University of St.Gallen

Institute of Computer Science

Web-based Autonomous Systems, FS2023
Danai Vachtsevanou, Andrei Ciortea
{firstname}.{lastname}@unisg.ch

Exercise 2: Automated Planning

Deadline: **March 7, 2023; 23:59 CET**

In this exercise, you will gain experience with how agents can reason about action through classical planning. You will:

1.) use the Planning Domain Definition Language (PDDL) 1.2 to model a planning domain of interest;

2.) implement parts of a STRIPS-like automated planner;

3.) apply your planner to your PDDL model to reason about action in the domain of interest;

4.) explore how your planner addresses some of the limitations of a classical STRIPS planner.

① Task 1 (3.5 points): PDDL-based Scheduling and Planning

Consider this problem: Many *inhabitants* move into a newly constructed building that has 10 *rooms*. The inhabitants are starting to arrive today and it is known in advance which *room* each inhabitant will occupy (e.g. `room1`), and at which *time* an inhabitant can arrive. An inhabitant may arrive at one or more specific times within the day (e.g. at 9 a.m. and 10 a.m.). When an inhabitant arrives, a *maintenance worker* needs to *set up* the inhabitant to their room by showing them to the room.

Task 1.1 (2 points): Define the PDDL Domain and Problem Your first task is to model the presented planning domain in PDDL and to create an instance of a planning problem in the domain. Before you start, consider the following questions: What aspects are part of the *planning domain*? What aspects are part of a *planning problem* in that domain?

In this task, you can consider that for setting up an inhabitant only the action `showRoom` is required, e.g.: `<showRoom am9 inhabitant1 room1>`, where `showRoom` would be the ground action applied to the instances `am9`, `inhabitant1`, and `room1`. The maintenance worker should schedule the appointments such that (i) they do not overlap with each other, and (ii) all inhabitants are set up by the end of the day. For example, if `inhabitant1` can arrive at 9 a.m. and at 9:15 a.m. to be set up in `room1`, and `inhabitant2` can arrive at 9 a.m. to be set up in `room2`, the maintenance worker should schedule the appointment with `inhabitant1` at 9:15 a.m. and the appointment with `inhabitant2` at 9 a.m. If both inhabitants can only arrive only at 9 a.m., the problem cannot be solved.

Define the PDDL domain and problem, so that the maintenance worker schedules the appointments with all the inhabitants. Use the online PDDL editor available at Planning.Domains[1] and consult the documentation of PDDL 1.2[2] to 1) define the domain and problem, and 2) solve the problem in the domain.

[1]Online PDDL editor: http://editor.planning.domains/
[2]Documentation of PDDL 1.2: https://planning.wiki/ref/pddl

Task 1.2 (1.5 points): Extend the PDDL Domain Apart from resolving the scheduling presented in Task 1.1, now the maintenance worker needs also to plan their series of actions for setting up an inhabitant to a room: For setting up the inhabitant, the maintenance worker needs to `be at` the correct room at the time of the appointment. Additionally, for entering the room, the maintenance worker needs to first `unlock` the room. As a result, it is expected that for setting up an inhabitant, three sequential actions are required (e.g., `<unlock am9 room1>`, `<join am9 room1>`, `<showRoom am9 inhabitant1 room1>`).

Extend the PDDL domain so that the maintenance worker schedules all the appointments and plans the actions needed for each appointment.

② Task 2 (3.5 points): Working with a State-space Planner

In this task, you will complete a STRIPS-like planner and apply it to solve planning problems in the domain defined in Task 1. To help you focus on the central aspects of the planner, we provide you with an almost complete implementation of the planner. The exercise template is available on GitHub[3], and it is based on the `pyperplan` planner[4].

To complete this implementation, we want you to focus on two aspects of the planner: (1) the internal representation of planning tasks and operators (these are the abstractions that adopt the STRIPS representation), and (2) the A* search algorithm used by the planner for solving tasks. Visit the exercise template's README[5] for more details about the source files you need to update for this task. Follow the pointers so that you get an overview of how the planner works overall.

First, complete the implementation of `task.py`, which includes two classes (see README): `Operator` and `Task`. Study the class `Operator` to see how this planner should handle the removal and addition of predicates. To complete the implementation of the `Operator` class, you need to implement the methods `applicable()` and `apply()`. Then, complete the implementation of the `Task` class by implementing the methods `get_successor_states()` and `goal_reached`().

Second, complete the implementation of `a_star.py` (see README). Study the class `Task` to see how the search algorithm should handle a STRIPS planning task. Study the class `SearchNode` to see how the search algorithm should handle the nodes of the search space. Then, complete the implementation of `astar_search()` by implementing the A* search algorithm. For an easy-to-follow introduction to the A* algorithm, we recommend the online article in the footnote[6].

Note: This exercise template is based on an open-source implementation of a STRIPS-like planner and we made no efforts to obfuscate the connection to this implementation. It is therefore not surprising that you *could* simply copy the code from that implementation. However, we ask you to implement the omitted aspects of the planner yourselves instead.

③ Task 3 (3 points) Exploring the Sussman Anomaly

In this task, you will study a planning domain and problem susceptible to Sussman Anomaly. The problem requires for a maintenance worker to plan their actions towards cleaning a room. See the exercise template's README for more details on how to access and study the domain and problem, and then reply to the questions that follow in a short report (max. 2 pages of text, the more concise the better).

1.) What is the anomaly that occurs when solving the problem in the domain?
2.) Under what circumstances does the anomaly occur?
3.) What specifically in the problem and the domain make it susceptible?
4.) Why is the behavior not observable with your planner implementation from Task 2?

[3]Exercise template: https://github.com/HSG-WAS-SS23/exercise-2
[4]Pyperplan implementation: https://github.com/aibasel/pyperplan/
[5]Exercise README: https://github.com/HSG-WAS-SS23/exercise-2/blob/main/README.md
[6]Intoduction to the A* algorithm: https://www.redblobgames.com/pathfinding/a-star/introduction.html

④ Hand-in Instructions

By the deadline, hand in via Canvas a **zipfile** that contains:

1.) your **PDDL documents for Task 1**;

2.) a PDF file that includes the link to the GitHub fork containing your **code for Task 2**, and your **report for Task 3**;

3.) any additional instructions for how to run your code (if non-obvious).