Learn to Floorplan through Acquisition of Effective Local Search Heuristics

Zhuolun He[1], Yuzhe Ma[1], Lu Zhang[1], Peiyu Liao[1], Ngai Wong[2], Bei Yu[1], Martin D.F. Wong[1]
[1]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[2]Department of Electrical and Electronic Engineering, The University of Hong Kong
{zlhe,yzma,lzhang,pyliao,byu,mdfwong}@cse.cuhk.edu.hk, nwong@eee.hku.hk

***Abstract*—Automatic heuristic design through reinforcement learning opens a promising direction for solving computationally difficult problems. Unlike most previous works that aimed at solution construction, we explore the possibility of acquiring local search heuristics through massive search experiments. To illustrate the applicability, an agent is trained to perform a walk in the search space by selecting a candidate neighbor solution at each step. Specifically, we target the floorplanning problem, where a neighbor solution is generated through perturbing the sequence pair encoding of a floorplan. Experimental results demonstrate the efficacy of the acquired heuristics as well as the potential of automatic heuristic design.**

***Index Terms*—Floorplanning, sequence pair, reinforcement learning.**

I. Introduction

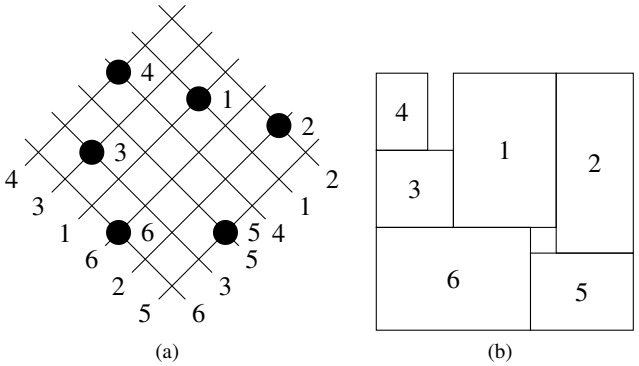
Electronic Design Automation (EDA) lies at the heart of modern computer science technologies. Various computationally challenging (viz. NP-hard) problems gave birth to the exciting progress in solving techniques, most among which are hand-crafted heuristics that are carefully designed by domain experts and scientists. Yet, the fantasy of automatic algorithm design for difficult problems has never been shattered.

Reinforcement learning has recently offered a promising direction for such a dream. By interacting with the environment and training an agent to survive, a learning system can in principle create new knowledge about the space which the agent live in. In addition to the breakthroughs in game playing [1] and robotic control [2], the community also spent efforts in the discipline of combinatorial optimizations. An end-to-end *actor-critic* training framework [3] based on the *pointer network* architecture (a variant of recurrent neural network) is proposed to tackle the Travelling Salesman Problem (TSP). Later on, *structure2vec* (a graph embedding network) and the off-policy *Q-learning* are utilized [4] to solve TSP and other optimization problems over graph. Apart from TSP, researchers have also achieved significant results on Vehicle Routing Problem (VRP) [5], Job Scheduling [6], and Satisfiability Problem (SAT) [7]. There has been literature on applying reinforcement learning techniques to EDA problems. An RL agent is trained through *Proximal Policy Optimization* (PPO) to place macro blocks in a canvas [8]. In their settings, the netlist graph and other input features are encoded through an embedding network that is trained in a supervised manner, then the policy network generates the placement position of the macro blocks in their descending order of size, followed by conventional standard cell placement and legalization; the weighted design cost including wirelength and congestion are used as the reward signal. There are also attempts to global routing [9], detailed routing [10], and Network-on-Chip design [11].

In this paper we consider the problem of floorplanning, the first step in physical synthesis, which aims to roughly determine geometric relationship among circuit modules and to estimate the cost of the design. Various data structures are introduced for the representation of the geometric relation. A slicing floorplan, where the whole design can be recursively divided horizontally or vertically until each part contains only one module, is naturally encoded by a binary tree, whose internal nodes are for the horizontal or vertical cuts and the leaves denote the modules. Equivalently, polish expression is used [12] to encode the postfix of the same binary tree. As for general floorplans without a slicing structure (i. non-slicing), many other elegant representations are invented, e.g., O-tree [13], B*-tree [14], Sequence Pair [15], and Twin Binary Sequences [16]. With a flexible and effective representation, good floorplan results could be achieved through constructing or perturbing a data structure in a systematic way [13], by heuristics [17], or by some means of meta-heuristics like *genetic algorithms* [18] or *simulated annealing* [19]. Despite of that, specialized knowledge is a must for a successful design, as well as a considerable amount of trial-and-errors, prohibiting the development of new algorithms in some sense. Besides, as the transistor technology node scaling down, modern circuit design has become much more complicated, and the ever-increasing number of modules in a chip brings about the scalability issue, emphasizing the demand for effective algorithms that work well on large-scale cases. All the above problems motivate us to utilize reinforcement learning for automatic algorithm design.

The idea of enhancing local search heuristics with reinforcement learning is not that new. Researchers seek to boost the local search by selecting a good starting point [20], [21], by tuning search parameters [22], by scaling a regularization term [23], or by switching between heuristics on the fly [24]. Basically, all these work adopt an existing local search algorithm, and improve a few settings of that algorithm with reinforcement learning. Our work takes one step forward: we aim to acquire a local search algorithm from the scratch, i.e., we avoid to introduce too much prior human knowledge during the search that might mislead the learning.

This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14209420).



(a) (b)

Fig. 1 (a) The oblique grid [25] shows the relative position between blocks for sequence pair (<4 3 1 6 2 5>, <6 3 5 4 1 2>); (b) The corresponding packing. The dimensions for the 6 blocks are: 1(4×6), 2(3×7), 3(3×3), 4(2×3), 5(4×3), 6(6×4).

The remainder of the paper is organized as the following. Section II recaps the preliminaries. Section III illustrates the selection of features and the model. Section IV discusses the training pipeline of the model. Section V is for experimental results, and Section VI concludes the paper.

II. Preliminaries

A. Floorplan

In general, floorplanning is to generate relative locations for modules. Given a set of n rectangular blocks $B = \{b_1, b_2, \cdots, b_n\}$ and a netlist N specifying their connections, a floorplan F seeks a planar location assignment $(\boldsymbol{x}, \boldsymbol{y})$ of B, providing no module overlap, to minimize the total chip area and to reduce the total wirelength.

Based on area function $A(\cdot)$ and wirelength function $W(\cdot)$, the optimization problem is formulated as follows

$$\begin{aligned} \min_{F} \quad & A(F) + \alpha W(F) \\ \text{s.t.} \quad & F \text{ is a legal solution.} \end{aligned} \tag{1}$$

Definition 1 (Sequence Pair Representation)**.** *A sequence pair* (Γ_+, Γ_-) *is a pair of sequences of* n *elements that imposes the relationship between each pair of blocks as follows [25]:*

$$(< ..b_i..b_j.. >, < ..b_i..b_j.. >) \implies b_i \text{ is to the left of } b_j;$$
$$(< ..b_j..b_i.. >, < ..b_i..b_j.. >) \implies b_i \text{ is below } b_j.$$

As an example, Fig. 1 shows the imposed relationship by the sequence pair (<4 3 1 6 2 5>,<6 3 5 4 1 2>) in an oblique grid. It can be inferred from the figure that all the imposed relationship constraints are satisfiable, see [15] for the prove.

In fact, given any sequence pair, one of the area-optimal packing subject to the constraints can be obtained in $\mathcal{O}(n \log\log n)$ time [25] based on a fast longest common subsequence computation.

B. Local Search

Local search is a popular heuristic method for solving combinatorial optimization problems. Roughly speaking, a local search algorithm starts off with an initial solution and then continually tries to find better solutions by searching neighbourhoods [26].

Simulated annealing (SA) [27] is a probabilistic technique inspired from annealing in metallurgy. Formally, let S be the finite set of all complete solutions, $\mathcal{E} : S \to \mathbb{R}$ be an energy function defined on S, and $\mathcal{N} : S \to S$ be a neighbor function. Note that for each $s \in S, \mathcal{N}(s) \subset S - \{s\}$. SA starts at a state $s \in S$. At each step, SA considers some neighboring state $s' \in \mathcal{N}(s)$ of the current state s, and decides between moving the system to state s' or staying in state s based on the acceptance probability given as follows:

$$P(s', s, T) = \exp\left[\frac{1}{T} \max\left(0, \mathcal{E}(s') - \mathcal{E}(s)\right)\right], \tag{2}$$

where $T \in \mathbb{R}$ is the temperature to control how 'bad' moves are accepted. These probabilities ultimately lead the system to move to states of lower energy.

C. Reinforcement Learning

Reinforcement learning learns the mapping from states to actions, so as to maximize a numerical reward signal [28]. We use the agent-environment interface [28] to describe it specifically: at each time step t, the agent receives state s_t that represents the current state of the environment, and on that basis selects an action a_t. One time step later, the agent receives a numerical reward $R_{t+1} \in \mathbb{R}$, and finds itself in a new state s_{t+1}. This framework is a considerable abstraction of the goal-directed learning problem from interaction.

A *Markov Decision Process* (MDP) is a fundamental formalization for reinforcement learning in stochastic domain. A typical Markov decision process contains the following components:

- States S: a finite set of representations of the environment state.
- Actions A: a finite set of responses to the stimulus an agent can take.
- Transition Model $\mathcal{T}(s, a, s')$: the transition probability from state s to s' under action a.
- Rewards R: the reward signal.

Given an MDP (S, A, T, R), a policy is a mapping from state space to action space $\pi : S \to A$. For any given MDP, our goal is to find a best policy π^* that receives the highest expected reward.

III. Algorithm

The purpose of this work is to acquire an effective local search heuristic, with which good floorplans could be obtained through local search. In this section, we will first formulate the above problem as a reinforcement learning problem, and then we discuss how to select features to represent the states, and how to construct an agent that makes decision.

A. Local Search as a Reinforcement Learning Problem

We first formally define the MDP for our local search problem.

a) State Space: As introduced in Section II-B, a state s is a complete solution. Here a complete solution includes both sequences Γ_+, Γ_-, as well as the orientation of each block.

b) Action Space: Several perturbations are defined on a solution s to generate $\mathcal{N}(s)$. At each step, a subset of neighbours $s' \subset \mathcal{N}(s)$ are sampled, and the agent decides to accept one of the neighbours or reject to move. Therefore, the action space size is the number of sampled neighbours plus one (for the reject). We allow 6 types of perturbation on a solution:

1) Exchange two blocks in Γ_+.
2) Exchange two blocks in Γ_-.
3) Exchange two blocks in both Γ_+ and Γ_-.
4) Delete one block and insert back to a random position in both Γ_+ and Γ_-.
5) Rotate one block by $90°$.
6) Flip one block.

c) Transition: The transition model of our MDP is deterministic, viz., no randomness is involved once the state and the action are given. For this reason, we reload the transition model with signature $\mathcal{T} : S \times A \to S$. Hereafter, we use $s' \leftarrow \mathcal{T}(s, a)$ to denote a state transition.

d) Reward: Assigning rewards to the actions is critical in reinforcement learning. Since the agent seeks to maximize the total reward, the action of maximizing total rewards should be consistent with the target of the problem. Basically, the rewards are assigned whenever a global better solution is found. We use the reduction of energy (i.e., $\Delta\mathcal{E}$) as the reward, so that maximizing the total reward $\sum \Delta\mathcal{E}$ is equivalent to minimizing the cost. The reward is normalized to $[0, 1]$ by comparing with the energy of the initial state:

$$R = \frac{\Delta\mathcal{E}}{baseline}. \tag{3}$$

Intuitively, the reward encourages the design with lower cost, which aligns with our target of reducing area and wirelength.

To accelerate training, we empirically add two adversarial rewards to discourage useless explorations. First, if the agent decides to reject, while one of the sampled neighbour has lower energy than the current state, a negative reward of -0.01 is given. In other words, the agent is always encouraged to move to a neighbor state with lower energy. Second, if the agent accepts a state, whose energy is higher than 1.2 times of the lowest energy neighbor, a negative reward of -0.01 is given. This is to discourage the agent to search a high energy region that can hardly contain a good solution.

B. Features

As mentioned above, neighbour solutions are sampled from all the 6 types of perturbation. To help the agent make better decisions, providing helpful features to guide the local search is critical. Here three sets of features are included:

TABLE I Features for decision making. All the items are normalized to $[-1, 1]$ or $[0, 1]$.

Index	Item	Range	Description
0	$\mathcal{E}(s)$	[-1, 1]	Current energy
1	$\mathcal{E}(s')$	[-1, 1]	Neighbor energy
2	$\mathcal{E}(s^*)$	[-1, 1]	Lowest energy so far
3	$\overline{\mathcal{E}}$	[-1, 1]	Average energy
4	$\overline{\mathcal{E}^*}$	[-1, 1]	Average energy since s^*
5	$\mathcal{E}'_*$	[-1, 1]	Lowest energy of sampled neighbours
6	$Area(b_i)$	[0, 1]	Size of the perturbed block
7	aff	[0, 1]	Number of effected blocks
8	t	[0, 1]	Search progress

- **Energy**: The central goal of the local search is to find a state with minimal energy, which is defined as the negative cost given by Equation (1) after packing all the blocks. In simulated annealing, accepting or rejecting a move is based on current state energy $\mathcal{E}(s)$ and the neighbor state energy $\mathcal{E}(s')$. Besides that, we provide a set of other energy-related statistical information, including the lowest and average energy through the search path ($\mathcal{E}(s^*)$ and $\overline{\mathcal{E}}$), the average energy on the search path since the lowest energy state ($\overline{\mathcal{E}^*}$), and the lowest energy among the sampled neighbors ($\mathcal{E}'_*$).
- **Effect**: How does a move affect the whole floorplan solution? We identify the effect by both the size of the perturbed block(s), as well as the number of blocks that are affected. A block is considered affected if and only if the location of the block is related to the perturbed block(s). For example, if we rotate one block, those blocks above or on the right of this block will be affected. Other situations are also analyzed according to the packing of sequence pair.
- **Progress**: Intuitively, the search progress has a similar role to the temperature scheduling. Therefore we included the search progress as a feature. Although we also included an early-stop mechanism, the progress of early-stopping is not visible to the agent because it is considered irrelevant to the local search.

We list in TABLE I the detailed features, which we concatenate into a vector to feed the agent. Note that all the features are normalized for easier training. For this purpose, we denote the energy of the initial state as e_0, and normalize all the energy related entries by $\min(1, \mathcal{E}/e_0 - 1)$.

In practice, we realize that including noisy features does harm to, or at least slows down training. For example, we tried to include an indicator to tag the perturbation type, which turns out to be a noise as an input feature.

C. Neural Network as the Agent

Recall that the policy π of an agent is a mapping from the state space to the action space. A straightforward way to construct such a policy is to store the best action of each possible state. However, a quick estimation shows that for n blocks, there are totally $(n!)^2 \times 8^n$ different states (two permutation and rotation/flip of the blocks), which makes the tabular method infeasible. Therefore, we utilize a neural

network as the policy approximator. Since the input features are concatenated into an 1D vector, a simple multi-layer perceptron (MLP) should be good in our case, where the input dimension equals to the feature dimension, and the output dimension is always 1 for the value prediction. A ReLU layer is inserted after all but the last layers as the activation. The neural network is trained with back-propagation, as will be illustrated in detail in Section IV.

IV. Training

A. Dealing with Large Action Spaces

Reasoning in an environment with a large number of possible discrete actions is always challenging, as exploring a large action space will be unstable and inefficient, and thus requires much more training efforts. As concrete examples, there will be totally 16575 possible actions in each state for a floorplan problem on 25 blocks, and 1015050 actions for 100 blocks. Prior work proposed several strategies to improve learning in large action spaces. Factorizing the action space into binary subspaces [29] is natural for the cases with many action variables or with a finely discretized continuous action space. However the method requires a fixed size action space, while our action space size is a polynomial of the size of the problem. Other paper suggested embedding the discrete actions into a continuous space [30] and eliminating actions with an extra training signal [31], both of which aim to prune irrelevant solutions to improve the speed and the quality of training. Despite of that, evaluating an action is extremely costly in our local search formulation, since both the energy of a state and the estimated value of the action need to be calculated, making the above solutions still intractable. Inevitably, we have to sample actions during both training and testing.

B. Deep Q-Learning

We use deep Q-learning [32] (DQN) to estimate the value of each move. Following the common practice, we define the value q of taking action a at state s under policy π as the expected return starting from that state:

$$q_\pi(s,a) := \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s, A_t = a \right], \quad (4)$$

where t is the timestamp and γ is the discounting factor. The optimal action value is therefore given by $q^*(s,a) := \max_\pi q(s,a)$. As discussed in III-C, we use a neural network to approximate the action value, $Q(s,a;\theta) \approx q^*(s,a)$. The loss function is defined according to the *bellman equation*:

$$\begin{aligned} L(\theta_i) &= \mathbb{E}_{s,a\sim\rho(\cdot)} \left[\left(y_i - Q(s,a;\theta_i) \right)^2 \right] \\ &= \mathbb{E}_{(\cdot)} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i) \right)^2 \right], \end{aligned} \quad (5)$$

where $\rho(\cdot)$ is the behaviour distribution over s and a, and y_i is called the target of the update. Then we can update the model through back-propogating the loss with respect to the weights ($\nabla_{\theta_i} L(\theta_i)$). DQN is considered as *off-policy* because it estimates the greedy policy ($\max_a(Q(s,a;\theta))$, while samples the state space following a behaviour policy. The overall algorithm is listed as Algorithm 1.

Algorithm 1 DQN for Floorplan Local Search

```
function Train()
    Initialize replay memory D;
    Initialize Q with random weights;
    Initialize ϵ;
    for episode ← 1, ..., M do
        Reset s ← s_1;
        for t ← 1, ..., T do
            if ϵ > u ∼ U(0,1) then
                Sample a random action a_t;
            else
                Select a_t ← max_a Q(s_t, a_t; θ);
            end if
            s_{t+1} ← T(a_t, s_t);
            Save experience (s_t, a_t, r_{t+1}, s_{t+1}) to D;
            UpdateModel();
            Update ϵ;
        end for
    end for
end function

procedure UpdateModel()
    Sample a batch of (s_j, a_j, r_{j+1}, s_{j+1}) from D;
    if s_{j+1} is a terminal state then
        y_j ← r_{j+1};
    else
        y_j ← r_{j+1} + γ max_{a'} Q(s_{j+1}, a'; θ);
    end if
    Calculate L ← (y_j − Q(s_j, a_j; θ))^2;
    Update θ with back-propagation;
end procedure
```

We select to use an off-policy algorithm for the following reasons:

1) A local search episode is typically very long (e.g. 10k steps). The consecutive moves are highly correlated, which does harm to training. Instead, in an off-policy algorithm we sample experiences from the replay memory to break the strong correlations and thus reduce the variance.
2) Sampling is not free. Packing and calculating wirelength is somehow time-consuming compared to running a model. With the replay memory, each data sample is potentially used many times in training, which increase data efficiency.
3) Exploration is necessary. If we learn online a greedy policy that picks the action with the largest expected return, then we always select the same action during training. An off-policy algorithm naturally decouples sampling behaviour and the learnt policy, allowing more random exploration in training while still being greedy in testing.

C. Convergence Analysis

With the MDP defined (S, A, T, R) in Section III, the Q-learning algorithm is given by the iteration rule

$$Q_{t+1}(s_t, a_t) \longleftarrow Q_t(s_t, a_t) + \alpha(s_t, a_t)\left[r_t + \gamma \max_{b \in A_t} Q_t(s_{t+1}, b) - Q_t(s_t, a_t)\right], \quad (6)$$

where A_t is a subset of the original action space A after random sampling, $\gamma < 1$. Considering that the action space A is extremely large and intractable for efficient training, intuitively we randomly sample the action space in each iteration.

Obviously, $\max_{b \in A_t} Q_t(s_{t+1}, b) \leq \max_{b \in A} Q_t(s_{t+1}, b)$, and thus iteration rule (6) will not return us the same Q-function if the action space differs, even if it converges. We will prove that the random iterative process (6) converges to a Q-function which is different from the optimal one $Q*$.

Theorem 1. *Given the MDP defined (S, A, T, R) in Section III, the iteration rule (6) converges with probability 1, as long as constraints*

$$\sum_t \alpha_t(s, a) = \infty \qquad \sum_t \alpha_t^2(s, a) < \infty \quad (7)$$

are satisfied for all $(s, a) \in S \times A$.

To establish this theorem some mathematical results from stochastic approximation are required. The following theorem is much more powerful and general than Theorem 1. We present it as a lemma and the proof can be found in [33].

Theorem 2. *The random process $\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \beta_t(x)F_t(x)$ converges to zero with probability 1 (w.p.1) under the following assumptions:*

1) *The state space is finite.*
2) *The following constraints are satisfied.*

$$\sum_t \alpha_t(x) = \infty \qquad \sum_t \alpha_t^2(x) < \infty$$
$$\sum_t \beta_t(x) = \infty \qquad \sum_t \beta_t^2(x) < \infty \quad (8)$$

   and $\mathbb{E}[\beta_t(x)|P_t] \leq \mathbb{E}[\alpha_t(x)|P_t]$ uniformly w.p.1.
3) *$\|\mathbb{E}[F_t(x)|P_t]\|_W < \gamma \|\Delta_t\|_W$, where $\gamma \in (0, 1)$.*
4) *$\mathbf{Var}[F_t(x)|P_t] \leq C(1 + \|\Delta_t\|_W)^2$, where C is a constant.*

Here $P_t = \{\Delta_t, \Delta_{t-1}, \cdots, F_{t-1}, \cdots, \alpha_{t-1}, \cdots, \beta_{t-1}, \cdots\}$ stands for the past at step t. $F_t(x)$, $\alpha_t(x)$ and $\beta(x)$ are allowed to depend on the past insofar as the above conditions remain valid. The notation $\|\cdot\|_W$ refers to some weighted maximum norm.

The full proof of Theorem 2 is complicated and out of the scope of this paper. With the help of Theorem 2, we are able to prove Theorem 1.

V. Experimental Results and Discussions

A. Setup

We implemented the proposed solution in python, while trained the neural model with PyTorch [34]. We use Adam [35] as the optimizer with an initial learning rate of $5 * 10^{-4}$.

To encourage exploration, the exploration factor ϵ is initially set to be 1. Then it is linearly annealed to 0.1 in the first 15000 steps and fixed afterwards. We use a replay memory of size 20000 that stores the most recent experiences. During training, a batch of 128 experiences are sampled from the replay memory. The discount factor is set to be 0.995. To stabilize training, we fix the target network and synchronize the trained policy network to it every 10 episodes.

B. Data Generation

Random netlists are generated as training samples. Due to the great difference in problem sizes, we train two models, one of which (the lite one) is for MCNC and the other (the large one) is for GCRS benchmarks. Each netlist for training the lite model consists of 50 blocks with integer width and height in the range of $[10, 100]$. Each block has 3 pins at random locations, and 50 signals are randomly generated, each of which connects 3 random pins. The large network is trained with netist consisting of 250 blocks, 10 pins for each block, and 250 signals. The rest settings are the same.

C. Baseline Tuning

Parameters of simulated annealing greatly affect the performance of the algorithm. For example, a low initial temperature may have the search stuck at a poor local minima, while an over-high temperature just accepts all the moves, walking in the search space in vain. Therefore, we first ran a batch of experiments to tune the parameters of simulated annealing, as listed in TABLE II. Best obtained result for each case appears in different settings, while in general higher initial temperature and higher number of inner loops yield better results, which is intuitively very reasonable. To this regard, we will use setting 8 in subsequent experiments, where the initial temperature is set to be 10^7, end temperature to be 10^{-10}, inner loop of 200, and an exponential cooling factor of 0.97.

D. MCNC Benchmark

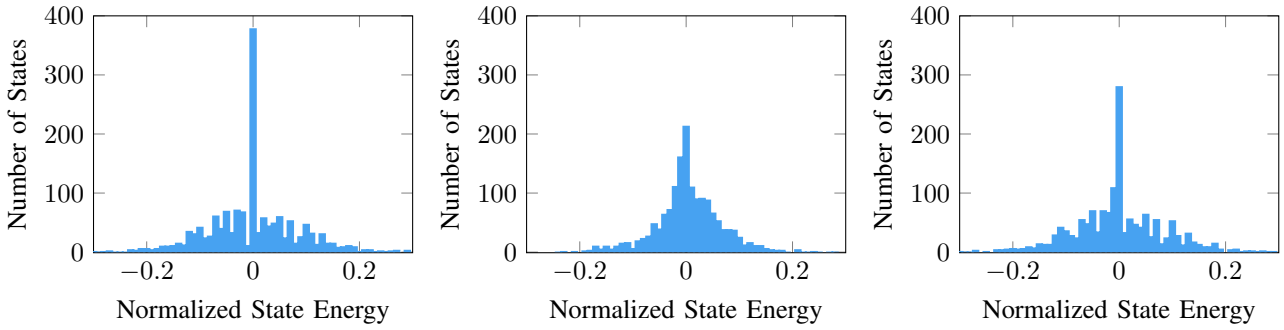
We tested our agent on the MCNC netlist [36] benchmark.

1) Area Minimization: We first investigate the problem of area minimization. Since we trained our model on area and wirelength minimization, we directly use the area of the resulted solution to compare. We compare the results of our agent and Simulated Annealing (SA). The results are listed in TABLE III.

2) Area and Wirelength Minimization: Then we switch to the problem of minimizing both area and wirelength. The multi-objectives actually make the problem much more difficult. In area minimization, it is often the case that we switch two internal blocks and the area does not change at all. With wirelength minimization, however, a bad swap may

TABLE II Tuning Simulated Annealing parameters, including initial temperature, end temperature, number of inner loop, and cooling schedule. Best result (in bold) for each case is obtained in different settings.

Setting	Parameters				Area (mm^2)				
	Init T	End T	Inner loop	Cooling	apte	xerox	hp	ami33	ami49
1	10^6	10^{-10}	50	0.97	48.28	21.77	9.72	1.24	38.87
2	10^7	10^{-10}	25	0.97	**47.08**	20.54	9.38	1.31	39.65
3	10^7	10^{-10}	50	0.95	47.56	20.40	9.63	1.30	39.29
4	10^7	10^{-9}	50	0.97	47.56	20.36	9.33	1.26	40.84
5	10^7	10^{-10}	50	0.97	47.56	20.36	9.33	1.25	40.84
6	10^7	10^{-11}	50	0.97	47.56	20.36	9.33	1.25	40.70
7	10^7	10^{-10}	100	0.97	48.71	21.10	9.21	**1.20**	38.38
8	10^7	10^{-10}	200	0.97	**47.08**	**20.31**	9.26	1.22	**38.10**
9	10^8	10^{-10}	50	0.97	47.52	21.23	**9.17**	1.28	38.72



(a) (b) (c)

Fig. 2 Neighbor solution distribution for (a) area minimization; (b) wirelength minimization; (c) area and wirelength minimization.

TABLE III Area Minimization on MCNC Benchmark. Our results are directly from minimizing area and wirelength together, while the two other columns are area minimization only. Better results are emphasized in bold.

Circuit	module #	Area ($\times 10^5$)		
		Ours	SA	FAST-SP [25]
apte	9	**47.08**	**47.08**	*46.92*
xerox	10	20.42	**20.31**	*19.80*
hp	11	**9.21**	9.26	*8.95*
ami33	33	1.24	**1.22**	*1.21*
ami49	49	38.65	**38.10**	*36.50*

greatly increase the wirelength, and thus the overall cost. To verify this idea, we invested a random state in ami33 and the distribution of the neighbor solution are shown in Fig. 2.

TABLE IV shows the results for area and wirelength minimization. Our agent outperforms the simulated annealing algorithm in all the 5 cases.

E. GCRS Benchmark

We further conducted experiments on the GCRS benchmark [37] with larger instances of hundreds of blocks. We tested area and interconnect optimization and listed the results in TABLE V. Due to the long runtime of the instances, we carefully tune the early-stop criteria on both simulated annealing and our agent. In simulated annealing, the search will finish if no move was accepted in the last 20 temperatures. In our agent, the search will finish if no better solution was found in the last 100 steps.

F. Result Visualizations and Discussions

1) Search Progress Visualization: We recorded the search progress of Simulate Annealing and our agent, and visualized them in Fig. 3. From the figure, we observe that Simulated Annealing explores the action space during searching while our agent searches much smoother. We believe this is because our agent acquires a rather greedy and deterministic heuristic.

2) Floorplan Visualization: We visualized the floorplans generated by Simulated Annealing and our agent on the n100 netlist of GCRS Benchmark in Fig. 4. The dead space of floorplan by our agent is only 7.95% compared to 8.88% dead space by Simulated Annealing.

3) Runtime Profiling: Where is the runtime spent? Instead of roughly showing a total runtime, we profiled both our agent and the simulated annealing algorithm to make better comparisons. Profiling results are in Fig. 5. According to the profiling, more than half of the runtime (63.1% and 52.3%, respectively) are spent on wirelength calculation. Packing is the second largest consumer (26.1% and 23.0%), and in other words solution evaluation takes most (89.2% and 75.3%, respectively) of the runtime. Random sampling only takes a little portion of time (2.1% and 1.2%).

TABLE IV Area and Wirelength Minimization on MCNC Benchmark. Better results are emphasized in bold.

Circuit	Statistics		Area ($\times 10^6$)		Wirelength ($\times 10^5$)		Cost ($\times 10^6$)		Runtime (s)	
	module #	net #	Ours	SA	Ours	SA	Ours	SA	Ours	SA
apte	9	97	**47.08**	47.31	4.03	**3.43**	**28.41**	28.53	**15.9**	38.1
xerox	10	203	**20.42**	20.64	**6.33**	6.62	**12.51**	12.65	**17.2**	98.8
hp	11	83	**9.21**	9.40	**1.95**	2.62	**5.60**	5.74	**11.6**	44.3
ami33	33	123	**1.24**	1.25	0.69	**0.46**	**0.77**	**0.77**	**43.1**	82.2
ami49	49	408	**38.65**	39.47	**17.24**	12.31	**23.88**	24.18	**66.8**	165.0

TABLE V Area and Wirelength Minimization on GCRS Benchmark. Better results are emphasized in bold.

Circuit	Statistics		Area ($\times 10^5$)		Wirelength ($\times 10^5$)		Cost ($\times 10^5$)		Runtime (s)	
	module #	net #	Ours	SA	Ours	SA	Ours	SA	Ours	SA
n100	100	576	**1.95**	1.97	1.55	**1.54**	**1.79**	1.80	**389.4**	396.2
n200	200	1274	2.15	**2.01**	3.48	**3.34**	2.68	**2.54**	**784.9**	1101.9
n300	300	1632	3.40	**3.29**	**5.25**	5.44	**4.14**	4.15	3766.9	**2062.3**

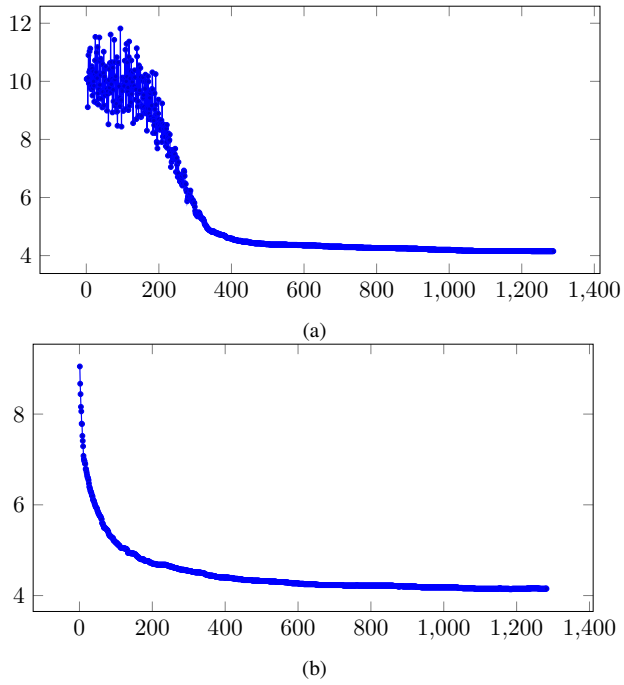


Fig. 3 Search progress visualizations on the n300 netlist of GCRS benchmark: (a) Simulated Annealing; (b) Our Agent. Our agent searches in a smoother way, indicating a more greedy and deterministic heuristic.

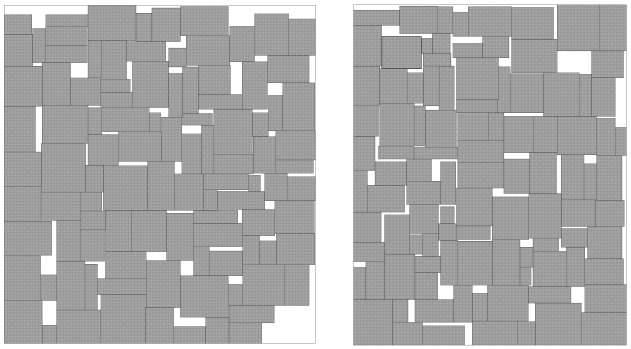


Fig. 4 Floorplan visualizations of the n100 netlist of GCRS benchmark: (a) n100 floorplan by simulated annealing; (b) n100 floorplan by our agent. The dead space of the floorplan by our agent is lower (7.95%) compared to that (8.88%) by Simulated Annealing.

VI. Conclusion

In this work, we investigated the possibility of leveraging reinforcement learning to acquire a floorplanner. The key motivation of our work is to 'learn' new algorithms for difficult combinatorial problems without human expert knowledge. Specifically, we explored the possibility of acquiring local search heuristics through numerous search experiments. To illustrate the applicability, an agent has been trained to perform a walk in the search space by selecting a candidate neighbor solution at each step. We trained the agent using a novel deep Q-learning algorithm with action sampling, and the experimental results have demonstrated the effectiveness of our proposed methods. This work represents the first systematic attempt on leveraging the idea of reinforcement learning to floorplanning. We expect more research along this line due to the importance of floorplanning/placement amidst the physical design flow.

Acknowledgment

The authors would like to thank Ching-Yun Irene Ko for the useful discussion.

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[2] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 3389–3396.

[3] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

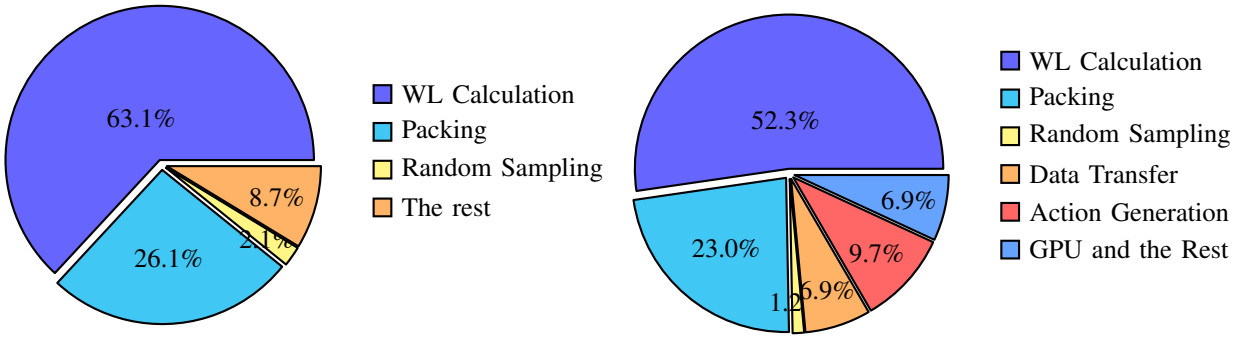


Fig. 5 Runtime profiling of the Simulated Annealing algorithm (left) and our agent (right). Most of the runtime (89.2% and 75.3%, respectively) is spent on solution evaluation (wirelength calculation and packing).

[4] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 6351–6361.

[5] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!" in *International Conference on Learning Representations*, 2019.

[6] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Advances in Neural Information Processing Systems*, 2019, pp. 6278–6289.

[7] E. Yolcu and B. Poczos, "Learning local search heuristics for boolean satisfiability," in *Advances in Neural Information Processing Systems*, 2019, pp. 7990–8001.

[8] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[9] H. Liao, W. Zhang, X. Dong, B. Poczos, K. Shimada, and L. Burak Kara, "A deep reinforcement learning approach for global routing," *Journal of Mechanical Design*, vol. 142, no. 6, 2020.

[10] H. Liao, Q. Dong, X. Dong, W. Zhang, W. Zhang, W. Qi, E. Fallon, and L. B. Kara, "Attention routing: track-assignment detailed routing using attention-based reinforcement learning," *arXiv preprint arXiv:2004.09473*, 2020.

[11] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "High-performance, energy-efficient, fault-tolerant network-on-chip design using reinforcement learnin," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1166–1171.

[12] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, ser. DAC '86. Piscataway, NJ, USA: IEEE Press, 1986, pp. 101–107.

[13] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, "An o-tree representation of non-slicing floorplan and its applications," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. IEEE, 1999, pp. 268–273.

[14] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, "B*-trees: a new representation for non-slicing floorplans," in *Proceedings of the 37th Annual Design Automation Conference*. ACM, 2000, pp. 458–463.

[15] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Rectangle-packing-based module placement," in *The Best of ICCAD*. Springer, 2003, pp. 535–548.

[16] E. F. Young, C. C. Chu, and Z. C. Shen, "Twin binary sequences: a nonredundant representation for general nonslicing floorplan," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 4, pp. 457–469, 2003.

[17] J. Cong, M. Romesis, and J. R. Shinnerl, "Fast floorplanning by look-ahead enabled recursive bipartitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1719–1732, 2006.

[18] C.-T. Lin, D.-S. Chen, and Y.-W. Wang, "An efficient genetic algorithm for slicing floorplan area optimization," in *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No. 02CH37353)*, vol. 2. IEEE, 2002, pp. II–II.

[19] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on fast simulated annealing," in *Proceedings of the 2005 international symposium on Physical design*. ACM, 2005, pp. 104–112.

[20] J. A. Boyan and A. W. Moore, "Learning evaluation functions for global optimization and boolean satisfiability," in *AAAI/IAAI*, 1998, pp. 3–10.

[21] Y. Zhou, J.-K. Hao, and B. Duval, "Reinforcement learning based local search for grouping problems: A case study on graph coloring," *Expert Systems with Applications*, vol. 64, pp. 412–422, 2016.

[22] U. Benlic, M. G. Epitropakis, and E. K. Burke, "A hybrid breakout local search and reinforcement learning approach to the vertex separator problem," *European Journal of Operational Research*, vol. 261, no. 3, pp. 803–818, 2017.

[23] D. Beloborodov, A. Ulanov, J. N. Foerster, S. Whiteson, and A. Lvovsky, "Reinforcement learning enhanced quantum-inspired algorithm for combinatorial optimization," *arXiv preprint arXiv:2002.04676*, 2020.

[24] A. Nareyek, "Choosing search heuristics by non-stationary reinforcement learning," in *Metaheuristics: Computer decision-making*. Springer, 2003, pp. 523–544.

[25] X. Tang and D. Wong, "Fast-sp: a fast algorithm for block placement based on sequence pair," in *Proceedings of the 2001 Asia and South Pacific design automation conference*, 2001, pp. 521–526.

[26] E. Aarts, E. H. Aarts, and J. K. Lenstra, *Local search in combinatorial optimization*. Princeton University Press, 2003.

[27] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[28] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[29] J. Pazis and R. Parr, "Generalized value functions for large action sets," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1185–1192.

[30] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," *arXiv preprint arXiv:1512.07679*, 2015.

[31] T. Zahavy, M. Haroush, N. Merlis, D. J. Mankowitz, and S. Mannor, "Learn what not to learn: Action elimination with deep reinforcement learning," in *Advances in Neural Information Processing Systems*, 2018, pp. 3562–3573.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[33] T. Jaakkola, M. I. Jordan, and S. P. Singh, "On the convergence of stochastic iterative dynamic programming algorithms," *Neural Computation*, vol. 6, no. 6, pp. 1185–1201, 1994.

[34] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[36] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.

[37] A. Caldwell, A. Kahng, and I. Markov. Gsrc bookshelf for vlsi cad algorithms. [Online]. Available: http://vlsicad.cs.ucla.edu/GSRC/bookshelf