

Informatics 2D Coursework 2: Situation Calculus and Planning

Deadline: 3pm on Thursday 30th March 2017

Introduction

This assignment is about the Situation Calculus and Planning. It consists of three parts. Part 1 (worth 40%) is a written exercise that requires you to formalise a planning problem, using Situation Calculus to represent the world. Part 2 (worth 25%) is about implementing the model and verifying its correctness using a planner based on the Golog syntax. And part 3 (worth 35%) is about extending the model as well as its implementation in order to deal with additional aspects of the environment.

Important: Please attend the lecture on Thursday, 9th March, at 3:10pm in AT LT3, for a presentation of this assignment together with hints and tips on how to complete it and how to program in Prolog. Please also attend the lab session on Friday 11am-1pm Forest Hill Room 1.B32 in weeks 8 and 9 (17th/24th March).

The files that you need are available from:

<http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/>

Inf2D-Coursework2.tar.gz

Please put your matriculation number in all files you submit (except `planner.pl` and `plan.sh`, which you should not edit). Note that you do not need to put your name in the files. The deadline for submission is **3pm on Thursday 30th of March 2017**.

The paper write-up will be done in the `answer.txt` file and for the implementation part you will need to copy and edit the `*-template.pl` files available in the coursework archive.

To submit your assignment, create a new archive file with the files: `answer.txt`, `domain-*.pl`, and `instance-*.pl` in it. You can do this using the following command in a **DICE** machine:

```
tar cvzf Inf2d-Assignment2-s<matriculation>.tar.gz <assignment-dir>
```

where `<matriculation>` is your matriculation number (e.g. 0978621), and `<assignment-dir>` is the directory in which your assignment files are stored.

Submit this archive file using the command:

```
submit inf2d 2 Inf2d-Assignment2-s<matriculation>.tar.gz
```

The archive `Inf2d-Assignment2-s<matriculation>.tar.gz` should contain all original files, namely

- Part 1 answers should be in `answer.txt`
- Part 2
 - a domain file: `domain-task21.pl`
 - problem instances: `instance-task22.pl`, `instance-task23.pl` and `instance-task24.pl`
- Part 3
 - `domain-task31.pl`, `instance-task31.pl`
 - `domain-task32.pl`, `instance-task32.pl`
 - `domain-task33.pl`, `instance-task33a.pl` and `instance-task33b.pl`

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).

Demonstrator for this coursework is Kay LI <Kay.Li@ed.ac.uk>.

In this document, you will find two icons:

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

 means that the following sentence describes the question/task that should be answered/done.

1 Modelling the Planner (40%)

The first part of this assignment requires you to develop a model for a planning problem. You are initially asked to formalise the domain using the Situation Calculus, based on the same formalism as described in Russell & Norvig 3rd Edition, section 10.4.2. You will need to describe the state of the world using predicates and fluents for specifying the initial and goal states, and also some basic actions that affect the world. The actions will be available to the planner. You will then use the axioms you defined to infer a plan for an instance of the problem. In the assignment archive, you will find a text file called `answer.txt`. Please fill in your answer in the relevant sections of this template.

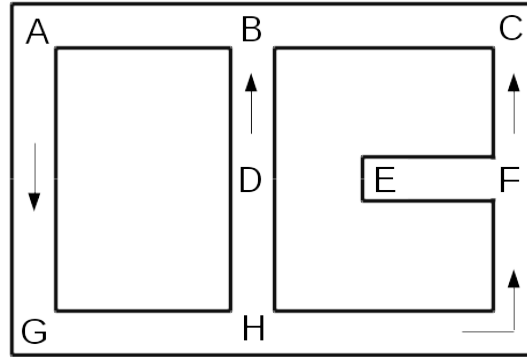


Figure 1: Map of the taxi problem, showing eight locations in a small city, and the one-way routes.

Problem Description

The task that you need to model is taxi dispatch service, whose job is to collect passengers located on the map, and take them to their destinations. The destination is known at the beginning of the run, and will not change. The map for this problem is divided in eight locations of particular interest where people typically get on or off. These places are labelled with letters from *A* to *H*, as shown in Fig. 1. The taxi-agent usually starts in place *A*, and moves in discrete steps between the eight locations using the available connections. Passengers are not expected to move except when in a taxi. Taxis can move freely from place to place in either direction, except in a few cases where the transition is only allowed in one direction. There are four “one-way streets”, and they are marked by arrows on the map.

1.1 Knowledge Base (10%)

The first step in the creation of a model is the design of the knowledge base, i.e. the structures that will hold information about the environment that the dispatcher (i.e. the planner) can look up when it chooses an action. The initial model should include information about the locations, about the locations of the taxi (i.e. the agent), the passenger, and its destination. You should define a minimal set of predicates that can encode every state of the problem; some of them will be atemporal predicates, which don’t change as time progresses, and some will be fluents, predicate whose value depend on the current situation.

✎ Fill in the first section in the answers.txt file:

- ✎ write how you would describe the street plan in the KB. You can specify which pairs of locations are adjacent to each other, but remember that some boundaries can be crossed in one direction only; [2%]
- ✎ explain how to keep track of the position at which the agent and passenger are located at any particular moment; [2%]
- ✎ describe what can be used for the destination of a passenger, which does not change with time. [2%]

d) ✎ Using the symbols you just defined, describe the initial state of the problem, which encodes the map in Figure 1, and locates the agent, the passenger and its destination at the generic locations L_a , L_p , and L_d respectively. [4%]

1.2 Actions (20%)

On the map, the agent can move from location to location, as long as these are connected; it can also pick up a passenger when it is located in the same place. At this point, the taxi is carrying a passenger, and can transport them to another location. In principle, the taxi can drop them at any time, but only when it arrives at the destination of the current passenger, it will complete the ride successfully. Once delivered, a passenger cannot be picked up again.

In the `answers.txt` file, formalize the following four actions in terms of possibility axioms and effect axioms:

- a) ✎ the agent can move from one place to an adjacent one, if it can cross the boundary in that direction; [2%]
- b) ✎ the agent can pick up objects, in this case passengers; once it is holding something, it can carry it around. It should not pick up itself, or passengers that have already been transported; [3%]
- c) ✎ when the agent carries a passenger, it can choose to drop it at any point in the current street, even if this is not the correct destination; [2%]
- d) ✎ only when the agent has reached the correct location, it can transport the passenger. After transport the passenger should be considered to have arrived, and is of no further interest to the taxi driver. [3%]

📖 You can omit universal quantifiers. Use `and`, `or`, `not`, and `=>` instead of \wedge , \vee , \neg , and \Rightarrow .

However, effect axioms are not sufficient: they describe how the new situation has been affected by the action executed, but they do not update information unrelated to the specific action, which may (or may not, if not updated) remain the same. Briefly explain this well known problem that, in the Situation Calculus, can be solved introducing successor-state axioms.

e) ✎ How do other formalisms for planning (e.g. STRIPS) solve the issue? [3%]

A successor-state axiom define the state of a fluent, based on its state in the previous situation and the new action executed. At a high level, it formalizes the idea that a fluent will be true if the most recent action makes it true, or if it was already true in the previous situation, and the most recent action has not changed its state.


f) ✎ Write the successor-state axioms for the fluents in your model. [7%]

1.3 Resolution and refutation (10%)

Using the definition developed in this first part of the assignment, it should be possible to prove the feasibility of a plan or, given a goal, to find a plan that achieves it. One way of doing this is a proof by resolution¹. You will have to convert the possibility and successor-state axioms to Conjunctive Normal Form, and then try to prove the negation of the goal. If there is a contradiction in the theory, the resolution process will find it, showing that the goal is indeed true. The sequence of axioms used shows which actions are needed to reach the goal.

Assuming that the initial state includes the following information:

- the agent is located in place a , on the map in Fig. 1
- the passenger is also located in place a
- the passenger must be taken to place h

 prove by refutation that the following goal can be reached

$\exists s. arrived(passenger, s)$

and show the plan found. [10%]

2 Implementation (25%)

The second part of the assignment is centred on the implementation of the model that you developed in Part 1. Once we have translated the axioms into rules that a planner can understand, we can work on more complex instances of the planner. The conversion is fairly straightforward. It consists mostly of a translation of logical symbols into ASCII characters, as we shall see in this section.

Note, that the constant A, B, \dots appear now in lower case letters a, b, \dots

A planner and the Golog syntax for Situation Calculus

Golog (alGOrithms in LOGic) is a macro language which extends the Situation Calculus to include constructs usually found in imperative language, such as conditional branches, loops, and procedure. This assignment simply uses the syntax proposed by Golog to represent the axioms of the Situation Calculus. Since the Golog interpreter is written in Prolog, the syntax is similar, but no knowledge of Prolog is required for this assignment.

In the coursework archive file you will find a bash script named `plan.sh`. Running this script without parameters will print a short explanation of its command line. The script calls the planner (`planner.pl`), loads the chosen problem file, and runs the planning procedure.

¹Russel & Norvig, Section 9.5

In the same folder, you will find two examples implementing a simple blocks world, `sample-blocks.pl` and `sample-blocks-domain.pl`. You can run the planner for this example using the command:

```
./plan.sh sample-blocks.pl
```

Inside the examples you will find additional documentation on the format.

To show the differences and similarities between situation calculus and the language read by the planner, we can compare two versions of the blocks world example (simplified). What follows are the possibility and successor-state axioms for the *move* action.

Following the conventions on R & N, we have predicates starting with an uppercase letter and variables in lower-case, quantified:

$$\forall x; y; s. Clear(x; s) \wedge Clear(y; s) \Rightarrow Poss(Move(x; y), s) \quad (1)$$

In Golog:

- the opposite is true: predicates begin with lower-case letters and variables with capitals
- quantifiers are dropped
- logical connectivities change: the implication symbol is $:$ — and it points from right to left. In other words, you should read $p : \neg q$ as *if q then p*
- a comma $,$ represents a conjunction
- disjunctions are marked by semi-colons $;$
- the end of a rule is marked by a dot $.$

For example, the above axiom 1 is written in Golog as follows:

```
poss(move(What, Where), S) :- clear(What, S), clear(Where, S).
```

In logic, a successor-state axiom is guarded by the predicate that verifies if the action is possible.

$$Poss(a, s) \Rightarrow on(x; y; Result(a, s)) \Leftrightarrow a = Move(x; y) \vee (on(x; y; s) \wedge a \neq Move(y; z))$$


This is done automatically by the planner or Golog interpreter, and can be dropped. Moreover, we are interested in the **planning task**, so we only keep one direction of the iff in the formula above, the \Leftarrow . The resulting Golog successor-state axiom for axiom 2 is:

```
on(Block, Support, result(A, S)) :- A = move(Block, Support); on(Block, Support, S), not(A = move(Block, _)).
```

where the semicolon $;$ is a disjunction ($_$), and the underscore $_$ is an anonymous variable that unifies with anything.


2.1 Translate axioms (10%)

After reading the sample files and the included documentation, make a copy of the `domain-template.pl` file and rename it `domain-task1.pl`.


 Translate the axioms of your model and save them in this file. [10%]

Simple experiments

The following three 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, and all plans do not exceed 15 actions in length; if the planner fails to find a plan, there might be something incorrect in the model.

 For each task, make a new copy of the file `instance-template.pl`, and rename it `instance-task#.pl`. Any comment or description can go inside the `.pl` source file.


2.2 Hello world

 Implement and test the simple problem from the previous section. [4%]

In this instance of the problem:

- the agent is located in place a , on the map in Fig. 1
- the passenger is also located in place a
- the passenger must be brought to place h
- the goal is that the passenger arrives at the destination.

2.3 Multiple goals


 Implement and test the simple problem from the previous section. [5%]

In this instance of the problem:

- the agent is located in place a
- the passenger is located in place g
- the passenger must be transported to place h
- the goal is that the passenger arrives *and* the taxi is parked in location e .

2.4 Reverse problem

Place a passenger, decide its destination, decide a goal; the resulting plan should touch all eight locations in the map at least once.

 Implement and test [6%]


In this instance of the problem:

- the agent is located in place a
- you can choose the initial location of the passenger
- you can choose the destination of the passenger
- you can choose the goal, but the resulting plan should touch every location on the map.

3 Extending the domain (35%)

In this section you will extend the original problem to include new action effects and goals. Only the Golog implementation of the axioms is required, but:


 make sure the code is properly commented when defining new predicates.

 For each task, make a new copy of the file `domain-template.pl`, and rename it `domain-task#.pl`. Any comment or description should go inside the `.pl` source file.

3.1 Multiple passengers and multiple taxis

The first extension should allow more than one passenger in the map at the same time, similar also more taxis are managed by one dispatcher. So, now the dispatcher is the agent rather than the single taxi.

Each taxi, however, can only carry one passenger (or rather one party of passengers) at the time: if a second passenger is calling, the current passenger needs to be brought to their destination first or a second taxi needs to be recruited. While a taxi stops at a location, another taxi cannot pass nor stop. Also, the bidirectional streets are so narrow that only one taxi can pass. You may choose to introduce other actions, such as `Wait`, in order to accomodate more than one taxi easily, but solutions with just one taxi are also acceptable in 3.1.

 Once implemented, test your code on this problem: [8%]

- all taxis start in place a
- passenger p_1 is located in place g , destination place d
- passenger p_2 is located in place h , destination place e
- passenger p_3 is located in place c , destination place b

3.2 Dependencies

🔍 Reusing existing code, add the option to specify dependencies between different taxi rides: e.g. if the ride of passenger p_1 depends on the ride of passenger p_2 , p_1 can be picked up only *after* p_2 has reached their destination. By default, there was no such constraint to start with.

✎ Implement the extended model and test this problem: [11%]

- the taxi starts in place a
- passenger p_1 is located in place a , destination place h , depends on the ride of p_2
- passenger p_2 is located in place h , destination place f
- the goal is to transport p_1 , which implicitly requires p_2 to be transported first.

3.3 Number of steps

✎ For the last exercise you should add the possibility to choose between plans that lead to longer waiting times for (a fixed number of) the passengers and the use of less taxis. Time is counted in steps, pick-up and drop-off count also one step each. [8%]

- all taxis start in place a
- at step 0 passenger p_1 is calling from place d , destination place g
- at step 2 passenger p_2 is calling from place h , destination place c
- at step 3 passenger p_3 is calling from place e , destination place h
- the goal is to transport all passengers and then park the agent in place a .

✎ Try also a second configuration which is different from the above by including more taxis and/or more passengers and/or more conflicts. [8%]