Learning and Recognizing Human Behaviour with Relational Decision Trees

Anonymous submission

Abstract

The recognition of activities performed by humans is crucial in human-robot interaction. However, assuming humans always follow rational behaviour in executing activities may not be accurate since individual preferences influence their decision-making. This paper proposes a method for learning human behaviour that involves capturing how humans select actions to solve problems. This behaviour is represented by a Relational Decision Tree. We define two sets of features that can be automatically extracted from the planning domain. A behaviour library is created and used to identify the behaviour followed by a person when executing a plan in a new situation. This approach allows to anticipate the person's needs and act accordingly. The method was tested in three different domains, showing its validity.

Introduction

Behaviour recognition is commonly defined as the process of identifying and categorizing activities based on observations captured by video or sensor-based systems. Most of the research in this field is oriented towards recognizing the activity a person is doing, which has become increasingly popular in smart homes (Chua, Marsland, and Guesgen 2011; Guesgen and Marsland 2016; Degardin and Proença 2021). As a result, research focuses on using image processing and machine learning techniques to convert low-level information into understandable activities or actions (Bouchabou et al. 2021).

In this paper, human behaviour is interpreted as a set of habits that define a person's pattern of action in a given situation, which is influenced by their personal preferences. The problem input is a set of sequences of activities (plans) performed by a person as well as the specification of the domain in which the activity is taking place, and the objective is to discover patterns of relationships between the actions of the plans. For example, a person delivering packages in a three-story building may start at the third floor and go down, or go from the ground floor to the top; the delivery person may use the elevator or climb the stairs, visit the offices of a story in a particular order, etc. Ultimately, the goal is to uncover the structure of an individual's behaviour from their actions' demonstrations, with the aim of anticipating their needs and taking appropriate measures.

In the context of Automated Planning, the task of behaviour recognition has been formulated as a *goal/plan recognition* problem and addressed from different perspectives (Ramírez and Geffner 2009; Mirsky, Keren, and Geib 2021). While plan recognition is the task of identifying **what** the agent is doing from observations of its acting, we focus instead on **how** the agent is acting, that is, on the behaviour or strategy the actor displays when solving a task. Imagine, for instance, a robot navigating a grid for reaching a particular position. The robot may follow the least costly (optimal) path to reach the goal. This is the common assumption in plan recognition, that the observed agent is a perfectly rational agent that behaves optimally (Meneguzzi and Pereira 2021). The robot, however, may be *non-fully rational* (Masters and Sardiña 2019), and follow a non-optimal path, say a zigzagging route, or it may visit the corners of the grid in its route toward the goal. Thus, deciphering an agent's behaviour when solving a task is about inferring the agent's preferences regarding the history of observed actions (Masters and Sardiña 2021).

This paper aims to learn the behaviour of an agent, either a robot or a human, from a set of plans reflecting their doing for achieving a goal, that is, how the agent solves similar planning problems. We focus on high-level actions rather than on low-level activity recognition as well as on problems/domains where multiple solution plans exist for a problem, and the agent has a preference for some plans over others. Specifically, given a planning model that includes the set of possible actions, various problems specified by their initial state and the desired goals, and the solution plans of the agent to solve these problems in the form of state-action pairs, we aim to learn the **strategy** or **behaviour** of the agent to achieve the tasks' goals. The learned behaviour is defined as a policy, specifically as a set of rules that determine which action to take in a state that satisfies certain conditions.

Our proposal builds on a Relational Decision Tree (RDT), also known as a Logical Decision Tree, to learn and represent the behaviour. Unlike classification trees such as ID3 or CART where a learning example is a collection of attribute-value representations, and queries of intermediate nodes are of the form $v \leq x$, where v is a variable that denotes a property of the sample and $x \in \mathbb{R}$ is the value of such property, in RDT examples and queries are described as a conjunction of logical facts. We use the Top-down Induction of

Logical Decision Trees (TILDE) (Blockeel and De Raedt 1998; Blockeel et al. 1999), which is a first-order logical decision tree learner. The learning algorithm of TILDE is biased according to a specification of syntactic restrictions called language bias. Background knowledge can also be incorporated to facilitate the deduction of new information, thus enhancing the reasoning process. In our approach, the language bias and the background knowledge are automatically extracted from the PDDL specification of the planning domain. Our findings demonstrate that these two elements offer valuable insights into constructing a more precise depiction of the targeted behaviour. Another advantage of using RDTs is that far fewer samples are needed compared to classification trees.

As a second objective, we will create a library with the behaviours learned from the plans followed by the agents when solving various planning tasks. Then, we will design a classification algorithm to identify the closest behaviour in the library for a given incomplete sample of an agent's plan execution. Ultimately, the purpose is to check the accuracy of the behaviour patterns of the library to unequivocally identify the behaviour of an agent from a partially-observed execution of its plan.

This paper is organized as follows. The next section presents some related work, and section **Problem definition** defines identification problems, sections **Learning behaviours** and **Behaviour identification** detail the steps of the behaviour learning and identification processes, respectively. Then, section **Experiments** shows the experiments performed to assess the validity of our approach and their results. We finish with some conclusions and future work.

Related work

The formulation of our human behaviour recognition problem falls close to Imitation Learning (IL), a paradigm that aims to mimic human behaviour without the need for explicit reward functions as it happens in Reinforcement Learning. The key components of IL are the demonstrations, typically given in the form of state-action pairs, and a specification of the environment. The task consists in learning a mapping between the observations (states) and the actions taken by the agent (Hussein et al. 2017). Most works on IL approach the learning problem using neural-based methods, requiring a significant number of samples and resulting in models that do not generalize well on new instances of the task (Duan et al. 2017). Our proposal, however, relies on the use of a symbolic method such as an RDT, which requires far few samples, it is able to better capture the mapping between states and actions, and, more importantly, the decisions are easily interpretable by humans.

Our learning human behaviour problem is closely related to a certain extent with the well-known goal/plan recognition problem in Automated Planning (Sukthankar et al. 2014) in the sense that both problems reason about the goals and execution process of an intelligent agent from plan observations, that is, from observations of its sequence of activities. Techniques to plan recognition encompass library-based methods whose objective is to recognize the plan an agent is executing from a plan library (Mirsky, Keren, and Geib 2021) and library-free approaches such as the 'plan recognition as planning' scheme, where the task of recognizing the goals and plans of the agent is translated into a plan generation problem (Ramírez and Geffner 2009).

While plan recognition seeks the plan that best fits an incomplete plan observation, our problem puts the emphasis on how the agent acts, on uncovering the relationships between the observed activities of the agent. The main difference lies in that plan recognition aims to infer the plan for a given (partial) plan observation while behaviour recognition requires multiple observation sequences so as to extract the common relationships that point at a common behaviour. Put differently, we may observe an agent zigzagging toward the goal in one sequence of activities, but we will not be able to deduce this is the agent's behaviour unless we observe the same zigzagging behaviour in other demonstrations. A behaviour is therefore interpreted as a set of general rules or an abstract policy that determines what operation must be done under particular state conditions to comply with the given behaviour. Some approaches address this task with classification trees using samples where the states are abstractly represented by means of a set of features (Sitanskiy, Sebastia, and Onaindia 2021). In contrast, generalized planning aims to infer the action to apply in any state of a planning domain, and it is typically expressed as a policy, an algorithmic program, or as a logical specification (Srivastava, Immerman, and Zilberstein 2011; Celorrio, Aguas, and Jonsson 2019; Fitzpatrick et al. 2021).

In this paper, we use TILDE, a first-order logical decision tree learner which has also been used for guiding control in forward-state planning problems modeled as a relational classification task (De la Rosa et al. 2011). This task captures the preferred action to select in a planning context, which is defined by the set of helpful actions of the current state, the remaining goals to achieve, and the static predicates of the planning task. The learned action policy is used to guide the search and the authors show that this approach solves larger problems than state-of-the-art planners in some domains (De la Rosa et al. 2011). However, the policy performs poorly in other domains because the context is not able to represent concepts that are necessary to discriminate between good and bad actions. For this reason, a bagging approach for learning ensembles of RDTs that yields, on average, plans of better quality is introduced in (de la Rosa and Fuentetaja 2017).

Problem definition

We assume that the input to our behaviour recognition problem is a set of sequences of activities (plans) performed by a person, as well as the specification of the domain in which the activity is taking place. We define the structure of these elements:

Definition 1 *A* **planning domain** *is represented by a pair* $\mathcal{D} = \langle \mathcal{P}, \Theta \rangle$*, where* $\mathcal{P}$ *is the set of predicates and* Θ *is the set of operators of the domain. A predicate* $p_i \in \mathcal{P}$ *is defined as* $p_i = (name(p_i)\ arg(p_i))$ *where the first element is the predicate name and the second element is the list of arguments. A* **planning task** *is represented by a triple*

```
(define (domain trolley-robot)
   (:requirements :strips :typing)
   (:types package - object place - object)
   (:predicates
     (at ?obj - package ?loc - place)
     (in ?pkg - package)
     (at-robot ?loc - place))
  (:action LOAD
     :parameters (?pkg - package
                  ?loc - place)
     :precondition (and (at-robot ?loc)
                    (at ?pkg ?loc) )
     :effect (and (not (at ?pkg ?loc))
                  (in ?pkg) ))
  (:action UNLOAD
     :parameters (?pkg - package
                  ?loc - place)
     :precondition (and (at-robot ?loc)
                    (in ?pkg) )
     :effect (and (not (in ?pkg))
                  (at ?pkg ?loc) ))
  (:action MOVE
     :parameters (?loc-from - place
                  ?loc-to - place )
     :precondition (and (at-robot ?loc-from))
     :effect (and (not (at-robot ?loc-from))
                  (at-robot ?loc-to)))
)
```

Figure 1: PDDL description of the *TrolleyRobot* domain

$P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, in which $\mathcal{I}$ is the initial state, and $\mathcal{G}$ is the goal state. We denote by $\mathcal{F}$ and $\mathcal{A}$ the set of facts and actions, respectively, that can be instantiated from $\mathcal{I}$; that is, $\mathcal{F}$ and $\mathcal{A}$ define the search space of the planning task P. A **solution** *to a planning task P is a plan π that reaches $\mathcal{G}$ from $\mathcal{I}$ by following a sequence of transitions defined in $\mathcal{A}$.*

As an illustrating example, we focus on a simplified version of a transport domain, named *TrolleyRobot*, that pictures a robot responsible for delivering a set of packages. There is a single robot that carries a trolley to put the packages in. This domain is defined by three predicates, { `at, at-robot, in` }, which denote where a package is, where the robot is and that a package is inside the trolley, respectively; additionally, the domain has three operators, { `load, unload` and `move` }, that describe the conditions and effects for loading a package inside the trolley, unloading a package from the trolley and moving the robot between two locations. Figure 1 shows the details of this domain.

An *agent* is a human or any software agent with capabilities to plan. Different agents can have different strategies or behaviours for solving a planning task; even the same agent can exhibit different behaviours depending on the context. For example, in the *TrolleyRobot* domain, a behaviour that an agent can stick with is to **deliver the packages one by one** (we will refer to this behaviour as ByOne).

Definition 2 *A* **behaviour** *β is a set of rules of the form $\beta = \{(\rho_1, \theta_1), \ldots, (\rho_r, \theta_r)\}$, where a rule (ρ_i, θ_i) indicates that operator θ_i should be executed in a state that satisfies the set of conditions ρ_i in order to follow the behaviour β.*

The conditions ρ_i express different constraints to be satisfied in a state. For example, in the *TrolleyRobot* domain, two possible conditions are: *the robot is at the same location as a package*, or *the number of packages in the trolley is zero*. Extending the work in (Sitanskiy, Sebastiá, and Onaindia 2020), where some behaviour models are created paying attention only to the operator executed at each situation, our rules determine which operator and with which parameters should be executed under which conditions. For instance, in the ByOne behaviour, we can find a rule like this: ({the robot is at the same location l as a package p and the number of packages in the trolley is zero}, `load`(l, p)).

Definition 3 *A* **corpus** *for a behaviour β is a pair $\Xi_\beta = \langle \mathcal{D}, \mathcal{Q} \rangle$, where $\mathcal{D}$ is the planning domain that defines the environment of the agent and $\mathcal{Q} = \{(P_1, \pi_1), \ldots, (P_n, \pi_n)\}$, where P_i is a planning task, and π_i is the solution plan executed by the agent for solving P_i.*

A corpus implicitly reflects the strategy followed by the agent for solving the set of planning tasks in $\mathcal{Q}$. We are interested in extracting the patterns that make this behaviour explicit, which is formalized in Definition 2.

Definition 4 *Given a corpus $\Xi_\beta = \langle \mathcal{D}, \mathcal{Q} \rangle$, the* **behaviour learning** *(BL) problem consists in generating a set of rules $\beta = \{(\rho_1, \theta_1), \ldots, (\rho_r, \theta_r)\}$ such that these rules encapsulate the behaviour followed by the agent when solving the tasks P_i with plans π_i of $\mathcal{Q}$.*

Definition 5 *A* **behaviour library** *$\Gamma_\mathcal{D}$ for a planning domain $\mathcal{D}$ is a set of all the behaviours β that have been learned for $\mathcal{D}$.*

For example, we can also learn another behaviour in the *TrolleyRobot* domain, the LoadAll behaviour, which consists of loading all the packages onto the robot before starting to unload them. In this case, both ByOne and LoadAll behaviours will compose $\Gamma_{TrolleyRobot}$.

Definition 6 *Given a planning domain $\mathcal{D}$ and a behaviour library $\Gamma_\mathcal{D}$, a* **behaviour identification** *(BI) problem is a tuple $\langle \mathcal{D}, \Gamma_\mathcal{D}, \mathcal{O} \rangle$ such that $\mathcal{O} = [o_1, \ldots, o_k]$ is a sequence of observations, where o_i is a pair (ϕ_i, a_i) composed of a state ϕ_i and the observed action $a_i \in \mathcal{A}$ executed in ϕ_i. The solution of a BI problem is the behaviour $\beta \in \Gamma_\mathcal{D}$ that better matches with the observation sequence $\mathcal{O}$.*

Learning behaviours

This section details our approach for solving the behaviour learning problem from a corpus $\Xi_\beta = \langle \mathcal{D}, \mathcal{Q} \rangle$. The output of the BL problem is a set of rules β that encapsulate the behaviour represented in Ξ_β. We use a Relational Decision Tree to learn and generate these rules. The whole BL process is depicted in Figure 2. The main steps are the following:

1. **Feature extraction**. A rule indicates the operator that must be executed in a state that satisfies the conditions in ρ_i (Def. 2). These conditions can be expressed in different ways. In this work, we will define and experiment with two sets of features to represent states. The first set

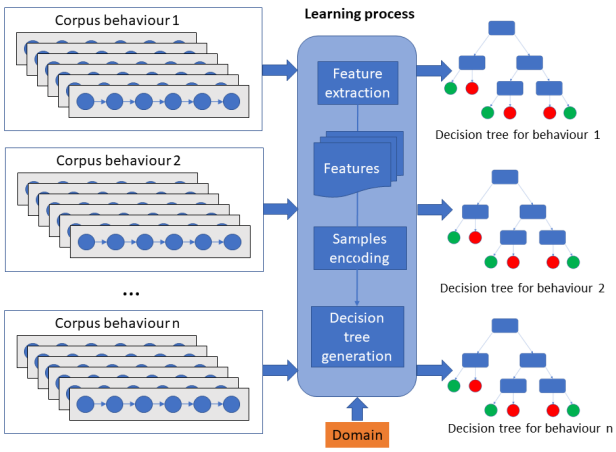

Figure 2: Learning process

is built directly from the predicates defined in the planning domain $\mathcal{D}$. The second is a more sophisticated feature set aimed to capture relationships between the facts of a state and the goal of the problem.

2. **Samples encoding.** A sample is a pair $\langle \mathbf{s_i}, \alpha_\mathbf{i} \rangle$ where $\mathbf{s_i}$ is a state and $\alpha_i \in \mathcal{A}$ is the action executed by the agent in $\mathbf{s_i}$. Samples for learning the RDT of a behaviour β are extracted from the corpus Ξ_β, specifically from the plans in $\mathcal{Q}$. Furthermore, the state $\mathbf{s_i}$ of a sample $\langle \mathbf{s_i}, \alpha_\mathbf{i} \rangle$ is encoded with the features obtained in the previous step.
3. **Construction of a decision tree**. We train a RDT for each Ξ_β with samples $\langle \mathbf{s_i}, \alpha_\mathbf{i} \rangle$. As already mentioned, given that we are using a logical representation of each state, we can use inductive reasoning to uncover the conditions under which a particular action is executed. This way, we will train a decision tree based on inductive learning, which induces an RDT. The RDT is then used to obtain the action (operator and parameters) to apply in a new encoded state. Subsequently, the RDT will be used to test how well a behaviour is recognized as the number of correct actions returned for a set of states.

Feature definition

We present the two sets of features that will be used to characterize a state of a planning problem. Our aim is to extract features automatically from the planning domain without any human intervention. First, we define two types of *basic predicates*:

1. **Precondition basic predicates**: they are simply formed with the predicates $\mathcal{P}$ that appear in the preconditions of the actions.
2. **Effect basic predicates**: they are generated from the *add* effects of the actions in $\mathcal{A}$. We add a suffix `_goal` to these predicates to denote that they can appear in $\mathcal{G}$.

For example, given the `load` action, the corresponding precondition basic predicates are {`(at-robot ?l)`, `(at ?p ?l)`} and the effect basic predicate is {`(in_goal ?p)`}.

We will denote as F_b the first set of features composed of the precondition and effect basic predicates. The second set, denoted as F_c, comprises *combined predicates*. The idea is to represent the state more compactly, linking predicates that are needed for the execution of an action or that are obtained as a result of the action execution. The features in F_c are built by combining two or more predicates that produce a new (more compact) predicate representing the combined concept. For example, from the combination of predicates `(at-robot ?l)` and `(at ?p ?l)`, the new combined predicate `(same-location-as-robot ?p ?l)` expresses that the robot is at the same location as a package. Our hypothesis is that this compact representation will facilitate identifying patterns from the plans.

In the context of logic programming, a combined predicate can be obtained from the application of rules. For example, the previous predicate could be obtained by applying the following rule:

```
(same-location-as-robot ?p ?l) :-
(at-robot ?l), (at ?p ?l)
```

The set F_c comprises the basic predicates as well as the combined predicates that are automatically built by means of the following two types of rules:

1. **Precondition rules**: Let $AS_\theta = 2^{pre(\theta)}$ be the power set of the preconditions of the operator $\theta \in \Theta$, that is, the set that comprises all the subsets that can be formed with the preconditions of each operator in Θ. The precondition rules are obtained by combining two or more predicates of each set in AS_θ. That is, a new rule is built for each set $S \in \{AS_\theta : |S| \geq 2\}$:

$$b_i \text{ :- } S \;\; where \;\; arg(b_i) = \bigcup_{\forall p_j \in S} arg(p_j) \qquad (1)$$

Assuming the set $\{p_1, p_2, \ldots, p_n\} \in S$, these predicates are combined in the body of the rule ($\forall p_j \in S$), $name(b_i)$ is the name of the new predicate, and $arg(b_i)$ is the parameters of the new predicate, which is the union of the parameters of predicates $\{p_1, p_2, \ldots, p_n\}$.

2. **Combined rules**: Given an operator θ, a combined rule is created from the combination of a predicate b_i obtained from a precondition rule (built from the preconditions of θ) and an effect basic predicate p that appears in the *add* effects of the operator θ. Specifically, given b_i and p, a combined rule is built as follows:

$$c_i \text{ :- } b_i, p \;\; where \;\; arg(c_i) = arg(b_i) \cup arg(p) \qquad (2)$$

These rules try to capture the relationship between all the predicates in the preconditions of each action and the relationship of these predicates with the effects of the action.

Let's take the `load` operator of the *TrolleyRobot* domain (see Figure 1), which has two preconditions (`(at-robot ?l)` and `(at ?p ?l)`) and one add effect (`(in ?p)`), to show some examples of how the precondition and combined rules can be built:

$AS_L = \{\, \emptyset, \{\texttt{(at-robot ?l)}\}, \{\texttt{(at ?pkg ?l)}\},$
$\{\texttt{(at-robot ?l), (at ?p ?l)}\}\,\}$

Taking only those sets with a cardinality of 2 or higher, the following predicate rule is obtained, according to Eq. 1:

```
(at_at-robot ?p ?l) :-
                (at ?p ?l),(at-robot ?l)
```

This new predicate `at_at-robot` and the add effect are then used to build the next combined rule, according to Equation 2:

```
(at_at-robot_in_goal ?p ?l) :-
        (at_at-robot ?p ?l),(in_goal ?p)
```

States encoding

In order to apply inductive learning, the states traversed when executing the observed plans are used as input samples for the decision tree learner, according to Figure 2. This section will explain the steps needed to prepare these samples depending on which set of features we are using.

The inputs to the TILDE algorithm are several files that, broadly speaking, include the knowledge base (set of examples), the language bias and the background knowledge. Each example has a class label and a set of facts, which encodes properties and relations that occur in the example. The language bias specifies restrictions in the values of arguments of the learning examples. Also, some background knowledge may be provided, which allows deducing additional facts from those in the examples.

The first step is extending the precondition basic predicates with two extra arguments: `St` (state) that links the literals belonging to the same example, and `Pr` (problem) that links the samples belonging to the same problem. For example, predicate `(at ?p ?l)` becomes `at(St,Pr,P,L)`. The effect basic predicates are extended only with the `Pr` argument because they may denote the problem goal and do not change for each specific state.

The **knowledge base** contains the learning examples. Each example is described by a heading that denotes the task name, the state identifier, the problem identifier and the known class, which in our case, denotes the operator to be executed in that state. For example, `trolleyrobot(1,1, move)` indicates that this is the first example in the first problem, and the corresponding operator is `move`. Then, a set of extended facts (that is, instantiated extended predicates as outlined above) describe the state in which the operator would be applied. Figure 3 shows the samples for one problem in the *TrolleyRobot* domain. As can be observed, the facts in each example correspond to the instantiation of the precondition basic predicates. Once a goal is achieved, we change the precondition basic predicates with the corresponding effect basic predicates, as in example 4 in Figure 3. The final state reached after executing the whole plan is also included; in this case, as no action will be executed in this state, we use the class `ok`.

The **language bias** specifies restrictions in the values of arguments of the learning examples. In our case, it is automatically extracted from the definition of the features explained in the previous section and consists of the type and

```
trolleyrobot(1,3, load).        trolleyrobot(3,3, unload).
at(1,3,obj21,pos2).             in(3,3,obj21).
at-robot(1,3,pos2).             at-robot(3,3,pos1).

trolleyrobot(2,3, move).        trolleyrobot(4,3, ok).
in(2,3,obj21).                  at_goal(3,obj21,pos1).
at-robot(2,3,pos2).             at-robot(4,3,pos1).
```

Figure 3: Some samples for learning the by-one behaviour

access mode of all the features. For example, we can find the following specifications for the basic predicates:

```
type(at_goal(problem,package,place)).
rmode(at_goal(+Pr,+Package,+Place)).
type(at(state,problem,package,place)).
rmode(at(+St,+Pr,+-Package,+-Place)).
```

In this case, predicate `at` comes directly from the precondition basic predicates, and predicate `at_goal` comes from the effect basic predicates. If we are using F_c, additional predicates are described in the language bias. In the following snippet, predicate `at_at-robot` comes from the precondition rules, and predicate `in_at_goal` comes from the combined rules.

```
type(in_at_goal(state,problem,package,place)).
rmode(in_at_goal(+St,+Pr,+-Package,+-Place)).
type(at_at-robot(state,problem,package,place)).
rmode(at_at-robot(+St,+Pr,+-Package,+-Place)).
```

The **background knowledge** contains the (automatically derived) rules that can be used to generate the combined predicates from the precondition and effect basic predicates. For example:

```
at_at-robot(St,Pr,X,Y):-at(St,Pr,X,Y),
                          at-robot(St,Pr,Y).

in_at_goal(St,Pr,X,Y):-in(St,Pr,X),
                         at_goal(Pr,X,Y).

at-robot_in_at_goal(St,Pr,X,Y):-
            at-robot(St,Pr,Y), in(St,Pr,X),
            at_goal(Pr,X,Y).
```

This way, the combined features will be automatically generated using the basic predicates in each example. It is important to remark that this knowledge is only provided when we are using the set of features F_c

Additionally, for each problem, the set of facts in $\mathcal{G}$ are also included as background knowledge. For example, `at_goal(3,obj21,pos1).`

Construction of the relational decision tree

The language bias file also includes a statement that indicates the task to solve. For example, `(predict(logtruck(+St,+Pr,-action))` specifies that the task is to predict the action of a given state (`+St`) and problem (`+Pr`). Additionally, it is necessary to list which classes can be predicted; for instance, in this case: `classes([load, unload, move, ok])`, which correspond to the operators in the domain plus the class `ok` to indicate that the goal has been reached.

With all the information in the knowledge base, language bias and background knowledge files, TILDE builds a relational decision tree that represents a set of disjoint rules of action selection that compose the learned behaviour β. The internal nodes of the tree contain the set of conditions under which the decision can be made. The leaf nodes contain the corresponding class.

Figure 4 shows the decision tree learned for the ByOne behaviour of the *TrolleyRobot* domain using the set of features F_c. Regarding this tree, the first branch states (`at-robot_in_at_goal(A,B,-D,-E)` that when the robot is at the same location `E` as the goal location of a package `D` that it is carrying, then the operator to be executed is `unload`; this was the operator selected in ten over ten examples of the knowledge base. The second branch says that if a package `F` is at the same location `G` as the robot (`at_at-robot(A,B,-F,-G)`), but it is already carrying another package `H` (`in_at_goal(A,B,-H,-I)`), then the robot has to move to a different location and not to load package `F`, because only one package can be carried at the same time; this operator was selected five over five examples. However, if no package is inside the trolley, then the operator to execute is `load` (selected ten over ten examples). The remaining conditions result in the `move` operator or indicate that the goal has been reached. The accuracy of this tree is 1.

Figure 5 shows the RDT learned for the ByOne behaviour with F_b. In this case, the tree is simpler, but it is not able to return the operator `unload` in any leaf. In fact, the first leaf `move` is learned with 20 examples, but 10 examples correspond to the `unload` class. This indicates that the RDT is not capable of distinguishing in which situations should use one operator or another. The accuracy of this tree is 0.75.

Finally, Figure 6 shows the RDT learned for the LoadAll behaviour with F_c. This behaviour consists in collecting all the packages before starting to deliver them. In this case, we can observe that, unlike with ByOne behaviour, when there is a package in the same location of the robot (`at_at-robot(A,B,-D,-E)`), the selected operator is always `load`. Moreover, if the robot is at the location where a package must be delivered (`at-robot_in_at_goal(A,B,-F,-G)`) but there is still a package that has not been picked up (`at(A,B,-H,-I)`), then the selected operator is `move`; that is, the package `F` is not delivered because package `H` has to be picked up first. Otherwise, package `H` is delivered. In this case, we can observe that this leaf is supported by 17 `unload` examples and 1 `move` example. The remaining conditions result in the `move` operator or indicate that the goal has been reached. The accuracy of this RDT is 0.98.

Behaviour identification

This section details how to use the RDT to solve the behaviour identification problem. Given a behaviour library $\Gamma_{\mathcal{D}}$ for domain $\mathcal{D}$ and a sequence of observations $\mathcal{O}$, the goal is to find the behaviour β^* that better matches these observations. Then, β^* is defined as follows:

$$\beta^* = \arg \max_{\beta_i \in \Gamma_{\mathcal{D}}} prob(\beta_i, \mathcal{O})$$

Function $prob(\beta_i, \mathcal{O})$ computes the probability that the RDT of β_i matches the observations in $\mathcal{O}$:

$$prob(\beta_i, \mathcal{O}) = \frac{|\{\phi_j : comp(\theta_j, \theta_{jRDT}), \forall(\phi_j, \theta_j) \in \mathcal{O}\}|}{|\mathcal{O}|}$$

Function $comp(\theta_j, \theta_{jRDT})$ returns whether the action θ_j is compatible with θ_{jRDT}, which is the action predicted by the RDT β_i when state ϕ_j is provided as a sample. To obtain this prediction, it is necessary first to encode the state ϕ_j into $\mathbf{s_j}$ as explained previously. Then, the encoded state $\mathbf{s_j}$ is given to the RDT, which returns the selected class, i.e. the operator to apply in $\mathbf{s_j}$. Additionally, the RDT returns a list of instantiated parameters that match the conditions in the RDT that lead to the corresponding leaf. These parameters, however, may not match directly with the parameters in θ_j. For example, if θ_j =`move(pos2, pos1)` and θ_{jRDT} =`move(obj4, pos2, obj2, pos1)`, we can deduce that both actions are compatible even though the parameters do not match one-to-one.

We evaluate the compatibility between both sets of parameters as follows. Let $\theta_j = a(p_1, \ldots, p_n)$ be the action in the plan and $\theta_{jRDT} = a_{RDT}(q_1, \ldots, q_m)$ be the action predicted by the RDT. Algorithm 1 returns whether θ_j and θ_{jRDT} are compatible. The first condition establishes that both operators must match and the set $match$ will contain the parameters of both actions. Thus, in the example, we find two parameters in θ_{jRDT} that match the arguments of θ_j: $match = \{\texttt{pos1}, \texttt{pos2}\}$. If $match$ is empty, the actions are not compatible. The sets rem_{θ_j} and $rem_{\theta_{jRDT}}$ contain the parameters of the action and the predicted action that did not match in the previous step. If there is a parameter $p_i \in rem_{\theta_j}$ so that there exists $q_k \in rem_{\theta_{jRDT}}$ and $type(p_i) = type(q_k)$, then both actions are not compatible because, in this case, the types match, but the values do not. However, if there is not such a parameter q_k then the actions are compatible. For example, if θ_{jRDT} =`move(obj4, pos2)`, $match = \{\texttt{pos2}\}$; therefore, $rem_{\theta j} = \{\texttt{pos1}\}$ and $rem_{\theta j}RDT = \{\texttt{obj4}\}$; $type(pos1) \neq type(obj4)$ and, consequently, both actions are compatible. If θ_{jRDT} =`move(pos4, pos2)`, then the actions would be not compatible.

The function $prob(\beta_i, \mathcal{O})$ returns the hitting rate of β_i with respect to the observations in $\mathcal{O}$. Therefore, β^* is the RDT with the highest hitting rate. If two (or more) RDTs return the same hitting rate, we assume that the behaviour is not correctly identified as it is not possible to unequivocally distinguish it among the collection of behaviours.

Experiments

This section describes the experiments we performed to analyze the validity of our approach. We aim to check to which extent the RDTs can identify specific behaviours. To do so, we first need to create the corpus Ξ_β, for each behaviour β that will be part of the behaviour library $\Gamma_{\mathcal{D}}$ for domain $\mathcal{D}$. To this end, we used synthetic data for experimentation, meaning that the corpus for each behaviour is artificially generated. As stated in Def. 3, a corpus is a list of planning tasks along with the solution plan of the agent for each

```
trolleyrobot(-A,-B,-C)
at-robot_in_at_goal(A,B,-D,-E) ?
+--yes: [unload] 10.0 [[load:0.0,unload:10.0,move:0.0,ok:0.0]]
+--no:  at_at-robot(A,B,-F,-G) ?
        +--yes: in_at_goal(A,B,-H,-I) ?
        |       +--yes: [move] 5.0 [[load:0.0,unload:0.0,move:5.0,ok:0.0]]
        |       +--no:  [load] 10.0 [[load:10.0,unload:0.0,move:0.0,ok:0.0]]
        +--no:  at(A,B,-J,-K) ?
                +--yes: [move] 9.0 [[load:0.0,unload:0.0,move:9.0,ok:0.0]]
                +--no:  in_at_goal(A,B,-L,-M) ?
                        +--yes: [move] 3.0 [[load:0.0,unload:0.0,move:3.0,ok:0.0]]
                        +--no:  [ok] 3.0 [[load:0.0,unload:0.0,move:0.0,ok:3.0]]
```

Figure 4: RDT obtained for the ByOne behaviour with F_c

```
trolleyrobot(-A,-B,-C)
in(A,B,-D) ?
+--yes: [move] 20.0 [[load:0.0,unload:10.0,move:10.0,ok:0.0]]
+--no:  at(A,B,-E,-F) ?
        +--yes: att(A,B,F) ?
        |       +--yes: [load] 10.0 [[load:10.0,unload:0.0,move:0.0,ok:0.0]]
        |       +--no:  [move] 7.0 [[load:0.0,unload:0.0,move:7.0,ok:0.0]]
        +--no:  [ok] 3.0 [[load:0.0,unload:0.0,move:0.0,ok:3.0]]
```

Figure 5: RDT obtained for the ByOne behaviour with F_b

Algorithm 1: compatible($\theta_j, \theta_{j\,RDT}$)

if $a \neq a_{RDT}$ **then**
  return false
end if
$match = \{p_i / p_i \in \{p_1, \ldots, p_n\} \wedge q_k \in \{q_1, \ldots, q_m\} : p_i = q_k\}$
if $match == \varnothing$ **then**
  return false
end if
$rem_{\theta_j} = \{p_1, \ldots, p_n\} - match$
$rem_{\theta_{j\,RDT}} = \{q_1, \ldots, q_m\} - match$
for all $p_i \in rem_{\theta_j}$ **do**
  if $\exists q_k \in rem_{\theta_{j\,RDT}} \wedge type(p_i) = type(q_k)$ **then**
    return false
  end if
end for
return true

task. In order to generate the corpus for a new behaviour in a basic domain $\mathcal{D}$, we handcrafted an alternative domain $\mathcal{D}_i$ from $\mathcal{D}$ that represents that behaviour β_i. Then, for each $\mathcal{D}_i$, we executed several planning tasks obtaining the corresponding corpus Ξ_i for such behaviour. We used the LAMA planner (Richter and Westphal 2010) for solving the problems. Therefore, behaviour β_i associated with the domain $\mathcal{D}$ is exhibited in the corresponding corpus Ξ_i.

To learn a RDT from a corpus Ξ_i, the samples are obtained as explained in Section *States encoding* and are used as input to the RDT algorithm. Finally, we used a different set of samples to perform the behaviour identification. For each domain, two types of experiments were performed: (1) Action prediction, which focuses on measuring the accuracy of the function *comp* for the specific behaviours defined for a domain and (2) Behaviour identification, which checks the performance of the behaviour identification process.

Behaviours description

The **TrolleyRobot** domain is about moving packages between locations within a city with a single robot truck as our focus is on simulating the behaviour of a single *acting agent*. We define two behaviours for this domain:

1. Behaviour ByOne: it consists of transporting packages to their destinations one at a time.
2. Behaviour LoadAll: it consists of loading all the packages onto the robot before starting to unload them.

The **Kitchen** domain is dedicated to solving the problem of making tea using the available resources: water, cups, kettle, and tea bags. To solve the task, the agent boils water using a kettle and prepares the tea using a tea bag, a clean cup, and boiled water. The problem may involve preparing several cups of tea. In this domain, we defined two behaviours:

1. Behaviour ByOne consists in collecting resources required to prepare one cup of tea, repeating the process for a new cup of tea and so on.
2. Behaviour TakeFirst consists in collecting all the needed resources before starting the making-tea process for all the required cups

We also work with the classical **Blocksworld** domain (Bacchus 2001). Aside from the behaviour comprised in the original domain, we define two more behaviours:

```
trolleyrobot(-A,-B,-C)
at_at-robot(A,B,-D,-E) ?
+--yes: [load] 17.0 [[load:17.0,unload:0.0,move:0.0,ok:0.0]]
+--no:  at-robot_in_at_goal(A,B,-F,-G) ?
        +--yes: at(A,B,-H,-I) ?
        |       +--yes: [move] 3.0 [[load:0.0,unload:0.0,move:3.0,ok:0.0]]
        |       +--no:  [unload] 18.0 [[load:0.0,unload:17.0,move:1.0,ok:0.0]]
        +--no:  in_at_goal(A,B,-J,-K) ?
                +--yes: [move] 16.0 [[load:0.0,unload:0.0,move:16.0,ok:0.0]]
                +--no:  at(A,B,-L,-M) ?
                        +--yes: [move] 1.0 [[load:0.0,unload:0.0,move:1.0,ok:0.0]]
                        +--no:  [ok] 5.0 [[load:0.0,unload:0.0,move:0.0,ok:5.0]]
```

Figure 6: RDT obtained for the load_all behaviour

Domain	Training	Testing	Behaviour	F_b acc	F_c acc
Trolley Robot	5 pr.	85 probs.	ByOne	0.75	1
	2-5 pck.	5-21 pck.	LoadAll	0.42	0.98
Kitchen	4 pr.	43 probs.	ByOne	0.66	0.97
	1-3 cups	2-5 cups	TakeFirst	0.81	1.0
Blocks world	4 pr.	43 probs.	UnstackAll	0.9	0.99
	3-11 blc	3-94 blc	LimitedTable	0.76	0.89

Table 1: Experiment settings

1. Behaviour UnstackAll does not allow the robot hand to stack a block onto another block before all existing stacks have been disassembled. The robot hand first unstacks all the blocks, placing them on the table, and then stacks the blocks to get the target configuration.
2. Behaviour LimitedTable forces the robot to build a towers, limiting space on the table by only 2 additional empty slots.

Results

This section presents the results from our experiments. Table 1 shows the number and characteristics of the samples for training and testing the RDTs and the obtained accuracy when training with features F_b and F_c. In all cases, the accuracy of the RDT-F_c is higher than RDT-F_b, which indicates that the combined features capture better the interactions between the state facts.

To analyze the results of the **action prediction** and the accuracy of Algorithm 1, we take an observation (ϕ_j, θ_j) and obtain the corresponding $\theta_{j\,RDT}$ using the RDF β_i. We distinguish between two levels of predictions: *operator prediction*, where a hit is counted when the action name of θ_j coincides with the action name of $\theta_{j\,RDT}$ and *params prediction*, where a hit is counted when function $comb(\theta_j, \theta_{j\,RDT})$ returns true. We can observe in Table 2 that the accuracy of RDT-F_c is higher in all cases than the accuracy of RDT-F_b, saved for the *params prediction* of the behaviour LimitedTable of the **Blocksworld** domain. This behaviour is more complex than others because it does not impose an ordering in the action execution, but a preference for not using more space than necessary on the table. Following the analysis of this result, we discovered that the RDT-F_b gives a slightly higher accuracy because its rules are less restrictive regarding the parameter selection than rules in the RDT-F_c. These rules perform well on problem sizes similar to training data, but with the growing difficulty of the problems, it starts to affect negatively the RDT-F_c accuracy.

Domain	Behaviour	F_x	Operator pred.	Params. pred.
Trolley Robot	ByOne	F_b	0.718	0.605
		F_c	0.968	0.936
	LoadAll	F_b	0.381	0.000
		F_c	0.999	0.999
Kitchen	ByOne	F_b	0.284	0.181
		F_c	0.958	0.543
	TakeFirst	F_b	0.780	0.540
		F_c	0.976	0.889
Blocks World	UnstackAll	F_b	0.964	0.661
		F_c	0.972	0.964
	LimitedTable	F_b	0.792	0.623
		F_c	0.855	0.541

Table 2: Accuracy in action prediction for F_b and F_c

Our second experiment is to analyze the **behaviour identification** process using RDTs-F_c. Table 3 shows the results obtained for each behaviour. We can observe that, in general, the process correctly recognizes almost all observation sequences. It is important to remark that the RDTs were trained with few small problems (see Table 1), but tested with much larger problems, which indicates that the RDTs generalize well the behaviours. However, for the ByOne behaviour in the kitchen domain and the LimitedTable behaviour in the blocksworld domain, the performance is really poor. This is a consequence of the low performance in the action precondition accuracy (see Table 2), which impacts the quality of the behaviour recognition results.

Conclusions

In this paper, we have presented a method for identifying different behaviours of an agent when solving a planning problem. Our approach defines two sets of features that are automatically extracted from the planning domain. Both sets describe a state using logical facts and are used to learn a Re-

Domain	Behaviour	Operator pred.	Params. pred.
TrolleyRobot	ByOne	1,000	1,000
	LoadAll	1,000	1,000
Kitchen	ByOne	1,000	0,049
	TakeFirst	1,000	1,000
Blocksworld	UnstackAll	0,860	0,930
	LimitedTable	0,907	0,116

Table 3: Behaviour recognition accuracy

lational Decision Tree. Our experiments show that, for most of the domains and behaviours, and using the combined set of features, we can predict the action to be executed in a given state with an accuracy higher than 90%, and we can perfectly identify the behaviour in 4 out of 6 behaviours. However, we have detected some problems regarding the parameters prediction in two behaviours, that need a deeper analysis.

An advantage of our approach is that there is no need for many samples to train the RDTs and reach good performance. Moreover, training a new behaviour only requires providing the knowledge base as the language bias and the background knowledge are general for a given domain and independent of the behaviour.

As for future work, we will study augmenting the set of combined features, trying to include features that reflect negative situations in the state, for example, there is *not* any package inside the truck. Additionally, we will perform experiments with new domains and behaviours, to analyze the generality of this approach.

References

Bacchus, F. 2001. AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *AI magazine*, 22(3): 47–47.

Blockeel, H.; and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2): 285–297.

Blockeel, H.; De Raedt, L.; Jacobs, N.; and Demoen, B. 1999. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3: 59–93.

Bouchabou, D.; Nguyen, S. M.; Lohr, C.; Leduc, B.; and Kanellos, I. 2021. A Survey of Human Activity Recognition in Smart Homes Based on IoT Sensors Algorithms: Taxonomies, Challenges, and Opportunities with Deep Learning. *Sensors*, 21(18): 6037.

Celorrio, S. J.; Aguas, J. S.; and Jonsson, A. 2019. A review of generalized planning. *Knowl. Eng. Rev.*, 34.

Chua, S.; Marsland, S.; and Guesgen, H. W. 2011. Behaviour Recognition in Smart Homes. In Walsh, T., ed., *IJCAI 2011*, 2788–2789. IJCAI/AAAI.

de la Rosa, T.; and Fuentetaja, R. 2017. Bagging strategies for learning planning policies. *Annals of Mathematics and Artificial Intelligence*, 79: 291–305.

De la Rosa, T.; Jiménez, S.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research*, 40: 767–813.

Degardin, B.; and Proença, H. 2021. Human Behavior Analysis: A Survey on Action Recognition. *Applied Sciences*, 11(18): 8324.

Duan, Y.; Andrychowicz, M.; Stadie, B.; Jonathan Ho, O.; Schneider, J.; Sutskever, I.; Abbeel, P.; and Zaremba, W. 2017. One-Shot Imitation Learning. In *Advances in Neural Information Processing Systems*, volume 30.

Fitzpatrick, G.; Lipovetzky, N.; Papasimeon, M.; Ramírez, M.; and Vered, M. 2021. Behaviour Recognition with Kinodynamic Planning Over Continuous Domains. *Frontiers Artif. Intell.*, 4: 717003.

Guesgen, H. W.; and Marsland, S. 2016. Using Contextual Information for Recognising Human Behaviour. *Int. J. Ambient Comput. Intell.*, 7(1): 27–44.

Hussein, A.; Gaber, M. M.; Elyan, E.; and Jayne, C. 2017. Imitation Learning: A Survey of Learning Methods. *ACM Comput. Surv.*, 50(2).

Masters, P.; and Sardiña, S. 2019. Goal Recognition for Rational and Irrational Agents. In Elkind, E.; Veloso, M.; Agmon, N.; and Taylor, M. E., eds., *AAMAS 2019*, 440–448. International Foundation for Autonomous Agents and Multiagent Systems.

Masters, P.; and Sardiña, S. 2021. Expecting the unexpected: Goal recognition for rational and irrational agents. *Artif. Intell.*, 297: 103490.

Meneguzzi, F.; and Pereira, R. F. 2021. A Survey on Goal Recognition as Planning. In Zhou, Z., ed., *IJCAI 2021*, 4524–4532. ijcai.org.

Mirsky, R.; Keren, S.; and Geib, C. 2021. *Introduction to Symbolic Plan and Goal Recognition*, volume 16 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers.

Ramírez, M.; and Geffner, H. 2009. Plan Recognition as Planning. In Boutilier, C., ed., *IJCAI 2009*, 1778–1783.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Sitanskiy, S.; Sebastiá, L.; and Onaindia, E. 2020. Agent behaviour recognition using text analysis. In *ICAPS 2020 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Sitanskiy, S.; Sebastia, L.; and Onaindia, E. 2021. Learning Behaviour Based On Automated Feature Extraction. In *IJCAI 2021 Workshop on Generalization in Planning*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artif. Intell.*, 175(2): 615–647.

Sukthankar, G.; Geib, C.; Bui, H. H.; Pynadath, D. V.; and Goldman, R. P., eds. 2014. *Plan, Activity, and Intent Recognition: Theory and Practice*. Morgan Kaufmann.