Tea at home - report

1. Choice of planner

Since the given task asks for the usage of a HTN-Planner, we choose a implementation of SHOP. This planner is widely used, which implies easy usage and good performance. Also SHOP is sound and complete. First we try to work with JSHOP, a Java-based implementation. While trying to get a working version, we run into problems compiling and running JSHOP under Windows / Linux. Classpath errors and a readme mentioning nonexistent files are additional problems. Also the precompiled version from sourceforge does not work. Because of this, we decide to use Pyhop instead, a Python-based implementation of SHOP.
In contrast Pyhop is easy to set up: Only Python 2.7 is required, the planner runs out of the box. It consists of roughly 150 lines of code, so it can be read and understand easily. Examples with known domains (travel-problem, blocksworld) are included, which eases the understanding of how to encode own domains. The syntax for methods / operators in Pyhop consists of functions, so with a little bit of programming and HTN planner knowledge it is quite easy to handle. This allows the usage of regular python functions inside operators and methods, like printing out text. This means, anyone who is using Pyhop does not need to learn a special syntax for domain encoding. There are no logical propositions, instead a state object consisting of variables describes the state of the domain. For example, a variable *state.loc['kettle'] = Location.shelf1* represents an description of a part of the domain, which is easy to read and understand. The state object during runtime is implicitly handed over between functions.
The fact that python is often used in artificial intelligence, knowledge discovery and robotics is also a good reason to use a python based planner.

2. Challenges installing/ running Pyhop

Installation instruction

1) Download and install Python 2.7.11
2) Git clone the repository from https://github.com/dpyka/TeaAtHomeHTN or download the content as zip file and extract it somewhere
3) Download and set up the enumeration library for your python environment https://pypi.python.org/pypi/enum34 (Note: There are quite a lot of different enumeration libraries for python, make sure you have exactly this one)
4) In the python console interpreter, switch to the directory in which you have cloned the repository using the commands
   import os
   os.chdir('/path/to/teaathome')
5) For each test, we provide an executable file named test#.py (# stands for 1 to 3). You can run them using the command
   execfile('test1.py')
   execfile('test2.py')
   execfile('test3.py')
   They will set up a fixed environment for each test and call the planner.
   The resulting log file is stored in the "logs" subfolder, log files are named test#.log
6) The planner file pyhop and the modeled domain file teaathome.py are not designed for direct execution. They are used in the tests.

7) There are several other test files with a slightly different environment called test#alt#.py. They are used for testing the domain model in slightly changed environments.

At the beginning, the declaration of tasks and methods is quite confusing. In our first attempt, the planner searches the left side of each branch. Because methods are used as alternative ways to solve a task, it resolves only this very first method, it does not visit the other possible methods if it does not fail. Restructuring tasks and methods solves this error.

We do not set up an entire IDE to work with Python. Instead, we only use the Python interpreter to test our domain. This leads to some effects related imported resources during runtime. It is necessary to reload the interpreter after certain changes to the teaathome domain. Otherwise some of these changes do not take effect, probably due to old present global variables. Notepad++ is used for writing the code.

Pyhop stores its world observation into simple variables. We use a library for enumeration, which basically is a backport from the original one in Python version 3.4 to version 2.7. The advantage of this solution is simple: Instead of using hardcoded strings, which is very error-prone (typos), we focus on using enumeration fields. Possible errors using the enumeration fields are much more likely to be detected by the Python interpreter. An example for the location enumeration: Location.kitchensink or Location.kettlebase.

A significant drawback of the Pyhop HTN planner is the missing visualisation. You have to rely entirely on the console output for debugging, which is most of the time not easy to comprehend. Drawing the tree in addition to the output would be handy. Especially for someone who is not familiar with this planner, it is hard to understand the intermediate planning steps (especially failures) of the planner.
There are four levels of logging: Verbose level 0 only prints the solution, but nothing else. Verbose level 1 displays all operators and methods in addition. Verbose level 2 features a message on each recursive call. Using this level of logging, you can see which task is divided into methods or operators by seeing different functions in the array that has to be computed. Also backtracking can be observed. The highest level of debugging, level 3, prints the whole state object after each step and any action the planner is performing. Having a big domain with lots of entries will lead to excessive text output when using verbose level 3.
The general approach on understanding what the planner is doing is by comparing the array of tasks/ methods/ operators from step n to step n+1. In this way you can see the decomposition of high level tasks or methods. The same applies for the state object. After executing an operator, parameters of the state object have to be changed to have an effect. For example, after running the operator “open(state, robot, kettle)” the kettle state should be changed from “Itemstate.closed” to “Itemstate.open”.

3. Analysis of the results

To test our result, we add three additional test cases to each test with a different world state. For each case this also shows different world states which can be handled successfully. Alternative test cases consist of the following changes:

1. Kettle is closed, full and hot
2. Kettle is open, full and cold

3. Kettle is open, empty and it is standing on shelf 1

The tests are run with minimal debugging output and include time logging, to keep the result readable and measure, how fast the planner solves the given task. We also add a variation of test 2 with a high amount of teacups. This variation is used to check the performance within a bigger domain. One performance problem we encountered using a sized up domain is the deep copy of state objects on every operator / method call. This probably slows down the planner unnecessarily (call by value).
Also the dynamic creation of the “RobotArm” enum for a high number of tea cups is very slow and needs optimization. Enumerations are usually used in a static context. But this backdraw is caused by our implementation of the teaathome domain, not by pyhop itself.

Times measured for the planner (without setting up tasks) is as follows:

Testcase	Time (seconds)
test1	0.031
test2	0.157
test3	0.063
test2 with 5000 teacups	8.400
test2 with 10000 teacups	24.000

The quality of the plans seems to be sufficient, since there are no unnecessary tasks/operators included. Only exception is the usage of the goto-Operator, which in our case does nothing when the given position is already reached, but it still shows up in the plan. The result seems flexible and can handle different world states, also it is possible to make more than 2 cups of tea.
Test 3 could be improved and shortened by taking two cups at once, if possible for the robot. Since SHOP (and also pyhop which is based on SHOP) only generates totally ordered plans, this could only be hardcoded.

To use the results for a robot in an actual environment, a possibility to include acting into the planning process has to be found. Especially it would be of importance to include changed domain knowledge (e.g. if a cup is dirty / clean) into the plan. Running the planner in a control loop is a simple idea to solve this problem. The planner and the acting-controller return, in addition to the partial or failed plan, the state of the domain to this loop. New knowledge is included into the state, so the planner can be called again with this updated domain state. This procedure can be repeated, until a successful plan is found or no new knowledge is added. At this point, it is certain that in an otherwise unchanged domain a plan can not be found.