Dynamic Thinker: Optimizing Decision-Time Planning with Costly Compute

Kevin A. Wang
Brown University
kevin_a_wang@brown.edu

Jerry Xia
Brown University

Stephen Chung
University of Cambridge

Amy Greenwald
Brown University

ABSTRACT

Decision-time planning (DTP) agents use significant amounts of time and compute to search before taking an action. Such agents have been instrumental in achieving strong performance in domains like chess, go, and poker. Some DTP agents, like AlphaZero, are trained via reinforcement learning. However, the learning objectives that these agents optimize for do not typically include the cost of using time and compute. Instead, algorithm designers control the trade-off between performance and compute cost by tuning coarse-grained hyperparameters or writing hardcoded heuristics.

In this work, we introduce Dynamic Thinker, which is a modification of an existing state-of-the-art tree-search agent, Thinker. Dynamic Thinker is capable of optimizing its DTP behavior, under objective functions which account for the cost of computation. We design such objective functions for some toy environments, and show that Dynamic Thinker outperforms Thinker and AlphaZero.

Qualitatively, we find that Dynamic Thinker performs well by learning to use compute resources efficiently. We also highlight Dynamic Thinker's interesting emergent behavior, such as using more search in the start of the episode in one environment, and using more search near the end of the episode in another.

KEYWORDS

Reinforcement Learning, MCTS, Tree Search, Search, Decision-Time Planning, System 2 Thinking, metareasoning, rational metareasoning

1 INTRODUCTION

Reinforcement learning (RL) is concerned with producing agents that maximize some utility function. Typically, the utility of an agent for some environment is defined as the expected return of its policy – that is, the expected sum of discounted rewards that its policy earns [36, 39].

However, the field of rational metareasoning [28] argues that the utility of an agent can't be described just in terms of the policy that the agent implements, but is instead a function of the *behavior* of the agent – namely, that the time and resource cost of computation should be considered as part of the utility of the agent.

For example, it is easy to write a program that implements a Nash equilibrium policy for chess: brute-force search of the entire game tree will do the trick. However, such a program will be of no use to anyone, as it will take longer than the age of the universe to decide on most of its moves.

Many reinforcement learning agents use an insignificant and constant amount of compute per decision [21, 32], so optimizing for the return of the agent's policy is equivalent to optimizing for the behavior of the agent with compute costs taken into account.

Alternatively, an agent can spend a non-trivial amount of time and compute to decide on an action, a phase called **decision-time planning (DTP)** [39]. A prominent example of this is the AlphaZero family of agents [31, 35]. DTP has indeed proven beneficial in domains like chess, go [34], poker [5, 22], and mathematical reasoning [10, 17, 24, 44].

When building these agents, our goal is that of *limited rationality* from Russell and Wefald [28]: How can we optimize the behavior of decision-making agents, under an objective function that accounts for the cost of computation? In this paper, we specifically focus on DTP agents that use lookahead search in deterministic environments.

Broadly, there are two ways in which an algorithm designer can control the behavior of a DTP agent: first, by directly programming the computation of the agent, and second, by training parameters that control the computation of the agent. Generally, allowing more of the behavior to be controlled by learned parameters can result in better agents [38].

With the advent of deep learning, more and more of DTP behavior is controlled by learned parameters (neural network weights) rather than hand-crafted heuristics [1, 4, 12, 13, 19, 24, 30, 31, 37, 40].

AlphaZero decides which nodes to traverse during search using its action-selection heuristic (a hardcoded formula) [11], together with a learned policy network and value network. It acts by taking the action visited most often during its search phase, another hardcoded rule. In the original description of AlphaZero, the agent always used 800 steps of search during training, and 1/20th of the remaining time budget at test-time [35].

The Thinker algorithm [7] replaces AlphaZero's hardcoded rules with learned behavior, parameterized by neural network weights and trained through reinforcement learning. While Thinker achieves state-of-the-art performance in environments where search is useful, the number of states explored during search is still a hardcoded hyperparameter. Thus, we are unable to automatically train Thinker to find the optimal tradeoff between the better performance that can be obtained by investing resources in decision-time planning vs. its higher cost.

Our contributions are:

Proc. of the Adaptive and Learning Agents Workshop (ALA 2025), Avalos, Aydeniz, Müller, Mohammedalamen (eds.), May 19 – 20, 2025, Detroit, Michigan, USA, ala-workshop.github.io. 2025.

- We introduce **Dynamic Thinker**, by modifying Thinker with an additional action, which lets it stop searching and take a real action at any point.[1]
- We model the cost of compute in Dynamic Thinker's reward function, and train it via reinforcement learning on toy knapsack environments.
- We demonstrate that Dynamic Thinker outperforms Thinker, AlphaZero, and model-free RL in these settings, by using its search efficiently.
- We demonstrate that Dynamic Thinker displays non-trivial learned behavior, by using different amounts of search based on the characteristics of the state and environment.

In this paper, we are strictly concerned with the utility of (including computational cost of the behavior of) trained agents. Although offline training costs are also important, they are not our focus.

Examples of compute resources include wall time, CPU cycles, and energy consumption. For simplicity, in this paper we refer to all resources used while planning as "compute."

2 MODELING COMPUTE COSTS

In this section, we first present the typical formulation of MDPs and policies. Then, we present a description of Thinker as a decision-making agent in a metalevel MDP.

2.1 Traditional MDPs and Policies

When building an RL agent, we typically model the environment as a **Markov decision process (MDP)** or a variant thereof,[2] and we model the agent as a **policy**.

We write $\triangle(\mathcal{X})$ to denote the set of probability distributions over an arbitrary (finite) set $\mathcal{X}$.

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $P : \mathcal{S} \times \mathcal{A} \to \triangle(\mathcal{S})$ is a probabilistic transition function, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, $\gamma \in [0, 1]$ is a discount factor, and μ is an distribution over initial states.

An MDP starts at time step $t = 0$ with the agent in a state s_0 randomly drawn from μ. It then samples an action a_0 from $\pi(s_0)$. The environment then draws a new state s_1 from $P(s_0, a_0)$, and the agent receives reward $r_1 = R(s_0, a_0)$. This process then repeats.

A **policy** $\pi : \mathcal{S} \to \triangle(\mathcal{A})$ is a function that maps each state to a distribution over actions. Every policy π in an MDP induces an expected discounted return $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi\right]$. An optimal policy is one that maximizes this return.

We make an important distinction between computational agents (which we refer to in this paper as **agents**) and policies. A computational agent is a computer program that implements a policy. These programs take a state as input, perform computation, and output an action.

[1]We use the term **dynamic** as it was introduced for checkers/chess in Markovitch and Sella [20]: Compute usage can be static (amount of compute per decision is determined before the episode begins), semi-dynamic (amount of compute for a given decision is determined before planning for that decision), or dynamic (amount of compute may be conditional on the state of the planning process itself).

[2]Other variants of the MDP include the partially observable MDP (POMDP), the Decentralized POMDP (Dec-POMDP), the Markov game [33], and the factored-observation stochastic game [18].

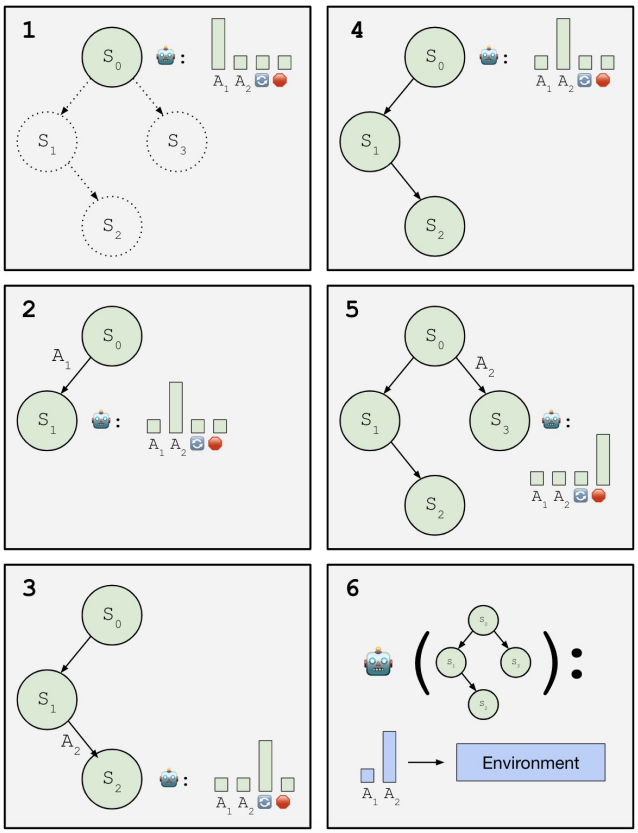


Figure 1: Overview of Dynamic Thinker. Step 5 is unique to Dynamic Thinker. The other steps describe both Thinker and Dynamic Thinker.
1. The planning agent starts planning at the current game state, which becomes the root of the search tree.
2. To plan, the agent takes virtual search actions to explore the search tree, conditioned on the search state.
3. At any time during the search, the agent can return to the root node to explore another branch of the tree.
4. The virtual search is informative, because the agent explores states to determine the best action.
5. Dynamic Thinker can take a stop search action, which it chooses when using additional compute provides little marginal benefit.
6. When the virtual search ends, the agent takes a real action conditioned on the search tree explored during virtual search.

A typical policy doesn't model an agent's decision-making time. A policy is an excellent model for the types of agents that are typically studied in RL, namely "fast" agents that do not use significant amounts of time or compute to make decisions. For instance, RL agents may simply query a policy table or a Q-value table (or their respective neural network counterparts). Then, they return either the result of the query, or the result of extremely simple computation on the results, such as taking the argmax. This process takes a small, constant amount of time.

Now, consider a hypothesis class of agents in which some agents use more compute than others to make decisions. We call such hypothesis classes **compute-varying**. Two agents π and ρ may implement the same compute-independent policy; that is, they may choose the same actions at all states. However, they may also reach these decisions using different amounts of compute. If ρ takes 1 year of computation to reach each decision while π only takes 1 second, π is likely preferable. Thus, policies are insufficient models of agents when our hypothesis class is compute-varying.

When building a search agent, such as AlphaZero for chess, the hypothesis class is typically compute-varying: the amount of search used at each state can be tuned. To date, engineers have largely controlled this by designing hand-crafted heuristics and setting hyperparameters [2]. However, as search algorithms are becoming increasingly general and governed by learned parameters, it seems reasonable to also incorporate learned parameters that dictate search time, and cost functions that evaluate resource usage. Agents would then learn the right balance between the benefits of resource use vs. its costs.

2.2 Thinker Augmented MDP

Thankfully, we can still model the problem as an MDP – one in which the state represents not only the state of the environment, but also the state of the internal computation of the agent, in which actions represent performing computation, and rewards include the cost of compute. Then, we train a "fast" RL agent to maximize returns in the MDP. This is referred to in metareasoning as a **meta-level MDP** [6, 28]

Luckily, **Thinker** [7] is already described in this way – termed an **augmented MDP** – but without a cost of compute[3].

Thinker is a tree search algorithm that is heavily controlled by learned parameters. Because of its ability to learn parameters that are hardcoded in other tree search algorithms, it can be thought of as a generalized version of other algorithms like AlphaZero.

In Thinker, the search tree is considered external to the agent. An MDP is transformed into an augmented MDP, in which the agent can not only take actions in the original MDP, but it can also take actions to explore nodes in the search tree, which we call **planning actions**. Both types of actions are chosen conditionally on the state of the current search tree.

We refer to the agent in the augmented MDP as the **internal agent**. We refer to the induced agent in the original MDP as the **holistic agent**. The internal agent itself is "fast", yet by acting in its search-tree environment, it performs planning. That is, the internal agent uses a small, fixed amount of compute resources per decision, yet it spends a large amount of compute between real actions in the real environment (i.e., by taking multiple planning actions before taking a real action). This formulation of planning is elegant and biologically plausible [16].

We denote an Augmented MDP as

$$\mathcal{M}^{aug} = (\mathcal{S}^{aug}, \mathcal{A}^{aug}, P^{aug}, R^{aug}, \gamma^{aug}, \mu^{aug}) .$$

An Augmented MDP is transformed from a base MDP as $\mathcal{M}^{aug} = f(\mathcal{M}, \theta^V, \theta^P)$, where $\mathcal{M}$ is the base MDP, θ^V are the parameters of a value network, and θ^P are the parameters of a policy network.

- $\mathcal{S}^{aug}$: Each state $(s, T) \in \mathcal{S}^{aug}$ is composed of a real MDP state $s \in \mathcal{S}$, and the internal state of a search tree, denoted T.
- $\mathcal{A}^{aug}$: There are two kinds of actions in the augmented MDP: $\mathcal{A}^{aug} = \mathcal{A} \cup \mathcal{A}'$. The set $\mathcal{A}$ is the set of real MDP actions, and the set $\mathcal{A}'$ is the set of planning actions. An planning action can be thought of as traversing the search tree.
- P^{aug} and R^{aug}: After the Internal Agent takes an planning action $a \in \mathcal{A}'$ at a state (s, T), the environment transitions to a new state (s, T') and receives 0 reward[4], where $T' = traverse(T, a)$. After the Internal Agent takes a real action $a \in \mathcal{A}$ at a state (s, T), the environment transitions to a new state (s', T_0) and receives reward r, where $s' = P(s, a)$, $r = R(s)$, and T_0 is an empty search tree.
- γ^{aug} may be kept equal to γ or rescaled.
- $\mu^{aug}((s, T_0)) = \mu(s)$, where $\mu^{aug}((s, T_0))$ is the probability of (s, T_0) under μ^{aug}, and $\mu(s)$ is the probability of s under μ. $\mu^{aug}((s, T)) = 0$, where $T \neq T_0$.

A search tree node is a tuple $(s, v, p, u, children)$, where

- $s \in \mathcal{S}$ is the state represented by the node,
- v is the value of s according to θ^{VP} (called the base value of s),
- p is the policy at s according to θ^{VP} (called the base policy of s),
- $children$ is an array, where $children[i]$ points to the search tree node corresponding to the state $s' = P(s, a_i)$,
- u, called **hints**, contains statistics such as the mean and max base value of all visited descendant nodes, the number of times the node has been visited, etc.

A search tree T is composed of a rooted tree of search tree nodes, and the node that the agent is currently at.

At a state (s, T_0), the Internal Agent is at the root node of the tree, which corresponds to the real state s. Taking the action $a'_i \in \mathcal{A}'$, results in $T' = traverse(T, a'_i)$, and corresponds to descending the tree by moving to T_0.children$[a_i]$. If the node is not already in the tree, it is created, by calling the simulator to get $s' = P(s, a)$, running the value network and policy network to produce v' and p', and initializing $children'$ and u'. The node $(s', v', p', children', u')$ is then added to the tree, and the hints of all ancestor nodes are updated. The Internal Agent also has the option to take the "reset" action $a'_* \in \mathcal{A}'$ which moves it back to the root node.

As a succinct summarization of the tree, the Internal Agent observes the search tree node it is currently at, and of the root node.

A Thinker agent thus plans and acts using the following algorithm:

(1) We observe the initial MDP state s_0. Initialize an augmented MDP state $(s, T) := (s, T_0)$
(2) query the Internal Agent for its search action a, conditioned on the observation of (s, T)

[3] Although in environments where rewards are non-negative, the discount factor imposes a cost of compute.

[4] In practice, we design rewards to shape the agent to search well, but they are only for shaping and are decayed to 0 over the course of training.

(3) if a is a planning action, then update the augmented MDP state by traversing and updating the search tree ($T = traverse(T, a)$). Repeat from Step 2.
(4) otherwise, a is a real action, so take action a in the real environment. This will lead to a new real state s', so we transition the augmented MDP state to (s', T_0). Repeat from Step 2.

2.3 Compute Costs

We can model the cost of compute as a constant penalty for each planning step taken. That is, when the agent takes a planning action $a \in \mathcal{A}'$, the agent receives reward λ.

We'll call the rewards (returns) without the penalty **compute-independent rewards (returns)**, and the rewards (returns) including the penalty the **compute-dependent rewards (returns)**.

We can also model AlphaZero using the augmented MDP, and calculate its compute-dependent returns.

3 DYNAMIC THINKER

3.1 Dynamic Thinker

Thinker uses a fixed number of planning actions before it takes a real action. The number of planning actions, denoted k, is a hyperparameter.

We instead modify Thinker so that its internal agent has an extra action in its action space: "finish search." If this action is taken, then the agent's next action is a real action. We call this modified algorithm **Dynamic Thinker**. We will refer to the original, unmodified Thinker algorithm as **Fixed-k Thinker**.

Formally, fixed Thinker must take a real action $a \in \mathcal{A}$ on every step i where $i \equiv 0 \mod k$, and must take a planning action $a \in \mathcal{A}'$ otherwise. Dynamic Thinker can use any action $a \in \mathcal{A}^{aug}$ at any step.

Since Dynamic Thinker can use a variable number of planning actions for each real action, the set of possible holistic agents is a compute-varying hypothesis class. Further, the optimization of the holistic agent can then naturally be performed through deep reinforcement learning, by simply training the internal agent to maximize compute-dependent rewards. Importantly, it is dynamic: it does not need to pre-allocate the amount of compute used for planning, but can make decisions (such as finishing search) during search itself.

4 EXPERIMENTS

We have argued that our objective function should contain a cost for compute usage. However, for this idea to be of practical use, we must be able to train compute-varying hypothesis classes of agents. Towards this end, we implemented Dynamic Thinker and trained it on a variety of environments and compute costs. We implemented Dynamic Thinker by modifying Thinker using its open-source code [7].

Although modifying Thinker into Dynamic Thinker is a small change, one may be afraid that it causes learning to be much harder. Fortunately, our experiments show that Dynamic Thinker is able to learn well in small knapsack environments: It not only matches the best baseline in terms of compute-dependent returns for each penalty, but it also outperforms it in some cases (showing that there is benefit to *dynamically* spending compute). Unexpectedly, in some environments, with certain search penalties, Dynamic Thinker also outperforms the best baselines in terms of the *raw, compute-independent returns*. These positive results motivate the continued investigation of learned, dynamic DTP agents.

4.1 Knapsack Environments

We performed experiments on three different versions of the classic knapsack problem. In the problem, you are given a set of N items and their respective weights $w_i \in \mathbb{N}$ and values $v_i \in \mathbb{N}$, as well as a knapsack that can carry up to M pounds, and your goal is to pick some subset of the items with maximum total value, with the constraint that the sum of the weights does not exceed the knapsack's capacity. In our environments, we use M=20 and N=9.

We formulate the problem as a sequential decision-making problem by defining an MDP where the agent picks items to add to the knapsack sequentially. The agent starts with an empty knapsack, and at each state, can pick one item to add to the knapsack. An episode terminates when no further items can be added to the knapsack. The agent receives a reward of v_i when it adds item i to the knapsack. If the selected item is illegal (either it has already been used, or it would exceed the capacity of the knapsack), then the agent receives a negative reward and the episode continues. These rewards are normalized by dividing by the maximum achievable total weight for the instance, so that the optimal policy will receive a cumulative reward of 1.

In these experiments, we choose the number of internal agent actions (both search tree traversals and real actions) as our unit of compute. We model planning cost as a constant negative reward for each internal agent action. Each time the internal agent takes an action—that is, it traverses a node in its search tree, or it decides on a real action—it incurs a λ reward.

The three versions are:

- Knapsack: Standard environment.
- Knapsack Bonus (1) for Optimal Solve: On the last step of the episode, if the agent has achieved an optimal knapsack value, then it receives an additional reward of 1.
- Knapsack Bonus (2) for Optimal Solve: Same as above, but with a reward of 2.

Formal Environment Description: Let $\mathcal{M}$ be the set of items and let w_i be the reward for legally adding item i to the knapsack. We model the knapsack problem as an MDP:

- The state space $\mathcal{S} = 2^{\mathcal{M}}$ is the set of all possible sets of items in the knapsack.
- The action space $\mathcal{A} = \mathcal{M}$ includes an action for adding each item to the knapsack.
- At a state S and action i, the transition function places item i into the knapsack, if the item can be legally added: $S \cup \{i\}$. Otherwise, the knapsack is unchanged: S.
- The reward function returns w_i if item i can be legally placed into the knapsack, or c if i is illegal.

We model the cost of compute as a penalty of λ added to the reward of the metalevel MDP for each planning step taken.

We do not sort the items in the representation, and illegal actions are penalized, not masked. These difficulties mean that learning an

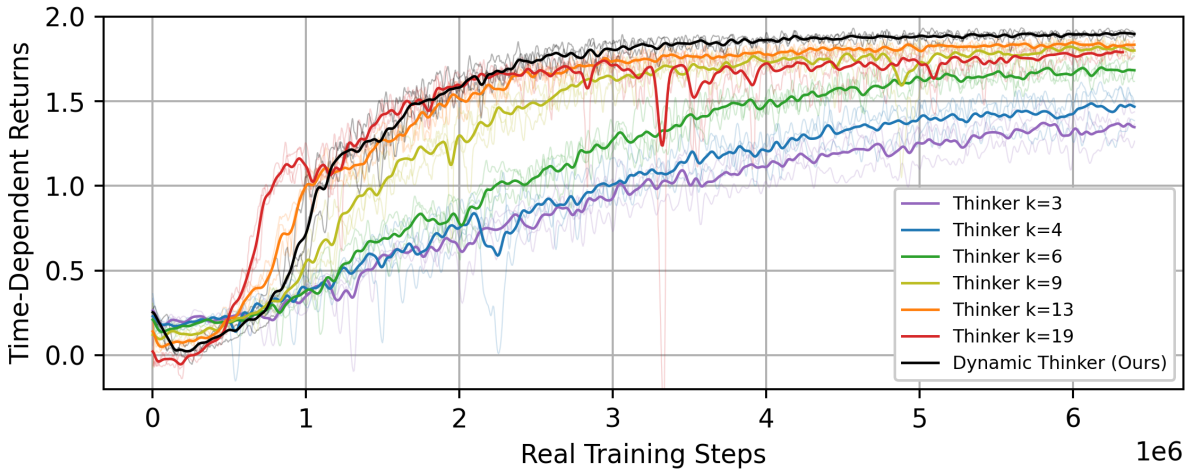


Figure 2: Compute-dependent returns during training on Knapsack Bonus (1) environment, when each agent receives a penalty of $\lambda = -0.002$ for each search step they use. The y-axis shows the cumulative compute-dependent rewards ($\mathcal{R}$) per episode. The colored lines are Fixed-k Thinker, where the agent is forced to use k steps of search for each real step. Dynamic Thinker learns to outperform Fixed-k Thinker for any k.

optimal policy for this environment is not trivial, despite its small size. Learned tree search can be very useful: an agent can easily learn to use even 1-step lookahead to avoid illegal actions. However, doing more search is not always useful: for example, if search finds a set of items that results in maximum capacity (10) by taking the items with highest value-to-weight ratio, then no further search is necessary.

Setup: We trained Dynamic Thinker on each environment, with penalties ranging from 0 to -0.008 per search step. We run all methods for 6.4 million real steps. We used 1 GPU and 10 to 12 CPUs for training. We compare Dynamic Thinker with the following baselines:

(1) Unmodified, Fixed-k Thinker with search length per step set to $k \in \{3, 4, 6, 9, 13, 19\}$.
(2) AlphaZero-style Monte Carlo tree search with various amounts of fixed search per decision ($k \in \{12, 25, 50\}$).
(3) Model-free actor-critic RL agent (IMPALA) ($k = 1$) (this is the same actor-critic algorithm as the internal agent).

Although we don't vary the search penalty of the baselines[5], we can calculate what its compute-dependent returns would be for any value of λ by subtracting $Nk\lambda$ from the compute-independent returns, where N is the number of real steps in the episode.

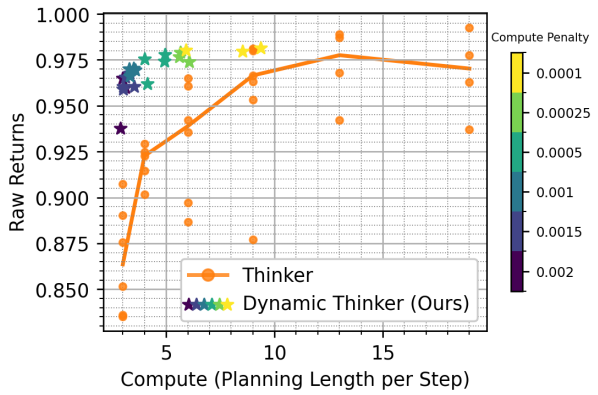
Results: Table 1 shows the performance of Dynamic Thinker compared to Fixed-k Thinker on each environment and penalty. For each environment and penalty, we calculate the compute-dependent returns of Fixed-k Thinker for that penalty for each k, and pick the k with the highest compute-dependent returns, as the best baseline. We note the compute-dependent returns (Best Thinker Score) and the value of k (Best Thinker Search Length) in the table. All values are averaged over multiple runs. We indicate superior performance in the table by bolding the value.

We note that Dynamic Thinker outperforms Fixed-k Thinker in many scenarios. In the Knapsack environment, Dynamic Thinker achieves more compute-dependent returns than any Fixed-k Thinker, for all tested search penalties. In Knapsack Bonus (1), Dynamic Thinker achieves more compute-dependent returns than any Fixed-k Thinker, for all search penalties except the very large 0.008, where it performs very little search and achieves very low returns. In Knapsack Bonus (2), Dynamic Thinker outperforms all Fixed-k Thinker when the penalty is moderate, but it underperforms when the penalty is very small or very large.

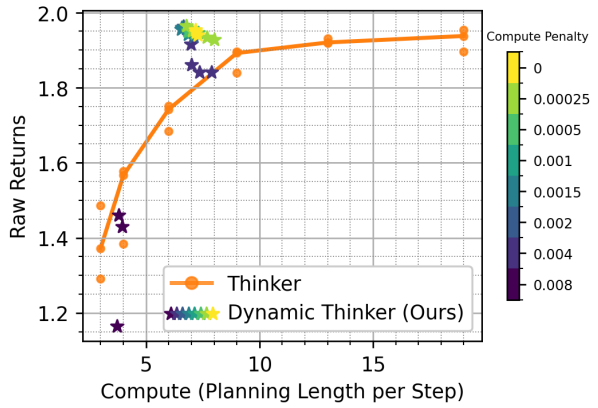
Figure 2 shows the training curves corresponding to one row of the table (Knapsack Bonus (1), $\lambda = 0.002$). In this environment, among all Fixed-k Thinker agents, $k = 13$ makes the most optimal trade-off between raw returns and amount of search. Ideally, we would like Dynamic Thinker to match this performance automatically via training, without having to choose a hyperparameter like k. As the figure shows, Dynamic Thinker not only matches this performance, it significantly exceeds it.

Figure 3 plots raw (compute-independent) performance on the y-axis and average amount of compute on the x-axis for Dynamic Thinker and Fixed-k Thinker. (Additional baselines are shown in Figure 4.) Each Fixed-k Thinker run is an orange dot, and each Dynamic Thinker run is a colored star. As the value of k increases for Fixed-k Thinker, we observe that raw performance increases, forming a Pareto curve of the compute/returns trade-off (orange line). If a Dynamic Thinker run is up and to the left of the Pareto curve, then it is using less compute to achieve at least the same raw performance as some Fixed-k Thinker, and it is achieving higher performance using at most the same compute as some Fixed-k Thinker run.

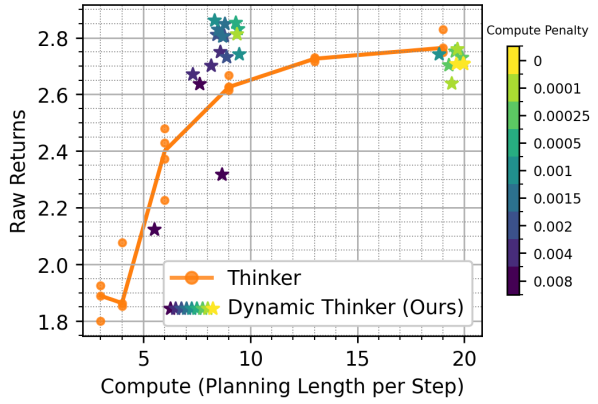
[5]Different penalty values shouldn't affect the baselines, since they use a fixed amount of search. Only when the penalty is extremely high would a baseline's behavior change, since it could benefit from ending episodes as early as possible to avoid additional search. For these experiments, we use a penalty of 0.001 for the baselines, although a penalty of 0 yields the same results.



(a) Knapsack



(b) Knapsack Bonus (1)



(c) Knapsack Bonus (2)

Figure 3: Raw (Compute-Independent) Returns plotted against Compute (Number of Planning Steps per Real Step) for Fixed and Dynamic Thinker

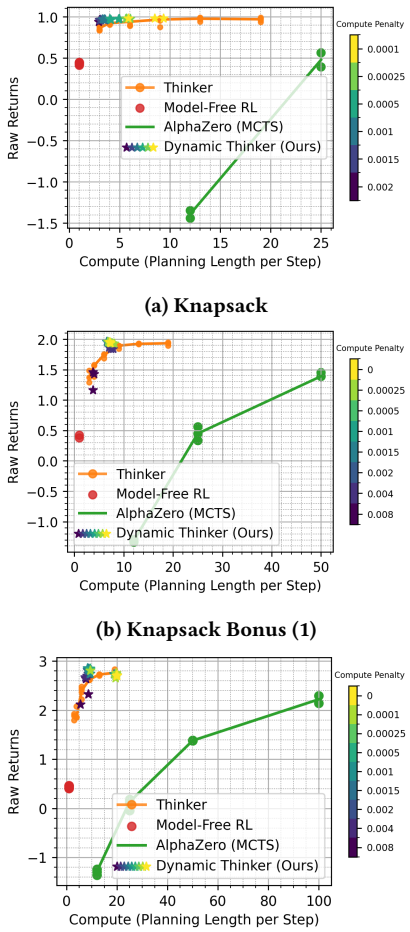
Search Penalty λ	Dynamic Thinker Returns (Ours)	Best Thinker Returns	Dynamic Thinker Search Length	Best Thinker Search Length
Knapsack				
-0.0001	**0.981**	0.962	7.9	19
-0.00025	**0.971**	0.956	5.7	9
-0.0005	**0.960**	0.948	4.5	9
-0.001	**0.954**	0.932	3.4	9
-0.0015	**0.944**	0.916	3.2	9
-0.002	**0.933**	0.900	3.0	9
Knapsack Bonus (1)				
0	**1.938**	1.933	7.2	13
-0.00025	**1.936**	1.921	7.4	13
-0.0005	**1.930**	1.909	7.1	13
-0.001	**1.921**	1.886	7.2	13
-0.0015	**1.909**	1.862	6.9	13
-0.002	**1.899**	1.839	6.9	13
-0.004	**1.767**	1.745	7.3	13
-0.008	1.212	**1.600**	3.8	9
Knapsack Bonus (2)				
0	2.672	**2.767**	19.8	19
-0.0001	2.718	**2.760**	17.1	19
-0.00025	2.717	**2.749**	19.5	19
-0.0005	**2.789**	2.731	9.8	19
-0.001	**2.731**	2.696	14.0	19
-0.0015	**2.768**	2.669	8.7	13
-0.002	**2.710**	2.643	8.8	13
-0.004	**2.611**	2.538	7.9	13
-0.008	2.108	**2.389**	7.3	9

Table 1: Experimental Results. For each row, Best Thinker is the Fixed-k Thinker with highest average returns. All values are averages over multiple runs.

We would expect that by decreasing λ, the amount of planning increases and the raw performance of Dynamic Thinker increases. In the Knapsack environment, this is true, and the Dynamic Thinker runs form their own Pareto curve. However, in the environments with bonuses for optimal solves, Dynamic Thinker tends to converge to a certain amount of planning steps, unless the penalty is set to a very high or very low amount. Surprisingly, when the penalty is set very low in these settings, Dynamic Thinker will perform more search, but its raw performance will decrease. This suggests that the search behavior that Dynamic Thinker converges to otherwise is close to optimal. Indeed, we unexpectedly observe that Dynamic Thinker achieves higher *raw* returns for these penalty values than even the best Fixed-k Thinker.

4.2 Qualitative Observations

So far, we have seen that Dynamic Thinker can outperform Fixed-k Thinker, achieving greater compute-dependent returns by efficiently trading off between raw returns and amount of compute. In

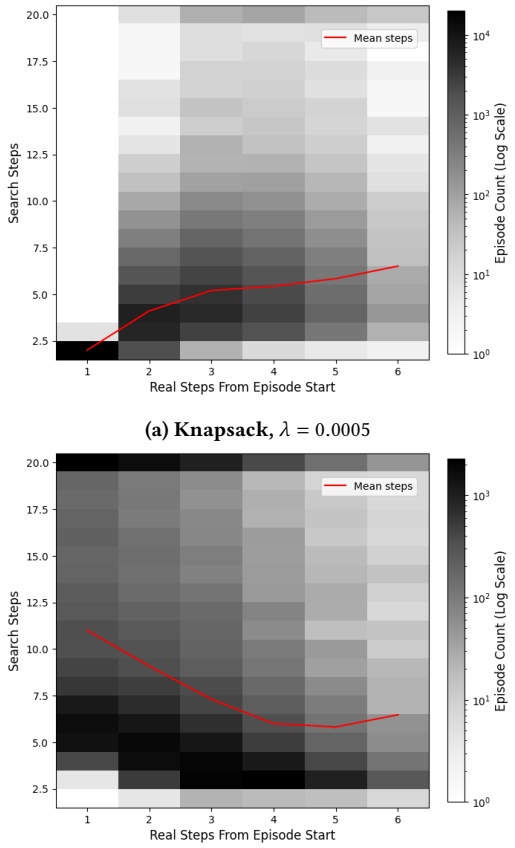


(a) Knapsack

(b) Knapsack Bonus (1)

(c) Knapsack Bonus (2)

Figure 4: Raw (Compute-Independent) Returns plotted against Compute (Number of Planning Steps per Real Step) for all methods



(a) Knapsack, $\lambda = 0.0005$

(b) Knapsack Bonus (2), $\lambda = 0.0015$

Figure 5: Heat map and mean of search steps plotted as a function of step number.

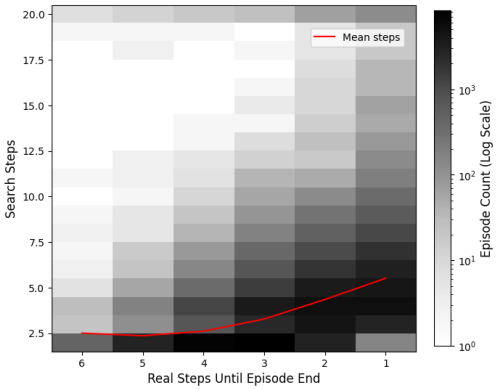
this section, we seek to gain some insight into how it does this, by examining its behavior.

Figure 5 shows how the amount of search performed changes over the course of an episode. Note that the episodes are variable length but often end after just 3 or 4 steps. Note that the search behavior is drastically different between Figure 5a and Figure 5b. This suggests that Dynamic Thinker does not just happen to use compute well, but that it learns how to use compute well in different environments. Specifically, in Knapsack with $\lambda = -0.0005$, Dynamic Thinker learns to perform *no* search on the first step, and to use more as the episode progresses. On the other hand, in the Knapsack Bonus (2) environment, one misstep on the first action can preclude the agent from earning the important bonus, so it often performs the maximum amount of search (20 steps) on the first step and performs less as the episode continues, since it may have already found the optimal solution.

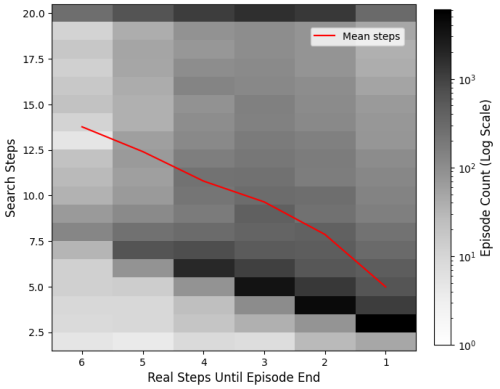
Figure 6 shows the amount of search used as the agent gets closer to the end of the episode. As before, note that in Knapsack Bonus (2), where getting a precise answer is important, the agent often performs the maximum amount of search. However, there is a general trend of using less search towards the end of the episode. This is expected, since when the episode end is imminent, the agent should be confident in its next few actions, especially if it has already searched and found the optimal solution. However, in Knapsack, the agent performs *more* search as the episode end nears. Perhaps this is because the remaining optimization problem becomes tractable near the end of an episode, as the number of possible actions shrinks significantly, and so each marginal search step yields significant gains. Regardless, it is clear that Dynamic Thinker learns behavior more complex than just using a fixed amount of search in each step, and that it is capable of learning drastically different search behavior for different environments.

5 RELATED WORKS

The mismatch in RL between MDPs and the time-dependent real world has been noted before[26, 28, 41, 42]. In addition, any researcher of decision-time planning algorithms knows that compute



(a) Knapsack, $\lambda = 0.0005$



(b) Knapsack Bonus (2), $\lambda = 0.0015$

Figure 6: Heat map and mean of search steps plotted as a function of step number until the end.

costs matter, at least implicitly. However, the aspect in which search algorithms use more or less time to search is typically (A) controlled by hand-crafted heuristics, not learned parameters or (B) simplistic and either static or semi-dynamic, not dynamic.

Other metareasoning techniques have been developed that use deep learning and MDPs [6, 29], and that concern lookahead search [23, 28], but generally do not condition neural-network-controlled search behavior on rich search-tree information.

In this paper, we base our experiments and discussions on the existing search algorithm Thinker [7]. MCTSNets [13] are similar to Thinker and could potentially be modified in the same way, but did not result in empirical state-of-the-art performance. Imagination-based Planner [25] is similar to Thinker, and includes a penalty for computation (termed a **resource cost**), but also did not result in state-of-the-art performance.

Many previous works have explored using less compute by training neural nets: [3, 14, 15]. In particular, Rosenberg et al. [27] explores adaptively learning a horizon length for planning.

Simple, extremely general algorithms such as DRC [9, 12] or autoregressive large language models (LLMs) [24, 43] could also be modified into dynamic, learned DTP agents. Research on LLMs[8, 19] are similar to this work, in that they seek to use inference-time compute efficiently. However, they either do not rigorously define a goal or utility to maximize[19], or they use a more complex, hand-coded system to allocate compute, instead of allowing an RL agent to learn when and where to use marginal compute by itself [8].

6 CONCLUSION

In this work, we addressed the critical but often overlooked issue that decision-time planning comes with real computational costs that should be factored into an agent's objective function. We introduced Dynamic Thinker, which extends the Thinker algorithm by allowing the agent to adaptively control how much search it performs at decision time, based on the specific context of each decision.

Our key contributions include:

(1) Modifying Thinker to create Dynamic Thinker, enabling learned, dynamic control of search depth through the addition of a "stop search" action.
(2) Demonstrating that Dynamic Thinker outperforms Thinker and AlphaZero in knapsack environments when optimizing for an objective that accounts for computational costs.
(3) Showing that Dynamic Thinker learns interesting emergent behaviors, such as varying its search depth based on the stage of the episode and the characteristics of the environment.

These results demonstrate that incorporating compute costs into the reward function leads to more efficient agents that can make reasonable trade-offs between performance and compute usage. Dynamic Thinker learns to search more when the marginal benefit of search is high, and less when further search provides diminishing returns or when the outcome is already determined. Our work represents a step toward the goal of limited rationality in RL agents: optimizing the behavior of decision-making agents under objective functions that account for the cost of computation. By making the cost of computation explicit in the objective function and allowing agents to learn how to manage this cost, we can develop more efficient and practical RL systems for real-world applications.

REFERENCES

[1] Thomas William Anthony. 2021. *Expert iteration.* Doctoral. UCL (University College London). https://discovery.ucl.ac.uk/id/eprint/10123580/ Conference Name: UCL (University College London) Meeting Name: UCL (University College London) Publication Title: Doctoral thesis, UCL (University College London).

[2] Hendrik Baier and Mark H. M. Winands. 2016. Time Management for Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 3 (Sept. 2016), 301–314. https://doi.org/10.1109/TCIAIG.2015.2443123

[3] Andrea Banino, Jan Balaguer, and Charles Blundell. 2021. PonderNet: Learning to Ponder. https://doi.org/10.48550/arXiv.2107.05407 arXiv:2107.05407 [cs].

[4] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. 2020. Combining Deep Reinforcement Learning and Search for Imperfect-Information Games. http://arxiv.org/abs/2007.13544 arXiv:2007.13544 [cs].

[5] Noam Brown and Tuomas Sandholm. 2019. Superhuman AI for multiplayer poker. *Science* 365, 6456 (Aug. 2019), 885–890. https://doi.org/10.1126/science.aay2400 Publisher: American Association for the Advancement of Science.

[6] Matthew Budd, Bruno Lacerda, and Nick Hawes. 2024. Stop! planner time: metareasoning for probabilistic planning using learned performance profiles. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24, Vol. 38).* AAAI Press, 20053–20060. https://doi.org/10.1609/aaai.v38i18.29983

[7] Stephen Chung, Ivan Anokhin, and David Krueger. 2023. Thinker: Learning to Plan and Act. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 22896–22933. https://proceedings.neurips.cc/paper_files/paper/2023/hash/4761fab863f0900d90cf601fce6d5155-Abstract-Conference.html

[8] Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. 2024. Learning How Hard to Think: Input-Adaptive Allocation of LM Computation. http://arxiv.org/abs/2410.04707 arXiv:2410.04707 [cs].

[9] Adrià Garriga-Alonso, Mohammad Taufeeque, and Adam Gleave. 2024. Planning behavior in a recurrent neural network that plays Sokoban. https://doi.org/10.48550/arXiv.2407.15421 arXiv:2407.15421 [cs] version: 1.

[10] Google Deepmind. 2024. AI achieves silver-medal standard solving International Mathematical Olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

[11] Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Remi Munos. 2020. Monte-Carlo Tree Search as Regularized Policy Optimization. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 3769–3778. https://proceedings.mlr.press/v119/grill20a.html ISSN: 2640-3498.

[12] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sebastien Racaniere, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy Lillicrap. 2019. An Investigation of Model-Free Planning. In *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 2464–2473. https://proceedings.mlr.press/v97/guez19a.html ISSN: 2640-3498.

[13] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. 2018. Learning to Search with MCTSnets. https://doi.org/10.48550/arXiv.1802.04697 arXiv:1802.04697 [cs, stat].

[14] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. 2019. Classification with Costly Features Using Deep Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (July 2019), 3959–3966. https://doi.org/10.1609/aaai.v33i01.33013959 Number: 01.

[15] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. 2024. Classification with costly features in hierarchical deep sets. *Machine Learning* (May 2024). https://doi.org/10.1007/s10994-024-06565-4

[16] Kristopher T. Jensen, Guillaume Hennequin, and Marcelo G. Mattar. 2024. A recurrent network model of planning explains hippocampal replay and human behavior. *Nature Neuroscience* 27, 7 (July 2024), 1340–1348. https://doi.org/10.1038/s41593-024-01675-7 Publisher: Nature Publishing Group.

[17] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. http://arxiv.org/abs/2205.11916 arXiv:2205.11916 [cs].

[18] Vojtěch Kovařík, Martin Schmid, Neil Burch, Michael Bowling, and Viliam Lisý. 2021. Rethinking Formal Models of Partially Observable Multiagent Decision Making. http://arxiv.org/abs/1906.11110 arXiv:1906.11110 [cs].

[19] Lucas Lehnert, Sainbayar Sukhbaatar, DiJia Su, Qinqing Zheng, Paul Mcvay, Michael Rabbat, and Yuandong Tian. 2024. Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping. https://doi.org/10.48550/arXiv.2402.14083 arXiv:2402.14083 [cs].

[20] Shaul Markovitch and Yaron Sella. 1996. Learning of Resource Allocation Strategies for Game Playing. *Computational Intelligence* 12, 1 (1996), 88–105. https://doi.org/10.1111/j.1467-8640.1996.tb00254.x _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8640.1996.tb00254.x.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. https://doi.org/10.48550/arXiv.1312.5602 arXiv:1312.5602 [cs].

[22] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. 2017. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science* 356, 6337 (May 2017), 508–513. https://doi.org/10.1126/science.aam6960

[23] Dylan O'Ceallaigh and Wheeler Ruml. 2015. Metareasoning in Real-Time Heuristic Search. *Proceedings of the International Symposium on Combinatorial Search* 6, 1 (2015), 87–95. https://doi.org/10.1609/socs.v6i1.18362 Number: 1.

[24] OpenAI. [n.d.]. Introducing OpenAI o1. https://openai.com/index/introducing-openai-o1-preview/

[25] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. 2017. Learning model-based planning from scratch. https://doi.org/10.48550/arXiv.1707.06170 arXiv:1707.06170 [cs].

[26] Simon Ramstedt and Christopher Pal. 2019. Real-Time Reinforcement Learning. https://doi.org/10.48550/arXiv.1911.04448 arXiv:1911.04448 [cs, stat].

[27] Aviv Rosenberg, Assaf Hallak, Shie Mannor, Gal Chechik, and Gal Dalal. 2023. Planning and Learning with Adaptive Lookahead. https://doi.org/10.48550/arXiv.2201.12403 arXiv:2201.12403 [cs].

[28] Stuart Russell and Eric H. Wefald. 2003. *Do the Right Thing: Studies in Limited Rationality*. The MIT Press. https://doi.org/10.7551/mitpress/2474.001.0001

[29] C. Nicolò De Sabbata, Theodore R. Sumers, and Thomas L. Griffiths. 2024. Rational Metareasoning for Large Language Models. https://doi.org/10.48550/arXiv.2410.05563 arXiv:2410.05563 [cs] version: 2.

[30] Martin Schmid, Matej Moravčík, Neil Burch, Rudolf Kadlec, Josh Davidson, Kevin Waugh, Nolan Bard, Finbarr Timbers, Marc Lanctot, G. Zacharias Holland, Elnaz Davoodi, Alden Christianson, and Michael Bowling. 2023. Student of Games: A unified learning algorithm for both perfect and imperfect information games. *Science Advances* 9, 46 (Nov. 2023), eadg3256. https://doi.org/10.1126/sciadv.adg3256 Publisher: American Association for the Advancement of Science.

[31] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (Dec. 2020), 604–609. https://doi.org/10.1038/s41586-020-03051-4 Number: 7839 Publisher: Nature Publishing Group.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. https://doi.org/10.48550/arXiv.1707.06347 arXiv:1707.06347 [cs].

[33] L. S. Shapley. 1953. Stochastic Games*. *Proceedings of the National Academy of Sciences* 39, 10 (Oct. 1953), 1095–1100. https://doi.org/10.1073/pnas.39.10.1095 Publisher: Proceedings of the National Academy of Sciences.

[34] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (Jan. 2016), 484–489. https://doi.org/10.1038/nature16961 Publisher: Nature Publishing Group.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (Dec. 2018), 1140–1144. https://doi.org/10.1126/science.aar6404 Publisher: American Association for the Advancement of Science.

[36] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. 2021. Reward is enough. *Artificial Intelligence* 299 (Oct. 2021), 103535. https://doi.org/10.1016/j.artint.2021.103535

[37] Samuel Sokota, Hengyuan Hu, David J. Wu, J. Zico Kolter, Jakob Nicolaus Foerster, and Noam Brown. 2021. A Fine-Tuning Approach to Belief State Modeling. https://openreview.net/forum?id=ckZY7DGa7FQ

[38] Rich Sutton. 2019. The Bitter Lesson :#: http://www.incompleteideas.net/IncIdeas/BitterLesson.html. http://www.incompleteideas.net/IncIdeas/BitterLesson.html

[39] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning, second edition: An Introduction*. MIT Press. Google-Books-ID: uWV0DwAAQBAJ.

[40] David Sychrovský, Michal Šustr, Elnaz Davoodi, Michael Bowling, Marc Lanctot, and Martin Schmid. 2024. Learning not to Regret. https://doi.org/10.48550/arXiv.2303.01074 arXiv:2303.01074 [cs].

[41] Pierre Thodoroff, Wenyu Li, and Neil D. Lawrence. 2022. Benchmarking Real-Time Reinforcement Learning. In *NeurIPS 2021 Workshop on Pre-registration in Machine Learning*. PMLR, 26–41. https://proceedings.mlr.press/v181/thodoroff22a.html ISSN: 2640-3498.

[42] Jaden B. Travnik, Kory W. Mathewson, Richard S. Sutton, and Patrick M. Pilarski. 2018. Reactive Reinforcement Learning in Asynchronous Environments. *Frontiers in Robotics and AI* 5 (June 2018). https://doi.org/10.3389/frobt.2018.00079 Publisher: Frontiers.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903 arXiv:2201.11903 [cs].

[44] Shuo Yin, Weihao You, Zhilong Ji, Guoqiang Zhong, and Jinfeng Bai. 2024. MuMath-Code: Combining Tool-Use Large Language Models with Multi-perspective Data Augmentation for Mathematical Reasoning. https://doi.org/10.48550/arXiv.2405.07551 arXiv:2405.07551 [cs].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation CISE Graduate Fellowships under Grant No. 2313998. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We thank David Wu for his insight and useful feedback. We thank George Konidaris, Nihal Nayak, Arjun Prakash, and David Tao for helpful conversations.