Introduction to planning

We now look at how an agent might construct a *plan* enabling it to achieve a goal.

Aims:

- to examine the difference between on the one hand, problem-solving by search, which we have already addressed, and on the other hand, specialised planning algorithms;
- to look in detail at the basic *partial-order planning algorithm*.

Reading: Russell and Norvig, chapter 11.

Copyright © Sean Holden 2002-2005.

Problem solving is different to planning

In search problems we:

- **Represent states:** and a state representation contains *everything* that's relevant about the environment.
- **Represent actions:** by describing a new state obtained from a current state.
- **Represent goals:** all we know is how to test a state either to see if it's a goal, or using a heuristic.
- **A sequence of actions is a 'plan':** but we only consider *sequences of consecutive actions*.

Problem solving is different to planning

Representing a problem such as: ‘obtain a copy of the course text book’ is hopeless:

- There are far too many possible actions at each step.
- A heuristic can only help you rank states. In particular it does not help you *ignore* useless actions.
- We are forced to start at the initial state, but you have to work out *how* to get the book—that is, go to the library, borrow it from a friend *etc*—*before* you can start to do it.

Planning algorithms work differently

Difference 1:

- planning algorithms use a language, often first order logic (FOL) (or a subset of FOL) to represent states, goals, and actions;
- states and goals are described by sentences;
- actions are described by stating their *preconditions* and their *effects*.

So if you know the goal includes (maybe among other things)

`Have(AI_book)`

and action `Borrow(x)` has an effect `Have(x)` then you know that a plan including

`Borrow(AI_book)`

might be good.

Planning algorithms work differently

Difference 2:

- Planners can add actions at *any relevant point at all*, not just at the end of a sequence starting at the start state.
- This makes sense: I may determine that `Have(Car_keys)` is a good state to be in without worrying about what happens before or after finding them.
- By making an important decision, like requiring `Have(Car_keys)`, early on we may reduce branching and backtracking.
- State descriptions are not complete—`Have(Car_keys)` describes a *class* of states—and this adds flexibility.

Difference 3:

It is assumed that most elements of the environment are *independent* of most other elements.

- A goal including several requirements can be attacked with a divide-and-conquer approach.
- Each individual requirement can be fulfilled using a subplan...
- ...and the subplans then combined.

This works provided there is not significant interaction between the subplans.

Running example: gorilla-based mischief

We will use the following simple example problem, which as based on a similar one due to Russell and Norvig.

The intrepid little scamps in the *Cambridge University Roof-Climbing Society* wish to attach an inflatable gorilla to the spire of a famous College. To do this they need to leave home and obtain:

- **An inflatable gorilla:** these can be purchased from all good joke shops.
- **Some rope:** available from a hardware store.
- **A first-aid kit:** also available from a hardware store.

They need to return home after they've finished their shopping.

How do they go about planning their jolly escapade?

The STRIPS language

STRIPS: “Stanford Research Institute Problem Solver” (1970).

States: are *conjunctions* of *ground literals* with *no functions*.

$$\begin{aligned}\texttt{At}(\texttt{Home}) \wedge \neg\texttt{Have}(\texttt{Gorilla})\\ \wedge \neg\texttt{Have}(\texttt{Rope})\\ \wedge \neg\texttt{Have}(\texttt{Kit})\end{aligned}$$

Goals: are *conjunctions* of *literals* where variables are assumed existentially quantified.

$$\texttt{At}(x) \wedge \texttt{Sells}(x, \texttt{Gorilla})$$

A planner finds a sequence of actions that makes the goal true when performed. This is different to a theorem-prover.

The STRIPS language

STRIPS uses *operators* specifying:

- An *action description*: what the action does.
- A *precondition*: what must be true before the operator can be used. A conjunction of positive literals.
- An *effect*: what is true after the operator has been used. A conjunction of literals.

The STRIPS language

For example:

$$\texttt{At}(x), \texttt{Path}(x,y)$$

$$\boxed{\texttt{Go}(y)}$$

$$\texttt{At}(y), \neg\texttt{At}(x)$$

$$
\begin{aligned}
&\text{Op}(\text{Action: } \texttt{Go}(y), \\
&\quad \text{Pre: } \texttt{At}(x) \land \texttt{Path}(x,y) \\
&\quad \text{Effect: } \texttt{At}(y) \land \neg\texttt{At}(x))
\end{aligned}
$$

All variables are universally quantified.

The space of situations

Standard search algorithms could be used with STRIPS to construct sequences of actions working forward from the start state. This is:

- a *situation space* planner;
- a *progression* planner. It searches from initial state to goal.

A *regression planner* exploits the new language by searching backward from the goal.

This can still be too inefficient.

The space of plans

Alternatively we can search in *plan space*:

- start with an empty plan;
- operate on it to obtain new plans;
- continue until we obtain a plan that solves the problem.

Operations on plans can be:

- adding a step;
- instantiating a variable;
- imposing an ordering that places a step in front of another;
- and so on.

The space of plans

Incomplete plans are called *partial plans*.

Refinement operators add constraints to a partial plan.

All other operators are called *modification operators*.

Representing a plan: partial order planners

When putting on your shoes and socks:

- it *does not matter* whether you deal with your left or right foot first;
- it *does matter* that you place a sock on *before* a shoe, for any given foot.

It makes sense in constructing a plan, *not* to make any *commitment* to which side is done first *if you don't have to*.

Representing a plan: partial order planners

Principle of least commitment: do not commit to any specific choices until you have to. This can be applied both to ordering and to instantiation of variables.

A *partial order planner* allows plans to specify that some steps must come before others but others have no ordering.

A *linearision* of such a plan imposes a specific sequence on the actions therein.

Representing a plan: partial order planners

A plan consists of:

1. A set $\{S_1, S_2, \ldots, S_n\}$ of steps. Each of these is one of the available operators.
2. A set of *ordering constraints*. An ordering constraint $S_i < S_j$ denotes the fact that step S_i must happen before step S_j. $S_i < S_j < S_k$ and so on has the obvious meaning. $S_i < S_j$ does *not* mean that S_i must *immediately* precede S_j.
3. A set of variable bindings $v = x$ where v is a variable and x is either a variable or a constant.
4. A set of *causal links* or *protection intervals* $S_i \xrightarrow{c} S_j$. This denotes the fact that the purpose of S_i is to achieve the precondition c for S_j.

Representing a plan: partial order planners

The *initial plan* has:

- two steps, called `Start` and `Finish`;
- a single ordering constraint `Start` $<$ `Finish`;
- no variable bindings;
- no causal links.

In addition to this:

- the step `Start` has no preconditions, and its effect is the start state for the problem;
- the step `Finish` has no effect, and its precondition is the goal;
- neither `Start` or `Finish` has an associated action.

Solutions to planning problems

A solution to a planning problem is any *complete* and *consistent* partially ordered plan.

Complete: each precondition of each step is *achieved* by another step in the solution.

A precondition c for S is achieved by a step S' if:

1. the precondition is an effect of the step

   $$S' < S \text{ and } c \in \text{Effects}(S')$$

   and;

2. there is no *other* step that can cancel the precondition:

   $$\text{no } S'' \text{ exists where } S' < S'' < S \text{ and } \neg c \in \text{Effects}(S'')$$

Solutions to planning problems

Consistent: no contradictions exist in the binding constraints or in the proposed ordering. That is:

1. for binding constraints, we never have $v = X$ and $v = Y$ for distinct constants X and Y;
2. for the ordering, we never have $S < S'$ and $S' < S$.

An example of partial-order planning

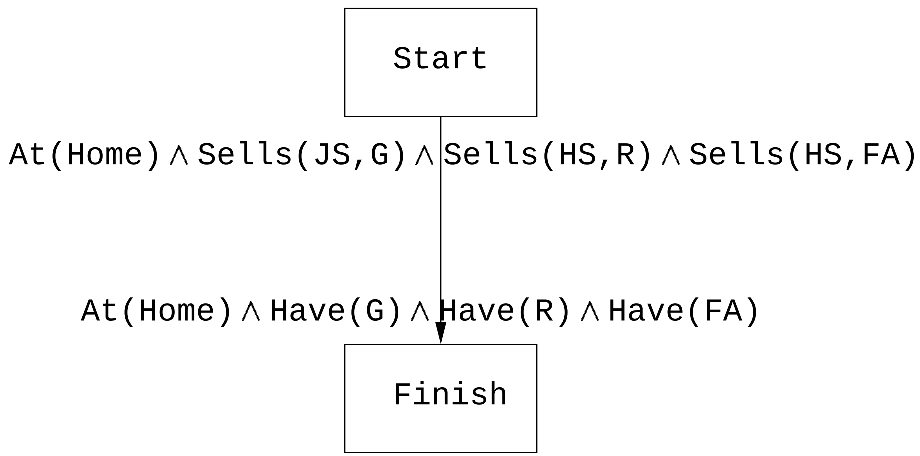
Returning to the roof-climber's shopping expedition.

Here is the basic approach:

- start with only the `Start` and `Finish` steps in the plan;
- at each stage add a new step;
- always add a new step such that a currently non-achieved pre-condition is achieved;
- backtrack when necessary.

An example of partial-order planning

Here is the initial plan:



Thin arrows denote ordering.

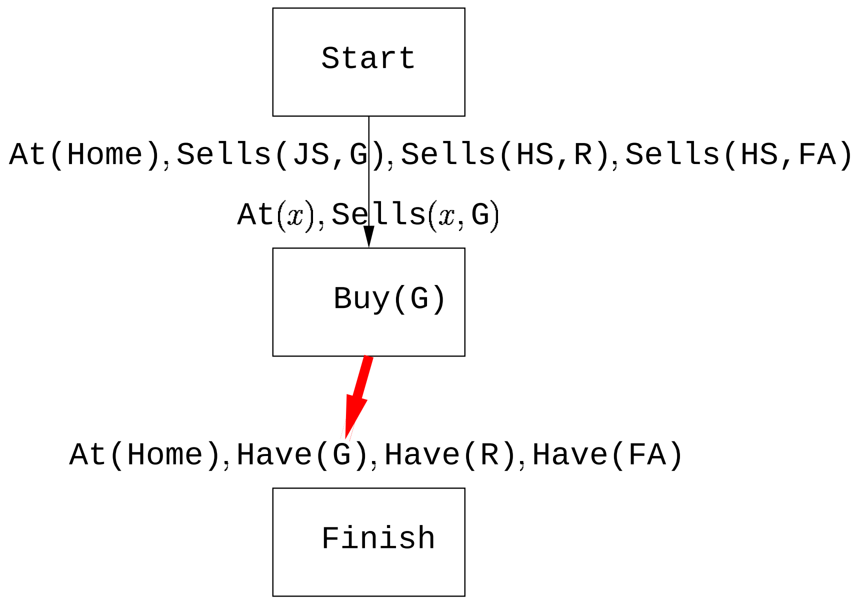
An example of partial-order planning

There are two actions available:

A planner might begin, for example, by adding a `Buy(G)` action in order to achieve the `Have(G)` precondition of `Finish`.

Note: the following order of events is by no means the only one available to a planner. It has been chosen for illustrative purposes.

An example of partial-order planning



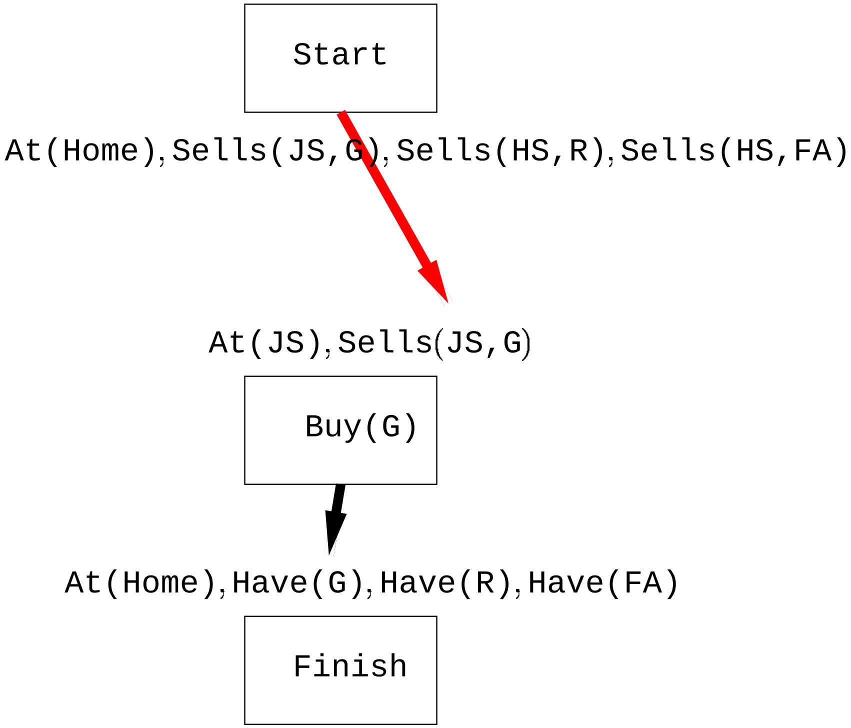
Thick arrows denote causal links.

Here, the new `Buy` step achieves the `Have(G)` precondition of `Finish`.

Thick arrows can be thought of as having a thin arrow underneath.

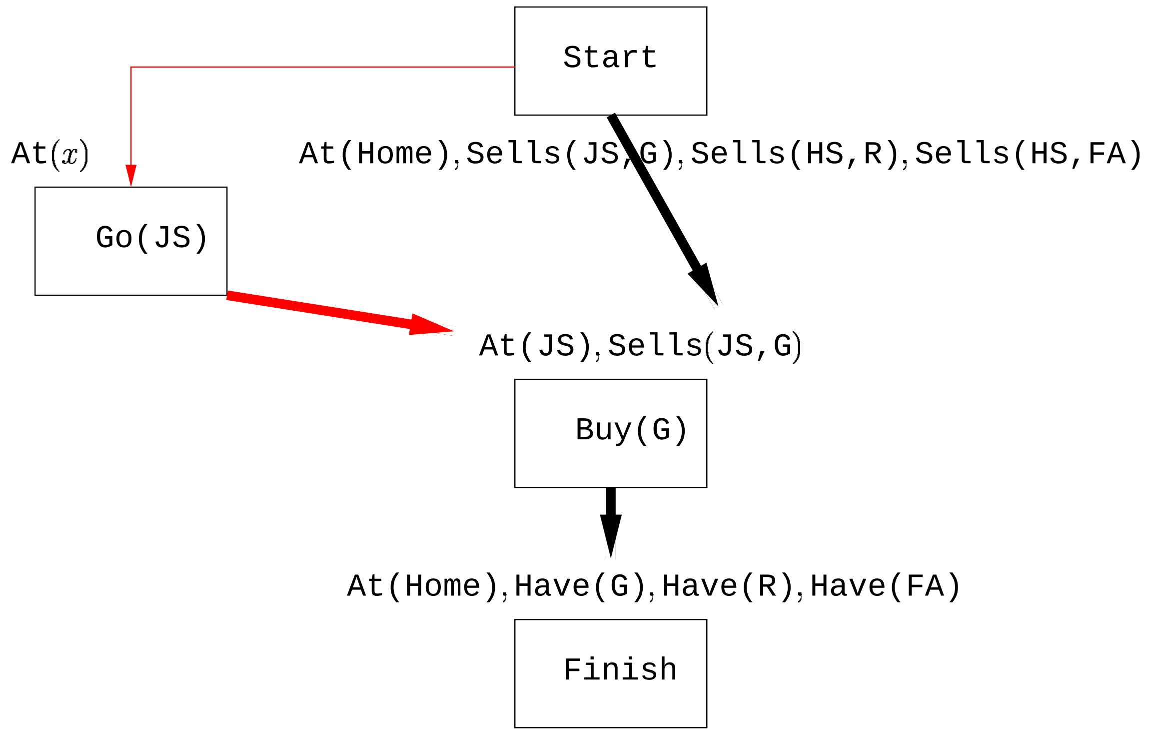
An example of partial-order planning

The planner can now introduce a second causal link from `Start` to achieve the `Sells`$(x,$ `G`$)$ precondition of `Buy(G)`.



An example of partial-order planning

The planner's next obvious move is to introduce a `Go` step to achieve the `At(HS)` precondition of `Buy(G)`.

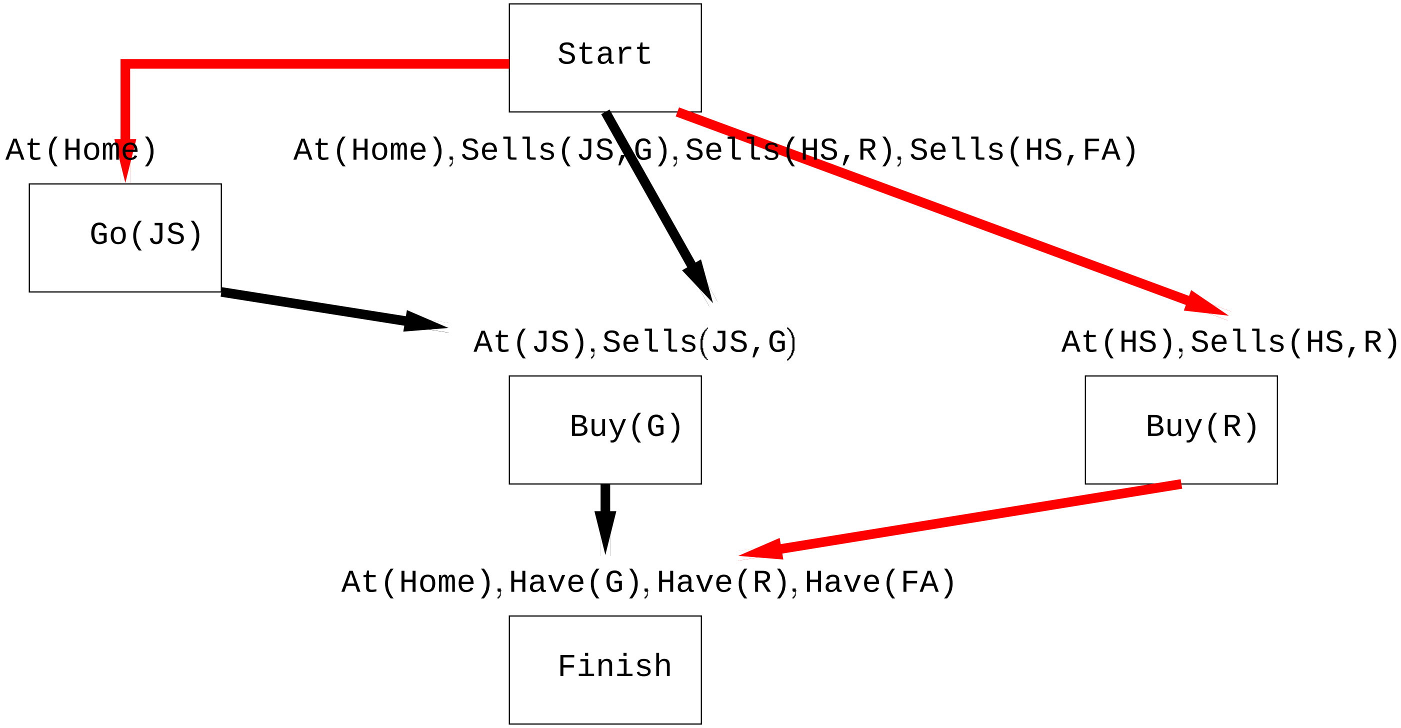


An example of partial-order planning

Initially the planner can continue quite easily in this manner:

- Add a causal link from `Start` to `Go(JS)` to achieve the `At`(x) precondition.
- Add the step `Buy(R)` with an associated causal link to the `Have(R)` precondition of `Finish`.
- Add a causal link from `Start` to `Buy(R)` to achieve the `Sells(HS,R)` precondition.

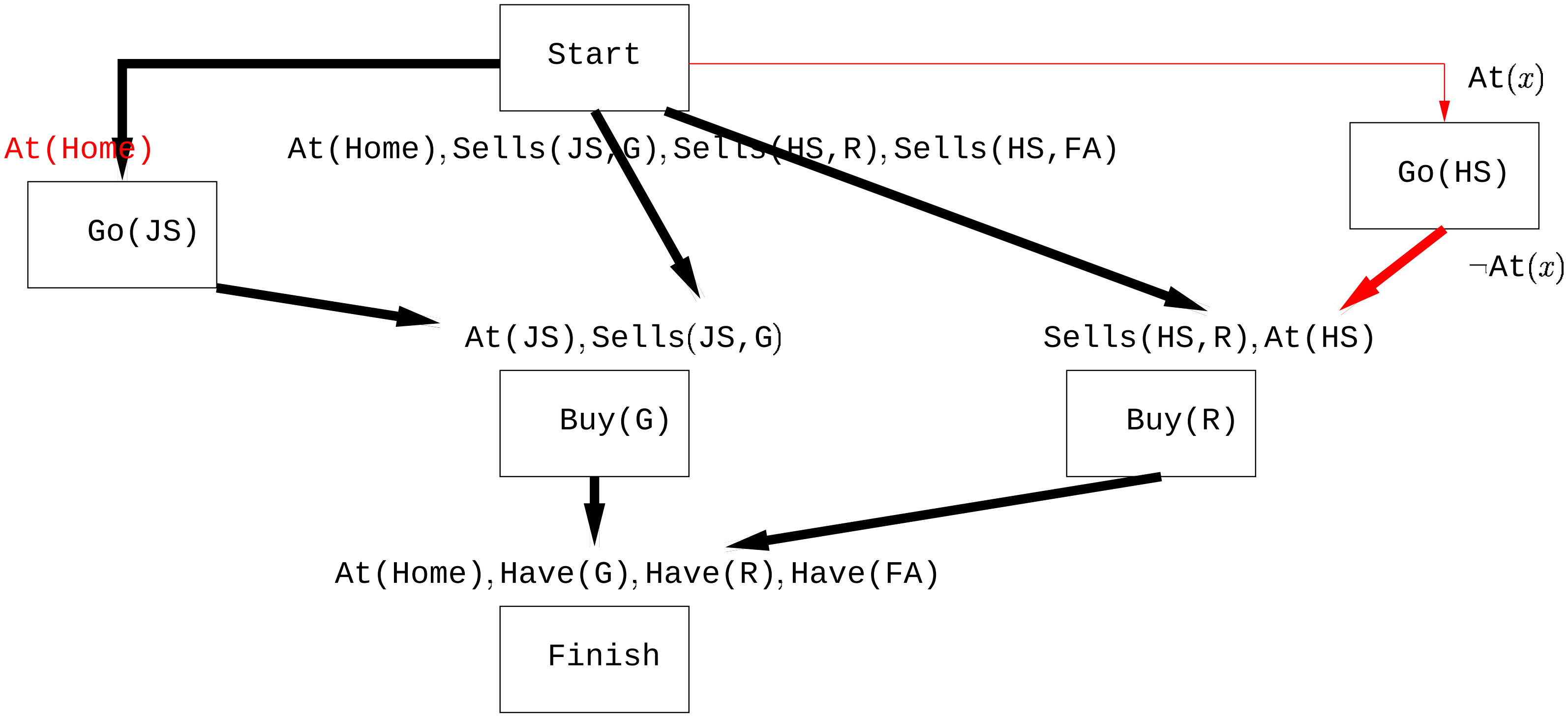
An example of partial-order planning



At this point it starts to get tricky...

The `At(HS)` precondition in `Buy(R)` is not achieved.

An example of partial-order planning



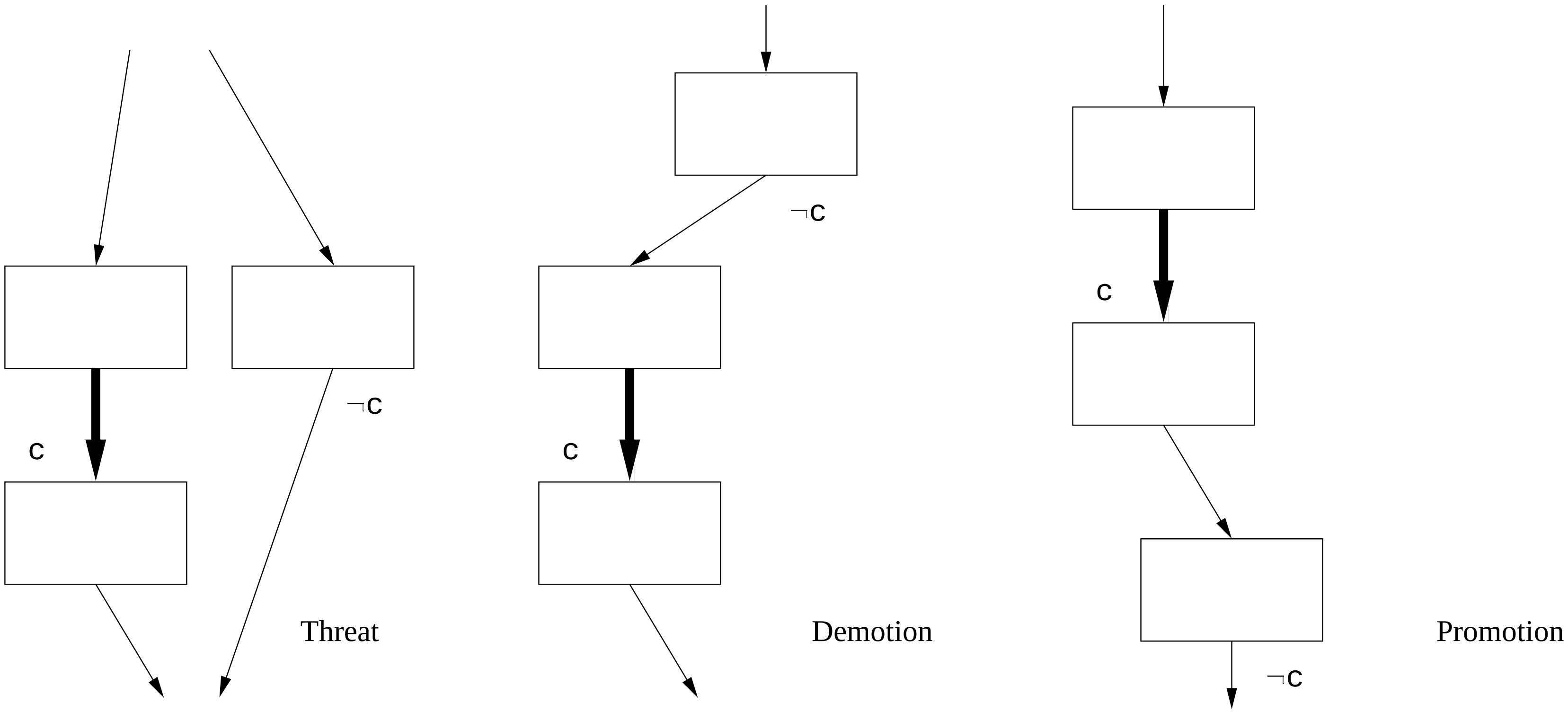
The `At(HS)` precondition is easy to achieve.

But if we introduce a causal link from `Start` *to* `Go(HS)` *then we risk invalidating the precondition for* `Go(JS)`.

An example of partial-order planning

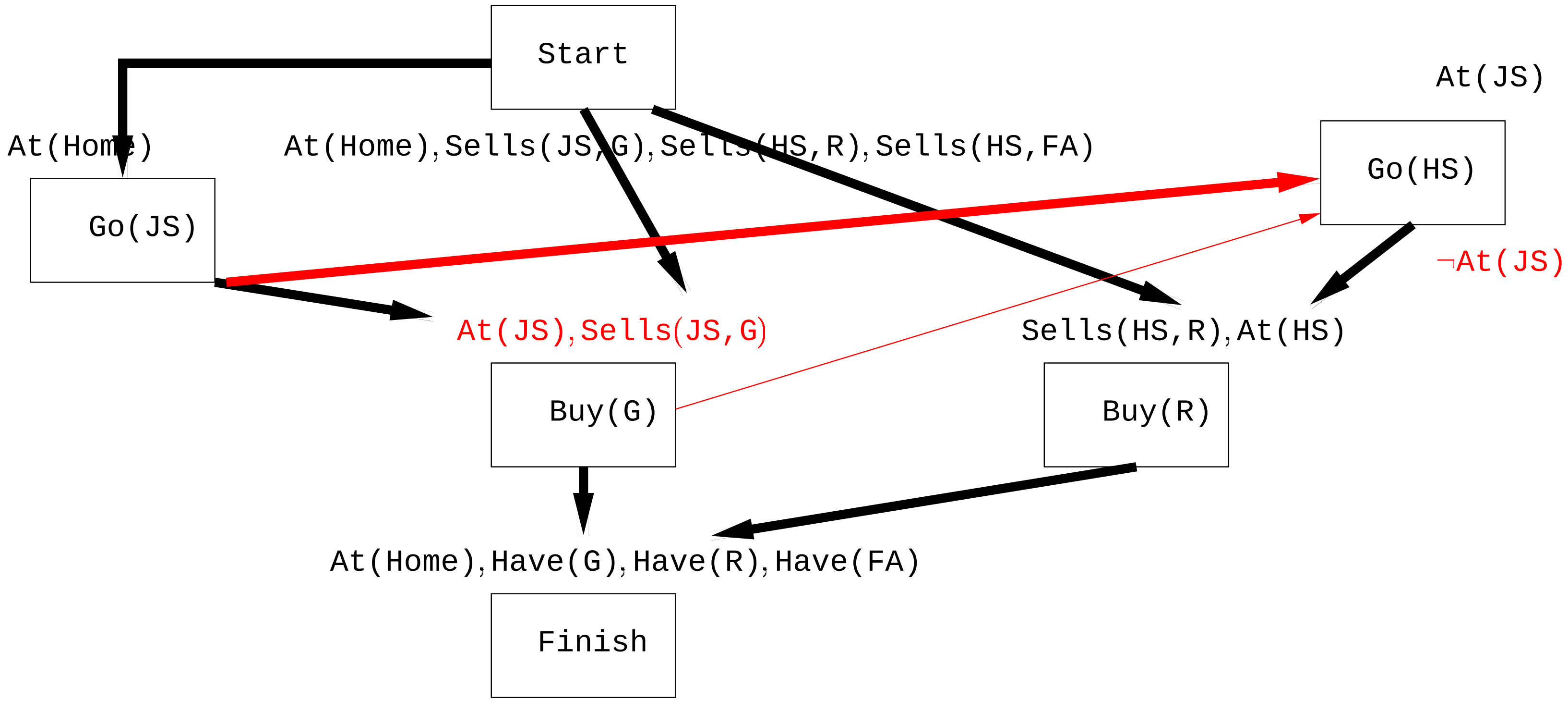
A step that might invalidate (sometimes the word *clobber* is employed) a previously achieved precondition is called a *threat*.

A planner can try to fix a threat by introducing an ordering constraint.



An example of partial-order planning

The planner could backtrack and try to achieve the At(x) precondition using the existing Go(JS) step.



This involves a threat, but one that can be fixed using promotion.

The algorithm

```
plan partial_order_plan(start,finish,ops)
{
  plan=empty_plan(start,finish);

  while(true)
  {
    if (solution(plan))
      return plan;
    else
    {
      (step,pre)=get_subgoal(plan);
      choose_op(plan,ops,step,pre);
      resolve_threats(plan);
    }
  }
}
```

The algorithm

```
(step,pre) get_subgoal(plan)
{
  pick some step from steps in plan for which
  a precondition pre is not yet achieved;

  return (step,pre);
}
```

The algorithm

```
choose_op(plan,ops,step,pre)
{
  choose S from ops or current steps in plan
  having effect pre;

  if (no S exists)
    fail;
  include a causal link from S to step in the plan;
  include S < step in the plan;
  if(S doesn’t yet appear in the plan)
  {
    add S;
    add Start < S < Finish;
  }
}
```

The algorithm

```
resolve_threats(plan)
{
  for (all steps S threatening some causal link from
  S’ to S’’)
  {
    choose
      1. add S < S’ to the plan (promotion)
      2. add S’’ < S’ to the plan (demotion)

    if (the plan is not consistent)
      fail;
  }
}
```

Possible threats

If at any stage an effect $\neg$`At`(x) appears, is it a threat to `At(JS)`?

Such an occurrence is called a *possible threat* and an algorithm can be made to deal with it in three different ways:

1. use an equality constraint to resolve immediately;
2. use an inequality constraint to resolve immediately;
3. leave the choice of x's value until later.