GRACE: A Simulator for Continuous Goal Recognition over Changing Environments

Zihang Su[1], Artem Polyvyanyy[1], Nir Lipovetzky[1], Sebastian Sardina[2] and Nick van Beest[3]

[1]*The University of Melbourne*
[2]*RMIT University*
[3]*Data61 | CSIRO*

Abstract

Goal Recognition (GR) is a research problem that studies ways to infer the goal of an intelligent agent based on its observed behavior and knowledge of the environment. A common assumption of GR is that the underlying environment is stationary. However, in many real-world scenarios, it is necessary to recognize agents' goals over extended periods. Therefore, it is reasonable to assume that the environment will change throughout a series of goal recognition tasks. This paper introduces the problem of *continuous* GR over a changing environment. The solution to this problem is a GR system capable of recognizing agents' goals over an extended period where the environment in which the agents operate changes. To support the evaluation of candidate solutions to this new GR problem, in this paper, we present the Goal Recognition Amidst Changing Environments (GRACE) tool for generating instances of the new problem. Specifically, the tool can be configured to generate GR problems that account for different environmental changes and drifts. GRACE can generate a series of modified environments over discrete time steps and the data induced by agents operating in the environment while completing different goals.

Keywords

Goal recognition, Changing environment, Environment drift, GRACE

1. Introduction

Goal recognition (GR) techniques aim to develop a GR system capable of detecting the goal/intention of an agent embedded in an environment based on the observed behavior of the agent. The existing GR techniques, such as the plan library-based approaches [1, 2, 3], the planning-based approaches [4, 5], and the process mining (PM-)based approach [6], focus on developing algorithms to solve a "single-shot" GR problem (that is, to correctly infer the most likely goal the agent is trying to achieve now) and assume that the underlying environment is *stationary*. However, in many real-world scenarios, a GR system is expected to continuously tackle multiple GR tasks over an extended period where the environment in which the agents operate changes. When the environment changes, the agents' behaviors for achieving the goals also change. This phenomenon is observed in business processes when the behavior of process participants

changes to address new regulations, compliance rules, and innovative ways of doing business [7]. We refer to this GR problem as the problem of *continuous GR over a changing environment.*

The research on the problem of continuous GR is impeded by the lack of benchmarks for evaluating candidate solutions. Thus, in this paper, we present the Goal Recognition Amidst Changing Environments (GRACE) tool for generating changes in environments where agents work towards various goals. As seed environments, that is, those environments over which changes are applied, we use the static domains widely used in classical GR.[1]

We identify a change in an environment based on two aspects. Firstly, we characterize a change in the environment in which agents fulfill goals based on changes in the components that define the corresponding GR problem. For example, such components can be the initial and goal states of the agents and actions the agents can perform in the environment. Secondly, by interpreting an environment as a *signal*, and, consequently, a change in the original environment as a change in the original signal, we characterize changes in the environment based on the different types of concept drift [7]. For example, a change in the environment can be sudden or can be such that it progresses gradually over time.

Concretely, this paper makes the following two contributions:

- It motivates and defines the problem of continuous GR over a changing environment; and
- It presents a tool, called GRACE, that can imitate changes in the environment in which agents strive to achieve goals and, thus, can serve as a *simulator* for GR systems for solving problems of continuous GR over a changing environment.

In the next section, we define the problem of continuous GR over a changing environment and discuss different types of changes in the environment our tool aims to support. Section 3 presents the tool. Section 4 examines related work before conclusions are given in Section 5.

2. Continuous Goal Recognition

This section starts with an example that motivates the problem of continuous GR over a changing environment (Section 2.1). Then, we define this problem (Section 2.2) and discuss different types of changes (Section 2.3) and drifts (Section 2.4) that may happen in an environment over time and, thus, must be accounted for when solving the continuous version of the GR problem.

2.1. Motivating Example

Figure 1 presents a synthetic example of an environment change from environment A to environment B, both given as 11×11 grids. In environment A, there are four keys (objects) located in two rooms and four locked cells (see the cells with the gray background). The locked cells can be unlocked using the corresponding keys; for example, locked cell 1 can be unlocked using *Key1*. A robot (an agent) starts from the initial cell I at cell $(0,0)$ and can move to any adjacent cell if there is no wall between those two cells; walls are shown as solid black lines in the figure. Once the robot reaches a cell with a key, it can pick up the key in that cell. It can subsequently carry the key. If the robot that carries a key reaches the corresponding locked cell, it can unlock it. In the environment, there are two goal states, $G1$ and $G2$, surrounded by walls or locked cells. Once a cell is unlocked, the robot can pass it to reach a goal state.
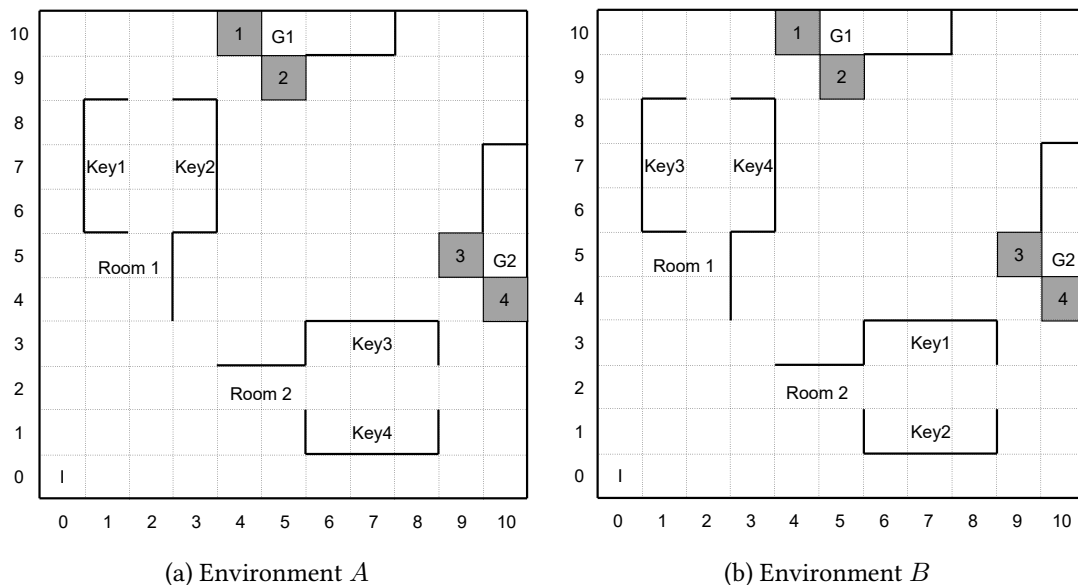
---

[1]https://github.com/pucrs-automated-planning/goal-plan-recognition-dataset

(a) Environment A (b) Environment B

Figure 1: An example of a change in the environment.

In environment A, *Key1* and *Key2* are located in Room 1, while *Key3* and *Key4* are located in Room 2. Later, environment A changes to environment B, in which *Key1* and *Key2* from Room 1 are swapped with *Key3* and *Key4* from Room 2.

We simulated a sequence of 20 GR problems. Each problem is composed of historical behaviors of agents in the environment and observations for which a goal must be recognized. These behaviors are the only available information about the environments where the agents operate; complete descriptions of the environment are not available. The first ten problems in the sequence were obtained based on the behaviors of agents in environment A. Then, the environment undergoes a sudden change, and the subsequent ten problems are based on the behaviors of agents in environment B. This sequence of 20 GR problems is publicly available.[2]

We tackled the GR problems in the sequence using the GR system grounded in process mining techniques [6]. This GR system does not require the description of the environment and learns it based on the historical observations of agents' behaviors. Based on the assumption that the environment is static, we learned the environment once, based on the observations of agents in environment A. Hence, the GR system learned that to achieve goal $G1$ (goal $G2$), the agent must visit Room 1 (Room 2). Based on this knowledge, the system can achieve high accuracy when solving the first 10 GR problems in the sequence. However, it performs poorly on the remaining 10 GR problems. Indeed, the learned knowledge is outdated after the change to environment B and needs to be updated to match the status quo. This example suggests that it is reasonable to incorporate mechanisms that allow the GR system to adapt to environmental changes, especially when operating over prolonged periods.

---

[2]A sequence of 20 GR problems: https://github.com/zihangs/GRACE/tree/main/motivating_example.

2.2. Definition

The state-of-the-art GR approaches focus on solving a GR problem in a stationary environment. This version of the GR problem usually takes three inputs: (1) a sequence of actions, also called an *observation*, denoted by O, that has been executed by an autonomous agent and observed by the GR system; (2) *knowledge* about the domain, or environment, in which the agent operates, denoted by K; and (3) a set of possible *goals* that an agent may pursue in the environment, namely goal candidates, denoted by $\mathcal{G}$. A solution to the problem often takes the shape of a probability distribution over the goals, that is, $P(G \in \mathcal{G})$. Therefore, an approach for solving the classical GR problem defines a belief function $b : O \times K \times \mathcal{G} \to P(G \in \mathcal{G})$ that maps observations, knowledge about the environment, and goal candidates onto probability distributions over the goals. The concrete distribution then expresses a belief of the observer, the GR system, about the true goal being pursued by the agent.

Domain knowledge K is different for different GR approaches. For example, for a plan library-based GR approaches [1, 2, 3], domain knowledge K is a handcrafted plan library that for each candidate goal $G \in \mathcal{G}$ contains a collection of representative plans for achieving G. In turn, for planning-based GR approaches [4, 5], domain knowledge K amounts to a planning model of the domain. Using the first-order representation [8], $K = \langle D, I \rangle$, where D is a first-order planning domain composed of a set of action schemas and a set of predicate symbols, and I is a set of information instances. Each information instance $I_i \in I$ is a tuple $\langle Obj_i, Init_i, G_i \rangle$, where Obj_i is a set of objects, $Init_i$ is the initial state, and $G_i \in \mathcal{G}$ is one of the goal candidates. Finally, for the PM-based GR approach [6], K is a set of historical plans P, captured as traces of an event log, such that each plan in P is labeled with the goal $G \in \mathcal{G}$ it leads to.

It is reasonable to assume that only partial information about the environment is available to a GR system. As K_i and $\mathcal{G}_i$ contribute to the explanation of the environment, we accept them as all and only knowledge about the environment available to a GR system, denoted by E_i, and rewrite the belief function for the i-th GR problem in the series as $b_i : O_i \times E_i \to P_i(G \in \mathcal{G}_i)$, $i \in [1 .. m]$. In the setting of the continuous GR problem, any two environments E_i and E_j, $1 \le i, j \le m$, may be different, i.e., $E_i \ne E_j$. A naïve solution to a problem of continuous GR in a changing environment can be achieved by solving each GR problem in the series independently, for example, by applying some state-of-the-art GR system. However, a GR system that operates over an extended period and solves a series of GR problems can benefit from *at least* two factors. Firstly, when solving the next GR problem in the series, a GR system can reuse all the experiences and results obtained from solving all the preceding GR problems. Secondly, a GR system with access to historical trends and experiences can project them into the future and then use these forecasts to tune its subsequent goal recognition practices.

2.3. Changes

In this work, we rely on the first-order representation of environments [8]. Such a representation consists of predicate symbols P, action schemas, objects Obj, goal candidates, and an initial state. Thus, a change in an environment can stem from a change in any of these components:

1. **Initial state:** The initial state is defined by a set of ground predicates. A ground predicate $p_i(c_1, \ldots, c_k)$ consists of predicate symbol $p_i \in P$ and ground values $c_1, \ldots, c_k \in Obj$.

To implement a change of the initial state, one can perform a random sequence of actions (a walk) from the original initial state to some other state and then accept that other state as a new initial state.

2. **Ground actions:** A ground action is obtained by assigning concrete values to the parameters of an action schema. A ground action defines the preconditions for the action occurrence and the effects it produces. A synthetic environment (an instance of a synthetic domain) contains a finite number of ground actions, and one can modify the environment, for example, by removing an arbitrary number of ground actions.

3. **Goal candidates**: A goal state, similar to the initial state, is given by a set of ground predicates. A GR problem usually specifies multiple goal states. Again, a goal state can be changed by executing some actions from that goal state and then accepting the resulting state as a new goal state.

4. **Objects**: Objects are values that ground the predicates. They are used to identify states. One can augment objects, for instance, by removing them, which requires a subsequent removal of the ground predicates defined over the removed objects. Note that such removal of predicates can augment the initial state and/or some goal states.

Once a new environment is obtained, it can be changed again. By applying changes iteratively, one can generate a series of modified environments. Note that a modified environment can be constructed by applying any subset of the above four changes simultaneously.

2.4. Drifts

Environment changes can progress differently over time. For example, one can experience abrupt changes that get into effect over a short period or observe small changes accumulating over time. Different patterns of changes in the environment can be classified according to five types of *concept drifts* [7], discussed below.

1. **Sudden drift**, see Figure 2a, concerns the situation when the original environment ($env0$) changes to the new environment ($env1$) over a short period. After the change is implemented, the original environment is never observed again. In the figure, ten environments are shown($E_1, \ldots, E_{10}$), where environments $E_1, \ldots, E_5$ are equal to $env0$ while environments $E_5, \ldots, E_{10}$ are equal to $env1$.

2. **Gradual drift**, see Figure 2b, concerns the situation where observations of the original environment ($env0$) alternate with increasingly frequent observations of the new environment ($env1$) until $env0$ is no longer observed. The series of environments in the figure consists of sixteen environments ($E_1, \ldots, E_{16}$). Environments $E_1, \ldots, E_4$ are equal to $env0$, environments $E_{12}, \ldots, E_{16}$ are equal to $env1$, and the gradual drift from $env0$ to $env1$ happens from E_4 to E_{12}.

3. **Incremental drift** concerns the situation when the environment "moves" from $env0$ to $env1$ through a number of subsequent intermediate environments. Figure 2c shows a series of environments $E_1, \ldots, E_{12}$ that make up an incremental drift. Environments E_1 to E_4 are equal to $env0$, then environment $env0$ moves to $env1$ through four intermediate environments E_5, E_6, E_7, and E_8; not that environments E_8 to E_{12} are all equal to $env1$.

4. **Reoccurring drift**, see Figure 2d, concerns the situation when two distinct environments $env0$ and $env1$ repeatedly alternate, where each environment is observed for some period.
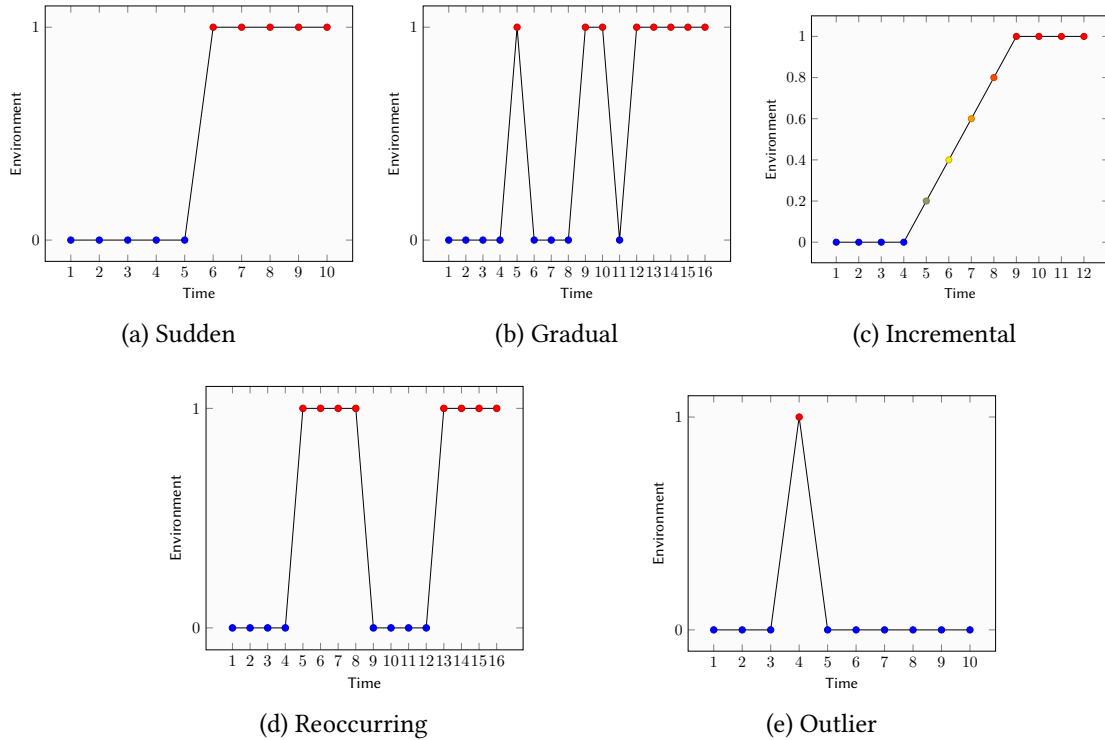
Figure 2: Five types of concept drift.

Note that reoccurring drift is different from gradual drift as there is no incremental transition from *env0* and *env1* and the transitions between the environments happen abruptly. For example, in the figure, environments E_1 to E_4 and E_9 to E_{12} are equal to *env0*, while environments E_5 to E_8 and E_{13} to E_{16} are equal to *env1*.

5. **Outlier drift**, see Figure 2e, concerns the situation when environment *env0* is observed for most of the time with an instantaneous and isolated occurrence of environment *env1* at some point in time. In the figure, only environment E_4 is equal to *env1*, and environment *env0* is observed for the rest of the time.

3. GRACE

This section presents GRACE, a tool for generating input to a GR system that aims to solve a continuous GR problem. GRACE is implemented in Python and is publicly available.[3] Section 3.1 and Section 3.2 present the architecture and the interface of GRACE, respectively.

3.1. Architecture

The architecture of GRACE is shown in Figure 3. It consists of three components: Environment Modifier, Planner, and Drift Generator. As input, GRACE takes a description of a GR problem in Planning Domain Definition Language (PDDL) [9], which specifies the environment, including
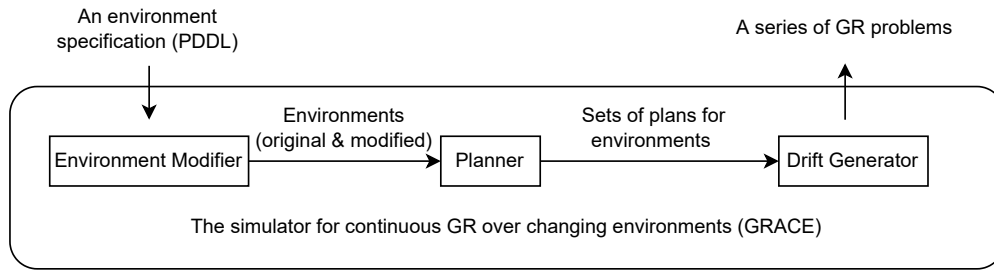
---

[3]https://github.com/zihangs/GRACE

Figure 3: The architecture of GRACE.

the initial state, objects, goal candidates, and action schemes available in the environment. The output is a series of GR problems, each comprising a PDDL description of the environment of the problem and an observation of an agent's behavior in the environment. The environments of the GR problems in the generated series are modified according to the requested types of changes and drifts, as described in Section 2.3 and Section 2.4, respectively.

The Environment Modifier component of GRACE transforms the description of an input environment according to the requested types of environmental changes described in Section 2.3, producing a set of modified environments. The number of modified environments depends on the requested type of concept drift discussed in Section 2.4. For example, incremental drift entails the intermediate stages of the environmental change, resulting in multiple modified environments generated by GRACE. Conversely, the other drift types are defined over two environments and, thus, GRACE generates one modified environment for them. Next, the Planner component takes the environments (original and modified) and generates multiple sets of plans induced by agents that operate in them. Finally, the Drift Generator component constructs the requested drift by selecting and arranging the environments and the generated plans on the timeline; the latter are used as observations of how agents achieve their goals in the environments. GRACE returns all the generated artifacts to support the evaluation of various GR systems with different assumptions and, thus, data requirements.

3.2. Interface

Table 1 gives an overview of the parameters for configuring each component of GRACE. *Mandatory* parameters must be configured for every call to GRACE, while *optional* parameters can be omitted. Optional parameters can be *mutually exclusive*, that is, never specified together, and *associated*, that is, specific to some selected option. The `Input` parameter is mandatory for all three components. Next, we discuss the parameters of each component.

3.2.1. Environment Modifier

GRACE implements all four methods for automatically changing environments discussed in Section 2.3. These methods can be invoked using options `-InitRW`, `-ObjectRM`, `-ActionRM`, and `-GoalRW` of Environment Modifier. Option `-InitRW` is used to modify the initial state by executing a number of possible random actions and accepting the resulting state as the new initial state in the modified environment. The number of random actions is specified using the associated parameter "*Steps of random walk from start*". The `-ObjectRM` option

Table 1
Parameters of GRACE components.

Environment Modifier		Planner		Drift Generator	
Input (an original environment)		Input (all environments)		Input (sets of plans)	
-InitRW	Steps of random walk from start	-Topk	-	-Sudden	Cases per env
-ObjectRM	Delete count	-Diverse	-	-Gradual	Stable period / Changing period
-ActionRM	Delete count			-Incremental	Stable period / Intermediate stages
-GoalRW	Steps of random walk from goal	Number of plans		-Reoccurring	Stable period / Occurrence count
Number of environments		Time limit		-Outlier	Cases / Probability

is used to modify the environment by randomly deleting objects from the initial state. The associated parameter "*Delete count*" specifies the number of objects to be deleted. When an object is deleted from the environment, all predicates in the initial state containing the value of that object are deleted as well. The -ActionRM option is used to modify the environment by randomly deleting ground actions. The associated parameter "*Delete count*" specifies the number of ground actions to be deleted from the environment. Finally, the -GoalRW option is introduced to support modifications that augment the states that define the goal candidates. To implement this change, we apply the backward search method [10] to randomly regress an arbitrary number of steps from each goal state. The associated parameter "*Steps of random walk from goal*" specifies the number of random backward actions to be executed from each goal state, resulting in the modified goal states. Finally, the mandatory parameter "*Number of environments*" must be configured for each call to Environment Modifier. This parameter specifies the number of modified environments to generate.

An example call to Environment Modifier (env_modifier.py) is shown below.

```
python env_modifier.py original_env -ObjectRM 5 1
```

We use original_env as input, which is a directory that stores files specifying the domain model of the original environment.[4] The call uses option -ObjectRM to remove five objects from the environment (see parameter 5) and requests to construct one modified environment (see parameter 1). The instructions on how to use Environment Modifier are available in the documentation of GRACE.[5]

The original environment is a ten-by-ten grid. In this environment, the solid black lines between cells are walls that agents cannot pass. The cells in gray are locked, where the shape number in the locked cells indicates the number of the key that the agent can use to unlock this cell. For example, the cell with shape 21 can be unlocked with $Key21$. The initial state where an agent starts is cell $(0, 0)$, and there are ten goal states the agent might aim to achieve; see $G1$ to $G10$ in the figure. Figure 4b shows the modified environment produced after executing the command. In the modified environment, the five cells in black are deleted permanently, which
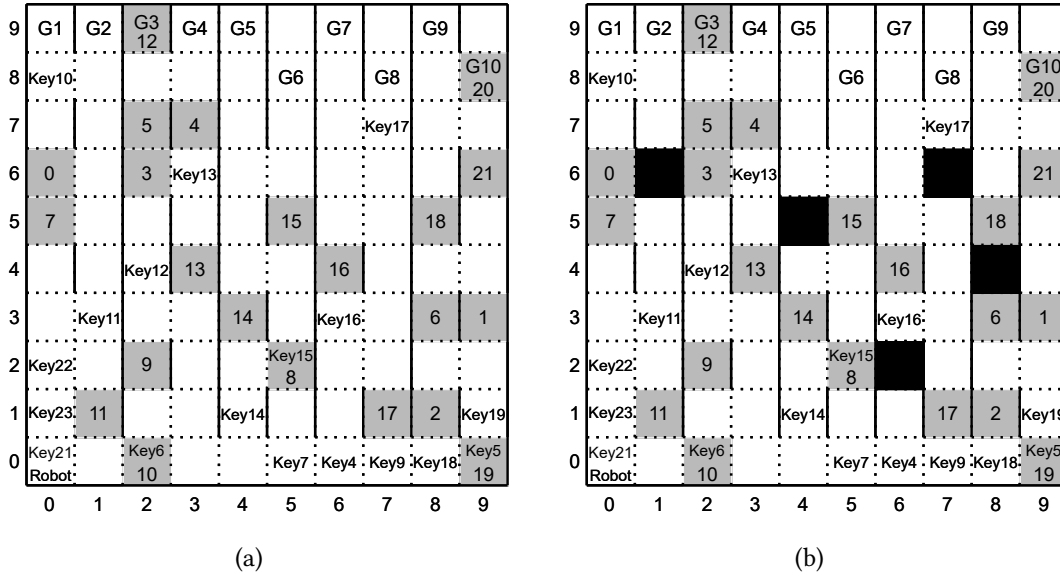

Figure 4: An example of modifying an original environment (a) to obtain a modified environment (b).

means the agent can never access or pass these cells. Note that other objects, for instance, keys, can be removed as well.

3.2.2. Planner

The Planner component of GRACE has two options: the Top-K planner [11] (option *-Topk*) and the diverse planner [12] (option *-Diverse*). The Top-K planner generates cost-optimal plans (plans with minimal costs of actions), while the diverse planner generates divergent plans (the plans may have higher costs than the cost-optimal plans). Both planners have the mandatory parameter "*Number of plans*" that specifies the number of plans to generate for each goal candidate. Additionally, the mandatory "*Time limit*" parameter instructs the planner to stop once it reaches the specified time limit, returning all found plans up to that point.

An example call to Planner (`planner.py`) is proposed below.

```
python planner.py all_envs -Topk 50 20
```

The `all_envs` input describes the original and the modified environments shown in Figure 4. We use the Top-K planner, see option `-Topk`, to generate 50 plans for each goal in the environments with the time limit set to 20 seconds (per goal); see the last two parameters in the call. For more details on using the Planner component, please refer to the documentation.[6]

A call to Planner returns two sets of plans, one containing the plans generated in the original environment and the other with plans in the modified environment. Since both environments have ten goal candidates, refer to the figure, the generated plans, for each of the two environments, are grouped into ten subsets, each containing the plans for the corresponding goal.

---

[6]https://github.com/zihangs/GRACE/tree/main/planner

3.2.3. Drift Generator

The Drift Generator component takes a plan pool generated by the Planner component as input, selects plans from the plan pool, and queues the selected plans to simulate one of the concept drifts discussed in Section 2.4. One can simulate a sudden drift by selecting option -Sudden. This option requires the associated parameter "*Cases per env*", which specifies the number n of observations to use from each environment in the resulting drift. Specifically, after n observations from the original environment, n observations from the modified environment follow. To simulate a gradual drift, option -Gradual should be used. Parameter "*Stable period*" specifies the requested number of observations during each stable period for which the environment does not change, while parameter "*Changing period*" specifies the requested number of observations during the changing period. Option -Incremental is used to simulate an incremental drift. This option also uses the "*Stable period*" to specify the periods of no changes for the original and the resulting environments, while parameter "*Intermediate stages*" specifies the number of intermediate transient environments to generate between the two stable environments. Option -Reoccurring simulates a reoccurring drift. Two associated parameters are "*Stable period*", which, as above, specifies the period of stability for each environment, and "*Occurrence count*", which defines the number of times the change between the environments happens. Finally, option -Outlier simulates an outlier drift, where parameter "*Cases*" specifies the total number of observed plans during the entire period and parameter *Probability* specifies the probability for an outlier to occur at each timestamp. For all the discussed options, each of the observations included in the simulated period aims to reach one random candidate goal.

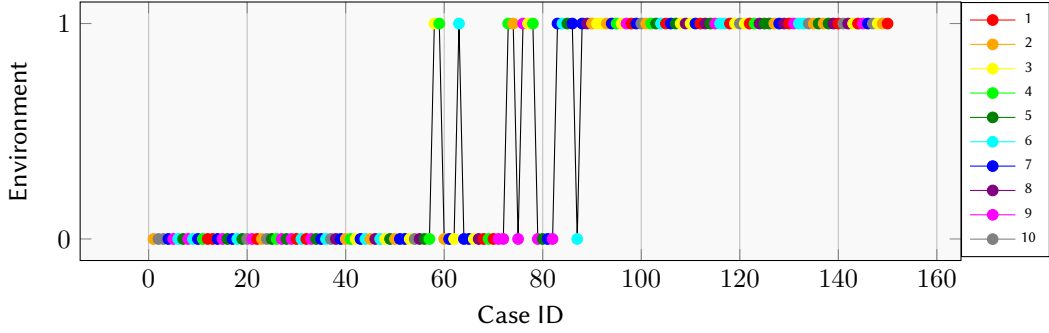An example call to Drift Generator (`drift_generator.py`) is proposed below.

```
python drift_generator.py plan_pool -Gradual 50 50
```

Input `plan_pool` contains sets of plans generated by the Planner component. In this call, Drift Generator is requested to generate a gradual drift, see option -Gradual, while the associated parameters "*Stable period*" and "*Changing period*" are both set to 50. The execution of the command resulted in a series of observations shown in Figure 5. Note that there are ten goal candidates in each environment, and each plan achieves one of the goal candidates indicated by different colors in the figure; for example, in Figure 5b, the observed plan of GR problem 51 is to achieve goal 7 in environment 0. For more details on the use of the component, see the documentation.[7]

4. Related Work

The problem of continuous GR over a changing environment is an extension of the classical GR problem in which the environment is accepted to be stationary. Similar to the problem of concept drift detection in process mining [7, 13, 14], the new GR problem aims to detect changes in the environment by analyzing the changes in the observed data, like traces of agents captured as sequences of their actions. However, in addition to detecting a change, the new problem requests adapting the GR system to the new environment, which is similar to what is requested in the problems of model reconciliation and maintenance [15, 16].

---

[7]https://github.com/zihangs/GRACE/tree/main/drift_generator

(a) A gradual drift over 150 observations with the change occurring between time steps 51 and 100.
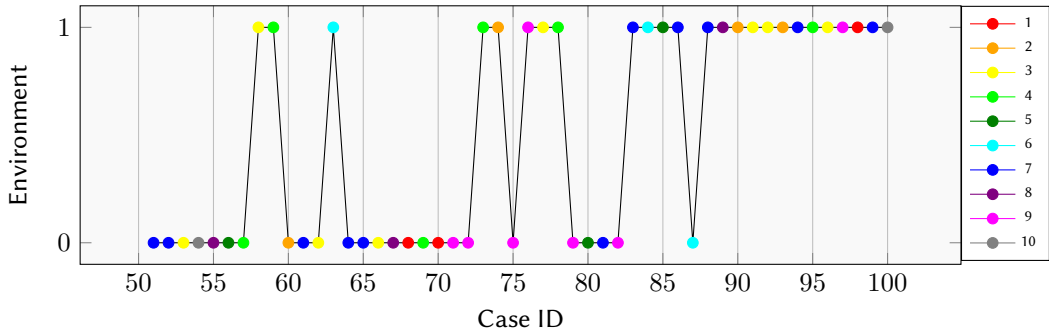
(b) A time window from step 51 to step 100 showing the gradual change period.

Figure 5: An example gradual drift generated by GRACE.

The GRACE tool presented in this paper works with environments captured in Planning Domain Definition Language (PDDL) [9]. Given a classical GR problem as input, it generates instances of the problem formulated for different types of drifts in the original environment, hence serving as a simulator for GR systems to exercise their GR capabilities. GRACE uses the Tarski tool for parsing and manipulating synthetic domains [17]. In addition, the random walk method in the environment modifier relies on the breadth-first search tool provided by the Tarski framework. Another tool used for validating the modified environment is LAPKT [18], an automated planning tool. When randomly removing objects in the environment, one can remove essential objects required to reach the goal state. To this end, LAPKT can check if a plan exists from the initial state to the goal state. After obtaining a valid modified environment, we use the Top-K planner [11] and the diverse planner [12] to generate multiple plans to be used by the drift generator. The Top-K planner generates cost-optimal plans, while the diverse planner generates divergent plans.

5. Conclusion

This paper introduces the problem of continuous goal recognition (GR) over a changing environment, which requests a GR system to continuously tackle GR problems over a period during which the underlying environment may change. We define this new problem as an extension of the classical GR problem for stationary environments. To generate the experimental data

that can be used to test solutions to the new problem, we present a tool, called GRACE, which can automatically modify an environment given as a planning domain, to obtain a series of GR problems defined over the environments modified according to a wide range of types of changes. The paper explains the architecture and interface of GRACE. Hence, the results reported in this paper enable future work on developing and evaluating GR techniques for solving GR problems in changing environments.

References

[1] H. A. Kautz, J. F. Allen, Generalized plan recognition, in: AAAI, Morgan Kaufmann, 1986.

[2] E. Charniak, R. P. Goldman, A bayesian model of plan recognition, Artif. Intell. 64 (1993).

[3] G. Sukthankar, K. P. Sycara, A cost minimization approach to human behavior recognition, in: AAMAS, ACM, 2005.

[4] M. Ramírez, H. Geffner, Probabilistic plan recognition using off-the-shelf classical planners, in: AAAI, AAAI Press, 2010.

[5] R. F. Pereira, N. Oren, F. Meneguzzi, Landmark-based approaches for goal recognition as planning, Artif. Intell. 279 (2020).

[6] A. Polyvyanyy, Z. Su, N. Lipovetzky, S. Sardiña, Goal recognition using off-the-shelf process mining techniques, in: AAMAS, IFAAMAS, 2020.

[7] A. Yeshchenko, C. D. Ciccio, J. Mendling, A. Polyvyanyy, Visual drift detection for event sequence data of business processes, IEEE Trans. Vis. Comput. Graph. 28 (2022).

[8] B. Bonet, H. Geffner, Learning first-order symbolic representations for planning from the structure of the state space, in: ECAI, volume 325 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2020.

[9] P. Haslum, N. Lipovetzky, D. Magazzeni, C. Muise, An Introduction to the Planning Domain Definition Language, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2019.

[10] C. Lei, N. Lipovetzky, Width-based backward search, in: ICAPS, AAAI Press, 2021.

[11] D. Speck, R. Mattmüller, B. Nebel, Symbolic top-k planning, in: AAAI, AAAI Press, 2020.

[12] M. Katz, S. Sohrabi, Reshaping diverse planning, in: AAAI, AAAI Press, 2020.

[13] V. Denisov, E. Belkina, D. Fahland, BPIC'2018: Mining concept drift in performance spectra of processes, in: 8th International Business Process Intelligence Challenge, 2018.

[14] B. Hompes, J. C. A. M. Buijs, W. M. P. van der Aalst, P. M. Dixit, H. Buurman, Detecting change in processes using comparative trace clustering, in: SIMPDA, volume 1527 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015.

[15] T. Chakraborti, S. Sreedharan, Y. Zhang, S. Kambhampati, Plan explanations as model reconciliation: Moving beyond explanation as soliloquy, in: IJCAI, ijcai.org, 2017.

[16] D. Bryce, J. Benton, M. W. Boldt, Maintaining evolving domain models, in: IJCAI, IJCAI/AAAI Press, 2016.

[17] G. Francés, M. Ramirez, Collaborators, Tarski: An AI planning modeling framework, https://github.com/aig-upf/tarski, 2018.

[18] M. Ramirez, N. Lipovetzky, C. Muise, Lightweight Automated Planning ToolKiT, http://lapkt.org/, 2015.