Automated Planning (TDDD48)

Jendrik Seipp
Mika Skjelnes

Linköping University

Lab 1

Important: For submission, consult the rules at the end of this document. Non-adherence to these rules might lead to a penalty in the form of a deduction of points. Some points are *bonus points*. These can help you reach the point quota per lab (4/12 points) and the overall point quota ($50\% \cdot 7 \cdot 12 = 42$ points).

Exercise 1.1 (Setting Up Fast Downward, 1 point)

The *Fast Downward* planning system is a tool that we use frequently for demos in the lecture and for the labs. Your task in this exercise is to get access to the course repository as well as to install the planner. We describe two ways of setting up your system in the following: we start with the recommended way that uses *Vagrant* and *VirtualBox*, which should be possible on all Intel and AMD architectures. Afterwards, we point to relevant repositories and installation instructions in case you want to or have to set up your system manually. Please follow the instructions for **one** of these two ways.

Installation with *Vagrant* and *VirtualBox* Start by installing *Vagrant* and *VirtualBox* by following the installation instructions for your operating system at `https://www.vagrantup.com` and `https://www.virtualbox.org`, respectively. If your operator system is Ubuntu 22.04, you can install both tools by running the following commands in a console:[1]

```
sudo apt update
sudo apt install virtualbox vagrant
```
(for Ubuntu 22.04)

With both tools installed, you can set up the virtual machine that runs the *Fast Downward* planner:

(a) Download the Vagrant configuration file (`Vagrantfile`) from the course website.

(b) Move the downloaded file to an *empty* directory. Make sure that your operating system didn't add a (possibly hidden) file extension (we have seen this happen frequently on Windows).

(c) Open a console in that directory and execute `vagrant up` (this may take a while).[2]

Over the course of the semester, you'll have to interact with the virtual machine set up with *Vagrant* repeatedly. Here are the most important commands to do so:

- `vagrant up` to start the virtual machine (after the first time, this won't take as long)
- `vagrant halt` to stop the virtual machine
- `vagrant ssh` to connect to the virtual machine
- `exit` to disconnect from the virtual machine

You have now set up the virtual machine, cloned all relevant repositories and installed required packages and tools. You have not yet compiled the Fast Downward planner that is used for this exercise, though. To do so, connect to the virtual machine (with `vagrant ssh`), then

[1]At the time of writing, Ubuntu 22.04 provides Vagrant 2.2.19 and VirtualBox 6.1.32, which are compatible. However, Vagrant 2.2.19 is not compatible with VirtualBox 7. If you have such a newer version of VirtualBox installed, we recommend installing Vagrant $\geq$ 2.3.2 manually via the Vagrant download page.

[2]There is some error output for compiling VAL, which you can ignore.

(a) change to the directory with the Fast Downward version used for this lab with

```
cd /vagrant/tddd48/lab1/fast-downward
```

(b) compile the planner with `./build.py`.

(c) run the planner on an example task with

```
./fast-downward.py ../gripper/domain.pddl ../gripper/problem.pddl \
        --heuristic "h=ff()" --search "eager_greedy([h])"
```

Congratulations, you have successfully set up your system for the labs!

Manual Installation An alternative (which we do **not** recommend) is to install everything that is required manually. Start by cloning the following repositories:

- the repository of the course at `https://github.com/mrlab-ai/tddd48-labs`
- the plan validator `VAL` at `https://github.com/KCL-Planning/VAL`

You can find the individual steps that are required to install both plan validators in the Vagrantfile. To give you an idea of what is happening in the Vagrantfile, please note that

- we are not using the latest version of `VAL`
- the `VAL` revision we use raises warnings during compilation that are treated as errors
- the produced binaries are moved to a folder on `PATH` (such that they can be executed from anywhere without providing a path to the binary)

If you manage to successfully install the validator, you still need to compile the Fast Downward planner that is used for this lab:

(a) Change to the directory with the *Fast Downward* version used for this lab. You can find it in the directory `lab1/fast-downward` of the repository of the course.

(b) Follow the instructions on how to compile *Fast Downward* at `https://github.com/aibasel/downward/blob/main/BUILD.md`.

When you have successfully compiled *Fast Downward*, run the planner on an example task with

```
./fast-downward.py ../gripper/domain.pddl ../gripper/problem.pddl \
        --heuristic "h=ff()" --search "eager_greedy([h])"
```

Exercise 1.2 (Solving Tasks with Fast Downward, 2+1 points)

In this exercise, you'll play around with the Fast Downward planner. The files required for this exercise are in the directory `lab1` of the course repository (`/vagrant/tddd48` in your course VM). Update your clone of the repository with `git pull` to see the files.
For this exercise, set a time limit of 1 minute and a memory limit of 2 GB. Using Linux, such limits can be set with `ulimit -t 60` and `ulimit -v 2000000`, respectively.
The directory `lab1/tile` contains four variants of the 15-Puzzle:

- original formulation (`puzzle.pddl`, `puzzle01.pddl`)
- variant with weighted tiles (`weight.pddl`, `weight01.pddl`)
- variant with glued tiles (`glued.pddl`, `glued01.pddl`)
- variant with cheating action (`cheat.pddl`, `cheat01.pddl`)

To run Fast Downward, use the script `fast-downward.py` with the corresponding domain and problem files, specifying the search algorithm and the heuristic. Example for the original formulation, greedy best-first search and the FF heuristic:

```
./fast-downward/fast-downward.py tile/puzzle.pddl tile/puzzle01.pddl \
     --heuristic "h=ff()" --search "eager_greedy([h])"
```

(a) Run Fast Downward on the original formulation of the 15-puzzle, using greedy best-first search and different heuristics:

- additive heuristic: `add()`
- blind heuristic: `blind()`
- causal graph heuristic: `cg()`
- FF heuristic: `ff()`

Summarize your results with respect to time (`Planner time`), number of expanded and generated states (`Expanded` and `Generated`), and solution quality (`Plan cost`) in a table.

(b) Run Fast Downward again to solve the other variants of the domain. Discuss in 2–3 sentences how the results differ from those for the original formulation.

Hint: You do not need to report explicit numbers as in part (a).

Exercise 1.3 (Modeling Package Delivery in PDDL, 4 points)
Consider the following package delivery problem:

- There is a set of *cities*, *trucks*, and *packages*.
- The cities are *connected* by a road network.
- A truck can *move* from one city to another along the roads.
- A package can be *loaded* into and *unloaded* from a truck.
- Every package has a *target location* where it should be delivered.

Model this domain in PDDL. Then model at least two different instances (e.g., different road networks, different target locations, different number of trucks) and find plans for them with Fast Downward. Submit both the PDDL files and the plans you found.
We recommend to use VS Code for this exercise with the PDDL extension (`https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl`). It reports common modeling errors.
For an introduction to PDDL, see `https://www.ida.liu.se/~TDDD48/labs/2023/pddl.en.shtml`.
For the complete PDDL syntax, see `https://helios.hud.ac.uk/scommv/IPC-14/repository/kovacs-pddl-3.1-2011.pdf`.

Exercise 1.4 (2+2 points)

Consider the following transition system:

$$\begin{tikzcd}
& & A \arrow[rr, "\ell_2"] & & B \arrow[dr, "\ell_1"] & & C \arrow[dr, bend right, "\ell_1"'] \arrow[dr, bend left, "\ell_0"] & \\
{} \arrow[r] & D \arrow[ur, bend left, "\ell_1"] \arrow[rr, "\ell_3"] \arrow[dr, bend left, "\ell_0"] & & E \arrow[loop above, "\ell_1"] \arrow[rr, "\ell_2"] \arrow[dr, "\ell_0"] & & F & & G \arrow[dlll, bend left, "\ell_1"] \\
& & H \arrow[ul, bend left, "\ell_2"] \arrow[rr, "\ell_1"] & & I & & &
\end{tikzcd}$$

$$c(\ell_0) = 0, c(\ell_1) = 1, c(\ell_2) = 2, c(\ell_3) = 3$$

(a) Formalize the depicted transition system as a 6-tuple $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$.

Hint: When writing down something formally, always ask yourself what kind of mathematical object you are using where, then use correct notation. For instance, a tuple such as $\mathcal{T}$ is formalized using angle $\langle \cdots \rangle$ or round $(\cdots)$ brackets; a set such as S needs braces $\{\cdots\}$; a function such as $c\colon L \to \mathbb{R}_0^+$ describes how objects in L are mapped to an object in $\mathbb{R}_0^+$; and a single object such as s_0 uses no brackets at all.

(b) Answer the following questions about the depicted transition system.

i) Is $\mathcal{T}$ deterministic? Justify your answer.

ii) Which states of $\mathcal{T}$ are *unreachable* from state H?

iii) What are the *predecessors* of state E in $\mathcal{T}$?

iv) What is the cheapest *solution* of $\mathcal{T}$?

Bonus Exercise 1.5 (1+1 **bonus** points)

Consider the following formula over propositions $\{X, Y, Z\}$:

$$\varphi = ((X \land \neg Y) \lor (\neg Z \land \neg(X \lor Y)))$$

(a) Transform φ into an equivalent formula that is in conjunctive normal form (CNF). Provide your transformation step by step so that it is easy to verify equivalence from one step to the next.

Hint: If you need a reminder of equivalence rules, you may consider the first propositional logic lecture from the TDDC17 course: `https://www.ida.liu.se/~TDDC17/`,

(b) Provide two interpretations $\mathcal{I}$ and $\mathcal{J}$ of propositions $\{X, Y, Z\}$ such that $\mathcal{I} \models \varphi$ and $\mathcal{J} \not\models \varphi$.

Submission rules:

- Lab sheets must be submitted in groups of 2–3 students. Clone the labs repo (`https://github.com/mrlab-ai/tddd48-labs`) and push it to a repo at the University GitLab instance `https://gitlab.liu.se`. Make sure the repo is **private** and give read access to Mika Skjelnes (`mika.skjelnes@liu.se`).
- For non-programming exercises, create a single PDF file at the location `labX/solution.pdf`. If you want to submit handwritten solutions, include their scans in the single PDF. Make sure it is in a reasonable resolution so that it is readable. Put the names of all group members on top of the first page. Either use page numbers on all pages or put your names on each page. Make sure your PDF has size A4 (fits the page size if printed on A4).
- For programming exercises, directly edit the code in the cloned repository and only create those code text file(s) required by the lab. Put your names in a comment on top of each file. Make sure your code compiles and test it. Code that does not compile or which we cannot successfully execute will not be graded.