Agent+P: Guiding UI Agents via Symbolic Planning

Anonymous ACL submission

Abstract

Large Language Model (LLM)-based UI agents show great promise for UI automation but often hallucinate in long-horizon tasks due to their lack of understanding of the global UI transition structure. To address this, we introduce AGENT+P, a novel framework that leverages symbolic planning to guide LLM-based UI agents. Specifically, we model an app's UI transition structure as a UI Transition Graph (UTG), which allows us to reformulate the UI automation task as a pathfinding problem on the UTG. This further enables an off-the-shelf symbolic planner to generate a provably correct and optimal high-level plan, preventing the agent from redundant exploration and guiding the agent to achieve the automation goals. AGENT+P is designed as a plug-and-play framework to enhance existing UI agents. Evaluation on the AndroidWorld benchmark demonstrates that AGENT+P improves the success rates of state-of-the-art UI agents by up to 14.31% and reduces the action steps by 37.70%. Our code is available at: https://anonymous.4open.science/r/agentp-F7AF.

1 Introduction

With mobile applications (apps) woven into all parts of our daily life, it is critically important to ensure the high quality of apps. User interface (UI) automation, the process of programmatically executing sequences of UI interactions, has become an essential method for improving app quality by enabling automated testing for bug and vulnerability detection (Lai and Rubin, 2019; Ma et al., 2025) and supporting user task automation (Orrù et al., 2023; Rawles et al., 2024; Li et al., 2025b; Zhang et al., 2023).

While recent advances in UI automation, notably the integration of LLM-based UI agents (Liu et al., 2025; Zhao et al., 2024; Ran et al., 2024) that explicitly model the available actions in each UI screen, have demonstrated encouraging results, the ever-growing complexity of modern UIs continues to hinder effective and efficient automation. In particular, existing approaches struggle in **long-horizon planning** tasks that require navigating via multiple UIs since such multi-step planning often leads to increased hallucination rates (Liu et al., 2023; Wei et al., 2025; Xie et al., 2025; Wu et al.). For example, the LLM-based agents employed by these approaches typically follow a depth-first strategy to find valid action sequences, making decisions based on local UI states without understanding the global transition structure, where different UIs can lead to distinct subsequent actions. Consequently, they often fail to derive valid sequences that accomplish task goals and repeatedly waste effort on actions that diverge from those goals.

Key Insights. To address this fundamental limitation, *we introduce an external planner module that provides LLM-based UI agents with global transition knowledge extracted through program analysis. This module leverages established planning algorithms to prevent redundant exploration and guide the agent toward diverse action sequences that are more likely to achieve the automation goals*. Specifically, we model an app's global transition structure via a UI Transition Graph (UTG) (Sun et al., 2025; Wen et al., 2024), with nodes representing UIs and edges representing user-triggered UI transitions. A UI automation problem, navigating from a start UI to a target UI, can thus be formulated as a *pathfinding problem* on the UTG, where the objective is to find an optimal path from the start node to the target node. This formalization enables the use of off-the-shelf symbolic planners to derive provably correct and optimal plans, thereby fundamentally mitigating high-level planning hallucinations and enhancing the reliability of LLM-based UI agents.

Our Method. Building upon these insights, in this paper, we introduce AGENT+P, an agentic framework that leverages symbolic planning to guide UI

agents. Given a natural language UI automation goal and the app for automation, AGENT+P operates iteratively in four stages, each executed by an LLM-based module, until the goal is achieved: the **UTG Builder** constructs a static UTG of the app and dynamically updates it during automation. The **Node Selector** maps the natural language goal to a targeted node in the UTG. The **Plan Generator** translates the UTG into a pathfinding problem using Planning Domain Definition Language (PDDL) (Aeronautiques et al., 1998), which is then solved by an external symbolic planner. The resulting symbolic plan is subsequently converted into natural language instructions. Finally, the **UI Explorer** interacts with the app to execute the translated instructions to navigate to the goal.

AGENT+P is designed as a plug-and-play planning framework that can be incorporated with and enhance existing UI agents. We evaluate AGENT+P by integrating it with four state-of-the-art agents on the AndroidWorld benchmark. Our results demonstrate that by leveraging symbolic planning on the UTG, AGENT+P increases the success rates of baseline agents by up to 14.31% and reduces the action steps by 37.70%.

Our primary contributions are as follows:

- We propose AGENT+P, a novel framework that leverages symbolic planning to provide LLM-based UI agents with global transition information derived from program analysis, mitigating the long-horizon planning failures commonly faced by these agents.
- We present a novel formalization that maps the problem of UI automation into a pathfinding problem in the UTG, making it solvable with provably correct symbolic planners.
- We conduct extensive evaluation on the AndroidWorld benchmark, demonstrating that AGENT+P substantially improves the success rate and efficiency of three state-of-the-art UI agents.

2 Background and Motivation

2.1 UI and UI Transition Graph

To motivate our method, we begin by introducing the concepts of widget, UI, and the modeling of UI transitions (i.e., UTG).

Definition 1 (Widget, Action)**.** A *widget,* denoted as w, is a basic interactive element on a UI screen. An *action,* denoted as a, is a 2-tuple $a = (w, e)$, where w is a widget, e is the user event (e.g., `click`, `input`).

Table 1: Average number of UTG nodes and edges for apps. $SR_{app} > \overline{SR}$ represents apps where the agent's success rate exceeds its overall average, while $SR_{app} \leq \overline{SR}$ represents apps where it underperforms.

	Nodes		Edges	
Agent	$SR_{app} > \overline{SR}$	$SR_{app} \leq \overline{SR}$	$SR_{app} > \overline{SR}$	$SR_{app} \leq \overline{SR}$
DroidRun	27.0	71.2	62.7	168.0
LX-GUIAgent	29.3	53.8	65.9	129.2
AutoGLM	39.4	42.0	90.6	100.3
Finalrun	23.8	55.0	59.5	125.6
UI-Venus	18.0	50.7	66.0	108.0

Following existing UI agents that represent a UI as a sequence of widgets and supported actions (Android World, 2025; Ye et al., 2025; Dai et al., 2025; Li et al., 2025b), we define the UI state (shortened as UI) as follows:

Definition 2 (UI)**.** A *UI,* denoted as u, is an n-tuple of all unique actions available on the screen, $u = (a_1, a_2, \ldots, a_n)$, where n is the total number of available actions.

This level of abstraction is sufficient to represent UI transitions while avoiding state explosion (Valmari, 1996).

Definition 3 (UI Transition Graph)**.** *A UTG for an app is a directed graph $G = (\mathscr{U}, \mathscr{T}, \varepsilon)$ that models the transition structure of the app.*

- $\mathscr{U}$ is a finite set of nodes, where each node $u \in \mathscr{U}$ represents a UI u in the app.
- $\mathscr{T} \subseteq \mathscr{U} \times \mathscr{U}$ is a set of directed edges. An edge $(u_i, u_j) \in \mathscr{T}$ represents a transition from UI u_i to UI u_j.
- $\varepsilon : \mathscr{T} \to \mathscr{A}$ is an edge-labeling function. It maps each transition (u_i, u_j) to the action $a = (w, e)$ that triggers it, where the widget w is an element of the source UI u_i.

Figure 2 shows an example of UTG of an Android app named Simple Calendar Pro in AndroidWorld benchmark.

2.2 Motivational Study

To investigate how UI complexity affects agent performance, we conduct a motivational study using the AndroidWorld benchmark (Rawles et al., 2024). This benchmark consists of 116 programmatic tasks across 20 Android apps, where an agent must navigate a given app to satisfy a natural language instruction.

Performance is evaluated using a task-level "success rate". While published results typically report a single average across all tasks, we aim to uncover performance variations across different applications. We define $\overline{SR}$ as the agent's overall

average success rate across the entire benchmark. For any given app, we calculate $\mathrm{SR}_{\mathrm{app}}$, the success rate specific to that app.

Based on these metrics, we classify apps into two categories: those where the agent exceeds its average performance ($\mathrm{SR}_{\mathrm{app}} > \overline{\mathrm{SR}}$) and those where it underperforms ($\mathrm{SR}_{\mathrm{app}} \leq \overline{\mathrm{SR}}$).

Table 1 compares the UI complexity, measured by the number of UTG nodes and edges, between these two categories. Our statistical analysis shows that apps where agents underperform ($\mathrm{SR}_{\mathrm{app}} \leq \overline{\mathrm{SR}}$) have significantly more UTG nodes and edges than those where they succeed ($p = 0.03$). This indicates that existing agents struggle with high-complexity UIs, motivating our approach to leverage the app's transition structure.

3 Problem Formulation

In this section, we first define the problem of targeted UI automation, and how to convert it into an equivalent classical planning problem.

3.1 Problem Definition

With the structure of the app modeled as a UTG, we can now formally define the task of UI automation.

Definition 4 (UI Automation)**.** *UI automation is the process of programmatically executing a sequence of actions $\pi = \langle a_1, a_2, \ldots, a_N \rangle$, where each action $a_i \in \mathscr{A}$.*

In this work, we focus on a specific, goal-oriented variant of this task.

Definition 5 (Targeted UI Automation)**.** *Given an app with an initial UI u_{init} and a target UI u_{target}, the objective is to find a valid sequence of actions $\pi = \langle a_1, a_2, \ldots, a_N \rangle$, where $a_i = (w_i, e_i)$, that navigates the app from u_{init} to u_{target}.*

We formulate targeted UI automation as a pathfinding problem on the UTG. Let $\kappa : \mathscr{A} \to \mathbb{R}^+$ be a cost function that assigns a positive cost to each action, representing computational resources, execution time, or other relevant metrics. The problem is then to find a path from u_{init} to u_{target} that minimizes the total execution cost:

$$\pi^* = \arg\min_{\pi} \sum_{i=1}^{N} \kappa(a_i).$$

To simplify the formulation, in this work, we adopt a uniform action cost. This reduces the cost-minimization task to the classical shortest path problem, where the objective is to find the path from u_{init} to u_{target} with the fewest actions.

Listing 1: Domain PDDL for targeted UI automation.

```
(define (domain utg-automation)
  (:requirements :strips :typing)

  (:types
    node - object
  )

  (:predicates
    (at ?n - node)
    (connected ?from - node ?to - node)
    (visited ?n - node)
    (goal-node ?n - node)
    (goal-achieved ?n - node)
  )

  (:action navigate
    :parameters (?from - node ?to - node)
    :precondition (and
      (at ?from)
      (connected ?from ?to)
    )
    :effect (and
      (not (at ?from))
      (at ?to)
      (visited ?to)
      (when (goal-node ?to) (goal-achieved ?to))
    )
  )
)
```

3.2 Symbolic and Classical Planning

Symbolic planning is a long-standing area of AI concerned with finding a sequence of actions to achieve a predefined goal. The most fundamental and widely studied form is **classical planning** where a planning problem instance, P, is formally described as a tuple $P = \langle \mathscr{D}, s_{init}, \mathscr{G} \rangle$, where $\mathscr{D} = \langle \mathscr{F}, \mathscr{A} \rangle$ is the planning domain. These components are defined as follows:

- **States**: $\mathscr{F}$ is a set of fluents or predicates that describe the properties of the world. A state s is a complete assignment of truth values to all fluents in $\mathscr{F}$. The set of all possible states is the state space $\mathscr{S}$.
- **Initial State**: $s_{init} \in \mathscr{S}$ is the initial state of the world.
- **Goal**: $\mathscr{G}$ is the goal specification, a set of conditions on states. Any state $s \in \mathscr{S}$ that satisfies all conditions in $\mathscr{G}$ is a goal state.
- **Actions**: $\mathscr{A}$ is a set of actions. Each action $a \in \mathscr{A}$ is defined by its preconditions, $\mathrm{pre}(a)$, and its effects, $\mathrm{eff}(a)$. An action can only be executed in a state where its preconditions are met, and its effects describe how the state changes after its execution.

A plan, π, is a sequence of actions $\langle a_1, a_2, \ldots, a_N \rangle$ that transforms the initial state s_{init} into a goal state. This is achieved by applying the actions sequentially, where each action a_i is applicable in the state resulting from the execution of a_{i-1}, and the final state after executing a_N satisfies $\mathscr{G}$.

Table 2: Mapping of UI automation notation to classical planning equivalent.

UI Automation Notation	Classical Planning Equivalent
A UI state (UI) $u \in \mathscr{U}$	A state $s \in \mathscr{S}$ where the predicate $at(u)$ is true
The set of all UIs $\mathscr{U}$	The state space $\mathscr{S}$
The initial UI u_{init}	The initial state s_{init}, defined by $at(u_{init})$
The target UI u_{target}	The goal $\mathscr{G}$, specified by the condition $at(u_{target})$
A UI transition via the edge (u_i, u_j) with label a	A planning action
Precondition: the app is on UI u_i	$pre(a) : at(u_i)$
Effect: the app moves to UI u_j	$eff(a) : \neg at(u_i) \wedge at(u_j)$
A path from u_{init} to u_{target} via the sequence of action π	A plan π

The Planning Domain Definition Language (PDDL) (Aeronautiques et al., 1998) is the standard language for representing such planning problems, typically using two files: a domain file defining $\mathscr{F}$ and $\mathscr{A}$, and a problem file defining s_{init} and $\mathscr{G}$.

3.3 UI Automation to Classical Planning

The pathfinding problem can be naturally cast into a classical planning problem, allowing us to leverage classical planners to compute the solution, i.e., the shortest path. Table 2 illustrates the mapping from the UI automation domain to the classical planning domain, which is elaborated as follows:

- **States**: A planning state s corresponds to the app being at a specific UI u. We can define a predicate $at(u)$ which is true if the app is currently on UI $u \in \mathscr{U}$. The state space $\mathscr{S}$ is the set of all possible UIs, $\mathscr{U}$.
- **Initial State**: The initial state s_{init} is defined by the predicate $at(u_{init})$ being true.
- **Goal**: The goal $\mathscr{G}$ is specified by the condition that the predicate $at(u_{target})$ must be true.
- **Actions**: For each UI transition (u_i, u_j) in the UTG triggered by an action $a = (w, e)$, we define a planning operator with precondition $at(u_i)$ and effects $\neg at(u_i)$ and $at(u_j)$.

Following this formulation, we define a general PDDL domain file template applicable to any targeted UI automation task, as shown in Listing 1. Any specific user task, represented by UTG with a specified target UI, u_{target}, can be translated into a corresponding PDDL problem file via this template. For instance, to solve the "Change the time zone" task in the Simple Calendar Pro app, the UTG from Figure 2 is converted into the problem file presented in Listing 4. This symbolic representation allows us to employ a classical planner to efficiently compute a valid and optimal sequence of actions to navigate the app from the initial UI, u_{init}, to the target, u_{target}.

4 AGENT+P

Figure 1 illustrates the overall architecture of AGENT+P, while the step-by-step workflow is detailed in Algorithm 1. Specifically, given a natural language goal specified by the user, AGENT+P operates through four primary modules that interact in a continuous loop until the goal is achieved (success or failure): the **UTG Builder**, the **Node Selector**, the **Plan Generator**, and the **UI Explorer**. In the following subsections, we elaborate on the design rationale and functionality of each module.

4.1 UTG Builder

Existing methods for constructing a UTG rely on either dynamic analysis (Wen et al., 2024; Sun et al., 2025), which is accurate but often suffers from high cost and incomplete coverage, or static analysis (Azim and Neamtiu, 2013; Yang et al., 2018), which is more comprehensive but can introduce infeasible transitions (Liu et al., 2022).

To overcome these limitations, AGENT+P utilizes a hybrid approach that synergizes both techniques to build UTG. AGENT+P begins by performing static analysis to construct an initial UTG (Line 1 of Algorithm 1), following established practices that track API calls responsible for UI transitions in the app's source code (Yang et al., 2018; Liu et al., 2022).

Based on the environment feedback, the initial UTG is dynamically verified and refined as the UI Explorer explores the app (Line 10 and Line 16) (Zhu et al., 2025b). Specifically, let the current UTG be $G = (\mathscr{U}, \mathscr{T}, \varepsilon)$. During UI automation, an observed transition (u_i, a_{obs}, u_j), where $u_i \in \mathscr{U}$, updates the graph to $G' = (\mathscr{U}', \mathscr{T}', \varepsilon')$ in one of three ways:

- **Update Edge:** An action leads from an existing source node to a target node, but the recorded action does not match the corresponding one in the

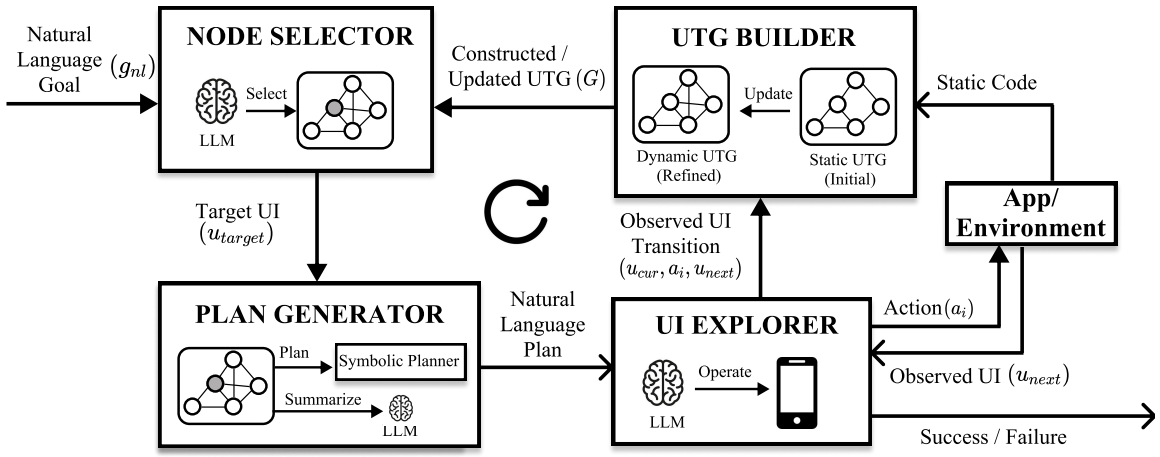

Figure 1: Overview of AGENT+P.

UTG. Formally, if $(u_i, u_j) \in \mathscr{T}$ and $\varepsilon((u_i, u_j)) \neq a_{obs}$, the labeling function is updated such that $\varepsilon'((u_i, u_j)) = a_{obs}$, while $\mathscr{U}' = \mathscr{U}$ and $\mathscr{T}' = \mathscr{T}$.

- **Add Edge:** An action connects two existing UI nodes, but no corresponding edge exists in the UTG. This occurs when $u_j \in \mathscr{U}$ and $(u_i, u_j) \notin \mathscr{T}$. A new transition is added by setting $\mathscr{T}' = \mathscr{T} \cup \{(u_i, u_j)\}$ and extending ε' with $\varepsilon'((u_i, u_j)) = a_{obs}$.
- **Add Node:** An action leads to a UI that is not yet in the UTG. If $u_j \notin \mathscr{U}$, a new node and edge are added to the graph: $\mathscr{U}' = \mathscr{U} \cup \{u_j\}$, $\mathscr{T}' = \mathscr{T} \cup \{(u_i, u_j)\}$, and ε' is extended with $\varepsilon'((u_i, u_j)) = a_{obs}$.

This approach allows AGENT+P to maintain a UTG that is both comprehensive and dynamically accurate, combining the breadth of static analysis with the precision of real-time exploration.

4.2 Node Selector

A critical challenge in AGENT+P is mapping an unstructured, natural language goal (e.g., "Create a playlist") to a concrete UI state (u_{target}) within the app (Line 5). To bridge the semantic gap between user intent and UI elements (Baechler et al., 2024; Li et al., 2021, 2025a), we implement a hierarchical identification strategy that leverages both the reasoning capabilities of LLM and efficient semantic retrieval.

MLLM-based Identification. We query a Multimodal Large Language Model (MLLM) with the user's goal and the representations of candidate UIs. The MLLM is prompted to analyze the screen content and directly identify which UI screen corresponds to the user's objective. The specific prompt template used for this reasoning process is detailed in Listing 5.

Algorithm 1: Workflow of AGENT+P

Input: Natural language goal g_{nl}, App $\mathscr{A}$, Max running steps *maxStep*
Output: Automation outcome: *Success* or *Failure*
Aliases: ***UB*** $\leftarrow$ *UtgBuilder;* ***NS*** $\leftarrow$ *NodeSelector;* ***PG*** $\leftarrow$ *PlanGenerator;* ***UE*** $\leftarrow$ *UiExplorer*

1 $G \leftarrow$ UB.buildStaticUTG($\mathscr{A}$);
2 $u_{cur} \leftarrow$ getCurrentUI($\mathscr{A}$);
3 $steps \leftarrow 0$;
4 **while** $steps \leq maxStep$ **do**
5 $u_{target} \leftarrow$ NS.selectTargetNode(g_{nl}, G);
6 $Plan \leftarrow$ PG.generatePlan(u_{cur}, u_{target}, G);
7 **if** *Plan is valid* **then**
8 **for** *each action* a_i *in Plan* **do**
9 $u_{next} \leftarrow$ UE.act(a_i);
10 $G \leftarrow$ UB.update($G, u_{cur}, a_i, u_{next}$);
11 $u_{cur} \leftarrow u_{next}$;
12 **else**
13 $neighbors \leftarrow$ PG.getNeighbors(u_{cur}, G);
14 $a \leftarrow$ UE.decideAction($neighbors, u_{cur}$);
15 $u_{next} \leftarrow$ UE.act(a);
16 $G \leftarrow$ UB.update($G, (u_{cur}, a, u_{next})$);
17 $u_{cur} \leftarrow u_{next}$;
18 **if** *UE.evaluate*(g_{nl})) **then**
19 **return** *Success*;
20 $steps \leftarrow steps + 1$;
21 **return** *Failure*;

Embedding-based Fallback. In cases where the MLLM fails to return a confident result, AGENT+P computes semantic embeddings for both the user's input query and the textual representations of all UI nodes in the UTG. We then calculate the cosine similarity between the query embedding and each node embedding, selecting the node with the highest similarity score as the target.

4.3 Plan Generator

Once the target node u_{target} is identified, the task becomes a classical planning problem: finding the shortest sequence of actions from the current UI state to the target state. The Plan Generator orchestrates this by first converting the UTG into the

PDDL format. It then invokes a classical planner to solve for an optimal PDDL plan (Line 6).

This symbolic plan is subsequently translated back into a sequence of clear, natural language instructions for the UI Explorer to execute. An example of a raw PDDL plan and its corresponding natural language translation are shown in Appendix in Listing 6 and Table 6, respectively. In cases where the classical planner fails to find a valid path (e.g., if the target is unreachable), AGENT+P implements a fallback strategy. Instead of a plan, it generates a textual summary of the k-hop neighboring nodes from the current UI, providing contextual information to help the agent decide on its next steps (Line 12-14).

4.4 UI Explorer

The UI Explorer acts as the AGENT+P's execution engine, emulating user interactions with the app (Line 9, Line 14-15, and Line 18-19). It takes the natural language plan and executes each step programmatically. After each action, it evaluates whether the resulting UI state matches the expected goal.

A key design feature of the UI Explorer is its modularity. It is a plug-and-play component, allowing AGENT+P to be integrated with various types of UI agents, including those based on LLMs, MLLMs, or other specialized models that incorporate capabilities like reflection and grounding. This flexibility is demonstrated in our evaluation (Section 6), where we integrate AGENT+P with different UI agents to showcase its broad applicability.

5 Evaluation Setting

5.1 Datasets

We evaluate AGENT+P in two distinct UI automation scenarios: user task execution and automated UI testing.

User Task Execution. We use AndroidWorld (Rawles et al., 2024), which is a standard benchmark for evaluating how well UI agents perform real-world tasks. It provides automatic metrics to assess the success rate of an agent. For this scenario, we evaluate a subset of apps from the AndroidWorld benchmark. This subset consists of apps where baseline agents most frequently failed, as identified in our motivational study in subsection 2.2. Table 5 summarizes the characteristics of these apps, including the number of tasks and the complexity of their UTGs (nodes and edges).

Automated UI Testing. No established benchmark currently exists. Therefore, we design a custom test case that requires the agent to navigate to an app's privacy policy page. We choose this case for two primary reasons: (1) *Relevance*: Every app is mandated to include a privacy policy page, yet these pages often require security auditing for violations (Yang et al., 2022; Slavin et al., 2016), creating a strong need for automated UI testing. (2) *Complexity*: The visual implementation and location of entry points (e.g., settings icons, hamburger menus) vary significantly across apps, posing a non-trivial navigation challenge for agents attempting to locate the target UI without errors. For this scenario, we evaluate on all the apps from the AndroidWorld benchmark.

5.2 Baselines

We compare AGENT+P against two categories of state-of-the-art approaches in UI automation:

General UI Agents. We select *DroidRun*, *MobileUse*, and the *T3A* agent from the official AndroidWorld leaderboard (Android World, 2025). We chose these specific agents because other leaderboard entrants are either not open-source or their released code is non-functional. For our evaluation, we integrate each as the UI Explorer module within AGENT+P. Note that T3A is tightly coupled with the AndroidWorld infrastructure and cannot be adapted for the UI testing scenario.

UI Testing Tools. We also compare against three approaches specifically for the automated UI testing:

- *Guardian (Ran et al., 2024):* An LLM-based UI agent for automated testing with traditional planning.
- *AutoDroid (Wen et al., 2024):* An LLM-based UI agent for UI Testing that utilizes UTGs without symbolic planning.
- *GoalExplorer (Lai and Rubin, 2019):* A fully symbolic UI automation approach based on UTGs.

Since these three approaches do not support the AndroidWorld environment, we evaluate them exclusively on the automated UI testing scenario.

5.3 Metrics

Consistent with prior work (Rawles et al., 2024; Li et al., 2025b), we employ *Success Rate* (SR) as the primary metric to evaluate the effectiveness of each

Table 3: AGENT+P compared with existing UI agents and traditional UI Automation approaches. We highlight the rows where AGENT+P is integrated. SR is short for Success Rate.

Approaches	User Task Execution			Automated UI Testing		
	SR (%)	Steps	Time (s)	SR (%)	Steps	Time (s)
DroidRun	15.91	5.18	33.38	60.00	6.10	19.90
+ AGENT+P	**28.41**	17.09	99.50	**60.00**	**3.80**	**16.90**
MobileUse	9.09	20.45	256.02	40.00	7.36	22.32
+ AGENT+P	**11.36**	**19.61**	**219.28**	**50.00**	**4.00**	**18.45**
T3A	24.32	17.89	114.39		–	
+ AGENT+P	**38.63**	**13.43**	**95.90**		–	
AutoDroid		–		35.00	9.42	46.43
Guardian		–		35.00	12.31	128.37
GoalExplorer		–		15.00	3.60	8.40

UI automation approach.

Since the integration of a symbolic planner introduces planning latency, it is critical to assess whether the improved effectiveness justifies this cost. Therefore, to evaluate the overall efficiency, we report the average number of *Steps* (UI actions) and the total *Time* required to complete a task.

6 Evaluation Results

Table 3 presents our evaluation results across two UI automation scenarios.

6.1 User Task Execution

Effectiveness. As detailed in the left-hand section of Table 3, AGENT+P improves the success rate for all three baseline agents. The integration yields the most substantial gains for *DroidRun*, where the success rate nearly doubles from 15.91% to 28.41%. Similarly, *T3A* achieves a gain 14.31% absolute improvement (24.32% $\rightarrow$ 38.63%). Notably, the combination of *T3A* + AGENT+P achieves the highest overall success rate of 38.63%, surpassing all standalone baselines. These results confirm that augmenting LLM-based agents with global transition information effectively resolves navigation bottlenecks, allowing them to complete complex tasks that standalone agents fail to solve.

Efficiency. The impact of AGENT+P varies depending on the agent's baseline behavior. For *T3A* and *MobileUse*, AGENT+P reduces both the average steps and execution time, indicating that the planner helps the agent avoid "dead ends" and solve tasks more directly. Conversely, for *DroidRun*, we observe an increase in average steps (5.18 $\rightarrow$ 17.09) and total time. This inverse trend is because of the drastic increase in success rate; the standalone *DroidRun* agent tends to fail quickly on complex tasks (resulting in low step counts), whereas AGENT+P enables it to persist and successfully navigate deeper into the app to complete harder tasks, naturally incurring a higher step count.

6.2 Automated UI Testing

Effectiveness. As shown in the right-hand section of Table 3, the success rates of general UI agents surpass those of dedicated UI testing tools. Specifically, *DroidRun* + AGENT+P achieves a success rate of 60.00%, outperforming *AutoDroid* and *Guardian* by 25.00%. Furthermore, the purely symbolic approach *GoalExplorer* struggles (15.00%).

Efficiency. While the integration of AGENT+P yields varying improvements in effectiveness (boosting *MobileUse* by roughly 10% while *DroidRun* remains consistent), it drastically reduces the effort required to reach the target UI. Since the privacy policy page is a fixed target, the step count provides a direct measure of navigation optimality. *DroidRun* + AGENT+P reduces the average path length from 6.10 steps to 3.80 steps (a 37.70% reduction) and total time from 19.90s to 16.90s compared to the standalone configuration. Similarly, *MobileUse* + AGENT+P reduces the average steps from 7.36 to 4.00. These results demonstrate that while standalone agents often rely on exploration or trial-and-error to find a specific page, AGENT+P guides them via the optimal path derived from the underlying UTG, minimizing redundant interactions and accelerating the testing process.

7 Discussion

Symbolic Planning as a Remedy for LLM Hallucination. Our study demonstrates that integrating classical symbolic planning with LLM-based agents substantially mitigates long-horizon planning failures. Unlike end-to-end LLM reasoning, which often suffers from hallucination and myopic exploration, symbolic planners provide *global guarantees* on correctness and optimality. This synergy leverages the complementary strengths of both paradigms: the interpretability and reliability of symbolic reasoning, and the perception and linguistic versatility of LLMs. We believe that such hybrid architectures represent a promising direction for future UI automation and broader embodied AI research.

Generalization. AGENT+P's methodology extends beyond Android UI automation to any do-

main modeled by state–transition graphs, including web (Zhou et al., 2023) and robotics (Zhu et al., 2025a). By substituting the UTG with analogous structures (e.g., DOM), AGENT+P offers a blueprint for a unified symbolic–neural framework for general UI reasoning.

Multi-Goal Automation. Our current implementation simplifies UI automation tasks into single-goal navigation problems, where the objective is to reach a single target UI. However, many practical user tasks are inherently multifaceted and require achieving a sequence of sub-goals. For instance, a task like *"Add an item to the shopping cart and then proceed to checkout"* involves successfully reaching the item's page (goal 1) and subsequently navigating to the checkout screen (goal 2).

Extending AGENT+P to handle multi-goal scenarios would involve evolving the Node Selector into a more sophisticated *"Goal Decomposer"* capable of parsing a complex natural language instruction into an ordered list of target UI nodes $\{u_{target_1}, u_{target_2}, \ldots, u_{target_n}\}$. Subsequently, the Plan Generator would leverage the native ability of PDDL to support multiple goal predicates, enabling the generation of a single, cohesive plan that traverses the UTG to satisfy all sub-goals. This effectively extends AGENT+P to a hierarchical planning paradigm. However, this approach also raises new research questions regarding goal ordering and dependency resolution, especially when goals become infeasible at runtime.

8 Related Work

8.1 UI Automation

Traditionally, research in UI automation has centered on *automated testing*, where the primary objective is to systematically explore an app's UIs to discover bugs (Ran et al., 2024; Hu and Neamtiu, 2011; Lai and Rubin, 2019), or security vulnerabilities (Shahriar and Zulkernine, 2009; Moura et al., 2023; Liu et al., 2020). With the recent advent of LLMs, the focus has expanded to *task-driven GUI agents*, which aim to complete specific real-world tasks rather than maximizing the exploration coverage (Wen et al., 2024; Rawles et al., 2024).

8.2 LLM + Symbolic Planner

Prior work demonstrates the mutual benefits of combining LLMs with symbolic planners. One prominent direction involves using LLMs to formalize natural language problems into PDDL, allowing classical solvers to guarantee reliable execution (Liu et al., 2023; Dagan et al., 2023; Guan et al., 2023; Valmeekam et al., 2023). Conversely, other approaches use LLMs to augment symbolic planners by interpreting environment feedback (Zhu et al., 2025b,a). AGENT+P synthesizes insights from both directions: it employs a symbolic planner to guide the LLM's decision-making, while simultaneously leveraging the LLM to interpret UI feedback and refine the symbolic planning space.

8.3 UI Transition Modeling

Existing works construct UTGs (or similar concepts) using various methods across multiple platforms. One primary approach is *dynamic analysis*, where the application is executed to observe real transitions. This is seen in early, multi-platform (e.g., web, mobile) systems like GUITAR (Nguyen et al., 2014) and more recent tools such as GUI-Xplore (Sun et al., 2025) and Autodroid (Wen et al., 2024). The other approach is *static analysis*, which infers control flow from the app's code, such as A^3E's activity transition graph (Azim and Neamtiu, 2013) and Gator's window transition graph (Yang et al., 2018). AGENT+P constructs the UTG to model UI transitions by synergistically combining both static and dynamic techniques.

9 Conclusion

In this paper, we introduced AGENT+P, a novel agentic framework designed to address the critical challenge of long-horizon planning in UI automation. By modeling an app's transition structure as a UTG and leveraging an external symbolic planner, AGENT+P provides LLM-based agents with a globally optimal, high-level plan, effectively mitigating hallucination that causes common automation failure. Our evaluation on the AndroidWorld benchmark demonstrates that AGENT+P can be integrated as a plug-and-play module to substantially improve the success rates of state-of-the-art UI agents. We believe AGENT+P lays the foundation for future research into neuro-symbolic planning paradigms for creating more robust and reliable UI agents for complex UI automation tasks.

Limitations

While AGENT+P effectively bridges global planning with local reasoning, several challenges remain. First, constructing accurate UTGs still de-

pends on program analysis quality. Static analysis may include infeasible edges, while dynamic exploration may yield incomplete coverage. Second, translating between natural language and PDDL representations introduces an additional layer of abstraction; errors in this translation can propagate through the pipeline. Finally, classical planners assume deterministic transitions, yet real-world GUIs often contain stochastic behaviors (e.g., pop-ups or async UI updates), which may lead to plan divergence during execution. Addressing these issues requires tighter integration between the planner and the environment to support real-time re-planning and uncertainty handling.

Ethical Considerations

Automated UI agents that can execute complex tasks on real applications must be deployed with care. Potential misuse, such as automating sensitive or privacy-related operations without explicit consent, highlights the need for transparent auditing and permission control. In our implementation, all experiments are conducted on benign benchmark apps under controlled environments.

References

Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, and 1 others. 1998. Pddl—the planning domain definition language. *Technical Report, Tech. Rep.*

Android World. 2025. Leaderboard of AndroidWorld.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269.

Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 641–660, New York, NY, USA. Association for Computing Machinery.

Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Cărbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. 2024. Screenai: A vision-language model for ui and infographics understanding. *arXiv preprint arXiv:2402.04615*.

Gautier Dagan, Frank Keller, and Alex Lascarides. 2023. Dynamic planning with a llm. *arXiv preprint arXiv:2308.06391*.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. 2025. Advancing mobile gui agents: A verifier-driven approach to practical deployment. *arXiv preprint arXiv:2503.15937*.

Google Developers. 2025. Introduction to activities.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094.

Cuixiong Hu and Iulian Neamtiu. 2011. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83.

Duling Lai and Julia Rubin. 2019. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 115–127. IEEE.

Jiawei Li, Jiahao Liu, Jian Mao, Jun Zeng, and Zhenkai Liang. 2025a. Ui-ctx: Understanding ui behaviors with code contexts for mobile applications. In *NDSS*.

Ning Li, Xiangmou Qu, Jiamu Zhou, Jun Wang, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, and Weinan Zhang. 2025b. Mobileuse: A gui agent with hierarchical reflection for autonomous mobile operation. *arXiv preprint arXiv:2507.16853*.

Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*.

Changlin Liu, Hanlin Wang, Tianming Liu, Diandian Gu, Yun Ma, Haoyu Wang, and Xusheng Xiao. 2022. Promal: precise window transition graphs for android via synergy of program analysis and machine learning. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1755–1767.

Chenxu Liu, Zhiyu Gu, Guoquan Wu, Ying Zhang, Jun Wei, and Tao Xie. 2025. Temac: Multi-agent collaboration for automated web gui testing. *arXiv preprint arXiv:2506.00520*.

Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé Bissyandé, and Jacques Klein. 2020. MadDroid: Characterizing

and detecting devious ad contents for android apps. In *Proceedings of The Web Conference 2020*, pages 1715–1726.

Shang Ma, Chaoran Chen, Shao Yang, Shifu Hou, Toby Jia-Jun Li, Xusheng Xiao, Tao Xie, and Yanfang Ye. 2025. Careful about what app promotion ads recommend! detecting and explaining malware promotion via app promotion graph. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Thiago Santos de Moura, Everton LG Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. 2023. Cytestion: Automated gui testing for web applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 388–397.

Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering*, 21(1):65–105.

Graziella Orrù, Andrea Piarulli, Ciro Conversano, and Angelo Gemignani. 2023. Human-like problem-solving abilities in large language models using chatgpt. *Frontiers in artificial intelligence*, 6:1199350.

Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A runtime framework for llm-based ui exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 958–970.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, and 1 others. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*.

Hossain Shahriar and Mohammad Zulkernine. 2009. Automatic testing of program security vulnerabilities. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 2, pages 550–555. IEEE.

Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International conference on software engineering*, pages 25–36.

Yuchen Sun, Shanhui Zhao, Tao Yu, Hao Wen, Samith Va, Mengwei Xu, Yuanchun Li, and Chongyang Zhang. 2025. Gui-xplore: Empowering generalizable gui agents with one exploration. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19477–19486.

Antti Valmari. 1996. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023. On the planning abilities of large language models-a critical investigation. *Advances in Neural Information Processing Systems*, 36:75993–76005.

Hui Wei, Zihao Zhang, Shenghua He, Tian Xia, Shijia Pan, and Fei Liu. 2025. Plangenllms: A modern survey of llm planning capabilities. *arXiv preprint arXiv:2502.11221*.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 543–557.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and 1 others. Osatlas: A foundation action model for generalist gui agents, 2024c.

Yuquan Xie, Zaijing Li, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Dongmei Jiang, and Liqiang Nie. 2025. Mirage-1: Augmenting and updating gui agent with hierarchical multimodal skills. *arXiv preprint arXiv:2506.10387*.

Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. Iccbot: fragment-aware and context-sensitive icc resolution for android applications. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, pages 105–109.

Shao Yang, Yuehan Wang, Yuan Yao, Haoyu Wang, Yanfang Ye, and Xusheng Xiao. 2022. Describectx: context-aware description synthesis for sensitive behaviors in mobile apps. In *Proceedings of the 44th International Conference on Software Engineering*, pages 685–697.

Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for android. *Automated Software Engineering*, 25:833–873.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, and 1 others. 2025. Mobile-agent-v3: Foundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. Appagent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*.

Kangjia Zhao, Jiahui Song, Leigang Sha, Haozhan Shen, Zhi Chen, Tiancheng Zhao, Xiubo Liang, and Jianwei Yin. 2024. Gui testing arena: A unified benchmark for advancing autonomous gui testing agent. *arXiv preprint arXiv:2412.18426*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

Wang Bill Zhu, Miaosen Chai, Ishika Singh, Robin Jia, and Jesse Thomason. 2025a. Psalm-v: Automating symbolic planning in interactive visual environments with large language models. *arXiv preprint arXiv:2506.20097*.

Wang Bill Zhu, Ishika Singh, Robin Jia, and Jesse Thomason. 2025b. Language models can infer action semantics for symbolic planners from environment feedback. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8751–8773.

A Implementation and Ablation Study

A.1 Implementation

At the time of our experiments, MobileUse and DroidRun claimed 0.91 on the official AndroidWorld leaderboard. However, the exact configurations used to obtain these results (e.g., backbone LLMs, enabling reasoning or vision capabilities) were not publicly disclosed. To ensure a fair comparison, we rerun each agent using three state-of-the-art LLMs: GPT-5, Gemini-3, and Grok 4, under multiple parameter configurations. We adopt the configuration with the best performance for our evaluation.

The static UTG is constructed using ICCBot (Yan et al., 2022) and FlowDroid (Arzt et al., 2014). For the classical planner, we employ Fast Downward with an A^* search algorithm to find optimal paths. To ensure the reliability of our findings, all experiments were conducted three times, and we report the average values.

A.2 Ablation Study

Table 4: Ablation study results for AGENT+P. We evaluate the impact of removing or replacing components in the UTG Builder, Node Selector, and Plan Generator.

Component / Variant	SR (%)
UTG Builder	
w/o Dynamic Update	34.88
Node Selector	
w/o Embedding	37.21
w/o MLLM	27.91
Plan Generator	
DFS	23.26
BFS	24.65
AGENT+P (Full Method)	**38.63**

To validate the design choices within AGENT+P, we evaluate the individual contribution of its three core components: the UTG Builder, Node Selector, and Plan Generator. Table 4 summarizes the success rates of AGENT+P (integrated with T3A) compared to variants where specific features are removed or replaced.

Impact of Plan Generator. We first assess the effectiveness of our symbolic planning approach by replacing it with standard graph search algorithms: Depth-First Search (DFS) and Breadth-First Search (BFS). In this case, the output of Plan Generator is

UTG paths containing the target node. As shown in the table, replacing the symbolic planner leads to a huge performance drop, with DFS and BFS achieving only 23.26% and 24.65% SR, respectively. This degradation occurs because standard search algorithms blindly explore the large state space of Android apps, often exhausting the time budget or step limit before achieving the goal. In contrast, our symbolic planner leverages the high-level logic of the UTG to prune irrelevant paths, achieving a substantially higher SR of 38.63%.

Impact of Node Selector. We investigate the role of the Node Selector by ablating its hierarchical strategies. Removing the MLLM ("w/o MLLM") results in a sharp decline in performance to 27.91%. This underscores that the visual and semantic reasoning capabilities of the MLLM are critical for correctly identifying relevant UI elements. Removing the embedding-based filtering ("w/o Embedding") yields a smaller reduction (37.21%), indicating that while embeddings help refine the selection, the MLLM provides the primary guidance.

Impact of UTG Builder. Finally, we examine the UTG Builder by disabling the dynamic update mechanism ("w/o Dynamic Update"). This variant, which relies on a static or append-only graph without refining existing transitions, achieves an SR of 34.88%. The drop in performance confirms that dynamically updating the UTG is beneficial for maintaining an accurate UTG.

B Details on Static UTG Construction

We construct the UTG $G = (\mathscr{U}, \mathscr{T}, \varepsilon)$ via a three-step static analysis of the source code of an Axndroid app.

UI Identification ($\mathscr{U}$). First, we identify the set of all possible UIs, $\mathscr{U}$. In the context of Android applications, a UI state generally corresponds to one of three primary components:

- **Activity:** An `Activity` represents a single, focused screen with a user interface. It serves as the entry point for interacting with the user and usually occupies the entire display window.
- **Dialog:** A `Dialog` is a small window that appears in front of the current Activity. It captures user focus for critical decisions or additional input without navigating away from the underlying screen.
- **Fragment:** A `Fragment` represents a reusable portion of the user interface within an Activity. Fragments have their own lifecycle and input handling, allowing for dynamic and flexible UI designs (e.g., tabbed views or sidebars).

Listing 2: Code snippet of an activity.

```
public class CalendarActivity extends Activity {
    @Override
    protected void onCreate(Bundle
     savedInstanceState) {
        super.onCreate(savedInstanceState);
        // We identify this onCreate call as the
     creation of the activity.
        setContentView(R.layout.activity_calendar);
    }
}
```

Transition Extraction ($\mathscr{T}$). Next, we construct the set of directed edges $\mathscr{T}$. We analyze methods containing navigation-related API calls to determine target UIs.

- **Intents:** For `startActivity(Intent)`, we resolve the target component class from the `Intent` constructor or `setClass` method.
- **Fragment Transactions:** For internal transitions, we track the target fragment class used in `FragmentTransaction`.

If a source UI u_i (e.g., `CalendarActivity`) contains a reachable call that launches a target UI u_j (e.g., `EventActivity`), we add an edge $(u_i, u_j) \in \mathscr{T}$.

Edge Labeling (ε). Finally, we define the edge-labeling function ε by linking each transition (u_i, u_j) back to the specific widget interaction that triggers it. We perform a reverse reachability analysis from the navigation call (identified in Step 2) to the enclosing event listener (e.g., `onClick`).

This allows us to characterize the action $a = (w, e)$ where w is the widget bound to the listener and e is the event type. As shown in Listing 3, the button `btnAddEvent` is identified as the widget w responsible for the transition to `EventActivity`.

Listing 3: Linking a transition to an action $a = (w, \texttt{click})$.

```
Button btnAddEvent = findViewById(R.id.btn);
// The event listener defines the action type $e$ (
     click)
btnAddEvent.setOnClickListener(new View.
     OnClickListener() {
    @Override
    public void onClick(View v) {
        // We link this navigation to widget $w_{
     btnAddEvent}$
        // Creates edge $(u_{calendar}, u_{event})$
     labeled by $a$
        Intent intent = new Intent(CalendarActivity.
     this, EventActivity.class);
        startActivity(intent);
    }
});
```

C Additional Figures and Tables

Table 5: Statistics of apps used in evaluation.

App	Tasks	Nodes	Edges
VLC	3	85	190
Simple Calendar Pro	17	24	25
Tasks	6	86	127
Markor	14	14	17
OsmAnd	3	152	508

Listing 4: Example problem PDDL for the Simple Calendar Pro app with the task *Change the time zone*.

```
(define (problem change-time-zone)
  (:domain utg-automation)

  (:objects
    SplashActivity - node
    MainActivity - node
    EventActivity - node
    SettingsActivity - node
    AboutActivity - node
    TaskActivity - node
    SelectTimeZoneActivity - node
    ManageEventTypesActivity - node
    WidgetListConfigureActivity - node
    ContributorsActivity - node
    FAQActivity - node
    LicenseActivity - node
  )

  (:init
    (at SplashActivity)

    (goal-node SelectTimeZoneActivity)

    (connected SplashActivity MainActivity)
    (connected MainActivity EventActivity)
    (connected MainActivity SettingsActivity)
    (connected MainActivity AboutActivity)
    (connected MainActivity TaskActivity)
    (connected EventActivity EventActivity)
    (connected EventActivity SelectTimeZoneActivity)
    (connected SettingsActivity
     ManageEventTypesActivity)
    (connected SettingsActivity
     WidgetListConfigureActivity)
    (connected AboutActivity ContributorsActivity)
    (connected AboutActivity FAQActivity)
    (connected AboutActivity LicenseActivity)
    (connected TaskActivity TaskActivity)
  )

  (:goal
    (goal-achieved SelectTimeZoneActivity)
  )
)
```

Listing 5: Prompt template of Node Selector.

```
Given the user goal: "{user_goal}".

Select the most relevant UTG nodes to achieve this
     goal.

Available UTG nodes:
{chr(10).join(nodes_information)}

Return your response in JSON format:
{{
    "nodes": ["node1", "node2", ...],
    "confidence": 0.8,
    "reasoning": "brief explanation"
}}

For example,
Available UTG nodes are:


Return ONLY the JSON, no other text.
```

Listing 6: The plan generated by Fast Downward A^* for the Simple Calendar Pro app with the task *Add a new event type named "1-on-1 meeting"*.

```
(navigate  SplashActivity MainActivity)
(navigate MainActivity SettingsActivity)
(navigate SettingsActivity ManageEventTypesActivity)
; cost = 3 (unit cost)
```

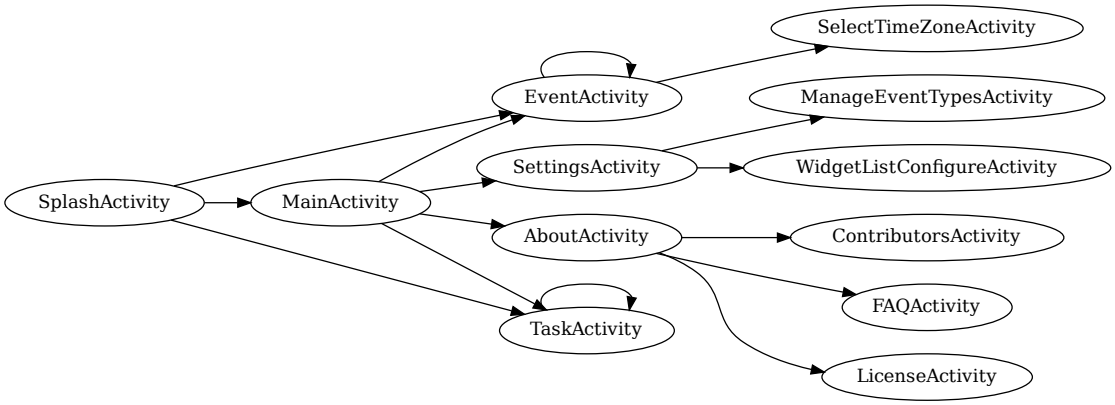

Figure 2: Graphviz visualization of the UTG of Simple Calendar Pro in AndroidWorld. An activity is an unit of Android UI (Google Developers, 2025). Edge labels are removed for visual clarity.

Table 6: Example natural language prompt for the Simple Calendar Pro app with the task *Add a new event type named '1-on-1 meeting'*.

Category	Content
Natural Language Instructions	— UTG Navigation Guide — Current UI: MainActivity — NAVIGATION PLAN FOR YOUR GOAL — Goal Analysis: Based on your goal, the system identified these target destinations: • ManageEventTypesActivity Confidence: 95% OPTIMAL PATH (Follow these steps in order): Step 1: Navigate from MainActivity to SettingsActivity Step 2: Navigate from SettingsActivity to ManageEventTypesActivity IMMEDIATE NEXT ACTION: → on click the ImageView widget with content-description "more options" via API call "virtualinvoke r0.<android.content.Context: void startActivity(android.content.Intent)>(r1)' ()" This will take you to: SettingsActivity Total steps in optimal path: 2
Usage Tips	This path was computed using PDDL planning for guaranteed optimality — USAGE TIPS — • If a NAVIGATION PLAN is shown above, follow it step by step for optimal path • The plan was computed using formal PDDL planning algorithms • If no plan exists, the target is unreachable from current location • Some UI elements may not be visible - scroll if needed
Fallback	— ALL AVAILABLE NAVIGATION OPTIONS FROM HERE — • TO REACH: EventActivity → on click ImageButton widget with context-description "New Event" via API call "virtualinvoke r0.<android.content.Context: void startActivity(android.content.Intent)>(r1)' ()" • TO REACH: SettingsActivity → on click Button widget with content-description "Settings" via API call "virtualinvoke r0.<android.content.Context: void startActivity(android.content.Intent)>(r1)' ()" • TO REACH: TaskActivity → on click ImageButton widget via api call "'virtualinvoke r2.<android.content.Context: void startActivity(android.content.Intent)>(r3)' ()" — End Navigation Guide —