Informatics 2D Coursework 2: Situation Calculus and Planning

Matt Crosby

February 6, 2012

Introduction

This assignment is about the Situation Calculus and planning. It will evaluate your skills in formalising, implementing and testing a planning problem, presented as an English language description of possible actions and goals.

Part 1 is a written exercise that requires you to formalise a planning problem using Situation Calculus. Part 2 requires you to implement the model and verify its correctness using a planner based on the "GOLOG" syntax. In Part 3, you will extend the model and its implementation to deal with additional aspects of the environment.

The files that you need are available from :

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

Marks

This assignment is marked out of a total of 100%, and it contributes a total of 12.5% towards your overall grade for Inf2D.

Submission

Please put your matriculation number in all the files you submit (except `planner.pl` and `plan.sh`, which you should not edit). You do not need to put your name in the files. The deadline for submission is

2pm on Friday 23rd of March.

Written answers are to be placed in the `answers.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: `answers.txt`, `domain-*.pl`, and `instance-*.pl` in it. You can do this using the command:

```
tar czf Inf2d-Assignment-2-completed.tar.gz $Assignment-dir/
```

where `$Assignment-dir` is the directory in which your assignment files are stored.

The files are submitted using:

```
submit inf2d 2 Inf2d-Assignment-2-completed.tar.gz
```

Part 1: Modelling A Warehouse (Sokoban) Problem (45%)

The first part of the assignment requires you to develop a model for a planning problem. You will need to formalise the domain using the Situation Calculus based on the formalism found in Russel & Norvig, Section 10.3 (2nd Edition).[1] You will then use the axioms you defined to infer a plan for a simple instance of the problem.

Problem Description

You need to model a simple sokoban warehouse domain. In this domain, an agent moves around a grid world pushing boxes into desired locations. The agent can move to any orthogonally connected *empty* grid square. Additionally, there are crates in the domain that an agent can push. An agent can push a crate only in a straight line and only if the space behind it is empty. There may only be one crate (or agent) in any grid location at any given time.

Figure 1 shows an example problem. It contains eleven grid squares, three crates and one agent. Each crate has a respective goal location that it must be pushed to. We will refer to locations numbered so that the bottom left location is `loc1-1` and the two top-right locations are `loc2-4` and `loc3-3`.

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 planner can use when it chooses an action. The initial model should include information about the grid world, the crates and the location of the player.

You should define a 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 fluent predicates whose values depend on the current situation. Briefly comment all predicates you introduce. You should place your answers in the `answers.txt` file:

1.1) Specify how you would show which locations are <u>connected</u>. This should include the *direction* in which the locations are connected as this will help when defining the `push` action. *[2%]*

[1] Available at `http://www.inf.ed.ac.uk/teaching/courses/inf2d/timetable/SitCalc.pdf`.

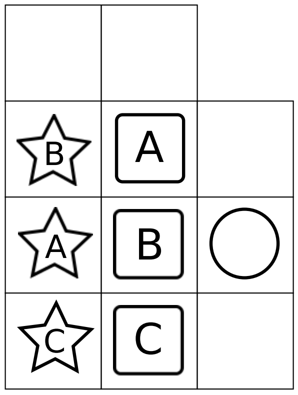

Figure 1: Example Sokoban problem. The grid is a 3x4 rectangle with the upper-right square missing. The agent is represented by the circle. There are three crates (represented by the boxes) A, B and C. The stars represent the goal locations for the respective crates.

1.2) Explain how to keep track of the position *at* which th e agent and crates are located at any particular moment and which locations are *empty*. *[2%]*

1.3) Using the symbols you just defined, write down the initial state of the problem depicted in Figure 1. *[4%]*

1.4) Using the symbols you just defined, describe how to specify the set of goal states of the problem depicted in Figure 1. *[2%]*

Actions (15%)

The agent can move from the space it occupies to any adjacent *empty* space. Alternatively, it can push a crate in a straight line as long as the space behind the crate is empty. For example, in Figure 1 the only `push` action the agent can perform (in the initial state) is to push crate B left into `loc1-3` which would leave the crate in `loc1-3` and the agent in `loc2-3`.

In the `answers.txt` file, formalise the following actions in terms of *possibility axioms* and *effect axioms*. You can omit universal quantifiers and use `and`, `or`, `not`, and => instead of $\wedge, \vee, \neg$ and $\Rightarrow$.

1.5) The agent can **move** to an adjacent *empty* space. *[2%]*

1.6) The agent can **push** a crate that it is next to into an empty space behind the crate. Note that a crate can only be pushed in a straight line and only into the square directly behind it. This moves the agent into the space the crate originally occupied. You should not need to use any arithmetic operations here. You can make use of the direction information you encoded with your *connected* predicate. *[5%]*

Effect axioms alone 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.

1.7) Briefly explain this well-known problem that, in Situation Calculus, can be solved by introducing *successor-state axioms.* How do other formalisms for planning (e.g. PDDL) solve this issue? *[3%]*

A successor-state axiom defines the state of a fluent, based on its state in the previous situation and the new action executed. At a high level, it formalises 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.

1.8) Write the successor-state axioms for the fluents in your model. *[5%]*

Resolution and refutation (20%)

Using the definitions you have just developed, 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 resolution. You will have to convert the possibility and successory-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 plan guarantees that the goal is reached. The sequence of axioms used shows which actions are needed to reach the goal.

1.9) Assuming that the initial state is as shown in Figure 1, prove by refutation that the following goal can be reached:

$$\exists s : \texttt{at(crateA, loc1-3, s)}$$

Note that the goal location for crate-A in this question is not the same as the goal location for crate-A in the figure. (You do not need to worry about crates-B and C for this question.) *[20%]*

Part 2: Implementation (20%)

The second part of the assignment is centred on the implementation of the model 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 problem. The conversion is fairly straightforward, mostly a translation of logical symbols into ASCII characters, as we shall see in this section.

A planner and the GOLOG syntax for the Situation Calculus

GOLOG (alGOrithms in LOGic) is a macro language which extends the Situation Calculus to include constructs usually found in imperative languages. Examples of these constructs are conditional branches, loops and procedures. This assignment merely 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 that planner (`planner.pl`), loads the chosen problem file, and runs the planning procedure.

In the same archive, 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 compare two (simplified) versions of the blocks world example. What follows are the possibility and successor-state axioms for the *move* action within the blocks world.

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

$$\forall x, y, s. \texttt{Clear}(x, s) \wedge \texttt{Clear}(y, s) \Rightarrow Poss(\texttt{Move}(x, y), s)$$

In Golog, the opposite is true; predicates begin with lower-case letters and variables with capitals. Quantifiers are dropped. Logical connectives change: the implication symbol is `:-` and points from right to left, so A`:-`B reads 'if B then A'. A comma represents a conjunction, while disjunctions are marked by semi-colons. The end of a rule is marked by a dot. The following statement means 'if What is clear and Where is clear in state S, then it is possible to move from What to Where in state S':

```
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.

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

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

```
on(Block, Support, results(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.

Task 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 goal of the following three exercises is 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`, where # is the number of the task. Any comment or description can go inside the `.pl` source file. Make sure to include both `instance-task#.pl` and `domain-task#.pl` for each task.

Task 2: The Planning Problem in Figure 1

Implement and test the problem shown in Figure 1. *[2%]*

Task 3: Crates go to Any Goal Location

Rewrite the problem so that the crates are allowed to end in any of the goal locations in Figure 1. Implement and test on the same problem. *[3%]*

Task 4: Inverse Problem

Find an initial state and goal specification for which the agent visits every location in the grid world in the resulting plan. The agent does not need to revisit its starting location but must visit the goal locations of crates and the crates' initial locations. You may change the number of crates, their locations and their goal locations as well as the agent's starting location (but not the size or shape of the grid world). The goal specification should only include the goal locations of crates (as before). *[5%]*

Part 3: Extending the Domain (35%)

In this section you will extend the original problem to include new actions, 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#.pd`. Any comment or description should go inside the `.pl` source file. Make sure to include both `instance-task#.pl` and `domain-task#.pl` for each task.

Each task is a separate extension to the basic problem (Part 2: Tasks 1 and 2) so that, for example, in Tasks 6 and 7 you should not include the rest action/requirement.

Task 5: Hard Work

In this version of the problem the agent cannot push a crate twice in a row, but must move or rest at least once in between pushes. Add a `rest` action to the domain definition and test it against the original problem setup. (The rest action is needed to ensure that a plan exists of length 15.) You may need to modify your original actions and introduce new predicate symbols to express whether an agent is tired/rested. You may want to use a `rest(X)` action (where X is the agent's location) as the planner does not deal well with actions with no parameters. *[8%]*

Task 6: Unlocking the Crates

In this version of the domain each crate starts locked to the ground. The agent cannot push the boxes until it has found and picked up the key for each crate. Add a `pickup` action that lets the agent pick up a key that is in its current location. You should not add any more actions; instead, update your definitions for the push action so that they require the agent to have picked up the correct key. Once implemented, test on the problem shown in Figure 2. *[12%]*

Task 7: Number of Actions

For this final exercise you should add the stipulation that the final plan has either an odd or an even number of steps. Since we are only interested in the parity, you do not need to introduce arithmetic operations to achieve this. Run the experiment for the problem shown in Figure 1 requiring an even and then an odd number of actions. (The agent may still move after pushing all the crates to their respective goal locations.) *[15%]*

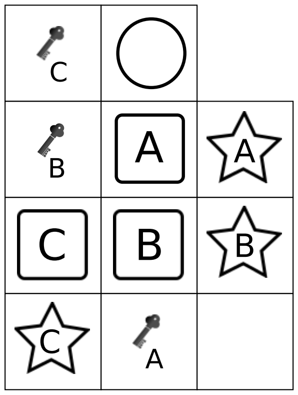

Figure 2: Problem for task 6. A key's label corresponds to the crate it unlocks.