Dipartimento di elettronica e informazione
Politecnico di Milano

Artificial Intelligence 2010-11

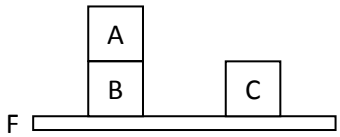
10. Planning in STRIPS

STRIPS (Stanford Research Institute Problem Solver) has been developed in the early 1970s (Fikes and Nilsson, 1971). STRIPS is a system for the automatic formation of action plans; it is loosely based on FOL, but it implements a number of ideas that allow one to solve the frame problem (see Section 9.5). The following description partially departs from the original version of STRIPS.

10.1 States, goals, and actions

States

In a STRIPS-like planning system, a state is represented as a finite set of *facts*. Every fact is a *positive ground literal*, that is, an atomic formula with a predicate symbol and a number of individual constants as arguments. For example, state S_0 below



can be represented as:

$S_0 = \{On(A,B), On(B,F), On(C,F), Clear(A), Clear(C), Clear(F)\}$

States should not be regarded as sets of axioms; rather, they represent the so-called *positive diagram* of a model, that is, the set of *all ground positive literals that are true in a specific model*. This implies that states obey the *Closed World Assumption* (CWA): every fact that is not explicitly asserted to be true is assumed to be false. Moreover, STRIPS adopts the *Unique Name Assumption* (UNA) (i.e., different constants are assumed to denote different objects of the domain) and the *Domain Closure Assumption* (DCA) (i.e., the domain contains only the named objects, that is, those objects that are denoted by a constant).

Goals

A *goal* is any positive ground literal; for example, we may want to reach a state in which

$On(B,C)$

A *goal list* is a finite conjunction of goals, for example:

$On(B,C) \wedge On(A,B)$

Note that the goal list above does not make any assumption on the final position of block C and on which blocks ought to be clear; in general a goal list is not a complete description of a state, but describes only a fragment of a state.

Action schemes

Actions are specified through parametric *action schemes*. Every action scheme specifies:

- the *name* and *parameters* of the action
- the *preconditions* for the execution of the action
- the *effects* of the execution of the action

Both the preconditions and the effects of an action are represented as finite conjunctions of positive and negative literals, which may contain the parameters of the action scheme as variables.

Here is the action scheme of move:

move(x,y,z):

preconditions $On(x,y) \wedge Clear(x) \wedge Clear(z)$

effects $\neg On(x,y) \wedge \neg Clear(z) \wedge On(x,z) \wedge Clear(y) \wedge Clear(F)$

We may also want to avoid certain impossible actions, like move(A,B,A), and certain useless ones, like move(A,B,B). To this purpose we can add some *constraints*:

move(x,y,z):

constraints $(x \neq y) \wedge (x \neq z) \wedge (y \neq z)$

preconditions $On(x,y) \wedge Clear(x) \wedge Clear(z)$

effects $\neg On(x,y) \wedge \neg Clear(z) \wedge On(x,z) \wedge Clear(y) \wedge Clear(F)$

Note that facts like $A \neq B$ are not part of the representation of states, and this is why we do not include constraints in the list of preconditions; however, such facts can be inferred thanks to the UNA.

Action execution

Let us see how a specific action, like for example move(C,F,A), can be executed. An action is executed in a given state by generating a new state, which represents the effects of the action. First, the variables of the action scheme are bound to the corresponding constants. Second, the system checks whether the constraints are satisfied and the preconditions hold in the current state. If this is the case, then:

(i) the current state is duplicated;

(ii) the negative effects (i.e., the negative literals in the list of effects) are deleted from the new state;

(iii) the positive effects (i.e., the positive literals in the list of effects) are added to the new state.

For example, let us execute action move(A,B,F) in S_0. First, the variables are bound to the corresponding constants:

$x = A$ $y = B$ $z = F$

As the constraints are satisfied, we check whether the preconditions

$On(A,B) \wedge Clear(A) \wedge Clear(F)$

hold in S_0. This is done by directly checking whether each of the three literals is contained in S_0. Given that this is the case, S_0 is duplicated:

{On(A,B), On(B,F), On(C,F), Clear(A), Clear(C), Clear(F)}

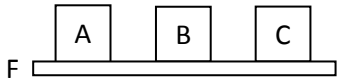
Then all the negative effects, On(A,B) and Clear(F), are deleted:

{On(B,F), On(C,F), Clear(A), Clear(C)}

Finally the positive effects On(C,A), Clear(B) and Clear(F) are added, and a new state is generated:

S_1 = {On(A,F), On(B,F), On(C,F), Clear(A), Clear(C), Clear(F)}

This state represents the configuration:

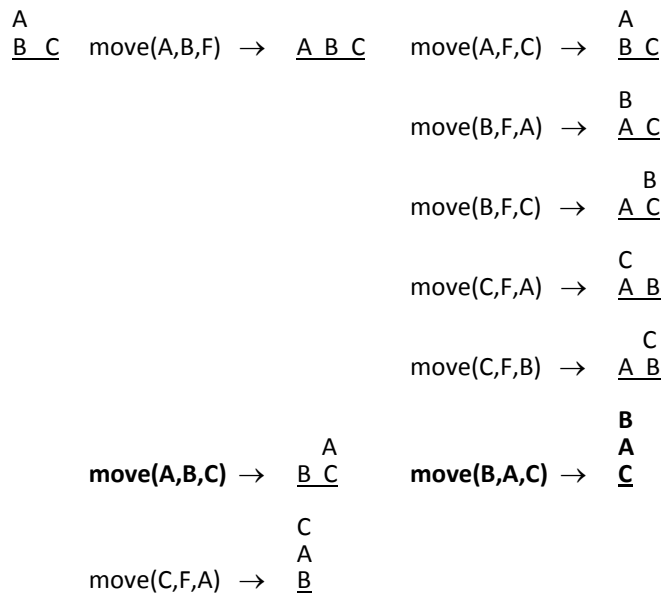


10.2 Plan formation

Suppose that we want to automatically build a plan that achieves the goal list

$$On(A,C) \wedge On(B,A)$$

starting from S_0. In principle, we could use one of the search strategies we have studies so far. For example, below we represent the search tree of a BF graph search (i.e., without the repetition of already explored states). Below states are represented graphically, but this is just a shorthand for the corresponding STRIPS representation as a set of facts.



The resulting plan is:

move(A,B,C)
move(B,F,A)

Let us understand how the search tree can be generated. States, as we have seen, are represented as sets of ground literals; moreover, with the action scheme of move(x,y,z) we know how to check if an action can be executed, and how to compute its results. The only problem is to specify the set of all possible actions, which is necessary to run a search algorithm; this could be done by associating to each action variable a domain of possible values, for example:

Block = {A,B,C}
Floor = {F}
Object = Block **or** Floor
move(x:Block, y:Object, z:Object):
constraints $(x \neq y) \wedge (x \neq z) \wedge (y \neq z)$
preconditions $On(x,y) \wedge Clear(x) \wedge Clear(z)$
effects $\neg On(x,y) \wedge \neg Clear(z) \wedge On(x,z) \wedge Clear(y) \wedge Clear(F)$

The set of all possible actions can now be generated by assigning to the variables of move(x,y,z) all possible combinations of values that satisfy the constraints.

It should be clear, however, that the branching factor of the search tree may be very large, in particular when we have a high number of blocks. Indeed, *forward search* (i.e., search processes starting from the initial state and moving forward towards the goal) is often inefficient in planning problems. Suppose for example that you are sitting on a chair in your office, and you want to go back home. If you perform a forward BF search, you start considering all possible actions you can perform in the current state, most of which will be completely irrelevant to reaching the goal: for example, opening a

drawer, grasping a book, drinking some water from a bottle, … How can we orient the search process so that we do not get stuck into considering a large collection of irrelevant actions?

A good approach is to replace forward search (from the initial state toward the goal) with *backward search* (from the goal back to the initial state). The idea is that only few actions can achieve the goal of bringing you home: for example, walking home or taking a bus. If you decide to take a bus, you now have a *subgoal*, that is, reaching the bus station. To achieve this subgoal, the only available option may be to walk there. This approach typically produces search trees with reasonably low branching factor.

But how can we implement backward search? In a state space approach, we would have to implement new procedures that produce the *previous state* (i.e., the state in which the action is executed) starting from the *resulting state*. But in a STRIPS-like system, we can derive this type of knowledge directly from action schemes. It is sufficient to select an action on the basis of one of its positive effects: the relevant features of the previous state are then described by the action's preconditions.

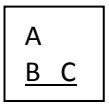
10.3 Plan formation

Let us see how a STRIPS-like backward search process may operate on our previous example (the complete algorithm is defined in Fikes and Nillsons' 1971 paper, available online). We start with the goal list

$$On(A,C) \wedge On(B,A)$$

and the initial state

$$S_0 = \{On(A,B), On(B,F), On(C,F), Clear(A), Clear(C), Clear(F)\}$$



As usual, the search process will build a tree of nodes. However in this example we shall not consider the generation of alternative paths, and shall try to reach the solution directly. Later we shall discuss why in general considering alternative paths is necessary.

Every node is a dynamic data structure with three components:

```
[state
 goalStack
 plan]
```

To initialise the planning process, we build the initial node

```
N  =  [state      S0
       goalStack  On(A,C) ∧ On(B,A)
       plan         ]
```

We then enter the main loop. We consider the goal list on top of the goal stack, $On(A,C) \wedge On(B,A)$. We analyse the goal list from left to right, and modify node N as soon as we discover a *difference*, that is, a literal in $On(A,C) \wedge On(B,A)$ that is not satisfied in the node's current state, S_0.

```
N  =  [state      S0
       goalStack  On(A,C)
                  On(A,C) ∧ On(B,A)
       plan         ]
```

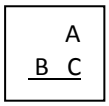
We now check whether the goal on top of the goal stack, On(A,C), holds in the node's state, S_0. As this is not the case, we look for an action that achieves this goal. By inspecting the positive effects of move(x,y,z), we find that a way to achieve On(A,C) is to move A from some place y to C; therefore we know that an action of the form move(A,y,C) will have to be part of the plan. We modify N, popping the goal stack and then pushing both the action and its the preconditions onto it:

N = [state S_0
goalStack On(A,y) ∧ Clear(A) ∧ Clear(C)
move(A,y,C)
On(A,C) ∧ On(B,A)
plan]

Again we consider the goal list on top of the goal stack, looking for a difference (in the previously defined sense). In doing so, we may assign values to variables if this satisfies a goal in the current state. In the case at hand, if we take y = B no difference is left; therefore the node is not expanded. We then pop the goal stack:

N = [state S_0
goalStack move(A,B,C)
On(A,C) ∧ On(B,A)
plan]

The top of the stack is now an action, move(A,B,C). We execute it in S_0, obtaining the state:



S_1 = {On(A,C), On(B,F), On(C,F), Clear(A), Clear(B), Clear(F)}

We then change the state in N, pop the action from the goal stack and add it to the plan:

N = [state S_1
goalStack On(A,C) ∧ On(B,A)
plan move(A,B,C)]

Again, we expand N by producing a difference from the goal list on top of the stack:

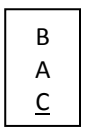
N = [state S_1
goalStack On(B,A)
On(A,C) ∧ On(B,A)
plan move(A,B,C)]

We check whether the goal on top of the goal stack holds in the current state, S_1. As this is not the case, we look for an action that achieves it. By inspecting the positive effects of move(x,y,z), we find that a way to achieve On(B,A) is to move B from some place y to A. We proceed as before:

N = [state S_1
goalStack On(B,y) ∧ Clear(B) ∧ Clear(A)
move(B,y,A)
On(A,C) ∧ On(B,A)
plan move(A,B,C)]

N = [state S_1
goalStack On(B,F) ∧ Clear(B) ∧Clear(A)
move(B,F,A)
On(A,C) ∧ On(B,A)
plan move(A,B,C)]

N = [state S_1
goalStack move(B,F,A)
On(A,C) ∧ On(B,A)
plan move(A,B,C)]



S_2 = {On(A,C), On(B,A), On(C,F), Clear(B), Clear(F)}

N = [state S_2
goalStack On(A,C) ∧ On(B,A)
plan move(A,B,C); move(B,F,A)]

At this point, the goal list on top of the goal stack does not produce any difference. Therefore we pop the goal stack:

N = [state S_2
goalStack
plan move(A,B,C); move(B,F,A)]

The goal stack being empty, we have successfully solved the problem. The final plan,

move(A,B,C); move(B,F,A)

is represented in the plan element of node N.

10.3 The need for search

In the previous section, we have neglected the fact that some of the steps were actually *choice points*, were different alternatives were open to us. For example, the first difference we have worked on is On(A,C), but we might as well have chosen On(B,A). Other choice points occur when we bind a variable to a value, because in general several different bindings are possible. Moreover, we have further choice points when different action schemes are suitable for reducing a difference.

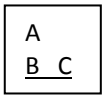
In general, therefore, the plan formation algorithm has to build a search tree representing different possible paths. However, due to the high branching factors (even with backward search), search strategies like BK or UC are unfeasible. Therefore, planning systems build search trees according to some heuristics; typically, these are domain-independent heuristics, like those in the area of CSPs.

10.4 Sussman's anomaly

Suppose we want to achieve the goal list

On(B,C) ∧ On(C,A)

Starting again from

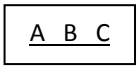


S_0 = {On(A,B), On(B,F), On(C,F), Clear(A), Clear(C), Clear(F)}

This planning problem, known as *Sussman's anomaly*, is particularly tricky. Below we sketch the construction of a plan (the reader may want to go through all steps, as in the previous example). First the systems tries to achieve goal On(B,C), and therefore plans the action

move(A,B,F)

which generates the state

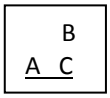


S_1 = {On(A,F), On(B,F), On(C,F), Clear(A), Clear(B), Clear(C), Clear(F)}

and then the action

move(B,F,C)

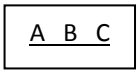
which generates the state



S_2 = {On(A,F), On(B,C), On(C,F), Clear(A), Clear(B), Clear(F)}

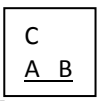
Now the system turns to the goal On(C,A), and therefore plans and executes the action

move(B,C,F)



S_3 = {On(A,F), On(B,F), On(C,F), Clear(A), Clear(B), Clear(C), Clear(F)}

At this point, the system is back to state S_1. However, this will not generate a loop, because the goal stack is different. Now action move(C,F,A) is planned and executed:



S_4 = {On(A,F), On(B,F), On(C,A), Clear(B), Clear(C), Clear(F)}

The system checks again the original goal list, and discovers that goal On(B,C) is no longer satisfied. Therefore the action move(B,F,C) is planned and executed:

S_5 = {On(A,F), On(B,C), On(C,A), Clear(B), Clear(F)}

Finally, the initial goal list is checked again. This time there is no difference, and the procedure terminates successfully with the plan:

(1) move(A,B,F); move(B,F,C); move(B,C,F); move(C,F,A); move(B,F,C)

This plan correctly achieves the initial goal list, but is not optimal. In fact, an optimal plan would be

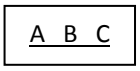
(2) move(A,B,F); move(C,F,A); move(B,F,C)

Note that the optimal plan can be obtained from plan (1) by eliminating the second and third actions,

move(B,F,C); move(B,C,F)

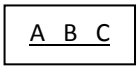
which in fact do nothing useful, given that the third action is just the inverse of the second action. Indeed, state

S_1 = {On(A,F), On(B,F), On(C,F), Clear(A), Clear(B), Clear(C), Clear(F)}



and state

S_3 = {On(A,F), On(B,F), On(C,F), Clear(A), Clear(B), Clear(C), Clear(F)}



are just the same!

One may wonder why the planning process does not go into an infinite loop, given that from state S_1 it goes back to S_3, which is identical to S_1: after all, the procedure we have described is deterministic, and a deterministic process should loop forever if it reaches a previous state. The answer is that we should not confuse a state of the world, S_i, with a state of the computation! The state of the computation, in fact, is determined by the whole node, and in particular by the *state of the world* and the *goal stack* (the *plan* component is only relevant to the final output, not to the state of the computation). The reader can verify that when the world reaches states S_1 and S_3 the goal stack is different. Intuitively, when it reaches S_1 the system is trying to achieve On(B,C), and when it reaches S_3 the system is trying to achieve On(C,A).

Finally, we may think that achieving a suboptimal plan depends on the order of the goals in the goal list. However, trying again with the goal list

On(C,A) $\wedge$ On(B,C)

would be even worse, because it would lead to the following plan:

(3) move(C,F,A); move(C,A,F); move(A,B,F); move(B,F,C); move(B,C,F);
move(C,F,A); move(B,F,C)

The discovery of such ‘anomalies’ in STRIPS has stimulated many important developments in planning techniques.

10.5 A planning problem with several action schemes

The Blocks World example is particularly simple, because it has only one action scheme. Here is another example, with several action schemes: a monkey is at location A in a room; there is a box at location B; the monkey wants to get the bananas that are hanging from the ceiling over location C, but to do so it needs to move the box under the bananas and climb on it.

Predicates:

At(x,y)	x is at location y
On(x,y)	x is on y
Has(x,y)	x has object y

Constants:

Mon	the monkey
Box	the box
Ban	the bananas
A, B, C	locations
Floor	the room’s floor

Initial state:

S_0 = {At(Mon,A), At(Box,B), At(Ban,C), On(Mon,Floor)}

Goal list:

G_0 = Has(Mon,Ban)

Action schemes:

walk(loc1,loc2):
constraints (loc1 ≠ loc2)
preconditions At(Mon,loc1) ∧ On(Mon,Floor)
effects ¬At(Mon,loc1) ∧ At(Mon,loc2)

pushBox(loc1,loc2):
constraints (loc1 ≠ loc2)
preconditions At(Box,loc1) ∧ At(Mon,loc1) ∧ On(Mon,Floor)
effects ¬At(Mon,loc1) ∧ ¬At(Box,loc1) ∧ At(Mon,loc2) ∧ At(Box,loc2)

climbUpBox(loc):
preconditions On(Mon,Floor) ∧ At(Box,loc) ∧ At(Mon,loc)
effects ¬On(Mon,Floor) ∧ On(Mon,Box)

climbDownBox(loc):
preconditions On(Mon,Box) ∧ At(Box,loc)
effects ¬On(Mon,Box) ∧ On(Mon,Floor)

graspBananas(loc):
preconditions At(Ban,loc) ∧ At(Box,loc) ∧ On(Mon,Box)
effects Has(Mon,Ban)

Examining the preconditions and the effects of all these action schemes, it appears that the only object of the world that can perform actions is the monkey (Mon). But this is reasonable, given that the other objects are not agents. Moreover the box, but not the bananas, can be pushed and climbed on; and the bananas, but not the box, can be grasped.

A plan:

walk(A,B); pushBox(B,C); climbUpBox(C); graspBananas(C)

Plan execution:

start
S_0 = {At(Mon,A), At(Box,B), At(Ban,C), On(Mon,Floor)}
walk(A,B)
S_1 = {At(Mon,B), At(Box,B), At(Ban,C), On(Mon,Floor)}
pushBox(B,C)
S_2 = {At(Mon,C), At(Box,C), At(Ban,C), On(Mon,Floor)}
climbUpBox(C)
S_3 = {At(Mon,C), At(Box,C), At(Ban,C), On(Mon,Box)}
graspBananas(C)
S_4 = {At(Mon,C), At(Box,C), At(Ban,C), On(Mon,Box), **Has(Mon,Ban)**}
end

References

Fikes, R., and N. Nilsson (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 189-208.