CompSci 367/761 ASSIGNMENT 4: Domain Independent Planning
Due 8 October 11:59PM worth 5%.

1 Introduction

Note that in this document, we will use the terms "action" and "operator" interchangeably.

In assignment 3, the domain knowledge was separated out from the search knowledge. However, for each new domain you still had to write prolog code to generate successors, etc. In this assignment, we are exploring how to standardise the representation of states and successor knowledge, so that no programming is needed to change domains. Instead, the domain actions are described declaratively, and the states and goals are described using a variation of logic notation. States are conjunctions of positive literals, e.g., [at(houston), unvisited(dallas), unvisited(austin)], is a conjunction of those 3 literals. Goals are conjunctions of positive and negative literals, [at(houston], not(unvisited(dallas))].

We introduce a new term, *step*, which is a record data structure defined in **planning.pl** as *step(opName, opParams, stepCost)* where:

- *opName* is the name of an action,
- *opParams* is a binding list for its parameters, i.e., the values associated with each of that actions variables,
- *stepCost* with is the cost of that action when used with that binding list.

The heart of domain independent planning is being able to:

1. check whether a list of goal literals is satisfied by a state;
2. determine whether a state satisfies the problem's goal.
3. determine which steps are applicable to a state;
4. apply an applicable step to a state to produce a new state;

Your assignment is code up these 4 core pieces for a domain-independent planner. You might find it easiest if you develop and test the code in the order presented above. These 4 predicates will be specified in more detail in the next section.

2 What you need to do

First, we need to clear up some terminology.

predicate a term of the form *predName(Argument*)* where "Argument*" indicates there may be zero or more arguments, where argument may be a bound or unbound variable, or a constant. Example: *not(edge(dallas,*

City, 3)) is a predicate with *predname* not, and one argument, *edge(dallas, City, 3)*, and that argument itself is a predicate with 3 arguments, *dallas*, *City*, and 3.

literal a predicate with no unbound variables.

negative predicate a not predicate as shown in example above.

There are 4 predicates mentioned in the Introduction, which you need to write the code for:

1. `satisfy(+Goals, +State)`
2. `satisfyGoal(+State)`
3. `op_Applicable(+State, ?OpName, ?Params)`
4. `op_ApplyOp(+OldState, +Step, ?NewState)`

where:

- *Goal* is predicate, e.g., **unvisited(houston)**,
- *Goals* is a list containing the list of *Goal*
- *State* is an ordered set (ordset library) of predicates,
- *OpName* is the name of an *op* specified in **tsp/domain.pl**,
- *Step* is an instance of a step data structure as described in the *step* record declaration in **planning.pl**,
- *Params* is the binding list, a list of the values to be bound to the variables in the preconditions and effects.

Goal and *State* are as described in the lectures. For example, a *State* data structure will only have positive fluents.

State is used as the index into closed list. It is used to detect duplicate states and to retrieve the solution. This requires that states have a canonical description (i.e., a unique representation, a 1-to-1 correlation between a state and its representation), which is why states are stored as ordered sets. Whenever you create a new state, you must ensure that it is still represented as an ordered set!

As mentioned in the lectures, the static predicates are stored in the problem's initial state description. For this assignment, this information is stored in the *InitialState* field in the predicate, *problem(problemName, InitialState, Goals)*, in the prolog database.

1. Predicate Specification

   (a) `satisfies(+Goal, +State)`

   Goal is a predicate. Examples of use of *satisfies*:

- `satisfies(edge(a, b, 3), State)`
- `satisfies(not(at(c)), State)`

What it means for a *State* to *satisfies* a *Goal* depends upon what type of goal it is, if the goal is a:

- positive fluent goal then it means *Goal* is an element of *State*
- positive static goal then it means that *Goal* is an element of the problem's initial state.
- positive metaLevel goal then it is defined by prolog code that needs to be called.
- negative goal, i.e., it is *not(Goal)*, then it means that *Goal* fails, regardless of which type of *Goal* it is.

(b) `satisfy(+Goals, +State)`

Goals is a list of *Goal*. A *State satisfy Goals* iff *State satisfies* every *Goal* in *Goals*.

State satisfyGoal iff *State satisfy* every goal in the problem's Goal list. The problem's Goal list is stored in problem. To access it, just execute "*problem(_, _, Goals)*" and *Goals* will be bound to the problems Goal list.

(c) `op_Applicable(+State, ?OpName, ?Params)`

OpName with *Params* is `op_Applicable` iff the *Preconditions* for *OpName* bound with the *Params* binding list makes *satisfy(Preconditions, State)* true.

(d) `op_ Apply(+OldState, +Step, ?NewState)`

NewState is `op_Apply` to *OldState* via *Step* iff using *NewState* = *OldState* - *Step*'s negative *Effects* + *Step*'s positive *Effects*. In other words, The step's negated effects are deleted from *OldState* and the positive effects are added to form *NewState*.

For example, if *OldState* = *[at(a), unvisited(b), unvisited(c)]*, the negative effects = *[not(at(a)), not(unvisited(b)]*, and the positive effects = *[at(b)]* then *NewState* = *[at(b), unvisited(c)]*. Note that even though the negative efects had *not* around them, it was their enclosed predicate that was deleted, e.g., *not(at(a))* deleted *at(a)* from the *NewState* list. Also note that *NewState* must be an ordered set, i.e., passes the `is_ordset(+Term)` test (see *ordset* library documentation).

3 Submission Information

1. What to submit

   You need to submit a zip archive, yourUpi.zip (e.g., mbar098.zip) containing ONLY the following files:

   - **counter.pl**

- **planning.pl**

planning.pl has all the original code plus your code for the 4 predicates described above. You need to also include in **planning.pl** any code that you have written that is required by your predicates in order to function properly.

2. When and where to submit

   You need to submit this to Canvas by 8 October 23:59. Late submissions are not accepted.

4 Marking Rubric

YOUR CODE WILL BE TESTED UNDER SWI PROLOG. IF IT DOES NOT RUN UNDER SWI PROLOG AS IS, IT WILL GET A ZERO!

There first predicate (`satisfies(+Goal, +State)`) is worth 2 marks and each of the other 3 predicates will be worth 1 mark for correct implementation. There may be partial credit in some cases.

1. Important Marking Information

   (a) Exceptions

   Late assignments WILL NOT BE ACCEPTED except under exceptional circumstances, e.g., illness or family emergencies. These will need to be documented as appropriate. Remember you should always backup your files somewhere other than on your computer, therefore if you computer dies with all your files on it. You are expected to be able to continue working in the lab machines from your backup files!

   (b) Copying is forbidden

   If any students are found to have cheated they will get zero marks for the assignment and may be brought up on academic dishonesty charges.