Multi-Modal LLM Reasoning and Agent Modeling

So Hirota* shirota@ucsd.edu
Trevan Nguyen* ttn077@ucsd.edu
Aaryan Agrawal* aaagrawal@ucsd.edu
Zihan Liu* zil065@ucsd.edu
Nathaniel del Rosario* nadelrosario@ucsd.edu
Samuel Zhang* saz004@ucsd.edu
Zhiting Hu zhh019@ucsd.edu

Abstract

Web-based agents using LLMs show promise in automating browser tasks, but scaling inference efficiently remains a challenge. This work explores the question of how best to structure search: implicit (greedy, depth-limited) or explicit (structured exploration like MCTS). Implicit search is potentially computationally cheaper but struggles with backtracking, while explicit search enables efficient exploration but relies on resettable states, which may be impractical in real-world web environments. Experiments on 106 WebArena tasks show explicit search achieves higher task completion rates and better environment interaction efficiency. While explicit search excels in controlled settings, implicit search remains more applicable to real-world tasks. Another aspect to consider is conducting an explicit search on an LLM world model, where the search occurs over predicted next states as opposed to the environment itself, which can potentially gain the benefits of both implicit and explicit search. These techniques extend beyond web environments, and should theoretically be applicable to OS automation (OsWorld) and dynamic game environments (MineDojo).

Code: https://github.com/oetia/llm-reasoners

*Equal contribution

1 Introduction

The web is an expansive and dynamic environment, hosting an immense wealth of information and variety in user interfaces. As the digital world has become an integral component of our world, automating interactions with the web has become an increasingly compelling goal. Web agents hold the promise of transforming how we fundamentally interact with online platforms. These agents could streamline workflows, assist in research, facilitate accessibility, and even execute complex multi-step tasks across different websites.

A key challenge in developing effective web agents lies in their ability to generalize across diverse web environments. Unlike domain-specific automation tools, such as web scrapers or task-specific scripts/extensions, a robust web agent must handle an open-ended action space, reasoning through arbitrary interfaces much like a human user would. This involves interpreting webpage structures, making decisions based on evolving states, and recovering from errors when unexpected behaviors arise.

Recent advances in large language models (LLMs) have significantly improved the viability of web agents. By leveraging LLMs' ability to process textual and structural web data, along with prompting techniques such as Tree of Thought prompting Yao et al. (2024), these agents can be guided by high-level reasoning rather than rigid scripts. However, as shown in works such as Tree Search for Language Model Agents Koh et al. (2024), even the most capable LLM-based agents face substantial obstacles, such as the complexity of web environments, the vast possible action space, and the difficulty of handling long-horizon tasks where errors compound over time. Increasing the computation available to an LLM agent can naturally increase the performance, however, the question remains how is it best to do so.

2 Methods

2.1 Browsergym

Browsergym Chezelles et al. (2024) is the primary environment that we are focusing on. Browsergym essentially provides an OpenAI gym-like environment Brockman (2016) for the web browser. The env object takes in an action represented as code and provides an observation at each step. By utilizing the browsergym library, we can test the performance of our web agent on two key browser task benchmarks: **WebArena** Zhou et al. (2023) and **Assistantbench** Yoran et al. (2024).

2.1.1 Actions

How the action is represented is something that can be slightly finicky. Since browsergym is built off of playwright, the action is going to eventually be JavaScript code that is executed to interact with the DOM. The space of all possible JavaScript code is a massive action

space, and allowing an agent to directly interact with this space creates an correspondingly massive search space. Setting aside issues with search complexity and code correctness, fundamentally all of the tasks that such an agent would be expected to solve, would be doable with the action space available to the average human end user, i.e. just a keyboard and mouse.

For this reason, functions have been predefined for actions such as ‘click‘, ‘fill‘, ‘go_back‘, ‘go_forward‘, etc. While the environment can accept arbitrary code, the agent has been instructed to only provide a specific set of function calls constrained to a ”human” action space.

2.1.2 Observations

After an action is used to step the environment, an observation is returned. This observation is also provided upon the environment instantiation. By default it contains the page HTML, AXTree, and a screenshot of the current state. Directly passing in all of this information into the context of an LLM, especially the HTML, seems to lead to a significant amount of noise, and degrades performance. For a webpage such as Reddit, the HTML you’d get from the homepage can easily be hundreds or even thousands of lines long. If the first step of your task is just to use the search bar to look for a specific subreddit, 99%+ of the elements will be irrelevant. The same to some extent also applies to the accessibility tree, however, the AXTree being a significantly more compact representation, takes up significantly less context. Only the AXTree ends up being passed into LLM context.

For screenshots, since current LLMs tend to struggle with grounding, the screenshots are further augmented with a set of marks (SoM) Yang et al. (2023).

2.2 Search

At every step, a single action can be taken to expand into a new state. For a given task, there can often be over a dozen steps needed to reach completion. If there’s a situation where the first ten steps are correct, but a minor mistake is made on the eleventh step, the task would become in-completable without backtracking.

2.2.1 Implicit Search on the Environment

You could backtrack by relying on the LLM agent to undo it’s action, i.e. if a subscribe button is clicked, it would then click the unsubscribe button to undo it. However, if the LLM clicked a button which brings up a modal form, where it’s still on the same page, then to close the modal, the LLM sends the go_back action, which navigates to the previous page, then while it has closed the modal, it has gone back too far and failed it’s backtrack. While such scenarios should be recoverable, empirically speaking, the LLM struggles to do so, and task execution becomes messy.

2.2.2 Explicit Search on the Environment

An alternative would be relying on a search algorithm, such as MCTS, to do backtracking for you. In an arbitrary environment, this could involve resetting the environment, then replaying all actions, but in the web case, you can do something more sophisticated with caching and reloading web pages. Having an explicit search algorithm like MCTS also provides other benefits in that an LLM doesn't need to identify the correct backtrack and subsequent next node to end up finding the correct trajectory.

Under an explicit search algorithm like MCTS, at every step when expanding a node, the LLM is re-prompted to generate hundreds of possible next actions, i.e. new nodes. Then another LLM can generate an evaluation of each action, so that the rollout isn't random. These evaluations can then be used to influence the subsequent Q-values and guide exploration.

It should be noted that a strong assumption is being made here that backtracking would be possible. In any situation where you are writing information to some external server that you don't control, you run the risk of a backtrack failing. Simply reloading a cached client state will not reset the server state. If a bank transfer is made on something like Zelle, and then you want to backtrack from that state, you cannot do so.

This does make implicitly searching on the environment preferable to explicitly doing so in many cases, as there is no dependence on such an assisted backtrack. Comparing the performance of these two would be interesting.

2.2.3 Comparing Explicit and Implicit Environment Search

Implicit search is essentially greedy search, and greedy search can be represented through MCTS with only a single iteration. Visitation statistics do not accumulate, so the only information being used to decide which node to expand next is the LLM evaluation of an action (fast reward). Implicit search will be implemented as MCTS(depth=..., iterations=1).

When it comes to comparing explicit search and implicit search and how they scale, implicit search can only be scaled by increasing the depth. A nice property is that when running to evaluate performance on a depth 100 implicit search, you can just ignore the later portions of trajectories to get results for all depths before 100. For gathering data on how implicit search scales, a single run of "MCTS" at MCTS(depth=100, iterations=1) will be sufficient.

For explicit search, there does not exist this same property for depth. If you have a 10 iteration 20 depth tree result for a task, if for the first iteration it reaches task completion on depth 15, then the search would end leading to no second iteration. If you try to extrapolate a 10 iteration 10 depth run from this data, you would not be able to do so. However, that does not mean that a 10 iteration 10 depth tree is incapable of also finding a viable solution. You have to do a separate run to find out.

However, for explicit search and iterations, such a convenient property does exist. Through ignoring the later iterations a 10 iteration 20 depth experiment would inherently provide a run for a 9 iteration 20 depth experiment and so on so forth.

With these bits of information in mind, for explicit search, there will be separate runs which will all share the same number of iterations at 10, but have varying depths of 5, 10, and 20.

In the context of browsergym, the benchmark used for these experimental runs will be the webarena benchmark. Of over 800 tasks provided, a subset of 106 tasks will be evaluated on. The main reason for a subset fundamentally comes down to a matter of cost. When running a single example on gpt-4o with MCTS(depth=20, iterations=10), the cost can already easily exceed $1 USD. Should you evaluate on the entire dataset with gpt-4o, a single run would likely take over $1000. With 4 runs planned, some of them likely to be far more expensive than $1 per task, the estimated cost of this entire experiment would likely be more than $4000.

With a provided key with $200 of credit, not only is a subset needed, but also a cheaper model. These experiments will be conducted on said subset of 106 tasks, and also utilize gpt-4o-mini instead of gpt-4o, which should further reduce the costs by approximately 15 fold. The number of action proposals at each step will be kept at 10. While it could be set much higher, with a proposal temperature of 0.7, often over half of the proposals end up being duplicate actions. Scaling n proposals at each step is another axis and increasing it to a 100 or more would likely also benefit performance, but that can be explored later. For now, keeping n proposals fixed at 10 should provide enough variety in responses for benefits to be attainable from search.

Despite gpt-4o-mini being a distill of gpt-4o, the scaling results may not necessarily generalize to gpt-4o or other models and should be taken with caution.

For this reason, alongside the axes of iterations and depth, there will be another axis of the LLM used. Separate runs will also be performed on other strong open source models to ensure that these findings generalize. The models of interest are Qwen2.5-32B-Instruct, and the deepseek r1 distill onto it. At the time of writing, Qwen 2.5 32B is one of the strongest open source 32B models, so it should be fairly representative. As these models can be hosted locally via fast inference frameworks such as SGLang Zheng et al. (2025), the cost should be minimal compared to APIs.

2.2.4 Search on an Internal World Model

Another alternative to addressing the backtracking issue in the explicit search on the environment case is to search and backtrack not on the actual environment, but instead on a simulated ”LLM dream”. On top of having a step function for the browsergym environment, you also have the LLM approximate the results on the step function. This addresses the issue of some actions being irreversible with the downside of becoming dependent on the LLM’s ability to ”dream” the browser environment accurately. As such, the LLM is considered an Internal (as opposed to the external, real browser environment) World Model (where our ”world” is the browser environment).

2.2.4.1 Experimental Settings

Within the Internal World Model scaling paradigm, there are multiple orthogonal axes where we can explore scaling. Some of these include the previously mentioned methods,

such as augmenting the LLM with implicit/explicit search. In this experiment, we experiment with another axis, often called ”Best of N”. This involves sampling N responses from the model at a given step and choosing the highest-ranked response. The ranking can be done in a variety of methods, and we explore two methods in this section. The first method is by simply using the negative log-likelihood of the output, which is typically provided by the API's response. We may interpret this value as the model's ”confidence” in a given answer; therefore, we select the highest negative log likelihood response in this method. In the second method, we use a slightly more sophisticated approach, where we prompt another model to assign ”rewards” to each of N responses, based on how promising they are in completing the goal, given the current browser state and proposed action.

Note that the negative log-likelihood method was used for the AssistantBench tasks, and the reward model method was used for the WebArena methods. AssistantBench is another benchmark that we used, but it was only used for this experiment, so we do not have a dedicated section. It is similar to WebArena in it's tasks, with the key difference being that the tasks require the model to access the open web. Overall, AssistantBench is a much more difficult benchmark.

2.3 MineDojo / Mindcraft

So far we've accomplished introducing planning algorithms in web based environments. This includes support for MCTS, BFS, Beam Search for a web agent's internal planning in BrowserGym. As a result, we can use LLM-Reasoners to run and evaluate experiments of Web Based Agents.

In light of the release of Deepseek's R1 reasoning model, we began brainstorming other environments for an LLM-Reasoners based agent to explore. Our question shifted to, ”What if we can apply LLM-Reasoners to video game environments, specifically open world sandbox environments?”. These environments are ones where the agent possesses free reign to do whatever it reasons the best action to be based on a combination of the environment and internal / external planning.

Our second surveyed environment after OSWorld was MineDojo, a framework for building open-ended, generally capable embodied agents. It can simulate and benchmarked tasks such as crafting, building, and surviving in Minecraft, all core elements of the game which each require their own set of prerequisites and therefore previous action sequences to arrive at the state. Trained on an internet scale knowledge base of YouTube videos, Reddit Posts, and the dedicated Minecraft Wiki, the environment seemed like a promising candidate.

However, due to the lack of recent updates and maintaining of the library / Python package, the framework was too buggy at its current state to provide a space to simulate on. Consequently, we turned to Mindcraft, a Javascript based library that handles API calls to current state of the art Language Models for agent planning and action. Currently we are working with this framework to simulate agent based environment + internal / external planning.

2.4 OS World

In addition to the above mentioned environments, we also perform experiments on the OS World environment Xie et al. (2024). The benchmark tests models' ability to perform operations in a real Operating System and perform various tasks. We have tested the baseline performance of the GPT 4 models and begin to explore potential test-time scaling methods.

2.5 Visualizer

To best support the development of reasoning algorithms, we have directly integrated the improved LLM reasoners visualizer into the AgentLab x-ray tool. Using AgentLab, every action the LLM takes first generates an MCTS tree that is searched for the best outcome. When looking at a run, it is inconvenient to move to a different website to look at the visualizer for each step. Thus, the visualizer is created automatically for each action and embedded directly in the x-ray tool.

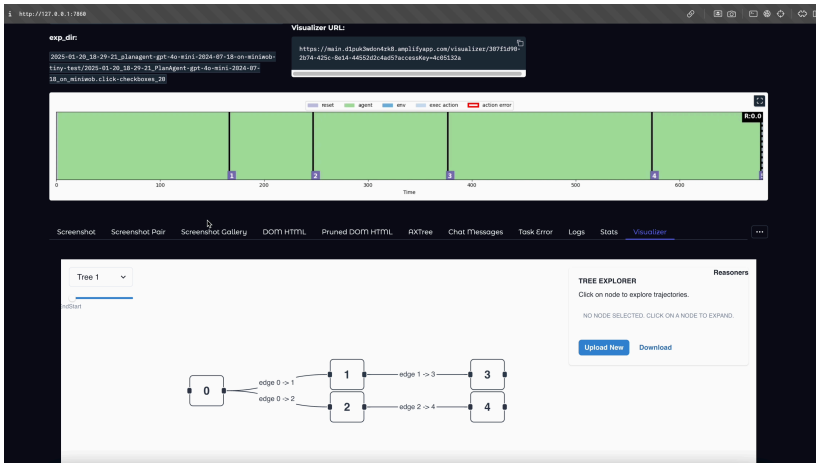

Figure 1: Visualizer embedded into AgentLab x-ray tool

3 Results

3.1 Search on Environment

The 4o-mini errors in the Table 1 are for when the LLM doesn't follow the required JSON format in it's response. The likelihood an LLM formats incorrectly is small, however, given that a task at depth 100, can end up performing close to a thousand LLM calls through action proposals and evaluations, such parsing errors will manifest. For the future, this can be addressed by redoing the LLM call upon a parse failure.

For the deepseek r1 distill on qwen2.5 32b instruct, these errors are usually due to timeouts from the environment. Another thing to note is that max_response_length has been capped

Table 1: Explicit Search on Environment Task Completion Info

Name	Successes	Failures	Errors
4o-mini MCTS(depth=5, iterations=10)	15	89	2
4o-mini MCTS(depth=10, iterations=10)	21	82	3
4o-mini MCTS(depth=20, iterations=10)	35	67	4
4o-mini MCTS(depth=100, iterations=1)	23	71	4
qwen2.5-32b MCTS(depth=5, iterations=10)	8	91	7
qwen2.5-32b MCTS(depth=10, iterations=10)	12	85	9
qwen2.5-32b MCTS(depth=20, iterations=10)	14	83	9
qwen2.5-32b MCTS(depth=100, iterations=1)	9	74	13
r1-distill-32b MCTS(depth=5, iterations=10)	25	81	0
r1-distill-32b MCTS(depth=10, iterations=10)	34	69	3
r1-distill-32b MCTS(depth=20, iterations=10)	35	70	1
r1-distill-32b MCTS(depth=100, iterations=1)	28	75	3

at 8192. Removing the cap, the performance would likely be better, however, the run times would become even longer. Can be explored later.

For the qwen2.5 32b errors, there is likely a bug. The errors are far too high given the baseline of 4o-mini, so the results should be taken with a grain of salt. They are probably lower than they should be.

Table 2: Explicit Search on Environment General Info

Name	Success Rate	Total Cost (USD)	Total Time (Hours)
4o-mini MCTS(depth=5, iterations=10)	0.1415	4.86	11.3430
4o-mini MCTS(depth=10, iterations=10)	0.1981	6.41	15.8691
4o-mini MCTS(depth=20, iterations=10)	0.3302	9.01	25.3912
4o-mini MCTS(depth=100, iterations=1)	0.2347	24.35	42.0037
qwen2.5-32b MCTS(depth=5, iterations=10)	0.0754	N/A	14.1157
qwen2.5-32b MCTS(depth=10, iterations=10)	0.1132	N/A	20.3258
qwen2.5-32b MCTS(depth=20, iterations=10)	0.1320	N/A	38.5611
qwen2.5-32b MCTS(depth=100, iterations=1)	0.0849	N/A	22.3750
r1-distill-32b MCTS(depth=5, iterations=10)	0.2358	N/A	50.1258
r1-distill-32b MCTS(depth=10, iterations=10)	0.3207	N/A	65.4553
r1-distill-32b MCTS(depth=20, iterations=10)	0.3301	N/A	107.2980
r1-distill-32b MCTS(depth=100, iterations=1)	0.2641	N/A	172.0004

In Figure 2 the total time taken for a task is broken down into the time taken for action proposals, the time taken for action evaluations, and the time needed to execute the chosen action on the environment. A datapoint in the total envstep histogram would refer to a single task, and the total time in that task spent on stepping the environment.

In Figure 6 the total token usage is broken down into prompt cache miss tokens, prompt cache hit tokens, and completion tokens. Using the pricing information on the API, you can calculate the total cost in USD for each task.

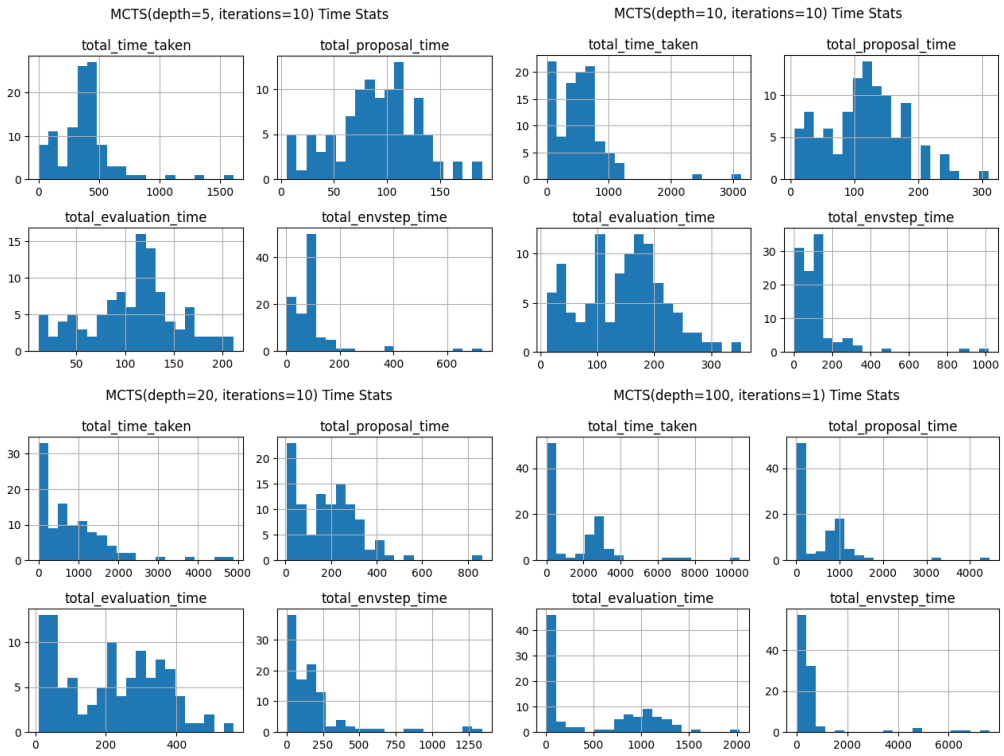

Figure 2: Time usage statistics across all runs.

3.2 Internal World Model

With a Best-of-N scaling strategy, we do not see any increase in performance as the N factor doubles. For AssistantBench, the number of experiments that we ran are quite small at 10 tasks, but this is due to the experiments taking a very long time. In the worst case, we saw a single task taking over 30 minutes.

In Table 3, we show the results of the Best of N scaling strategy on a subset of 10 tasks in AsisstantBench, using negative log likelihood ranking. For the base model, we used GPT-4o-mini with the temperature set to 0.1.

In Table 5, we show the results of the Best of N scaling strategy on a subset of 22 tasks in WebArena, using reward model ranking. For the base model, we used GPT-4o-mini with the temperature set to 0.1. We immediately observe the difference between AssistantBench and WebArena here, as GPT 4o-mini can actually solve some tasks in WebArena.

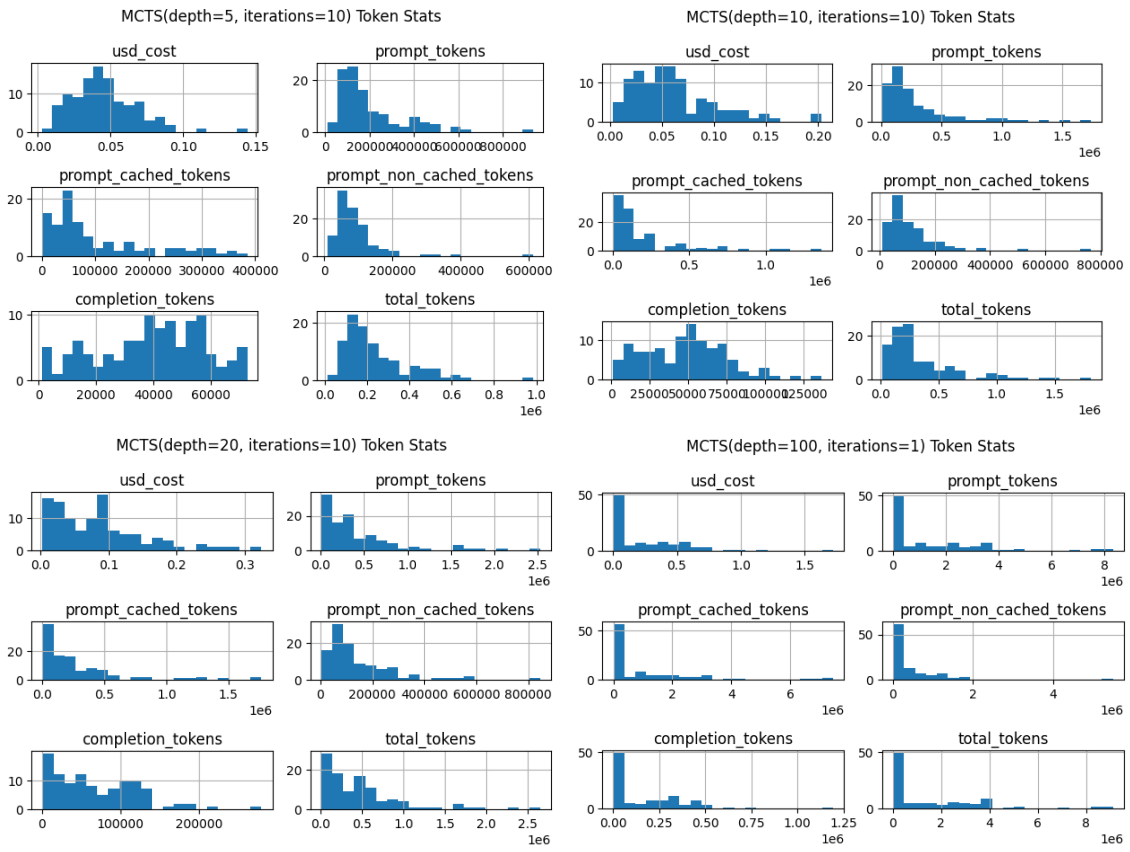

Figure 3: Token usage statistics across all runs.

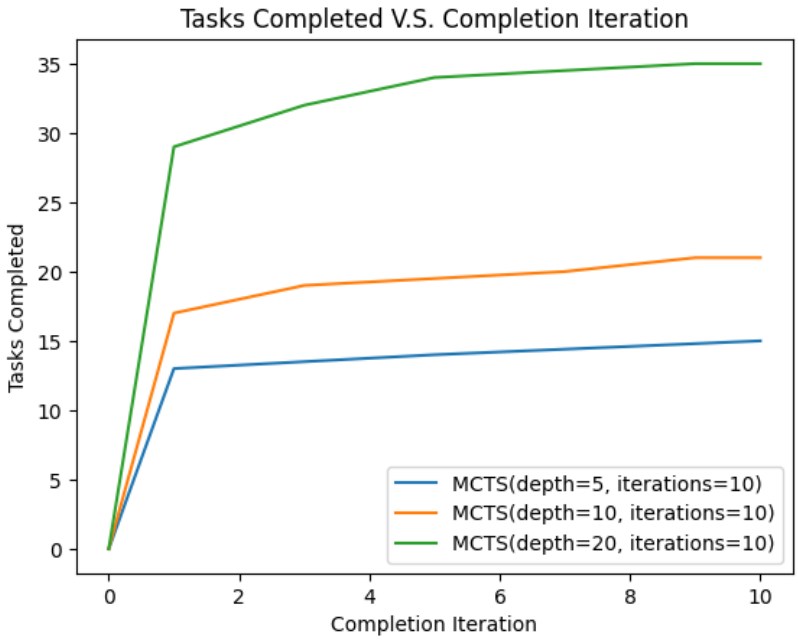

Figure 4: Tasks completed as allowed iterations increase for explicit search on the environment.

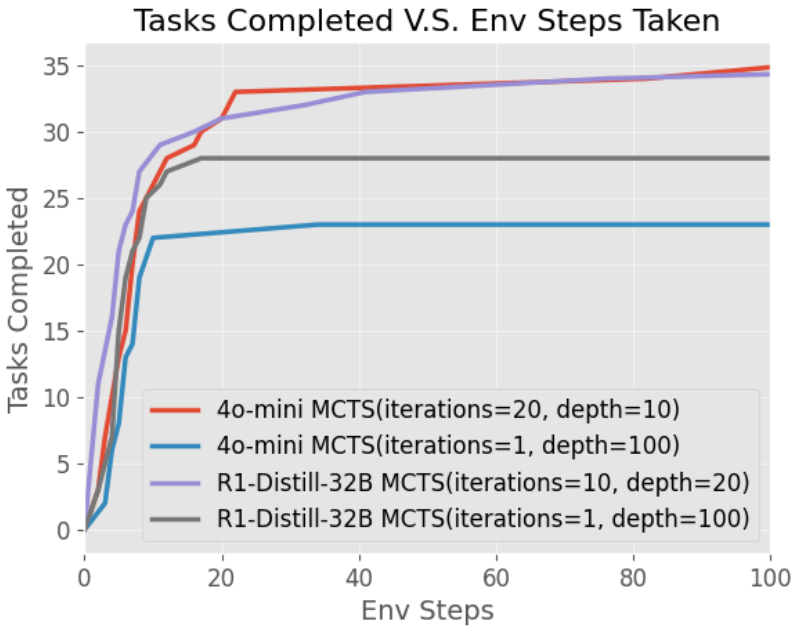

Figure 5: Scaling inference time compute with two separate methods. One with an emphasis on depth, and another with an emphasis on iterations.

Table 3: Search on Internal World Model Task Completion Info (AssistantBench)

Experiment Config	Successes	Failures	Errors
N = 2	0	10	0
N = 4	0	10	0
N = 8	1	9	0
N = 16	0	10	0
N = 32	0	10	0
N = 64	0	10	0
N = 128	0	10	0

Table 4: Internal World Model General Info (AssistantBench)

Name	Success Rate	Total Output Tokens	Total Cost (USD)
N = 2	0	62493	0.42
N = 4	0	119933	0.31
N = 8	0.1	203672	0.42
N = 16	0	467121	0.55
N = 32	0	855778	0.98
N = 64	0	1439588	1.1
N = 128	0	3027348	2.1

Table 5: Search on Internal World Model Task Completion Info (WebArena)

Experiment Config	Successes	Failures	Errors
N = 2	5	17	0
N = 4	5	17	0
N = 8	6	16	0
N = 16	5	17	0
N = 32	6	16	0
N = 64	5	17	0

Table 6: UI TARS Success Percentage

Name	10 steps	15 steps	30 steps	40 steps
UI-TARS 7b	6.1	6.38	8	14
UI-TARS 70b	8	14	14.58	19

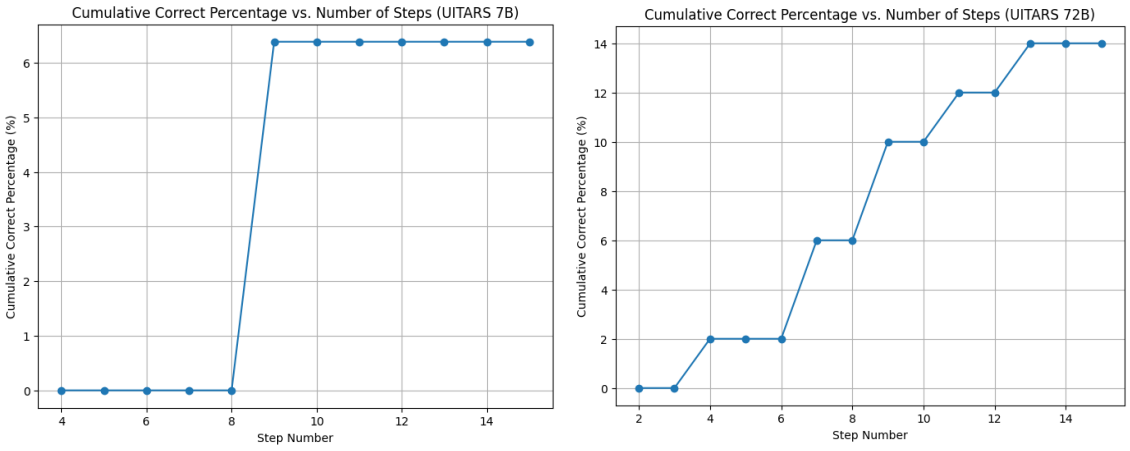

Figure 6: UITARS Cumulative Success Percentages Across Steps

4 Discussion

4.1 Search on Environment

4.1.1 Token Usage & Cost

First, on the topic of cost, referencing Table 2 the total cost of the 4o-mini experiments was (4.86 + 6.41 + 9.01 + 24.35) = **$44.63 USD**. Evaluating on the whole dataset would 8x, and evaluating on gpt-4o instead of 4o-mini would 15x leading to an estimated cost of (44.63 * 8 * 15) = **$5,355.60 USD**. As expected, that would've been expensive.

Since the deepseek and qwen models are hosted locally, the cost would not be calculated through APIs, but instead through electricity usage. The experiments are run through the "single program multiple data" paradigm (SPMD), so the "Total Time" column would be if running every single task in serial. Though not tracked, it's very likely that the kWh usage was less than that of running an H100 node at full power draw for 24 hours, which would be (0.7kW * 8 gpus * 24 hours) = 134.4 kWh. Assuming you can get electricity at 15 cents per kWh, that amounts to approximately $20 USD.

Looking at the token usages histograms in Figure 6, the proportion of prompt tokens cached, does increase in proportion as the experiments scale up, which makes sense as the prompts length will increase as the depth increases, and the prompt contents will also remain roughly the same. However, despite these being the cheapest type of token, the sheer amount of tokens used per task increases massively in general, which explains the numbers in the most expensive explicit search run, MCTS(depth=20, iterations=10), and the implicit search run, MCTS(depth=100, iterations=1).

The cost of time is another aspect, and Figure 2 contains relevant information. Similar to tokens, the subsequent increases are expected, and furthermore highly related to token usage.

4.1.2 Explicit Search on Env

Using a single run at MCTS(depth=..., iterations=10), runs at all lower iterations can be extrapolated. The results are in Figure 4. As the iterations increase, performance naturally does as well. However, something interesting to note is that most of the task completions occur during the first few iterations. When looking at just the first iteration, at MCTS(depth=5) 13 tasks have been completed, at MCTS(depth=10) 17 tasks have been completed, and at MCTS(depth=20) 29 tasks have been completed. This suggests that depth is important for completing the tasks on webarena, which makes sense. Many tasks in the webarena benchmark are designed to be completed in a little under 10 steps. However, have more depth likely benefits the agent through giving it more room to take suboptimal actions that may still lead towards task completion.

Having a greater depth, also seems to benefit the performance increase from iterations. This also makes sense. Iterations are really only useful so long as the search space available

contains a possible correct trajectory. With only 10 iterations, the searchable space isn't that large, which makes having a larger subspace of correct solutions beneficial. So long as the LLM evaluator can roughly "guide" the agent into this subspace, the performance should be better.

On the contrary, if the search space available has only a very narrow subspace of correct trajectories, the search can easily get stuck in a bad subspace and waste its iterations. The total number of tasks is 106, and yet the number of successes in the best case was 35. Almost every single failure ends up going to the maximum iterations, which is to be expected. The majority of the tokens used were on failures. If the agent is using lots of iterations, that's a strong indicator that it's stuck in a bad subspace and likely will fail.

4.1.3 Comparing Explicit and Implicit Search on Env

For the implicit search, observing the orange line in Figure 5, while having a extra depth does help, the returns from depth also do seem to diminish. There are many cases where after depth 20, the agent gets stuck and sends the noop() action 80 times until it hits the depth limit, or it ends up going back and forth between two actions until the depth limit. Similar to the iterations in the explicit search, as the depth increases, if the agent is stuck in a bad subspace, the depth increase won't do much to help.

As a note, in order to compare between the explicit and implicit search scenarios, the x-axis in this chart refers to the number of environment interactions used.

When comparing the two, as expected, since the explicit search is fundamentally designed to address the issue of backtracking difficulties with implicit search, along with an in-built exploration/exploitation trade-off via MCTS, it's hardly surprising that the performance is notably better. However, once again, one of the key advantages of implicit search is that it's usable in real world environments, with potentially constant updates to an external server state.

4.2 Search on Internal World Model

From Table 3, we could not observe any increase in performance, based on scaling up the number of action proposals. Notably, there is one example where the agent succeeds in the task. However, we believe this is due to random chance, rather than a benefit from scaling up the number of proposals. We see similarly disappointing results in Table 5 where we do not observe noticeable improvements in performance on WebArena with reward model based ranking.

From these findings, we assert that Best of N scaling based on log probability ranking or reward model ranking is not a suitable scaling strategy, at least for a weak model like 4o-mini. We believe this specific scaling strategy may work if we utilized a strong model as a reward model. In our setup, we used 4o-mini for price considerations. Our intuition tells us that if we used a stronger model, such as GPT-4o to guide the weaker action proposal model, we may have seen some scaling behavior. This can be explored further in the future.

5 Conclusion

Our experiments highlight the trade-offs between implicit and explicit search strategies in web-based agent environments. Explicit search via MCTS outperforms implicit search in task completion rates by addressing backtracking issues and leveraging exploration-exploitation trade-offs. However, its reliance on resettable states limits real-world applicability, where implicit search remains more practical despite its inefficiencies at higher depths.

Token and time costs scale significantly with depth and iterations, with most successful task completions occurring within the early iterations of explicit search. Depth plays a crucial role in WebArena tasks, as it allows agents more flexibility to recover from suboptimal actions. However, when trapped in poor search subspaces, both implicit and explicit searches struggle, with failures consuming the majority of resources.

For internal world model search, our preliminary findings on Best of N scaling show no significant performance gains in AssistantBench. We hypothesize that improvements may require better ranking mechanisms, such as LLM-based reward models, or stronger models like GPT-4o.

Future work should explore optimizing search efficiency, mitigating failure modes, and adapting explicit search for dynamic environments with external state dependencies.

References

Brockman, G. 2016. "OpenAI Gym." *arXiv preprint arXiv:1606.01540*

Chezelles, De, Thibault Le Sellier, Maxime Gasse, Alexandre Lacoste, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel et al. 2024. "The browsergym ecosystem for web agent research." *arXiv preprint arXiv:2412.05467*

Koh, Jing Yu, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. 2024. "Tree search for language model agents." *arXiv preprint arXiv:2407.01476*

Xie, Tianbao, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei et al. 2024. "Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments." *arXiv preprint arXiv:2404.07972*

Yang, Jianwei, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. "Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v." *arXiv preprint arXiv:2310.11441*

Yao, Shunyu, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. "Tree of thoughts: Deliberate problem solving with large language models." *Advances in Neural Information Processing Systems* 36

Yoran, Ori, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and

Jonathan Berant. 2024. "AssistantBench: Can Web Agents Solve Realistic and Time-Consuming Tasks?" *arXiv preprint arXiv:2407.15711*

Zheng, Lianmin, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez et al. 2025. "Sglang: Efficient execution of structured language model programs." *Advances in Neural Information Processing Systems* 37: 62557–62583

Zhou, Shuyan, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried et al. 2023. "Webarena: A realistic web environment for building autonomous agents." *arXiv preprint arXiv:2307.13854*

Appendices

A.1 Prompts Used

A.1.1 Action Proposal Prompt

```
# Task
{obs["goal_object"]}


# Instructions
Review the current state of the page and all other information to find the best
possible next action to accomplish your goal. Your answer will be interpreted
and executed by a program, make sure to follow the formatting instructions.


# Currently open tabs
Tab {page_index}{" (active tab)" if page_index == obs["active_page_index"] else ""
Title: {page_title}
URL: {page_url}


# Current page Accessibility Tree
{obs["axtree_txt"]}


# Current page DOM
{obs["pruned_html"]}


# Current page Screenshot
"image_url": {
    "url": image_to_jpg_base64_url(obs["screenshot"]),
    "detail": "auto",
},  # Literal["low", "high", "auto"] = "auto"
```

```
# Action Space
{action_set.describe(with_long_description=False, with_examples=True)}

Here are examples of actions with chain-of-thought reasoning:

I now need to click on the Submit button to send the form. I will use the click ac
‘‘‘click(”12”)‘‘‘

I found the information requested by the user, I will send it to the chat.
‘‘‘send_msg_to_user(”The price for a 15\\” laptop is 1499 USD.”)‘‘‘


# History of past actions
{action_history}


# Error message from last action
{obs[”last_action_error”]}


# Next action
You will now think step by step and produce your next best action. Reflect on your
```

A.1.2 Action Evaluation Prompt

```
# Task
{obs[”goal_object”]}


# Instructions
Review the current state of the page along with a proposed action and determine ho
{
    ”reasoning”: [your_reasoning]
    ”score”: [your_score]
}


# Currently open tabs
Tab {page_index}{” (active tab)” if page_index == obs[”active_page_index”] else ””
Title: {page_title}
```

```
URL: {page_url}


# Current page Accessibility Tree
{obs["axtree_txt"]}


# Current page DOM
{obs["pruned_html"]}


# Current page Screenshot
"image_url": {
    "url": image_to_jpg_base64_url(obs["screenshot"]),
    "detail": "auto",
},  # Literal["low", "high", "auto"] = "auto"


# Action Space
{action_set.describe(with_long_description=False, with_examples=True)}

Here are examples of actions with chain-of-thought reasoning:

I now need to click on the Submit button to send the form. I will use the click ac
```click("12")```

I found the information requested by the user, I will send it to the chat.
```send_msg_to_user("The price for a 15\\" laptop is 1499 USD.")```


# History of past actions
{action_history}


# Error message from last action
{obs["last_action_error"]}


# Proposed action
{action}
```

```
# Evaluation of Proposed Action
As mentioned before, considering all the information above in the context of the g
{
    ”reasoning”: [your_reasoning]
    ”score”: [your_score]
}
```