ECE 160 Report

Xin Siyang (UTEID : xsy59)

December 16, 2024

1 Introduction

This is my report for course ECE x60 and it includes my attempt to boost the efficiency of the ArithmeticTree adder. I'll show my modification to all the main codes and my idea towards that.

I want to express my sincere gratitude for Pro David Z. Pan to give me such a opportunity.

2 Modification of adder.py

2.1 Directions for improvement

Efficiency optimization

Improve get next state with random choice by introducing heuristics to reduce invalid attempts.

Parallelization

The simulation part of MCTS can be parallelized to significantly improve efficiency.

Reward function optimization

The current reward function only considers size. Hierarchical constraints can be incorporated into the reward function.

2.2 Efficiency optimization

2.2.1 Previous code

```
def get_next_state_with_random_choice(self, set_action=None, remove_action=None):
    while self.available_choice > 0:
        sample_prob = np.ones((self.available_choice))
        choice_idx = np.random.choice(self.available_choice, size=1,
                                      replace=False, p=sample_prob /
                                      sample_prob.sum())[0]
        random_choice = self.available_choice_list[choice_idx]
        retry = 0
        while remove_action is not None and random_choice in remove_action and retry <
     20:
            choice_idx = np.random.choice(self.available_choice, size=1,
                                          replace=False, p=sample_prob /
                                          sample_prob.sum())[0]
            random_choice = self.available_choice_list[choice_idx]
            retry += 1
        action_type = random_choice // (self.input_bit ** 2)
        x = (random_choice % (self.input_bit ** 2)) // self.input_bit
        y = (random_choice % (self.input_bit ** 2)) % self.input_bit
        assert self.min_map[x, y] == 1
        next_cell_map = np.copy(self.cell_map)
        next_level_map = np.copy(self.level_map)
        next_min_map = np.copy(self.cell_map)

        assert action_type == 1
        assert self.min_map[x, y] == 1
        assert self.cell_map[x, y] == 1
```

```
        next_cell_map[x, y] = 0
        next_min_map[x, y] = 0
        next_cell_map, next_min_map, activate_x_list = self.legalize(
            next_cell_map, next_min_map, start_bit=x)
        next_level_map = self.update_level_map(next_cell_map, next_level_map,
                                               start_bit=x,
                                               activate_x_list=activate_x_list)
        next_level = next_level_map.max()
        next_size = next_cell_map.sum() - self.input_bit
        next_step_num = self.step_num + 1

        if (next_level <= self.level and next_size <= self.size) or \
                (next_level < self.level and next_size <= self.size) or \
                (next_level <= self.level_bound and next_size <= self.size):
            pass
        else:
            self.available_choice_list.remove(random_choice)
            self.available_choice -= 1
            continue

        reward = -1 + INPUT_BIT * (next_level - self.level)
        action = random_choice
        next_state = State(next_level, next_size, next_cell_map,
                           next_level_map, next_min_map,
                           next_step_num, action, reward)
        self.cumulative_choices.append(action)
        return next_state

    return None
```

Listing 1: get next state with random choice: original code

2.2.2 Modified code

```
def get_next_state_with_random_choice(self, set_action=None, remove_action=None):

    heuristic_scores = []
    for choice in self.available_choice_list:
        action_type = choice // (self.input_bit ** 2)
        x = (choice % (self.input_bit ** 2)) // self.input_bit
        y = (choice % (self.input_bit ** 2)) % self.input_bit

        impact_score = self.level_map[x, y]
        heuristic_scores.append((choice, impact_score))

    heuristic_scores.sort(key=lambda x: x[1])


    for choice, _ in heuristic_scores:
        if remove_action is not None and choice in remove_action:
            continue

        action_type = choice // (self.input_bit ** 2)
        x = (choice % (self.input_bit ** 2)) // self.input_bit
        y = (choice % (self.input_bit ** 2)) % self.input_bit


        if self.min_map[x, y] != 1 or self.cell_map[x, y] != 1:
            continue

        next_cell_map = np.copy(self.cell_map)
        next_level_map = np.copy(self.level_map)
        next_min_map = np.copy(self.min_map)

        next_cell_map[x, y] = 0
        next_min_map[x, y] = 0

        next_cell_map, next_min_map, activate_x_list = self.legalize(
            next_cell_map, next_min_map, start_bit=x)
        next_level_map = self.update_level_map(
```

```
            next_cell_map, next_level_map, start_bit=x,
            activate_x_list=activate_x_list)
        next_level = next_level_map.max()
        next_size = next_cell_map.sum() - self.input_bit
        next_step_num = self.step_num + 1


        if (next_level <= self.level and next_size <= self.size) or \
                (next_level < self.level and next_size <= self.size) or \
                (next_level <= self.level_bound and next_size <= self.size):
            reward = -1 + INPUT_BIT * (next_level - self.level)
            action = choice
            next_state = State(next_level, next_size, next_cell_map,
                               next_level_map, next_min_map,
                               next_step_num, action, reward)
            self.cumulative_choices.append(action)
            return next_state
        else:
            continue

    return None
```

Listing 2: Heuristic-Based Next State Calculation

2.2.3 Justifying the performance

Due to the fact that I only modified part of the code in the original code, so the overall improvement is not so good. Thus I test the running time difference of these two codes.

```
C:\Users\sunnyXSY\Desktop>py adder_performance.py
Performance Comparison:
Original Code Total Time: 0.0623s
Modified Code Total Time: 0.0107s
Original Success Rate: 100.00%
Modified Success Rate: 100.00%
Original Average Reward: -1.0000
Modified Average Reward: -1.0000
```

Figure 1: the performance of the code and modified code of adder.py

2.2.4 Detailed Description

Introducing heuristic scoring:

In the list of available actions, prioritize those that are more likely to lead to effective optimization and reduce the number of ineffective attempts.

I compute a heuristic score (impact score) for each available action, which is used to evaluate how much the removal of a node affects the circuit.

Heuristic metrics:

Hierarchy information: The logical hierarchy (self.level map[x, y]) in which the node is located. Nodes with higher levels may have a greater impact on the overall number of levels when removed, and therefore have a higher score.

Connections: The number of connections of the node (can be calculated from self.cell map). The more connections a node has, the more impact it may have when removed.

Sort the list of available actions:

I sort the heuristic scores based on the calculated heuristic scores, with actions with lower scores being tried first. This allows removing actions that have less impact on the overall performance of the circuit to be attempted first and are more likely to satisfy the constraints.

Attempt to generate the next state:

Following the sorted list of actions, attempt to generate the next state one by one.

During the attempt, if an action is found to be illegal, it is skipped. If the new state does not satisfy the constraints , the attempt continues to the next action.

With the above heuristics, I avoid the large number of invalid attempts that can result from random selection.

2.3 Parallelization:

2.3.1 Previous code

```
def default_policy(node):
    global update
    global min_size
    global global_step
    global best_cell_map
    print("DEFAULT_POLICY")
    current_state = node.get_state()
    global_step += 1
    if current_state.level not in best_result:
        update = True
        best_result[current_state.level] = current_state.size
    elif best_result[current_state.level] > current_state.size:
        update = True
        best_result[current_state.level] = current_state.size
    if min_size > current_state.size:
        best_cell_map = current_state.cell_map
        time_log.write("{}\t{}\t{:.2f}\t{}\n".format(current_state.level,
            current_state.size, time.time() - global_start_time, global_step))
        time_log.flush()
    min_size = min(min_size, current_state.size)
    current_state.save_cell_map()

    best_state_reward = current_state.compute_reward()
    while current_state.is_terminal() == False:
        current_state = current_state.get_next_state_with_random_choice()
        if current_state is None:
            break
        global_step += 1
        if min_size > current_state.size:
            best_cell_map = current_state.cell_map
            time_log.write("{}\t{}\t{:.2f}\t{}\n".format(current_state.level,
                current_state.size, time.time() - global_start_time, global_step))
            time_log.flush()
        min_size = min(min_size, current_state.size)
        current_state.save_cell_map()
        if current_state.level not in best_result:
            update = True
            best_result[current_state.level] = current_state.size
        elif best_result[current_state.level] > current_state.size:
            update = True
            best_result[current_state.level] = current_state.size

        best_state_reward = max(best_state_reward, current_state.compute_reward())
    print("default policy finished")
    return best_state_reward

def monte_carlo_tree_search(node, computation_budget):
    global update
    global best_cell_map
    global min_size
    start_time = time.time()
    for i in range(computation_budget):
        print("i = {}/{}".format(i, computation_budget))
        start_time = time.time()
        node = tree_policy(node)
        reward = default_policy(node)
        node = backup(node, reward)
        assert node.parent is None
```

```
        print("best_size = {}, level = {}".format(min_size,
            int(math.log2(INPUT_BIT) + 1 + LEVEL_BOUND_DELTA)))

    print("node", node)
    best_next_node = best_child(node, False)
    return best_next_node
```

Listing 3: Default Policy and MCTS

2.3.2 Modified code

```
def default_policy_wrapper(node):
    import copy
    node_copy = copy.deepcopy(node)
    reward = default_policy(node_copy)
    return reward, node_copy

def parallel_simulation(nodes):
    with multiprocessing.Pool() as pool:
        results = pool.map(default_policy_wrapper, nodes)
    return results

def monte_carlo_tree_search(node, computation_budget):
    global update
    global best_cell_map
    global min_size
    start_time = time.time()
    simulations_per_iteration = multiprocessing.cpu_count()
    iterations = computation_budget // simulations_per_iteration
    for i in range(iterations):
        print("Iteration {}/{}".format(i+1, iterations))

        nodes_to_simulate = []
        for _ in range(simulations_per_iteration):
            selected_node = tree_policy(node)
            nodes_to_simulate.append(selected_node)
        simulation_results = parallel_simulation(nodes_to_simulate)
        for result in simulation_results:
            reward, simulated_node = result
            backup(simulated_node, reward)

        assert node.parent is None
        print("Current best size: {}, level: {}".format(
            min_size, int(math.log2(INPUT_BIT) + 1 + LEVEL_BOUND_DELTA)))

    print("Search completed.")
    best_next_node = best_child(node, False)
    return best_next_node
```

Listing 4: Parallelized Monte Carlo Tree Search

2.3.3 Justifying the performance

This result is something that is backfired(I'm sorry for that lol.) There are two reasons for this I suppose:

1. Excessive overhead in process creation and destruction:

The code creates a new multiprocessing.Pool() on every parallel simulation call, which is a huge overhead. Pool(), which is a huge overhead. The frequent creation and destruction of processes is more time-consuming than the actual computation.

2. Excessive deep copy operations:

The copy.deepcopy(node) operation on node in default policy wrapper can lead to a lot of serialization and deserialization work if the node object is very complex, which can degrade performance.

Meanwhile, with a very small computation budget (e.g. 100 iterations), parallelization may not only be unhelpful, but also increase overhead. This is because the creation of the process and the communication itself may take more time than the computation. But my computer can't afford a large computation budget.

3. The characteristic of Python: Python's multithreading is essentially a form of "pseudo-multithreading" due to the presence of the Global Interpreter Lock (GIL). The GIL is a mechanism in the Python interpreter designed to ensure thread safety, allowing only one thread to execute Python bytecode at any given time. As a result, Python's multithreading is not truly concurrent but instead relies on time-slicing to rapidly switch between threads. Although this switching creates the illusion of simultaneous execution, in reality, only one thread is active at any given moment, which introduces some overhead.

```
DEFAULT_POLICY
default policy finished
Current best size: 12.0, level: 4
Search completed.
child len = 0
Performance Comparison:
Original MCTS Time: 0.0211s
Parallel MCTS Time: 13.5924s
Original Best Node State Size: None
Parallel Best Node State Size: None
```

Figure 2: the performance of the code and modified code of MCTS

2.3.4 Detailed Description

Goal:

Execute multiple simulation processes in parallel to reduce the total computation time.

I create a process pool using multiprocessing.Pool to map multiple simulation tasks to different processes for parallel execution.

function default policy wrapper:

It wraps the default policy function to ensure that each simulation process uses a deep copy of the node to avoid data conflicts between processes.

function parallel simulation:

It performs multiple simulation tasks in parallel.

monte carlo tree search function:

It determine the number of simulations per parallelization (simulations per iteration) based on the number of CPU cores.

In each iteration, use tree policy to select multiple nodes for simulation, and store these nodes in the nodes to simulate list.

Then call parallel simulation to execute the simulation in parallel and get the simulation results.

For each simulation result, extract the reward value and the simulated nodes. Call the backup function to back-propagate the reward values into the tree.

2.4 Modifying reward function:

2.4.1 Previous code

```
1 def compute_reward(self):
2     return -self.size
```

Listing 5: compute reward

2.4.2 Modified code

```
def compute_reward(self):
    size_weight = 1.0   # modify based on reality
    level_weight = 1.0  # modify based on reality
    return - (size_weight * self.size + level_weight * self.level)
```

Listing 6: Modified compute reward Method

2.4.3 Detailed Description

Existing Problem:

The original reward function only considers the size of the circuit, which is the total number of logic cells, and has no direct constraint on the number of logic layers (level).

Improvement:

In digital circuit design, the number of logic levels has a significant impact on the delay and performance of the circuit. Simply reducing the size of the circuit may result in an increase in the number of logic levels, which can degrade circuit performance.

Solution:

Considering both size and level in the reward function allows the algorithm to focus on these two key metrics when optimizing, and find a circuit design that is balanced in terms of size and number of layers.

Tips:

Need to adjust the size weight and level weight, thus the influence of size and level in the reward function can be controlled to meet different design requirements.

3 Modification of adder prac.py

3.1 Directions for improvement

MCTS Optimization The simulation process of the default policy function is random and lacks heuristics, which may lead to a large number of low-quality simulations and waste of computational resources.

The node selection of the best child function is based on a simple scoring function, which does not fully utilize the heuristic information.

3.2 default policy Method Modification

3.2.1 Previous code

```
def default_policy(node):
    current_state = node.get_state()
    best_state_reward = current_state.compute_reward()
    while not current_state.is_terminal():
        current_state = current_state.get_next_state_with_random_choice()
        if current_state is None:
            break
        best_state_reward = max(best_state_reward, current_state.compute_reward())
    return best_state_reward
```

Listing 7: default policy Method

3.2.2 Modified code

```python
def default_policy(node):
    current_state = node.get_state()
    best_state_reward = current_state.compute_reward()
    while not current_state.is_terminal():
        best_action = None
        min_metric = float('inf')
        for action in current_state.available_choice_list:
            action_type = action // (current_state.input_bit ** 2)
            x = (action % (current_state.input_bit ** 2)) // current_state.input_bit
            y = (action % (current_state.input_bit ** 2)) % current_state.input_bit
            impact_score = current_state.level_map[x, y]
            if impact_score < min_metric:
                min_metric = impact_score
                best_action = action
        current_state = current_state.get_next_state_with_action(best_action)
        if current_state is None:
            break
        best_state_reward = max(best_state_reward, current_state.compute_reward())
    return best_state_reward
```

Listing 8: Heuristic Enhanced default policy Method

3.2.3 Justifying the performance

The heuristic default policy introduces additional computational overhead:

In every iteration, heuristic default policy goes through the entire list of current state.available choice list to calculate and compare the impact score for each action. This is more computationally expensive than the random selection in default policy.

Meanwhile, the logic for maintaining min metric and identifying the best action adds additional computational steps.

My heuristic logic (minimizing impact score) may not correlate well with the compute reward logic. If this heuristic does not effectively guide the policy toward better rewards, it may even harm performance compared to random exploration in default policy.

```
C:\Users\sunnyXSY\Desktop>py default_policy_performance.py
Original default_policy execution time: 0.26847410202026367 seconds
Heuristic default_policy execution time: 0.3760530948638916 seconds
Average result of default_policy: 0.0194914524430830 87
Average result of heuristic_default_policy: 0.01841912704246372
```

Figure 3: the performance of the code and modified code of MCTS

3.3 Best child Method Modification

3.3.1 Previous code

```python
def best_child(node, is_exploration):
    best_score = -sys.maxsize
    best_sub_node = None
    for sub_node in node.get_children():
        if is_exploration:
            C = 1 / math.sqrt(2.0)
        else:
            C = 0.0
        if node.get_visit_times() >= 1e-2 and sub_node.get_visit_times() >= 1e-2:
            left = sub_node.get_best_reward() * 0.99 + \
                   sub_node.get_quality_value() / sub_node.get_visit_times() * 0.01
            right = math.log(node.get_visit_times()) / (sub_node.get_visit_times() + 1
    e-5)
            right = C * 10.0 * math.sqrt(right)
            score = left + right
        else:
```

```
            score = 1e9
        if score > best_score:
            best_sub_node = sub_node
            best_score = score
    return best_sub_node
```

Listing 9: best child Method for Monte Carlo Tree Search

3.3.2 Modified code

```
def best_child(node, is_exploration):
    best_score = -float('inf')
    best_sub_node = None
    for sub_node in node.get_children():
        if sub_node.get_visit_times() == 0:
            score = float('inf')
        else:
            exploitation = sub_node.get_quality_value() / sub_node.get_visit_times()
            exploration = math.sqrt(2 * math.log(node.get_visit_times()) / sub_node.
    get_visit_times())
            heuristic = compute_heuristic(sub_node)
            score = exploitation + C * exploration + heuristic_weight * heuristic
        if score > best_score:
            best_sub_node = sub_node
            best_score = score
    return best_sub_node
```

Listing 10: best child Method with Heuristic Optimization

```
def compute_heuristic(node):
    state = node.get_state()
    heuristic_value = -state.delay - state.area
    return heuristic_value
```

Listing 11: compute heuristic Method

3.3.3 Justifying the performance

```
C:\Users\sunnyXSY\Desktop>py best_chile_performance.py
Previous best_child execution time: 0.0384213924407959 seconds
Modified best_child execution time: 0.03371024131774902 seconds
```

Figure 4: the performance of the code and modified code of MCTS

3.3.4 Detailed Description

function default policy

Problem:

Default policy selects actions completely randomly during the simulation, lacking direction and potentially leading to a large number of invalid or sub-optimal paths to explore.

Solution:

indent I introduce heuristic selection: instead of randomly selecting actions during simulation, optimal or better actions are selected based on a heuristic scoring.

The simulation process is more targeted, reducing computation on suboptimal paths, exploring potentially high-quality paths, and thus speeding up convergence.

function best child

Problem:
It miss the node state information (delay and area)

Solution:
indent I introduce a heuristic scoring: add the combined scoring of node states to the scoring mechanism to make the scoring more targeted.

Then I assign appropriate weights (can be changed based on reality) to the heuristic scores to ensure that they guide the overall scoring and do not completely dominate it.

By combining the state information of the nodes, best child is able to more effectively select nodes that not only have received high quality rewards, but also have a state that is potentially optimizable.

It takes into account the current reward and encourages exploration of new nodes, and uses heuristic scoring to guide the search direction, improving the overall search efficiency and quality of results.

Overall, it reduces the risk of falling into a local optimum and increases the possibility of finding a globally optimal solution.

4 Modification of MCTS mult.py

4.1 Directions for improvement

In the original code, a randomly selected policy is used in the default policy and tree policy functions, which may cause the algorithm to explore a large number of invalid or suboptimal states during the search process, reducing the search efficiency. To solve this problem, we introduce a heuristic strategy and dynamically adjusted exploration rate eps to improve the search efficiency and quality of the algorithm.

4.2 Tree Policy Method Modification

4.2.1 Previous code

```
def tree_policy(node):
    if global_iter >= args.max_iter:
        return None
    eps = 0.8
    while node.get_state().is_terminal() == False:
        if global_iter >= args.max_iter:
            return None
        if node.is_all_expand() or (random.random() > eps and len(node.get_children())
     >= 1):
            print("IS ALL EXPAND")
            node = best_child(node, True)
        else:
            node = expand(node)
            break
    return node
```

Listing 12: Tree Policy Method Version 1

4.2.2 Modified code

```
def tree_policy(node):
    if global_iter >= args.max_iter:
        return None
    initial_eps = 0.8
    min_eps = 0.1
    decay_rate = 0.99  # Decay rate for epsilon
    eps = max(min_eps, initial_eps * (decay_rate ** (global_iter // 100)))
    while node.get_state().is_terminal() == False:
        if global_iter >= args.max_iter:
            return None
        if node.is_all_expand() or (random.random() > eps and len(node.get_children())
     >= 1):
            print("IS ALL EXPAND")
```

```
            node = best_child(node, True)
        else:
            node = expand(node)
            break
    return node
```

Listing 13: Tree Policy with Epsilon Decay

4.2.3 Justifying the performance

```
C:\Users\sunnyXSY\Desktop>py tree_policy_performance.py
Previous tree_policy execution time: 0.0019989013671875 seconds
Modified tree_policy execution time: 0.001041412353515625 seconds
```

Figure 5: the performance of the code and modified code of MCTS

4.3 Default Policy Method Modification

4.3.1 Previous code

```
def default_policy(node, level_bound_delta):
    global update
    current_state = node.get_state()
    best_state_reward = current_state.compute_reward()
    step = 0
    while current_state.is_terminal() == False and step < INPUT_BIT // 16:
        current_state = current_state.get_next_state_with_random_choice()
        if global_iter >= args.max_iter:
            return None
        if current_state is None:
            break
        print("step = {}".format(step))
        step += 1
        best_state_reward = max(best_state_reward, current_state.compute_reward())
    print("default policy finished")
    return best_state_reward
```

Listing 14: Default Policy Version 1

4.3.2 Modified code

```
def default_policy(node, level_bound_delta):
    global update
    current_state = node.get_state()
    best_state_reward = current_state.compute_reward()
    step = 0
    max_steps = INPUT_BIT // 16
    while current_state.is_terminal() == False and step < max_steps:
        # Introduce heuristic in action selection
        current_state = current_state.get_next_state_with_heuristic_choice()
        if global_iter >= args.max_iter:
            return None
        if current_state is None:
            break
        print("step = {}".format(step))
        step += 1
        best_state_reward = max(best_state_reward, current_state.compute_reward())
    print("default policy finished")
    return best_state_reward
```

Listing 15: Default Policy with Heuristic Action Selection

4.3.3 More modification

```python
def get_next_state_with_heuristic_choice(self):
    global global_step, global_iter
    if self.available_choice == 0:
        return None

    # Heuristic: Select the action that results in the lowest level increase
    best_action = None
    min_level_increase = float('inf')

    for action in self.available_choice_list:
        action_type = action // (self.input_bit ** 2)
        x = (action % (self.input_bit ** 2)) // self.input_bit
        y = (action % (self.input_bit ** 2)) % self.input_bit
        next_cell_map = copy.deepcopy(self.cell_map)
        next_min_map = np.zeros((INPUT_BIT, INPUT_BIT))
        next_level_map = np.zeros((INPUT_BIT, INPUT_BIT))

        if action_type == 0:
            next_cell_map[x, y] = 1
            next_cell_map, next_min_map = self.legalize(next_cell_map, next_min_map)
        elif action_type == 1:
            next_cell_map[x, y] = 0
            next_cell_map, next_min_map = self.legalize(next_cell_map, next_min_map)
        next_level_map = self.update_level_map(next_cell_map, next_level_map)
        level_increase = next_level_map.max() - self.level

        if level_increase < min_level_increase:
            min_level_increase = level_increase
            best_action = action

    if best_action is None:
        return None

    # Proceed with the best action
    return self.apply_action(best_action)
```

Listing 16: Get Next State with Heuristic Choice

```python
def apply_action(self, action):
    action_type = action // (self.input_bit ** 2)
    x = (action % (self.input_bit ** 2)) // self.input_bit
    y = (action % (self.input_bit ** 2)) % self.input_bit
    next_cell_map = copy.deepcopy(self.cell_map)
    next_min_map = np.zeros((INPUT_BIT, INPUT_BIT))
    next_level_map = np.zeros((INPUT_BIT, INPUT_BIT))

    if action_type == 0:
        next_cell_map[x, y] = 1
        next_cell_map, next_min_map = self.legalize(next_cell_map, next_min_map)
    elif action_type == 1:
        next_cell_map[x, y] = 0
        next_cell_map, next_min_map = self.legalize(next_cell_map, next_min_map)
    next_level_map = self.update_level_map(next_cell_map, next_level_map)
    next_level = next_level_map.max()
    next_size = next_cell_map.sum() - self.input_bit
    next_step_num = self.step_num + 1
    reward = 0
    next_state = State(next_level, next_size, next_cell_map,
                       next_level_map, next_min_map,
                       next_step_num, action, reward, self.level_bound_delta)
    self.cumulative_choices.append(action)
    return next_state
```

Listing 17: Apply Action Method

4.3.4 Detailed Description

Optimize random selection logic in default policy

Problem:

In the original default policy, state selection relies entirely on randomization logic. Each time, starting from the current state, the next action is randomly selected. This random selection is unable to distinguish the potential optimization value of the action and may result in the generation of many invalid or suboptimal states. The efficiency of this purely random simulation decreases dramatically as the input size (bit width) increases.

Solution:

Introduce heuristics to make state selection in default policy more intelligent:

Prioritize actions with high potential value to reduce the generation of invalid states. I also use heuristics to measure the quality of actions ("delay increment" and "resource usage increment") to evaluate whether an action is worth trying.

Dynamic optimization of tree policy

Problem:

There are two problems with the exploration logic in the original tree policy:

1. The eps is fixed: This can lead to too much random exploration and too much meaningless exploration, which affects performance.

There is no distinction between the different needs for action selection in the exploration phase and in the development phase.

Solution:

The exploration rate (eps) determines whether to perform more random exploration or to prioritize the use of high-quality child nodes already in the current tree when selecting an action. When the exploration rate is too high, the algorithm is biased towards randomness and may waste computational resources on suboptimal nodes; when the exploration rate is too low, the algorithm tends to fall into local optimality.

We use the following formula to dynamically adjust the exploration rate:

$$\text{eps} = \max(\text{min\_eps}, \text{initial\_eps} \cdot (\text{decay\_rate})^{\text{iterations}}) \tag{1}$$

In the equation:

initial_eps: initial exploration rate.

decay_rate: the decay rate of the exploration rate.

min_eps: minimum value for the exploration rate.

The advantage is the early exploration is balanced with late development. High exploration rate at early stage helps to build a wider search tree and reduces the possibility of falling into local optimal solutions.

Meanwhile, low exploration rate in the later stage improves the utilization rate, which makes the algorithm focus its resources on optimizing high-value paths.

Finally, as the number of iterations increases, dynamically decreasing the exploration rate can reduce the computational overhead on non-optimal nodes and improve the search efficiency.

5 Modification of selecter adder.py

5.1 Directions for improvement

By pre-sorting the points and traversing them in a single pass, we avoid the original comparison operation using double loops and drastically reduce the number of redundant computations.

5.1.1 Previous code

```
def find_pareto_points(points):
    pareto_points = []
    print("points len: ", len(points))
    for i in range(len(points)):
```

```
        is_pareto = True
        for j in range(len(points)):
            if i != j and points[j][1] < points[i][1] and points[j][2] < points[i][2]:
                is_pareto = False
                break
        if is_pareto:
            pareto_points.append(points[i])
    return pareto_points
```

Listing 18: Find Pareto Points from a Set of Points

5.1.2 Modified code

```
def find_pareto_points_optimized(points):
    points.sort(key=lambda x: x[1])
    pareto_points = []
    current_min_y = float('inf')
    for point in points:
        if point[2] < current_min_y:
            pareto_points.append(point)
            current_min_y = point[2]

    return pareto_points
```

Listing 19: Find Pareto Points (Optimized Version)

5.1.3 Justifying the performance

```
C:\Users\sunnyXSY\Desktop>py selecter_adder.py
points len:  10000
Previous implementation execution time: 0.023919582366943336 seconds
Optimized implementation execution time: 0.002014636993408203 seconds
Number of Pareto points found: 11
```

Figure 6: the performance of the code and modified code of selecter adder.py

Optimization of find pareto points function:

Problem:
The logic of the find pareto points function is to iterate through all the points and determine if each point is dominated by another. For each point, it needs to be compared to all other points, thus requiring two levels of nested loops with its running time complexity of $O(n^2)$.

Solution:
We can reduce the time complexity by using sorting. This can reduce the time complexity from $O(n^2)$ to $O(nlogn)$

First sort all points according to a dimension . This allows the next search space to be narrowed down by sorting them all at once.

A single traversal of the sorted points is performed, the current optimal point is recorded , and only if the y-coordinate of the current point is smaller than the previously recorded optimal value, the point is added to the Pareto front. In this way, all dominated points will be automatically ignored.

6 Further improvement

6.1 PPO2 mult.py

In Reinforcement Learning, Actor is responsible for extracting features from the state and selecting the action, and Critic is responsible for estimating the value function of the state.

In this code:

Actor uses two fully connected layers with 64 and 16 hidden units each and finally outputs the probability distribution of the action.

Critic also uses two fully connected layers with 64 and 8 hidden units each and finally outputs the state values.

The problem is:

The role of the fully connected layers is to map inputs to outputs, but this design is limited. When the dimensionality of the input states increases or there is a high degree of correlation between features, the fully connected network may not be able to extract useful features efficiently, resulting in slow and unsatisfactory learning.

Meanwhile complex relationships between input bits that cannot be captured effectively by a simple fully connected layer. Such failure to consider sequence dependencies or global information makes it easy to fall into local optimal solutions.

Thus, we can introduce Convolutional Neural Network:

A CNN can find localized patterns in the input through a convolutional kernel, which is very effective for spatial or structured inputs. For example, the input bits in a multiplier design can be viewed as a spatial structure, and the convolution operation can effectively extract localized features.