Informatics 2D Coursework 2: Planning with PDDL

Craig Innes and Alex Lascarides

Deadline : **3pm, Thursday 29th March 2018**

Introduction

This assignment is about planning using the **Planning Domain Definition Language (PDDL)**. It consists of three parts:

- Part 1 - A written exercise in which you will formalize a planning problem using **PDDL** (worth 45%).
- Part 2 - Implementing and verifying the correctness of your written model using a browser-based PDDL planner (worth 30%).
- Part 3 - Extending the model to deal with additional complications in the environment (25%)

This assignment is marked out of 100, and it is worth 12.5% of your overall grade for Inf-2D.

The files you need are available from:

`http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/Inf2D-Coursework2.tar.gz`

In this document, you will find two icons:

☞ means that the following sentence describes how to answer the question/task.

✏ means this is task that should be done.

Submission

Create a directory in which you keep the files you submit for this assignment. This directory should be called `Inf2d-ass2-s<matric>` where `<matric>` is your matriculation number (e.g 0929508).

In this directory, write your answers to part 1 in a file you call `answers.txt`. Write your answers to part 1 in `answer.txt`. For the implementation in parts 2 and 3, copy and edit the corresponding `*-template.pl` files available in the coursework archive.

Submit your assignment by creating a new archive file from your directory `Inf2d-ass2-s<matric>`. You do this using the following command in a DICE machine:

```
tar cvzf Inf2d-ass2-s<matric>.tar.gz Inf2d-ass2-s<matric>
```

Submit this archive file using the command:

```
submit inf2d cw2 Inf2d-ass2-s<matric>.tar.gz
```

The deadline for submission is **3pm, Thursday 29th March 2018**.

You can submit more than once up until the submission deadline. All submissions are timestamped automatically. Identically named files will overwrite earlier submitted versions, so we will mark the latest submission that comes in before the deadline.

If you submit anything before the deadline, you may not resubmit afterward. (This policy allows us to begin marking submissions immediately after the deadline, without having to worry that some may need to be re-marked).

If you do not submit anything before the deadline, you may submit *exactly once* after the deadline, and a late penalty will be applied to this submission, unless you have received an approved extension. Please be aware that late submissions may receive lower priority for marking, and marks may not be returned within the same timeframe as for on-time submissions.

Warning: Unfortunately the `submit` command will technically allow you to submit late even if you submitted before the deadline (i.e. it does not enforce the above policy). Don't do this! We will mark the version that we retrieve just after the deadline, and (even worse) you may still be penalized for submitting late because the timestamp will update.

For additional information about late penalties and extension requests, see the School web page below. Do **not** email any course staff directly about extension requests; you must follow the instructions on the web page.

`http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests`

Good Scholarly Practice: Please remember the University requirement as regards all assessed work for credit. Details about this can be found at:

`http://www.ed.ac.uk/schools-departments/academic-services/students/undergraduate/discipline/academic-misconduct`

and at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

Part 1 : Modelling The Planner (Total Marks: 45%)

For the first part of this assignment, you will develop a model for a planning problem. You will formalize the domain using the Planning Domain Definition Language (PDDL, see Lecture 16 or Russell & Norvig 3rd Edition, section 10.1 for a reminder). You will need to define the predicates you use to describe the state of the world and the actions. You will also need to define the initial and goal states. The actions will be available to the planner. You will then use your definitions to infer a plan for an instance of the problem.

Write your answers in the relevant sections of the `answer.txt` file found in the assignment archive.

Problem Description

The university has decided to replace all its human bar staff in the Teviot lounge bar with fully automated bar tending agents. These agents can plan and execute a sequence of actions to accomplish some goals.

Figure 1 shows a map of the lounge bar divided into 7 discrete areas: The bar itself (BAR), the upper front area (UF), middle front (MF), lower front (LF), lower back (LB), middle back (MB), and upper back (UB). The agent can step directly between two adjacent areas (e.g UF and MF), but cannot move directly between areas blocked by grey dividers (e.g It cannot step directly from MB to MF).

For simplicity, lets assume :

- Customers place drink orders using a mobile app, so they no longer have to walk to the bar to order or pay.
- There is only one kind of drink available (Beer)
- The agent's movement within the building is discrete

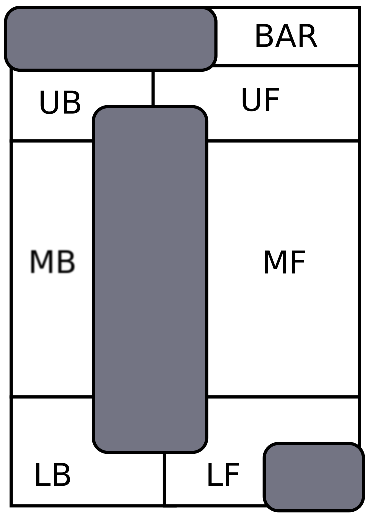

Figure 1: Map of the lounge bar

The setup is then as follows: The agent receives a list of customer orders which give their name and location in the bar. For each order, the agent must fill a glass with beer from behind the bar, then deliver this drink directly to the customer's table.

Describing World State (15%)

The first step in the creation of a model to design the structures that will hold information about the environment. Then planner can access these structures to construct a plan.

You should define a minimal set of predicates that can encode every state of the problem.

The initial model should include information about the 7 areas and their connections, the locations of the agent, customers and drink glasses, and the current state of both the agent and drinks.

☞ Fill in the first section in the `answers.txt` file:

✏ [**1.1 (2%)**] Define the predicates you will use to describe the map. Specifically, how will you establish that one location is **Adjacent** to another?

✏ [**1.2 (2%)**] Define how you will keep track of the **Location** that an agent, glass or customer is **At**.

✏ [**1.3 (3%)**] Define the predicates you will need to model a **Glass** and, in particular, whether or not it **Contains Beer**.

✏ [**1.4 (3%)**] Define the predicates you will need to model a **Customer**, and whether or not that customer has been **Served**

✏ [**1.5 (5%)**] Using the symbols you just defined and the map as depicted in Figure 1, describe the following *initial state* of the problem:

- The map is as depicted in Figure 1
- The agent starts off at the BAR
- There is a glass at the BAR
- There is a single customer at LB

Actions (15%)

In the `answers.txt` file, formalize the following actions in terms of the PDDL action schemata. That is, in terms of *Preconditions* and *Effects*:

[**1.6 (4%)**]The agent can **Pick up** a glass. Note, an agent can only hold one glass at a time, so if it is currently **Holding** a glass, it will be unable to pick up another.

[**1.7 (4%)**] The agent can **Hand over** a glass of beer to a customer that is in the same location as the agent, after which a customer should be said to be **Served**

[**1.8 (3%)**] At the BAR, the agent can **Pour** beer into a glass that it is **Holding** if the glass does not already **Contain Beer**.

[**1.9 (4%)**] The agent can **Move** from its current location to an **Adjacent** one.

Take particular care when defining these actions to consider what happens to the location of items that the agent is holding when she is moving. (*Hint*: Do we even need to explicitly know the location of a glass if we know the agent is holding it? At what point does its location become important? See Example 10.1.1 from Russel & Norvig 3rd Edition for additional hints).

Backward State Space Search (15 %)

Using the definitions you developed in the first part of this assignment, you should be able to construct a **plan** for a given initial/goal state. Remember, a plan is a sequence of actions that leads from the initial state to a state which satisfies the goal.

One way of finding such a plan is by using **Backwards State-Space search** (See Lecture 17 or Russell & Norvig 3rd Edition Section 10.2.2). Here, we start at the goal state and work backwards towards the initial state by considering only *relevant actions* at each step. That is, actions which could be the last step in a plan leading up to the current step's goal.

[**1.10 (15%)**] Formalize the following initial and goal states using the predicates you developed in the previous sections:

- **Initial State** - The agent is at the BAR, there is one empty glass G at the BAR, and there is a customer C who hasn't yet been served at UF.
- **Goal State** - The customer C has been served

Then, using **Backwards State-Space Search**, provide a plan which works backwards from the goal state towards the initial state. Show in your working:

- The relevant actions available at each step
- The current updated goal state at each stage of the search
- The final plan executed

Note in your working, you **do not** have to enumerate the entire search space and show every possible dead-end. Just assume at each step that the agent manages to choose the "correct" action and avoids backtracking.

Part 2: Implementation (Total Marks: 30%)

In this second part of the assignment, you will implement the model you developed in Part 1. We will be using the online PDDL editor provided by the *Planning.Domains* toolbox. It is available at:

`http://editor.planning.domains`

Writing PDDL using the online planner

Translating the PDDL statements written in the previous section into code that `editor.planning.domains` understands requires a little work. The notation used by the online editor is slightly different from the one used in the lectures and Russel & Norvig. It more closely resembles a Lisp-like language. [1]

[1]For those interested, the official language specification for PDDL can be found at: `http://icaps-conference.org/ipc2008/deterministic/data/mcdermott-et-al-tr-1998.pdf`

To see an example of the difference, click `File -> Load` in the online editor, and load up `blocks-world-domain.pddl` and `blocks-world-problem.pddl` from the assignment folder. These files give an implementation of the well known "Blocks World" problem (See Russel & Norvig 3rd Edition Section 10.1.3). To run the planner, click `Solve`, set the domain to `blocks-world-domain.pddl` and the problem to `blocks-world-problem.pddl`, then click `Plan`.

As you will see, the main differences are in the syntax:

- Brackets are placed around the *outside* of predicates rather than after them, and list items are *whitespace separated* rather than comma separated. For example, `On(B1, B2)` becomes `(On B1 B2)`
- Arguments in actions/predicate definitions must start with a *"?"*.
- Instead of using the $\wedge$ operator, conjunctions are bundled together using the `and` predicate. For example `Clear(a)` $\wedge$ `Clear(b)` becomes `(and (Clear a) (Clear b))`

The other big difference is the use of *types*. In previous exercises and lectures, you may be used to seeing single-argument, atemporal predicates which simply describe the type of thing an object is. For example, `Cup(X)`, `Dog(D1)`, or `TeaBag(T1)`. In the online editor, we replicate this behaviour by defining `types` in the domain definition. For example: `(:types cup dog teabag)`. We then explicitly annotate our definititions of predicate and action arguments with the type of object they accept using a dash (-).

For example, consider the blocks-world action `Move`. In our standard notation, we would write:

$$\begin{aligned}
&Action(Move(b,x,y)):\\
&\quad PRECOND: On(b,x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge\\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y)\\
&\quad EFFECT: On(b,y) \wedge Clear(x) \wedge \neg On(b,x) \wedge \neg Clear(y)
\end{aligned}$$

However, using the Lisp-like notation of the online editor, we would write:

```
(:action MOVE
  :parameters (?b - block ?x - object ?y - block)
  :precondition (and
    (On ?b ?x)
    (Clear ?b)
    (Clear ?y)
    (not (= ?b ?x)) (not (= ?b ?y)) (not (= ?x ?y))
  )
  :effect (and
    (On ?b ?y) (Clear ?x) (not (On ?b ?x)) (not (Clear ?y))
  )
)
```

Notice here that we can use the built-in `object` type if we wish to allow an action to accept an object of any type. We can also express inheritance via types. For example:

```
(:types cup flask teapot - container teabag)
```

Expresses that `cup`, `flask` and `teapot` are sub-types of `container`, while teabag is only a sub-type of the top-level type `object`

Task 2.1 : Translate to Official PDDL (10%)

After reading the sample files,

☞ Make a copy of the `domain-template.pddl` file and rename it as `domain-task-21.pddl`.

✏ **[2.1 (10%)]** Translate the predicates/actions of your model and save them in this file.

Simple experiments

The next two exercises are minimal experiments to learn the language accepted by the planner, and for you to test the correctness of the model. Each task has at least one solution, so if the planner fails to find a plan, there might be something wrong with your PDDL definitions.

☞ For each task, make a new copy of the file `problem-template.pddl` and rename it `problem-task-<#>.pddl` (replace `<#>` with 22 or 23 respectively). Any comment or description can go inside the `.pddl` source file (Comment lines in PDDL start with a semi-colon ;).

Task 2.2: Hello PDDL

In this instance of the problem:

- The agent starts at the BAR
- There is an empty glass at the BAR
- There is an unserved customer at LB
- The **goal** is for the customer to be served.

✏ [**2.2 (10%)**] Implement and test.

Task 2.3: A More Complex Problem

In this instance of the problem:

- The agent starts at MF
- There is one empty glass at MB and another at LB
- There are two customers—One at UB and one at LF
- The **goal** is that both customers are served, and the agent is at the BAR.

✏ [**2.3 (10%)**] Implement and test.

Part 3: Extending the Domain (Total Marks: 25 %)

In this section you will implement the following extension to the original problem:

A few customers have accidentally dropped their glasses on the floor, meaning some areas contain shards of broken glass. The agent cannot move onto an area with broken glass, and so must sweep it up from an adjacent area using a broom. The agent cannot hold a glass of beer and the broom at the same time, so must put down one to pick up the other.

For this extension task, you may find the the PDDL extension module `:quantified-preconditions` useful (The templates in the assignment folder all import the `adl` module, which automatically bundles `:quantified-preconditions` as a dependency). This extension allows you to avoid cumbersome book-keeping predicates by allowing existential and universal quantification (using `exists` and `forall` respectively) in action pre-conditions and goal states.

To see how this works, let's return to the `MOVE` action from the blocks-world example. Instead of having an extra `Clear` predicate to mark that nothing is on top of a block, we can define a block as being clear if there are no other blocks on top of it:

```
(:action MOVE
  :parameters (?b - block ?x - block ?y - object)
  :precondition (and
    (On ?b ?x)
    (not (= ?b ?x)) (not (= ?b ?y)) (not (= ?x ?y)
    (not (exists (?z - block) (On ?z ?b)))
    (not (exists (?z - block) (On ?z ?y)))
    )
  :effect (and
    (On ?b ?y)(not (On ?b ?x)))
  )
)
```

Note, it is entirely possible to complete the entire assignment *without* using quantified pre-conditions, but using them might result in shorter, more elegant solutions.

☞ For this task, make a copy of the file `domain-template.pddl` and rename it `domain-task-31.pddl`. Similarly, make a copy of `problem-template.pddl` and rename it `problem-task-32.pddl`. We'd reccommend you paste in your implementation from task 2 first, then make the appropriate modifications to extend it to task 3.

✎ [**3.1 (15%)**] Modify your existing implementation such that:

- The agent cannot **Move** into an area if it contains **Broken Glass**.
- The agent can remove broken glass from an area by **Sweeping it Up**. To sweep up, the agent must be **At** an adjacent location to the broken glass, and must be **Holding** a **Broom**
- The agent cannot hold a glass and the broom at the same time, so must **Put Down** one item to **Pick Up** the other.

✎ [**3.2 (10%)**] Implement an initial state where:

- The agent starts at the BAR
- There is an unserved customer at LB
- There is an empty glass at BAR
- There is a broom at BAR
- There is broken glass at MF and MB

And a goal of:

- All customers are served
- The agent is at the BAR
- There is not any broken glass on any part of the floor

Then test that your model generates a valid plan.