Informatics 2D Coursework 2: Symbolic Planning

Jay Park, Alex Lascarides

Deadline: **12 pm (noon), Thursday 28th March 2024**

Introduction

This coursework is about designing and evaluating symbolic planning domains and problems using the Planning Domain Definition Language (PDDL). It is marked out of 100 and worth 15% of the overall course grade. It consists of three components:

- **Modelling**: Creating PDDL problem and domain files for a given specification (35 marks);
- **Experiment**: Designing an experiment to evaluate a planner (15 marks);
- **Extensions**: Extending the domain to deal with real-world challenges (50 marks).

The files you need are on the course Learn website: click "Assessment" (under the main "Content" tab) and go down to"Coursework 2: Symbolic Planning". You will download a file `Inf2d-cw2.zip` which can be unpacked using the following command:

```
unzip Inf2d-cw2.zip
```

This will create a directory `Inf2d-cw2` which contains: example blocks world domain and problem files (`EXAMPLE-blocks-world-domain.pddl`,`EXAMPLE-blocks-world-problem.pddl`), the metric FF v1.0 planner (`ff`), and report templates: use `report.docx` or `report.tex`[1] to produce the report.

Submision

For this coursework, we will use Gradescope[2] for both submitting and marking. To submit your work, access Gradescope via an interface on the LEARN website for the course: click "Assessment" (under the main "Content" tab), go to "Assignment Submission" and click the Gradescope submission link for "Coursework 2: Symbolic planning". You will be transferred to the Gradescope website where you will be asked to submit a programming assignment "Coursework 2: Symbolic planning". Open the submission area for the coursework in which the new window will appear (see Figure 1). In this window drag & drop or click to browse so as to add all the files you will produce in this coursework (see example submission in Figure 2; **do not change the names of these files!**) and press "upload". After doing this, an autograder will run tests to check if the expected files are uploaded. If you do not attempt some tasks (e.g., Task 3.4), do not upload files for them; the autograder will notice this and will pass the tests. Note that you need to compile `report.tex` into `report.pdf` or export `report.docx` as `report.pdf` and submit that. The autograder makes sure that the files uploaded are syntactically valid and that the plans are produced in a reasonable amount of time (the autograder will timeout if running all planning problems instances takes more than 10 minutes).

[1] If you have not used LaTeXbefore, you may find https://computing.help.inf.ed.ac.uk/latex and https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes useful.

[2] https://gradescope.com/

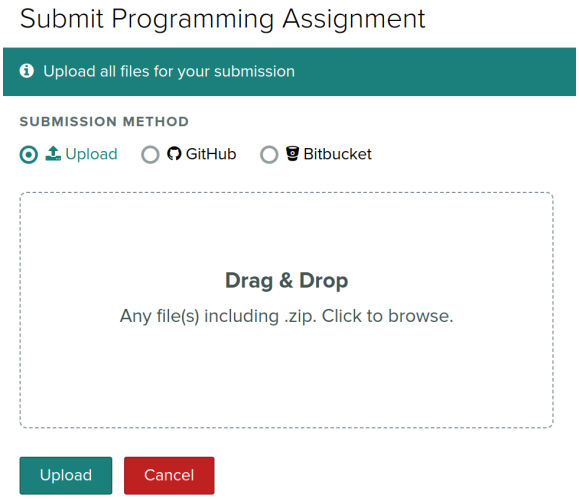

Figure 1: Submission window

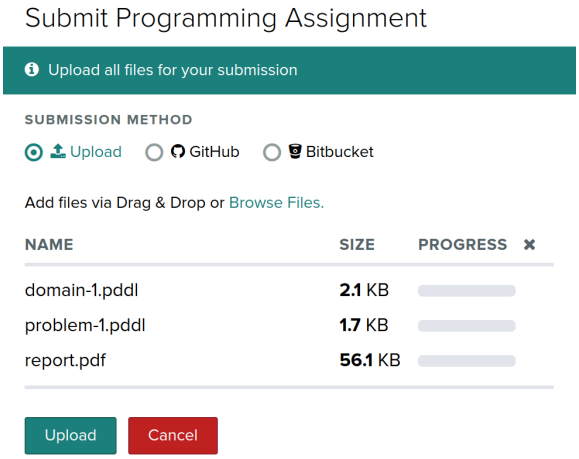

Figure 2: Example files to submit

If you attempt all tasks in this coursework, the upload should contain the following files:

```
All possible files to submit for coursework 2
├── domain-1.pddl
├── domain-2.pddl
├── domain-3.pddl
├── domain-4.pddl
├── domain-5.pddl
├── problem-1.pddl
├── problem-1-hard.pddl
├── problem-2.pddl
├── problem-3.pddl
├── problem-4.pddl
├── problem-5.pddl
└── report.pdf
```

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 time frame as for on-time submissions. For information about late penalties and extension requests, see the School web page here; Inf2D courseworks follow rule 1. Do not email any course staff directly about extension requests; you must follow the instructions on the web page.

Good Scholarly Practice: Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School web page `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 code repository (e.g., GitHub), then you must set access permissions appropriately (for this coursework, that means only you should be able to access it).

Task 1: Modelling (35 marks)

ShoppEd™, a successful supermarket chain, is seeking to expand its business via technical innovation! For this, they develop a robotic assistant called ShopBot, which has to travel within the supermarket to collect items designated by a shopping list and bring them to a checkout stand. Items of certain categories (e.g., fresh produce) may have to be weighed before checkout. The objective of this task is to design and test the PDDL domain and problem files for this scenario.

Task 1.1: Domain file (25 marks)

A PDDL domain file describes the predicates of the domain as well as the actions executable in it. Create a domain file called `domain-1.pddl` and define predicates for describing the following:

- the supermarket, i.e., the layout of aisle cells, shelves, weighing scale and checkout stand
- the locations of domain objects like ShopBot and shopping items (cabbage, etc.).
- whether or not ShopBot is holding an object.
- whether or not a shopping item has to be and/or has been weighed. (In Figure 3, weighable items are marked by grey weighing scale icons in their top right corners.)
- whether or not a shopping item has been checked out.

Additionally in `domain-1.pddl`, define the actions a ShopBot can perform. Specify the arguments, preconditions and effects for each of the actions:

- ShopBot can *move* (only) between two connected aisle cells. ShopBot cannot move diagonally.
- ShopBot can *pick up* an object if ShopBot is in a cell adjacent to the location containing the item. ShopBot can hold at most one object at a time.
- ShopBot can *drop* an object it is carrying to an adjacent target location, freeing its hand.
- ShopBot can *weigh* a shopping item at an adjacent weighing scale. The item has to be of type that needs to be weighed before checkout.
- ShopBot can *check out* a shopping item placed on an adjacent checkout stand. ShopBot must not be holding anything when checking out an item. Weighable items must have been weighed beforehand.

You can define any additional predicates and actions as long as they help effective and efficient modelling of the described domain.

Task 1.2: Problem file (10 marks)

A PDDL problem file describes the initial and the goal state for a planning problem. Create a `problem-1.pddl` file that defines the initial state and the goal conditions as depicted in Figure 3, using the predicates and actions from Task 1.1. In each problem, the goal is to check out the items in the shopping list (e.g., `potato`, `ketchup`, `toothpaste`, `pizza` for Figure 3).

Task 1.3: Testing (0 marks)

To test the correctness of your `domain-1.pddl` and `problem-1.pddl` files, use the planner executable `ff` included in the handout. If you are not working on DICE or the provided `ff` does not work for you, and if you are comfortable with manually compiling C programs, you can compile the planner from the source found here. Please note that there is **no support** for using `ff` on any operating system other than DICE. To run `ff`, execute the following command[3]:

```
./ff -o domain-1.pddl -f problem-1.pddl
```

[3] If you get a `Permission denied` error, try assigning the execute permission for `ff` by running `chmod u+x ff`.

Figure 3: The visualization of the initial state and the goal conditions of the shopping problem. Numbered grey squares represent different aisle cells that ShopBot can travel. Weighing scales and checkout stands are represented by orange/yellow squares with icons of weighing scale and cash register respectively. Shopping items are placed on shelves, and items that need weighing before checkout are marked with grey scale icons. The ShopBot is in the cell marked with the number 20. Finally, the goal conditions are specified by the shopping list on the top right corner.

Task 2: Experiment (15 marks)

To find the plan, the provided `ff` planner uses a best-first search with the following heuristic evaluation function $f(s)$ for each state s:

$$f(s) = w_g g(s) + w_h h(s)$$

where $g(s)$ is the cost so far to reach s, $h(s)$ is the estimated cost to get from s to the goal state, and $w_g, w_h \in \mathbb{Z}$ are weights. The default values for the weights in `ff` are $w_g = 1$ and $w_h = 5$.

The objective of this task is to design and perform the experiment to evaluate the effect of w_g and w_h on the planner's performance.

Task 2.1: Design (5 marks)

The current problem files are not challenging enough for the planner. For this subtask, design a harder problem called `problem-1-hard.pddl`, whilst keeping `domain-1.pddl` fixed. This problem instance will be used to benchmark the planner's performance. Describe the design of your problem instance in the report, and justify your choice.

Task 2.2: Evaluation (10 marks)

Using `domain-1.pddl` and `problem-1-hard.pddl`, design an experiment to evaluate the effect of different values of w_g and w_h on the planner's performance. To run the experiments with different values of w_g and w_h use the following command:[4].

```
./ff -E -g <w_g> -h <w_h> -o domain-1.pddl -f problem-1-hard.pddl
```

Give your experiment results in the report. Include the analysis which should be brief and have both *quantitative* and *qualitative* elements.

[4] `-E` flag turns off enforced hill-climbing (EHC), which is a fast-yet-incomplete search strategy used by default, after which `ff` falls back into the best-first search. Disabling EHC with `-E` lets the planner directly start with the best-first search.

Task 3: Extensions (50 marks)

The objective of this task is to extend the domain to make it closer to the real world.

Task 3.1: Put all your eggs in one basket (10 marks)

It seems well-motivated to constrain the number of items that ShopBot can hold. Nonetheless, it generates a behaviour that is rather inefficient and unrealistic; each shopping item must be hand-carried all the way from its shelf to the checkout, one after another! Fortunately, it turns out the real world has already provided an elegant solution to this problem: shopping baskets. You decide to incorporate the solution into the domain while preserving the maximum-held-item constraint.

In this subtask, enable ShopBot to carry multiple shopping items in a shopping basket. A basket is an object that ShopBot can hold, which can contain any number of shopping items. Shopping baskets can only be picked up near a stack of baskets (green square in Figure 4) and can be returned at the checkout. (As before, ShopBot should not be holding anything when checking out an item.) The plans generated by the `ff` planner should involve use of shopping basket.

Create a new domain file `domain-2.pddl` including new predicates and actions that implement the extension. Then, create a problem file `problem-2.pddl` containing the initial state as depicted in Figure 4 and the same goal as `problem-1.pddl`.

Figure 4: The shopping problem scenario for Task 3.1. Now the supermarket has a basket stack adjacent to cells 16 and 19. The goal state remains the same as in `problem-1.pddl`.

Task 3.2: Top it up (10 marks)

In the real world, of course, one does not simply walk out of a supermarket without paying the price for the shopped items. ShoppEd™ offers a member-exclusive prepaid card service, and you are asked to incorporate the payment system into the ShopBot domain.

In this subtask, introduce the concept of *price* for the shopping items, in the unit of 'credits'. Further, implement a new state variable that keeps track of the credit balance held by ShopBot. Prices must be appropriately deducted from the balance after each check-out. ShopBot cannot check out an item with price higher than the current balance. ShopBot can top up more credits at the top-up station (purple square in Figure 5) by a fixed amount of 5 credits each time.

Create a new domain file `domain-3.pddl` that implements the extension, and a problem file `problem-3.pddl` containing the initial state and the goal conditions as depicted in Figure 5 (*hint: numeric fluents*). Notice specifications of prices of the shopping items and ShopBot's starting credit balance. Don't forget to test your domain and problem files using the `ff` planner.

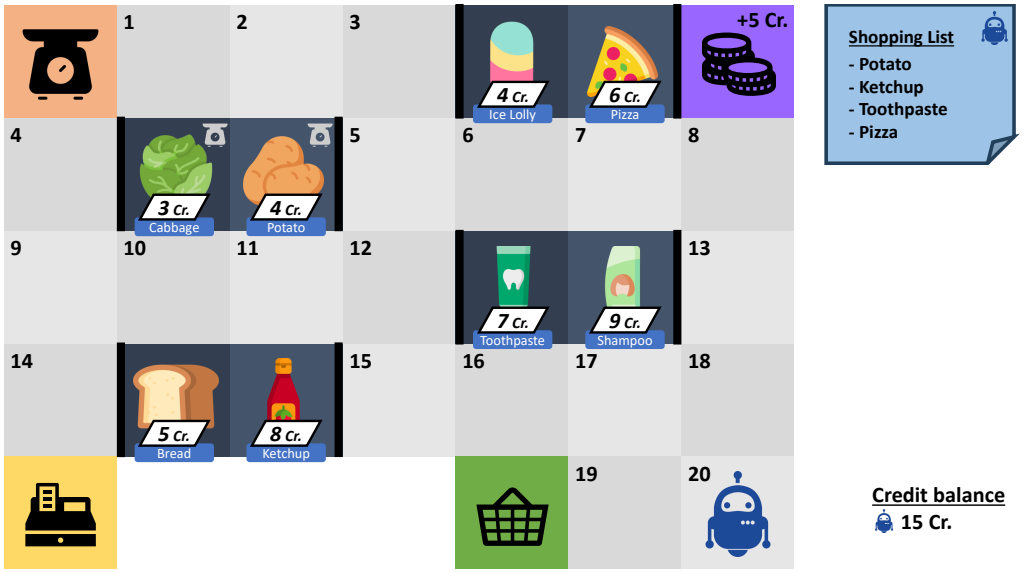

Figure 5: The shopping problem scenario for Task 3.2. Now the supermarket has a credit top-up station above cell 6, where ShopBot can add credits to its balance, 5 credits at a time. The initial state now describes the prices for the shopping items, along with the starting credit balance.

Task 3.3: Physical distancing (5 marks)

ShoppEd™ asks you whether your domain can model centralised control of multiple ShopBots dispatched by different users. In particular, they want to make sure multiple ShopBots can navigate and operate smoothly in a supermarket without bumping into each other.

In this subtask, enable presence of multiple ShopBots with distinct shopping goals, tracking which shopping items are checked out by which ShopBot. In addition, implement a constraint that prevents any ShopBot from moving into an aisle cell if it is already occupied by another ShopBot, so as to avoid collisions.

Create a new domain file `domain-4.pddl` that implements the extension, and a problem file `problem-4.pddl` containing the initial state and the goal conditions as depicted in Figure 6. Don't forget to test your domain and problem files using the `ff` planner.

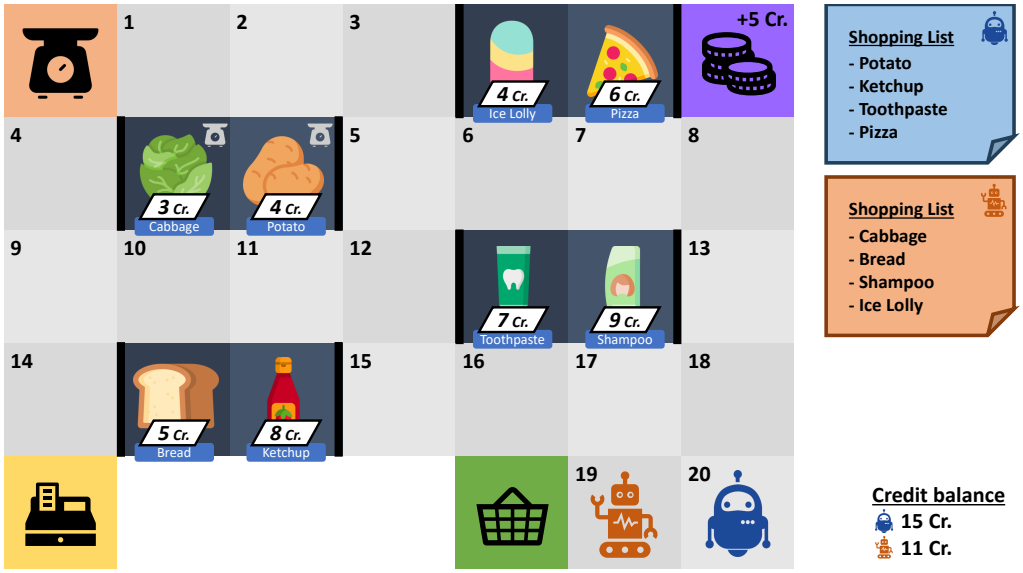

Figure 6: The shopping problem scenario for Task 3.3. Now the supermarket has an additional ShopBot dispatched by another user, with different shopping goals and starting credit balance. The ShopBots may not occupy the same aisle cell at any time.

Task 3.4: Your extension (25 marks)

In the real world, there are even more considerations that the above domain specifications don't take into account. For this last subtask, you need to motivate and design extended domain and problem files that make them more realistic. In particular, for this subtask you need to:

- Identify a factor that is a part of the real-world scenario but has not so far been a part of the domain specification you have formalised. Describe this factor and how it affects planning (informally, in English). Describe your extension in the report, and justify your choice.
- Create `domain-5.pddl` and implement this factor. You can add predicates and actions as necessary, but you have to describe them in the report.
- Create `problem-5.pddl` where this factor affects which plans are valid, and which aren't.
- Test your domain and problem files using the `ff` planner.

WARNING: you will only get credit for this task if your extension is well-motivated, correctly implemented and clearly explained. This task is intended to challenge students who already feel that they have mastered the course material, and want to go further. Do not attempt this question unless you have completed all the previous subtasks, and are sure that you have done a good job on those. Also, do not attempt this subtask if you have personally spent 10 or more hours on the coursework already. The maximum mark awarded for this subtask is only 3% of your overall course mark. Unless you really whizzed through the earlier subtasks, please stop now and spend your time and energy revising other course materials or getting more sleep. Both are likely to have a much bigger impact on your final mark for the course.