Informatics 2D Coursework 2: Symbolic Planning

Nick Ferguson and Craig Innes

Due at **12:00** on **27th March 2025** (Submission on Gradescope)

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.

Submission

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 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 skip 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/`

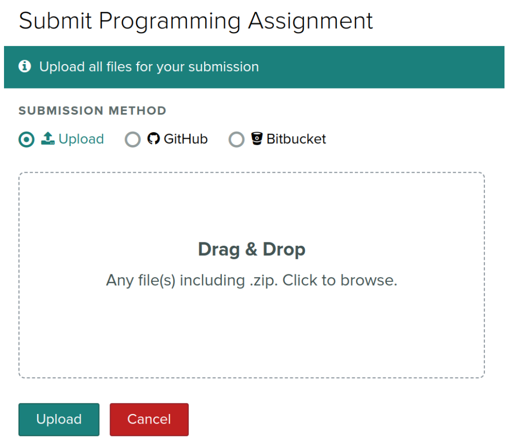
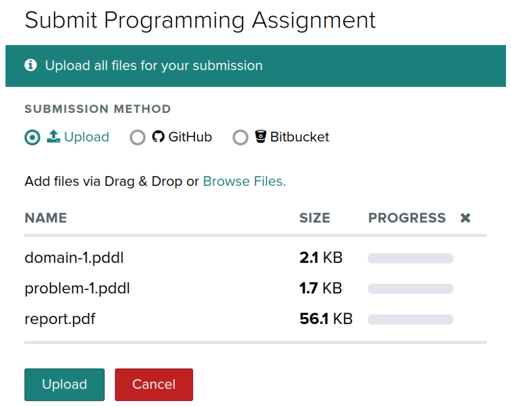

Submit Programming Assignment
Upload all files for your submission
SUBMISSION METHOD
[✓ Upload] GitHub Bitbucket
Drag & Drop
Any file(s) including .zip. Click to browse.
Upload
Cancel

Figure 1: figure
Submission window


Submit Programming Assignment
Upload all files for your submission
SUBMISSION METHOD
[✓ Upload] GitHub Bitbucket
Add files via Drag & Drop or Browse Files.
Upload
Cancel

NAME	SIZE	PROGRESS
domain-1.pddl	2.1 KB	
problem-1.pddl	1.7 KB	
report.pdf	56.1 KB	

Figure 2: figure
Example files to submit

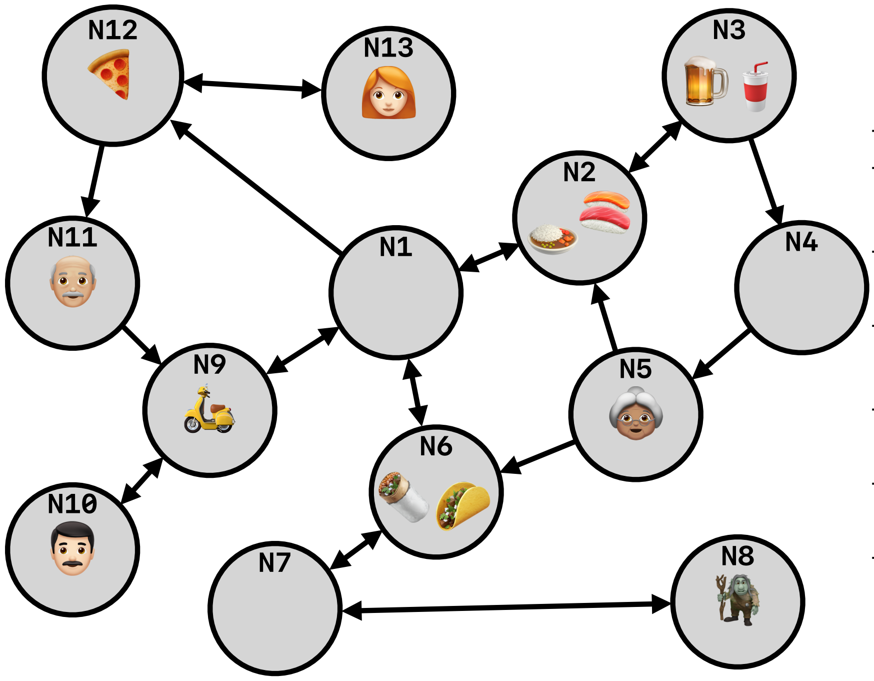
If you attempt all tasks in this coursework, the upload should contain the following files:

```
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, visit the Coursework Planner page on Learn.

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).



Person	Ordered
👨	🍺 Beer 🍕 Pizza
👵🏽	🍣 Sushi
👩‍🦰	🌯 Burrito 🥤 Soda
👴	🍛 Curry
🧌	🌮 Taco

Figure 3: The layout of the road network, locations of restaurants which serve different items, the courier's initial position, and the list of deliveries that need to be made.

Tasks

1 Modelling the Problem (35 marks)

Consider a 🛵 COURIER whose task is to deliver food around a city. The courier has to travel around the road network, picking up and making deliveries. Your task is to design and test PDDL domain and problem files for this scenario.

1.1 Domain File (20 marks)

Create a PDDL domain file called `domain-1.pddl` to describe the predicates of the domain as well as the actions executable in it. Declare predicates to describe the following.

Road network The layout of the road network, consisting of nodes connected by roads, noting the directionality of edges as indicated by the arrows in figure 3.

Location A predicate to denote that something is on a particular node.

Delivery made Whether a particular delivery has been made.

Picked up Whether a particular item has been picked up by the COURIER.

Serves Whether a location or node serves a particular item.

Order placed Whether a customer, located at a node, has ordered a particular item.

Also in `domain-1.pddl`, define the actions that the 🛵 COURIER can perform - specifically, the arguments, preconditions and effects for each of the following concepts.

Movement The COURIER can only move between two nodes which are connected by a road.

Picking up The COURIER can only pick up and item if it is on the location that serves that item, and the item is not already in their bag.

Making deliveries The COURIER can only make a delivery if it is at the location of the person who ordered the item, and the item is in the COURIER's bag.

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

1.2 Problem File (15 marks)

Create a PDDL problem file called `problem-1.pddl` which describes the initial and goal states for the problem. It should define the initial state depicted in figure 3, including: the road network in terms of which nodes are connected to each other; the COURIER's initial position, the locations at which different items are served, and the list of deliveries showing what each customer needs. Use the predicates created in task 1.1. Also define the goal, which is that each item in the list should be delivered to the person who ordered it.

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] (note that there is no space between the '1' and '.', that's just the way it has been rendered in this handout).

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

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 an experiment to evaluate the effect of w_g and w_h on the planner's performance.

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.

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
```

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

[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.

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

3 Extensions (50 marks)

In reality, the problem of delivering food around a city is more complex than the simplified scenario described in the previous tasks. In this section, we will consider three extensions to the scenario which will require modifications to the domain and problem files. Each extension is a modification of the original scenario, and can be completed independently of the others.

3.1 Multiple 🛵 Couriers (10 marks)

Let's consider the idea of multiple 🛵 Couriers working in the same network. In this extension, you will need to modify your domain and problem files from task 1 to include a second 🛵 Courier, which will start at node **N10**. More than one courier 🛵 Couriers cannot be on the same location at the same time. In other words, a courier cannot move on to a node if there exists a different courier that is on that node. Modify the movement action to model this constraint.

One approach to implementing the movement constraint is to use *existential quantifiers* in PDDL. We must first ensure that we import the existential quantifier requirement at the top of the domain file.

```
    (:requirements :adl :existential-preconditions)
```

Existential quantifiers allow you to express that at least one object exists which satisfies a condition, following the syntax below.

```
    (exists (?variable1 - type1 ?variable2 - type2 ...)
        (condition)
    )
```

In the classic *block world* example, we could use existential quantifiers to express that there is a block at a location.

```
    (exists (?b - block ?loc - location)
        (and
            (at ?b ?loc)
            (clear ?b)
        )
    )
```

You may also wish to use the equality predicate =, which can be used to compare two objects, and evaluates to true if the objects are the same.

```
    (= ?object1 ?object2)
```

To summarise: create new domain and problem files called `domain-2.pddl` and `problem-2.pddl` to model this extension and implement the following changes.

- Modify the movement action to account for the fact that a 🛵 Courier cannot move to a node if another 🛵 Courier is already there.
- Include the second 🛵 Courier and its initial location in the problem file.

3.2 Courier Capacity (5 marks)

In real life, 🛵 Couriers can't carry an infinite number of items in their bags, so we will model the idea of the 🛵 Courier's courier as having a limited capacity. Each item has a volume associated with it as shown

Item	**Volume**
🍕 Pizza	8
🍣 Sushi	4
🌯 Burrito	6
🍺 Beer	2
🥤 Soda	3
🍛 Curry	7
🌮 Taco	5

Table 1: The volume of each item in the scenario.

in table 1, and the sum of the volumes of the items that the courier is carrying cannot exceed the stated capacity. Therefore, a 🛵 COURIER can only pick up an item if it will not exceed the courier's capacity.

To implement this extension, we will need to make use of *numeric fluents* in PDDL. A numeric fluent is a variable which applies to an object, and maintains a value throughout the plan. You'll need import fluents at the top of the domain file.

```
    (:requirements :adl :fluents)
```

Fluents are declared as functions at the top of the domain file.

```
    (:functions
        (<variable_name> <parameter_name> - <object_type>)
        ...
        (<variable_name> <parameter_name> - <object_type>)
    )
```

To implement the capacity constraint, we will need to declare a numeric fluent to represent the capacity of the courier, one to represent the current load of the courier, and one to represent the volume of a particular item. Fluents can be used in both the preconditions and effects of actions, and their values can be modified in different ways depending on the operator. For +, -, /, *:

```
(+ <variable_name_x> <variable_name_y>) ; addition used for example purposes
```

Operators such as increase, decrease, and assign can modify the value of a variable:

```
(increase (<variable_name> <object_type>) <value>) ; increase <variable_name>
    by <value>
```

It is possible to use another numeric variable in place of value.

```
(decrease
(<variable_name_x> <parameter_name>) ; decrease <variable_name_x>
(<variable_name_y> - <object_type>) ; by <variable_name_y>
)
```

The value is the amount by which the fluent is modified. See https://planning.wiki/ref/pddl21/domain#numeric-fluents for further documentation on fluents.

Create new domain and problem files called domain-3.pddl and problem-3.pddl to model this extension, and implement the following changes.

- Add numeric fluents to represent the courier's capacity, current load, and the volume of an item.
- Update picking-up and delivery-making actions to account for the courier capacity constraint and changes to the courier's load.
- Modify the problem file to include the volume of each item in the scenario, the initial courier load of **0**, and the courier capacity of **10**.

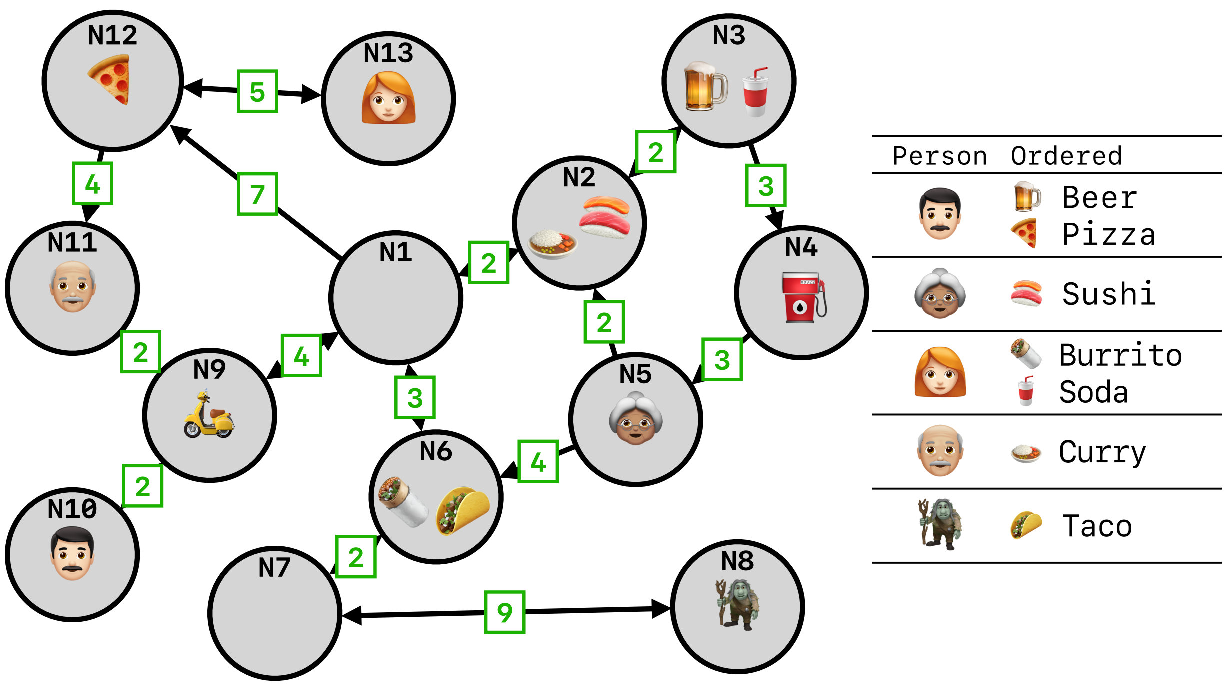


Figure 4: The same scenario from figure 3, but with the addition of road lengths and the location of the ⛽ Petrol Station on node **N4**.

3.3 Refuelling (10 marks)

Building on this application of numeric fluents, let's consider the idea of refuelling the 🛵 Courier. The 🛵 Courier has a fuel tank which has a limited capacity, and uses fuel to move between nodes. The road network has been updated to give each road a length, shown in figure 4. The fuel cost of moving between two nodes is equal to the length of the road between them. The 🛵 Courier can only move between two nodes if it has enough fuel to make the journey. The 🛵 Courier can however refuel at the location of the ⛽ Petrol Station, which is located at node **N4**. Once the 🛵 Courier is at the ⛽ Petrol Station, it can refuel its fuel tank to its full capacity.

For this extension, create new domain and problem files called `domain-4.pddl` and `problem-4.pddl` and implement the following changes.

- Add numeric fluents to represent the fuel level of the 🛵 Courier, the length of each road, and the maximum fuel capacity of the 🛵 Courier.
- Modify the movement action to account for the fuel cost of moving between nodes.
- Implement a new *refuelling* action, which allows the 🛵 Courier to fill its fuel tank to its maximum capacity.
- Modify the problem file to include road lengths, an initial fuel level of **20**, a maximum fuel capacity of **40**, and the location of the ⛽ Petrol Station.

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.