Multi-objective Reinforcement Learning with Adaptive Pareto Reset for Prefix Adder Design

Jialin Song, Rajarshi Roy, Jonathan Raiman, Robert Kirby, Neel Kant, Saad Godil, Bryan Catanzaro
NVIDIA

Abstract

Many hardware design problems require navigating a combinatorial search space to find solutions that balance multiple conflicting objectives, e.g., area and delay. While traditional approaches rely on hand-tuned heuristics to combat the large search space, reinforcement learning (RL) has recently achieved promising results, effectively reducing the need for human expertise. However, the existing RL method has prohibitively high sample complexity requirements. In this paper, we present a novel multi-objective reinforcement learning algorithm for combinatorial optimization and apply it to automating designs for prefix adder circuits, which are fundamental to high-performance digital components. We propose to track the evolving Pareto frontier to adaptively select reset states for an episodic RL agent. Our proposed reset algorithm balances exploiting the best-discovered states so far and exploring nearby states to escape local optima. Through empirical evaluations with a real-world physical synthesis workflow on two different design tasks, we demonstrate that our new algorithm trains agents to expand the Pareto frontier faster compared to other baselines. In particular, our algorithm achieves comparable quality results with only 20% of the samples compared to the scalarized baseline. Additional ablation studies confirm that both exploration and exploitation components work together to accelerate the Pareto frontier expansion.

1 Introduction

Many hardware design problems are combinatorial, such as chip design [1, 2] and verification [3]. They are often NP-hard so people design domain-specific heuristics to exploit problem structures to produce good solutions in a reasonable amount of time. Crafting those heuristics is time-consuming and requires a deep understanding of a particular problem. Automating this process has a significant impact on many applications and has been an increasingly popular area of research.

In this work, we focus on a multi-objective hardware design problem of prefix adder design [4], which is an essential building block for digital circuits. Recently, reinforcement learning (RL) agents achieved better performance than commercial tools [5] in this task. The existing approach circumvents the multi-objective property and applies linear scalarization [6] to reduce it to a scalar version. The scalarization method requires a large number of samples, leading to scalability issues. For example, the method in [5] takes over 8000 CPU days to optimize a single adder. We propose a much more sample-efficient multi-objective RL algorithm by resetting RL agents to promising initial states. Our empirical evaluations show a 5x reduction in sample complexity while matching or outperforming existing baselines.

In summary, our contributions are as follows:

- We propose adaptively resetting new episodes to states derived from the Pareto frontier constructed from an agent's cumulative past experience. Our proposed method consists of two components: 1) identify a particular state on the Pareto frontier for modifications; 2) a

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

state perturbation method to ensure diversity of resulting reset states. Such a reset algorithm tackles the classical exploration-exploitation challenges in the context of initial states.

- We provide a comprehensive study with multiple baseline reset methods on a real-world circuit design problem and demonstrate the efficacy of the proposed method in terms of final Pareto frontier quality and computation savings.
- We provide an ablation study on the two components to show that both contribute positively to the overall improvement.

2 Related Works

Multi-objective Reinforcement Learning. Several papers have studied applying reinforcement learning to solve multi-objective optimization problems [6, 7, 8, 9, 10, 11]. Different approaches include reducing to a scalarized version [6, 8] or tackling the multi-objective problem directly [9]. However, most empirical evaluations are on toy environments such as Deep Sea Treasure and a multi-objective version of Mountain Car [12]. In this work, we study a practical problem in prefix circuit design [5].

Data-driven Algorithm Design. In contrast to hand-crafting decision modules in heuristic algorithms, the principle of data-driven algorithm design seeks to learn them from data. There are two main categories based on interactions with a traditional algorithm [13]. The first one is to replace (parts of) an algorithm with learned components. For example, in the branch-and-bound algorithm for solving integer programs, researchers have studied replacing the branch variable selections [14, 15], node selections [16] and cutting plane generation [17]. Recently, end-to-end models have performed competitively in solving traveling salesman problems [18, 19]. The second category is "algorithm configuration", where one improves an algorithm by tuning its many parameters without explicitly changing its decision modules. For example, in the integer program solver Gurobi, there are many important parameters controlling its solving behavior, such as branching strategies [20]. Different data-driven approaches are applied toward learning to set those parameters to their best values, including genetic algorithms [21] and model-based optimization [22].

3 The Prefix Adder Design Problem

3.1 Markov Decision Process Formulation

We now describe the prefix adder design problem, originally proposed in [5]. Given k inputs, $x_0, x_1, \cdots, x_{k-1}$, their prefix sums are k terms such that $y_i = x_i \circ x_{i-1} \circ \cdots x_0$ for $0 \leq i \leq k-1$, where $\circ$ is a binary associative operator [23]. A prefix adder computes all the prefix sums and is represented as a prefix graph as in Figure 1. There are two competing objectives to optimize in a prefix graph: delay and area. For example, the left adder computes prefix sums sequentially and the right one parallelizes the process by introducing an intermediate node depicted in green, which computes $x_3 \circ x_2$. The benefit of parallelization is that we can finish computing all prefix sums faster, visually shown by a lower depth in the adder, but it requires more area, which appears as an extra node in the prefix graph. We create a multi-objective MDP by defining the state space as all valid prefix graphs and the action space as all modifying actions we can perform on a prefix graph. The actions include removing an existing node and adding a new node. This MDP has a deterministic transition function. For a state action pair (s, a) and its resulting new state s', the vector reward is $[area(s) - area(s'), delay(s) - delay(s')]$, where the areas and delays are computed by physical syntheses with OpenPhySyn [1] [24].

3.2 Evaluation Metrics

We evaluate optimization results with two metrics: hypervolume and linear scalarization scores. In a multi-objective optimization problem with n objectives, we say a solution x Pareto dominates another one x' if $x_i \leq x'_i$ for all $1 \leq i \leq n$. Given a set of solutions, the Pareto frontier consists of those not Pareto dominated by any solution, illustrated by the blue dots in Figure 2a. All the solutions on

[1] `https://github.com/scale-lab/OpenPhySyn`, BSD-3-Clause license

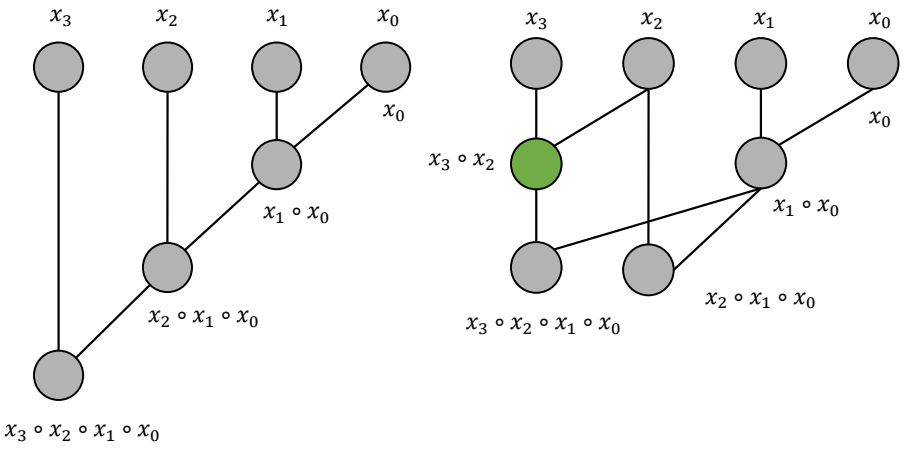
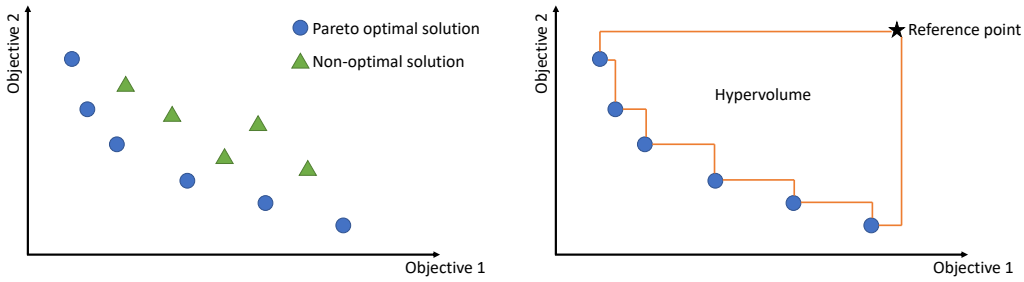


Figure 1: Two 4-bit prefix graph examples. The nodes at the top are inputs. Labels are outputs for each node. With the addition of the green intermediate node on the right, we have a prefix adder with a lower delay (lower depth) for the cost of a larger area (more nodes). For a k-bit adder, the valid state space has size $O(2^{k^2})$, so finding (near) optimal designs is challenging.



(a) (b)

Figure 2: A two-objective example to illustrate the definitions. (a) Every green triangle is Pareto-dominated by some blue circle. The blue circles together comprise a Pareto frontier. (b) Given a reference point, the enclosed region represents all points (within the boundary) Pareto-dominated by some point in the Pareto frontier. The hypervolume is the volume of this region.

the Pareto frontier is Pareto optimal. The hypervolume of a Pareto frontier is the region's volume within the boundary given by a reference point (Figure 2b. In our adder design problem, we have two objectives so the hypervolume is the area between the Pareto frontier and a reference point.

For a delay weight $w \in (0, 1)$, we can compute a preference vector $[w, 1 - w]$ which specifies the desired trade-off between area and delay. The linear scalarization score of a design is $w \cdot \text{delay} + (1 - w) \cdot \text{area}$. We will compare various methods across different delay weights.

4 Adaptive Pareto Reset Algorithm for Multi-objective RL

In this section, we describe our proposed adaptive Pareto reset algorithm (Algorithm 1) for multi-objective RL. Our reset algorithm creates a new reset state in three steps: first, we identify k states in the current Pareto frontier that align best for a preference (line 5-6) (having the lowest k linear scalarization scores). Next, we randomly choose one of the k states (line 7). Then we perturb the chosen state to create a new initial state (line 8). The perturbation function can be problem specific, and it serves to inject diversity into the initial state distribution. This three-step procedure is

Algorithm 1 Adaptive Pareto Reset for Multi-objective RL

Input: Preference set $\mathcal{W}$, an initial policy π, the size of a greedy set k, a perturbation function $perburb$, the number of training episodes N, the time horizon for each episode T

1: Initialize the Pareto optimal set $pareto \leftarrow \emptyset$
2: **for** $episode \leftarrow 1, \cdots, N$ **do**
3: Initialize the objective vector $v_0 \leftarrow \vec{\infty}$ (we aim to minimize each objective)
4: Sample a preference $w \in \mathcal{W}$
5: For each state in $pareto$, compute its linear scalarization score $w^t v$
6: Find the k states with the k smallest scores
7: Randomly choose 1 out of the k states as s
8: Perturb to create the initial state $s_0 \leftarrow perturb(s)$
9: **for** $t \leftarrow 0, \cdots, T$ **do**
10: Select an action $a_t = \pi(s_t, w)$
11: Observe a new state s_{t+1}
12: Receive a vector objective v_{t+1}
13: Reward $r_t \leftarrow v_t - v_{t+1}$
14: **if** s_{t+1} is not Pareto-dominated by all states in $pareto$ **then**
15: Store s_{t+1} with its objective values: $pareto \leftarrow pareto \cup \{(s_{t+1}, v_{t+1})\}$
16: **end if**
17: **end for**
18: **end for**

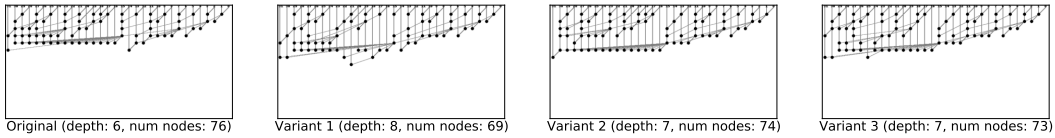


Figure 3: An example perturbation function for 32-bit prefix graphs. The first graph is an original and the following three are sample graphs obtained by randomly removing 10% of nodes from the original. Note that due to a legalization step to ensure a graph is a valid prefix graph, the number of nodes is not exactly 10% fewer. This procedure generates a diverse set of results, both in terms of depth and the number of nodes.

reminiscent of the fundamental exploration-exploitation problem in RL: first, we select a state in a greedy fashion mixed in with some randomization, then we explore its neighboring state space via perturbations. As a necessary step for the greedy selection step, we keep track of an evolving Pareto frontier by adding new states (line 14-15).

In combinatorial optimization problems, the perturbation function generates a neighborhood from a solution as is common in large neighborhood search algorithms [25]. For our prefix adder design problem, the perturbation function accepts a prefix graph and randomly removes a fraction of its nodes. Figure 3 shows an example of randomly removing 10% of nodes in a prefix graph. The goal of the perturbation operation is to generate a diverse set of graphs that share structural features with the original graph. This is confirmed in the examples where the resulting new graphs have different depths and numbers of nodes. The structural similarity is visually present as well.

Perturbation functions exist for other combinatorial optimization problems as well. For example, in TSP, one can modify an existing tour solution by performing k-opt operations [25]. In solving general integer programs, one can mutate an existing solution by changing the values of a subset of variables [26].

5 Experiments on Prefix Adder Design

In this section, we apply our adaptive Pareto reset algorithm to two prefix adder design problems, one for 32-bit and the other for 64-bit. The state space for a k-bit adder is $O(2^{k^2})$ [5], so these two problems pose significant challenges in exploration and thus make great case studies for validating any speed-up effect brought by a new reset method.

In Section 5.1, we provide details on the learning method, the model architecture, and baselines. We present the main results in Section 5.2 with several baselines. In Section 5.3, we provide an ablation study on the two components in the proposed reset algorithm: greedy selection and perturbation. The results suggest both components strengthen the algorithm individually.

5.1 Experiment Setup

Environment Details & Model Architecture. We adopt the double DQN model [27] to represent the RL policy. Following the model architecture by [28], our model takes a preference vector as part of the input. For the 32-bit design task, the preference set $\mathcal{W}$ has 12 elements: $\{(w, 1-w)\}$ for 12 values of w in the range between 0.05 and 0.95. Here, w is the delay weight specifying how much we wish to balance delay and area. For the 64-bit design task, we use a preference set of 19 delay weights. To ensure ample coverage for each preference, we divide each preference set into four groups and train a separate agent for each group. For each transition (s, a, s') and a delay weight, we compute Bellman update targets as follows, where θ_t and θ'_t parametrize the online and target networks respectively in double DQN [27].

$$\mathcal{T}Q(s, a, w) = r(s, a, w) + \gamma Q(s', \arg\max_{a' \in \mathcal{A}} w^T Q(s', a', w; \theta_t); \theta'_t) \tag{1}$$

We follow the feature representation in [5] to represent an N-bit prefix graph with an $N \times N$ image. We incorporate the preference as part of the input by first embedding the preference vector into an image of the same shape, and then adding it to the prefix graph image, as shown in Figure 5. Our DQN architecture combines convolutional layers and residual blocks similar to the architecture used in AlphaZero [29]. There are multiple output heads corresponding to estimated Q values for delay and area by adding or deleting nodes.

We follow the reset algorithm in Algorithm 1 with $k = 5$ and the perturbation function of pruning 10% of nodes in a prefix graph. We name this instantiation **GREP** (**GR**eedy-**E**xplore-**P**rune).

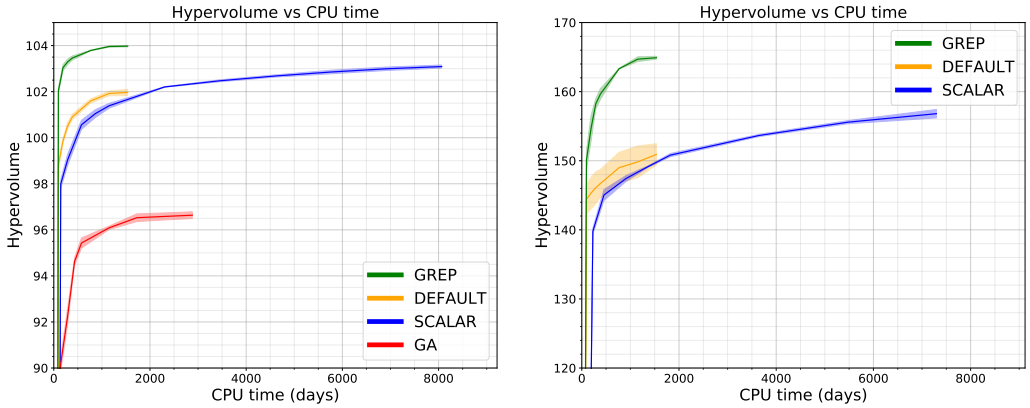
Baselines. We compare with 5 baseline methods. The first one (**SCALAR**) is the scalarized DQN algorithm in [5], where the multi-objective RL is reduced to a single-objective version by training a separate agent for each preference vector. The following three are multi-objective RL baselines. They differ in their episode reset methods. **DEFAULT** chooses to reset an episode to either a ripple-carry or a Sklansky graph [30], which are prefix graphs with the minimal number of nodes and the minimal depth, respectively. **RANDOM** randomly chooses a prefix graph from the current Pareto frontier as the new reset state. **GREEDY** chooses the prefix graph with the minimal weighted score for the preference vector of the new episode (line 4 in Algorithm 1). Finally, we have a non-RL baseline with **GA** (genetic algorithm) as a classical baseline popular in combinatorial optimizations for the 32-bit environment. Each GA run optimizes for a single preference vector. All of the RL-based methods ran for 4 days except for 32-bit SCALAR runs, which ran for 7 days. Each run used 4 NVIDIA A100 GPUs for double DQN training and 96 CPUs for physical synthesis. Due to the high computational resource demand, we repeat the 32-bit experiment with 3 random seeds and the 64-bit experiment with 2 random seeds. GA runs lasted 5 days and each uses 48 CPUs.

5.2 Main Results

Compare with Non-adaptive Reset Baselines and GA. We first compare GREP with SCALAR and DEFAULT, which are two methods with non-adaptive reset schemes. Figure 4 shows the hypervolume growth as the total CPU time grows. Because of the reduction approach SCALAR takes, it requires a lot more CPU time compared to GREP

Method	32 bit		64 bit	
	Hypervolume	# Min	Hypervolume	# Min
GA	96.64 ± 0.15	0	–	–
SCALAR	103.01 ± 0.08	6	156.80 ± 0.74	8
DEFAULT	101.85 ± 0.13	1	150.92 ± 1.18	2
GREP	$\mathbf{103.96 \pm 0.05}$	**8**	$\mathbf{164.91 \pm 0.19}$	**16**

Table 1: Compared to non-adaptive reset schemes (SCALAR, DEFAULT), GREP achieves higher hypervolumes at convergence and has a higher number of minimal linear scalarization scores. GA is significantly worse when compared to RL-based methods.



(a) Hypervolume progress for the 32-bit environment. (b) Hypervolume progress for the 64-bit environment.

Figure 4: In both environments, GREP increases hypervolume at a faster path than other methods and achieves the highest value at convergence.

and DEFAULT. GREP matches the hypervolume of SCALAR at convergence with only 2.4% and 3.9% of the CPU time for the 32-bit and 64-bit settings, respectively. Comparing GREP with DEFAULT at convergence, GREP achieves a 2.1% and 9.3% increase in hypervolume (Table 1). Given they have the same CPU compute resources, such an improvement demonstrates the efficacy of our proposed reset algorithm in improving the Pareto frontier. Moreover, GREP also performs competitively across the entire preference space. In the 32-bit environment, GA plateaus at a much lower hypervolume. This supports the conjecture that the prefix graph search space is hard for a stochastic search approach to navigate and that we need RL to explore more efficiently.

Compare with Adaptive Reset Baselines. We compare with two baselines utilizing adaptive reset, RANDOM and GREEDY. Table 2 shows the hypervolume and the number of preferences with minimal linear scores for the three methods. We provide the full table with all linear scalarization scores in the appendix. GREP outperforms other methods in both metrics. RANDOM proves to be relatively strong in the 32-bit environment but is clearly much worse in the 64-bit environment. RANDOM and GREEDY represent two extremes in the exploration-exploitation trade-offs. GREP strikes a balance and achieves better performance overall. An interesting observation is that in the 32-bit environment, RANDOM is particularly strong for preferences where one values delay more (see the full table in the appendix), which suggests exploration is more beneficial in this region of the Pareto frontier. We leave the question of adapting reset schemes according to preference vectors as future work.

Method	32 bit		64 bit	
	Hypervolume	# Min	Hypervolume	# Min
RANDOM	103.80 ± 0.05	7	160.56 ± 0.37	0
GREEDY	103.14 ± 0.17	2	161.52 ± 0.45	2
GREP	$\mathbf{103.96 \pm 0.05}$	**9**	$\mathbf{164.91 \pm 0.19}$	**17**

Table 2: GREP compares favorably with other adaptive reset baselines in the 32-bit environment and is significantly better in the 64-bit environment.

5.3 Ablation Studies

In this section, we provide ablation studies on two components in the GREP reset algorithm. We first remove the exploration step (line 7 in Algorithm 1) and replace it with the greedy selection, i.e., we always pick the prefix graph with the minimal weighted score to perform the pruning. We label this variant as **NO-EXPLORE**. The second variant, named **NO-PRUNE**, removes the pruning step (line 8 in Algorithm 1). Both components aim to increase the diversity of reset states. As shown in Table 3, removing either component results in a statistically significant reduction in hypervolume and worse overall linear scalarization scores.

6 Conclusion

This work introduces a new reset algorithm for multi-objective RL which balances the exploration-exploitation trade-offs in initializing new episodes. By modifying states from the evolving Pareto frontier, our algorithm places an agent in states that are simultaneously promising, based on their similarities to known good states, and diverse, thanks to the exploration and perturbation applied. We validate our approach with a challenging real-world prefix circuit design problem and our empirical results demonstrate that the proposed algorithm enables significant improvements in sample efficiency.

Method	32 bit		64 bit	
	Hypervolume	# Min	Hypervolume	# Min
NO-EXPLORE	103.66 ± 0.08	7	163.17 ± 0.25	5
NO-PRUNE	103.33 ± 0.14	4	161.61 ± 2.54	5
GREP	$\mathbf{103.96 \pm 0.05}$	**10**	$\mathbf{164.91 \pm 0.19}$	**15**

Table 3: Removing either the EXPLORE or the PRUNE step results in lower hypervolumes at convergence and worse linear scalarization scores.

References

[1] Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen. Combinatorial optimization in vlsi design. In *Combinatorial Optimization*, pages 33–96. IOS Press, 2011.

[2] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[3] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[4] Ming-Bo Lin. *Introduction to VLSI systems: a logic, circuit, and system perspective*. CRC press, 2011.

[5] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 853–858. IEEE, 2021.

[6] Kristof Van Moffaert, Madalina M Drugan, and Ann Nowé. Scalarized multi-objective reinforcement learning: Novel design techniques. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 191–199. IEEE, 2013.

[7] Kristof Van Moffaert and Ann Nowé. Multi-objective reinforcement learning using sets of pareto dominating policies. *The Journal of Machine Learning Research*, 15(1):3483–3512, 2014.

[8] Hossam Mossalam, Yannis M Assael, Diederik M Roijers, and Shimon Whiteson. Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707*, 2016.

[9] Runzhe Yang, Xingyuan Sun, and Karthik Narasimhan. A generalized algorithm for multi-objective reinforcement learning and policy adaptation. *Advances in Neural Information Processing Systems*, 32, 2019.

[10] Thanh Thi Nguyen, Ngoc Duy Nguyen, Peter Vamplew, Saeid Nahavandi, Richard Dazeley, and Chee Peng Lim. A multi-objective deep reinforcement learning framework. *Engineering Applications of Artificial Intelligence*, 96:103915, 2020.

[11] Xi Lin, Zhiyuan Yang, and Qingfu Zhang. Pareto set learning for neural multi-objective combinatorial optimization. In *International Conference on Learning Representations*, 2022.

[12] Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning*, 84(1):51–80, 2011.

[13] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[14] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[15] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In *International conference on machine learning*, pages 344–353. PMLR, 2018.

[16] Jialin Song, Ravi Lanka, Albert Zhao, Aadyot Bhatnagar, Yisong Yue, and Masahiro Ono. Learning to search via retrospective imitation. *arXiv preprint arXiv:1804.00846*, 2018.

[17] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2020.

[18] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2018.

[19] Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, pages 1–29, 2022.

[20] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[21] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. Model-based genetic algorithms for algorithm configuration. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[22] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[23] Guy E Blelloch. Pre x sums and their applications. Technical report, Citeseer, 1990.

[24] Ahmed Agiza and Sherief Reda. Openphysyn: An open-source physical synthesis optimization toolkit. In *2020 Workshop on Open-Source EDA Technology (WOSET)*, 2020.

[25] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 399–419. Springer, 2010.

[26] Edward Rothberg. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4):534–541, 2007.

[27] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[28] Axel Abels, Diederik Roijers, Tom Lenaerts, Ann Nowé, and Denis Steckelmacher. Dynamic weights in multi-objective deep reinforcement learning. In *International Conference on Machine Learning*, pages 11–20. PMLR, 2019.

[29] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[30] Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic computers*, (2):226–231, 1960.

[31] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z Pan. Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1517–1530, 2014.

[32] L Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[33] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

A Appendix

A.1 Prefix Adder Basics

Parallel prefix computations can be represented as a directed acyclic prefix graph where every computation unit $z_{i:j}$ is a graph node that performs a single operation on a pair of inputs: $z_{i:j} = z_{i:k} \circ z_{k-1:j}$ where $i \geq k > j$. We use the notation from [31] where the most and least significant bits (MSB, LSB) of computation node $z_{i:j}$ is (i, j). Using this notation we will term the node (i, k) as the *upper parent* of (i, j) and the node $(k - 1, j)$ as its *lower parent*.

Every legal N-input prefix graph must have input nodes (i, i), output nodes $(i, 0)$ for $1 \leq i \leq N-1$, and the input/output node $(0, 0)$. Furthermore, every non-input node must have exactly one upper parent (up) and one lower parent (lp) such that:

$$\begin{aligned} LSB(node) &= LSB(lp(node)) \\ LSB(lp(node)) &\leq MSB(lp(node)) \\ MSB(lp(node)) &= LSB(up(node)) - 1 \\ LSB(up(node)) &\leq MSB(up(node)) \\ MSB(up(node)) &= MSB(node) \end{aligned} \tag{2}$$

A.2 Reinforcement Learning Environment

The PrefixRL state space $\mathcal{S}$ consists of all legal N-input prefix graphs. N-input graphs can be represented in a $N \times N$ grid with rows representing MSB and columns representing LSB. Note that the input nodes ($MSB = LSB$) will lie on the diagonal, output nodes will lie on the first column ($LSB = 0$) and locations above the diagonal ($LSB > MSB$) cannot contain a node. The remaining $(N-1)(N-2)/2$ locations where non-input/output nodes may or may not exist define the $\mathcal{O}(2^{(N-1)(N-2)/2}) = \mathcal{O}(2^{N^2})$ state space of N-input prefix graphs. For example, 32-input graphs will have $|\mathcal{S}| = \mathcal{O}(2^{465})$ with a lower exact value because not all combinations of nodes in those locations will meet the legality constraints in (2).

The action space $\mathcal{A}$ for an N-input prefix graph consists of two actions (add or delete) for any non-input/output node i.e. where $LSB \in [1, N-2]$ and $MSB \in [LSB+1, N-1]$. Hence, $|\mathcal{A}| = (N-1)(N-2)/2$. The environment evolution through $\mathcal{T}$ always maintains a legal prefix graph by:

1. Applying a legalization procedure after an action that may add or delete additional nodes to maintain legality.
2. Forbidding redundant actions that gets undone by the legalization procedure.

During legalization, the upper parent of a node, $up(node)$, is the existing node with same MSB and the next highest LSB. The lower parent of a node is computed using the node and its upper parent (2):

$$(MSB_{lp(node)}, LSB_{lp(node)}) = (LSB_{up(node)} - 1, LSB_{node})$$

An illegal condition happens only when the lower parent $lp(node)$ of a node does not exist, so the legalization procedure adds any missing lower parent nodes.

The action of adding a node that already exists (in $nodelist$) is redundant and is forbidden. Deleting is limited to nodes in $minlist$ (nodes that are not lower parents of other nodes) to prevent legalization from adding back deleted nodes.

A.3 Genetic Algorithm Details

For our genetic algorithm baseline we use a standard genetic algorithm with crossover and mutation [32]. Similar to the SCALAR baseline we run multiple single-objective genetic algorithm runs and generate the Pareto frontier from the union of all explored prefix adders. We use a population size of 1000. The individuals of the population are the flattened representations of the prefix graph with a Boolean value representing whether or not that prefix node is present. A single mutation is represented by a random inversion of a single Boolean value. Each individual is initialized by randomly choosing either the ripple-carry or Sklansky prefix structure and performing 200 random mutations. Each generation we assign a fitness score to each individual as the weighted score of delay and area. Each successive generation is generated by taking the top 50% most fit individuals from the previous generation and from them generating a new population. The new population is generated 40% by a mutation procedure, 40% by a crossover procedure, 10% by preserving the top individuals from the previous generation unchanged and 10% by the random initialization procedure mentioned above. The mutation procedure creates a new individual by randomly sampling a single parent and then performing up to 50 random mutations. The crossover procedure creates a new individual by randomly sampling two parents and then randomly choosing one of those two parents to supply the value at each node location.

A.4 Double DQN Training Details

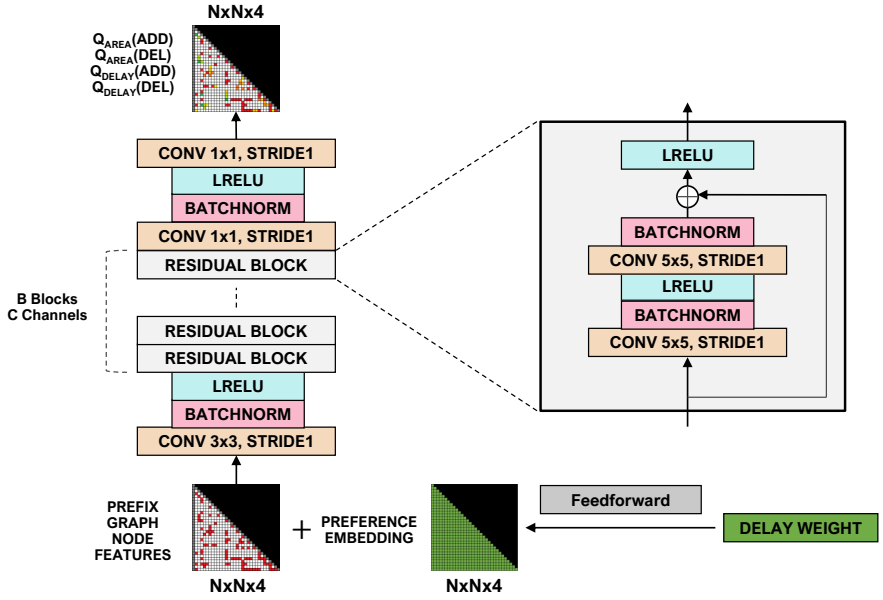


Figure 5: The DQN architecture in the double DQN model. We use $B = 32$ and $C = 256$.

Based on the $N \times N$ grid based representation of prefix graphs described in Section A.2, the input to the neural network is a $N \times N \times 4$ tensor where the 4 channels encode node features as:

1. 1 if node (MSB, LSB) in $nodelist$, 0 otherwise
2. 1 if node (MSB, LSB) in $minlist$, 0 otherwise
3. level of node (MSB, LSB) in $nodelist$, 0 otherwise
4. fanout of node (MSB, LSB) in $nodelist$, 0 otherwise

where the fanout of a node refers to the number of children it has and the level of a node refers to its topological depth from input nodes in the prefix graph. Features are normalized to $[0, 1]$.

For both 32-bit and 64-bit environments, we use the architecture in Figure 5 with 32 residual blocks and 256 channels. The preference embedding network consists of 2 feedforward layers, each with 64 hidden units and the ReLU activation function.

For the double DQN training, we use a learning rate of 4×10^{-5}. The batch size is 192 for 32-bit and 18 for 64-bit. The target network is synchronized with the online network after every 60 updates.

During policy rollout, we use ϵ-greedy to explore. Our ϵ value is randomized similar to the scheme in [33]. Specifically, we first sample a random number i between 0 and 1023, then the ϵ value is calculated as $0.4^{(1+7i/1023)}$ for a new episode.

A.5 Full Linear Scalarization Scores

For each delay weight w, the linear scalarization score is $w * delay + (1 - w) * area$, where the delay and area are normalized from their raw values. The delay normalization scalar is 0.1 and the area normalization scalar is 100.

A.5.1 Linear Scalarzation Scores for the 32-bit Environment

Delay Weight	GA	SCALAR	DEFAULT	GREP
0.05	2.105 +/- 0.0	2.092 +/- 0.0	2.101 +/- 0.001	**2.075 +/- 0.001**
0.08	2.194 +/- 0.007	2.191 +/- 0.002	2.206 +/- 0.002	**2.163 +/- 0.001**
0.1	2.239 +/- 0.007	2.235 +/- 0.002	2.256 +/- 0.003	**2.216 +/- 0.003**
0.2	2.417 +/- 0.006	2.412 +/- 0.003	2.442 +/- 0.006	**2.388 +/- 0.003**
0.3	2.575 +/- 0.009	2.562 +/- 0.004	2.593 +/- 0.006	**2.536 +/- 0.004**
0.4	2.717 +/- 0.01	2.708 +/- 0.004	2.724 +/- 0.004	**2.668 +/- 0.001**
0.5	2.846 +/- 0.009	**2.775 +/- 0.0**	2.781 +/- 0.002	2.786 +/- 0.003
0.6	2.967 +/- 0.007	**2.819 +/- 0.0**	2.826 +/- 0.004	2.826 +/- 0.004
0.7	3.078 +/- 0.006	**2.86 +/- 0.001**	**2.865 +/- 0.003**	**2.86 +/- 0.004**
0.8	3.168 +/- 0.012	**2.885 +/- 0.0**	2.889 +/- 0.003	**2.884 +/- 0.002**
0.9	3.191 +/- 0.013	**2.878 +/- 0.004**	2.889 +/- 0.003	2.884 +/- 0.002
0.95	3.169 +/- 0.016	**2.816 +/- 0.003**	2.822 +/- 0.001	2.844 +/- 0.006
# Min	0	6	1	8

Table 4: Linear scalarization scores for 32-bit non-adaptive reset baselines.

Delay Weight	RANDOM	GREEDY	GREP
0.05	**2.074 +/- 0.0**	2.078 +/- 0.001	2.076 +/- 0.001
0.08	2.168 +/- 0.005	2.168 +/- 0.004	**2.162 +/- 0.001**
0.1	2.22 +/- 0.005	2.218 +/- 0.004	**2.212 +/- 0.001**
0.2	2.399 +/- 0.002	2.4 +/- 0.004	**2.391 +/- 0.003**
0.3	2.557 +/- 0.005	2.559 +/- 0.008	**2.541 +/- 0.0**
0.4	2.702 +/- 0.01	2.714 +/- 0.008	**2.669 +/- 0.0**
0.5	**2.786 +/- 0.0**	2.796 +/- 0.015	**2.788 +/- 0.003**
0.6	**2.831 +/- 0.005**	**2.837 +/- 0.013**	**2.83 +/- 0.004**
0.7	**2.862 +/- 0.0**	2.867 +/- 0.003	**2.864 +/- 0.004**
0.8	**2.887 +/- 0.006**	**2.885 +/- 0.001**	**2.885 +/- 0.002**
0.9	**2.881 +/- 0.001**	2.886 +/- 0.0	2.886 +/- 0.002
0.95	**2.82 +/- 0.001**	2.832 +/- 0.006	2.846 +/- 0.01
# Min	7	2	9

Table 5: Linear Scalarization scores for 32-bit adaptive reset baselines.

Delay Weight	NO-EXPLORE	NO-PRUNE	GREP
0.05	2.077 +/- 0.001	2.082 +/- 0.002	**2.075 +/- 0.001**
0.08	2.166 +/- 0.001	2.167 +/- 0.001	**2.163 +/- 0.001**
0.1	**2.217 +/- 0.001**	**2.216 +/- 0.004**	**2.216 +/- 0.003**
0.2	**2.39 +/- 0.005**	2.4 +/- 0.007	**2.388 +/- 0.003**
0.3	**2.539 +/- 0.008**	2.557 +/- 0.01	**2.536 +/- 0.004**
0.4	2.677 +/- 0.006	2.706 +/- 0.011	**2.668 +/- 0.001**
0.5	**2.785 +/- 0.005**	2.795 +/- 0.009	**2.786 +/- 0.002**
0.6	**2.829 +/- 0.006**	2.846 +/- 0.013	**2.826 +/- 0.004**
0.7	**2.863 +/- 0.002**	2.87 +/- 0.006	**2.86 +/- 0.004**
0.8	**2.885 +/- 0.001**	**2.883 +/- 0.002**	**2.884 +/- 0.002**
0.9	2.894 +/- 0.002	**2.876 +/- 0.004**	2.888 +/- 0.002
0.95	2.839 +/- 0.005	**2.827 +/- 0.005**	2.845 +/- 0.006
# Min	7	4	10

Table 6: Linear Scalarization scores for 32-bit ablation studies.

A.5.2 Linear Scalarzation Scores for the 64-bit Environment

Delay Weight	SCALAR	DEFAULT	GREP
0.05	4.244 +/- 0.012	4.276 +/- 0.0	**4.201 +/- 0.004**
0.1	4.535 +/- 0.018	4.571 +/- 0.006	**4.341 +/- 0.011**
0.15	4.545 +/- 0.006	4.597 +/- 0.004	**4.368 +/- 0.0**
0.2	4.55 +/- 0.002	4.594 +/- 0.003	**4.369 +/- 0.004**
0.25	4.54 +/- 0.001	4.587 +/- 0.007	**4.364 +/- 0.002**
0.3	4.51 +/- 0.0	4.568 +/- 0.02	**4.358 +/- 0.002**
0.35	4.471 +/- 0.004	4.547 +/- 0.034	**4.352 +/- 0.001**
0.4	4.426 +/- 0.005	4.527 +/- 0.049	**4.336 +/- 0.001**
0.45	4.379 +/- 0.005	4.502 +/- 0.06	**4.307 +/- 0.007**
0.5	4.333 +/- 0.005	4.46 +/- 0.076	**4.277 +/- 0.013**
0.55	4.276 +/- 0.001	4.395 +/- 0.077	**4.248 +/- 0.019**
0.6	**4.217 +/- 0.002**	4.331 +/- 0.078	**4.218 +/- 0.025**
0.65	**4.156 +/- 0.004**	4.266 +/- 0.08	**4.169 +/- 0.035**
0.7	**4.09 +/- 0.003**	4.196 +/- 0.077	**4.108 +/- 0.036**
0.75	**4.024 +/- 0.001**	4.118 +/- 0.068	**4.045 +/- 0.037**
0.8	**3.958 +/- 0.0**	4.036 +/- 0.056	3.971 +/- 0.03
0.85	**3.879 +/- 0.006**	3.916 +/- 0.04	**3.883 +/- 0.012**
0.9	**3.76 +/- 0.007**	**3.771 +/- 0.022**	3.794 +/- 0.006
0.95	**3.622 +/- 0.006**	**3.626 +/- 0.003**	3.67 +/- 0.001
# Min	8	2	16

Table 7: Linear scalarization scores for 64-bit non-adaptive reset baselines.

Delay Weight	RANDOM	GREEDY	GREP
0.05	4.224 +/- 0.005	4.212 +/- 0.007	**4.201 +/- 0.004**
0.1	4.354 +/- 0.017	4.376 +/- 0.017	**4.341 +/- 0.011**
0.15	4.396 +/- 0.01	4.413 +/- 0.016	**4.368 +/- 0.0**
0.2	4.424 +/- 0.006	4.437 +/- 0.007	**4.369 +/- 0.004**
0.25	4.45 +/- 0.004	4.454 +/- 0.003	**4.364 +/- 0.002**
0.3	4.468 +/- 0.004	4.462 +/- 0.003	**4.358 +/- 0.002**
0.35	4.462 +/- 0.009	4.45 +/- 0.006	**4.352 +/- 0.001**
0.4	4.452 +/- 0.012	4.439 +/- 0.014	**4.336 +/- 0.001**
0.45	4.434 +/- 0.01	4.427 +/- 0.023	**4.307 +/- 0.007**
0.5	4.414 +/- 0.011	4.416 +/- 0.031	**4.277 +/- 0.013**
0.55	4.391 +/- 0.013	4.404 +/- 0.04	**4.248 +/- 0.019**
0.6	4.368 +/- 0.016	4.364 +/- 0.031	**4.218 +/- 0.025**
0.65	4.324 +/- 0.004	4.304 +/- 0.037	**4.169 +/- 0.035**
0.7	4.272 +/- 0.014	4.235 +/- 0.039	**4.108 +/- 0.036**
0.75	4.202 +/- 0.024	4.139 +/- 0.026	**4.045 +/- 0.037**
0.8	4.105 +/- 0.015	4.039 +/- 0.013	**3.971 +/- 0.03**
0.85	3.973 +/- 0.006	3.906 +/- 0.022	**3.883 +/- 0.012**
0.9	3.814 +/- 0.004	**3.748 +/- 0.021**	3.794 +/- 0.006
0.95	3.628 +/- 0.003	**3.582 +/- 0.016**	3.67 +/- 0.001
# Min	0	2	17

Table 8: Linear Scalarization scores for 64-bit adaptive reset baselines.

Delay Weight	NO-EXPLORE	NO-PRUNE	GREP
0.05	**4.197 +/- 0.013**	4.214 +/- 0.002	**4.201 +/- 0.004**
0.1	**4.306 +/- 0.004**	4.358 +/- 0.015	4.341 +/- 0.011
0.15	**4.391 +/- 0.033**	4.445 +/- 0.043	**4.368 +/- 0.0**
0.2	4.426 +/- 0.044	4.487 +/- 0.042	**4.369 +/- 0.004**
0.25	4.438 +/- 0.042	4.524 +/- 0.04	**4.364 +/- 0.002**
0.3	4.438 +/- 0.036	4.536 +/- 0.041	**4.358 +/- 0.002**
0.35	4.434 +/- 0.033	4.518 +/- 0.043	**4.352 +/- 0.001**
0.4	4.424 +/- 0.027	4.496 +/- 0.042	**4.336 +/- 0.001**
0.45	4.4 +/- 0.018	4.472 +/- 0.04	**4.307 +/- 0.007**
0.5	4.376 +/- 0.01	4.446 +/- 0.039	**4.277 +/- 0.013**
0.55	4.353 +/- 0.001	4.405 +/- 0.046	**4.248 +/- 0.019**
0.6	4.317 +/- 0.016	4.359 +/- 0.053	**4.218 +/- 0.025**
0.65	4.272 +/- 0.031	4.298 +/- 0.069	**4.169 +/- 0.035**
0.7	4.192 +/- 0.027	4.22 +/- 0.072	**4.108 +/- 0.036**
0.75	**4.104 +/- 0.017**	**4.112 +/- 0.056**	**4.045 +/- 0.037**
0.8	**4.011 +/- 0.004**	**3.994 +/- 0.037**	**3.971 +/- 0.03**
0.85	3.916 +/- 0.01	**3.866 +/- 0.019**	3.883 +/- 0.012
0.9	3.806 +/- 0.019	**3.729 +/- 0.008**	3.794 +/- 0.006
0.95	3.63 +/- 0.021	**3.588 +/- 0.0**	3.67 +/- 0.001
wins	5	5	15

Table 9: Linear Scalarization scores for 64-bit ablation studies.