SAPIENZA
UNIVERSITÀ DI ROMA

PROJECT PLANNING AND REASONING

Syntax checker for PDDL

Edoardo Montecchiani 1969618

1. Introduction

The following pages will explain the features and functionalities of a syntax checker for PDDL, which thus makes it possible to analyze domain files and PDDL problems and understand if and what syntactic errors have been made. In addition to this, this program is able to make minor modifications to the PDDL domain, enabling it to be optimize. In particular, it will be able to merge two actions that would always be executed consecutively anyway, or eliminate actions that could never be executed.
The program is entirely written in python and does not require any special external libraries to be installed. It has a simple command-line interface and it is possible to choose whether or not to use the optimizer, and for each proposed change you can choose whether or not to apply it. Another PDDL domain file containing the modified domain will then be generated.

2. Execution

To start the program, simply run the file 'syntaxChecker.py' in the root directory from the command prompt. The PDDL domain path must be added to the command as an argument "-d" or "--domain". As an optional argument, the PDDL problem path can also be added using "-p" or "--problem". Basically, program execution involves this string on the command line:

python checker.py -d DOMAIN_PATH -p PROBLEM_PATH

3. Structure

Figure 1 shows the entire structure of this project, which includes several *'.py'* files for better readability and to facilitate new implementations. The main directory contains the main file that starts the program *'syntaxChecker.py'*, a README.md explaining how to use the program, and some management files for GitHub repositories. It also contains 3 other directories:

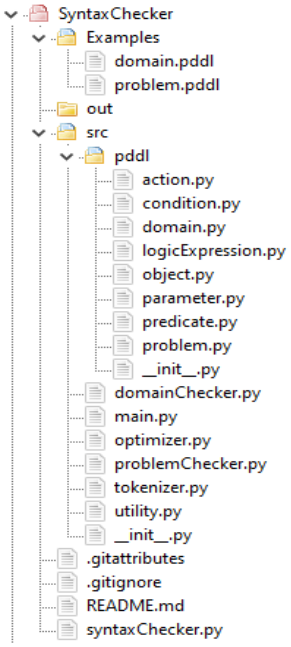

Figure 1. Directory tree

- *'Examples'* where some examples of PDDL domains and problems are present and where new ones can be added so that you do not have to use the full path but only the filename to find it;
- *'out'* which contain the domains modified by the program optimizer;
- *'src'* which contain all the *'.py'* code files for running the program.

'SyntaxChecker.py' analyses the arguments passed as parameters and simply calls the main function in the main.py file. The latter is responsible for calling the domainChecker and problemChecker in the

'.py' files of the same name and the optimization algorithms in the *'optimizer.py'* file.
Within the subfolder 'pddl' are all the classes that are useful for modelling the domain and problem PDDL as written in the next sections.

4. Classes

The Syntax Checker does not just check the syntax and report any errors by highlighting the wrong word and line, but reconstructs the entire domain and PDDL problem in Python classes. This was in fact necessary for the optimization phase, which we will see in the next chapter. In order to optimize, it is necessary to be able to analyze all actions and their preconditions and effects quickly, and this is possible in the program if the actions, preconditions and effects are efficiently stored in objects of object-oriented programming

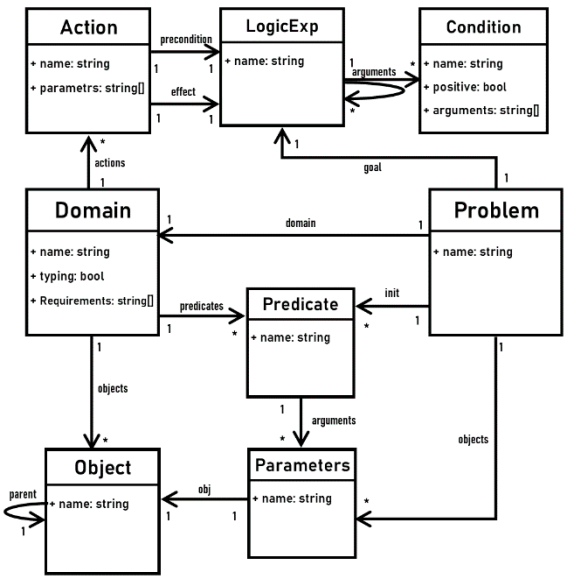

Figure 2. Class diagram

In Figure 2, we can see the class diagram in UML. These classes provide all the information of a domain and a PDDL problem. Some classes are intuitive in that they have the same name in PDDL as:

- Domain: This is the main class that contains the information of the entire domain and thus the domain name, requirements (saved as strings), types (see objects), predicates and actions. In addition to this, there is also a boolean variable (typing) which is true if there is typing in the requirements.
- Problem: As with Domain, Problem contains all the information of the problem and thus the name of the problem, the domain with which it is associated, the objects, the init which is a list of predicates and the goal corresponding to a logical expression.
- Predicate: A predicate has a name and as arguments a list of parameters (variables and so beginning with '?' if they are possible predicates in the domain or problem objects if they are in the problem init)
- Action: In addition to the name, they have parameters (list of strings) and preconditions and effects saved as logical expressions.

In addition to these classes, there are others that may be less immediate as they have different names from PDDL but are useful for the purpose of building the domain in a hierarchical manner.

- Condition: We know that the preconditions and effects of actions and the goal of the problem contain several predicates that can also be anticipated by a *'not'* denying it. To differentiate these from the actual predicates in the init they will be called Condition and will have a name (the same name as the predicate), a boolean indicating whether it is false that is preceded by a not, and a list of strings as an argument to indicate the variables.
- LogicExp: We have said that the preconditions and effects of actions and the goal of the problem are logical expressions. That is, there is always a logic keyword such as 'and' or 'or' (as names) followed by (as arguments) a list of Condition or other logical expressions.
- Parameters: They are the predicate variables or objects of the problem. They only have the name in case there are no types. Otherwise, each parameter will have a type, called Object.
- Object: These are exactly the types. If provided for in the requirements, each param-

eter will have a type and they can be inserted hierarchically. In fact, an Object may have a parent.

The latter accept different logic keywords depending on the requirements. Below is a list of all supported logic keywords:

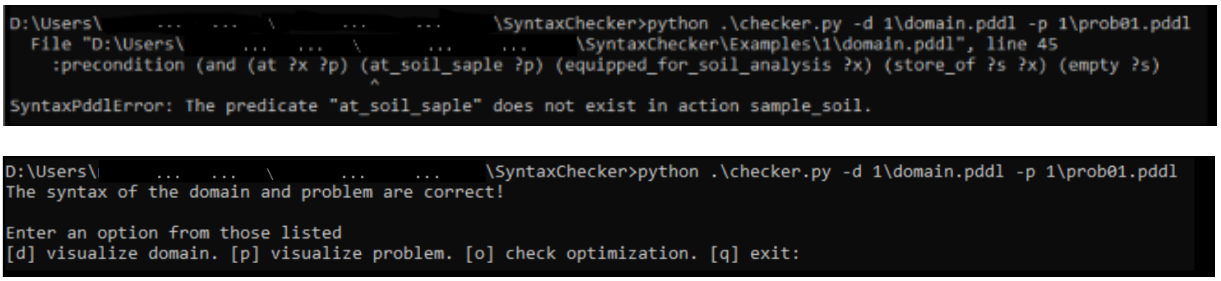

Figure 3. Above is an execution with an error in the domain. The error is that a predicate that does not exist within the preconditions of an action was used. Below an execution in which the domain and problem are correct. You will be asked whether you want to display the domain or the problem or proceed with optimization.

5. Features

This section lists all functionalities and compatibilities of this software. PDDL is in fact a fairly simple language, but it has received several modifications and additions over time. It is therefore very difficult to find a planner who considers the many features added to PDDL over time, many of which are rarely used. The features compatible with this syntax checker are all the major ones in PDDL 1.2[1]. In the details here is a list of all the requirements that are supported:

- Strips;
- Typing;
- Disjunctive-preconditions;
- Equality;
- Existential-preconditions;
- Universal-preconditions;
- Quantified-preconditions;
- Conditional-effects;
- Adl.

Obviously, it is possible to define types and it is necessary to define predicates and actions. As far as actions are concerned, they must necessarily be defined with parameters, preconditions and effects.

- And;
- Or;
- Imply;
- Not;
- Forall;
- When;
- Exists.

This program is designed as a syntactic checker for most classical planning problems. An important feature is that if the checker is successful and thus says that the syntax is correct, it will be correct (bugs aside). Conversely, an error detected by the checker could simply be an unsupported PDDL feature.

In figure 3 we can see two executions. For the top one, we used a domain that had an error, specifically a predicate misspelled: "*at_soil_saple*" instead of "*at_soil_sample*". As you can see, the error interface is similar to Python's console interface (we took a cue from that). A first line tells what file and what line the error is in, then the entire line is printed out and highlighted with a '^' below the errata word. Finally, it is written what the error is. This method makes it possible to highlight the error immediately and proceed with the correction. In figure 3 below, a correct domain and problem is

[1] PDDL 1.2 https://planning.wiki/ref/pddl Accessed: 2022-06-14.

used instead. As can be seen, the user is asked to input one of the proposed options, namely:

- 'd' (but also 'domain') in order to display the domain on the console;
- p' (or 'problem') to be able to display the problem on the console;
- 'o' (or 'optimize') to start the search procedure for possible optimization;
- q' (or 'quit') to exit the program.

If 'o' is chosen, the optimization algorithms described in the following section will start and for each thing that can be optimized, the user will be asked whether he wants to proceed with the change. If the user accepts even one change, a new file will be generated in the out folder containing the new optimized, correctly written and functioning domain.

6. Optimization

Thanks to the domain model and PDDL problem described in the section 4, it was possible to implement optimization algorithms to streamline the domain. Optimization processes are relative to the problem, which is why using the syntax checker only with the domain without a problem will not allow the optimizer to be used.
Optimization involves two actions:

- **Merge** two actions that relative to the current problem cannot be performed individually but only and always one consecutively to the other;
- **Eliminate** an action that relative to the current problem cannot or would not make sense to execute.

6.1 Merge two actions

In order for two actions *a1* and *a2* to be joined, the following conditions must be fulfilled:

- The effects of *a1* contain all the preconditions of *a2* this guarantees that by executing *a1* it will always be possible to also execute *a2*.
- No other action in the domain has as precondition even a condition present in the effects of *a1*, this guarantees that after executing *a1* one can only execute *a2* and no other action.
- There exists in the preconditions of *a2* some condition not present in any of the effects of the other actions in the domain or in the *init* of the problem. This guarantees that *a2* can only be executed after the action *a1*.
- There is no condition within the problem goal that is present in the effects of *a1*

If all these conditions are fulfilled, then it will mean that in an eventual plan, action *a1* and *a2*, if they are to be executed, will always and only be executed consecutively. For this reason, they can be merged into a new action *a1-a2* that has as preconditions the preconditions of *a1* and as effects the sum of the effects of *a1* and *a2*. Obviously by the sum of the effects is meant the concatenation of the two, taking care, however, that the conditions concerning the same predicate but one positive and one negative, in which case only the one present in *a2* is considered.
The merge functionality can be very useful for increasing the level of abstraction. In real problems, it is often the case that one initially thinks of low-level actions for planning purposes. Let us think of a robot that can pick up objects and transport them somewhere else, we decide to create an action that is responsible for putting the robot in a position to pick up an object and therefore its only effect is to true the predicate *'object_in_position_to_grab'* and another action only performs the actual grabbing if the *'object_in_position_to_grab'* predicate is true. This way of thinking is low-level and the two actions can be merged into one more high-level *'pick'* action.

6.2 Delete an action

An action *a* can be eliminated if, given a problem, an eventual plan for this problem certainly does not involve executing *a*. There are two reasons why we are sure that an action will not be performed:

1. There is no point in performing the action. This occurs if the conditions in the effects of *a* do not appear in any of the preconditions of the other actions nor in the goal. It only takes one condition effect of *a* present

in the precondition of another action to no longer consider the elimination of *a*.

2 The preconditions of *a* can never be reached. This happens if some condition exists in the preconditions of *a* that is not present in any of the effects of the other actions in the domain or in the *init* of the problem.

This feature is very useful because beyond the simplification of the domain, it can help one understand why sometimes planning fails unexpectedly. In fact, if the optimizer wants to eliminate an action, it will mean that that action can never be executed, and this may also mean an error in the design of the domain. In addition to this, if very complex domains are used, it may be useful to thin out the actions that are not needed for that problem. In this way, the PDDL domain gains in readability.