Planning algorithms

Reading: AIMA, chapter 11.

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*.

Search algorithms are good for solving problems that fit this framework. However for more complex problems they may fail completely...

Problem solving is different to planning

Representing a problem such as: *'go out and buy some pies'* is hopeless:

- There are *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 pies*—that is, go to town and buy them, get online and find a web site that sells pies *etc*—*before you can start to do it*.

Knowledge representation and reasoning might not help either: although we end up with a sequence of actions—a plan—there is so much flexibility that complexity might well become an issue.

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

- We look at how we might update our concept of *knowledge representation and reasoning* to apply more specifically to planning tasks.
- We look in detail at the *partial-order planning algorithm*.

Planning algorithms work differently

Difference 1:

- Planning algorithms use a *special purpose language*—often based on FOL or a subset— to represent states, goals, and actions.
- States and goals are described by sentences, as might be expected, but...
- ...actions are described by stating their *preconditions* and their *effects*.

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

$$\texttt{Have}(\texttt{pie})$$

and action $\texttt{Buy}(x)$ has an effect $\texttt{Have}(x)$ then you know that a plan *including*

$$\texttt{Buy}(\texttt{pie})$$

might be reasonable.

Planning algorithms work differently

Difference 2:

- Planners can add actions at *any relevant point at all between the start and the goal*, not just at the end of a sequence starting at the start state.
- This makes sense: I may determine that `Have(carKeys)` 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(carKeys)` early on we may reduce branching and backtracking.
- State descriptions are not complete—`Have(carKeys)` describes a *class of states*—and this adds flexibility.

So: you have the potential to search both *forwards* and *backwards* within the same problem.

Planning algorithms work differently

Difference 3:

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

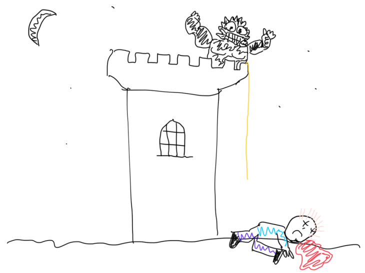
- A goal including several requirements can be attacked with a divide-and-conquer approach.
- Each individual requirement can be fulfilled using a subplan...
- ...and the subplans then combined.

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

Remember: the *frame problem*.

Running example: gorilla-based mischief

We will use a simple example, based on one from 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*. They must not include *function symbols*.

$$\begin{aligned}\texttt{At}(\texttt{home}) &\wedge \neg\texttt{Have}(\texttt{gorilla}) \\ &\wedge \neg\texttt{Have}(\texttt{rope}) \\ &\wedge \neg\texttt{Have}(\texttt{kit})\end{aligned}$$

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

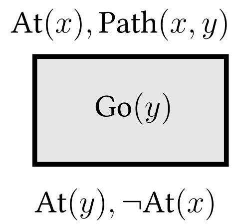
$$\texttt{At}(x) \wedge \text{Sells}(x, \texttt{gorilla})$$

A planner finds a sequence of actions that when performed makes the goal true.

We are no longer employing a full theorem-prover.

The STRIPS language

STRIPS represents actions using *operators*. For example



$$\text{Op}(\text{Action: Go}(y), \text{Pre: At}(x) \wedge \text{Path}(x, y), \text{Effect: At}(y) \wedge \neg\text{At}(x))$$

All variables are implicitly universally quantified. An operator has:

- 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 space of plans

We now make a change in perspective—we search in *plan space*:

- Start with an *empty plan*.
- *Operate on it* to obtain new plans. Incomplete plans are called *partial plans*. *Refinement operators* add constraints to a partial plan. All other operators are called *modification operators*.
- 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...

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*.

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 *linearisation* 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.

A causal link is *always* paired with an equivalent ordering constraint.

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.

We now need to consider what constitutes a *solution*...

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 *could* cancel the precondition. That is, no S'' exists where:
   - The existing ordering constraints allow S'' to occur *after* S' but *before* S.
   - $\neg c \in \text{Effects}(S'')$.

Solutions to planning problems

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

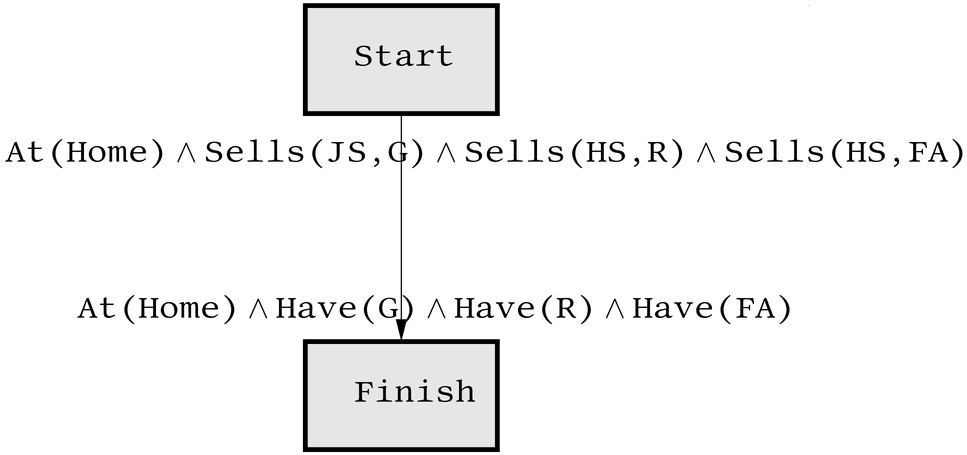
1. For binding constraints, we never have $v = X$ and $v = Y$ for distinct constants X and Y.
2. For the ordering, we never have $S < S'$ and $S' < S$.

Returning to the roof-climbers' shopping expedition, here is the basic approach:

- Begin 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 precondition 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*:

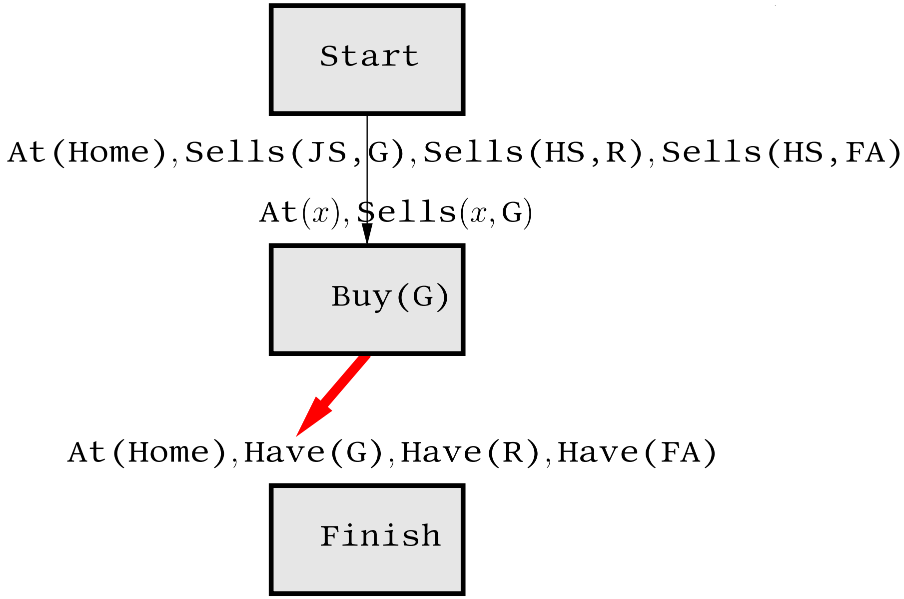
A planner might begin, for example, by adding a Buy(G) action in order to achieve the Have(G) precondition of Finish.

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

Incorporating the suggested step into the plan:

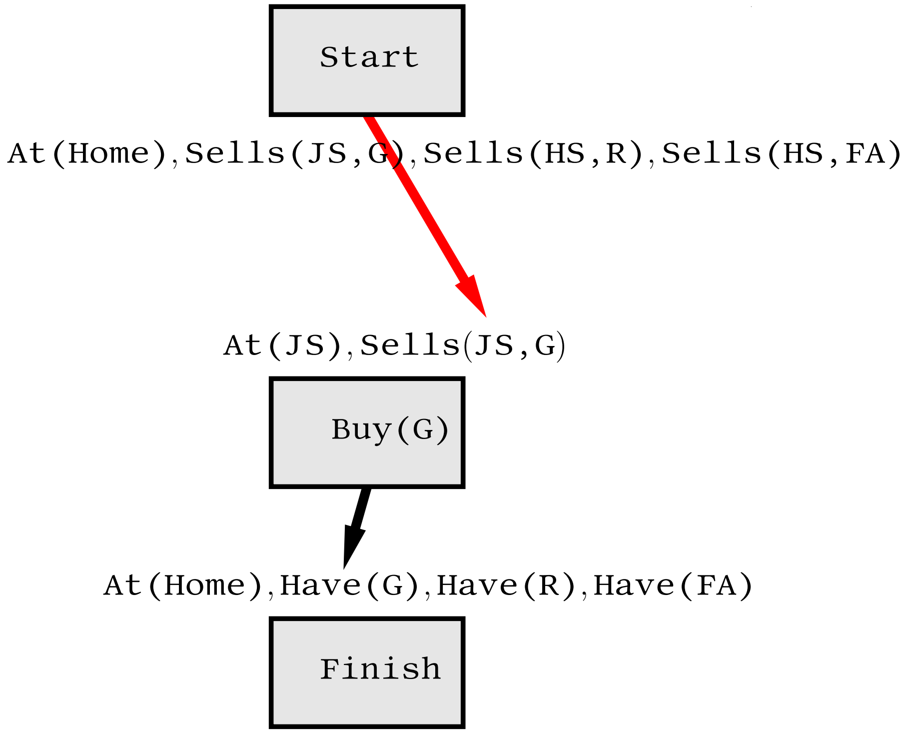


Thick arrows denote causal links. They always have a thin arrow underneath.

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

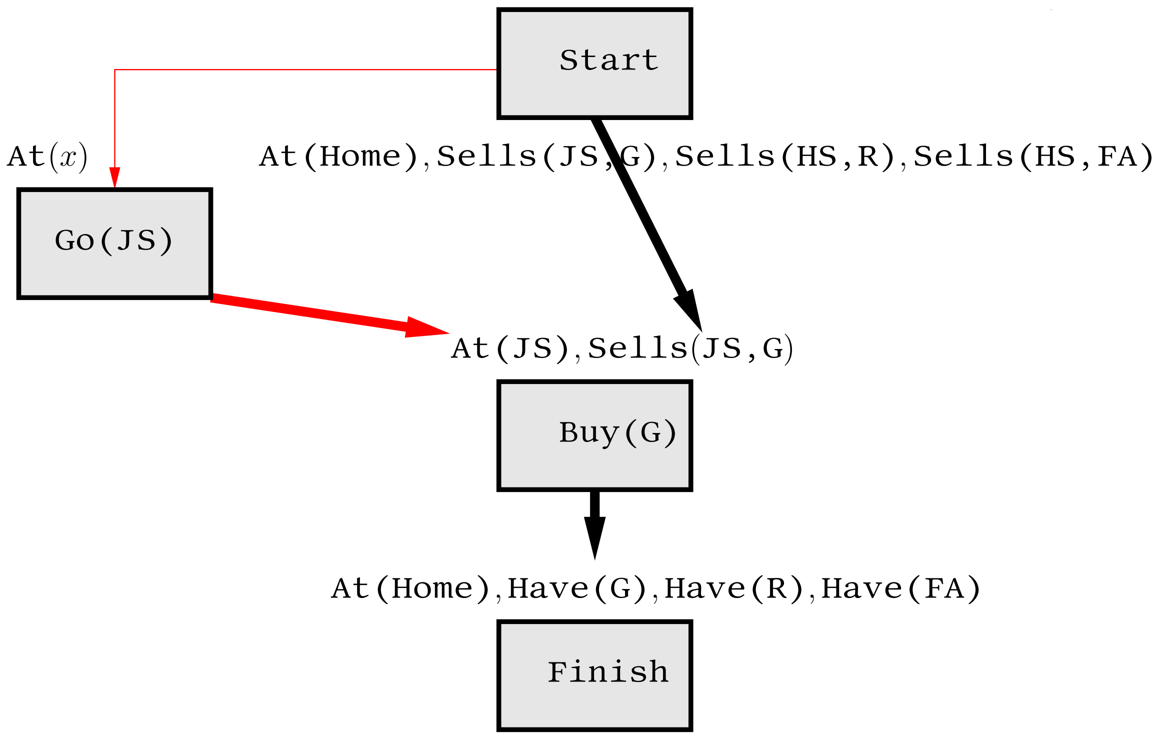
An example of partial-order planning

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(JS) precondition of Buy(G).



And we continue...

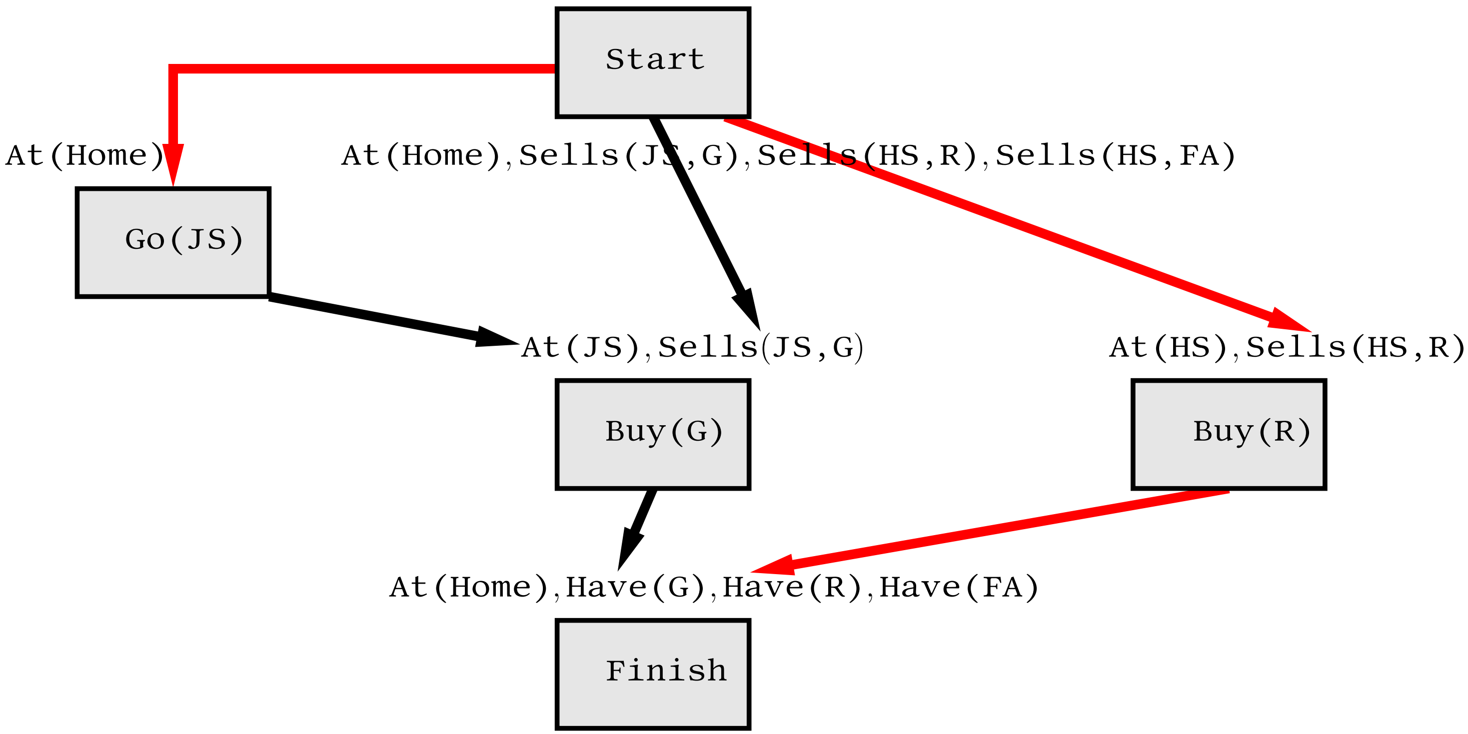
An example of partial-order planning

Initially the planner can continue quite easily in this manner:

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

But then things get more interesting...

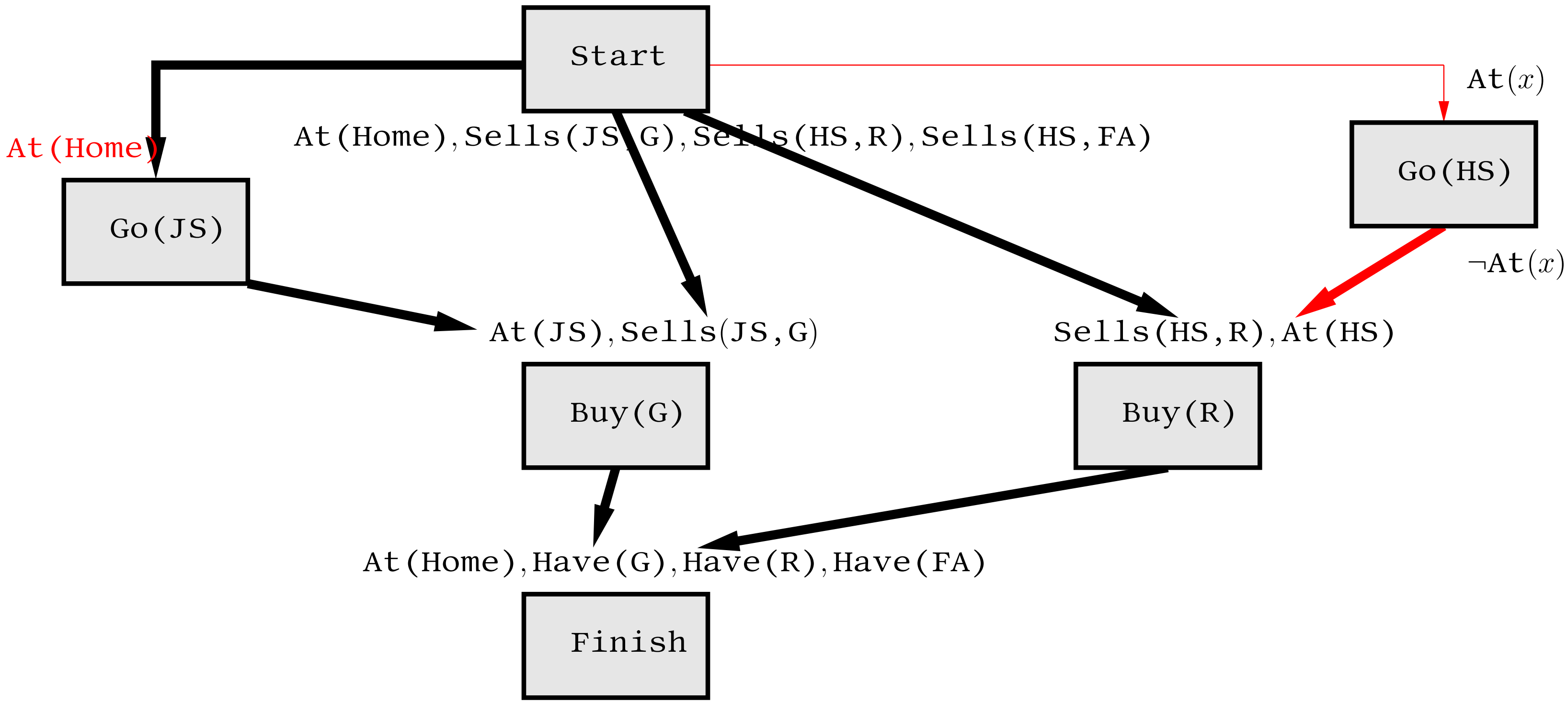
An example of partial-order planning



At this point it starts to get tricky...

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

An example of partial-order planning

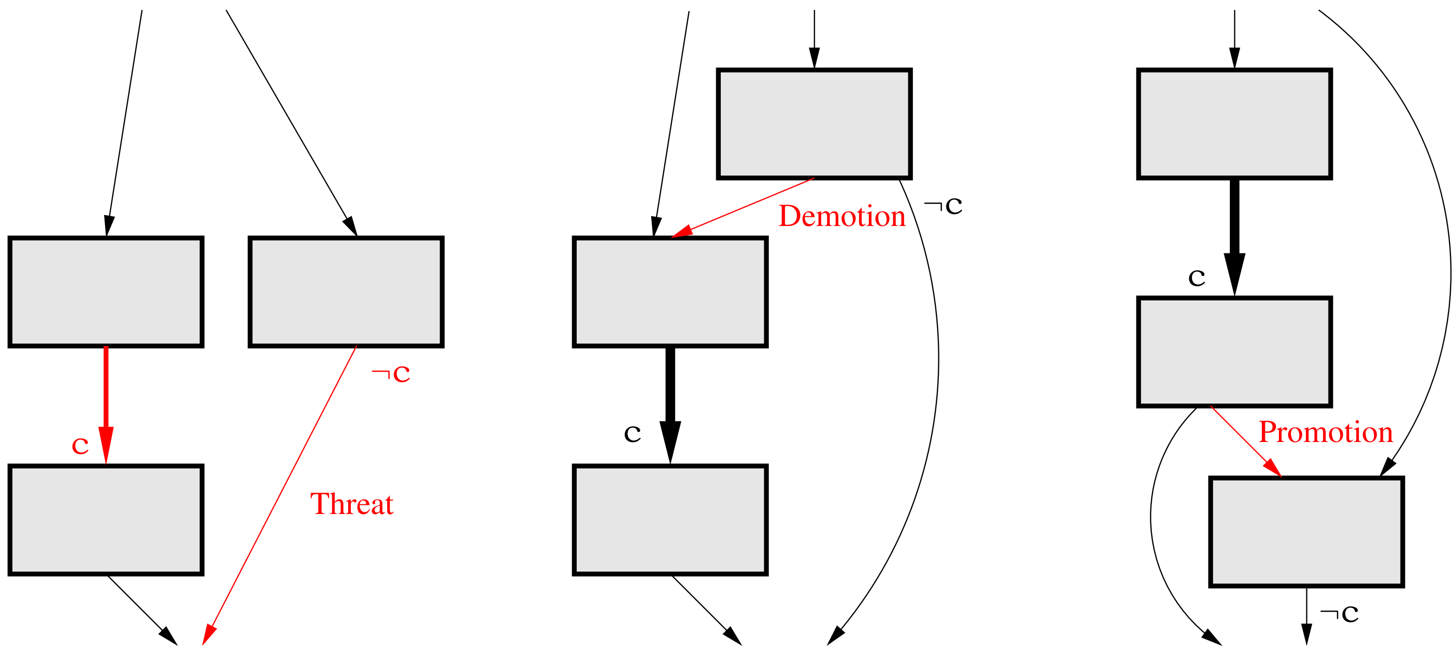


The At(HS) precondition is easy to achieve.

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

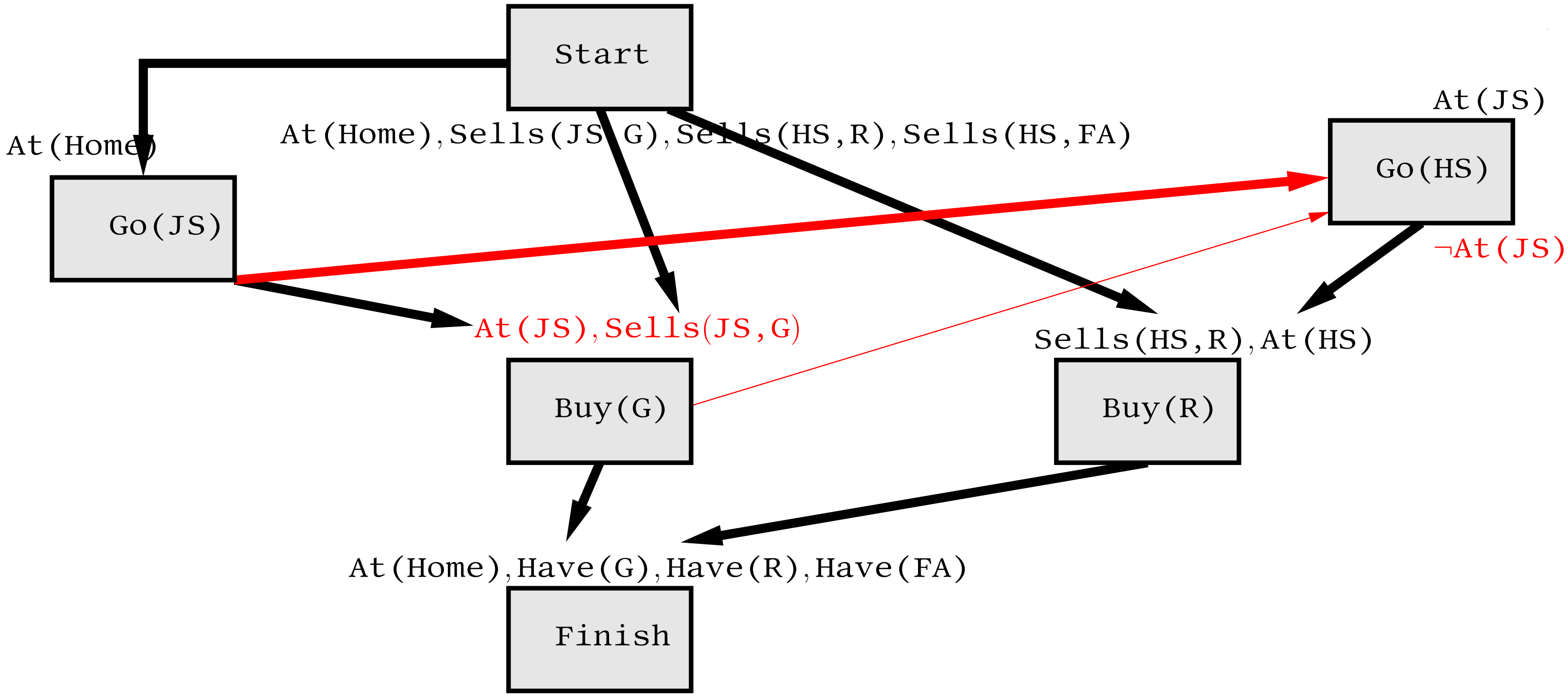
A step that might invalidate (sometimes the word *clobber* is employed) a previously achieved precondition is called a *threat*.



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

Simplifying slightly to the case where there are *no variables*.

Say we have a partially completed plan and a set of the preconditions that have yet to be achieved.

- Select a precondition p that has not yet been achieved and is associated with an action B.
- At each stage *the partially complete plan is expanded into a new collection of plans*.
- To expand a plan, we can try to achieve p *either* by using an action that's already in the plan or by adding a new action to the plan. In either case, call the action A.

We then try to construct consistent plans where A achieves p.

The algorithm

This works as follows:

- For *each possible way of achieving p*:
  - Add $\text{Start} < A$, $A < \text{Finish}$, $A < B$ and the causal link $A \xrightarrow{p} B$ to the plan.
  - If the resulting plan is consistent we're done, otherwise *generate all possible ways of removing inconsistencies* by promotion or demotion and *keep any resulting consistent plans*.

At this stage:

- If you have *no further preconditions that haven't been achieved* then *any plan obtained is valid*.

The algorithm

But how do we try to *enforce consistency*?

When you attempt to achieve p using A:

- Find all the existing causal links $A' \xrightarrow{\neg p} B'$ that are *clobbered* by A.
- For each of those you can try adding $A < A'$ or $B' < A$ to the plan.
- Find all existing actions C in the plan that clobber the *new* causal link $A \xrightarrow{p} B$.
- For each of those you can try adding $C < A$ or $B < C$ to the plan.
- Generate *every possible combination* in this way and retain any consistent plans that result.

Possible threats

What about dealing with *variables*?

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

Such an occurrence is called a *possible threat* and we can deal with it by introducing *inequality constraints*: in this case $x \neq \texttt{JS}$.

- Each partially complete plan now has a set I of inequality constraints associated with it.
- An inequality constraint has the form $v \neq X$ where v is a variable and X is a variable or a constant.
- Whenever we try to make a substitution we check I to make sure we won't introduce a conflict.

If we *would* introduce a conflict then we discard the partially completed plan as inconsistent.

Unsurprisingly, this process can become complex.

How might we improve matters?

One way would be to introduce *heuristics*. We now consider:

- The way in which *basic heuristics* might be defined for use in planning problems.
- The construction of *planning graphs* and their use in obtaining more sensible heuristics.
- Planning graphs as the basis of the *GraphPlan* algorithm.

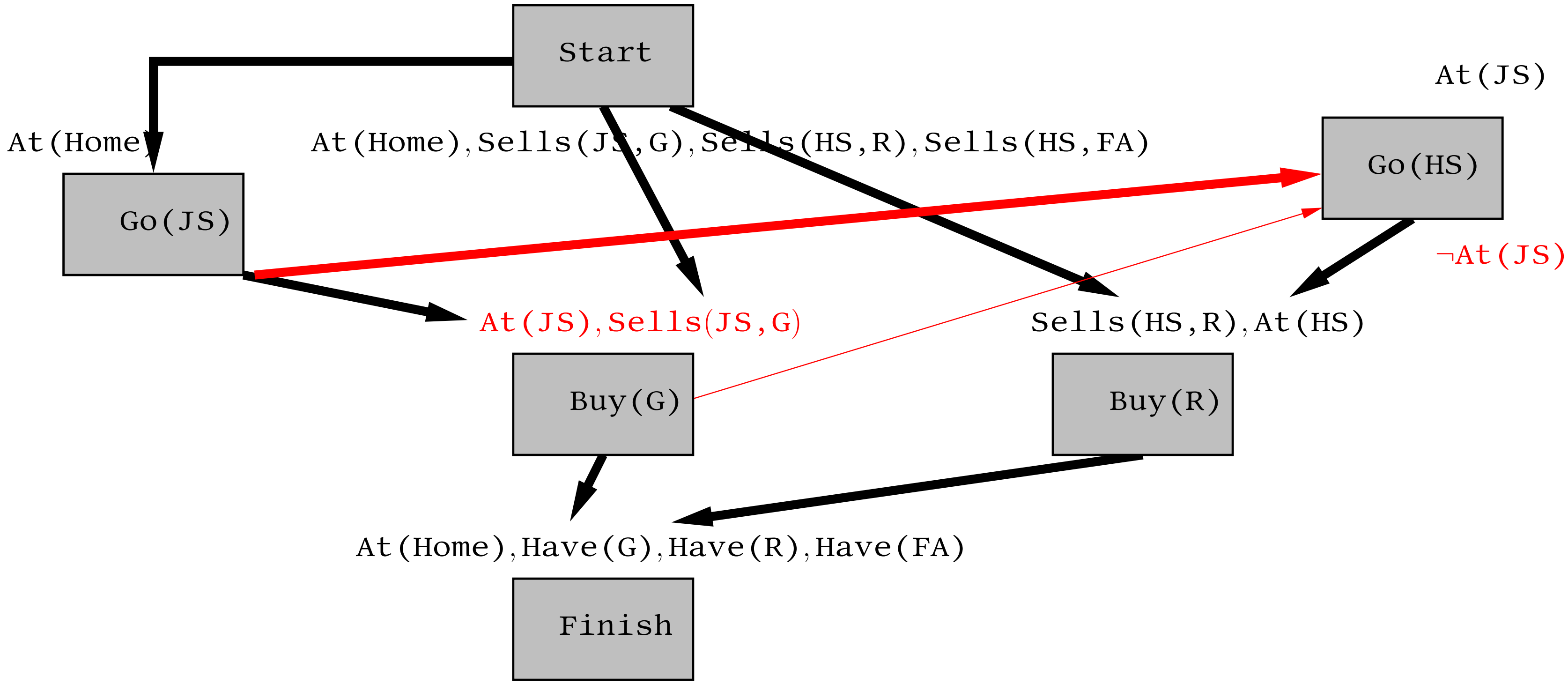
Another is to translate into the language of a *general-purpose* algorithm exploiting its own heuristics. We now consider:

- Planning using *propositional logic*.
- Planning using *constraint satisfaction*.

An example of partial-order planning

We left our example problem here:

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



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

Using heuristics in planning

We found in looking at search problems that *heuristics* were a helpful thing to have.

Note that now there is no simple representation of a *state*, and consequently it is harder to measure the *distance to a goal*.

Defining heuristics for planning is therefore more difficult than it was for search problems. Simple possibilities:

$$h = \text{number of unsatisfied preconditions}$$

or

$$\begin{aligned} h =& \text{number of unsatisfied preconditions} \\ & - \text{number satisfied by the start state} \end{aligned}$$

These can lead to underestimates or overestimates:

- Underestimates if *actions can affect one another in undesirable ways*.
- Overestimates if *actions achieve many preconditions*.

Using heuristics in planning

We can go a little further by learning from *Constraint Satisfaction Problems* and adopting the *most constrained variable* heuristic:

- Prefer the precondition *satisfiable in the smallest number of ways*.

This can be computationally demanding but two special cases are helpful:

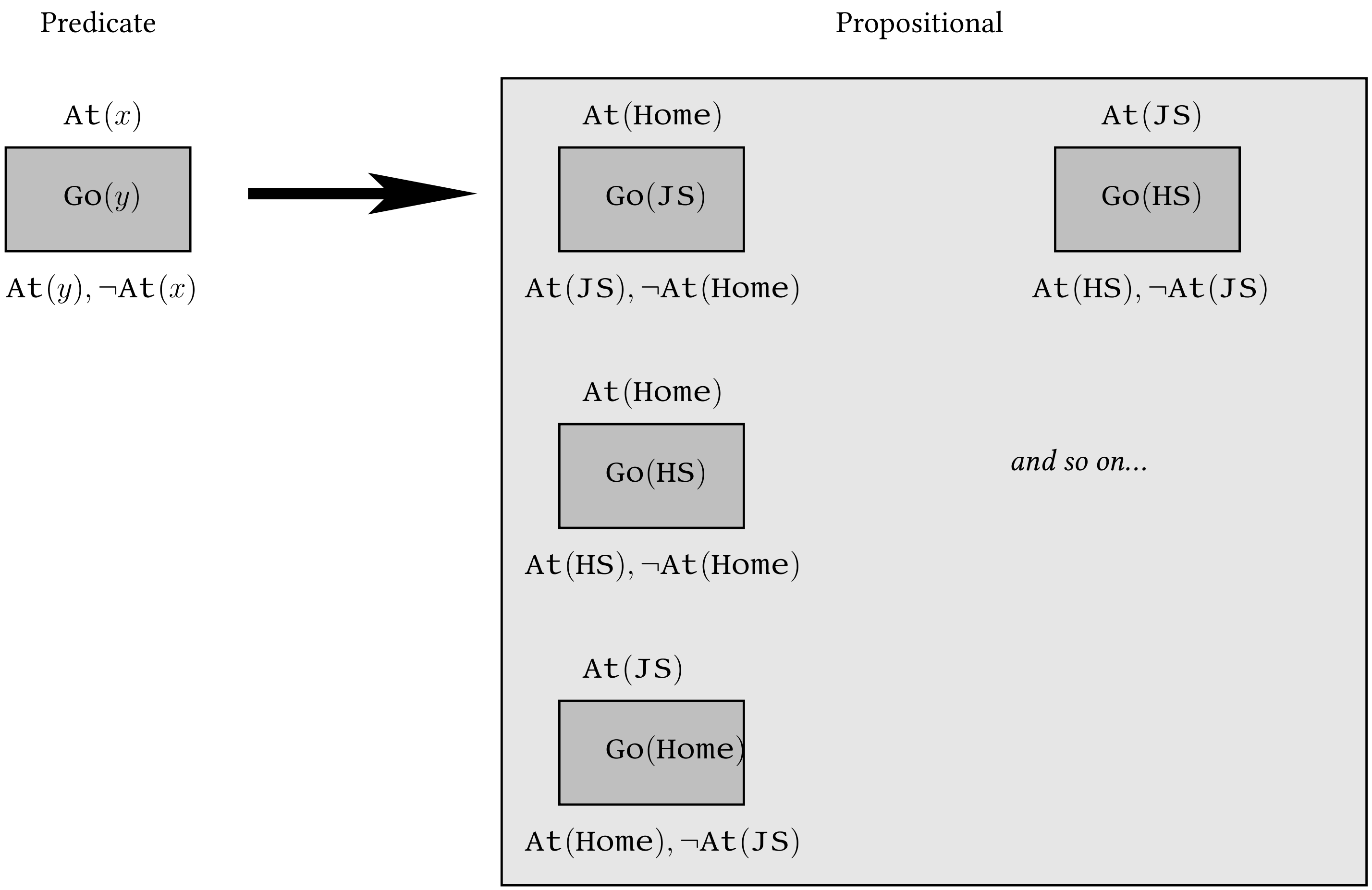
- Choose preconditions for which *no action will satisfy them*.
- Choose preconditions that *can only be satisfied in one way*.

But these still seem somewhat basic.

We can do better using *Planning Graphs*. These are *easy to construct* and can also be used to generate *entire plans*.

Planning graphs

Planning Graphs apply when it is possible to work entirely using *propositional* representations of plans. Luckily, STRIPS can always be propositionalized...



Planning graphs

A planning graph is constructed in levels:

- Level 0 corresponds to the *start state*.
- At each level we keep *approximate* track of all things that *could* be true at the corresponding time.
- At each level we keep *approximate* track of what actions *could* be applicable at the corresponding time.

The approximation is due to the fact that not all conflicts between actions are tracked. *So*:

- The graph can *underestimate* how long it might take for a particular proposition to appear, and therefore …
- …a heuristic can be extracted.

For example: the triumphant return of the gorilla-purchasing roof-climbers...

Planning graphs: a simple example

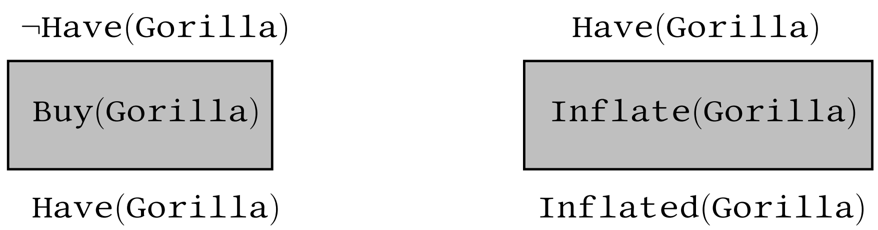
Our intrepid student adventurers will of course need to inflate their *gorilla* before attaching it to a *distinguished roof*. It has to be purchased before it can be inflated.

Start state: Empty.

We assume that anything not mentioned in a state is false. So the state is actually

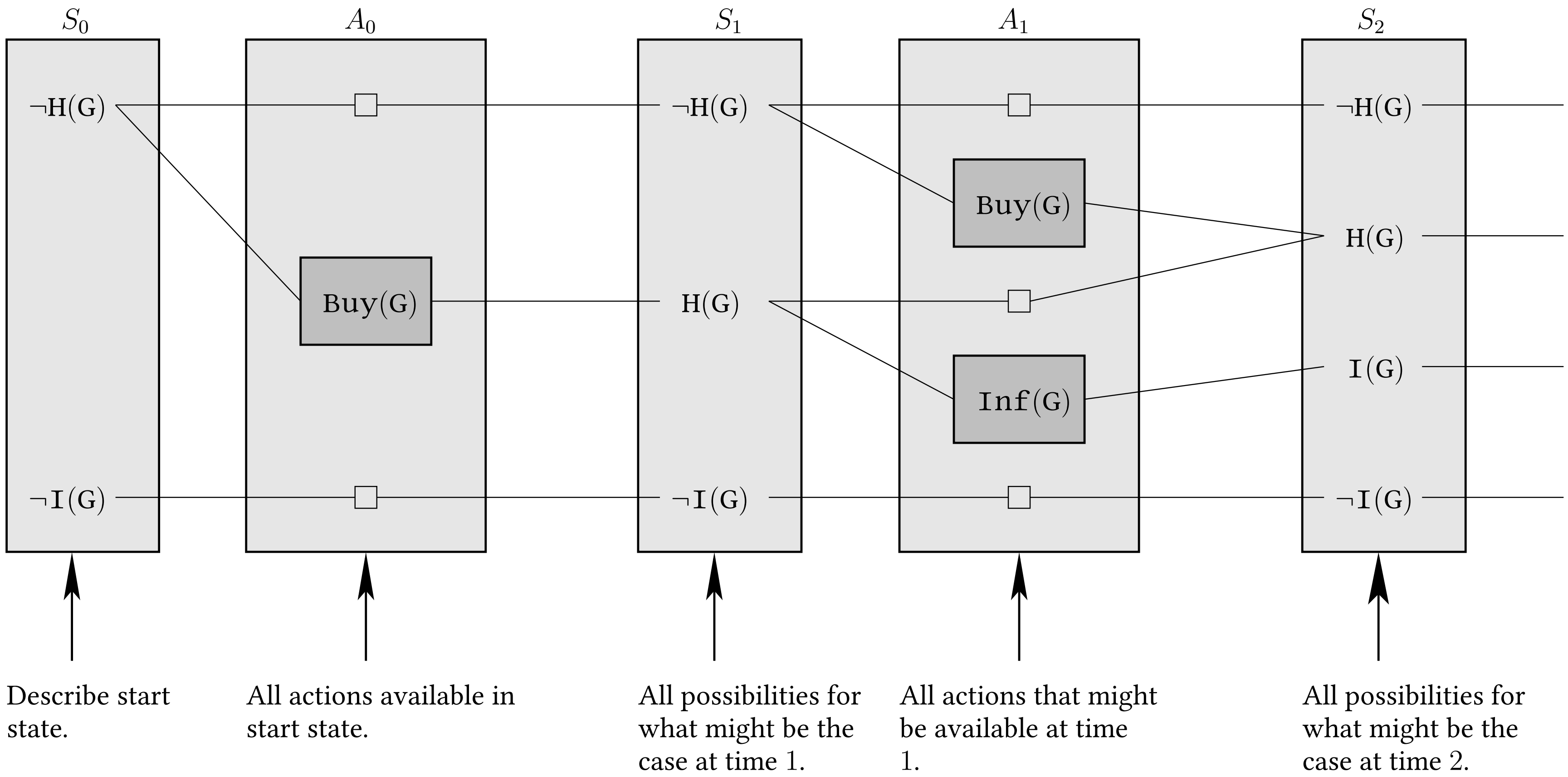
$$\neg\texttt{Have}(\texttt{Gorilla}) \text{ and } \neg\texttt{Inflated}(\texttt{Gorilla})$$

Actions:



Goal: $\texttt{Have}(\texttt{Gorilla})$ and $\texttt{Inflated}(\texttt{Gorilla})$.

Planning graphs



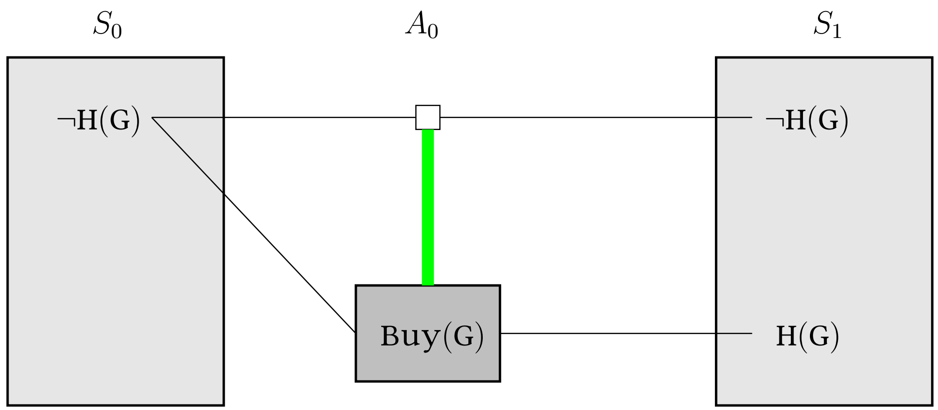
□ = a *persistence action*—what happens if no action is taken.

An action level A_i contains *all* actions that *could* happen given the propositions in S_i.

Mutex links

We also record, using *mutual exclusion (mutex) links* which pairs of actions could not occur together.

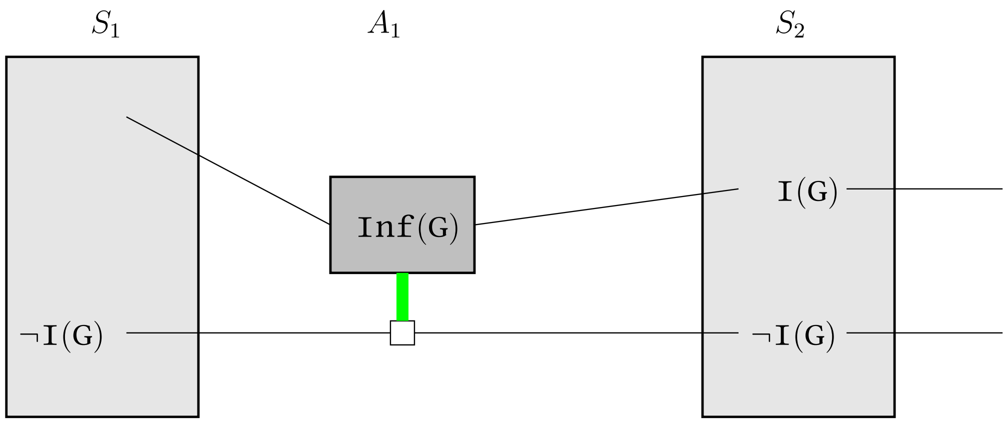
Mutex links 1: Effects are inconsistent.



The effect of one action negates the effect of another.

Mutex links

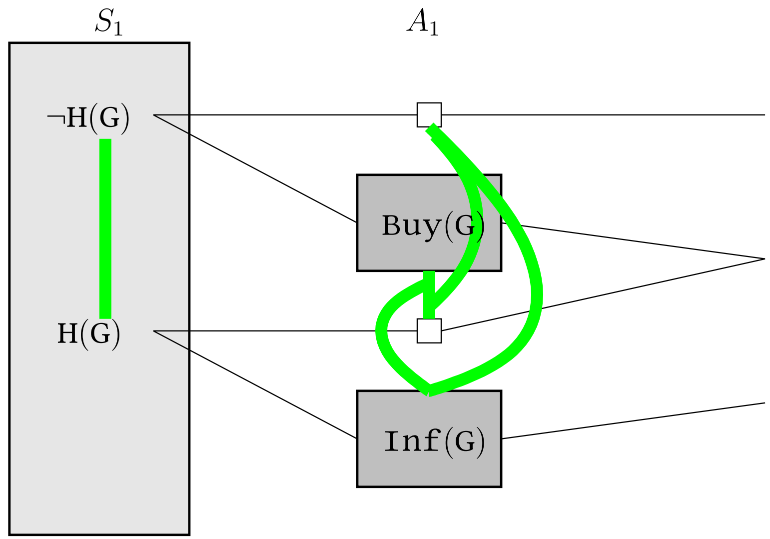
Mutex links 2: The actions interfere.



The effect of an action negates the precondition of another.

Mutex links

Mutex links 3: Competing for preconditions.



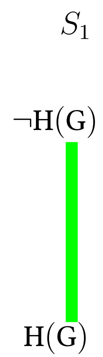
The precondition for an action is mutually exclusive with the precondition for another. (See next slide!)

Mutex links

A state level S_i contains *all* propositions that *could* be true, given the possible preceding actions.

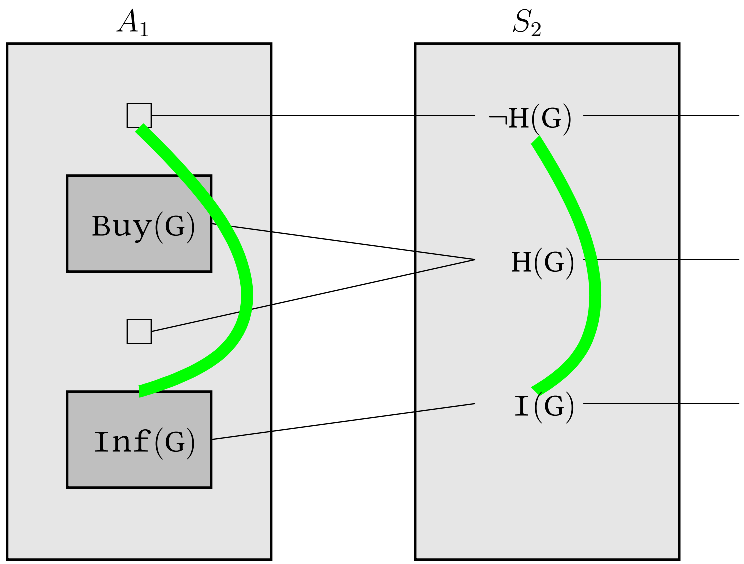
We also use mutex links to record pairs that can not be true simultaneously:

Possibility 1: pair consists of a proposition and its negation.



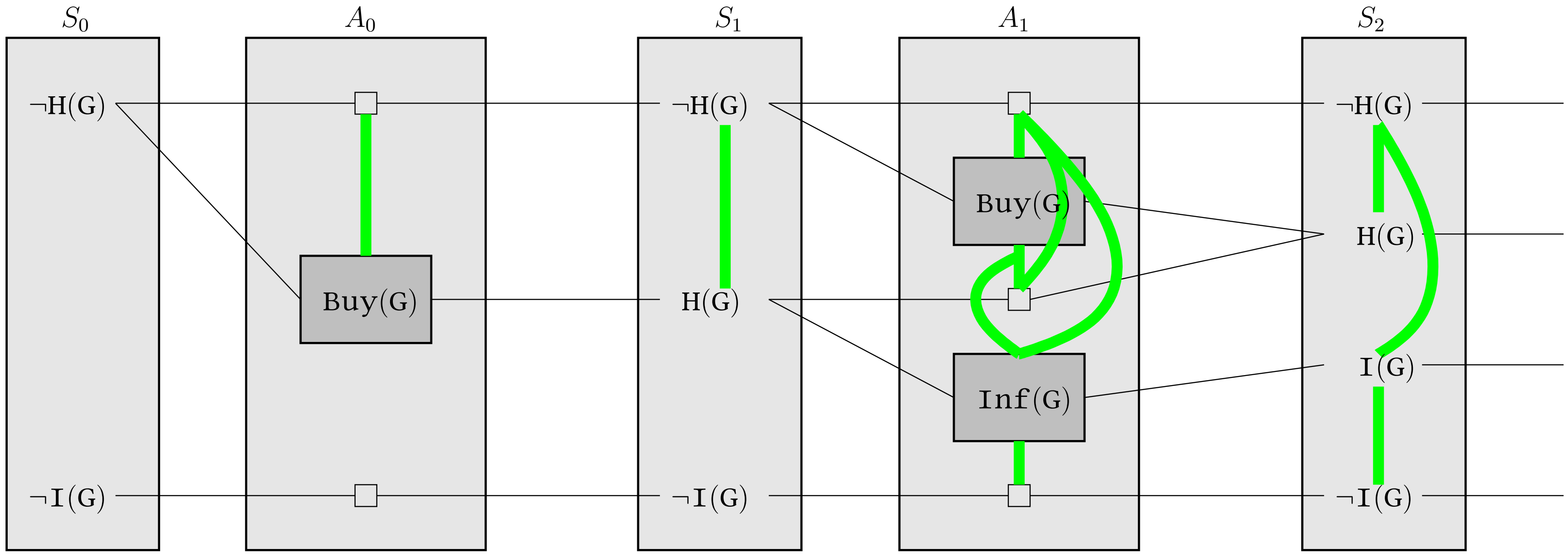
Mutex links

Possibility 2: all pairs of actions that could achieve the pair of propositions are mutex.



The construction of a planning graph is continued until two identical levels are obtained.

Planning graphs



Obtaining heuristics from a planning graph

To estimate the cost of reaching a single proposition:

- Any proposition not appearing in the final level has *infinite cost* and *can never be reached*.
- The *level cost* of a proposition is the level at which it first appears *but* this may be inaccurate as several actions can apply at each level and this cost does not count the *number of actions*. (It is however *admissible*.)
- A *serial planning graph* includes mutex links between all pairs of actions except persistence actions.

Level cost in serial planning graphs can be quite a good measurement.

Obtaining heuristics from a planning graph

How about estimating the cost to achieve a *collection* of propositions?

- *Max-level*: use the maximum level in the graph of any proposition in the set. Admissible but can be inaccurate.
- *Level-sum*: use the sum of the levels of the propositions. Inadmissible but sometimes quite accurate if goals tend to be decomposable.
- *Set-level*: use the level at which *all* propositions appear with none being mutex. Can be accurate if goals tend *not* to be decomposable.

Other points about planning graphs

A planning graph guarantees that:

1. *If* a proposition appears at some level, there *may* be a way of achieving it.
2. *If* a proposition does *not* appear, it can *not* be achieved.

The first point here is a loose guarantee because only *pairs* of items are linked by mutex links.

Looking at larger collections can strengthen the guarantee, but in practice the gains are outweighed by the increased computation.

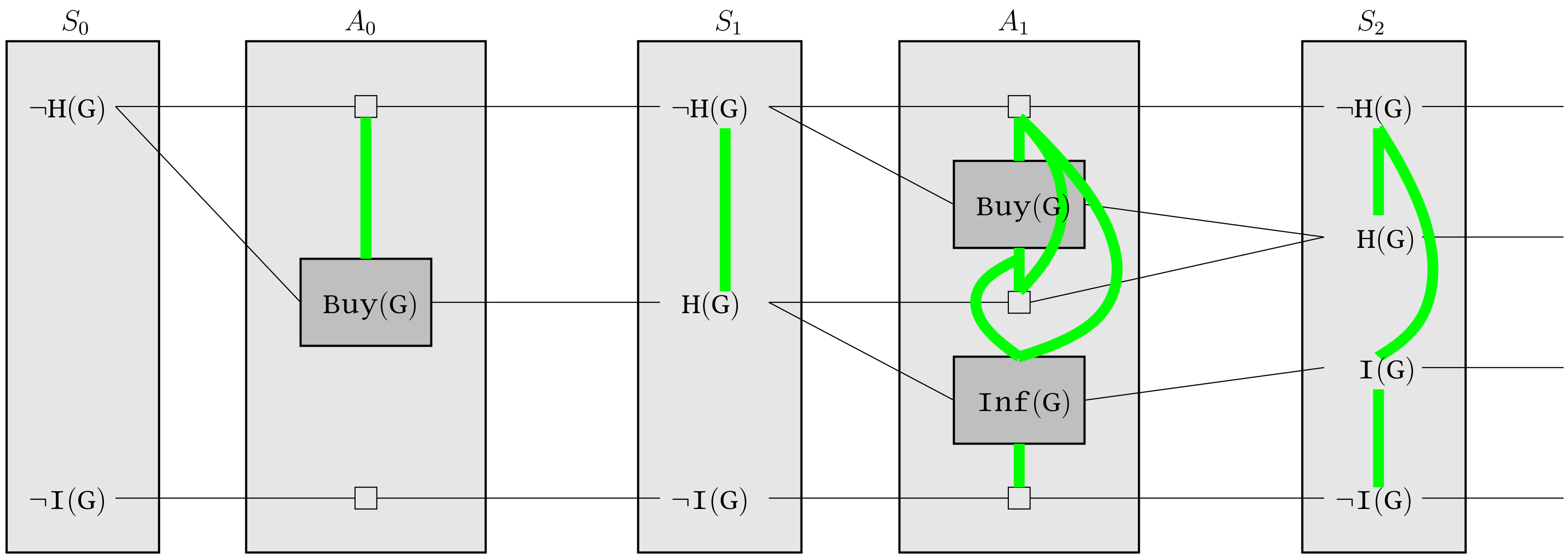
Graphplan

The *GraphPlan* algorithm goes beyond using the planning graph as a source of heuristics.

```
function GraphPlan()
    Start at level 0;
    while true do
        if All goal propositions appear in the current level AND no pair has a mutex link then
            Attempt to extract a plan;
            if A solution is obtained then
                return SOME solution;
            if Graph indicates there is no solution then
                return NONE;
        Expand the graph to the next level;
```

We *extract a plan* directly from the planning graph. Termination can be proved but will not be covered here.

Here, at levels S_0 and S_1 we do not have both `H(G)` and `I(G)` available with no mutex links, and so we expand first to S_1 and then to S_2.



At S_2 we try to extract a solution (plan).

Extracting a plan from the graph

Extraction of a plan can be formalised as a *search problem*.

States contain a *level*, and a collection of *unsatisfied goal propositions*.

Start state: the current final level of the graph, along with the relevant goal propositions.

Goal: a state at level S_0 containing the initial propositions.

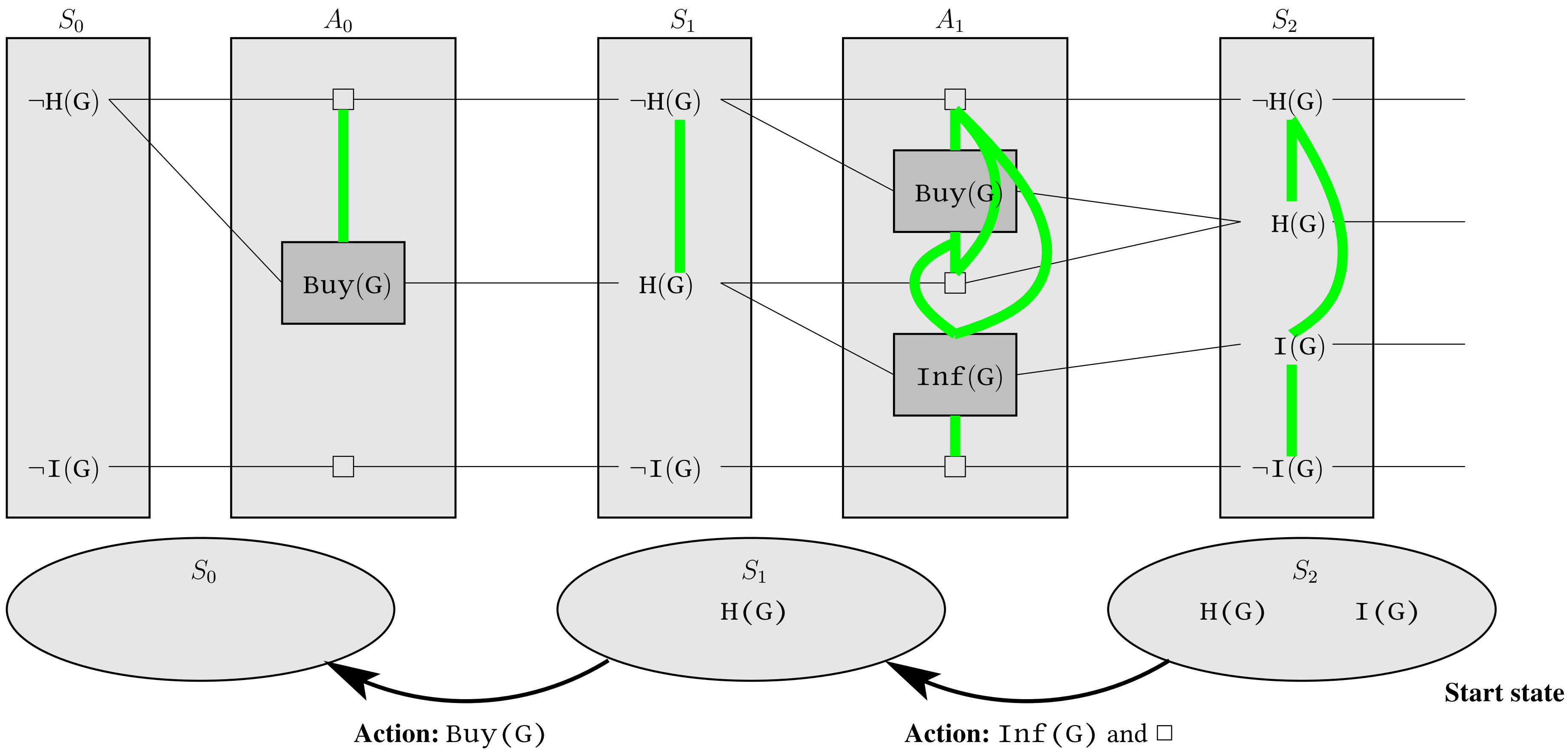
Actions: For a state S with level S_i, a valid action is to select any set X of actions in A_{i-1} such that:

1. no pair has a mutex link;
2. no pair of their preconditions has a mutex link;
3. the effects of the actions in X achieve the propositions in S.

The effect of such an action is a state having level S_{i-1}, and containing the preconditions for the actions in X.

Each action has a cost of 1.

Graphplan in action



Heuristics for plan extraction

We can of course also apply *heuristics* to this part of the process.

For example, when dealing with a *set of propositions*:

- Choose the proposition having *maximum level cost* first.
- For that proposition, attempt to achieve it using the action for which the *maximum/sum level cost of its preconditions is minimum*.