CompSci 366

Classical Planning

Outline

$\rightarrow$

- Main Data Structures
- Defining Main Relationships
- Search Space Structure

Main Data Structures

- *Problems* are represented as:

  problem(InitialSituation, GoalDescription)

  where ***InitialSituation***s are lists of object-level positive ground literals, and ***GoalDescription*** are lists of literals (both object-level and meta-level, both positive and negative).

Literals

- A *positive literal* is represented as simply a predicate, e.g., ***clear(a)***, ***on(a, b)***.
- A *ground literal* has no variables, ***clear(A)*** vs ***clear(a)***.
- A *negative literal* is ***not(P)*** where ***P*** is a positive literal and simply means that ***P*** is not true, e.g., ***not(clear(a))***, ***not(on(a, b))***.
- There is only one *meta-level predicate*, ***neq(P, Q)***, and simply means that ***P*** is not equal to ***Q***, e.g., ***neq(A, table)*** means that the variable ***A*** cannot be instantiated to the constant ***table***. All other domain predicates are *object-level predicates*, e.g., ***clear(X)***.

Examples

- The following could represent a situation: ***[on(a,b), on(b, table), clear(a), clear(c),on(c,table)]***, it is a list of positive ground object-level literals.
- The following represents a goal description: ***[on(a, X), on(Y, b), neq(X, Y), not(on(a, b))]***, it is a list containing both meta- and object-level literals, both positive and negative literals, and both ground and non-ground literals.

Examples cont’d

- The following represents a problem:
  problem(
  [on(a,b),on(b,table),clear(a),clear(c),on(c,table)],
  [on(a, X), on(Y, b), neq(X, Y), not(on(a, b))])

Operator Data Structures

- An *operator schema* is represented as: ***op(Name, Params, Preconds, Effects)***, where ***Params*** is a list of the parameters in the ***Preconds*** and ***Effects*** and which instantiate the schema, ***Preconds*** is a goal description that states when the operator can be applied, and ***Effects*** is an effects description that states how application of the operator changes a situation.

Operator Data Structures cont'd

- ***Effects*** are a list of both positive and/or negative object-level literals.
- Operator schema example:
  op(move,[Block,FromLoc,ToLoc],
  [on(Block,FromLoc),clear(Block), clear(ToLoc)],
  [not(on(Block,FromLoc)),clear(FromLoc),
  not(clear(ToLoc)), on(Block,ToLoc)])

Plan Data Structures

- A *plan* is a list of steps.
- A *step* is represented: ***step(OpName, Params)***, where ***OpName*** is the name of the operator (schema) to be executed at this point in the plan, and ***Params*** is a list of parameter bindings.
- Example of a step: ***step(move, [a,b,c])***
- Example of a two-step plan: ***[step(move,[a,b,c]), step(move,[b,d,a])]***

Outline

- Main Data Structures
- Defining Main Relationships

  $\longrightarrow$ – **solvedBy**
- Search Space Structure

solvedBy(+Problem, ?Plan)

- We will now define what it means for a problem to be solved by a plan.
- Example:

```
solvedBy(problem([on(a,b),on(b,table),clear(a)],
                 [on(b,a)] ),
         [step(move,[a,b,table]),
          step(move,[b,table,a]) ] )
```

Plans and Planning cont'd

- How/when does a plan represent a solution to the problem?
- Should we say that the plan ***[step(move,[a,b,table]), step(move,[b,table,a])]*** solves the problem ***problem([on(a,b),on(b,table),clear(a)],[on(b,a)])*** and if so, why?

Plans and Planning cont’d

- We can simulate applying the plan to the initial situation and see if the resulting situation satisfies the goal description.

[on(a,b),on(b,table),clear(a)]

move(a,b,table)

[on(a,table),on(b,table),clear(a), clear(b)]

move(b,table,a)

[on(a,table),$\underline{\textbf{\textit{on(b,a),}}}$ clear(b)]

Plans and Planning cont'd

- Informally, we showed that the plan achieved the goal situation from the initial situation by showing:
  - *The plan can be applied to the initial situation.*
  - *The resulting situation satisfies the goal description.*

Plans and Planning cont’d

- The idea is to keep applying each step of the plan to each new situation resulting from applying the earlier sequence of steps to the initial situation until the last step is applied and the final situation satisfies our goal.
- This is called *progression planning*, we are *progressing* the initial situation through the plan.

Progression Planning

- We now have an informal idea of how we can show the plan is indeed a solution.
- We will now formalise it!

Progression Planning cont’d

- *Why is this a good idea?*
- In the past there have been ad hoc planning algorithms that turned out to be “wrong”.
- Want it to be “obvious” that our planning algorithm is correct.

Progression Planning cont'd

- So, goal is to **formally** define what it means for a plan to be a solution to a problem.

- **Note:** If formalised appropriately, we can translate the formal definition into a Prolog program that implements that definition.
- **Hint:** *Try to make definitions recursive on plan structure!*

Progression Planning cont'd

- **Base case:** empty plan is a solution when problem's goal description is satisfied by problem's initial situation.
- ***solvedBy(problem(InitSit, GoalDesc), [ ]) :-***
  satisfiedBy(GoalDesc, InitSit).

Progression Planning cont’d

- **Inductive case:** non-empty plan is a solution when plan’s first step is applicable in problem’s initial situation and rest of plan is a solution to the new problem whose goal description is the same as before and whose initial situation is the one created by applying that step to the original problem’s initial situation.

Progression Planning cont’d

- ***solvedBy(problem(I, G), [Step | Rest]) :-***

  applicableIn(Step, I),

  applicationResult(Step, I, NextSit),

  solvedBy(problem(NextSit, G), Rest).

Progression Planning cont'd

- We've defined ***solvedBy(Problem, Plan)***, but have introduced:
  - ***satisfiedBy(GoalDesc, Situation)***
  - ***applicableIn(Step, Situation)***
  - ***applicationResult(Step, Sit, NewSit)***
- We now need to define these.
- **<u>Note:</u>** remember we are treating preconditions as goal descriptions.

Progression Planning cont’d

- To define ***satisfiedBy(GoalDesc, Situation)***, we need to define what can be in goal descriptions:
  - Positive and negative (i.e., *not/1*) literals.
  - Object-level and meta-level literals (i.e., *neq/2*).
  - Literals can contain both constants and/or variables.
- Also need to define what’s in a situation.

Progression Planning cont’d

- There are four cases:
  - Empty goal description.
  - First goal is a negative literal.
  - First goal is a meta-level literal.
  - First goal is a positive object-level literal.
- Variables in literals are taken care of by Prolog and treated as normal variables.

Progression Planning cont’d

- **Base case:** the empty goal description is satisfied by all situations.
- ***satisfiedBy([ ], _).***

Progression Planning cont’d

- Only positive object-level literals can be in a situation description.

- From the definitions of what can be in goal and in situation descriptions, we can define ***satisfiedBy(GoalDesc, Situation)*** using case analysis.

Progression Planning cont'd

- **Negative literal (*not(P)*) case:** is satisfied by a situation when ***P*** is not satisfied by that situation.
- ***satisfiedBy([not(P)| Rest], Sit) :- !,***

```
satisfiedBy([not(P)| Rest], Sit) :-  !,
        not(satisfiedBy([P] , Sit),
        satisfiedBy(Rest, Sit).
```

Progression Planning cont’d

- **Meta-level literal (*neq(P,Q)*) case:** is satisfied when ***P*** is not equal to ***Q***.

- ***satisfiedBy([neq(P, Q)|Rest], _) :- !,***
  not(P=Q),
  satisfiedBy(Rest, Sit).

Progression Planning cont'd

- **Positive object-level literal (*P*) case :** is satisfied by a situation ***S*** where ***P*** is represented in ***S***.
- ***satisfiedBy([P | Rest], Sit) :-***

```
satisfiedBy([P | Rest], Sit) :-
        member(P, Sit),
        satisfiedBy(Rest, Sit).
```

Progression Planning cont'd

- In defining ***satisfiedBy(Goal, Situation)***, we did not need to introduce new predicates.
- So only need to define:
  - ***applicableIn(Step, Situation)***
  - ***applicationResult(Step, Sit, NewSit)***

Progression Planning cont'd

- Need to define when a step can be applied to a situation.
- It can be applied when its operator's preconditions (with the current parameter bindings) are satisfied by the situation.
- ***applicableIn(step(Op, Params), Sit) :-***
  preconds(Op, Params, Preconds),
  satisfiedBy(Preconds, Sit).

Progression Planning cont’d

- Now only need to define:
  - ***applicationResult(Step, Sit, NewSit)***

- We will leave it to you to define this!

Recap

- By formally defining what relations we want to hold between terms, we are able to “fairly” transparently implement correct programs.
- Unfortunately, things as not trivial as they might seem, as we will see shortly.

Outline

- Main Data Structures
- Defining Main Relationships
- $\longrightarrow$ Search Space Structure

Search Space Structure

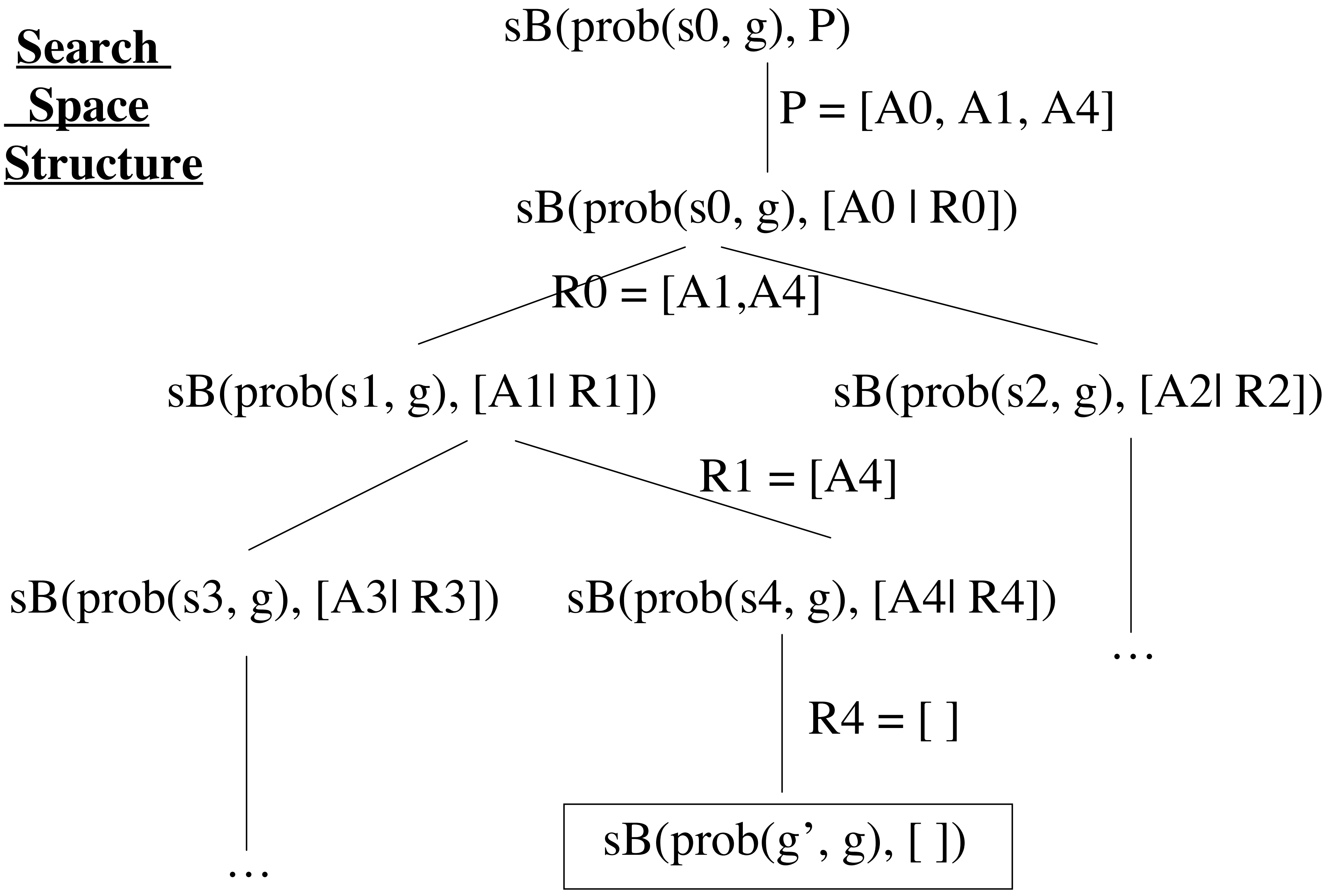
- Defined planning algorithm, but how will it work?
- Specifically, what will its search space of plans look like?
- To see this, need to revisit the ***solvedBy*** definition.

Search Space Structure cont'd

```
solvedBy(problem(InitSit, GoalDesc), [ ]) :-
        satisfiedBy(GoalDesc, InitSit).

solvedBy(problem(I, G), [Step | Rest]) :-
        applicableIn(Step, I),
        applicationResult(Step, I, NextSit),
        solvedBy(problem(NextSit, G), Rest).
```

Search Space Structure


sB(prob(s0, g), P)
P = [A0, A1, A4]
sB(prob(s0, g), [A0 | R0])
R0 = [A1,A4]
sB(prob(s1, g), [A1| R1])
sB(prob(s2, g), [A2| R2])
R1 = [A4]
sB(prob(s3, g), [A3| R3])
sB(prob(s4, g), [A4| R4])
…
R4 = [ ]
…
sB(prob(g', g), [ ])