GenGPT: A Package to Generate Synthetic Goal-Plan Trees

Yuan Yao

School of Computer Science and Technology
Zhejiang University of Technology
yaoyuan@zjut.edu.cn

GenGPT is a package to generate synthetic goal-plan trees (GPTs)[1, 2]. Goal-plan trees are widely used in BDI-based agent programming to represent the relations between goals, plans and actions, and to reason about the interactions between intentions. The root of a goal-plan tree is a top-level goal (goal-node), and its children are the plans that can be used to achieve the goal (plan-nodes). Plans may in turn contain primitive actions (action nodes) and subgoals (goal nodes), giving rise to a tree structure representing all possible ways an agent can achieve the top-level goal.

GenGPT can be used in a standalone fashion using the bundled Java application to generate a set of goal-plan trees (in XML format) as input to another program. Alternatively, the source code provided can be integrated directly into another program. Note that GenGPT is licensed under the GNU Public License (GPL) Version 3. See the file `LICENSE.txt` for details.

Section 1 below describes the characteristics of the synthetic goal-plan trees generated by GenGPT, and explains how to generate sets of tress using the GenGPT java application. Section 2 briefly describes the organisation of the code.

1 Running the GenGPT Application

The application is packaged as a jar file, and is invoked from the command-line as

```
java -jar GenGPT.jar <args>
```

The arguments required to generate a set of goal-plan trees are: the maximum depth of the goal-plan tree $\#\delta$, the number of subgoals in each non-leaf plan (leaf plans contain only action nodes) $\#\gamma$, the number of plans to achieve each goal $\#\pi$, the number of actions in each plan $\#\alpha$, the probability that a plan being a leaf plan ρ, the number of environment

variables $\#\nu$, the number of environment variables selected for each GPT $\#\varepsilon$ the number of GPTs generated $\#T$ and the output file path $Path$ to which the forest of goal-plan trees is saved. The flag for each argument, their default values and the constraints on each argument are shown in Table 1.

Table 1: Arguments required to generate the goal-plan trees

Symbol	Flag	Default	Constraints	Description
$\#\delta$	-d	3	$\#\delta \geq 1$	Maximum depth of the tree
$\#\gamma$	-g	3	$\#\gamma \geq 1$	Number of subgoals in each plan
$\#\pi$	-p	3	$\#\pi \geq 1$	Number of plans to achieve a goal
$\#\alpha$	-a	3	$\#\alpha \geq 2$	Number of actions in each plan
ρ	-l	0	$1 \geq \rho \geq 0$	Probability of a plan being leaf plan
$\#\nu$	-v	60	$\#\nu \geq 1$	Total number of environment variables
$\#\varepsilon$	-e	30	$\#\nu \geq \#\varepsilon \geq 1$	Number of selected variables for each GPT
$\#T$	-t	10	$\#T \geq 1$	Number of goal-plan trees
$Path$	-f	gpt.xml	File name	Output file path

The arguments $\#\delta$, $\#\gamma$, $\#\pi$, $\#\alpha$ and ρ together determine the shape of the GPTs in the forest.[1] $\#\delta$ decides the maximum depth of the GPT, that is, all plans in depth $\#\delta$ are leaf plans. All other plans that are not at depth $\#\delta$ have ρ chance to be a leaf plan. $\#\nu$ represents the total number of variables in the environment and $\#\varepsilon$ is the number of variables that may appear as the postcondition of actions in each GPT. By varying the value of $\#\nu/\#\varepsilon$, we can vary the likelihood of actions and plans in different GPTs having the same pre- and postconditions, and hence the probability of both positive and negative interactions between GPTs.

The set of GPTs is saved in XML format. The BNF for the XML format is shown in Figure 1. The XML file contains a forest of GPTs. Each goal γ has a set of plans $\pi_1, \ldots, \pi_n$ to achieve it. Each plan π consists of a sequence of execution steps, $\pi = \alpha_1; \ldots; \alpha_m$, where each α_i is either an action or a subgoal. The actions and subgoals in each plan are executed sequentially. We model the environment as a set of propositional variables, and define precondition, postcondition and goal-condition as sets of literals. Each literal is denoted by a pair $(ID, State)$ in the XML file, where ID is an identifier of the proposition and $Boolean$ is its current state, i.e., true or false. An example GPT generated by GenGPT with parameters $\#\delta = 5$, $\#\gamma = 1$, $\#\pi = 2$, $\#\alpha = 3$, $\rho = 0$, $\#\nu = 60$, $\#\varepsilon = 30$, $\#T = 1$, $Path =$ example-gpt.xml is shown in Figure 2).

[1] It would be more flexible to allow specification of the maximum and minimum number of actions and subgoals in each plan; we assume a fixed number of actions and subgoals for simplicity and precise control.

⟨Forest⟩ ::= ⟨Goal⟩ (, ⟨Goal⟩)*
⟨Goal⟩ ::= ⟨Name⟩ ⟨Goal-condition⟩ ⟨Plans⟩
⟨Plans⟩ ::= ⟨Plan⟩ (, ⟨Plan⟩)*
⟨Plan⟩ ::= ⟨Name⟩ ⟨Precondition⟩ ⟨PlanBody⟩
⟨PlanBody⟩ ::= ⟨Step⟩ (, ⟨Step⟩)*
⟨Step⟩ ::= ⟨Goal⟩ | ⟨Action⟩
⟨Action⟩ ::= ⟨Name⟩ ⟨Precondition⟩ ⟨Postcondition⟩

⟨Precondition⟩ ::= ϵ | ⟨Literals⟩
⟨Goal-condition⟩ ::= ϵ | ⟨Literals⟩
⟨Postcondition⟩ ::= ϵ | ⟨Literals⟩
⟨Literals⟩ ::= ⟨Literal⟩ (, ⟨Literal⟩)*
⟨Literal⟩ ::= (⟨ID⟩ , ⟨State⟩)
⟨ID⟩ ::= *String*
⟨State⟩ ::= *Boolean*

Figure 1: XML format in which goal-plan trees are saved

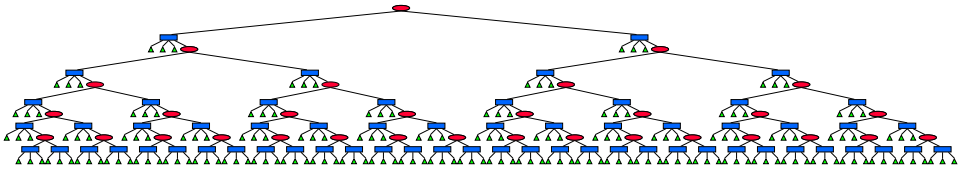

Figure 2: Example goal-plan tree.

2 The Code

The code consists of two main packages. The package `uno.gpt.structure` contains classes representing the tree nodes in the GPT and the literals appear in each node of the GPT. The package `uno.gpt.generator` contains classes to generate synthetic goal-plan trees, and the classes to save them in an XML file. In the remainder of this section, we focus on the tree generation code and briefly explain the rationale underlying tree generation.

2.1 GPT Structure

Goal nodes, plan nodes and action nodes in the GPT are represented using the java classes `GoalNode`, `PlanNode` and `ActionNode` . Each `GoalNode` consists of a name, a goal-condition, and a list of `PlanNodes` representing plans which achieve the goal. Each `PlanNode` contains its name, its context condition, and a list of nodes representing plan steps, which may be `GoalNodes` or `ActionNodes`. Finally, each `ActionNode` has its name, its precondition and postcondition.

All the literals in the pre-conditions, postconditions and goal-conditions appear in the nodes above are defined by the jave class *Literal*. The class *Literal* defines a tuple $(id, state)$, where id is the literal's identity and *state*

represents its current state. Functions *getID(), getState()* are used to get these two values respectively. Function *flip()* flips the state of the literal and return the state after flipping. The override function *equals()* compares two literals, and returns true if these two literals refer to the same object or these two literals have the same id and state.

2.2 Generation of GPT

The major GPT Generation process is defined in the Java class *SynthGenerator*. The method `genEnvironment` randomly generates the current environmnt states for synthetic GPTs. It first generates $\#T$ distinct litearals (initially false) as the goal-conditions of the top-level goals, then creates $\#v$ variables with random initial values. The variables and their corresponding values are then passed to the method `genTopLevelGoal` to generate GPTs (i.e. the top-level goal and all the hierarchies below). The method `genTopLevel Goal` is used to generate the n_{th} top-level goal which has the goal-condition $(G - n, true)$. The method `createGoal` is then used to recursively construct the top-level goal by adding plans to it. The method `createGoal` takes 4 parameters as input which are the current depth of the tree (`depth`), the set of selected literals for this GPT (`as`), the common preconditions for the plans to achieve the goal (`ps`), the goal condition (`gcs`). For each target goal, $\#\pi$ plans are generated. The context condition of the plans to achieve the target goal consists of two parts: the pure environment variables that won't be affected by the execution of the GPT (the value of these variables are determined by the current environment), and the variables that are established by the preceding steps of this goal. This ensures that there is at least one plan applicable to achieve each (sub)goal.

The method `createPlan` is then called to create the actions and subgoals in the plan. Similarly, the `createPlan` method also takes 4 parameters which are same to those required by the method `createGoal`. The first step and the last setp in a plan are actions by default.

The field `as` is used to represent the set of literals that can be used as the precondition of the action. When generating the first action in the plan, we assume it has the plan's precondition as its own precondition. And for each subsequent action that is not last step in the plan, we then generate the postcondition of the action by randomly selecting a literal l from `as`. And if the action is the last step of the plan, it's postcondition is the goal-condition this plan is going to achieve. The sets of literals `as` and `prec` are then updated. This updating process removes the literals which are made false by the new action, and adds the postcondition of the new action to both sets of literals. The updated sets are used for generating subsequent actions and subgoals.

A plan is a leaf plan if it reaches the depth of $\#\delta$ or, with probability ρ it is randomly chosen to be a leaf plan. Since leaf plan contains only actions,

if the plan is a leaf plan, when all $\#\alpha$ actions have been generated, then the generation process for this plan stops, if not, we have to generate $\#\gamma$ subgoals in the plan.

We keep creating actions, plans and (sub)goals until we reach the depth of the tree. The generation process will stop after we generate the last leaf action, and the whole goal-plan tree is translated and saved in an XML file by using the method in the java class `XMLWriter`.

References

[1] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting & avoiding interference between goals in intelligent agents. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 721–726, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[2] J. Thangarajah and L. Padgham. Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning*, 47(1):17–56, 2011.

[3] Yuan Yao, Lavindra de Silva, and Brian Logan. Reasoning about the Executability of Goal-Plan Trees Proceedings of the Fourth International Workshop on Engineering Multi-Agent Systems, 2016.