on this fact, we integrate the multiple MAB algorithms in the VPR and conduct experiments to show whether changing they can improve the FPGA Reinforcement Placement.

We notice that a wide variety of other AI techniques remain unexplored due to the difficulty of directly integrating them into the FPGA CAD flow based on C++, creating a significant obstacle for AI researchers who seek to contribute to the field without specialized knowledge in FPGA CAD. To reduce the entry barrier, we propose and develop VPR-Gym, a Python environment built on top of the OpenAI Gym framework for simulated annealing FPGA placement. Our platform enables seamless integration with several Python-based machine learning libraries, including PyTorch [19], TensorFlow [20], and Nevergrad [21], which allows researchers to focus on high-level algorithm design and reduces the engineering effort required for porting ML libraries from Python to C++.

Besides proposing Vpr-Gym, we introduce a learning objective that formulates the FPGA placement task into an optimization problem, expanding the range of AI

techniques that can be explored beyond algorithms for the MAB problem.

To demonstrate the capability of our platform and learning objective, we conducted experiments on VPR-Gym, comparing the performance of various MAB algorithms as well as different algorithms that belong to evolution strategy (ES). Note that ES is one of the AI techniques that has not been previously utilized for FPGA placement. Our experiments reveal that the ES approaches exhibit superior performance compared to the existing MAB approaches, demonstrating the potential of VPR-Gym to facilitate AI research for FPGA simulated annealing placement and enhance its performance.

1.4 Thesis Outline

The content of this thesis is organized in the following manner. Chapter 2 is a survey of related background. Chapter 3 shows the investigation of VPR's RL algorithms. Chapter 4 demonstrates our VPR-Gym and its design details. Chapter 5 details our experiment setting. Chapter 6 presents the experimental results of different MAB and AI techniques on VPR-Gym. Chapter 7 contains the conclusion and potential future works.

Chapter 2

Background

2.1 FPGA Simulated Annealing Placement

VPR, which is part of the FPGA CAD design flow Verilog-to-routing (VTR) 8 [22], applies Simulated Annealing (SA) to achieve high QoR in placement. SA randomly performs perturbations that move and swap the blocks on an FPGA board to optimize the placement. *RLPlace*, as illustrated in Fig. 2.1, is the SA-based FPGA Reinforcement Learning (RL) placement technique on VPR proposed by Elgammal et al.[9]. Directed moves, e.g., Bounded Median Move, Critical Random Move, etc. [12], are more likely to produce useful perturbations compared with random moves. Six different types of directed moves are added to the standard VPR 8 SA FPGA placement algorithm and can benefit QoR in different ways, e.g., in terms of wire length, critical path delay (CPD), or both. Table 2.1 shows the detailed information of each directed move.

Directed moves' effectiveness depends on the operating netlist, current placement progress, targeted architecture, etc. Since manually scheduling directed moves that can balance all these factors is challenging, a multi-armed bandit (MAB) agent is designed to learn and select the most effective directed move type for optimizing the SA placement process [9, 10]. Specifically, for every single step:

- The agent chooses a directed move type as an action;

Table 2.1: Information about different FPGA directed moves.

Directed Move Type	**Average Runtime**	**Prefer Optimization Target**
Random Move	Low	None
Bounded Median Move	High	Wirelength
Bounded Edge-Weighted Median Move	Median	Wirelength and CPD
Bounded Centroid Move	High	Wirelength
Bounded Weighted Centroid Move	Median	Wirelength and CPD
Critical Random Move	Low	CPD (Timing Performance)
Quasi-Bounded Feasible Region Move	Median	CPD (Timing Performance)

- The move generator decides the origin and destination of the next move;
- SA process swaps the logical blocks at the origin and destination and returns the change in cost and whether to accept the swap;
- The observer sends a reward to the agent based on SA's outcome.

Murray et al. [11] propose another way to enhance the VPR by defining logical block types as the action space and performing RL on it, which also outperforms the standard VPR 8 FPGA placement algorithm. Similarly, the agent is designed to solve an MAB problem, except that the available actions are logical block types instead of directed move types.

All these approaches [9–11] demonstrate strong scalability characteristics: As the FPGA size increases, neither the number of directed move types nor logical block types will experience a corresponding increase, which means the RL agent is facing the same problem size.

However, there are several challenges to investigating AI techniques for FPGA

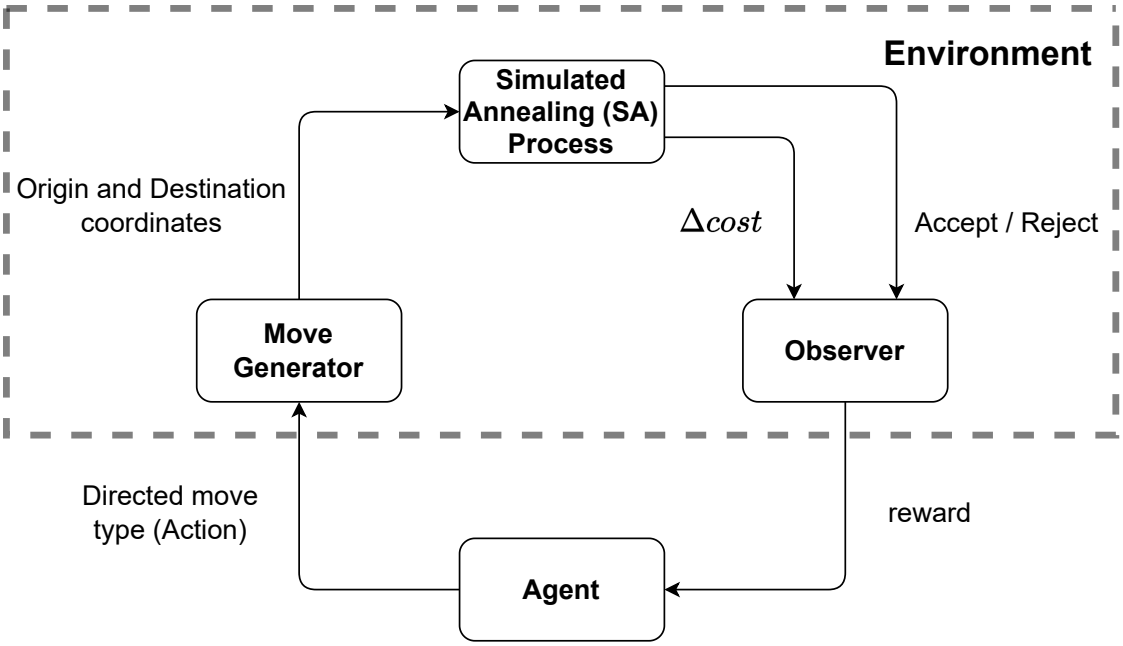


Figure 2.1: The *RLPlace* mechanism

placement:

- **Applying latest AI methods**: since VPR is written in C++, it is hard to apply popular state-of-the-art AI techniques to FPGA placement, as most of these techniques are developed in Python language.

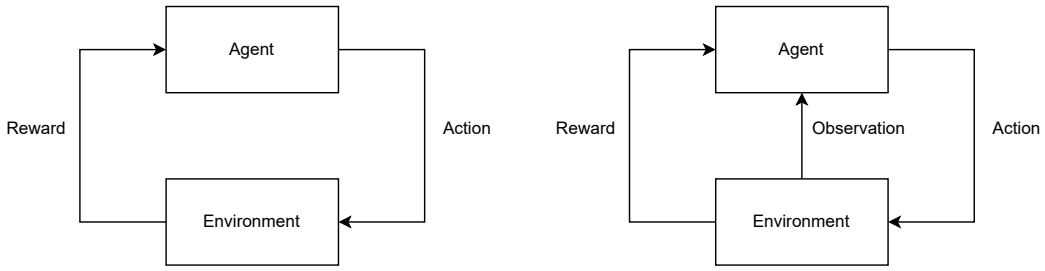
- **Algorithm-Environment Coupling**: Existing FPGA placement algorithms [9, 23] are all highly coupled with the rest parts of VPR, serving as one phase of VPR's FPGA CAD workflow. This means employing a new algorithm requires a deep understanding of VPR, and the new agent must be written inside the source code of VPR. Besides, every improvement to the agent learning code requires a recompilation of the entire VPR project, which is time-consuming.

- **Algorithm Limitation**: Existing FPGA placement algorithms [9, 23] only support a reinforcement learning formulation, specifically the MAB problem, maximizing the long-term rewards during the simulated annealing process, which are assumed to be stationary. However, *RLPlace* has investigated its agent behavior on different FPGA designs and observed that the optimal directed move

type is changing over time [9].

2.2 The Multi-armed Bandit Problem

The MAB problem is a classic problem in RL, in which the agent iteratively selects from multiple competitive actions in a way that can maximize the expected outcome. Note that the properties of those choices are unknown or partially known at the selecting moment, and may become better understood by choosing actions and receiving rewards. In fact, MAB is a simplified special case of RL, as shown in Fig. 2.2. Because there is no observation in MAB, the knowledge of each action is based on the historical rewards receives from the environment while taking this action.



(a) MAB (b) RL

Figure 2.2: Formulation of MAB problem and RL problem

In the MAB problem, the agent faces an exploration-exploitation trade-off dilemma:

- Exploration: Explore every action and improve the estimation of the action's expected reward.
- Exploitation: Exploit the optimal action in order to maximize the overall reward.

2.3 Evolution Strategy

ES is a heuristic algorithm inspired by biological evolution. A typical evolution strategy algorithm is demonstrated in Algorithm 1. It maintains a population of candidate

solutions and creates new candidate solutions by recombining and mutating the existing candidates. By eliminating the worse candidates in the population, all candidates generally converge to the optimal solution. ES has emerged as a highly effective solution for continuous optimization problems and its popularity has been further bolstered by the recent surge of research [24].

Algorithm 1 Evolution Strategy

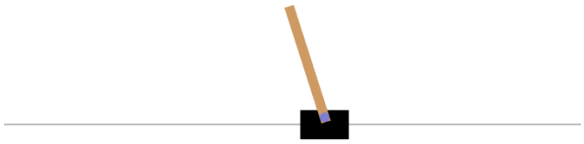
Require: Initial population size N, number of generations G
Ensure: Solution $\mathbf{x}^*$
1: Initialize population $\mathbf{P}$ with random solutions
2: **for** $g = 1$ to G **do**
3: Evaluate fitness for each solution $\mathbf{x}$ in $\mathbf{P}$
4: Select parents $\mathbf{P}_{\text{parents}}$
5: Generate offspring $\mathbf{o}$ by recombining $\mathbf{p}_1$ and $\mathbf{p}_2$
6: Mutate offspring $\mathbf{o}$
7: Update $\mathbf{P}$ by choosing the best solutions among $\mathbf{o}$ and $\mathbf{P}$
8: **end for**
9: **return** Best solution $\mathbf{x}^*$ found in $\mathbf{P}$

2.4 OpenAI Gym

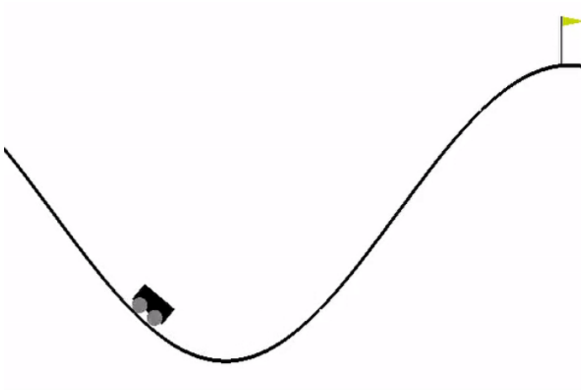
OpenAI Gym [25] is an open-source toolkit designed to facilitate research and development in reinforcement learning. It provides a collection of diverse environments that enable users to create and test reinforcement learning agents. Also, it provides a standardized API for users to create their own environment. The standard API is easy to work with popular ML libraries, such as Tensorflow [20] and Pytorch [19].

The original purpose of OpenAI Gym was to provide a standardized and accessible way for researchers and practitioners to experiment with reinforcement learning algorithms. Hence, those early environments are mainly derived from classic control problems [25] and simple arcade games [26], as shown in Fig. 2.3 and Fig. 2.4. They have become popular benchmarks to compare the performance of different RL agents and investigate the behaviours of those algorithms. However, there is an opinion [27] that believes that these toy environments hinder potential principled studies in RL

because some state-of-the-art methods are overfitted to those toy environments but not to real-world problems.



(a) Cart Pole



(b) Mountain car

Figure 2.3: Classic control problems: (a) is to keep the pole vertical and (b) is to move the car from the bottom to the right hill.

Fortunately, a variety of OpenAI Gyms have been built recently for different real-world application areas, such as compiler design [28], auto traffic control [29], Robotics [30], and networking [31], which provide more challenging but fruitful puzzles for RL development. Compared to toy environments, those real-world gyms no longer represent ideal problems. Instead, the problems have become more complex. The complexity lies in two perspectives:

- The difficulty of the problem increases. For example, real-world problems usually come with noise and disruption.
- The interaction between agent and environment requires a lot more effort.

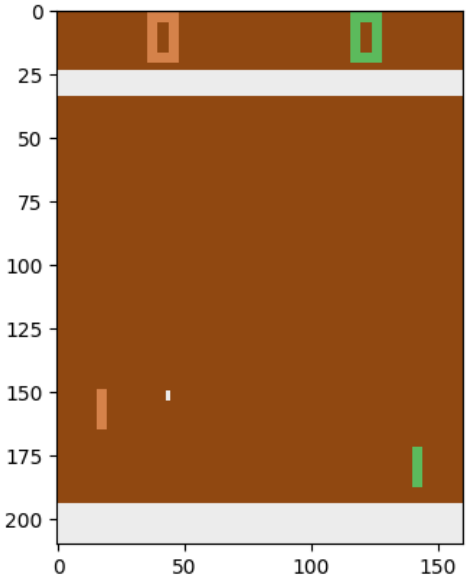


Figure 2.4: The Pingpong game in the OpenAI Atari Gym environment

The effort spent on interaction is greatly reduced by the availability of OpenAI Gym in different research areas. As an illustration, Piotr et. [31] have built Ns3-gym in order to apply ML techniques such as RL to the Ns-3 network simulator. They built a connection between Python and Ns-3 network simulator via sockets and abstract the complex details about the Ns-3 network simulator, as shown in Fig. 2.5. All details are hidden from the agent except the Gym API. The convenience brought by Ns3-gym catalyze the research in networking area such as in [32].

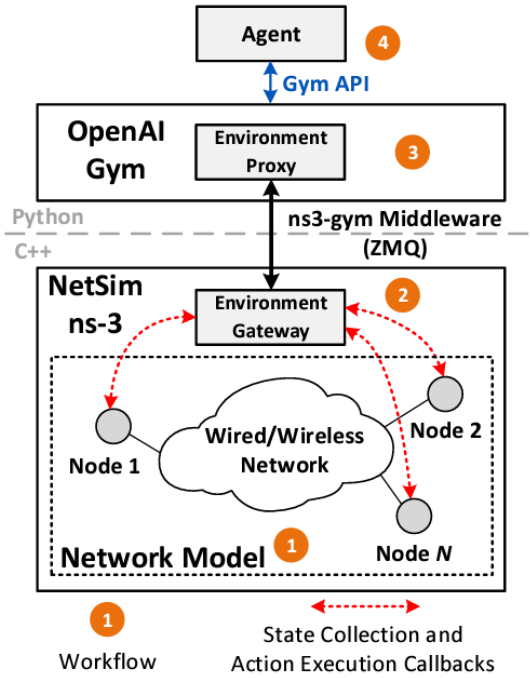


Figure 2.5: Architecture of the ns3-gym framework

Chapter 3

The MAB Algorithm in FPGA Placement

In this section, we first introduce the current MAB algorithm applied to VPR. Then we analyze the weakness of the current algorithms and provide MAB algorithms that can mitigate the shortcomings.

Before further discussion, we want to provide a clearer definition of the MAB problem: Let $\mu_1, \ldots, \mu_N$ be the mean values associated with the reward distributions for actions $a = 1, \ldots, N$. The agent is required to iteratively choose one action a_t at each round t and observe the associated reward r_t. The objective is to maximize the sum of the collected rewards $R_{sum} = \sum_{t=1}^{T} r_t$, i.e., the overall reward in the long run.

3.1 Introduction to Current VPR MAB Algorithms

Currently, VPR's MAB algorithms are Epsilon-greedy and Boltzmann Exploration (also called Softmax).

3.1.1 Epsilon-greedy

Algorithm 2 shows the procedure of Epsilon-greedy: Given a fraction ϵ, the agent randomly selects one from all available actions by this fraction; in other cases, the agent always selects the greedy action with the highest expected reward.

Algorithm 2 Epsilon-Greedy

Require: Number of actions N, exploration rate ϵ
Ensure: Selected action a

Initialize sum of reward $S(a)$ for all $a \in \{1, 2, \ldots, N\}$ to zeros
Initialize counter $N(a)$ for all a to zeros
while not end **do**
    if rand_uniform$(0, 1) < \epsilon$ **then**
        Select a random action a uniformly from $\{1, 2, \ldots, N\}$
    else
        Select action a with the highest estimated value: $a = \arg\max_a \frac{S(a)}{N(a)}$
    end if
    Take action a, observe reward r
    Update action-value estimate: $N(a) \leftarrow N(a) + r$
    Update action counter: $N(a) \leftarrow N(a) + 1$
end while

3.1.2 Boltzmann Exploration (BE)

The agent assigns a different probability of being chosen to each action based on the action value $Q(a)$, as in Algorithm 3. The action value $Q(a)$ represents the expected reward from selecting action a. The exploitation goal is achieved by choosing the best action with the highest probability. On the other hand, the other non-greedy actions can also be explored, hence the requirement of exploration is also met. The action value $Q(a)$ can be defined in various forms, while the average of accumulated rewards is a common choice [33, 34].

Algorithm 3 Boltzmann Exploration

Require: Number of actions N, temperature parameter τ
Ensure: Selected action a

Initialize action-value estimates $Q(a)$ for all $a \in \{1, 2, \ldots, N\}$ to zeros
while not end **do**
    Compute the softmax distribution over action-values:
    $P(a) = \frac{\exp(Q(a)/\tau)}{\sum_{b=1}^{N} \exp(Q(b)/\tau)}$
    Select action a based on the probability distribution $P(a)$
    Take action a, observe reward r
    Update action-value estimate: $Q(a)$
end while

3.2 Algorithm Analysis

3.2.1 Regret Analysis

Definition

One popular measure of a strategy addressing this exploration/exploitation dilemma is regret ρ. Recall that μ_1, ..., μ_n are the mean values of reward distributions for different actions. Note that μ_1, ..., μ_n are fixed values and remain unchanged in the regular MAB setting. The regret after T rounds is defined as:

$$\rho = T\mu^* - \sum_{t=1}^{T} r_t \tag{3.1}$$

where $\mu^* = \max(\mu_1, ..., \mu_n)$ is the mean value of the optimal action's reward distribution and r_t is the reward at round t. Therefore, the regret ρ is defined as the expected difference between the reward sum associated with the optimal action and the sum of the collected reward. Regret is a fundamental concept that can be used to compare different algorithms for MAB problems. Algorithms that achieve lower regret are considered to have better performance theoretically.

Lai and Robbins [35] prove that the regret R_T of MAB grows at least logarithmically, denoted by $\rho_T = \Omega(\log T)$. An algorithm is said to *solve* the MAB if its regret matches this lower bound [33].

Epsilon-greedy and BE Regret Bound

According to Kuleshov and Precup's summary [36] of the epsilon-greedy algorithm, its regret depends on whether the ϵ is a fixed value or not: if ϵ is fixed, its expected regret is linear $O(T)$; if ϵ is decreasing over time, Cesa-Bianchi and Fisher [37] prove it has polylogarithmic regret $O(\log T^2)$. As for the BE algorithm, Cesa-Bianchi and Fischer [37] prove that the BE algorithm with decreasing temperature τ has a polylogarithmic regret $O(\log T^2)$. Neither Epsilon-greedy nor BE is able to *solve* the MAB problem.

Upper Confidence Bound (UCB) algorithm

The UCB family of algorithms was proposed by Auer et al. [38]. It chooses actions based on the combination of historical observed reward and uncertainty, as shown in Algorithm 4.

Algorithm 4 UCB Algorithm

Require: Number of arms N, confidence parameter α
Ensure: Selected arm a
1: Initialize sum of reward $S(a)$ for all $a \in \{1, 2, \ldots, N\}$ to zeros
2: Initialize counter $N(a)$ for all a to zeros
3: Initialize time step $t = 1$
4: **while** not ended **do**
5: Select arm a with the highest UCB value:
6: $a = \arg\max_a \left(\frac{S(a)}{N(a)} + \alpha \sqrt{\frac{\log(t)}{N(a)}} \right)$
7: Take action a, observe reward r and new state
8: Update S(a), N(a)
9: Increment time step: $t \leftarrow t + 1$
10: **end while**

The UCB algorithm [38] was proposed based on the idea of optimism in the face of uncertainty from Lai and Robbins [35]. The expected regret of the UCB algorithm is bounded by $O(\log(t))$. To verify whether UCB can improve the performance of FPGA placement, we describe experiments in the later chapters.

Boltzmann–Gumbel Exploration (BGE)

BGE is a variant of Boltzmann Exploration proposed by [34] together with theoretical proof that BGE has near-optimal guarantees, in contrast to BE which has a monotone learning rate which is sub-optimal. BGE leverages the Gumbel-max trick [39] by adding a noise term drawn from a Gumbel distribution to increase the robustness of the exploration, as shown in Algorithm 5. Experiments are conducted in the later chapters to compare the performance among BGE and other MAB algorithms.

Algorithm 5 Boltzmann-Gumbel Exploration

Require: Number of arms N, temperature parameter τ
Ensure: Selected arm a

Initialize action-value estimates $Q(a)$ for all $a \in \{1, 2, \ldots, N\}$ to zeros
Initialize counter $N(a)$ for all a to zeros
Initialize time step $t = 1$
while not ended **do**
    Sample values from the Gumbel distribution for each arm a as $Z_{t,a}$
    Compute the Boltzmann-Gumbel probabilities:
        $P(a) = \frac{\exp((Q(a))/\tau)+\sqrt{\frac{C^2}{N(a)}}Z_{t,a}}{\sum_{b=1}^{N}\exp((Q(b))/\tau)+\sqrt{\frac{C^2}{N(b)}}Z_{t,b}}$
    Select arm a based on the probabilities $P(a)$
    Take action a, observe reward r
    Update $Q(a)$, $N(a)$
    Increment time step: $t \leftarrow t + 1$
end while

3.2.2 Non-stationary Analysis

A limitation of the above MAB algorithms is that they are designed to maximize the overall rewards, while concept drift might be present in the SA process during FPGA placement, i.e., the expected reward for a directed move type can change at every time step. Elgammal et al. [9] investigate the VPR's BE agent behavior on different designs. Fig. 3.1 shows the number of moves at different temperature indexes on the stereovision1 design from the VTR benchmark. Note that the B.W.Med., C.R., and QB.F.R. are designed to be activated near the end of the simulated annealing process. Each temperature index represents one epoch in the SA process and a higher temperature index means the epoch is closer to the end. At the early annealing, the agent almost equally chooses the move type. By the definition of the BE algorithm, the μ of each move type is almost equal. After the middle of the annealing process, diversity appears, which means the mean value μ of each directed move type is changed.

In this case, the problem is called the non-stationary bandit problem. Hartland et al. [40] propose that standard MAB algorithms are not appropriate for abruptly changing environments. Therefore, we apply Discounted Thompson Sampling (DTS)

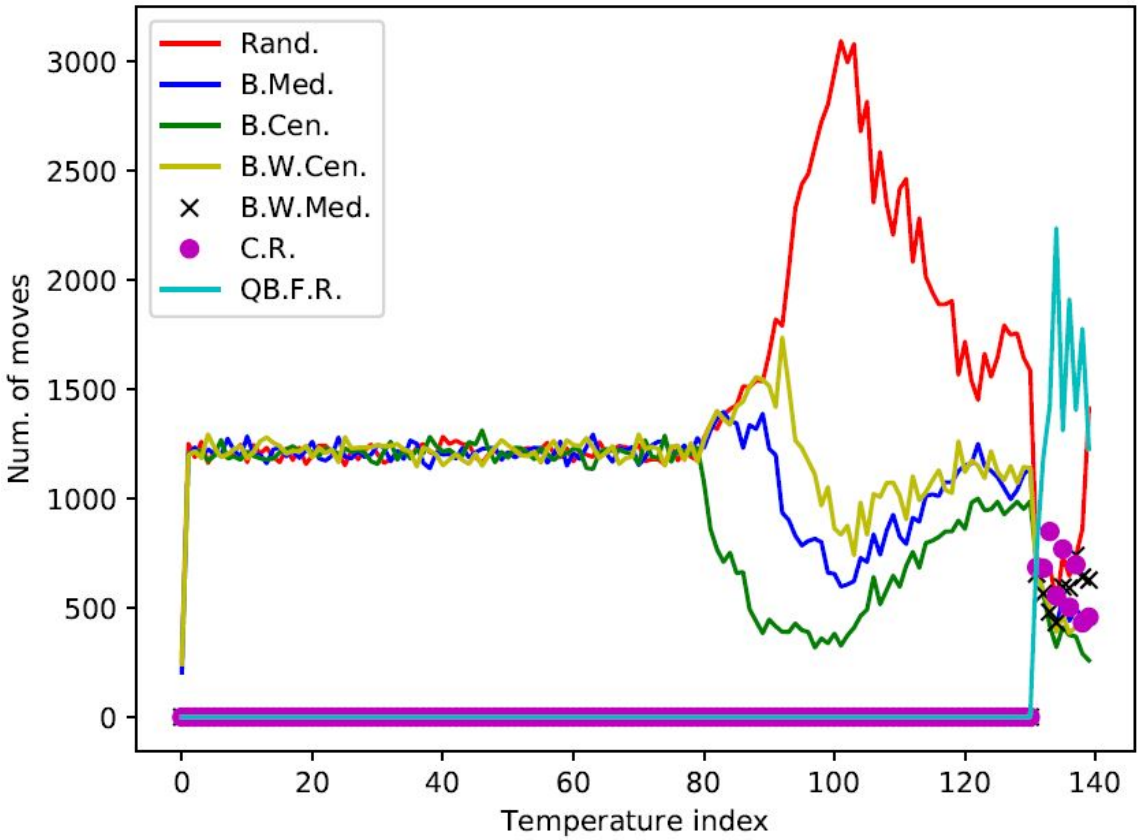


Figure 3.1: The number of directed move types for stereovision1 at different temperatures.

[41], a Thompson Sampling [42] variant designed for the non-stationary MAB problem, to the FPGA placement problem. Unlike other MAB algorithms, DTS systematically increases the variance of the unexplored actions' prior distribution in order to increase the probability of picking them. DTS's implementation is specified in Algorithm 6.

Algorithm 6 Discounted Thompson Sampling

Require: Number of arms N, discount factor γ
Ensure: Selected arm a

Initialize Bayesian distributions β for each action $a \in \{1, 2, \ldots, N\}$
while not ended **do**
    for $a = 1$ to N **do**
        Sample a value θ_a from the distribution for action a
    end for
    Select action a with the highest sampled value:
    $a = \arg\max_a (\theta_a)$
    Take action a, observe reward r
    Update β based on reward r and discount factor γ:
end while

Chapter 4

ES algorithm in FPGA placement

In this section, we first propose a new problem formulation for the FPGA SA-based placement problem in order to broaden the range of AI techniques that can be used on this specific problem, including but not limited to ES algorithms. Under the new formulation, the problem turns from an MAB problem into a continuous optimization problem. Next, we apply ES algorithms to solve the continuous optimization problem with a thorough discussion about their features and the reasons for adopting them.

4.1 Problem Formulation

The current learning objective in FPGA placement is to learn the best directed move at the current step without observations from the environment, which can only be solved by the MAB algorithms. In order to accommodate a wider range of AI optimization techniques, we propose a more general learning objective: let the agent learn the optimal or near-optimal ratio among all available actions. Our learning objective converts the current RL (MAB) formulation of FPGA placement into an optimization problem, as shown in Fig. 4.1, which allows us to solve the problem using ES, as well as a potentially broader range of other AI optimization algorithms. The MAB problem is a special case under this formulation when $k = 1$. For example, if the agent is the UCB algorithm, it will update its knowledge every time it receives a reward and propose an action by setting one action's probability to 1 and all others

to 0.

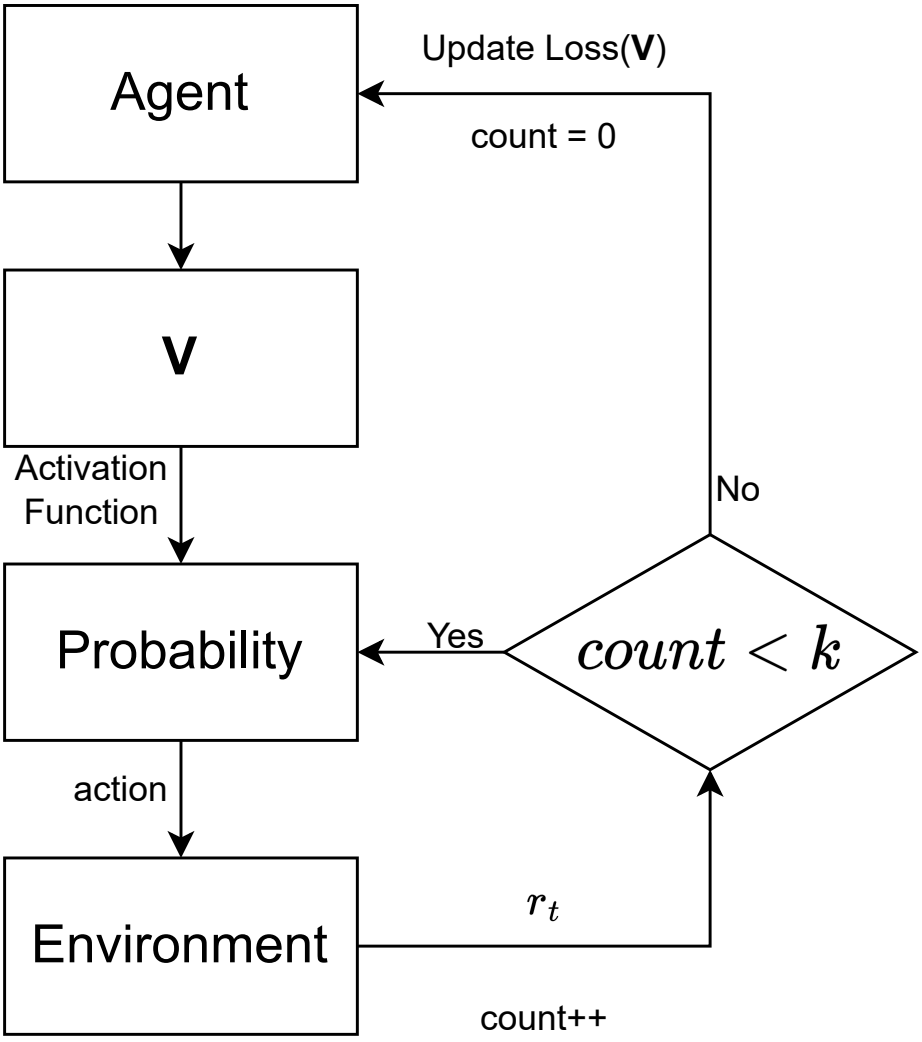


Figure 4.1: Problem Formulation

We define a control vector $\mathbf{V}$ as the optimization target to control the probability of choosing each action. The length of $\mathbf{V}$ is determined by the number of available actions. For example, if the agent needs to choose among 7 directed move types, the length of $\mathbf{V}$ is 7. Then, the probability P_a of each action a is given by

$$P_a = \frac{Sig(V_a)}{\sum_{i=1}^{N} Sig(V_i)}, \tag{4.1}$$

where $Sig(x) = 1/(1 - e^{-x})$ is the Sigmoid activation function over a domain of real values and N is the number of available actions. We apply the Sigmoid activation

function because it can map the real number domain to a range between 0 and 1, by which we don't need to set up constraints to use continuous optimization algorithms such as ES.

The optimization problem is to find an input vector $\mathbf{V}$ that minimizes a loss function $Loss(\mathbf{V})$, which is defined as

$$Loss(\mathbf{V}) = -\sum_{i=(t-k+1)}^{t} r_i. \tag{4.2}$$

Minimizing $Loss(\mathbf{V})$ will maximize the sum of reward during the past τ steps. As the agent maximizes the reward dynamically in moving windows, the overall placement performance is maximized. We will show that this formulation can accommodate multiple algorithms by changing the specific form of action sampling probability P_a and minimizing different losses as functions of step-wise rewards. In fact, by adjusting k, we can dynamically generate a non-stationary policy to sample different actions as the SA process progresses.

4.2 Apply ES Algorithms

ES, as discussed in Section 2.3, is a category of algorithms that mimic biological evolution. We introduce ES to solve FPGA placement with the problem shown in Fig. 4.1. ES agent proposes a candidate vector $\mathbf{V}$ from its population and transfers it into a probability distribution via a Sigmoid activation function. Action will be sampled from the probability distribution until k steps have been taken. After k steps have been taken, the ES agent receives $Loss(\mathbf{V})$ as the fitness of the candidate. Then the agent performs an update and proposes a candidate $\mathbf{V}$ again. By generating new candidates and eliminating worse candidates among them, the ES agent optimizes the probability P_a periodically and dynamically as moving windows, i.e., every k steps, the ES agent will propose $\mathbf{V}$ to minimize the loss term in (4.2).

Specifically, we introduce two ES algorithms into the VPR placement problem, including Test-Based Population Size Adaptation (TBPSA) [43] and Covariance Matrix

Adaptation (CMA) [44]. CMA's design follows the concept of ES which is shown in Algorithm 1, except it uses a covariance matrix to guide the mutation of its candidate, which can lead to faster convergence to optima. The covariance matrix is a matrix that summarizes the relationship between different variables. In ES cases, variables are the dimensions of the optimization problem's input. Fig. 4.2 demonstrates the intuition of how CMA works: the dotted line represents the covariance matrix and it move generally towards the global optima. Besides the traditional features with a covariance matrix, TBPSA has a population control features, adjusting its population size with a statical test.

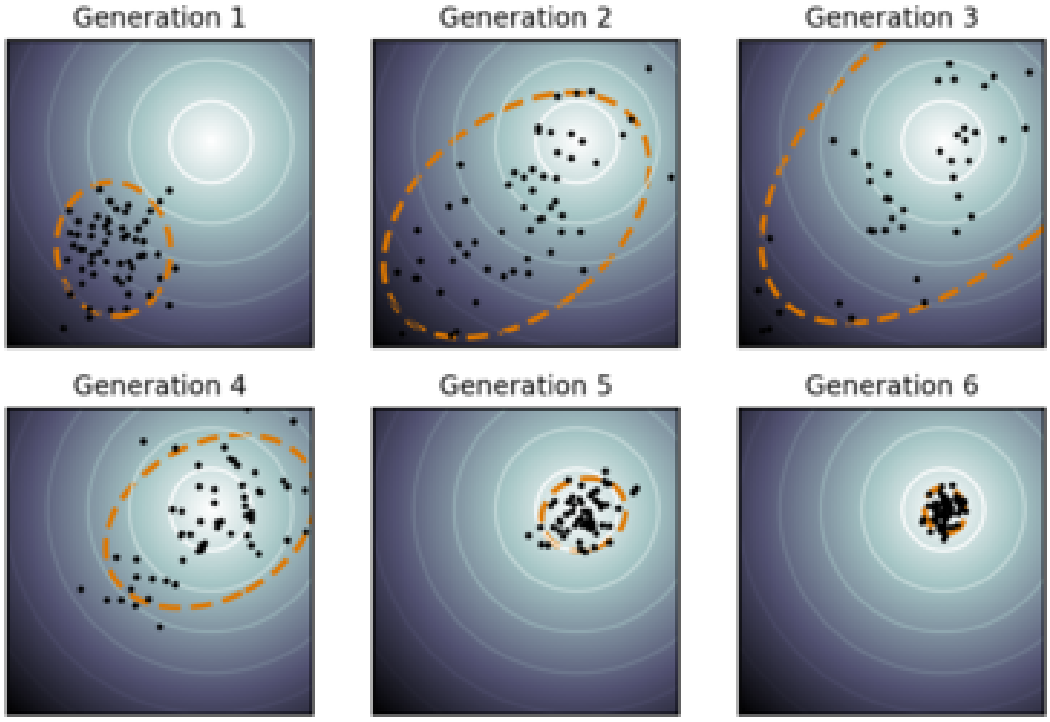


Figure 4.2: Illustration of the CMA algorithm [45].

TBPSA and CMA both have the ability to escape from local optima, increasing the robustness to the non-stationary in FPGA placement. TBPSA applies a population size adaptation technique [46] to increase the population size to explore the search space more thoroughly when it gets stuck in local optima or when there is a need to handle noise. Conversely, it can decrease the population size to accelerate

convergence. On the other hand, CMA uses a restart strategy [47] to escape from local optima, i.e., when the algorithm stagnates, it will restart from a random initial point. According to [43, 46], TBPSA outperforms CMA in noisy settings as well as multimodal settings which means the presence of many local optima.

Chapter 5

VPR-Gym

In this chapter, we introduce the design and implementation of VPR-Gym, a novel approach to interacting with VPR through the OpenAI Gym framework in Python. As illustrated in Fig. 5.1, VPR-Gym abstracts the SA-based placement in VPR away from the agent such that the researchers can employ different agents directly without having background knowledge about VPR C++ source code. In the following section, we will discuss different components of VPR-Gym.

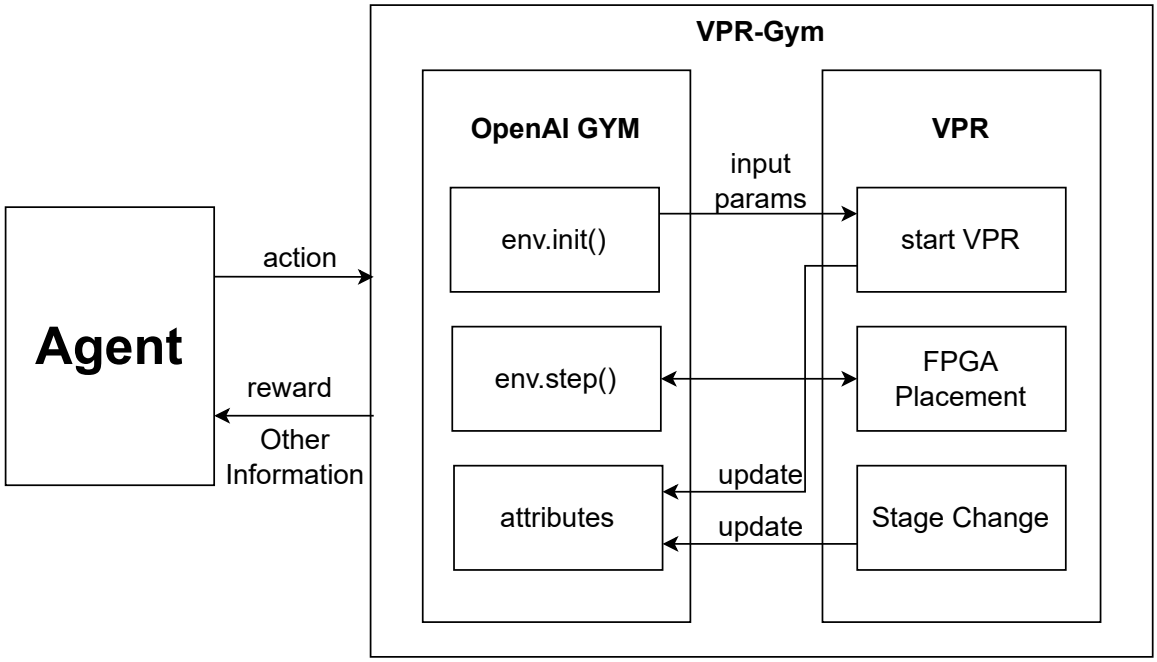


Figure 5.1: Architecture for VPR-Gym.

5.1 Overview

Fig. 5.2 shows an example of how to use VPR-Gym. Lines 2-3 show the initialization of VPR-Gym. We provide two options for the environment: **VprEnv** and **VprEnv_blk_type**, depending on whether *block type* will be a part of the action space, which will be introduced in Section 5.3. Lines 5-8 show how the agent learns using VPR-Gym. At each step, the agent takes an action and passes the action to VPR via function **env.step()**. The function will return a reward and other necessary information from the VPR environment to the agent. Updating the agent is not necessary after each **env.step()**: researchers are free to define the agent's behavior. For example, to follow the optimization learning objective, researchers can collect the rewards over any given number of steps and use that to update the agent action probability distribution. When the FPGA placement process terminates, the function **env.step()** will set *done* = *True* as a flag to indicate termination.

```
1 from src.vprGym import VprEnv, VprEnv_blk_type
2 env = VprEnv()
3 # env = VprEnv_blk_type()
4 agent = Agent()
5 while (done == False):
6     action = agent.action()
7     state, reward, done, info = env.step(action)
8     agent.getReward(reward)
```

Figure 5.2: Example of VPR-Gym usage.

5.2 Gym Initialization

The initialization of the gym environment starts a VPR kernel under the Gym mode. Under this mode, the VPR program will pause at certain points and communicate with the OpenAI Gym framework via the Zeromq socket.

Users can specify the arguments in the **VprEnv()** constructor to control the set-

ting of VPR. The OpenAI Gym framework uses those arguments to initialize the VPR kernel via the command line interface (CLI). Therefore, any viable simulation settings in the VPR kernel are still viable in the VPR-Gym. Note that our VPR-Gym supports parallel initialization and parallel execution. This, however, requires correctly assigning different working directories and port numbers for different tasks to avoid conflicts between multiple environments, which is shown in Fig. 5.3.

```
from src.vprGym import VprEnv, VprEnv_blk_type
env1 = VprEnv(directory='env1', port='5555',
    benchmark='...')
env2 = VprEnv_blk_type(directory='env2', port='6666'
    , benchmark='...')
agent1 = Agent()
agent2 = Agent()
train(env1, agent1)
train(env2, agent2)
```

Figure 5.3: Example of VPR-Gym parallel initialization.

5.3 Action Space

To build our basic environment **VprEnv** in VPR-Gym, we employ the same action space as that in *RLPlace* [9]: there are 7 directed move types, which are Random Move, Bounded Median Move, Bounded Edge-Weighted Median Move, Bounded Centroid Move, Bounded Weighted Centroid Move, Critical Random Move, and Quasi-Bounded Feasible Region Move. Therefore, in VPR-Gym, an action space is specified by a single discrete number from 1 to N, where the size of action space N is also subject to changes because there are two placement stages in VPR, for which the numbers of available directed move types are 4 and 7, respectively.

In VPR-Gym, we also introduce a new custom environment called **VprEnv_blk_type** where the logical blocks' types are added to the action space. The action space of **VprEnv_blk_type** is defined as a 2-tuple, where the first element represents

directed move types and the second element represents the block types. Note that for the two directed moves that only happen in the second stage (**Bounded Weighted Centroid Move** and **Critical Random Move**), selecting logical block types is not supported because these directed moves only manipulate blocks with highly critical connections. If there are some block types which do not contain highly critical blocks, this action pair will always lead to an aborted result and become a trivial action.

Because we define the action space in a similar way as demonstrated in 2.1, VPR-Gym inherits the scalability that the current VPR placement algorithm has. The action space size won't experience an increase as the targeted FPGA design's size increases.

5.4 Reward

VPR-Gym provides more freedom for researchers to design the agent's reward. Three variables are passed from VPR to the OpenAI Gym framework by storing them in the **info** dictionary, including the normalized change of the wire length cost ($\Delta cost$), the normalized change of the timing cost (Δt_{cost}), and the change of the bounding box (related to Wire Length) cost (Δbb_{cost}). All existing reward functions can be calculated from these three parameters. For example, the reward function of WLbiased_runtime_aware [9] is

$$r_t = \begin{cases} \frac{-1}{t(i)}((0.5 + R_s) * \Delta bb_{cost} \\ +(1 - R_s) * \Delta t_{cost}), & if\ \Delta cost < 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

where R_s is a constant empirically set to 0.4. Equation (5.1) can be computed locally at the Python end with these three variables. Moreover, in VPR-Gym, the value of R_s becomes an adjustable variable, which can be learned by the agent or tuned by a hyperparameter optimization mechanism.

5.5 Environment Attributes

Other important information is stored in the attributes of the environment, including:

- num_moves: An integer shows the number of logical block types in the targeted FPGA design;
- num_types: This is a list consisting of the number of blocks for each logical block type category;
- horizon: An integer that shows the number of steps within one temperature in the SA Process. After each temperature ends, some FPGA placement settings are changed, such as the maximum distance a block can be moved. We assume that this change brings an abrupt change to the reward distribution of each action. The horizon indicates how often an abrupt change happens. Note that some multi-armed bandit algorithms require the horizon information in order to better plan the exploration and exploitation;
- is_stage2: This is a Boolean variable showing a stage change of the SA process. As mentioned above, it would affect the number of available directed move types.

Chapter 6

Experimental Result

6.1 Experimental Setting

6.1.1 Environment

We compare the AI algorithms in two different environments setting, which include a primary environment **VprEnv** and a custom environment **VprEnv_blk_type**, as defined in Section 5.3. The number of available actions in each environment is shown in Table 6.1. The number of block types depends on the benchmark being used. The available blocks that were considered in VPR placement are Input/Output (IO), phase-locked loop (PLL), logic array block (LAB), digital signal processing (DSP), and memory blocks M9K and M144K.

Table 6.1: The number of available actions in VprEnv and VprEnv_blk_type.

Environment	**Stage 1**	**Stage 2**
VprEnv	4	7
VprEnv_blk_type	4 * (# of block types)	7 * (# of block types)

6.1.2 Benchmark

The experiments on **VprEnv** were conducted on eight circuits from the Titan23 benchmark suite [6], which are stereo_vision, bitonic_mesh, neuron, SLAM_spheric, dart, mes_noc, denoise, and cholesky_mc. The experiments on **VprEnv_blk_type**

were conducted on five circuits from the Titan23 benchmark suite [6], which are stereo_vision, bitonic_mesh, neuron, SLAM_spheric, and dart.

6.1.3 Metrics

We evaluate the runtime/QoR tradeoffs of different placement techniques by the post-placement estimated wirelength (WL) (based on VPR's fanout-adjusted WL metric) and critical path delay (CPD) versus runtime to align with [8, 9]. For each benchmark, we set 6 different numbers of moves attempted at each temperature, which will provide 6 runtime budgets. We use geometric means (Equation (6.1)) to combine the algorithms' performance on each benchmark versus 6 different runtime budgets. At each runtime budget, 3 separate experiments are conducted under different random seeds. Their average wirelength/CPD is used to perform the evaluation and comparison.

$$\text{Geometric Mean} = \sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n} \tag{6.1}$$

6.1.4 Algorithm Hyperparameter

We evaluate MAB algorithms including Softmax (Boltzmann Exploration), UCB, BGE and DTS, as discussed in Chapter 3. Note that we don't evaluate Epsilon-greedy because in [9] a thorough experiment demonstrated that Softmax is superior to Epsilon-greedy in SA-based FPGA placement. The MAB algorithms are from SMPyBandits [48], an open-source library for MAB problems.

- Softmax: Temperature $\tau = 0.01$
- UCB: Constant $\alpha = 1$
- BGE: Constant $C = 0.5$
- DTS: Discount factor $\gamma = 0.999$

Also, we evaluated two ES algorithms: TBPSA and CMA. Their hyperparameters are set as the default value in [21]. The ES algorithms are updated every $k = 100$ moves. The reward that is used for updating the MAB and ES algorithms is calculated based on *basic* reward function from [8] as suggested in [9].

6.2 Result in VprEnv

Fig. 6.1 presents the post-placement estimated WL and CPD of different algorithms on the Titan23 benchmark suite. We discuss the performance to showcase different algorithms' performance on **VprEnv**.

6.2.1 MAB Algorithms vs. ES Algorithms

Compared to the ES algorithms, the MAB agents require a longer runtime to complete all the runtime points in each SA temperature, as they update on every move, leading to the increased computational cost. In terms of the estimated WL, the ES algorithms exhibit superior performance across all runtime points. In terms of the estimated CPD, DTS shows the best overall performance. However, all six algorithms demonstrate CPD that doesn't monotonically decrease, which is consistent with the results of *RLPlace* method [9].

6.2.2 TBPSA vs. CMA

TBPSA demonstrates better performance compared with the CMA algorithm on both estimated WL and CPD, although its runtime is slightly higher under the same runtime setting. Recall that TBPSA outperforms CMA in noisy settings and multimodal settings as supported by experimental and theoretical results [43, 46]. Either the robustness to noise or the stronger ability to escape local optima helps TBPSA achieve superior performance compared with CMA.

6.2.3 BGE vs. Softmax

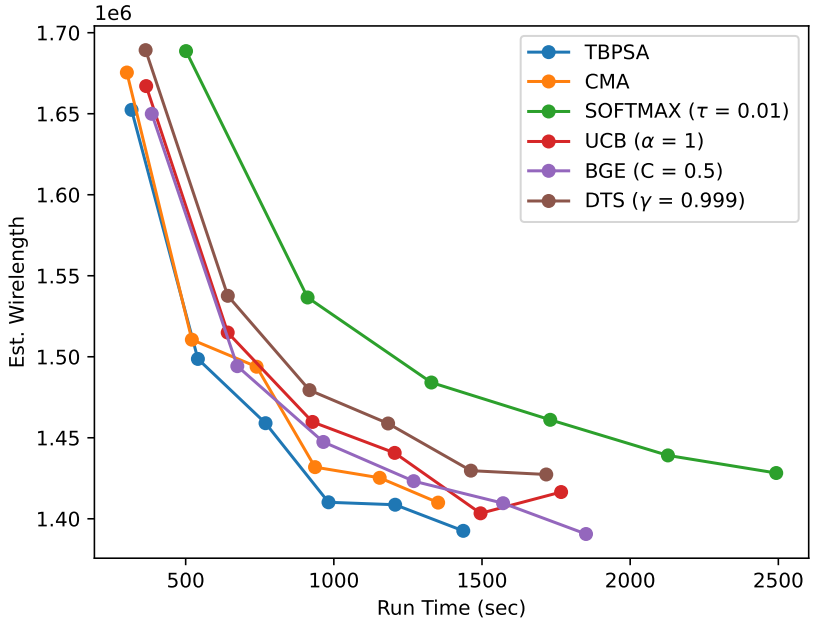
Cesa-Bianchi, Nicolò, et al. [34] suggest that the Boltzmann Exploration strategy with a monotone learning-rate sequence is sub-optimal and they propose a non-monotone schedule strategy, which is the BGE algorithm. Although according to the conclusion by Cesa-Bianchi [34] BGE algorithm should be better than the softmax algorithm, the BGE agent doesn't show a dominant result compared with softmax: BGE is better in wire length but worse in CPD compared with softmax. As the optimal directed move type is subject to change over time, we anticipated this result since it violates the assumption underlying the MAB problem.

6.2.4 DTS vs. Other MAB Algorithms

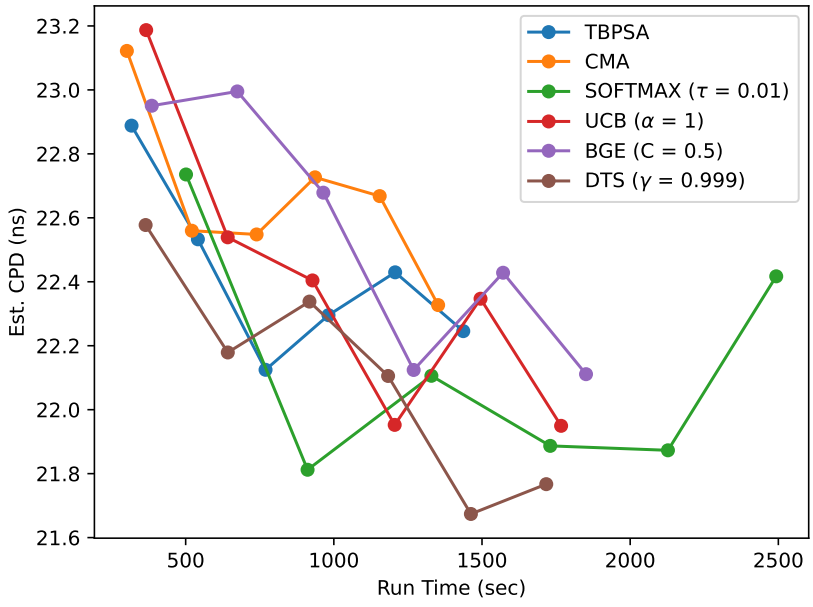
DTS is a variant of Thompson Sampling designed for non-stationary multi-armed bandit problems. Non-stationary multi-armed bandit means that the expected reward of each arm (action) is subject to change, which matches the SA-based FPGA RL placement problem's nature, as different directed moves would be preferred at different annealing temperatures. Hartland et al. [40] propose that standard MAB algorithms, such as UCB and Boltzmann Exploration, are not appropriate for abruptly changing environments. The results demonstrate that DTS outperforms other MAB algorithms, which matches Hartland et al.'s conclusion. The results indicate that it's fruitful to consider the non-stationary problem in designing the SA-based FPGA RL placement agents.

6.3 Result in VprEnv_blk_type

Fig. 6.2 shows the comparison of post-placement estimated wirelength and CPD between multiple algorithms on the custom environment. We see that the TBPSA outperforms all other algorithms in both wirelength and CPD. This result demonstrates that the TBPSA is able to learn better than the MAB agents in environments



(a)



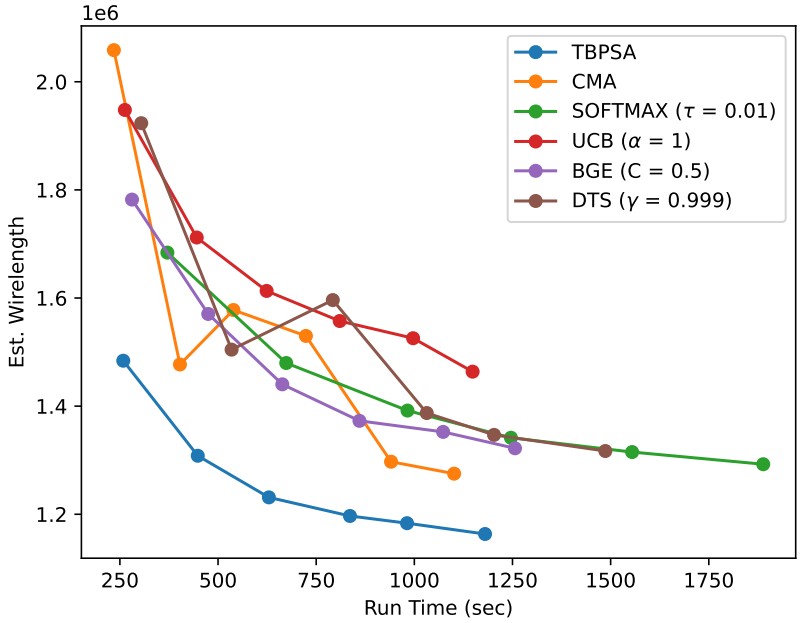
(b)

Figure 6.1: Comparison of different algorithms in VprEnv at 6 different runtime points (3-seed average). The lower the better. TBPSA and CMA are ES algorithms. SOFTMAX, UCB, BGE, and DTS are MAB algorithms

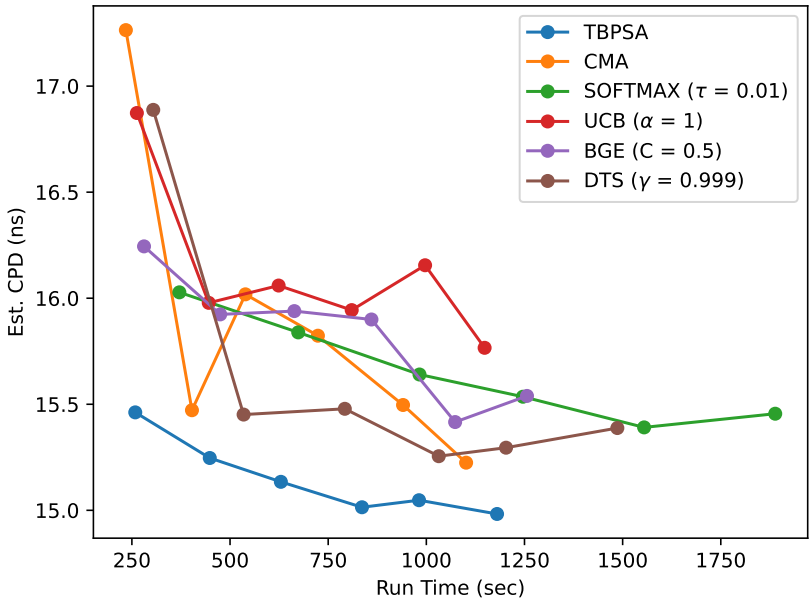
with larger action space. In the case of employing more directed move types and block types to the toolkit, using the TBPSA algorithm can better enhance the CAD tool's performance compared to other algorithms.

6.4 Discussion

Through the above experiments, the proposed ES has proven to be superior and more efficient than the traditional RL (MAB) approaches for FPGA placement, especially in environments with a more complex action space. Besides, VPR-Gym enables the evaluation of a diverse range of AI optimization algorithms without implementing them in the VPR source code in C++, leading to insightful discoveries such as introducing discounting factors into MAB agents and adding robustness to noise. Furthermore, Table 6.2 shows VPR-Gym's overhead using the CMA agent on the neuron circuit which belongs to the Titan23 benchmarks, in terms of the time spent between the point the agent receives a reward and the point the agent takes an action in the next step, and its proportion in the total running time. Although the CMA agent can perform calculations fast, the overhead is mainly caused by Inter-process communication between VPR and the OpenAI Gym framework. This means that VPR-Gym can mainly be used as a tool to fast investigate, prototype, select, and improve a wide range of AI techniques, while the optimized algorithms should still be transplanted to the original VPR kernel to further conduct and finalize FPGA placement in practice.



(a)



(b)

Figure 6.2: Comparison of different algorithms on VprEnv_blk_type at 6 different runtime points. The lower the plotted metrics better. TBPSA outperforms other algorithms on both post-placement estimated wirelength and CPD

Table 6.2: CMA agent's Overhead at 6 different runtime points

Inner_num	Overhead (s)	Total Time (s)	Overhead (%)
0.1	48.40	97.92	49.43
0.2	93.11	155.93	59.71
0.3	129.42	204.03	63.43
0.4	185.86	280.81	66.19
0.5	209.44	312.96	66.92
0.6	275.28	386.41	71.24

Chapter 7

Conclusions & Future Work

7.1 Conclusions

In this thesis, we conducted an investigation of a wide range of AI techniques' performance on SA-based FPGA placement. In order to fully explore the potential of different AI techniques, we propose the VPR-Gym for the sake of lowering the barrier-of-entry to the development of AI optimization algorithms for FPGA RL placement. VPR-Gym significantly simplifies the implementation of agents in the VPR placement research area and enables the interaction between VPR and some advanced Machine Learning and optimization libraries such as Nevergrad [21]. Furthermore, we performed experiments on VPR-Gym and provided exciting findings that can help enhance the FPGA CAD tools' placement performance from the algorithmic perspective as the following:

- In the original setting, applying algorithms with robustness to non-stationarity, is able to improve the performance;
- The effectiveness of ES algorithms demonstrates that it is unnecessary to perform updates upon every single SA move, which can significantly reduce the time spent on the AI end;
- When appending the action space, such as adding block type selection, ES algorithms, especially TBPSA, perform better than MAB algorithms.

7.2 Future Work

In the future, we plan to extend VPR-Gym's features. The current trend of RL is using deep neural networks as the agent policy. We will focus on adding state information to VPR-Gym to facilitate studies of Deep Q-Network (DQN) [49], contextual bandit [50, 51], and other RL techniques in our future work. Another interesting avenue is shaping the reward function, which can better relate to the post-placement estimated wirelength and CPD metrics. Last but not least, we plan to consider multi-agent systems. At present, the Sequential Annealing (SA) process within VPR operates sequentially and is thus unable to harness the CPU's multi-core capabilities. By duplicating the VPR with multiple copies to different agents and periodically replacing the placement result with the best solution among all the agents, multiple agents can work in a parallel and competing manner.

We can also consider the techniques investigated in this thesis in other FPGA CAD steps outside of the placement problem. For example, [52] uses epsilon-greedy to speed up the routing step while maintaining a similar or better QoR as compared to the conventional negotiation-based congestion-driven routing solution. According to our findings in this thesis, there are multiple possible ways to further improve the routing performance: one way is to use more robust MAB algorithms such as DTS; the other way is to re-formulate the problem and apply other AI techniques such as ES.

Bibliography

[1] C. Nast and W., *Microsoft Supercharges Bing Search With Programmable Chips*, https://www.wired.com/2014/06/microsoft-fpga/, Jun. 2014.

[2] *Project Catapult - Microsoft Research*, https://www.microsoft.com/en-us/research/project/project-catapult/.

[3] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006. DOI: 10.1109/N-SSC.2006.4785860.

[4] *With 18.5 million logic cells, AMD's Versal VP1902 Premium Adaptive SoC becomes "World's Largest FPGA"*, https://www.eejournal.com/article/with-18-5-million-logic-cells-amds-versal-vp1902-premium-adaptive-soc-becomes-worlds-largest-fpga/, Jul. 2023.

[5] J. H. Anderson and T. S. Czajkowski, "Computer-aided design for FPGAs: Overview and recent research trends,"

[6] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic cad," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–8. DOI: 10.1109/FPL.2013.6645503.

[7] W. Wang, Q. Meng, and Z. Zhang, "A survey of FPGA placement algorithm research," in *2017 7th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2017, pp. 498–502. DOI: 10.1109/ICEIEC.2017.8076614.

[8] K. E. Murray *et al.*, "VTR 8: High-performance cad and customizable FPGA architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[9] M. A. Elgammal, K. E. Murray, and V. Betz, "Rlplace: Using reinforcement learning and smart perturbations to optimize FPGA placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2532–2545, 2022. DOI: 10.1109/TCAD.2021.3109863.

[10] M. A. Elgamma, K. E. Murray, and V. Betz, "Learn to place: FPGA placement using reinforcement learning and directed moves," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 85–93. DOI: 10.1109/ICFPT51103.2020.00021.

[11] K. E. Murray and V. Betz, "Adaptive FPGA placement optimization via reinforcement learning," in *2019 ACM/IEEE 1st Workshop on Machine Learning for CAD (MLCAD)*, 2019, pp. 1–6. DOI: 10.1109/MLCAD48534.2019.9142079.

[12] K. Vorwerk, A. Kennings, and J. W. Greene, "Improving simulated annealing-based FPGA placement with directed moves," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 179–192, 2009. DOI: 10.1109/TCAD.2008.2009167.

[13] P. Maidee, C. Ababei, and K. Bazargan, "Timing-driven partitioning-based placement for island style FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 3, pp. 395–406, 2005. DOI: 10.1109/TCAD.2004.842812.

[14] J. Zhao, Q. Zhou, and Y. Cai, "Fast congestion-aware timing-driven placement for island FPGA," in *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2009, pp. 24–27. DOI: 10.1109/DDECS.2009.5012092.

[15] P. Maidee, C. Ababei, and K. Bazargan, "Fast timing-driven partitioning-based placement for island style FPGAs," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, 2003, pp. 598–603. DOI: 10.1145/775832.775984.

[16] Z. Abuowaimer *et al.*, "Gplace3.0: Routability-driven analytic placer for ultrascale FPGA architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 5, 2018, ISSN: 1084-4309. DOI: 10.1145/3233244. [Online]. Available: https://doi.org/10.1145/3233244.

[17] W. Li, Y. Lin, and D. Z. Pan, "Elfplace: Electrostatics-based placement for large-scale heterogeneous FPGAs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942075.

[18] T. Martin, D. Maarouf, Z. Abuowaimer, A. Alhyari, G. Grewal, and S. Areibi, "A flat timing-driven placement flow for modern FPGAs," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[19] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[20] M. Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[21] J. Rapin and O. Teytaud, *Nevergrad - A gradient-free optimization platform*, https://GitHub.com/FacebookResearch/Nevergrad, 2018.

[22] K. E. Murray *et al.*, "Vtr 8: High-performance cad and customizable FPGA architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[23] A. Al-Hyari, A. Shamli, T. Martin, S. Areibi, and G. Grewal, "An adaptive analytic FPGA placement framework based on deep-learning," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 3–8. DOI: 10.1145/3380446.3430618.

[24] Z.-H. Zhan, L. Shi, K. C. Tan, and J. Zhang, "A survey on evolutionary computation for complex continuous optimization," *Artificial Intelligence Review*, pp. 1–52, 2022.

[25] G. Brockman *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[26] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[27] K. Azizzadenesheli, "Maybe a few considerations in reinforcement learning research?," 2019.

[28] C. Cummins *et al.*, "Compilergym: Robust, performant compiler optimization environments for ai research," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2022, pp. 92–105.

[29] L. N. Alegre, *SUMO-RL*, https://github.com/LucasAlegre/sumo-rl, 2019.

[30] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[31] P. Gawłowicz and A. Zubow, "Ns-3 meets openai gym: The playground for machine learning in networking research," in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019, pp. 113–120.

[32] M. Aboelwafa, G. Alsuhli, K. Banawan, and K. G. Seddik, "Self-optimization of cellular networks using deep reinforcement learning with hybrid action space," in *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, IEEE, 2022, pp. 223–229.

[33] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[34] N. Cesa-Bianchi, C. Gentile, G. Lugosi, and G. Neu, "Boltzmann exploration done right," *Advances in neural information processing systems*, vol. 30, 2017.

[35] T. L. Lai, H. Robbins, *et al.*, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[36] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[37] N. Cesa-Bianchi and P. Fischer, "Finite-time regret bounds for the multiarmed bandit problem.," in *ICML*, Citeseer, vol. 98, 1998, pp. 100–108.

[38] T. L. Lai, H. Robbins, *et al.*, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[39] I. A. M. Huijben, W. Kool, M. B. Paulus, and R. J. G. van Sloun, "A review of the gumbel-max trick and its extensions for discrete stochasticity in machine learning," *CoRR*, vol. abs/2110.01515, 2021. arXiv: 2110.01515. [Online]. Available: https://arxiv.org/abs/2110.01515.

[40] C. Hartland, S. Gelly, N. Baskiotis, O. Teytaud, and M. Sebag, "Multi-armed bandit, dynamic environments and meta-bandits," 2006.

[41] V. Raj and S. Kalyani, "Taming non-stationary bandits: A bayesian approach," *arXiv preprint arXiv:1707.09727*, 2017.

[42] P. A. Ortega and D. A. Braun, "A minimum relative entropy principle for learning and acting," *Journal of Artificial Intelligence Research*, vol. 38, pp. 475–511, 2010.

[43] M. Oquab, J. Rapin, O. Teytaud, and T. Cazenave, "Parallel noisy optimization in front of simulators: Optimism, pessimism, repetitions, population control," in *Workshop Data-driven Optimization and Applications at CEC*, 2019.

[44] N. Hansen, "The CMA evolution strategy: A comparing review," *Towards a new evolutionary computation: Advances in the estimation of distribution algorithms*, pp. 75–102, 2006.

[45] Wikipedia contributors, *CMA-ES — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=CMA-ES&oldid=1169643638, [Online; accessed 11-August-2023], 2023.

[46] M. Hellwig and H.-G. Beyer, "Evolution under strong noise: A self-adaptive evolution strategy can reach the lower performance bound-the pccmsa-es," in *Parallel Problem Solving from Nature–PPSN XIV: 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings*, Springer, 2016, pp. 26–36.

[47] A. Auger and N. Hansen, "A restart CMA evolution strategy with increasing population size," in *2005 IEEE Congress on Evolutionary Computation*, vol. 2, 2005, 1769–1776 Vol. 2. DOI: 10.1109/CEC.2005.1554902.

[48] L. Besson, *SMPyBandits: an Open-Source Research Framework for Single and Multi-Players Multi-Arms Bandits (MAB) Algorithms in Python*, Online at: GitHub.com/SMPyBandits/SMPyBandits, Code at https://github.com/SMPyBandits/SMPyBandits/, documentation at https://smpybandits.github.io/, 2018. [Online]. Available: https://github.com/SMPyBandits/SMPyBandits/.

[49] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[50] J. Langford and T. Zhang, "The epoch-greedy algorithm for contextual multi-armed bandits," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, ser. NIPS'07, Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, 817–824, ISBN: 9781605603520.

[51] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proceedings of the 19th international conference on World wide web*, ACM, 2010. DOI: 10.1145/1772690.1772758. [Online]. Available: https://doi.org/10.1145%2F1772690.1772758.

[52] I. Baig and U. Farooq, "Efficient detailed routing for FPGA back-end flow using reinforcement learning," *Electronics*, vol. 11, no. 14, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11142240. [Online]. Available: https://www.mdpi.com/2079-9292/11/14/2240.

[53] A. Mirhoseini *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[54] A. Alhyari, A. Shamli, Z. Abuwaimer, S. Areibi, and G. Grewal, "A deep learning framework to predict routability for FPGA circuit placement," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 334–341. DOI: 10.1109/FPL.2019.00060.

[55] Z. He, L. Zhang, P. Liao, Y. Ma, and B. Yu, "Reinforcement learning driven physical synthesis : (invited paper)," in *2020 IEEE 15th International Conference on Solid-State Integrated Circuit Technology (ICSICT)*, 2020, pp. 1–4. DOI: 10.1109/ICSICT49897.2020.9278350.

[56] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL*, vol. 97, 1997, pp. 213–222.

[57] M. Leeser, S. Handagala, and M. Zink, "Fpgas in the cloud," *Computing in Science Engineering*, vol. 23, no. 6, pp. 72–76, 2021. DOI: 10.1109/MCSE.2021.3127288.