WATCH THE UNOBSERVED: A SIMPLE APPROACH TO PARALLELIZING MONTE CARLO TREE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Monte Carlo Tree Search (MCTS) algorithms have achieved great success on many challenging benchmarks (e.g., Computer Go). However, they generally require a large number of rollouts, making their applications to planning costly. Furthermore, it is also extremely challenging to parallelize MCTS due to its inherent sequential nature: each rollout heavily relies on the statistics (e.g., node visitation counts) estimated from previous simulations to achieve an effective exploration-exploitation tradeoff. In spite of these difficulties, we develop an algorithm, *P-UCT*, to effectively parallelize MCTS, which achieves linear speedup and exhibits negligible performance loss with an increasing number of workers. The key idea in P-UCT is a set of statistics that we introduce to track the number of on-going yet incomplete simulation queries (named as *unobserved samples*). These statistics are used to modify the UCT tree policy in the selection steps in a principled manner to retain effective exploration-exploitation tradeoff when we parallelize the most time-consuming expansion and simulation steps. Experimental results on a proprietary benchmark and the public Atari Game benchmark demonstrate the near-optimal linear speedup and the superior performance of P-UCT when compared to these existing techniques.

1 INTRODUCTION

Recently, Monte Carlo Tree Search (MCTS) algorithms such as UCT (Kocsis et al., 2006) have achieved great success in solving many challenging artificial intelligence (AI) benchmarks, including video games (Guo et al., 2016) and Go (Silver et al., 2017). However, they rely on a large number (e.g., thousands) of Monte Carlo rollouts to construct search trees for planning actions in decision-making systems, which leads to high cost in time (Browne et al., 2012). For this reason, there has been an increasing demand for parallelizing MCTS over multiple workers. However, parallelizing MCTS without degrading its performance is difficult (Segal, 2010; Mirsoleimani et al., 2018a; Chaslot et al., 2008), mainly due to the intrinsic sequential nature in the algorithm (Figure 1(a)). Specifically, to select the most urgent nodes that allow effective exploration-exploitation tradeoff, MCTS needs to estimate the most up-to-date statistics from the previous rollouts. On the other hand, the expansion and the simulation steps are generally the most time-consuming part compared to the other two steps, and should be intensively parallelized. However, when paralleling these two steps using multiple workers, it becomes inevitable that the workers can only access a set of outdated statistics (Section 2.2). The key question is therefore how to keep track of the correct statistics in the search tree when we parallelize the expansion and the simulation steps, with the hope of retaining effective exploration-exploitation tradeoff as the original non-parallel UCT (Section 2).

To this end, we propose P-UCT, a novel parallel MCTS algorithm, that attains linear speedup without sacrificing the performance. This is achieved by a conceptual innovation (Section 3.1) as well as an efficient real system implementation (Section 3.2). Specifically, the key idea that we introduce in P-UCT to overcome the aforementioned challenge is a set of statistics that tracks the number of on-going yet incomplete simulation queries (named as *unobserved samples*). We combine these newly devised statistics with the original statistics of *observed samples* to modify the UCT node-selection policy in the selection steps in a principled manner, which, as we shall show in Section 4, effectively retains exploration-exploitation tradeoff during parallelization. Our proposed approach has been successfully deployed in a production system for *efficiently* and *accurately* estimating the user pass-rates of a mobile game "Joy City", with the purpose of reducing their design cycles. On

this proprietary benchmark, we show that P-UCT achieves near-optimal linear speedup and superior performance in predicting user pass-rate (Section 5.1). In addition, we further evaluate P-UCT on the Atari game benchmark and compare it to existing state-of-the-art parallel MCTS algorithms (Section 5.2), which also demonstrate our superior speedup and performance.

2 On the Difficulties of parallelizing MCTS

In this section, we briefly introduce the MCTS and the UCT algorithms, and then discuss the key challenges of their parallelization.

2.1 Monte Carlo Tree Search and Upper Confidence bound for Trees (UCT)

We consider the Markov Decision Process (MDP) $\langle \mathcal{S}, \mathcal{A}, R, P, \gamma \rangle$, where an agent interacts with the environment in order to maximize a long-term cumulative reward. Specifically, an agent at state $s_t \in \mathcal{S}$ takes an action $a_t \in \mathcal{A}$ according to a *policy* π, so that the MDP transits to the next state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ and emits a reward $R(s_t, a_t, s_{t+1})$. The objective of the agent is to learn an optimal policy π^* such that the long-term cumulative reward is maximized:

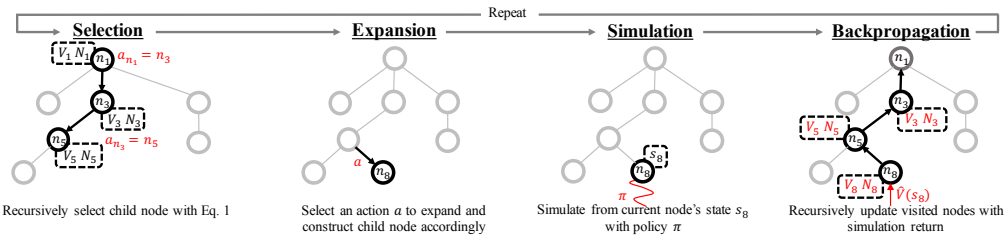
$$\max_{\pi} \mathbb{E}_{a_t \sim \pi, s_{t+1} \sim P} \Big[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \,|\, s_0 = s \Big], \tag{1}$$

where $s \in \mathcal{S}$ denotes the initial state and γ is the discount factor. Many reinforcement learning (RL) algorithms have been developed to solve the above problem (Sutton & Barto, 2018), including model-free algorithms (Mnih et al., 2013; 2016; Williams, 1992; Konda & Tsitsiklis, 2000; Schulman et al., 2015; 2017) and model-based algorithms (Nagabandi et al., 2018; Weber et al., 2017; Bertsekas, 2005; Deisenroth & Rasmussen, 2011). Monte Carlo Tree Search (MCTS) is a model-based RL algorithm that *plans* the best action at each time step (Browne et al., 2012). Specifically, it uses the MDP model (or its sampler) to identify the best action at each time step by constructing a search tree (Figure 1(a)), where each node n represents a visited state, each edge from n denotes an action a_n that can be taken at that state, and the landing node n' denotes the state it transits to after taking a_n. As shown in Figure 1(a), MCTS repeatedly performs Monte Carlo rollouts that consist of four *sequential* steps: selection, expansion, simulation and backpropagation. The selection step traverses over the existing search tree until the leaf node (or other termination conditions are satisfied) by choosing actions (edges) a_n at each node n according to a tree policy. One widely used node-selection policy is the one used in the Upper Confidence bound for Trees (UCT) (Kocsis et al., 2006):

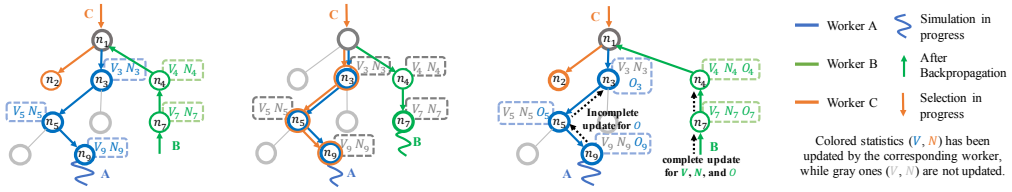
$$a_n = \arg\max_{c \in \mathcal{C}(n)} \left\{ V_c + \beta \sqrt{\frac{2 \log N_n}{N_c}} \right\}, \tag{2}$$

where $\mathcal{C}(n)$ denotes the set of all child nodes for n; the first term V_c is an estimate for the long-term cumulative reward that can be received when starting from the state represented by node c, and the second term represents the uncertainty (size of the confidence interval) of that estimate. The confidence interval is calculated based on the Upper Confidence Bound (UCB) (Auer et al., 2002; Auer, 2002) using N_n and N_c, which denote the number of times that the nodes n and c have been visited during the MCTS rollouts, respectively. Therefore, the key idea of the UCT policy (2) is to select the best action according to an optimistic estimation (i.e., the upper confidence bound) of the expected return, which strikes a balance between the exploitation (first term) and the exploration (second term) with β controlling their tradeoff. Once the selection process reaches a leaf node of the search tree (or other termination conditions are met), we will expand the node according to a prior policy by adding a new child node. Then, in the simulation step, we estimate its value function (long-term cumulative reward) $\hat{V}_n$ by running the environment simulator with a default (simulation) policy. Finally, during backpropagation, we update the statistics V_n and N_n along the traversed path according to:

$$N_n \leftarrow N_n + 1, \qquad V_n \leftarrow \frac{N_n - 1}{N_n} V_n + \frac{1}{N_n} \hat{V}_n. \tag{3}$$



(a) Each (non-parallel) MCTS rollout consists of four *sequential* steps: selection, expansion, simulation and backpropagation, where the expansion and the simulation steps are generally most time-consuming.



(b) Ideal parallelization (c) Naive parallelization (d) P-UCT

Figure 1: The MCTS algorithm and its parallelization. (a) An overview of MCTS. (b) The ideal parallelization: the most up-to-date statistics $\{V_n, N_n\}$ (in chromatic color) are assumed to be available to all workers as soon as a simulation begins (unrealistic in practice). (c) The key challenge in parallelizing MCTS: the workers can only access outdated $\{V_n, N_n\}$ (in gray-color), leading problems like *collapse of exploration*. (d) P-UCT tracks the number of incomplete simulation queries, which is denoted as O_n, and modifies the UCT policy in a principled manner to retain effective exploration-exploitation tradeoff. It achieves comparable speedup and performance as the ideal parallelization.

2.2 The intrinsic difficulties of parallelizing MCTS

The above discussion implies that the MCTS algorithm is intrinsically sequential: each selection step in a new rollout requires the previous rollouts to complete in order to deliver the updated statistics, V_n and N_n, for the UCT tree policy (2). Although this requirement of up-to-date statistics is not mandatory for implementation, it is in practice intensively required to achieve effective exploration-exploitation tradeoff (Auer et al., 2002). Specifically, up-to-date statistics best help the UCT tree policy to explore and prune non-rewarding branches as well as extensively visit advantageous path for additional planning depth. Likewise, to achieve the best possible performance, when using multiple workers to parallelize MCTS rollouts, it is also important to ensure that each worker uses the most recent statistics (the colored V_n and N_n in Figure 1(b)) in its own selection step. However, this is impossible in parallelizing MCTS based on the following observations. First, the expansion step and the simulation step are generally the most time-consuming parts compared to the other two steps, because they involve a large number of interactions with the environment simulator (Section 3.2). Therefore, as exemplified by Figure 1(c), when a worker C initiates a new selection step, the other workers A and B are most likely still in their simulation or expansion steps. This prevents them from updating the (global) statistics for other workers like C, which happens at the backpropagation step. Using outdated statistics (the gray-colored V_n and N_n in Figure 1(c)) at different workers could lead to a significant loss in speedup and performance, due to behaviors like *collapes of exploration* or *exploitation failure*, which we shall discuss thoroughly in Section 4. To give an example, Figure 1(c) illustrates the collapse of exploration, where worker C traverses over the same path as the worker A in its selection step due to the determinism of (2). Specifically, if the statistics are unchanged between worker A and C begins the selection step, they will choose the same node according to (2), which greatly reduces the diversity of exploration. Therefore, the key question that we want to address in parallelizing MCTS is how to keep track of the correct statistics and modify the UCT policy in a *principled* manner, with the hope of retaining effective exploration-exploitation tradeoff at different workers.

3 P-UCT

In this section, we first develop the conceptual idea of our P-UCT algorithm (Section 3.1), and then we present a real system implementation using a master-slave architecture (Section 3.2).

3.1 WATCH THE UNOBSERVED SAMPLES IN UCT TREE POLICY

As we pointed out earlier, the key question we want to address in parallelizing MCTS is how to deliver the most up-to-date statistics $\{V_n, N_n\}$ to each worker so that they can achieve effective exploration-exploitation tradeoff in its selection step. This is assumed to be the case in the ideal parallelization in Figure 1(b). Algorithmically, it is equivalent to the standard (non-parallel) MCTS except that the rollouts are performed in parallel by different workers. Unfortunately, in practice, the statistics $\{V_n, N_n\}$ available to each worker are generally outdated because of the slow and incomplete simulation and expansion steps at the other workers (Figure 1(c)). Specifically, since the estimated value $\hat{V}_n$ is unobservable before simulations complete and workers should not wait for the updated statistics to proceed, the (partial) loss of statistics $\{V_n, N_n\}$ is unavoidable. Now the question becomes: is there an alternative way to addressing the issue? The answer is in the affirmative and will be explained below.

Aiming at bridging the gap between naive parallelization (Figure 1(b)) and the ideal case (Figure 1(b)), we closely examine their difference in terms of the availability of statistics. As illustrated by the colors of the statistics, their only difference in $\{V_n, N_n\}$ is caused by the on-going simulation process. As suggested by (3), although V_n can only be updated after a simulation step is completed, the newest N_n information can actually be available as early as a worker initiates a new rollout. This is the key insight that we leverage to enable effective parallelization in our P-UCT algorithm. Motivated by this, we introduce another quantity, O_n, to count *the number of rollouts that have been initiated but not yet completed*, which we name as *unobserved samples*. That is, our new statistics O_n watch the number of unobserved samples, and are then used to correct the UCT tree policy (2) into the following form:

$$a_n = \arg\max_{c \in \mathcal{C}(n)} \left\{ V_c + \beta \sqrt{\frac{2\log(N_n + O_n)}{N_c + O_c}} \right\}. \tag{4}$$

The intuition of the above modified node-selection policy is that when there are O_n workers simulating (querying) node n, the confidence interval at node n will eventually be shrunk after they complete. Therefore, adding O_n and O_c to the exploration term considers such a fact beforehand and let other workers be aware of it. Despite its simple form, (4) provides a principled way to retain effective exploration-exploitation tradeoff under parallel settings; it corrects the confidence bound towards better exploration-exploitation tradeoff. As the confidence level is instantly updated (i.e., at the beginning of simulation), more recent workers are guaranteed to observe additional statistics, which prevent them from extensively querying the same node as well as finding better nodes to query. For example, when multiple children are in demand for exploration, (4) allows them to be explored evenly. In contrast, when a node has been sufficiently visited (i.e., large N_n and N_c), adding O_n and O_c from the unobserved samples have little effect on (4) because the confidence interval is sufficiently shrunk around V_c, allowing extensively exploitation of the best-valued child.

3.2 SYSTEM IMPLEMENTATION USING MASTER-SLAVE ARCHITECTURES

We now proceed to explain the system implementation of P-UCT, where the overall architecture is shown in Figure 2 (see Appendix A for the details). Specifically, we use a master-slave architecture to implement the P-UCT algorithm with the following considerations. First, recall that the expansion and the simulation steps are the most time-consuming part compared to the selection and the backpropagation steps. Therefore, these two steps should be intensively parallelized, and in fact, they are relatively easy to parallelize (e.g., different simulations could be performed independently). For this reason, we will use multiple simulation (expansion) workers to run them in parallel. Second, as we discussed earlier, different workers need to access the most up-to-date statistics $\{V_n, N_n, O_n\}$ in order to achieve successful exploration-exploitation tradeoff. To this end, a centralized architecture for the selection and backpropagation step is more preferable as it allows adding strict restrictions to the retrieval and update of the statistics, making them up-to-date. Specifically, we use a centralized master process to maintain a *global* set of statistics (in addition to other data such as game

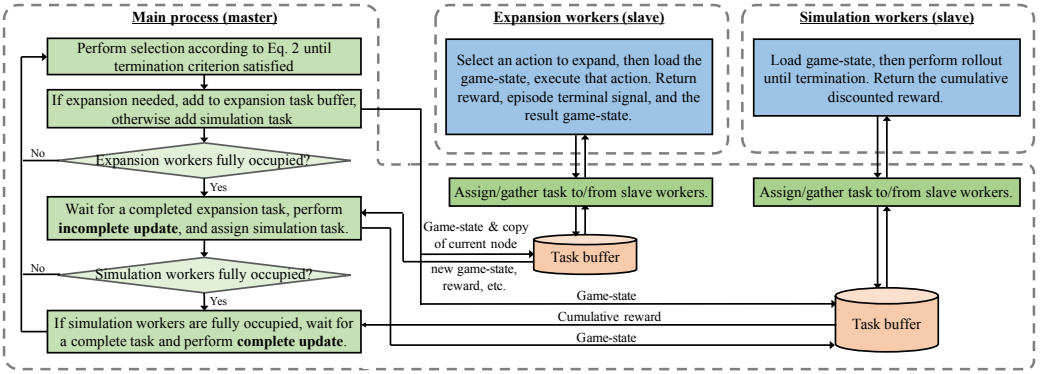
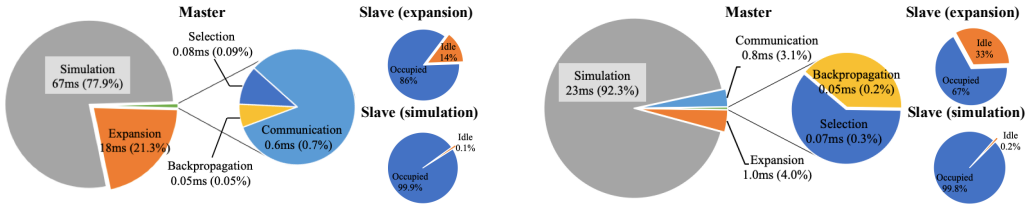


Figure 2: The overall system architecture that implements the P-UCT algorithm. The Green blocks and the task buffers are operated at the master (main) process, while the blue blocks are executed by and the slave processes (workers). Pseudo-code is provided in Appendix A.



(a) Tap-elimination game

(b) Atari games

Figure 3: The breakdown of the time consumption on two game benchmarks (Section 5). The simulation (expansion) time at the master process is defined as the time that it waits for the simulation (expansion) workers to complete, which happens only when the corresponding set of workers is full (Figure 2). The backpropagation time is the sum of the time spent on *incomplete update* and on *complete update*. All speed-tests are performed under 16 expansion workers and 16 simulation workers, and are averaged over 5 game-levels in task (a) or 5 games in task (b).

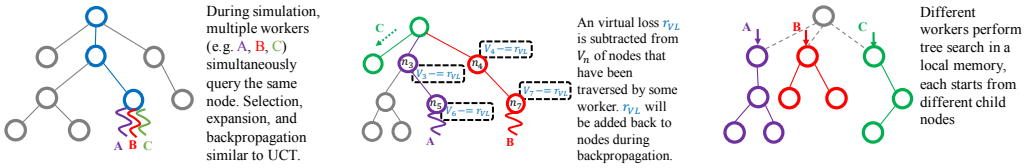
states), and let it be in charge of the backpropagation step (i.e., updating the glocal statistics) and the selection step (i.e., using the global statistics). As shown in Figure 2, the master process repeatedly performs rollouts until a predefined number of simulations is reached. During each rollout, it selects nodes to query, assign expansion and simulation tasks to different workers, and collect the returned results to update the global statistics. In particular, we use the following *complete update* and *incomplete update* to track N_n and O_n along the traversed path (see Figure 1(d)):

$$[\text{complete update}] \quad O_n \leftarrow O_n - 1;\ N_n \leftarrow N_n + 1 \tag{5}$$

$$[\text{incomplete update}] \quad O_n \leftarrow O_n + 1 \tag{6}$$

In addition, V_n is also updated in the complete update step using (3). Such a clear division of labor between the master process and the slave workers provides sequential selection and backpropagation steps when we parallelize the costly expansion and simulation steps. It ensures up-to-date statistics for all workers by the centralized master process and achieves linear speedup without much performance degradation (see Section 5 for the experimental results).

To justify the above rationale of our system design, we perform a set of running time analysis for our developed P-UCT system and report the results in Figure 3. We show the breakdown of the time-consumption for different parts at the master process and at the slave workers. First, we focus exclusively on slave workers. With close-to-100% occupancy rate for the simulation workers, the simulation step is fully parallelized. Although the expansion workers are not fully utilized, the expansion step is maximumly parallelized since the number of required simulation and expansion tasks is identical. This suggests the existence of an optimal (task-dependent) ratio between the



(a) Leaf parallelization (LeafP) (b) Tree parallelization (TreeP) (c) Root parallelization (RootP)

Figure 4: Three popular parallel MCTS algorithms. LeafP parallelizes the simulation steps, TreeP uses virtual loss to encourage exploration, and RootP parallelizes the subtrees of the root node.

number of expansion workers and the number of simulation workers that fully parallelize both steps with the least resources (e.g. memory). Returning to the master process, on both benchmarks, we see a clear dominance of the time spent on the simulation and the expansion steps even they are both extensively parallelized by 16 workers. This supports our design to parallelize only the simulation and expansion steps. We finally focus on the additional communication overhead caused by parallelization. Although more time-consuming compared to simulation and backpropagation, the communication overhead is negligible compared to the time used by the expansion and the simulation steps. Other details in our system, such as the centralized game-state storage and the reduction of the communication overhead, are further discussed in Appendix A.

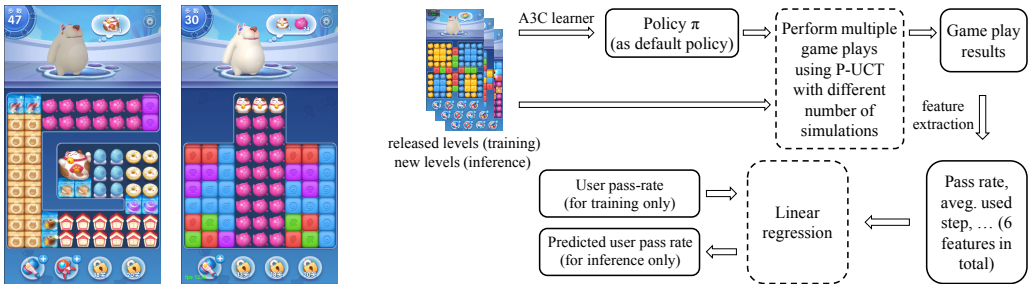
4 The Benefits of Watching Unobserved Samples

In this section, we discuss the benefits of watching unobserved samples in P-UCT, and compare it with several popular parallel MCTS algorithms (Figure 4), including Leaf Parallelization (LeafP), Tree Parallelization (TreeP) with virtual loss, and Root Parallelization (RootP).[1] LeafP parallelizes the leaf simulation, which leads to an effective hex game solver (Wang et al., 2018). TreeP with virtual loss has recently achieved great success in challenging real-world tasks such as Go (Silver et al., 2017). And RootP parallelizes the subtrees of the root node at different workers, and aggregates the statistics of the subtrees after all the workers complete their simulations (Soejima et al., 2010).

We argue that, by introducing the additional statistics O_n, P-UCT achieves a better exploration-exploitation tradeoff than the above existing methods. First, LeafP and TreeP represent two extremes of parallelization in such a tradeoff. LeafP lacks diversity in exploration as all its workers are assigned to simulating (querying) the same node, leading to performance drop caused by *collapse of exploration* in much the same way as the naive parallelization. This could be observed from our experiments in Section 5. In contrast, although the virtual loss used in TreeP could encourage exploration diversity, this hard additive penalty could cause *exploitatin failure*: workers will be less likely to co-simulating the same node even when they are certain that it is optimal (Mirsoleimani et al., 2017). RootP tries to avoid these issues by letting workers perform an independent tree search. However, this reduces the equivalent number of rollouts at each worker, decreasing the accuracy of the UCT policy (2). Different from the above three approaches, P-UCT achieves a much better exploration-exploitation tradeoff in the following manner. It encourages exploration by using O_n to "penalize" the nodes that have many in-progress simulations. Meanwhile, it allows multiple workers to exploit the most rewarding node since this "penalty" vanishes when N_n becomes large (see (4)).

While we have seen the severe disadvantages of neglecting information of the unobserved samples (as is done by LeafP, TreeP, and RootP), we now examine whether keeping track of them provides substantial benefits. Specifically, we examine how far away P-UCT is from the ideal parallelization in Figure 1(b). Recall that the blue-colored statistics in Figure 1(b)-(c) at nodes 3, 5 and 9 imply that they are available to workers, while the gray-colored statistics means they are outdated. Although the unobserved sample counts O_n compensates the outdated N_n and make it up-to-date, the expected return V_n can not be corrected before the simulation completes and returns $\hat{V}$ (i.e., $\hat{V}$ is missing). Therefore, we now focus on analyzing the influence of missing $\hat{V}$, and we argue that the tree structure of the search tree could mitigate its harm on the V_n's along the selected path. To see

[1] We refer the readers to Chaslot et al. (2008) for more details.



(a) Screenshots of the Joy City game (b) The user pass-rate prediction system

Figure 5: Overview of the Joy City game and our deployed user pass-rate prediction system.

this, first note that all the selected trajectories start from the root node and ends at some terminal node. Therefore, shallower nodes are generally visited more frequently, and their V_n are thus more robust to missing $\hat{V}$'s. In contrast, for nodes with deeper path depth, they are less frequently visited and their V_n estimates are supposed to be more sensitive to missing $\hat{V}$'s. Recall that the problem of missing $\hat{V}$ is caused by in-progress simulations at the leaf node, and the number of missing $\hat{V}$ is determined by the number of in-progress simulations (i.e., the number of unobserved samples). However, these deeper nodes are less likely to have a large number of unobserved samples (in-progress simulations) because the probability for selecting the same path decreases significantly as the node depth increases. Therefore, the V_n's at these deeper nodes are generally less outdated.

5 Experiments

In this section, we evaluate the proposed P-UCT algorithm on two benchmarks. First, we successfully deploy our P-UCT algorithm in a production system to accurately and efficiently predict the user pass-rate of a proprietary level-oriented mobile game (Section 5.1), with the objective being significantly reducing the game design cycle. Due to the $16\times$ speedup by P-UCT with negligible performance loss, our system saves a huge amount of human cost in game-testing. In addition, we further evaluate P-UCT on the public Atari Game benchmark and compare it with several state-of-the-art baseline algorithms, which also demonstrates its superior performance and speedup.

5.1 Experiments on the "Joy City" game

Joy City is a proprietary level-oriented game with diverse and challenging gameplay (Figure 5(a)). Its basic rule is tapping the connected items to eliminate them (see Appendix B.1 for the details), and to pass a level, players are required to complete certain level-goals within a given number of steps. For this reason, we refer it as the *tap game* below. It is a challenging reinforcement learning task due to its large number of game-state ($12^{9\times 9} \approx 2.5 \times 10^{87}$) and high randomness of the transition probability. In particular, since new items could be randomly dropped into the game board, there are hundreds of possible outcomes after a single move, which results in a large branching factor.

The user pass-rate prediction system During a game design cycle, to achieve the desired game pass-rates, a game designer needs to hire many human testers to extensively test all the levels before its release, which generally takes a long time and is inaccurate. Therefore, it would greatly reduce the game design cycle if we can develop a testing system that is able to provide *quick* and *accurate* feedback about the user pass-rates. Figure 5(b) gives an overview of our deployed pass-rate prediction system, where P-UCT is used to mimic average user performance and provide features for predicting the human pass-rate. As we will show later, it can achieve significant speedup without significant performance loss,[2] allowing the game designer to get the feedback in 20 minutes instead of 12 hours. Specifically, we use P-UCT with different numbers of rollouts to mimic players with

[2]Due to the complexity the tap game, model-free RL algorithms such as A3C (Mnih et al., 2016) and PPO (Schulman et al., 2017) fail to achieve satisfactory performance and thus cannot perform an accurate prediction. On the other hand, MCTS could achieve good performance but takes a long time in testing.

Table 1: Pair-wise sample t-test of pass-rate across 130 levels between two AI bots (different number of MCTS rollouts) and the players. "Avg. diff" means the average difference between the pass-rate of the bot and that of the human players. p-value measures the likelihood that two sets of paired samples are statistically similar (i.e. larger means similar). Effect size measures the strength of the difference (larger means greater difference).

AI bot	# rollouts	Avg. diff.	Effect size	p-value
P-UCT	10	-1.54	0.07	0.4120
P-UCT	100	22.18	0.88	0.0000

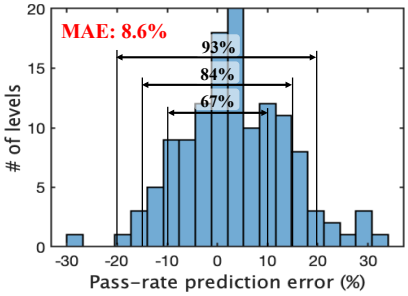
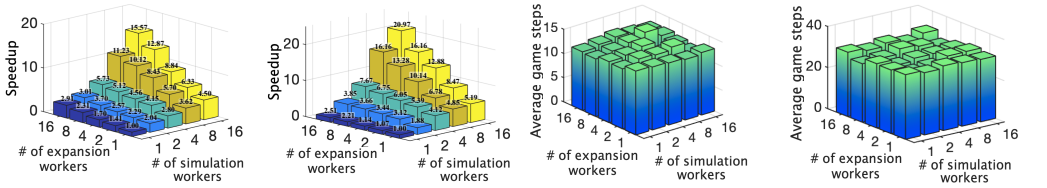


Figure 6: Distribution of the pass-rate prediction error on 130 game-levels.



(a) Speedup (level 35) (b) Speedup (level 58) (c) Performance (level 35) (d) Performance (level 58)

Figure 7: P-UCT speedup and performance. Results are averaged over 10 runs. P-UCT achieves linear speedup without much loss in performance (measured in average number of game steps).

different skill levels, where P-UCT with 10 rollouts is used to represent average players while the agent with 100 rollouts mimics skillful players. This is verified by the pair-wise sample t-test result provided in Table 1. With 10 simulations, the P-UCT agent performs statistically similar (p-value $> 5\%$) to human players, while with 100-simulation, the agent performs statistically better (p-value $< 5\%$). Besides, Figure 6 shows that our pass-rate prediction system achieves 8.6% mean absolute error (MAE) on 130 released game-levels, with 93% of them having MAE less than 20%.

Speedup and performance loss We now examine the speedup and the performance of P-UCT on two typical levels (Level-35 and Level-58)[3] of the tap game. An ideal parallel algorithm should exhibit linear speedup with an increasing number of workers without significant performance loss. We evaluate P-UCT with different numbers of expansion and simulation workers (from 1 to 16) and report the speedup results in Figures 7(a)–(b). First, note that when we have the same number of expansion workers and simulation workers, P-UCT achieves ideal linear speedup. Furthermore, Figures 7 also suggest that both the expansion workers and the simulation workers are important. For example, when the number of the expansion worker is small, the speedup is significantly restricted. The full speedup numbers can also be found in Appendix B.3. Besides the ideal speedup property, P-UCT suffers negligible performance loss with the increasing number of workers, as shown in Figures 7(c)–(d). The standard deviations of the performance (measured in the average number of game steps) over different numbers of expansion and simulation servers are only 0.67 and 1.22 for Level-35 and Level-58, respectively, which are much smaller than their average game steps (10 and 30).

5.2 Experiments on the Atari Game benchmark

We now further evaluate P-UCT on the Arcade Learning Environment (ALE) (Bellemare et al., 2013), a classical benchmark for reinforcement learning (RL) and planning algorithms. ALE is an ideal testbed for MCTS algorithms as it is challenging for the following reasons. First, the Atari

[3]Level-35 is relatively simple, requiring 18 steps for an average player to pass, while Level-58 is relatively difficult and needs more than 50 steps to solve.

Table 2: The performance on 15 Atari games. Average episode return ($\pm$ standard deviation) over 3 trials are reported. Bold indicates the best average episode return. * indicates statistically better performance, indicated by p-value < 0.05 in t-test, between P-UCT and TreeP (not marked if both methods perform statistically similar). Similarly, † means that P-UCT performs statistically better than LeafP, and ‡ implies that P-UCT performs statistically better than RootP.

Environment	P-UCT	TreeP	LeafP	RootP	PPO
Alien	**6536**±1093*†	3990±980	4715±295	5063±365	1850
Boxing	**100**±0*†‡	99±0	93±4	99±0	94
Breakout	**413**±14†‡	407±15	325±31	269±21	274
Centipede	**703561**±122633*†‡	238421±23865	129974±42368	145569±8759	4386
Freeway	**32**±0†	**32**±0	31±1	**32**±0	32
Gravitar	**5238**±221†	4926±514	3365±216	4230±643	737
MsPacman	**20941**±1250*†‡	14030±2450	5060±210	7820±643	2096
NameThisGame	**31155**±5645†	22616±3376	17455±625	24463±2637	6254
RoadRunner	**43800**±2050*†‡	20830±4540	25900±351	34300±2160	25076
Robotank	**95**±10	83±7	79±12	78±15	5
Qbert	17953±225	**18033**±218	16525±125	14137±1630	14293
SpaceInvaders	3000±813*†	2488±705	2931±230	**3587**±63	942
Tennis	**4**±2*†‡	-1±0	-1±0	-1±1	-14
TimePilot	**48390**±6721*	33800±1099	38453±707	37833±612	4342
Zaxxon	38200±1560†‡	**39200**±4440	11100±351	13920±420	5008

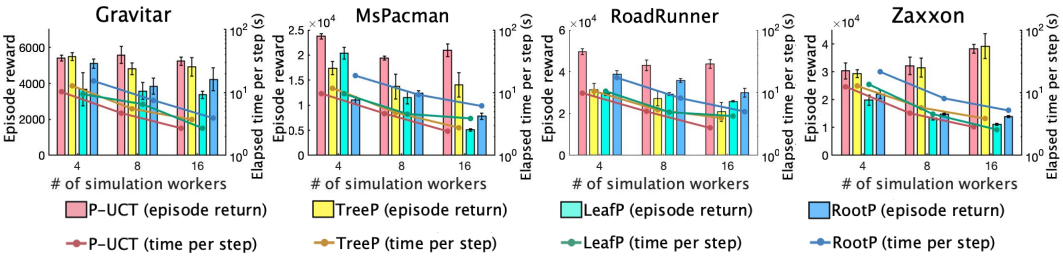


Figure 8: Speed and performance test of our P-UCT along with three baselines on four Atari games. All experiments are repeated three times and the mean and standard deviation (for episode reward only) is reported. For P-UCT, the number of expansion workers is fixed as one.

games in ALE generally last thousands of time steps, which requires a substantially long planning horizon. Second, many Atari games have a sparse reward as well as complex game strategy, which adds additional difficulties to effective exploration. We compare P-UCT to three parallel MCTS algorithms discussed earlier in Section 4: (i) TreeP with virtual loss, (ii) LeafP, and (iii) RootP. Additionally, we compare the results with PPO (Schulman et al., 2017), which is regarded as one of the most stable and effective on-policy RL algorithms. Though not exhaustive, this set of baselines still embodies many of the latest advancements and can be indeed regarded as the state-of-the-art. In P-UCT, we use a small policy network distilled (Hinton et al., 2015; Rusu et al., 2015) from a pre-trained policy network using PPO as the simulation policy. Using a pre-trained simulation policy in MCTS is shown to achieve more accurate estimations of the outcome and better overall performance (Silver et al., 2017). More details about the experiment can be found in Appendix C.

We first compare the performance, measured by average episode reward, between P-UCT and the baselines on 15 Atari games, which is done with 16 simulation workers and one expansion worker (for a fair comparison). Each task is repeated 3 times with the mean and standard deviation reported in Table 2. Due to the better exploration-exploitation tradeoff during selection, P-UCT out-performs all comparison models in 12 out of 15 tasks. Student t-test further show that P-UCT performs significantly better (p-value < 0.05) than TreeP, LeafP, and RootP in 8, 12, and 7 tasks, respectively. Next, we examine the influence of the number of simulation workers on the speed and the performance. In Figure 8, we compare the average episode return as well as time consumption (per step) for 4, 8, and 16 simulation workers. The bar plots in Figure 8 indicates that P-UCT experiences little performance loss with an increasing number of workers. In contrast, the baseline approaches exhibit significant performance degradation when parallelized using more workers. P-UCT also achieves

the fastest speed compared to the baselines, thanks to the efficient master-slave architecture that minimizes the time consumption of non-parallelized steps (Section 3.2). In conclusion, our proposed P-UCT not only out-performs baseline approaches significantly under the same number of workers but also achieves negligible performance loss with the increasing level of parallelization. Moreover, this performance is achieved without compromise for efficiency.

6 Related Work

MCTS Monte Carlo Tree Search is a planning method for optimal decision making in problems with either deterministic (Silver et al., 2016) or stochastic (Schäfer et al., 2008) environments. It has already had a profound influence on Artificial Intelligence applications (Browne et al., 2012). Recently, there has been a wide range of work that combines MCTS and other reinforcement learning (RL) methods, providing mutual improvements to both methods. For example, Guo et al. (2014) harnesses the power of MCTS to boost the performance of model-free RL approaches, achieving both significant performance gain and policy effectiveness; Shen et al. (2018) bridges the gap between MCTS and graph-based search with recurrent neural networks (RNN), achieving significant performance gain compared to RL and other knowledge base completion (KBC) baselines.

Parallel MCTS There have been many approaches developed to design parallel MCTS methods, with the objective being two-fold: achieve near-linear speedup under a large number of workers and maintain the performance of the non-parallel algorithm. In the following, we summarize related work on both aspects. Popular parallelization approaches of MCTS include leaf parallelization, root parallelization, and tree parallelization (Chaslot et al., 2008). Leaf parallelization aims at collecting better statistics by assigning multiple workers to query the same node (Cazenave & Jouandeau, 2007). However, this comes at the cost of wasting diversity of the tree search. Therefore, its performance degrades significantly despite the near-ideal speedup with the help of a client-server network architecture (Kato & Takeuchi, 2010). In root parallelization, multiple search trees are built and assigned to different workers. Additional work incorporates periodical synchronization of statistics from different trees, which results in better performance in real-world tasks (Bourki et al., 2010). However, a case study on Go reveals its inferiority with even a small number of workers (Soejima et al., 2010). Compared to the above approaches, tree parallelization is more promising for its potential to use more accurate statistics. In this setting, multiple workers traverse, perform queries, and update on a shared search tree. Tree parallelization benefits significantly from two techniques. First, a virtual loss is added to avoid querying the same node by different workers (Chaslot et al., 2008). This has been adopted in various successful applications of MCTS such as Go (Silver et al., 2016) and Dou-di-zhu (Whitehouse et al., 2011). Additionally, architecture side improvements such as using pipeline (Mirsoleimani et al., 2018b) or lock-free structure (Mirsoleimani et al., 2018a) speedup the algorithm significantly. However, though being able to increase diversity, virtual loss degrades the performance under even four workers (Mirsoleimani et al., 2017; Bourki et al., 2010).

7 Conclusion

In this paper, we developed P-UCT, a novel parallel MCTS algorithm. It addresses the problem of outdated statistics when parallelizing MCTS by watching the number of unobserved samples. Based on the newly devised statistics, it modifies the UCT node-selection policy in a principled manner, which achieves effective exploration-exploitation tradeoff. Specifically, it avoids the undesirable behaviors like the collapse of exploration or exploitation failure as in other existing parallel MCTS algorithms. Together with our efficiency-oriented system implementation, P-UCT achieves near-optimal linear speedup as well as negligible performance degradation across a wide range of tasks. Specifically, it has been successfully deployed in a real-world production system, breaking the efficiency barrier that limits many MCTS applications. In particular, it could be used to efficiently and accurately predict the user pass-rate of a game, which provides timely feedback for a game designer and greatly reduces the game design cycle. In addition, it also achieves superior performance and speed on 15 Atari games compared to existing state-of-the-art algorithms.

REFERENCES

Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, 2013.

Dimitri P Bertsekas. Dynamic programming and suboptimal control: A survey from adp to mpc. *European Journal of Control*, 11(4-5):310–334, 2005.

Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hérault, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, et al. Scalability and parallelization of monte-carlo tree search. In *International Conference on Computers and Games*, pp. 48–58. Springer, 2010.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Tristan Cazenave and Nicolas Jouandeau. On the parallelization of uct. In *proceedings of the Computer Games Workshop*, pp. 93–101. Citeseer, 2007.

Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pp. 60–71. Springer, 2008.

Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472, 2011.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pp. 3338–3346, 2014.

Xiaoxiao Guo, Satinder Singh, Richard Lewis, and Honglak Lee. Deep learning for reward design to improve monte carlo tree search in atari games. *arXiv preprint arXiv:1604.07095*, 2016.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Hideki Kato and Ikuo Takeuchi. Parallel monte-carlo tree search with simulation servers. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pp. 491–498. IEEE, 2010.

Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pp. 1008–1014, 2000.

S Ali Mirsoleimani, Aske Plaat, H Jaap van den Herik, and Jos Vermaseren. An analysis of virtual loss in parallel mcts. In *ICAART (2)*, pp. 648–652, 2017.

S Ali Mirsoleimani, Jaap van den Herik, Aske Plaat, and Jos Vermaseren. A lock-free algorithm for parallel mcts. In *ICAART (2)*, pp. 589–598, 2018a.

S Ali Mirsoleimani, Jaap van den Herik, Aske Plaat, and Jos Vermaseren. Pipeline pattern for parallel mcts. In *ICAART (2)*, pp. 614–621, 2018b.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7559–7566. IEEE, 2018.

Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

Jan Schäfer, Michael Buro, and Knut Hartmann. The uct algorithm applied to games with imperfect information. *Diploma, Otto-Von-Guericke Univ. Magdeburg, Magdeburg, Germany*, 2008.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Richard B Segal. On the scalability of parallel uct. In *International Conference on Computers and Games*, pp. 36–47. Springer, 2010.

Yelong Shen, Jianshu Chen, Po-Sen Huang, Yuqing Guo, and Jianfeng Gao. M-walk: Learning to walk over graphs using monte carlo tree search. In *Advances in Neural Information Processing Systems*, pp. 6786–6797, 2018.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. Evaluating root parallelization in go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):278–287, 2010.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Shiqi Wang, Meng Ding, and Shuqin Li. Hex game system based on p-mcts. In *2018 Chinese Control And Decision Conference (CCDC)*, pp. 6639–6642. IEEE, 2018.

Théophane Weber, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017.

Daniel Whitehouse, Edward J Powley, and Peter I Cowling. Determinization and information set monte carlo tree search for the card game dou di zhu. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pp. 87–94. IEEE, 2011.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

SUPPLEMENTARY MATERIAL

A ALGORITHM DETAILS FOR P-UCT

Pseudo-code of the proposed P-UCT algorithm is provided in Algorithm 1. Specifically, it provides the workflow of the master process. When the number of completed updates ($t_{complete}$) has not exceeded the maximum simulation step T_{max}, the main process keeps performing selection, expansion, simulation, and backpropagation. Selection and backpropagation are performed in the main process, while the two others are assigned to slave workers. The backpropagation step is divided into two sub-routines *incomplete update* (Algorithm 2) and *complete update* (Algorithm 3). The former is evoked before simulation, while the latter is called after receiving simulation results. Task index τ is added to help the main process to track different tasks returned from slave workers. To maximize efficiency, the master process keeps assigning expansion and simulation tasks until all slave workers are occupied.

Communication overhead of P-UCT The choice for centralized game-state storage stems from the following observations: (i) size of the game-state is usually small, which allows efficient inter-process transformation, and (ii) each game-state is used at most $|\mathcal{A}|$ times, thus is inefficient to store in multiple processes. Although this design may not be ideal for all tasks, it is at least a reasonable one. During rollouts, game-states generated by any expansion worker may be later used by any other expansion and simulation workers. Therefore, either a per-task transformation or decentralized storage is needed. For the latter case, however, since a game-state will be used at most $|\mathcal{A}|$ times, most workers will not need it, which results in inefficiency of the decentralized storage.

Another possible solution is to store the game-states in shared memory. However, to receive benefit from it, the following conditions should be satisfied: (i) each process can access (read/write) the memory relatively fast even if some collisions may happen, and (ii) the shared memory is big enough to hold all game-states that may be accessed. If the two condition holds, we may be able to reduce the communication overhead. Although, since the communication overhead is negligible even with 16 simulation and expansion workers, we should consider using more workers to speedup the algorithm.

B EXPERIMENT DETAILS AND SYSTEM DESCRIPTION OF THE JOY CITY GAME

This section describes the basic rules of the Joy City game (Appendix B.1) as well as details about the deployed pass-rate prediction system (Appendix B.2).

B.1 DESCRIPTION OF THE JOY CITY GAME

This section serves as an introduction to the basic rules of the tap game. We can click cells with connected color regions to eliminate them. The remaining cells then collapse to fill in the gaps of exploded ones. To goal is to fulfill all level requirements (goals) within a fixed number of clicks. Figure 9(a) provides consecutive snapshots for playing level 10 of the game. The goal of this level is depicted on the top, which is 3 "cats" and 24 "balloons". The top-left corner represents the remaining steps. Players have to accomplish all given goals before the step runs out. Figure 9(a) demonstrates successful gameplay, where only 6 steps are used to complete the level. In each of the three left frames, the cell noted by the purple circle is clicked. Immediately, the same-color region marked with a red frame is eliminated. Different goals/obstacles react differently. For instance, when some cell is exploded beside a balloon, it will also explode. Frame two demonstrates the use of props. Tapping regions with connectivity above a certain threshold will provide prop as a bonus. They have special effects that can help players pass the level faster. Finally, in the last screenshot, all goals are completed and we pass the level.

Figure 9(b) further demonstrates the variety of levels. Specifically, the left-most frame depicts a special "boss level", where the goal is the "kill" the evil cat. The cat will randomly throw objects to

Algorithm 1 P-UCT

Input: environment emulator $\mathcal{E}$, root tree node n_{root}, maximum simulation step T_{max}, maximum simulation depth d_{max}, number of expansion workers N_{exp}, and number of simulation workers N_{sim}
Initialize: expansion worker pool $\mathcal{W}_{exp}$, simulation worker pool $\mathcal{W}_{sim}$, game-state buffer $\mathcal{B}$, $t \leftarrow 0$, and $t_{complete} \leftarrow 0$
while $t_{complete} < T_{max}$ **do**
  Traverse the tree top down from root node n_{root} following (4) until (i) its depth greater than d_{max}, (ii) it is a leaf node, or (iii) it is a node that has not been fully expanded and *random()* < 0.5
  if expansion is required **then**
    $\bar{n} \leftarrow$ shallow copy of the current node; $s \leftarrow$ game state of node n (retrieved from $\mathcal{B}$)
    Assign expansion task $(t, \bar{n}, s)$ to pool $\mathcal{W}_{exp}$ // t **is the task index**
  else
    $s \leftarrow$ game state of node n; assign simulation task (t, s) to pool $\mathcal{W}_{sim}$ if episode not terminated
    Call **incomplete_update**(n); if episode terminated, call **complete_update**$(t, n, 0.0)$
  end if
  if $\mathcal{W}_{exp}$ fully occupied **then**
    Wait for a expansion task with return: (task index τ, game state s, reward r, terminal signal d, task index τ); expand the tree according to s, τ, r, and d; assign simulation task (τ, s) to pool $\mathcal{W}_{sim}$
    Call **incomplete_update**(t, n)
  else continue
  if $\mathcal{W}_{sim}$ fully occupied **then**
    Wait for a simulation task with return: (task index τ, node n, cumulative reward $\bar{r}$)
    Call **complete_update**$(\tau, n, \bar{r})$; $t_{complete} \leftarrow t_{complete} + 1$
  else continue
  $t \leftarrow t + 1$
end while

Algorithm 2 incomplete_update

input: node n
while $n \neq$ null **do**
  $O_n \leftarrow O_n + 1$
  $n \leftarrow \mathcal{PR}(n)$ // $\mathcal{PR}(n)$ **denotes the parent node of** n
end while

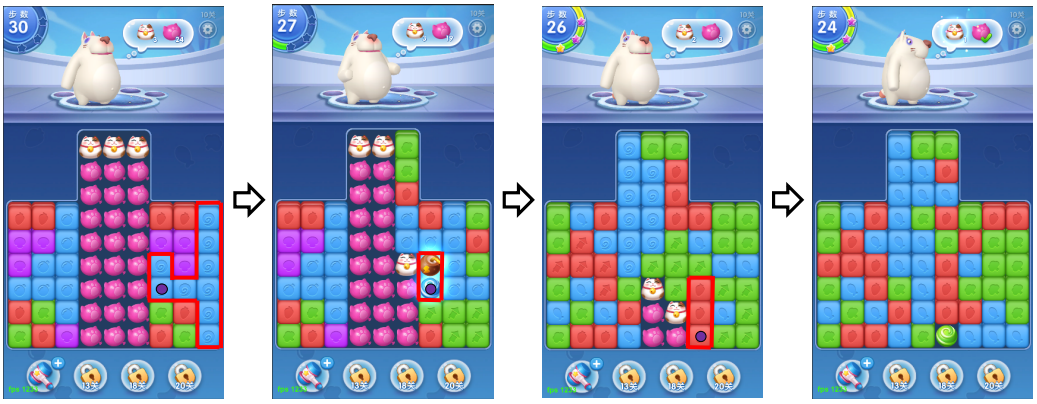
Algorithm 3 complete_update

input: task index t, node n, reward $\bar{r}$, γ
while $n \neq$ null **do**
  $N_n \leftarrow N_n + 1$; $O_n \leftarrow O_n - 1$
  Retrieve reward r according to task index t
  $\bar{r} \leftarrow r + \gamma \bar{r}$; $V_n \leftarrow \frac{N_n - 1}{N_n} V_n + \frac{1}{N_n} \bar{r}$
  $n \leftarrow \mathcal{PR}(n)$ // $\mathcal{PR}(n)$ **denotes the parent node of** n
end while

the cells, adding additional randomness. Three other frames illustrate relatively hard levels, which is revealed from their low-connectivity, abundance and complexity of the obstacles, and special layout.

B.2 Details of the level pass-rate prediction system

Workflow of the pass-rate prediction model is provided in Figure 5(b). The system has two working phases, i.e., training and inference. Specifically, training and validation are done on 300 levels that have been released in a test version of the game. In the training phase, the system has access to both the level and players' pass-rate, while only levels are available in the inference phase. In both phases, the levels are first fed into an asynchronous advantage actor-critic (A3C) (Mnih et al., 2016) learner for a base policy π. It is then used by the P-UCT agent as a prior to select expand action as well as the default policy for simulation. We then use P-UCT to perform multiple gameplays. The maximum depth and width (maximum number of child nodes for each node) of the search tree is 10 and 5, respectively. The number of simulations is set to 10 and 100 to get AI bots with different skill levels. Six features (three for both the 10-simulation and 100-simulation agent) are extracted from the gameplay results. Specifically, the features are AI's pass-rate, average used step divided by the provided step (the number at the top-left corner in the screenshots in Figure 9), and median used step divided by the provided step. During training, the features, as well as the players' pass-rate, is used to learn a linear regressor, while in the inference phase, the regression model is used to predict user pass-rate.



(a) A demonstrated game play in level 10 of the tap game. Purple dots refers to the tapped cell, and red regions indicate directly eliminated cells.

A "boss level" Small initial connectivity Various kinds of hard-to-remove obstacles and spiral layout

(b) Examples of levels with different rule, difficulty, and layout.

Figure 9: Snapshots of the Tap-elimination game.

Table 3: Speedup on two levels of the tap game. M_e is the number of expansion workers and M_s is the number of simulation workers.

Lv.	Level 35					Level 58				
M_e \ M_s	1	2	4	8	16	1	2	4	8	16
1	1.0	2.0	2.8	3.6	4.5	1.0	1.8	4.1	4.8	5.1
2	1.4	2.2	4.1	5.7	6.3	1.1	3.1	5.3	6.7	8.4
4	1.7	2.5	4.5	8.4	8.8	1.1	3.4	6.1	10.1	12.8
8	2.3	3.0	5.	10.1	12.8	1.2	3.6	6.7	13.2	16.1
16	2.9	3.7	5.7	11.2	15.5	1.2	3.8	7.6	16.1	20.9

B.3 Additional Experimental Results

In this section, we list the additional experimental results. In Table 3, we report the specific speedup number for different numbers of expansion worker and simulation workers.

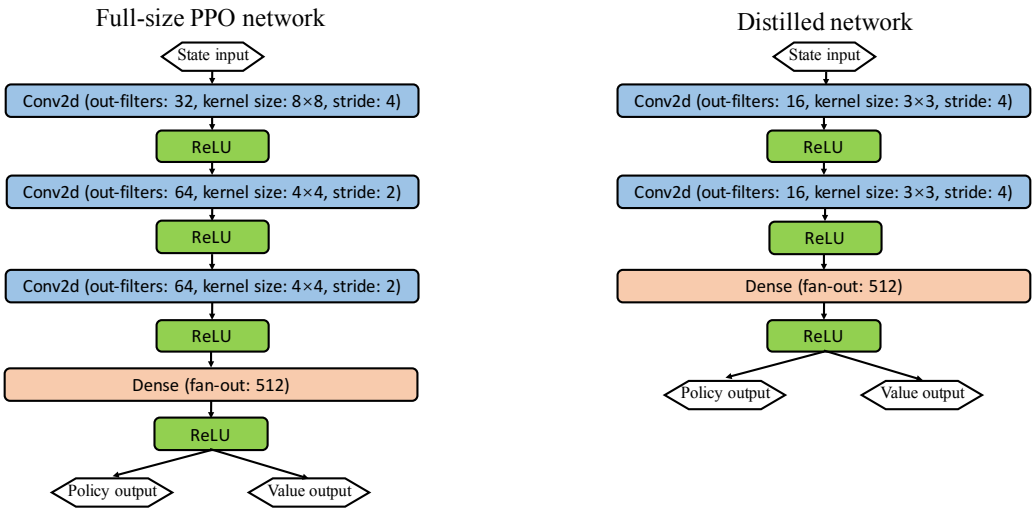


Figure 10: Architecture of the original PPO network (left) and the distilled network (right).

Table 4: Performance in 15 Atari benchmarks between the original PPO policy and our distilled policy.

Environment	Origin PPO policy	Distilled policy
Alien	1850	850
Boxing	94	7
Breakout	274	191
Centipede	4386	1701
Freeway	32	32
Gravitar	737	600
MsPacman	2096	1860
NameThisGame	6254	6354
RoadRunner	25076	26600
Robotank	5	13
Qbert	14293	12725
SpaceInvaders	942	1015
Tennis	-14	-10
TimePilot	4342	4400
Zaxxon	5008	3504

C EXPERIMENT DETAILS OF THE ATARI GAMES

This section provides the implementation details of the experiments on Atari games. Specifically, we first describe the training pipeline of the default policy. We then illustrate how the default policy is connected with MCTS algorithm to perform simulation.

Training default policy for MCTS To allow better overall performance, we used the Proximal Policy Gradient (PPO) (Schulman et al., 2017), one of the state-of-the-art on-policy reinforcement learning (RL) algorithms. We adopted the highest-starred third-party code of PPO on GitHub. The implementation uses the same hyper-parameters with the original paper. The architecture of the policy network is shown in Figure 10. The original PPO network is trained on 10 million frames for each task. To reduce computation count, we shrank the network using network distillation (Hinton et al., 2015). Specifically, it is a teacher-student training framework where the student (distilled) network mimics the output of the teacher network. Samples are collected by the PPO network with the ϵ-greedy strategy ($\epsilon = 0.1$). The student network optimizes its parameters to minimize the mean square error of the policy's logits as well as the value. Performance of the original PPO policy network as well as the distilled network is provided in Table 4.

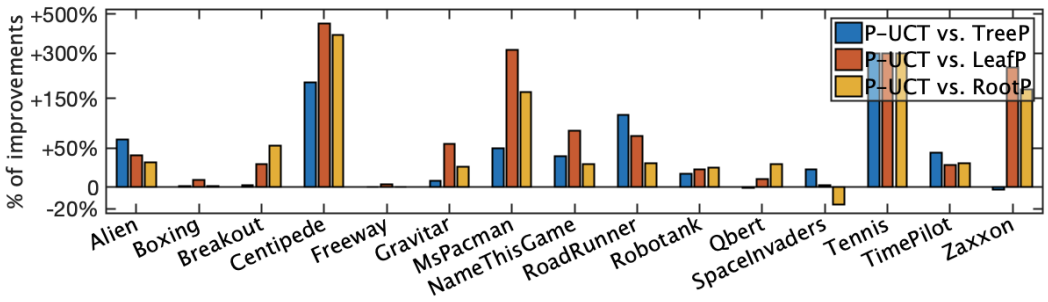


Figure 11: Relative performance between P-UCT and three baseline approaches on 15 Atari benchmarks. Relative performance is calculated according to the mean episode reward in 3 trials. The average percentile improvement of P-UCT on TreeP, LeafP, and RootP is 56%, 109%, and 83%, respectively.

MCTS simulation Both the policy output and the value output of the distilled network is used in the simulation phase. Particularly, if simulation is started from state s, rollout is performed using the policy network with an upper bound of 100 steps, where it results in state s'. If the environment does not terminate, the full return is compensated by the value function at state s'. Formally, the cumulative reward provided by the simulation is $R_{simu} = \sum_{i=0}^{99} \gamma^i r_i + \gamma^{100} V(s')$, where $V(s)$ denotes the value of state s. To reduce the variance of Monte Carlo sampling, value function $V(s)$ at state s is used additionally. The final return of the simulation phase is then $R = 0.5R_{simu} + 0.5V(s)$.

Hyperparameters and experiment details for P-UCT For all tree search based algorithms (i.e., P-UCT, TreeP, LeafP, and RootP), the maximum depth of the search tree is set to 100. The search width is limited by 20 and the maximum number of simulations is 128. The discount factor γ is set to 0.99 (note that the reported score is not discounted). When performing gameplays, a tree search subroutine is invoked to plan for the best action in each time step. The sub-routine iteratively constructs a search tree from its initial status with a root node only. Experiments are deployed on 4 Intel® Xeon® E5-2650 v4 CPUs and 8 NVIDIA® GeForce® RTX 2080 Ti GPUs. To minimize the speed fluctuation caused by the different workload on the machine, we ensure the total number of simulation slave workers is smaller than the total number of CPU cores, allowing each process to fully occupy one single core. The P-UCT is implemented with multiple processes, with an inter-process pipe between the master process and each slave process.

Hyperparameters and experiments for baseline algorithms Failing to find appropriate third-party packages for baseline algorithms (i.e., tree parallelization, leaf parallelization, and root parallelization), we built our implementation of them based on the corresponding papers. Building all algorithms in the same package additionally allows us to accurately conduct speed-tests as it eliminates other factors (e.g. different language) that may bias the result. Specifically, leaf parallelization is implemented with a master-slave structure: when the main process enters the simulation step, it assigns the task to all slave workers. When return from all workers is available, the master process performs backpropagation according to these statistics and begin a new rollout.

As suggested by Browne et al. (2012), tree parallelization is implemented using a decentralized structure, i.e., each worker performs rollouts on a shared search tree. At the selection step, each traversed node is added a fixed virtual loss $-r_{VL}$ to guarantee diversity of the tree search. When performing backpropagation, r_{VL} is again added to the traversed nodes. r_{VL} is chosen from 1.0 and 5.0 for each particular task. In other words, we ran TreeP with $r_{VL} = 1.0$ and $r_{VL} = 5.0$ for each task, and report the better result.

Root parallelization is implemented according to Chaslot et al. (2008). Similar to leaf parallelization, root parallelization consists of sub-processes that do not share information with each other. At the beginning of the tree search process, each sub-process is assigned several actions of the root node to query. They then perform sequential UCT rollouts until reaches a pre-defined step size. When all sub-processes complete the jobs, statistics from them is gathered by the main process, which then uses this information to choose the best action.