Glider: A Reinforcement Learning Approach to Extract UI Scripts from Websites

Yuanchun Li
Microsoft Research
Beijing, China
yuanchun.li@microsoft.com

Oriana Riva
Microsoft Research
Redmond, WA, USA
oriana.riva@microsoft.com

ABSTRACT

Web automation scripts (tasklets) are used by personal AI assistants to carry out human tasks such as reserving a car or buying movie tickets. Generating tasklets today is a tedious job which requires much manual effort. We propose *Glider*, an automated and scalable approach to generate tasklets from a natural language task query and a website URL. A major advantage of Glider is that it does not require any pre-training. Glider models tasklet extraction as a state space search, where agents can explore a website's UI and get rewarded when making progress towards task completion. The reward is computed based on the agent's navigating pattern and the similarity between its trajectory and the task query. A hierarchical reinforcement learning policy is used to efficiently find the action sequences that maximize the reward. To evaluate Glider, we used it to extract tasklets for tasks in various categories (shopping, real-estate, flights, etc.); in 79% of cases a correct tasklet was generated.

CCS CONCEPTS

• **Information systems → Data extraction and integration**; • **Computing methodologies → Reinforcement learning**.

KEYWORDS

UI script; Web interfaces; Reinforcement learning; Task completion.

ACM Reference Format:
Yuanchun Li and Oriana Riva. 2021. Glider: A Reinforcement Learning Approach to Extract UI Scripts from Websites. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21), July 11–15, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404835.3462905

1 INTRODUCTION

Intelligent personal assistants (Google Assistant, Apple Siri, etc.) perform tasks on behalf of a user based on natural language commands or questions. They usually operate through a two-step process: *(i)* task understanding and *(ii)* task execution. *Task understanding* uses NLP techniques [27, 32, 58, 74] to extract a task's intent and input parameters from a natural language query. For example, it infers that in the command "*Estimate ride fare from 1st Ave, New York to Central Park*" "estimate ride fare" describes the user intent and "1st Ave, New York" and "Central Park" represent the parameters "start location" and "end location", respectively. *Task execution* involves executing the parsed command through a relevant service. Traditionally, this meant connecting to an API (e.g., Uber get_price_estimates API), but recent AI assistants have demonstrated how a task may also be carried out by directly driving the GUI of a website supporting it [19, 40, 61]. This second option is appealing because it can give AI assistants access to a set of human tasks much larger than what traditional service APIs support. In this paper, we focus on how to make website-based execution of human tasks more accessible to AI assistant developers.

The web has a long tradition of UI tools that can be leveraged to automate web task execution [4, 25, 35, 47, 52]. In the simplest scenario, a user can use a record-and-replay tool such as Selenium [52] to record themselves interacting with a website and obtain a UI script to replay the interaction. Despite its simplicity, this approach is time consuming as it requires manually recording every single task to automate, possibly for every variation of input parameters. As developers of AI assistants need to automate a wide range of tasks and websites, the effort would be cumbersome. Moreover, maintaining UI scripts is generally costly due to website updates. An alternative option is to train AI agents to automatically carry out web tasks, possibly described in natural language [19, 40, 59]. However, current solutions can automate only tiny hand-crafted websites (MiniWob [56]) and still require collecting large amounts of manual demonstrations to successfully train the agents.

To reduce this overhead and simplify the process of web task automation, we propose *Glider*, an unsupervised approach to automatically generate executable UI scripts for web tasks, which we call *tasklets*. The only inputs Glider requires are an English sentence describing a relevant task and one or more website URLs where to execute the task. To be compatible with existing AI assistants, Glider assumes a task description is parsed and annotated as by standard task understanding modules. An example query may be "Estimate ride fare from 1st Ave, New York [start-location] to Central Park [end-location]". Given this query and say the website lyft.com, Glider's goal is to automatically generate a tasklet that takes as input the pick-up and drop-off locations, inputs them in the corresponding text boxes, clicks the "get estimate" button, and returns the price list page. Tasklets resemble familiar Selenium UI scripts and can be replayed using the Selenium web driver [53].

Glider models the problem of automatically generating tasklets as a *state space search*. A website is represented as a set of *states* (webpages) and *actions* (interactions with buttons, lists, menus, etc.). An *agent* searches the space by taking some actions to reach a *goal state* (i.e., task completion). Actions are rewarded based on their

SIGIR '21, July 11–15, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-8037-9/21/07…$15.00
https://doi.org/10.1145/3404835.3462905

progress towards task completion. When the goal state is reached, the sequence of actions taken is encoded as a tasklet.

To make search practical in real websites without requiring pre-training or website modifications, Glider must deal with two challenges: *(i)* how to efficiently navigate the large search spaces associated with websites, and *(ii)* how to determine progress towards task completion without prior knowledge of the task or website.

The search space is large because the DOM tree of an average website contains hundreds of UI elements, many of which are interactable. A search agent can click any link, button, list, etc., thus making the number of possible trajectories extremely large. Our first intuition to solve this problem is that a successful search strategy should *mimic* as much as possible human behavior. For example, in western languages, we typically read a page and take actions left-to-right and top-to-bottom (directionality pattern) and do not "jump" around a page (locality pattern). Whenever possible, Glider automatically hides from the agent "misleading" actions, and emits negative rewards based on *locality* and *directionality* patterns.

The second challenge is about determining progress towards completion of a task in a generalized way, which is notoriously hard [71]. If we consider the example task "Translate the word cat to Spanish" an agent may be rewarded when after some UI interactions the word "gato" is found on the page; this reward is *task-specific*, but holds true across websites. In another task, such as making a restaurant reservation, the reward may be determined based on the final confirmation message; such a message is generally unknown beforehand and different for different restaurant websites, hence in this case the reward strategy is *task-* and *website-specific*. In summary, inferring how a task is progressing towards completion usually relies on task- or website-specific hints, but relying on them would require manual effort and conflict with our scalability goal. We address this problem by computing rewards based on the *text similarity* between the states (webpages) visited, the navigation path (the UI actions executed so far) and the task description.

Locality, directionality and text similarity are the cornerstones of Glider's reward model. However, as these types of reward are noisy and known only *after* an action is executed (e.g., an action may trigger content updates necessary to compute similarity), Glider cannot adopt classical search algorithms or rely on basic backtracking (e.g., the web client's state may not be reproducible). Instead, Glider relies on reinforcement learning [43, 44, 57, 67] as a search strategy. The strategy is formalized as a hierarchical policy [31], including a master policy that controls navigation between pages and a sub-policy that determines inputs for forms in a page. Finally, to further deal with noisy rewards, in a post-processing phase, Glider replays the highest-rewarded tasklets, and corrects them by removing unnecessary actions or injecting others. We evaluated Glider by building a dataset of 164 tasks and collecting 44 more from actual developers. Overall, Glider generated a correct tasklet in 79% of cases, while baseline approaches achieved 56–60% success rates.

In summary, this paper makes three contributions: *(i)* an unsupervised approach for generating UI automation scripts from natural language inputs; *(ii)* an implementation based on reinforcement learning that unlike current solutions [19, 40, 59] works on real websites; and *(iii)* an evaluation based on a new dataset of 208 tasks across 10 task categories. We make code and dataset available at https://github.com/microsoft/glider_tasklet_crawler.

2 GOAL AND CHALLENGES

Glider's overall goal is to enable developers to generate tasklets for *real websites* and *automatically*. By "automatically" we mean that Glider should *not* require developers to collect manual demonstrations for pre-training and that Glider should *not* need to be tuned or customized to work on new tasks and websites.

A Glider-generated tasklet resembles a traditional Selenium UI script [52], consisting of a sequence of commands for performing actions in a web browser (click a button, select an item, type text in an input field, etc.). Glider aims to generate *correct* tasklets, meaning those that lead to task completion. For efficiency, we prefer short execution paths and reject paths that exceed a maximum number of steps computed based on the task complexity (see §4.1). On the other hand, we do not require a shortest path for correctness. For instance, in a shopping website one may find a product by searching its name or by navigating the product catalog.

We envision developers can write task queries directly (as in our developer study), obtain them by processing user logs with task understanding techniques [27, 32, 58] or crowdsource their collection. As in AI agents for web navigation [19, 40], we assume query parameters are annotated. A task's target websites can be specified manually or obtained from web traffic ranks (alexa.com).

To meet its goal, Glider must deal with two challenges: *(i)* how to search the large action spaces of websites, and *(ii)* how to estimate progress towards completion of an arbitrary task.

Large UI trees. The number of available actions (interactable UI elements, such as buttons, menus, links, etc.) in a webpage can be in the order of one hundred. In our tests, a webpage had on average 76 actions. For a task of 3–5 steps (a step represents a UI action) and a webpage with 76 actions, the probability of randomly finding a correct execution path is in the range $[10^{-10}, 10^{-6}]$. To illustrate, we took one of the simplest website in our dataset which is an electronic medical record website [45], and considered the 4-step task "*Lookup cancelled [appointment status] dermatology [service type] appointments at laboratory [location]*"(see Figure 1). We first tried to discover an execution path for the task using a simple Random Walk approach that randomly selects actions. We gave it as entry point the start page of the website and executed it for 100 episodes where in each episode the maximum number of allowed actions was 10. We inspected the captured traces and found no correct path for the task. Only by increasing the number of episodes to 500 (which took over 2.5 hours) it was able to succeed at episode 358. In comparison, Glider found a correct path after 39 episodes.

How to deal with the large action spaces of websites is a known problem. Recent work on training reinforcement learning agents for web navigation [19, 40] uses hand-crafted test cases [56] which are smaller (160×210 pixels) and simpler (10–50 UI elements per page) than actual websites. We apply Glider to real websites and address this issue by computing *partial* rewards based on locality and directionality (§3.2) and by adopting a hierarchical policy (§3.3).

Reward model. Even when Random Walk is able to generate a correct path, the question is how to recognize it among the many visited paths. A current approach consists of comparing the DOM tree of each visited webpage with that of the "goal" page (i.e., the page at task completion) [19]. However, this approach cannot scale because *(i)* a goal page's DOM tree has to be captured in advance

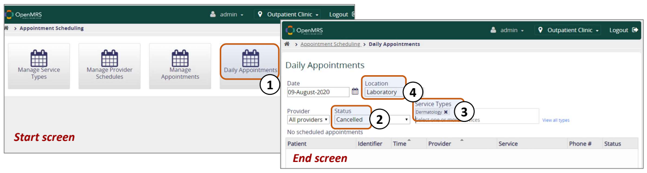
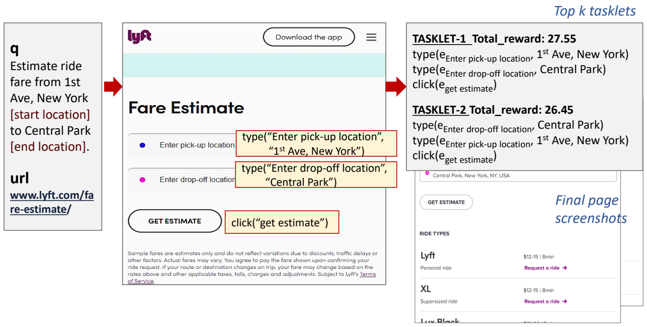


Figure 1: The 4-step task "Lookup cancelled dermatology appointments at laboratory" in our test medical website [45].



q
Estimate ride fare from 1st Ave, New York [start location] to Central Park [end location].

url
www.lyft.com/fare-estimate/

type("Enter pick-up location", "1st Ave, New York")
type("Enter drop-off location", "Central Park")
click("get estimate")

TASKLET-1_Total_reward: 27.55
type($e_{\text{Enter pick-up location}}$, 1st Ave, New York)
type($e_{\text{Enter drop-off location}}$, Central Park)
click($e_{\text{get estimate}}$)

TASKLET-2 Total_reward: 26.45
type($e_{\text{Enter drop-off location}}$, Central Park)
type($e_{\text{Enter pick-up location}}$, 1st Ave, New York)
click($e_{\text{get estimate}}$)

lyft
Download the app
Fare Estimate
Enter pick-up location
Enter drop-off location
GET ESTIMATE
Sample fares are estimates only and do not reflect variations due to discounts, traffic delays or other factors. Actual fares may vary. You agree to pay the fare shown upon confirming your ride request. If your route or destination changes on trip, your fare may change based on the rates above and other applicable taxes, tolls, charges and adjustments. Subject to Lyft's Terms of Service.
Top k tasklets
Central Park, New York, NY, USA
GET ESTIMATE
Final page screenshots
RIDE TYPES
Lyft
Personal ride
$12-15 | 8min
Request a ride →
XL
Supersized ride
$12-15 | 8min
Request a ride →
Lux Black

Figure 2: Tasklet discovery for the fare estimate task in lyft.com. The output is a ranked list of k tasklets.

and for every target website, and *(ii)* websites are dynamic so DOM trees change all the time. Another option is to rely on specific texts or images appearing *only* in the goal page. For example, the task "Convert 10 kg in pounds" could be deemed completed when the number "22.0462262" is detected. This approach, however, can work only for few task categories where the output is deterministic, such as unit conversion or dictionary tasks. Finally, by collecting many instances of pages obtained at completion of the same task in multiple websites, one could train a prediction model for task completion [71]. However, building this dataset would require lots of effort. To be low-effort, Glider adopts a task- and website-independent reward model that constantly evaluates the similarity of the visited webpages and executed actions with the task query (§3.2).

3 OUR APPROACH: GLIDER

Figure 2 gives a high-level overview of Glider. Given q, an English sentence describing a task, and a website *url*, Glider returns maximum k tasklets (e.g., $k = 5$) ranked by a confidence score. To assist developers, for each generated tasklet Glider also returns a screenshot of the webpage obtained upon its execution, and allows developers to replay the tasklets to obtain a preview.

Once the pair <q,url> is specified, the whole process (Figure 3) is automated. Glider searches the target website to find one or more tasklets that satisfy q. A website is represented as a *state space* (§3.1), in which the action sequences are rewarded by a reward model (§3.2) based on their progress towards task completion. The search is carried out by a *reinforcement learning agent* (§3.3). In the last step, the agent's learning episodes are post-processed to maximize output correctness (§3.4). We describe each step next.

3.1 Problem definition

Glider's input queries are natural language sentences with one or more parameters p and with annotations p_{ann} for each p. In the example query "*Estimate ride fare from* <u>*1st Ave, New York*</u> *[start location] to* <u>*Central Park, New York*</u> *[end location]*", each p_{ann} (in square brackets) is a term classifying the associated p (underlined).[1]

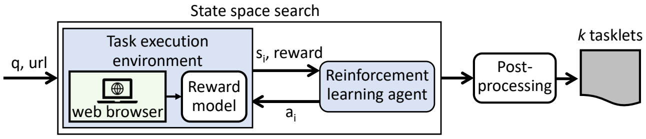


Figure 3: Glider overview.

Each website is represented as a finite state machine $<W, A>$, where W is the set of states and A is the set of actions that can lead to state transitions. A state $w \in W$ corresponds to a page in a website, where w_0 is the start page. Webpages with different DOM trees or different content (e.g., a different value typed or selected in a UI element) are regarded as different states in $<W, A>$. Each w is represented by its DOM tree w_{dom} and a screenshot w_{screen}. Each node in w_{dom} corresponds to a UI element e in the webpage uniquely identified by its xpath. From the DOM tree we extract various properties of e: *(1)* e_{loc}, its location represented as a rectangle in the page; *(2)* e_{type}, whether the element is "clickable" or "editable" inferred from various HTML tags (<p> for text elements, <img> for images, <button> for elements accepting click events, etc.); *(3)* e_{text}, whether the element has associated some text inferred using a combination of HTML attributes (text, placeholder, value, etc.), as they are not all always available; and *(4)* e_{font}, its font size.

$A_w \subseteq A$ is the set of actions that can be executed in $w \in W$. Actions can lead to state transitions: clicking a link can redirect to another page or typing text into a text field can lead to a content change in the current page. Currently, we constrain the action space to the following four categories of actions, which have been sufficient to complete most tasks on the websites we tested.

(1) click(e): clicking on element e, where e must be a clickable UI element, such as a button, a hyperlink, etc.
(2) select(e,i): selecting the i-th child of element e; e must be a <select> DOM element, and i must be smaller than the number of options in e (e.g., selecting an item from a menu).
(3) type(e,t): typing text t into e, where e is an editable UI element (e.g., text field) and t one or more words in q.
(4) enter(e): submitting the content in e by pressing the "Enter" keyboard key, where e must be an editable UI element.

We define **tasklet(**q, *url***)** as a sequence of actions $<a_1, a_2, \ldots>$ in *url* that can complete the task described in q. To discover tasklets, we define the problem of **state space search** as follows. At each time step i, an agent selects an action $a_i \in A$ based on the current state $s_i \in S$. In response, the *environment*, a web browser instrumented to execute UI actions, returns to the agent a new state s_{i+1} and a $reward(s_{i+1}, a_i)$. At initialization, the environment loads *url* and passes the initial state w_0 and the set of possible actions A_{w_0} to the agent. The goal of the agent is to learn a sequence of actions (or tasklet) that maximizes the cumulative reward R.

In the next section we describe how rewards are computed and in §3.3 we describe our implementation using reinforcement learning.

[1] We assume these annotations are produced by standard named entity extraction tools [15, 58]. For flexibility, we avoid tying ourselves to any one specific NLP tool or ontology. Here, the annotations "pickup", "origin" or "arrival" would work too.

3.2 Reward model design

To meet our zero-demonstration and scalability goals, we discussed earlier how rewards must be computed in a website and task-independent manner. Our insight is that one can recognize whether an agent is making progress towards completing a task based on the current task description, the current state (the content in the current page) and the sequence of actions executed so far, *without* the need to know the expected result of the task. More concretely, we infer task progress based on the following factors:

- *Action locality*: when interacting with a webpage, users usually focus on a small portion of the page at the time, rather than "jumping" around. In the task "*Find cookie recipes*" the agent may try to enter the word "cookie" in a text box and click the search button *next* to it; the agent should not click a link "Cookies, Terms & Privacy" at the *bottom* of the page.
- *Action directionality*: as websites are designed for humans, the placement of UI elements is usually optimized to aid reading and task completion. For example, in western languages we read a page from left to right and from top to bottom (also called Z and F patterns [10]), hence subsequent actions to carry out a web task tend to flow accordingly.
- *Task-webpage similarity*: a webpage contains textual content and actions which also have associated some classifying text (e.g., a button label); both types of text should semantically match the task query, including the words specifying the task intent, parameter values and parameter annotations.

Reward indicators. We map the above observations to concrete *task progress indicators* that the environment computes upon each action's execution. Given a query q with one or more parameters p, each annotated with p_{ann}, the current page w, and the sequence of actions so far executed as=<$a_1, a_2, \ldots, a_k$>, we define:

(1) num_ld: the number of long-distance action pairs in as, meaning subsequent actions whose associated UI elements are located at a distance bigger than half of the window's width or height (i.e., 500 pixels for 1000×1000 pixels pages).
(2) num_rd: the number of reverse-directional actions in as; if an action's UI element is located on the top or on the left of the UI element of the preceding action in the sequence, the action is considered reverse-directional.
(3) $task_{sim}$: the similarity between words in q (excluding parameters and their annotations) and texts appearing in the current page w (using a subset of the texts to minimize noise, as described later). Rewards based on task similarity are designed to encourage the agent to navigate to and stay in webpages relevant to the task's *intent.*
(4) $par_{sim}(p)$: the similarity between a parameter p in q and texts present in the current state w. Rewards based on parameter similarity are designed to encourage the agent to interact with UI elements relevant to the task's *parameters.*

Similarity scores. To compute text similarity for the last two reward indicators above, we opted for a conservative approach based on edit distance. We tested also more advanced semantic similarity models [8, 55, 58] but they turned out to be "confusing" for our agent (see §4.3). We define the following similarity function:

$$sim(t_1, t_2) = \max\{sim_{1-gram}(t_1, t_2), sim_{n-gram}(t_1, t_2), sim_{date}(t_1, t_2), sim_{num}(t_1, t_2)\}$$

where sim_{1-gram}, sim_{n-gram}, sim_{date}, and sim_{num} are the similarity between the texts t_1 and t_2 by considering them as single-word units (1-gram), multi-word units (n-gram), dates, or numbers, respectively. Specifically, $sim_{1-gram}(t_1, t_2) = 1 - \frac{lev\_dist(t_1,t_2)}{max\{|t_1|,|t_2|\}}$ where lev_dist is the Levenshtein distance [36][2] and sim_{n-gram} is the normalized ratio of words in common to the two input sentences. For dates and numbers we cast to the respective data objects and use their comparison functions.

Computing similarity between a task query and only the texts strictly associated with actionable UI elements (e.g., name, placeholder value, etc.) turned out to be too restrictive. Developers may not always semantically annotate their UI elements in the DOM tree or may use generic labels (e.g., a label "Let's go" for a button initiating a restaurant search). On the other hand, a webpage may contain lots of misleading texts hence matching "all" texts in it may also not work. Hence, we compute similarity using a subset of the UI elements appearing in the current page w, denoted as w^*. We say $e \in w^* \subseteq w$ if e is *(i)* a non-interactable UI element (e.g., title, text box's label, etc.) associated with "short" text (defined using a threshold, to remove item lists or long paragraphs), or *(ii)* an interactable UI element which was subject to a previous interaction in current or previous webpages.

Then, we define the task similarity score as follows:

$$task_{sim} = \underset{tw \in q^*}{\text{mean}}\{\max_{e \in w^*}\{sim(tw, e_{text})\}\}$$

where q^* is defined as q with parameters and prepositions omitted and $sim(tw, e_{text})$ is the similarity between a task word tw in q^* and e_{text}, the text associated with e in w^*.

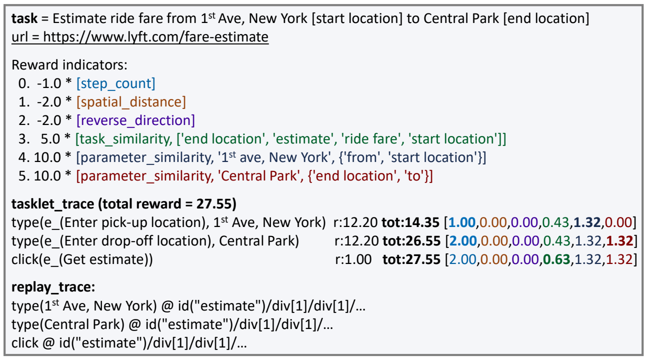
To compute the parameter similarity score par_{sim} for a parameter p, we consider both the parameter value p_{val} specified in the query and the parameter's metadata $p_{metadata}$ (i.e., p's annotations and words surrounding p in q, such as prepositions). Correspondingly, we introduce *(i)* $s_v(p, e) = \theta * sim(p_{val}, e_{text})$ (scaled by a factor θ[3]) and *(ii)* $s_m(p, e) = sim(p_{metadata}, e_{text})$, and define

$$par_{sim}(p) = \max_{e_i, e_j \in w^*}\{s_v(p, e_i) + \frac{s_v(p, e_i) * s_m(p, e_j)}{1 + \lambda * dist(e_i, e_j)}\}$$

where λ is a hyperparameter to scale the spatial distance $dist(e_i, e_j)$ (it is set to 0.02 by default to scale a window size of 1000x1000 pixels). The intuition behind this equation is the following. Besides evaluating whether the name of a UI element matches a parameter's annotation or value, it is important to consider the surrounding *context* both in the page UI and in the query. This is especially important when interacting with icons or UI elements that do not have any labels. More specifically, when considering the parameter "Central Park" in the example query in Figure 2, par_{sim} evaluates *(i)* whether the parameter value p_{val} (e.g., "Central Park") has high similarity with the text of any e_i in w^*, *(ii)* whether the parameter's annotation p_{ann} (e.g., "end location") or any of the surrounding text of p_{val} in q (e.g., the preposition "to") has high similarity with the text associated with any e_j in w^*, and *(iii)* whether e_i and e_j correspond to UI elements that are spatially close (e.g., whether the drop-off location text box in the webpage

[2] t_1 and t_2 are lemmatized before comparing. Words with different first letters have zero similarity (e.g., "weight" vs. "eight").

[3] θ is a hyperparameter aimed to avoid typing or setting parameter values into UI elements greedily, to gain more rewards. It is 0.7 if e is a text field and 1.0 otherwise.



```
task = Estimate ride fare from 1st Ave, New York [start location] to Central Park [end location]
url = https://www.lyft.com/fare-estimate

Reward indicators:
 0.  -1.0 * [step_count]
 1.  -2.0 * [spatial_distance]
 2.  -2.0 * [reverse_direction]
 3.   5.0 * [task_similarity, ['end location', 'estimate', 'ride fare', 'start location']]
 4. 10.0 * [parameter_similarity, '1st ave, New York', {'from', 'start location'}]
 5. 10.0 * [parameter_similarity, 'Central Park', {'end location', 'to'}]

tasklet_trace (total reward = 27.55)
type(e_(Enter pick-up location), 1st Ave, New York)  r:12.20 tot:14.35 [1.00,0.00,0.00,0.43,1.32,0.00]
type(e_(Enter drop-off location), Central Park)       r:12.20 tot:26.55 [2.00,0.00,0.00,0.43,1.32,1.32]
click(e_(Get estimate))                               r:1.00   tot:27.55 [2.00,0.00,0.00,0.63,1.32,1.32]

replay_trace:
type(1st Ave, New York) @ id("estimate")/div[1]/div[1]/...
type(Central Park) @ id("estimate")/div[1]/div[1]/...
click @ id("estimate")/div[1]/div[1]/...
```

Figure 4: Rewards computed for the "Estimate ride fare" task in Figure 2. Partial rewards (r), total rewards (tot) and breakdown across reward indicators are reported.

appears next to a label "to" and whether in q the word "to" precedes the drop-off value "Central Park").

Reward function. Using the reward indicators, we define the cumulative reward function $R(q, w, as)$ as follows:

$$R(q, w, as) = w_{dist} * num_{ld} + w_{dir} * num_{rd} + w_{task} * task_{sim} + w_{par} * \sum_{p \in P} par_{sim}(p) - k$$

where P is the set of parameters in q, and w_{dist}<0, w_{dir}<0, w_{task}>0 and w_{par}>0 are weights that we empirically learn (see §4.3). The discount factor k is a step penalty that encourages the agent to discover short paths; k is set to 1 for each action taken except for "submit" actions (including enter actions or interactions with UI elements with the HTML tag submit, as they are usually the last action in an execution). After every action taken by the agent, the environment returns a *partial reward* computed as the increment of the cumulative reward, i.e., the reward for the $i - th$ step is $r_i = R(q, w, as_i) - R(q, w, as_{i-1})$. Figure 4 shows an example of such reward computation.

In the case of tasks spanning multiple webpages, if the last-executed action led to a page transition, the reward accumulated so far is adjusted as follows. If the action that caused the page transition is a "submit" action, then it is assumed that the agent is making progress towards completing the task, hence the reward accumulated so far is kept. Otherwise, it means that the agent has dropped the ongoing task, thus the accumulated reward is cleared.

3.3 Reinforcement learning agent

We now describe how Glider searches the state space using the reward model above. The design of the search approach was guided by two requirements. First, our state space is only *partially visible* and the reward of an action is known only *after* an action is executed, thus the search algorithm must work in an *exploration-exploitation* manner, i.e. collecting information by randomly exploring the website to guide the selection of future actions. Second, backtracking and jumping between different states is expensive because a webpage's client state may be associated with a server state that cannot be reproduced; hence, *local search* (e.g., Hill climbing) is preferred over uniformed search (e.g., depth-first, breadth-first, etc.).

Algorithm 1: LearnMaster: Train the master policy.

```
Input: $Q$: the value function to train, $q$: the task query, $url$: the website URL, $ENV$:
       the task execution environment, $LearnForm$: the algorithm to learn the
       form-filling sub-policy
Hyperparameters: $N$: the number of episodes, $M$: the number of steps in each episode,
                 $\alpha$: learning rate, $\gamma$: reward discount factor
Global variables: $RH_F$: reward history of form $F$, $R_F$: reward function for form $F$
begin
    initialize the set of action sequences $AS$ as $\emptyset$
    for episode $j$ from 1 to $N$ do
        reset $ENV$ and load $url$ in the browser
        initialize the action sequence $as$ as empty
        initialize $s_0$=<$q$, $w_0$, $as$> where $w_0$=$url$'s start page
        calculate $\epsilon$ as $1 - j/N$
        for step $i$ from 0 to $M$ do
            set $A_i$ as the set of possible actions in $s_i$
            break current episode if $A_i = \emptyset$
            convert $A_i$ to $A'_i$ by grouping select/type actions into fill-form actions
            generate a random number k in the range $[0.0, 1.0)$
            if $k < \epsilon$ then
                randomly choose an action $a_i$ from $A'_i$
            else
                select an action $a_i = argmax_{a \in A'_i} Q(s_i, a)$
            if $a_i$ is a form-filling action fill-form($F$) then
                get actions $as_F = LearnForm(F, \epsilon)$ and execute in $ENV$
            else
                execute $a_i$ in $ENV$
            observe $r_i$ and $s_{i+1}$
            update $Q(s_i, a_i) \leftarrow (1 - \alpha)\, Q(s_i, a_i) + \alpha(r_i + \gamma \max_a Q(s_{i+1}, a))$
            put $a_i$ into $as$
        put $as$ into $AS$
        put <$as_F$, $R_{max}$> into $RH_F$ if $F$ has been filled. $R_{max}$ is the maximum
          cumulative reward in this episode
        train $R_F$ with a batch sampled from $RH_F$ for each form $F$
    return $Q$ and the set of action sequences $AS$
```

We explored how to adapt reinforcement learning (RL) to our problem. In addition to meeting the two requirements above, RL works well for non-deterministic search, which in our case occurs when a website's content changes dynamically during exploration.

Hierarchical policy. In standard RL, at each step i, an agent selects an action $a_i \in A$ in the current state $s_i \in S$ based on a policy $\pi : S \longrightarrow A$. The agent aims to maximizes the cumulative reward R by rolling out episodes as suggested by π. Unlike traditional applications of RL [30, 37, 43, 44, 57], A becomes very large when considering real websites [19, 40]. In addition to negative rewards based on locality and directionality, we further address this problem by adopting a hierarchical policy [31].

We introduce the concepts of *form element* and *form-filling action*. A form element F consists of multiple selectable or editable UI elements that belong to the same form. Forms can be identified by a <form> HTML tag in the DOM tree. If no <form> element exists in the DOM tree, the union of all selectable and editable UI elements in the page is viewed as a form element. A form-filling action, denoted as fill-form(F), is a meta-action representing the process of generating inputs for each element $e \in F$.

Using this terminology, we represent the agent's policy in a two-level hierarchy: a *master policy* $\pi : S \longrightarrow A'$ where A' is the set of click, enter and fill-form actions in the current state, and several *sub-policies* $\pi_i : S \longrightarrow A_i$, where each sub-policy π_i corresponds to a form element F_i with A_i being the set of select and type actions in F_i. The master policy π drives the overall navigation process by picking actions from A'. If the fill-form(F_i) action is selected, π_i takes over until F_i is completed.

Algorithm 2: LearnForm: Train sub-policy for a form.

```
   Input: $F$: the form to fill, $\epsilon$: randomness value
   Hyperparameters: $K$: the number of episodes
   Global variables: $Q_F$: Q network of $F$, $R_F$: reward function for $F$
1  begin
2      create an empty replay buffer $D^{replay}$
3      initialize the set of action sequences $AS$ as $\emptyset$
4      for episode j from 1 to $K$ do
5          initialize the action sequence $as$ as empty
6          for $i$-th element $e$ in $F$ do
7              set $A_i$ as the set of possible actions for $e$
8              generate a random number k in the range $[0.0, 1.0)$
9              if $k < \epsilon$ then
10                 randomly choose an action $a_i \in A_i$
11             else
12                 select an action $a_i = argmax_{a \in A_i} Q_F(s_i, a)$
13             compute $r_i$ with $R_F$ and get $s_{i+1}$ by simulating $a_i$ in $s_i$
14             put $<s_i, a_i, r_i, s_{i+1}>$ into $D^{replay}$
15             put $a_i$ into $as$
16         sample a batch from $D^{replay}$ (if enough entries)
17         perform a gradient descent step to update $Q_F$ (as in Double Q-Learning [67])
18         put $as$ into $AS$
19     return the action sequence $as \in AS$ with the highest reward
```

Master policy. To learn π we use Q-learning [69]. The algorithm (Algorithm 1) trains a function $Q : S \times A' \longrightarrow \mathbb{R}$ to represent the long-term value of taking action a in $s_i \in S$. Q is represented as a mapping table, in which each key is a state-action pair $<s, a>$ and the value is $Q(s, a)$. An action is uniquely identified by its type, value, and target element's xpath, and a state is identified by the identifiers of all actions in the state. To train the Q function, the agent is allowed to try for N episodes and take M steps in each one. To begin, Q is 0 and the agent explores the state randomly. The agent updates Q based on the collected experience and gradually moves from exploration (choosing actions randomly, line 14) to exploitation (choosing actions based on Q values, line 16).

Form sub-policies. We use deep Q-network (DQN) [44] to train the form sub-policies. DQN is similar to Q-learning except that the value function $Q(s, a)$ is approximated with a deep neural network (DNN) instead of a mapping table. For sub-policies we prefer DQN over Q-learning because a DNN can capture commonalities between state-action pairs in forms (while in Q-learning each state-action pair is represented as a unique identifier). For example, the action of typing a pick-up location and typing a drop-off location into the correct text boxes have similar representations in a DNN model, so that the experience about one can be used to learn the other.

The algorithm to train the sub-policy (Algorithm 2) aims to find the best action for each form element that can help the master maximize the cumulative reward. Since the sub-policy only deals with a form and is unable to get the long-term cumulative reward, it trains the Q network using an approximate reward function, R_F (line 13). Specifically, $R_F(q, w, as) = R(q, w, as) + LR_F(as)$. $R(q, w, as)$ is the reward function defined in §3.2 to evaluate whether a form with the filled values has high similarity with q, representing a short-term reward of filling F with as. LR_F is a linear regression model learned by the master policy (line 26 in Algorithm 1) to predict the long-term benefit. The input to $LR_F(as)$ is the concatenation of one-hot encodings of each action value in as, and the output is the difference between the maximum reward R_{max} achieved by the master policy after filling F with as and $R(q, w, as)$ (LR_F is large if F is the correct form and as is the correct sequence of inputs).

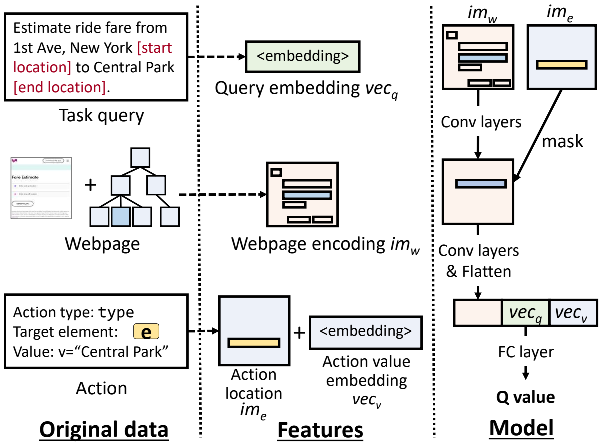


Figure 5: The Q network used in the sub-policy.

Another benefit of sub-policies for forms is that since interacting within a form does not lead to page re-directions (which instead can happen with the master policy), we can explore a form by *simulating* its possible state transitions. For example, the state/reward obtained after executing type(e, "Central Park") in s can be simulated by simply setting the value of e to "Central Park" in s. Through simulation, we can run many more episodes to train the sub-policy.

Q-network implementation. The Q network used to train the sub-policy is shown in Figure 5. The input consists of features extracted from the task query q and the state-action pair $<s,a>$. The task query q is encoded as a vector vec_q, which is the mean of the embeddings of all non-parameter words in q (computed using spaCy [58]). The state s consists of a webpage w and is encoded as a single-channel image im_w, in which the pixels in each element e are set to the similarity between e_{text} and p_{ann}. An action a is represented with its target element e, type, and input value v (the entered text or the selected value). Suppose p is the parameter in q that has the highest similarity with v, and p_{ann} is the annotation of p; v's encoding vec_v is the concatenation of the embeddings of v, p and p_{ann}. If no parameter p has similarity with v above a threshold (e.g., 0.5), vec_v is empty. An element e is encoded as a single-channel image im_e in the same scale of the webpage, in which the pixels are set to 1 inside e and 0 elsewhere. To reduce model size, we resize im_e and im_w to 100×100 pixels.

im_w is passed through three 7×7 convolutional layers to spread the similarity information across elements. Then, the feature map is masked with im_e to exclude information outside the target element, and converted to a vector using four 3×3 convolutional layers. Finally, the vector is concatenated with vec_q and vec_v and processed with a fully connected (FC) layer to generate the Q value.

3.4 Tasklet generation

We now describe how the actual output of Glider is assembled. The RL agent may succeed at discovering a tasklet, but with unnecessary actions interleaved with good ones; or it may produce an incomplete tasklet. The goal of the *post-processing* module is to analyze the agent's learning trace (consisting of N episodes) to identify correct tasklets and possibly fix failing ones. It is a 4-step process. (1) Based on the final cumulative reward, the k highest-rewarded

Table 1: Details for Glider-int: 164 unique task-website pairs covering 10 site categories, 40 task queries and 66 websites.

Category	#tests	Example of test case (q and url pair)
Restaurants	16	q=Reserve a table for 4 people at 8pm in Boston, url=opentable.com
Transport	14	q=Estimate ride fare from 1st Ave, New York to Central Park, url=uber.com
Real estate	20	q=Rent a house in Seattle with 3 bedrooms, url=rent.com
Shopping	17	q=Find women shoes, url=ebay.com
Books	24	q=Find books by Umberto Eco, sort by publication date, url=aadl.org
Academic	18	q=Citations of “ImageNet: A Large-Scale hierarchical...”, url=scholar.google.com
Dictionary	18	q=Translate intelligent from English to Spanish, url=translate.com
Tools	16	q=Convert length from 7333 inches to feet, url=unitconversion.org
Flights	16	q=Flights from BOS to SFO departing 09/01 for 2 adults, url=aa.com
Health	5	q=Lookup canceled dermatology appointments at laboratory, url=openmrs.org
Total	**164**	

task executions are selected (by default $k = 5$). (2) Any action that produced a negative reward is removed after verifying that the new action sequence can be successfully replayed and yields a higher reward. (3) Possibly missing “submit” actions usually occurring at the end of a task execution (e.g., missing enter action) are automatically injected and tested. (4) The final sequence of actions is replayed to verify whether the tasklet works and a screenshot of the page obtained at task completion is captured. The final output consists of at most k tasklets ranked by their cumulative reward. For each tasklet, the output contains partial and total rewards, the replay trace to execute the tasklet (example in Figure 4), and a screenshot of the result page. DOM elements are currently identified by xpaths, but other invariants (e.g., id, class, name, etc.) could also be included.

4 EVALUATION

We evaluate Glider through *(i)* an assessment of how well it can extract correct tasklets compared to two baselines, *(ii)* an analysis of its varying configurations, and *(iii)* a small developer study.

4.1 Experimental setup

Datasets. Since no standard dataset existed to test Glider, we collected two new datasets. Compared to previous work [19, 40], our datasets consist of more complex tasks on real websites.[4]

Initially, we collected an internal dataset, Glider-int, as follows. We chose 10 site categories and wrote website-independent task descriptions relevant to each category for a total of 40 unique queries. We chose tasks that are common in the web, with a medium range of complexity. Then, we selected various websites for each category based on popularity (Google’s rank in search results). We dropped websites that blocked our agent or Selenium – this happened mainly with flight and real estate websites. In total, we selected 66 websites. We combined task descriptions with websites supporting such tasks[5] and obtained 164 test cases, each one corresponding to a unique query-website pair. Table 1 summarizes the distribution of tasks across categories. Overall, the generated tests covered a good range of complexity. The length of a task measured as the number of steps (UI interactions) necessary to complete the task ranged from 2 to 9 (avg=3.8) and the number of different webpages visited to complete the task ranged from 1 to 6 (avg=1.9). In terms of action space size, across all tests an average webpage had 76.4 actions.

We also collected an external dataset, Glider-dev, produced through a small developer study (more details in §4.4). It consists of 44 test cases in the same site categories as above.

Methodology. 20 of the 164 test cases in Glider-int were used for setting hyperparameters and reward weights; all other test cases were introduced at evaluation time (Glider does not require any pre-training). We executed all tests online, on a cluster of GPU nodes (NVidia Tesla K40 and M40 with 12GB of RAM) and manually inspected the outputs consisting of maximum k=5 top-rewarded tasklets and screenshots of their final states. Despite Glider being automated, manually inspecting the outcome of each test limited the number of task-website combinations we could test.[6] A tasklet was deemed correct if the actions completed the goal described in the query in maximum M=5+$len(split(q))$ steps.

Settings. Tests were executed for N=100 episodes with M steps per episode for the master policy and K=20 episodes for the sub-policies. The duration of an episode for the master policy was usually around 18 seconds. Unless otherwise noted, the reward weights were set as follows: w_{dist}=w_{dir}=-2, w_{task}=5, and w_{par}=10.

Metrics. We compute the recall@k (**R@k**) by manually inspecting the top k tasklets returned. We compute it for $k = 1$ and $k = 5$.

Baselines. A direct comparison of Glider with other related approaches is not possible, since they differ in the type of supervision (large number of demonstrations and/or pre-training), website environments (MiniWoB [56] instead of real websites), or inputs needed (step-by-step instructions instead of a task query). Nonetheless, we benchmark Glider’s performance using the following two search algorithms which meet the requirements outlined in §3.3:

(i) Hill climbing is a local search algorithm. At each step, it picks the action that gives the highest reward, and repeats until there are no better moves. In our settings, since the reward of an action is unknown beforehand, we approximate it as the similarity between an action and the task description (i.e, $\max_{tw \in q}\{sim(tw, a_{text})\}$, where a_{text} is the text associated with the interacted UI element).

(ii) Monte Carlo tree search (MCTS) is a heuristic search algorithm. The decision process is modeled as a tree and the goal is to move from the root to a leaf node that can maximize the outcome by picking the most promising action at each step. We used the standard *UCT* formula [29]. In each state s_i, the action a_i with the highest *Upper Confidence Bound* $R_i/n_i + 2\sqrt{(lnN_i)/n_i}$ is chosen, where R_i is the maximum cumulative reward among all past episodes in which a_i is taken, n_i is the number of episodes in which a_i is taken, and N_i is the number of episodes in which s_i is visited.

[4] Liu et al. [40] and Gur et al. [19] test with 53 and 14 MiniWob [56] tasks, respectively. MiniWob queries (e.g., “Click the red box”) describe UI actions rather than end goals (e.g., “Rent a house”), hence are not compatible with Glider’s reward approach.

[5] For example, a website for searching restaurants may or may not support making reservations, hence the search task but not the reservation task may be tested on it.

[6] To automate the tests we tried recording demonstrations for each task-website pair but comparing Glider’s output with them turned out unreliable because demonstrations age quickly (due to website and content updates) and capture only one valid path. To freeze the test environment we tried building a cache, but even after saving 2,000+ states per site we had many cache misses because too many execution paths existed.

Table 2: Performance of Glider, MCTS and Hill climbing with the Glider-int **dataset.**

Category	Glider		MCTS		HillClimb	
	R@1	R@5	R@1	R@5	R@1	R@5
Restaurants	75.0	81.3	68.8	68.8	56.3	62.5
Transport	78.6	92.9	64.3	78.6	78.6	85.7
Real estate	75.0	90.0	60.0	65.0	40.0	65.0
Shopping	52.9	82.4	52.9	58.8	47.1	82.4
Books	50.0	70.8	37.5	45.8	33.3	45.8
Academic	55.6	66.7	22.2	33.3	27.8	44.4
Dictionary	50.0	72.2	44.4	66.7	38.9	77.8
Tools	68.8	81.3	31.3	43.8	31.3	43.8
Flights	75.0	81.3	43.8	56.3	18.8	50.0
Health	60.0	60.0	40.0	40.0	0.0	40.0
Summary	**63.4**	**78.7**	**46.3**	**56.1**	**39.0**	**60.4**

4.2 Performance results

Main results. We executed Glider and the two baselines on Glider-int. As Table 2 shows, Glider achieved R@1 of 63.4% and R@5 of 78.7%. The most successful categories were Restaurants, Transport, Real estate, and Flights where R@5 was on average 86.4%. Glider outperformed both baselines which achieved R@1 of 39–46% (R@5 was 56–60%),[7] thus confirming the effectiveness of our approach. MCTS would have likely performed better with more episodes, but on large webpages so many episodes are not realistic.

Task and website complexity. Success rates slightly increased for websites with more actions and UI elements (top 2 graphs in Figure 6) as these sites were more informative to compute rewards. However, the performance slightly dropped when there was too much noise (more than 240 UI elements per state). Tasks that spanned across multiple pages were more difficult (R@1 was 39.1% for 3-page tasks) as multi-page tasks usually imply larger action spaces and harder reward computations. Shorter tasks achieved higher recall (R@5 was 92.5% for tasks that can be completed in 2 steps); however, tasks that required more than 5 steps were not the most difficult because they usually involved large forms that were effectively handled by the form sub-policies.

Error analysis. We inspected the 35 tasks for which a correct tasklet did not appear in the top 5. We identified four failure causes. *(1) Reward model (21 tasks)*: the correct tasklet was not the highest-rewarded although it was explored by the agent. A wrong action (e.g., clicking a wrong button that has high similarity with the task query) may get over-rewarded whereas a correct one may be under-rewarded (e.g., in the task "Find papers that cited...", clicking "cited by" caused a transition to a page that had low similarity with the query). *(2) Widget support (6 tasks)*: the agent failed to interact with some date pickers, counters and drop-down menus. Date pickers are particularly hard because the apparently-simple task of setting a date actually involves many steps. Glider currently handles only date pickers that support typing. *(3) Website incompatibility (5 tasks)*: some critical actions were problematic for Selenium, e.g., DOM elements embedded in an <iframe>. *(4) Algorithm issues (3 tasks)*: due to the limited time budget, the agent did not explore enough.

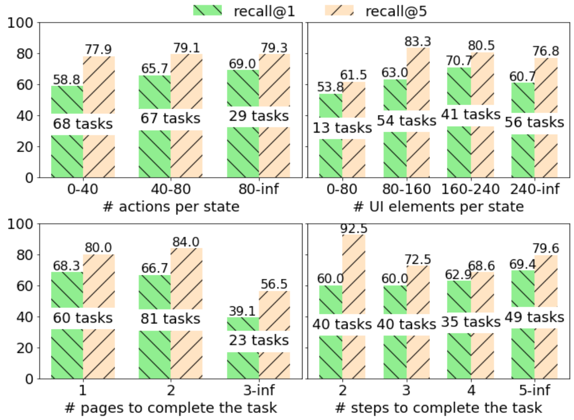


Figure 6: Recall rates with varying task/website complexity.

4.3 Varying configuration analysis

Empirically, we found an optimal configuration (Glider-default) for the reward weights (w_{dist}=w_{dir}=-2, w_{task}=5 and w_{par}=10). In this test, we varied one reward weight at the time and measured the impact on recall. We also conducted an ablation analysis by testing Glider with the form sub-policies disabled, with query annotations omitted, and with its similarity function replaced by semantic similarity (implemented using spaCy [58]). We tested 30 cases, 3 randomly-selected from each category in Glider-int. Table 3 reports R@k normalized by the corresponding R@k of Glider-default.

Sub-policy. Without sub-policies, the agent uses one uniform policy which degraded R@1 by 79%. This demonstrates the need for hierarchical policy, especially in the presence of large forms.

Query annotations. When ignoring annotations, R@1 dropped by 36% and R@5 by 19%. Reward computation was less accurate, and UI elements for query parameters were harder to identify.

Semantic similarity. Semantic similarity caused R@1 to drop by 50% and R@5 by 37%. A task like "Translate intelligent from English to Spanish" failed because "Spanish" was considered similar to "English" so it was never selected. To be effective semantic similarity may require to be fine tuned on a per task/category basis.

Reward weights. Overall, our default configuration achieved the best performance. The worst recall was for w_{par}=0, confirming the importance of parameter similarity. Disabling action locality (w_{dist}) and directionality (w_{dir}) decreased R@1 by 43–50%, while w_{task}=0 was less harmful because the parameters and annotations were often enough to match the pages with the task queries.

4.4 Developer study

We tested Glider also with real user queries (Glider-dev dataset). We recruited 5 developers who had not heard about Glider before. We gave them a one-sentence description of Glider and a sample task query and website. Then, we asked them to describe a task in each of our 10 site categories and provide a website for its execution.

Our developers reported that defining the tasks was easy and took less than a couple of minutes per task. Despite the little information provided and lack of prior experience with Glider, the quality of their queries was comparable to that of our internal dataset.

[7] We also tested the Random Walk baseline introduced in §2. Due to space constraints we do not report its performance, but its overall success rate was poor (R@1=14.6%).

Table 3: Normalized R@k for different configurations. Results based on 30 test cases from Glider-int**.**

Condition	R@1	R@5	Condition	R@1	R@5
Glider-default	1	1	w_{dir}=0	0.50	0.88
without sub-policy	0.21	0.88	w_{dir}=-5	0.79	0.88
without annotations	0.64	0.81	w_{task}=0	0.86	1
with semantic sim	0.50	0.63	w_{task}=10	0.86	0.94
w_{dist}=0	0.57	0.88	w_{par}=0	0	0
w_{dist}=-5	0.93	0.94	w_{par}=5	0.79	0.94

Success rates. From the 50 developers' query-website pairs, we excluded Selenium-incompatible websites and obtained a total of 44 task queries across 41 websites (with 17 websites overlapping with Glider-int). We ran Glider on them. Glider yielded an average R@1 of 65.9% and R@5 of 77.3%, similar to the Glider-int results. In addition to the failure causes previously identified, we observed two others. One developer wrote the query "Search for a recent paper on ..." and expected the agent to sort the results by date. Contextual expressions of this type cannot yet be resolved. Two other queries required entering multiple parameters into the same input field ("Buy a patagonia [brand] fleece [product] in size large [size]" required typing "patagonia" and "fleece" into the same field).

5 RELATED WORK

We are not aware of any other unsupervised approach to extract tasklets from current websites. In the following, we discuss work related to web automation and summarize its limitations.

Record and replay. "Programming by demonstration" is a popular approach in web automation tools [2–4, 6, 24, 25, 28, 33, 35, 38, 39, 50, 52, 63, 66, 73], where users record themselves interacting with a website and obtain a script to replay the interaction. Current work focuses on making replay resilient to UI changes [4] and vision-assisted [5, 54]. Overall, the manual effort required to record and maintain UI scripts makes these tools unsuitable for our goal.

Program synthesis. These techniques synthesize a program that satisfies a given specification, usually a logical specification that relates the inputs and outputs of the program [17]. They are mainly used for data or code transformation tasks [11, 21, 68] (e.g., transformations of Excel columns [18]). LaSEWeb [48] proposes a domain-specific language to automate search tasks, but the effort and technical skills required make it unsuitable for our goal.

Mapping instructions to actions. The problem of interpreting and executing NL instructions has long been studied [12, 13, 34, 64, 72]. Previous work has explored how to automatically generate executable actions from how-to documentation [7, 34, 64] and bug reports [14, 75]. While the goal is similar, these techniques are not widely applicable because *(i)* they assume how-to documents that list "steps" to reproduce a problem where each step maps to a UI action, *(ii)* they use a vocabulary which is tied to the app's UI, or *(iii)* they rely on app-specific model training [75] or static analysis [14]. In contrast, Glider does not require any per-app pre-processing, and its NL inputs do not follow any step structure and are UI oblivious.

AI agents for web navigation. These techniques do not generate UI scripts, but instead train AI agents that can execute web tasks described in natural language [1, 19, 40, 56, 59]. To learn a task, agents require human demonstrations and/or the final state of a task, which must be collected manually. The training overhead of these solutions makes them hard to scale. Moreover, all AI agents proposed so far [19, 40, 56] have been designed and tested on MiniWoB [56] websites that consist of small mock HTML pages. In contrast, Glider works on real websites and requires no pre-training.

Beyond these four bodies of work, RL agents for (deep) web crawling [20, 23, 26, 42, 49] are related to Glider, but also substantially different because *(i)* rewards are based on the number of crawled pages (not on task completion) and *(ii)* agents interact mainly with search boxes and hyperlinks [26, 42, 49]. Our work fits in the general theme of web knowledge extraction [9, 16, 22, 46, 60, 70] and extends it to actions and tasks. Finally, Glider leverages various RL techniques including DQN [44], prioritized experience replay [51], double Q learning [67], and hierarchical RL [31].

6 LIMITATIONS AND FUTURE WORK

GUI coverage. Glider does not support sliding bars, some types of date picker, and other customized widgets – sub-policies could be trained for each one of these. Scroll, drag, and double click actions are also not supported. Glider uses text and font of DOM elements, but a more comprehensive structural analysis of the DOM tree could be used to classify actions and their inter-dependencies.

Full automation. To detect failing tasklets due to Glider's errors or website updates a developer can inspect Glider's output including the captured screenshots. However, erroneous tasklets should be detected automatically by, for instance, comparing the achieved reward with an "expected reward score" computed based on the task query. Tasklets should also be updated on a regular basis.

Tasklet expansion. Extracted tasklets could be more comprehensive. For example, given the query "*Find a house with 2 bedrooms*", Glider could extract a tasklet that supports the input "number of bedrooms" but also "number of bathrooms", "price range", etc., which can be inferred using UI and DOM tree analysis.

False tasklets. So far we have tested tasklet extraction on websites *within* the task scope, but, in the future, Glider should detect whether a tasklet can or cannot be found in *any* website.

Ethical concerns. As with web crawlers and bots [65], Glider can cause denial of service, cost, privacy and copyright issues. Website owners should have the option to stop Glider from visiting some or all of their pages, as they stop web crawlers using robots.txt [41] and other metatags [62]. Glider should also act responsibly without assuming site owners have taken protective measures, such as by reducing crawling speeds and automatically constraining or blocking interactions with critical UI controls (e.g., payments).

7 CONCLUSION

We envision a future where AI assistants will be able to automatically use the Internet for executing human tasks. To accomplish this vision, we introduce Glider for automatically generating tasklets for current websites. While performance can be further improved, we demonstrated the approach is feasible and a generalized reward model can help tasklet generation scale. Glider requires neither modifications to existing websites nor website-specific pre-training. In the future, site owners may decide to annotate their websites to aid tasklet discovery (as with Schema.org tags for structured data).

REFERENCES

[1] James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. PLOW: A Collaborative Task Learning Agent. In *Proc. of AAAI'07*. 1514–1519.

[2] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. 2000. Automating Web Navigation with the WebVCR. In *Proc. of the 9th International World Wide Web Conference on Computer Networks*. 503–517.

[3] Automation Anywhere. 2021. https://www.automationanywhere.com.

[4] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Proc. f the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 748–764. https://doi.org/10.1145/2983990.2984020

[5] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating Video Recordings of Mobile App Usages into Replayable Scenarios. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, 309–321. https://doi.org/10.1145/3377811.3380328

[6] Blue Prism. 2021. https://www.blueprism.com.

[7] S. R. K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. 2009. Reinforcement Learning for Mapping Instructions to Actions. In *Proc. of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1* (Suntec, Singapore) *(ACL '09)*. Association for Computational Linguistics, USA, 82–90.

[8] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Universal Sentence Encoder. *CoRR* abs/1803.11175 (2018). arXiv:1803.11175 http://arxiv.org/abs/1803.11175

[9] Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom Mitchell, Kamal Nigam, and Seán Slattery. 2000. Learning to Construct Knowledge Bases from the World Wide Web. *Artif. Intell.* 118, 1–2 (April 2000), 69–113. https://doi.org/10.1016/S0004-3702(00)00004-7

[10] Creative Bloq. 2015. 5 ways to make your web designs more intuitive. https://www.creativebloq.com/web-design/5-ways-make-your-web-designs-more-intuitive-91516938.

[11] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program Synthesis Using Natural Language. In *Proc. of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 345–356. https://doi.org/10.1145/2884781.2884786

[12] Barbara Di Eugenio. 1992. Understanding Natural Language Instructions: The Case of Purpose Clauses. In *Proc. of ACL '92*. 120–127. https://doi.org/10.3115/981967.981983

[13] Oren Etzioni and Daniel Weld. 1994. A Softbot-Based Interface to the Internet. *COMMUNICATIONS OF THE ACM* 37 (1994), 72–76.

[14] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically Translating Bug Reports into Test Cases for Mobile Apps. In *Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, 141–152. https://doi.org/10.1145/3213846.3213869

[15] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In *Proc. of the 43rd Annual Meeting on Association for Computational Linguistics (ACL '05)*. Association for Computational Linguistics, 363–370. https://doi.org/10.3115/1219840.1219885

[16] John Foley, Michael Bendersky, and Vanja Josifovski. 2015. Learning to Extract Local Events from the Web. In *Proc. of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Santiago, Chile) *(SIGIR '15)*. Association for Computing Machinery, 423–432. https://doi.org/10.1145/2766462.2767739

[17] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proc. of PPDP '10*. 13–24. https://doi.org/10.1145/1836089.1836091

[18] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proc. POPL '11*. 317–330. https://doi.org/10.1145/1926385.1926423

[19] Izzeddin Gur, Ulrich Rueckert, Aleksandra Faust, and Dilek Hakkani-Tur. 2019. Learning to Navigate the Web. In *Proc. of ICLR '19*.

[20] Miyoung Han, Pierre-Henri Wuillemin, and Pierre Senellart. 2018. Focused Crawling Through Reinforcement Learning. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). 261–278.

[21] William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. *SIGPLAN Not.* 46, 6 (June 2011), 317–328. https://doi.org/10.1145/1993316.1993536

[22] Y. L. Hedley, M. Younas, A. James, and M. Sanderson. 2004. Query-Related Data Extraction of Hidden Web Documents. In *Proc. of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '04)*. Association for Computing Machinery, 558–559. https://doi.org/10.1145/1008992.1009119

[23] Inma Hernández, Carlos R. Rivero, and David Ruiz. 2018. Deep Web crawling: a survey. *World Wide Web* (05 Jun 2018). https://doi.org/10.1007/s11280-018-0602-1

[24] Darris Hupp and Robert C. Miller. 2007. Smart Bookmarks: Automatic Retroactive Macro Recording on the Web. In *Proc. of UIST '07*. 81–90.

[25] iMacros. 2021. https://imacros.net/.

[26] Lu Jiang, Zhaohui Wu, Qian Feng, Jun Liu, and Qinghua Zheng. 2010. Efficient Deep Web Crawling Using Reinforcement Learning. In *Proc. of PAKDD'10*. 428–439. https://doi.org/10.1007/978-3-642-13657-3_46

[27] Aishwarya Kamath and Rajarshi Das. 2018. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).

[28] Katalon Studio. 2021. https://www.katalon.com.

[29] Levente Kocsis and Csaba Szepesvàri. 2006. Bandit based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML'06)*. Springer, 282–293.

[30] N. Kohl and P. Stone. 2004. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proc. of ICRA '04*, Vol. 3. 2619–2624 Vol.3. https://doi.org/10.1109/ROBOT.2004.1307456

[31] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. 2016. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*. 3675–3683.

[32] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360* (2016).

[33] Tessa Lau, Julian A. Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. 2010. A Conversational Interface to Web Automation. *UIST 2010 - 23rd ACM Symposium on User Interface Software and Technology*, 229–238. https://doi.org/10.1145/1866029.1866067

[34] Tessa Lau, Clemens Drews, and Jeffrey Nichols. 2009. Interpreting Written How-to Instructions. In *Proc. of the 21st International Jont Conference on Artifical Intelligence* (Pasadena, California, USA) *(IJCAI '09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1433–1438.

[35] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proc. of CHI '08*. 1719–1728.

[36] V. I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10, 707–710.

[37] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2015. End-to-End Training of Deep Visuomotor Policies. *CoRR* abs/1504.00702 (2015). http://arxiv.org/abs/1504.00702

[38] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proc. of CHI '10*. 723–732.

[39] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proc. of CHI '17*. 6038–6049. https://doi.org/10.1145/3025453.3025483

[40] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 2018. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. In *Proc. of the 6th International Conference on Learning Representations, ICLR 2018*. http://arxiv.org/abs/1802.08802

[41] M. Koster. 1994. A standard for robot exclusion. https://www.robotstxt.org/orig.html.

[42] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. 2000. Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval* 3, 2 (01 Jul 2000), 127–163. https://doi.org/10.1023/A:1009953814988

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. (2013). http://arxiv.org/abs/1312.5602

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533. http://dx.doi.org/10.1038/nature14236

[45] OpenMRS. 2021. Open source enterprise electronic medical record system. https://openmrs.org/demo/.

[46] Panupong Pasupat and Percy Liang. 2014. Zero-shot Entity Extraction from Web Pages. In *Proc. of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 391–401. https://doi.org/10.3115/v1/P14-1037

[47] PlayWright. 2021. A Node library to automate Chromium, Firefox and WebKit with a single API. https://playwright.dev/.

[48] Oleksandr Polozov and Sumit Gulwani. 2014. LaSEWeb: Automating Search Strategies over Semi-structured Web Data. In *Proc. of KDD '14*. 741–750. https://doi.org/10.1145/2623330.2623761

[49] Jason Rennie and Andrew McCallum. 1999. Using Reinforcement Learning to Spider the Web Efficiently. In *Proc. of ICML '99*. 335–343.

[50] Alex Safonov, Joseph A. Konstan, and John V. Carlis. 2001. End-user Web Automation: Challenges, Experiences, Recommendations. In *Proc. of WebNet 2001*. 1077–1085.

[51] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[52] Selenium. 2021. Selenium IDE. https://www.selenium.dev/selenium-ide/.

[53] Selenium. 2021. Selenium WebDriver. https://www.selenium.dev/documentation/en/webdriver/.

[54] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. 2020. VASTA: A Vision and Language-Assisted Smartphone Task Automation System. In *Proc. of the 25th International Conference on Intelligent User Interfaces (IUI '20)*. ACM, 22–32. https://doi.org/10.1145/3377325.3377515

[55] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. 2014. Learning Semantic Representations Using Convolutional Neural Networks for Web Search. In *Proc. of WWW '14*. 373–374. https://doi.org/10.1145/2567948.2577348

[56] Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. 2017. World of Bits: An Open-Domain Platform for Web-Based Agents. In *Proc. of the 34th International Conference on Machine Learning*, Vol. 70. 3135–3144.

[57] 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

[58] spaCy. 2021. https://spacy.io/models/en.

[59] Shashank Srivastava, Oleksandr Polozov, Nebojsa Jojic, and Christopher Meek. 2020. Learning Web-based Procedures by Reasoning over Explanations and Demonstrations in Context. In *Proc. of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7652–7662. https://doi.org/10.18653/v1/2020.acl-main.684

[60] Fei Sun, Dandan Song, and Lejian Liao. 2011. DOM Based Content Extraction via Text Density. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '11)*. Association for Computing Machinery, 245–254. https://doi.org/10.1145/2009916.2009952

[61] Tech Crunch. 2019. Google is bringing AI assistant Duplex to the web. https://techcrunch.com/2019/05/07/google-is-bringing-ai-assistant-duplex-to-the-web/.

[62] TecSmash. 2019. How To Stop Googlebot From Crawling Your Website? Proven Methods. https://tecsmash.com/how-stop-google-bot-crawling-your-website/.

[63] Test Complete. 2021. https://smartbear.com/product/testcomplete/overview/.

[64] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating Test Automation. In *Proc. of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 881âĂŞ891.

[65] Milena Tsvetkova, Ruth García-Gavilanes, Luciano Floridi, and Taha Yasseri. 2016. Even Good Bots Fight: The Case of Wikipedia. *CoRR* abs/1609.04285 (2016). arXiv:1609.04285 http://arxiv.org/abs/1609.04285

[66] UiPath. 2021. https://www.uipath.com/.

[67] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proc. of AAAI '16*. 2094–2100.

[68] Ashwin J. Vijayakumar, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *CoRR* abs/1804.01186 (2018). arXiv:1804.01186 http://arxiv.org/abs/1804.01186

[69] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[70] Gerhard Weikum, Johannes Hoffart, and Fabian M Suchanek. 2016. Ten Years of Knowledge Harvesting: Lessons and Challenges. *IEEE Data Engineering Bulletin* 39, 3 (September 2016), 41–50. https://publications.cispa.saarland/965/

[71] Ryen W. White, Ahmed Hassan Awadallah, and Robert Sim. 2019. Task Completion Detection: A Study in the Context of Intelligent Systems. In *Proc. of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Paris, France) *(SIGIR'19)*. Association for Computing Machinery, New York, NY, USA, 405–414. https://doi.org/10.1145/3331184.3331187

[72] Terry Winograd. 1972. *Understanding Natural Language.* Academic Press, Inc.

[73] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proc. of the 22nd Annual ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, 183–192. https://doi.org/10.1145/1622176.1622213

[74] Hongfei Zhang, Xia Song, Chenyan Xiong, Corby Rosset, Paul N. Bennett, Nick Craswell, and Saurabh Tiwary. 2019. Generic Intent Representation in Web Search. In *Proc. of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Paris, France) *(SIGIR'19)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/3331184.3331198

[75] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 128–139. https://doi.org/10.1109/ICSE.2019.00030