Autonomous Systems

H. Geffner — Universitat Pompeu Fabra
A. Occhipinti (exercise description by G. Francès) — 2021-2022 Term 2

Exercise Sheet C

Due: March 9, 2022

Some of the files required for this exercise are in the directory `lab3` of the course repository (`https://github.com/AOccLib/autonomous-systems-21-22`). You can always update your clone of the repository with `git pull` to see new files.

In this assignment we'll work with classical planning languages and solvers. The objective is to get to grips with modeling deterministic planning problems and solving them effectively with some classical planner, the way we did the same with SAT solvers on the previous assignment.

Modeling. The standard language in the classical planning community is the Planning Domain Definition Language (PDDL). Other than what you see in the lecture, you can also have a look at this tutorial on PDDL modeling[1] by Patrik Haslum, or this other PDDL tutorial[2] by Malte Helmert. Be warned though that PDDL is less of a "standard" than e.g. the DIMACS cnf format you saw in the last assignment. Not all planners support all of the language features, and even when they do, not all solution techniques are well defined for all features.

Solvers. Once you have your problem specified as a PDDL problem, you should be able to use any classical planner to get a solution. There are several planners that periodically enter into the International Planning Competition (IPC) and that are open source. The last IPC edition was in 2018, and the source code of all participating planners is online.[3] We encourage you to install Fast Downward,[4] a planner which should be easy to install and has a fairly large array of available solution techniques.

Exercise C.1 (Monkey and Bananas: 1.5 points)

This one is a very simple problem (and a bit of a classic![5]), but will be useful as a warm-up. A monkey is alone in a room, and the room has some bananas that hang from the ceiling. The monkey cannot reach the bananas, but in some far corner of the room there is also a chair that the monkey could use to reach them. Model some instance of this simple problem in PDDL, where the goal should be that the monkey has eaten the bananas, and solve it with your planner of choice. Though this is one single problem, most planners will still expect you to provide two files: a "domain" file, and an "instance" file. The domain file typically has the definition of the *logical vocabulary* you will use to define the problem (e.g. predicates, and perhaps types), along with the definitions of the possible *actions*. The instance file will have a definition of the particular *initial situation* and *goal formula* of your problem.
Don't overthink this one, it's just a very simple exercise. You just need to think on what are a reasonable set of actions to model the problem, and what predicates you'll need to define their structure. There are of course countless variations you can model (one banana? More than one? Does the monkey grab the bananas with one action and then eat them with another, or all in one action? How many bananas can the monkey grab at the same time?). The choices you make to resolve these questions are not important, the only important thing for this exercise is that you model something that makes sense, and that some planner can process that and give you a plan that gets the monkey to eat some bananas.

[1] `http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html`.
[2] `http://www.cs.toronto.edu/~sheila/2542/w09/A1/introtopddl2.pdf`.
[3] `https://ipc2018-classical.bitbucket.io/#planners`.
[4] `https://www.fast-downward.org/ObtainingAndRunningFastDownward`.
[5] `https://en.wikipedia.org/wiki/Monkey_and_banana_problem`.

For this exercise you need to submit one paragraph describing your model, plus the actual pddl domain and instance files.

Exercise C.2 (Teleporting Sokoban: 2.5 + 3.5 + 2.5 points)

In the previous lab assignment we saw how to leverage the power of SAT solvers to solve other NP-hard problems such as Sudoku. Other interesting problems and games such as Sokoban,[6] however, are likely too hard for that.[7] Here you are asked to model in PDDL a variation of the standard Sokoban problem. In the standard Sokoban, an agent moves around a warehouse and needs to arrange certain boxes into their final, fixed locations. Quoting the Wikipedia description:

> *The player is confined to the board, and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The number of boxes is equal to the number of storage locations. The puzzle is solved when all boxes are at storage locations.*

(a) Describe how would you model in PDDL a variation of the standard Sokoban where the agent can *teleport* herself to any *empty* location of the grid, but can do that *only once*. Teleporting into a wall or in the same location as a box is not allowed; teleporting into a goal location is allowed. What predicates are you going to need to encode grid cells, wall, player and box positions, and also be able to describe the goal situation with a logical formula?

Submit a brief description of the PDDL predicates, objects, initial state, actions and goal of your planning encoding.

(b) Sokoban games are often encoded in a readable text format, where certain characters are used to denote wall, box, player, goal positions, etc., as described in `http://sokobano.de/wiki/index.php?title=Level_format`. We here ask you to program a Sokoban solver that transforms any command-line given Sokoban problem into a PDDL problem, according to your description above. Then, your solver feeds the PDDL into some classical planner, obtains a solution, and "transforms" the solution into some readable output describing the agent moves, e.g. (this solution is likely not correct!):

```
$./sokoban.py -i benchmarks/sasquatch/level5.sok
Solution:
Move up
Move up
Move right
Teleport to position (3, 5)
Move down
...
```

You should be able to define the problem dynamics in one single *domain* file that does not depend on the particular Sokoban problem P that you receive on the command line. Then, you'll only need to generate a single *instance* PDDL file based on P (that is the purpose of having separate domain and instance files, after all).

There are some Sokoban instances in the `lab3/benchmarks/sasquatch` directory of the course repo, which you can use to test your solver.[8] There is also some simple `sokoban.py` code to get you started — this is only to show how to parse the input string into an intermediate representation that you can use to generate the PDDL. Of course, you can also ignore this code and write yours from scratch.

---

[6] `https://en.wikipedia.org/wiki/Sokoban`.

[7] Joseph Culberson. 1997. Sokoban is PSPACE-complete. *University of Alberta, TR97-02*. `https://doi.org/10.7939/R3JM23K33`.

[8] These make up the "Sasquatch" level collection by David W. Skinner, that I have downloaded from `http://sokobano.de`.

For this section you need to submit the source code of your program, along with any static files (such as the domain PDDL file mentioned above) that are necessary to solve arbitrary Sokoban instances.

(c) Planners such as Fast Downward usually provide several algorithms and domain-independent heuristics that can be used to tackle any PDDL problem. You will discuss some of these in class. Different techniques might work better or worse on certain problems. Quite importantly, some of these techniques are *optimal*, in the sense that they guarantee that the solution found is optimal, and some of them are *satisficing*: they only return a plan, but with no guarantees.

Write a small script (or do that manually if you prefer) that runs your Sokoban solver on all of the 50 problems in `lab3/benchmarks/sasquatch` for a maximum of 1 minute CPU time, using at least one satisficing technique, and at least one optimal technique. How many of the problems can you solve in each case? How much do the number of steps in the non-optimal solutions differ from the optimal ones? Are the Sasquatch problems too easy or too difficult for your compilation-to-PDDL approach? If they are too easy, what do you think would make them more difficult? If they are too difficult, what could make them easier to solve?

You can run `./fast-downward.py -h` or check the planner website (e.g. `http://www.fast-downward.org/PlannerUsage`, `http://www.fast-downward.org/IpcPlanners`) in order to find some examples on how to run Fast Downward. Use option `--overall-time-limit 60` to limit the runtime to one minute. Use option `--plan-file` to tell Fast Downward in what file you want the solution plan, if any, to be printed. For instance, if you want to run the Fast Downward implementation of the LAMA planner, that uses a greedy best-first search with the landmark and FF heuristics, use:

```
./fast-downward.py --overall-time-limit 60 \
    --alias seq-sat-lama-2011 \
    --plan-file myplan.txt  \
    domain.pddl instance.pddl
```

For this section, submit a short report with your findings and answers to the questions above.

The exercise sheets should be submitted in groups of two or three students. Please submit one single copy of the exercises per group (only one member of the group does the submission), and provide all student names on the submission.