Compiler Optimization Using Genetic Algorithms

Alexia Arthur
anarthur@umich.edu
University of Michigan

Murphy Justian
justian@umich.edu
University of Michigan

Rohan Putcha
rputcha@umich.edu
University of Michigan

Shoma Sawa
shomas@umich.edu
University of Michigan

Vishwa Ramanakumar
vishram@umich.edu
University of Michigan

Abstract

Compiler optimization aims to improve the efficiency of compiled code, often through complex sequences of transformations called optimization passes. Traditional heuristic-based approaches are limited by the enormous search space of possible pass combinations. Recent work using CompilerGym has shown that reinforcement learning (RL) can automate this search, but RL may still suffer from local optima and limited exploration. In this paper, we propose a Genetic Algorithm (GA)-assisted approach to compiler optimization that leverages evolutionary principles to explore the vast optimization space more effectively. Our fitness function incorporates runtime, IR instruction count, and Autophase metrics to guide the GA toward producing highly optimized code. We evaluate both RL and GA on diverse benchmarks, demonstrating that while RL is more stable and efficient, GA can discover configurations that significantly outperform RL's best solutions. Although GA exhibits high variance and requires more computational effort, its ability to find the highest-performing sequences makes it a valuable alternative or supplement to RL-driven compiler optimization.

1 Introduction

Compiler optimization plays an essential role in software engineering today, leading to application speed-up and reduced memory usage, among other benefits. Modern compilers such as LLVM are equipped with a variety of optimization passes to aid in this task. However, selecting the ideal sequence of said passes is a challenge, as the search space can be very large. This makes efficiency a key issue.

CompilerGym [1], a library developed by Meta, offers a platform for integrating machine learning techniques, primarily reinforcement learning (RL), to automate the process of selecting compiler optimizations. Despite this innovation, though, it can still encounter issues such as converging on local maxima/minima.

We propose incorporating Genetic Algorithms (GA) to enhance compiler optimization within the existing CompilerGym framework for the GCC flag tuning task. This task involves choosing the optimal settings from a vast array of compiler flags—up to 10^{4461} possible combinations—which directly impact the performance of compiled code. We show that this method produces superior results. While GAs are generally more computationally intensive, they provide a global, population-based search mechanism which is well-suited for exploring extensive, combinatorial search spaces.

We evaluate performance using a fitness function that is based on 3 main metrics: runtime reduction (50%), IR instruction count reduction (30%), and Autophase feature reductions (20%). By balancing these objectives, we aim to direct the GA toward solutions that not only improve speed, but also produce smaller and structurally simpler code.

Our experiments on various CBench and CHStone benchmarks show that while RL methods offer improvements, our GA-based approach consistently achieves better results. This advantage comes at the cost of longer runtimes for the search process. In summary, our GA approach provides a robust, global search capability, identifying sequences of LLVM passes that significantly outperform those discovered by RL.

2 Related Work

2.1 CompilerGym

CompilerGym [1] offers a platform that treats compiler optimization as an RL environment. Moreover, it includes a series of environments based on LLVM. The aim is to simplify the integration of machine learning techniques with compilers.

The initial focus of CompilerGym was on RL algorithms, such as policy gradient methods and Q-learning, to explore the extensive action space of LLVM optimization passes. Despite showing promise, these RL methods often required significant hyperparameter tuning and could sometimes settle into suboptimal strategies. Moreover, RL agents might not fully utilize the combinatorial nature of the optimization pass sequences.

On the other hand, our GA-based approach can explore this space in a more direct manner. Although it might be a slower methodology given that RL uses function approximation and reuse, it provides a broader, population-based search methodology. Our GA approach seeks to overcome some of the limitations of the initial CompilerGym RL methods by applying evolutionary principles to identify better optimization sequences.

2.2 MLGO

MLGO [2], standing for machine learning guided optimization, is the first instance of machine learning integration with the LLVM compiler. MLGO aims to replace traditional heuristic-based optimizations with machine-learned models. Using Policy Gradient and Evolution Strategies, MLGO achieved up to a 7% reduction in code size compared to existing SOTA at the time. While not involving RL, this paper represents a big step in the direction of replacing traditional heuristics using ML. While CompilerGym furthers this work using RL, we believe GA presents a further extension of this goal.

3 Algorithm Implementation

3.1 Environment

The model uses a LLVM CompilerGym environment that creates a challenging environment for artificial intelligence agents. The environment uses the LLVM IR (llvm-v0) as the state space, optimization passes as actions, and a reward function to entice the agent to produce optimized code. The observation spaces represent what is shared with the agent at each step of the optimization process.

3.2 Action Space

The discrete action spaces represent the set of actions that the agent can take at each step of the algorithm optimization process. There are 120 different optimization passes that can be utilized. Example Actions:

- **-simplifycfg (CFG Simplification):** Removes unnecessary paths and merges blocks to simplify program execution.
- **-dce (Dead Code Elimination):** Removes instructions that do not have an effect on the program.
- **-loop-vectorize (Loop Vectorization):** Loops utilize vector instructions to run multiple iterations at one time.

3.3 Observation Spaces

Our algorithms use three of the 34 observation spaces.

- **Runtime:** The program execution after each iteration. The program's runtime serves a good indicator as to whether the algorithm is getting faster.
- **IR Instruction Count:** The static number of instructions residing in the intermediate representation. The observation is able to give insight into the complexity of a program.
- **Autophase Total Instruction Count:** The dynamic number of instructions derived from the intermediate representation. The instruction has the ability to forecast performance trends.

Using these three observations we will be able to get a complete understanding of the program's performance.

3.4 Reinforcement Learning Model

The CompilerGym Actor-Critic serves as a framework for code optimization using reinforcement learning (RL). The RL model consists of two components, Actor and Critic.

- **Actor:** The actor is responsible for selecting a set of actions dependent upon the current state.
- **Critic:** The critic evaluates the merit of the actions chosen by the agent. The quality of the action set is based on its reward value derived form the fitness evaluation. The higher the reward the better the algorithm is performing.

In order to further evaluate the performance of the Actor-Critic model against our genetic algorithm specific modifications had to be made. The bases of the model served to be quite helpful, however, some of the rewards, observations, and parameters required change.

3.4.1 Hyperparameters The model has eight different parameters that impact how the model behaves.

- **Episode Length:** Number of transitions per episode
- **Hidden Size:** Latent vector size
- **Log Interval:** Episodes per log output
- **Episode Count:** Number of episodes
- **Iterations:** Number of time required to train the model
- **Exploration:** Rate to explore random transitions
- **Mean Smoothing:** Smoothing factor for mean normalization
- **Standard Deviation Smoothing:** Smoothing factor for standard deviation normalization

3.4.2 Individual The episode length in this model represents the number of actions also known as flags. The flags are randomly selected and grouped into a set. An example Episode Length of four would consist of four flags such as [-slsr, -aggressive-instcombine, -coro-early, -bdce].

3.4.3 Fitness Evaluation The goal of fitness evaluation is to determine the quality of each Episode Length. The runtime, IR instruction count, and Autophase total instruction count are used to determine each episode length's reward. The higher the reward the better the algorithm performs. Runtime makes up 50% of the reward as it's important that the program executes at a faster rate. IR instruction is weighted at 30%. The goal is to have a lower instruction count signifying improved code efficiency and reduction in clock cycles for task execution. Lastly, the Autophase total instruction count makes up the remaining 20%. A lower count reduces the quantity of instructions that need to be processed by the pipeline. Ultimately, minimizing the chance of pipeline stalls. See Appendix 9.2 for more information on Autophase.

$$R = rt + (ic + ap) * 0.01$$

where:

- rt = Runtime * 0.5
- ir = IR instruction count * 0.3
- ap = Autophase total instruction count * 0.2

3.5 RL Best Sequence

The algorithm will run for a predefined set of iterations. Within the iterations each generation will determine its best Episode Length and its fitness. Said Episode Lengths will then be compared and the action sequence with the highest fitness evaluation will then be reflected at the end of each iteration.

3.6 Genetic Algorithm Model

3.6.1 Overview Figure 3 illustrates the end-to-end pipeline. We start with an input program, run it through LLVM via CompilerGym, and then apply a GA-based approach to propose sequences of optimization passes. The resulting LLVM IR is then compiled into an optimized binary. Our GA parameters, such as population size, generation count, mutation rate, and crossover rate, are used to evolve the population of solutions over several iterations.

3.6.2 Hyperparameters The model has six different parameters that impact how the model behaves.

- **Population Size:** Number of Individuals in the population
- **Generation Count:** Number of generations to evolve per iteration
- **Flag Count:** Number of flags per Individual

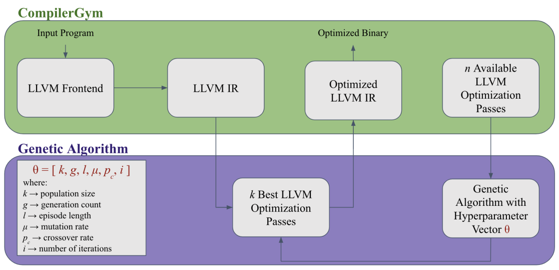

Figure 1: Overview of the Project Pipeline: The input program is passed through LLVM and CompilerGym, forming an environment for the GA agent. The GA evolves a population (of size k) of candidate optimization sequences using selected hyperparameters to output a list of k optimization passes. The optimized LLVM IR is then compiled into the final optimized binary.

- **Mutation Rate:** Probability of mutation
- **Crossover Rate:** Probability of crossover
- **Iterations:** Number of time required to train the model

3.6.3 Individual The Individuals in this model are made up of a group of randomly selected flags. The number of flags per Individual is strictly determined by the . An example Episode Length of four would consist of four flags such as [-slsr, -aggressive-instcombine, -coro-early, -bdce].

3.6.4 Fitness Evaluation The goal of fitness evaluation is to determine the quality of each Individual. The runtime, IR instruction count, and Autophase total instruction count are used to determine each Individual's reward. The higher the reward the better the algorithm performs. Runtime makes up 50% of the reward as it is important that the program executes at a faster rate. IR instruction is weighted at 30%. The goal is to have a lower instruction count signifying improved code efficiency and reduction in clock cycles for task execution. Lastly, the Autophase total instruction count makes up the remaining 20%. A lower count reduces the quantity of instructions that need to be processed by the pipeline. Ultimately, minimizing the chance of pipeline stalls.

$$R = rt + (ic + ap) * 0.01$$

where:

- rt = Runtime * 0.5
- ir = IR instruction count * 0.3
- ap = Autophase total instruction count * 0.2

3.6.5 Mutation With a small probability (μ), we replace an action in the sequence with a random pass. This ensures diversity and helps escape local optima. By randomly altering a pass, we maintain genetic diversity.

3.6.6 Crossover The optimization passes from two different Individuals are combined to create an optimal action sequence. The "offspring" will ultimately receive the best flags from both parents.

3.7 GA Best Sequence

The algorithm will run for a predefined set of iterations. Within the iterations each generation will determine its best Individual and its fitness. Said Individuals will then be compared and the action sequence with the highest fitness evaluation will then be reflected at the end of each iteration.

3.8 GA-Opt

Due to space constraints, we provide our full GA-Opt protocol in Algorithm 1 in the Appendix.

4 Evaluation

In this section, we detail the evaluation of our approach using the CBench and CHStone benchmark suites. Our experimental setup includes the use of RL and GA methods, assessed through multiple runs to ensure robustness and reliability of the results. The primary metric for comparison is the maximum fitness score achieved.

4.1 CBench Benchmarks

The CBench benchmarks selected for this evaluation are:

- **crc32:** A cyclic redundancy check algorithm that computes a 32-bit checksum value from input data.
- **bzip2:** A file compression algorithm utilizing the Burrows-Wheeler transform.
- **jpeg:** A widely-used algorithm for JPEG image compression.

For each CBench benchmark, multiple iterations of both RL and GA approaches were conducted. The RL algorithm was evaluated over episodes, while the GA was assessed over generations. Tables 1 and 2 below summarize the parameters used for the GA and RL algorithms:

GA Parameters	Value
population_size	10
generation_count	15
episode_len	5
mutation_rate	0.1
crossover_rate	0.8
iterations	3

Table 1: GA Algorithm Parameters

RL Parameters	Value
episode_len	5
hidden_size	64
log_interval	4
episodes_count	60
iterations	3
exploration	0.0
mean_smoothing	0.95
std_smoothing	0.4

Table 2: RL Algorithm Parameters

The maximum fitness score achieved during these iterations was recorded for comparative analysis.

4.2 CHStone Benchmarks

The CHStone benchmarks included in our evaluation are:

- **jpeg:** The same JPEG image compression algorithm as used in CBench.
- **blowfish:** A cryptographic algorithm for symmetric key block ciphering.
- **motion:** An algorithm for motion estimation in video processing.
- **gsm:** An algorithm implementing the Global System for Mobile Communications.

Similar to the CBench benchmarks, the CHStone benchmarks were subjected to multiple runs of both RL and GA methods. The evaluation process focused on recording the parameters and performance metrics for each run. Tables 3 and 4 below summarize the parameters used for the GA and RL algorithms for the CHStone benchmarks:

GA Parameters	Value
population_size	10
generation_count	100
episode_len	5
mutation_rate	0.1
crossover_rate	0.8
iterations	24

Table 3: GA Algorithm Parameters for CHStone Benchmarks

RL Parameters	Value
episode_len	5
hidden_size	64
log_interval	4
episodes_count	240
iterations	24
exploration	0.0
mean_smoothing	0.95
std_smoothing	0.4

Table 4: RL Algorithm Parameters for CHStone Benchmarks

The maximum fitness score achieved during these iterations was recorded for comparative analysis.

5 Results

To test the genetic algorithm in comparison to reinforcement learning, we compiled benchmarks from both the cbench and chstone benchmark systems. In cbench, we used the jpeg, crc32, and bzip2 benchmarks. In chstone, we used jpeg, blowfish, motion, and gsm.

As shown in Figure 3, compared to Figure 2, the genetic algorithm is more erratic than the reinforcement learning algorithm, which shows only increases in performance. The variation in the genetic algorithm is caused by some mutations being harmful, which is not

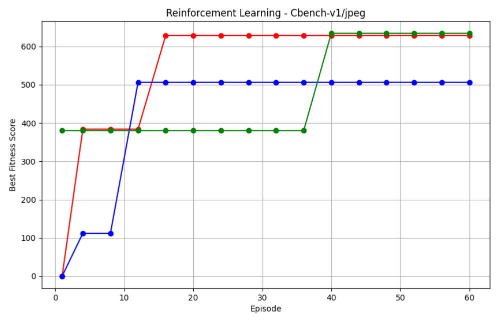

Figure 2: Graph of fitness of the cbench jpeg benchmark after 60 episodes of reinforcement learning.

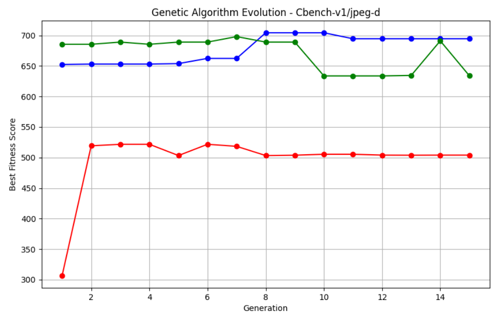

Figure 3: Graph of fitness of the cbench jpeg benchmark after 15 generations of the genetic algorithm.

a factor in reinforcement learning, however, the highest peak of the genetic algorithm is higher than that of the reinforcement learning algorithm. Since programs are usually compiled only once, the main heuristic from which to judge the algorithms is the maximum fitness overall, rather than an average or a trend.

As shown in Table 5, the maximum fitness produced after 6 iterations with either 15 generations for the genetic algorithm or 60 episodes for reinforcement learning yields a higher maximum fitness score for the genetic algorithm on all our benchmarks. Additionally, as shown in Table 4, all the chstone benchmarks also show higher maximum fitness for the genetic algorithm as opposed to reinforcement learning. Based on these results, even though the genetic algorithm is less regular in its performance increases, the maximum value is always greater than that from reinforcement learning. Since an end user will always compile with the single most efficient set of compiler flags, the genetic algorithm outperforms reinforcement learning across all our benchmarks.

6 Conclusion

We introduced a genetic algorithm-assisted approach for compiler optimization using CompilerGym. Our GA approach systematically explores the large space of LLVM optimization pass sequences, guided by a fitness function that considers runtime, instruction count and autophase metrics.

CBench Benchmarks		
Benchmark	**RL Max Fitness**	**GA Max Fitness**
jpeg (cbench)	634.13	634.13
crc32 (cbench)	2.99	3.06
bzip2 (cbench)	245.22	273.52

Table 5: Comparison of Maximum Fitness Scores for CBench Benchmarks

CHStone Benchmarks		
Benchmark	**RL Max Fitness**	**GA Max Fitness**
jpeg (chstone)	38.48	39.93
blowfish (chstone)	22.79	23.27
motion (chstone)	11.41	12.05
gsm (chstone)	19.94	21.81

Table 6: Comparison of Maximum Fitness Scores for CHStone Benchmarks

Although the original CompilerGym RL-based approach demonstrated the effectiveness of RL in this domain, our GA-based approach offers a compelling alternative. While the RL excels in stability and efficiency, GA can discover superior optimization sequences. This trade-off between efficiency and solution quality is crucial for practical applications. If computational resources are abundant, GA-based optimization can uncover better performing solutions than RL. Our findings highlights the potential of evolutionary algorithms in compiler optimization.

7 Future Work

While our current implementation of the GA shows promising results, there are other potential research directions worth exploring to further enhance its performance and applicability:

- **Expanding Genetic Algorithm Tuning:** The effectiveness of GA can be improved through careful tuning of their operators and parameters. Future work could explore enhancing genetic algorithm tuning by adopting advanced crossover techniques, such as multi-point or uniform crossover, to promote diversity and improve convergence. Adaptive mutation rates could further balance exploration and exploitation. Additionally, alternative selection strategies like rank-based selection may better control selective pressure and maintain genetic diversity.
- **Dynamic Fitness Functions:** Current implementation of the fitness functions employs a fixed weights for runtime, instruction count, and autophase metrics in the fitness function. Future work could explore dynamically modifying these weights during the evolutionary process. This dynamic approach could benefit more targeted optimization and improve the adaptability of the GA to varying optimization goals.
- **Hybrid Methods:** Combining RL and GA offers a promising approach to leverage the strengths of both techniques. RL ability to learn policies based on sequential decision-making can be applied to generate an initial population of optimization sequence for the GA. This would provide the GA with a good starting point, potentially accelerating convergence and improving the quality of solutions. Conversely, GA could refine the solutions discovered by the RL algorithm, exploring a broader search space and escaping local optima through its population-based stochastic search.

8 Acknowledgements

We would like to extend our heartfelt thanks to Professor Tang for her invaluable guidance throughout this project. She consistently took the time to discuss any changes and challenges we encountered, ensuring that we achieved the highest possible quality in our final product.

References

[1] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: robust, performant compiler optimization environments for AI research. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 92–105. https://doi.org/10.1109/CGO53902.2022.9741258

[2] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. arXiv:2101.04808 [cs.PL] https://arxiv.org/abs/2101.04808

[3] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. 2023. BenchDirect: A Directed Language Model for Compiler Benchmarks. arXiv:2303.01557 [cs.LG] https://arxiv.org/abs/2303.01557

9 Appendix

9.1 Code Base

Our code for generating results can be found in our Github repository.

9.2 Autophase Instructions

Autophase is an advanced feature extraction tool used in the realm of compiler optimization, developed by Huang et al. [3]. It is designed to capture a comprehensive range of program features, which are crucial for optimizing compiler performance. Autophase provides a detailed representation by encompassing 56 distinct dimensions of program characteristics. These dimensions include many features shared with InstCount, another feature extraction tool, but also introduce unique metrics that enhance its descriptive power. For example, Autophase tracks specific aspects such as the number of input arguments to PHI nodes and the total number of memory instructions. These metrics are not captured by InstCount, thus providing a richer set of data for optimization processes. The Autophase feature set enables our genetic algorithm to identify and exploit key aspects of the program's structure, ultimately leading to improved optimization.

9.3 GA-Opt Protocol

In this section, we outline the pseudocode for our GA implementation. Algorithm 1 can be found below.

Algorithm 3 rewards(env, initial_rt, initial_ic, initial_auto)

```
1: Initialize $after\_ic \leftarrow env.observation[”IrInstructionCount”]$
2: Initialize $after\_rt \leftarrow 0$
3: if $len(env.observation[”Runtime”]) == 0$ then
4:     $after\_rt \leftarrow 0$
5: else
6:     $after\_rt \leftarrow env.observation[”Runtime”][-1]$
7: end if
8: Initialize $runtime \leftarrow initial\_rt - after\_rt$
9: if $runtime < 0$ then
10:     $runtime \leftarrow 0$
11: end if
12: $runtime \leftarrow runtime * 0.5$
13: Initialize $ic \leftarrow initial\_ic - after\_ic$
14: if $ic < 0$ then
15:     $ic \leftarrow 0$
16: end if
17: $ic \leftarrow ic * 0.003$
18: Initialize $after\_auto \leftarrow env.observation[”Autophase”][51]$
19: Initialize $auto \leftarrow initial\_auto - after\_auto$
20: if $auto < 0$ then
21:     $auto \leftarrow 0$
22: end if
23: $auto \leftarrow auto * 0.002$
24: Initialize $combined \leftarrow runtime + ic + auto$
25: return $combined$
```

Algorithm 4 crossover(parent1, parent2)

```
1: Initialize $child1 \leftarrow parent1$
2: Initialize $child2 \leftarrow parent2$
3: if $random.random() < P_c$ then
4:     $point \leftarrow random.randint(1, \text{length}(parent1) - 1)$
5:     $child1 \leftarrow parent1[: point] + parent2[point :]$
6:     $child2 \leftarrow parent2[: point] + parent1[point :]$
7: end if
8: return $child1, child2$
```

Algorithm 5 mutate(individual)

```
1: Initialize $mutated\_individual \leftarrow$ copy of individual
2: for each gene in mutated_individual do
3:     if $random.random() < P_m$ then
4:         $gene \leftarrow random.choice(F)$
5:     end if
6: end for
7: return $mutated\_individual$
```

Algorithm 1 GA-OPT: Genetic Algorithm for Optimizing Compiler Flags

```
Require: Population size $N$, Generation count $G$, Episode length $L$, Mutation rate $P_m$, Crossover rate $P_c$, List of optimization flags $F$, Environment $env$
1: Initialize the population with $N$ Individuals, each a random sequence of length $L$ from $F$
2: Initialize $best\_fitness \leftarrow -\infty$
3: Initialize $best\_individual \leftarrow$ None
4: for each generation $g$ from 1 to $G$ do
5:     for each individual $ind$ in the population do
6:         $fitness \leftarrow \text{evaluate\_fitness}(env, ind)$
7:         if $fitness > best\_fitness$ then
8:             $best\_fitness \leftarrow fitness$
9:             $best\_individual \leftarrow ind$
10:         end if
11:     end for
12:     Select parents based on their fitness
13:     $new\_population \leftarrow [\ ]$
14:     while $\text{length}(new\_population) < N$ do
15:         Select two parents $parent1$ and $parent2$ from the population based on fitness
16:         $child1, child2 \leftarrow \text{crossover}(parent1, parent2)$
17:         $child1 \leftarrow \text{mutate}(child1)$
18:         $child2 \leftarrow \text{mutate}(child2)$
19:         Add $child1$ and $child2$ to $new\_population$
20:     end while
21:     $population \leftarrow new\_population$
22: end for
23: $best\_fitness$ and $best\_individual$
```

Algorithm 2 evaluate_fitness(env, individual)

```
1: Reset the environment
2: Initialize $total\_reward \leftarrow 0$
3: Initialize $initial\_ic \leftarrow env.observation[”IrInstructionCount”]$

4: Initialize $initial\_rt \leftarrow 0$
5: if $len(env.observation[”Runtime”]) == 0$ then
6:     $initial\_rt \leftarrow 0$
7: else
8:     $initial\_rt \leftarrow env.observation[”Runtime”][0]$
9: end if
10: Initialize $initial\_auto \leftarrow env.observation[”Autophase”][51]$
11: for each action in the individual do
12:     $action\_index \leftarrow$ index of action in $env.action\_space.flags$
13:     $observation, reward, done, info \leftarrow env.step(action\_index)$
14:     $combined \leftarrow \text{rewards}(env, initial\_rt, initial\_ic, initial\_auto)$

15:     $total\_reward \leftarrow total\_reward + combined$ if $combined$ is not None else $total\_reward$
16:     if $done$ then
17:         break
18:     end if
19: end for
20: return $total\_reward$
```