GenGPT: A Package to Generate Synthetic Goal-Plan Trees

Yuan Yao

School of Computer Science
University of Nottingham
yvy@cs.nott.ac.uk

GenGPT is a package to generate synthetic goal-plan trees [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 the set of goal-plan trees are: the 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 a goal $\#\pi$, the number of actions in each plan $\#\alpha$, the probability that actions and subgoals in a plan form part of a parallel composition ρ, the

number of environment variables that may appear in the tree as pre-, in-, and postconditions $\#\nu$, the number of goal-plan trees $\#T$ 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	$\#\delta \geq 1$	depth of the tree
$\#\gamma$	-g	1	$\#\gamma \geq 1$	number of subgoals in each plan
$\#\pi$	-p	2	$\#\pi \geq 1$	number of plans to achieve a goal
$\#\alpha$	-a	1	$\#\alpha \geq 1$	number of actions in each plan
ρ	-l	0	$1 \geq \rho \geq 0$	probability of parallel execution
$\#\nu$	-v	50	$\#\nu \geq 1$	number of environment variables
$\#T$	-t	1	$\#T \geq 1$	number of goal-plan trees
$Path$	-f	gpt.xml		output file path

The arguments $\#\delta$, $\#\gamma$, $\#\pi$ and $\#\alpha$ together determine the shape of the goal-plan trees in the forest.[1] By default, the actions and subgoals in each plan are executed sequentially. The parameter ρ specifies the probability that a plan will be generated in which the actions and subgoals are executed in parallel. $\#\nu$ represents the number of variables that may appear in each goal-plan tree (as the pre- and post-conditions of actions and plans). By varying the value of $\#\nu$ we can vary the likelihood of actions and plans in different goal-plan trees having the same pre- and postconditions, and hence the probability of both positive and negative interactions between goal-plan trees.

The set of goal-plan trees is saved in XML format. The BNF for the XML format is shown in Figure 1. The XML file contains a forest of goal-plan trees. Each goal γ has a set of plans $\pi_1, \ldots, \pi_n$ to achieve it. Each plan π consists of a sequence of actions and subgoals, $\pi = \alpha_1; \ldots; \alpha_m$, or a parallel composition of actions and subgoals, $\pi = \alpha_1 \| \ldots \| \alpha_m$, where each α_i is either an action or a subgoal. We model the environment as a set of propositional variables, and define pre, in- and postconditions as a set of literals. Each literal is denoted by an $(index, boolean)$ pair in the XML file, where *integer* is an identifier of the propositions and *boolean* is either true or false. An example goal-plan tree generated by GenGPT with parameters $\#\delta = 5$, $\#\gamma = 1$, $\#\pi = 2$, $\#\alpha = 3$, $\rho = 0$, $\#\nu = 100$, $\#T = 1$, $Path =$ example-gpt.xml is shown in Figure 2), and the XML can be found in the file `example-gpt.xml`.

[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.

⟨Forest⟩ ::= ⟨Goal⟩ (, ⟨Goal⟩)*
⟨Goal⟩ ::= ⟨Name⟩ ⟨Incondition⟩ ⟨Plans⟩
⟨Plans⟩ ::= ⟨Plan⟩ (, ⟨Plan⟩)*
⟨Plan⟩ ::= ⟨Name⟩ ⟨Precondition⟩ ⟨Incondition⟩ ⟨PlanBody⟩
⟨PlanBody⟩ ::= ⟨Step⟩ ((; ⟨Step⟩)* | (|| ⟨Step⟩)+)
⟨Action⟩ ::= ⟨Name⟩ ⟨Precondition⟩ ⟨Incondition⟩ ⟨Postcondition⟩

⟨Precondition⟩ ::= ϵ | ⟨Literal⟩
⟨Incondition⟩ ::= ϵ | ⟨Literal⟩
⟨Postcondition⟩ ::= ϵ | ⟨Literal⟩
⟨Literal⟩ ::= (⟨Variable⟩ , ⟨Value⟩)
⟨Variable⟩ ::= *Integer*
⟨Value⟩ ::= *Boolean*

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

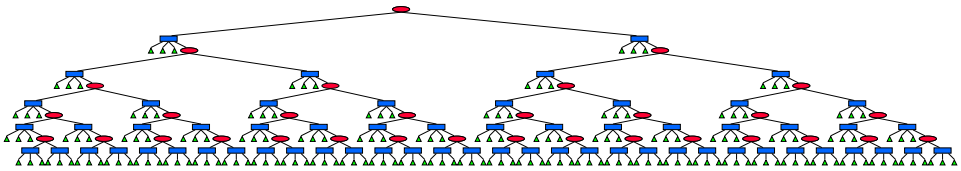

Figure 2: Example goal-plan tree.

2 The Code

The code consists of two main packages. The package `uno.gpt.nodes` contains classes representing the tree nodes (i.e., goal nodes, plan nodes, action nodes and parallel composition nodes) and literals. The package `uno.gpt.generator` contains the class `Generator` which generates a forest of goal-plan trees, and the class `XMLWriter` which saves the forest 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.

A *Literal* is defined as a tuple $(index, boolean)$, where $0 \leq index \leq \#\nu - 1$ (where $\#\nu$ is the number of environment variables), and *boolean* is either $True$ or $False$. Goals, plans, actions and parallel compositions are represented using the java classes `GoalNode`, `PlanNode`, `ActionNode` and `ParallelNode` respectively. Each `GoalNode` consists of a name, an in-condition and a list of `PlanNodes` representing plans which achieve the goal. Each `PlanNode` consists of a name, a pre-condition, an in-condition, and a list of nodes representing plan steps, which may be `GoalNodes`, `ActionNodes` or `ParallelNodes`. Each `ActionNode` consists of a name, together with the pre-, in- and postconditions of the action. Finally, each `ParallelNode` consists of a list of `ActionNodes` and `GoalNodes` to be executed in parallel.

The method `genTopLevelGoal` of the class `Generator` is used to iteratively generate goals, plans, and actions from the top-level goal to the leaf

nodes in the tree. It first generates the top-level goal, and then calls the method `constructGoal` to generate the plans to achieve the top-level goal, and then generates either a sequence of actions and subgoals or a parallel composition for each of these plans and so on. The method `constructGoal` is used to generate plans to achieve a target goal and takes 4 parameters as arguments which are the target goal (`gl`), the depth of the target goal (`currentdepth`), the set of literals which can be selected as the precondition of the plans to achieve the target goal (`var_pre`) and the set of literals which can be selected as the postcondition of the actions in the plans to achieve the target goal (`var_post`). To generate plans for a top-level goal, `gl` is set to be the top-level goal, `currentdepth` is 1, `var_pre` is empty and `var_post` contains all possible literals. For each target goal, $\#\pi$ plans are generated. If `var_pre` is not empty, the precondition of each plan is randomly selected from `var_pre`. Otherwise, a proposition i is randomly selected from the set of environment variables, where i is an index. The odd numbered plans to achieve the goal have the literal (i, *True*) as their precondition, and the even numbered plans have (i, *False*) as their precondition. This ensures that if $\#\pi > 1$, there is at least one plan applicable to achieve each (sub)goal.

As explained above, with probability 1-ρ a plan consists of an ordered sequence of actions and subgoals, and with probability ρ, it consists of a parallel composition of actions and subgoals. If a plan consists of an ordered sequence of actions and subgoals, the method `constructPlan` is called to create the actions and subgoals in it. The `constructPlan` method takes 4 variables as its input, which are the target plan (`pl`), the depth of the target plan in the goal-plan tree (`currentdepth`), the set of literals which can be selected as the precondition of the plans to achieve the subgoals in the target plan (`var_pre`) and the set of literals which can be selected as the postcondition of the actions in the target plan (`var_post`). (The parameters `currentdepth`, `var_pre`, `var_post` are the same as in the method `constructGoal`.) The actions in the plan are generated before the subgoals. The field `action_pre` 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, `action_pre` only contains the plan's precondition. Thus, the first action in each plan always has the plan's precondition as its own precondition. We then generate the postcondition of the action by randomly selecting a literal l from `var_post`. The sets of literals `action_pre` and `var_pre` are then updated as follows:

- if `action_pre` (or `var_pre`) contains the negation of l, remove the negation of l from `action_pre` (resp. `var_pre`);
- if `action_pre` (or `var_pre`) doesn't contain l, add l to `action_pre` (resp. `var_pre`).

This updating process removes the literals which are made false by the new

action from `action_pre` and `var_pre`, and adds the postcondition of the new action to both sets of literals. The updated sets `action_pre` and `var_pre` are used for generating subsequent actions and subgoals.

When all actions have been generated, if the plan is not a leaf plan (`currentdepth` != $\#\delta$), we also generate $\#\gamma$ subgoals. For each subgoal, the method `constructGoal` is called to add plans to achieve it. `gl` is the target subgoal, `currentdepth` is the current depth of the plan plus 1, and the other parameters are as in the method `constructPlan`. After generating each subgoal, we update the set of literals `var_pre`. As subgoals may be achieved by different plans, we use the *necessary postcondition* of the goal to present the set of literals that must be true after achieving this subgoal, and use *contingent postcondition* to present the set of literals that may be true after achieving this subgoal [3]. If the *necessary postcondition* and *contingent postcondition* of a subgoal are represented as $post_n$ and $post_c$ respectively, then the process of updating the literals in `var_pre` is as follows:

for each l in $post_c$ do
  if `var_pre` contains $\neg l$, remove $\neg l$ from `var_pre`

for each l in $post_n$ do
  if `var_pre` contains $\neg l$, remove $\neg l$ from `var_pre`
  if `var_pre` doesn't contain l, add l to `var_pre`

This process removes the literals that are (necessarily or contingently) made false by achieving this goal from `var_pre`, and adds the necessary postcondition of the goal to `var_pre`. The updated `var_pre` is then used to generate the subsequent subgoals in the plan. This process ensures the plan is executable if its precondition holds, and the choice of plans for the subgoal(s) in the plan is not affected by the postconditions of the actions in this plan.

If the plan consists of a parallel composition of actions and subgoals, the method `constructParallelPlan` is used to create a parallel composition of actions and subgoals in the plan. As with the method `constructPlan`, it takes 4 parameters as its arguments, which are the target plan (`pl`), the depth of the target plan in the goal-plan tree (`currentdepth`), the set of literals which can be selected as the precondition of plans to achieve subgoals in the target plan (`var_pre`) and the set of literals which can be selected as the postcondition of the actions in the target plan (`var_post`). The parameter `pl` is the plan whose plan body will be the new parallel composition, and the other three parameters are passed by the method `constructGoal`. If `pl` is a leaf plan, the parallel composition contains only $\#\alpha$ actions. Otherwise, the parallel composition contains $\#\alpha$ actions and $\#\gamma$ subgoals. Each action in the parallel composition has the plan's precondition as its precondition, and its postcondition, l, is randomly selected from `var_post`. Once a postcondition l is selected, l and its negation are removed from `var_post`, and thus cannot be used as the postcondition of all other actions in the same

parallel composition. By doing so, we ensure the postcondition of actions in one parallel composition do not conflict with each other, and there are no identical actions in the parallel composition (i.e., actions which have same precondition and postcondition). For the subgoals in the parallel composition, we divide the set of literals that can be selected as the postcondition of the actions in the plan(s) for a subgoal into $\#\gamma$ subsets. Each subset contains the same number of distinct variables, and is denoted as $\texttt{var\_post}_i$ where $0 \leq i \leq \#\gamma - 1$. For each subgoal in the parallel composition, the method `constructGoal` is called to add plans to achieve the goal. The parameter `gl` of `constructGoal` is the subgoal for which plans must be generated, the parameter `currentdepth` is incremented to be the current depth of the plan containing the parallel composition plus 1, the parameter `var_pre` is set to contain only the precondition of the plan containing the subgoal gl (i.e., whose plan body is the parallel composition containing gl), and finally the parameter `var_post` is set to be $\texttt{var\_post}_i$ if `gl` is the $i + 1^{th}$ subgoal in the parallel composition. The whole process of generating parallel composition ensures that there are no potential conflict between these actions and subgoals and the overall postcondition of the plan is "stable".

We keep creating actions, plans, (sub)goals and parallel compositions 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.