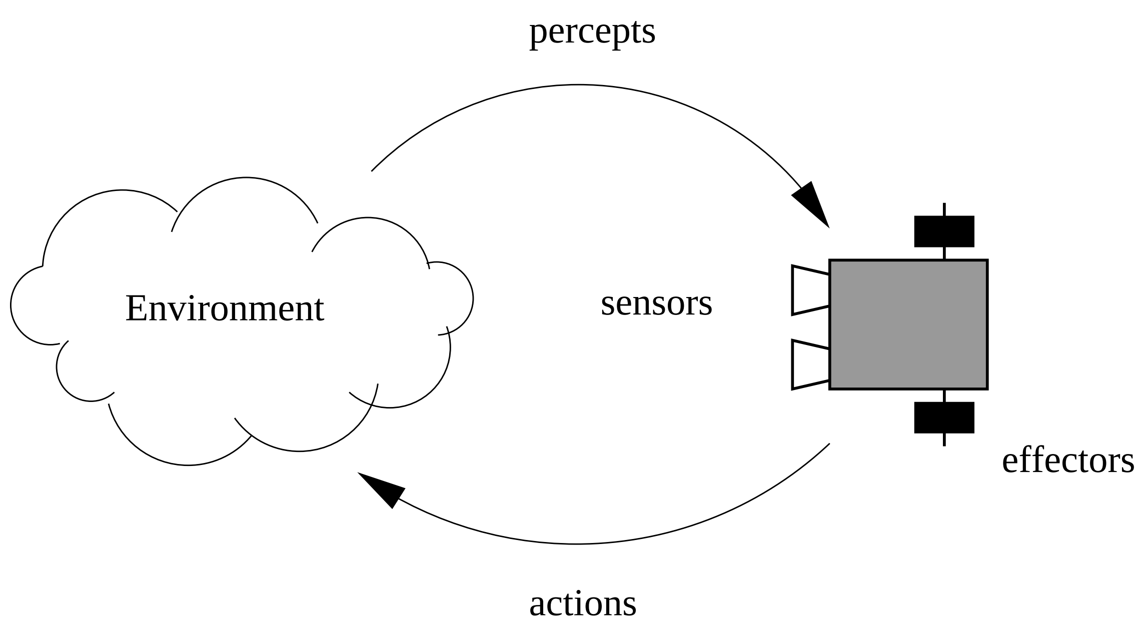
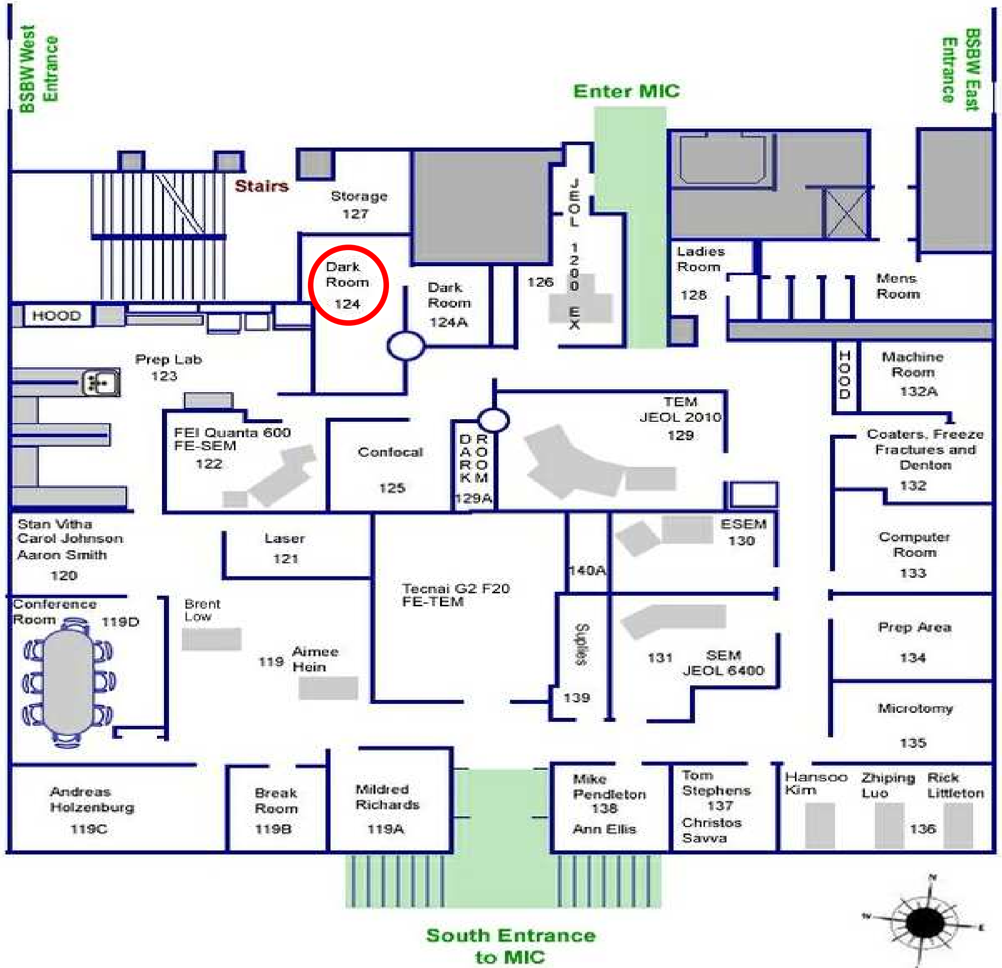
CLASSICAL PLANNING

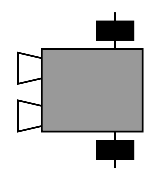
- We have talked about an agent's interaction with its environment:



- But what about when it has a more complex task to solve?



BSBW West Entrance
BSBW East Entrance
Enter MIC
Stairs
Storage 127
Dark Room 124
Dark Room 124A
126
JEOL 1200 EX
Ladies Room 128
Mens Room
HOOD
Prep Lab 123
Machine Room 132A
TEM JEOL 2010 129
FEI Quanta 600 FE-SEM 122
Confocal 125
DARK ROOM 129A
Coaters, Freeze Fractures and Denton 132
ESEM 130
Stan Vitha
Carol Johnson
Aaron Smith
120
Laser 121
Computer Room 133
140A
Tecnai G2 F20 FE-TEM
Conference Room
119D
Brent Low
119
Aimee Hein
Suplies
139
131
SEM JEOL 6400
Prep Area 134
Microtomy 135
Andreas Holzenburg 119C
Break Room 119B
Mildred Richards 119A
Mike Pendleton 138 Ann Ellis
Tom Stephens 137 Christos Savva
Hansoo Kim
Zhiping Luo
Rick Littleton
136
South Entrance to MIC



Texas A & M

- Could we use search techniques for this?

- Could we use search techniques for this?
- We could, but we'd need a lot of domain specific heuristics.
  - Hard to develop
- Prefer a more general solution.

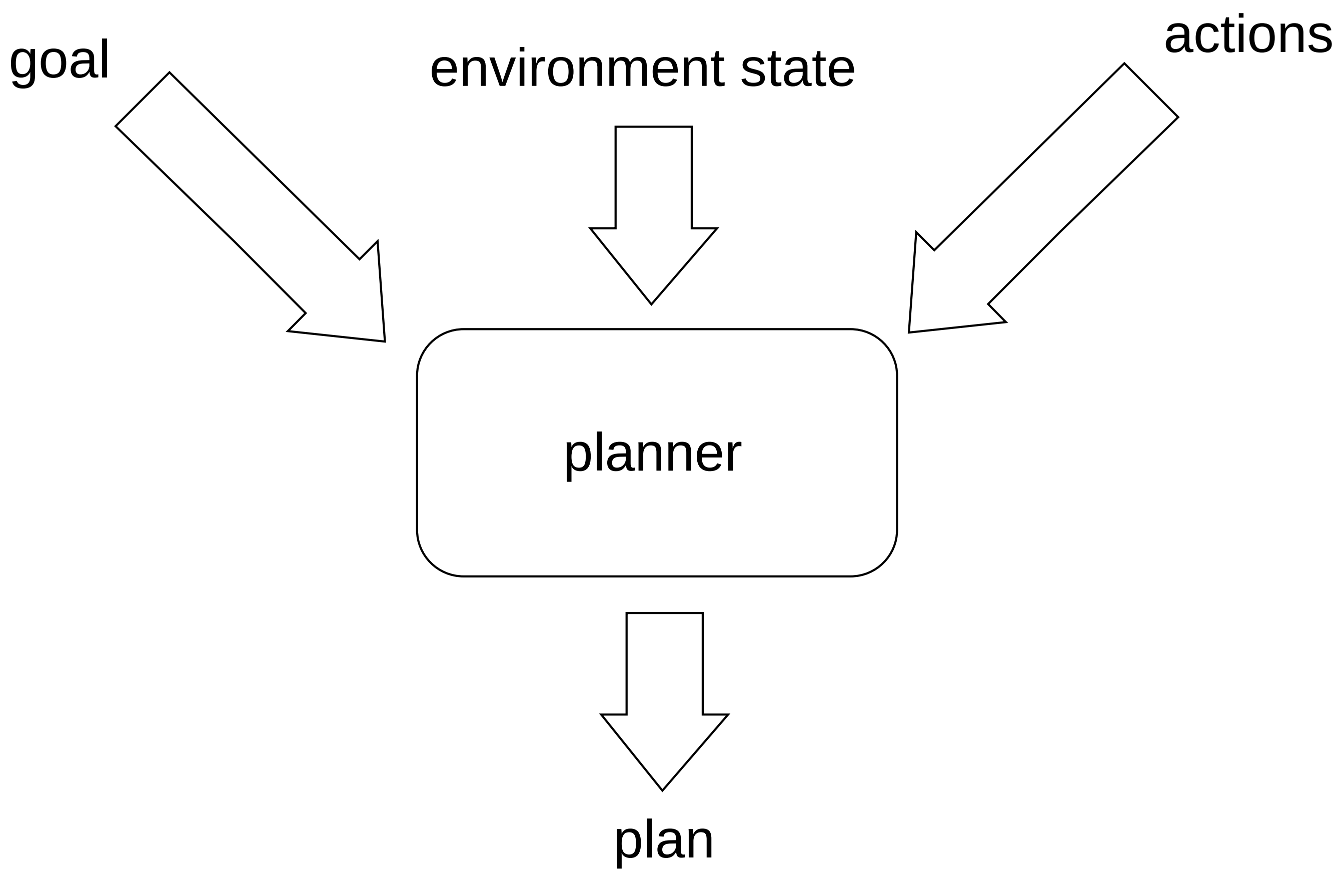
- Could we use Wumpus-world logic for this?

- Could we use Wumpus-world logic for this?
- We could, but we'd need a lot of computation.
  - Lots of reasoning to consider all the possible moves from each position.
- Prefer a faster solution

AI Planning

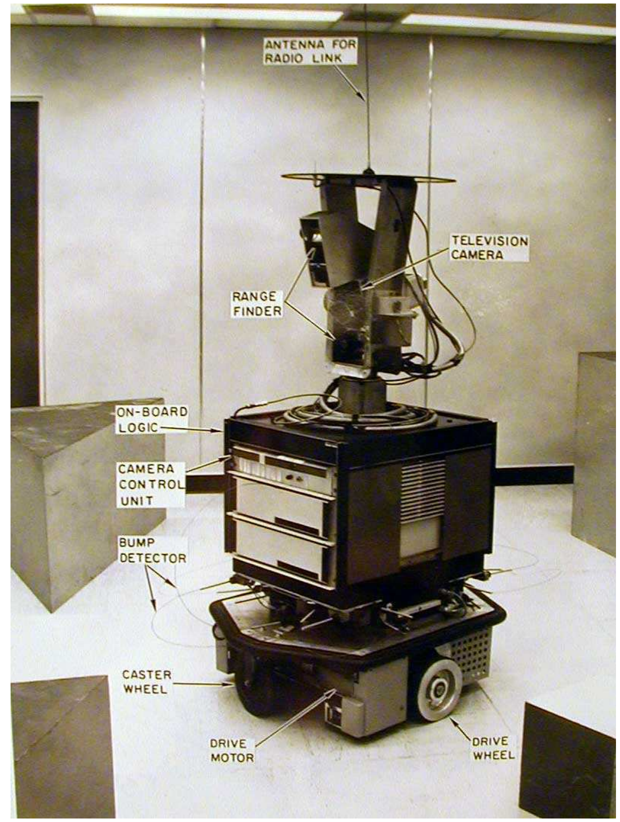
- Planning is the design of a course of action that will achieve some desired goal.
- Basic idea is to give a planning system:
  - (representation of) goal/intention to achieve;
  - (representation of) actions it can perform; and
  - (representation of) the environment;

  and have it generate a *plan* to achieve the goal.
- This is *automatic programming*.


goal
environment state
actions
planner
plan

- Given the problems with search and the use of simple logic, researchers turned to a more *factored* representation.
- An early successful approach to planning was STRIPS:
  - Stanford Research Institute Problem Solver.
- The textbook talks about PDDL rather than STRIPS, but the representations are very similar
  - PDDL can use negative literals in preconditions and goals.

- STRIPS was used in Shakey the robot:



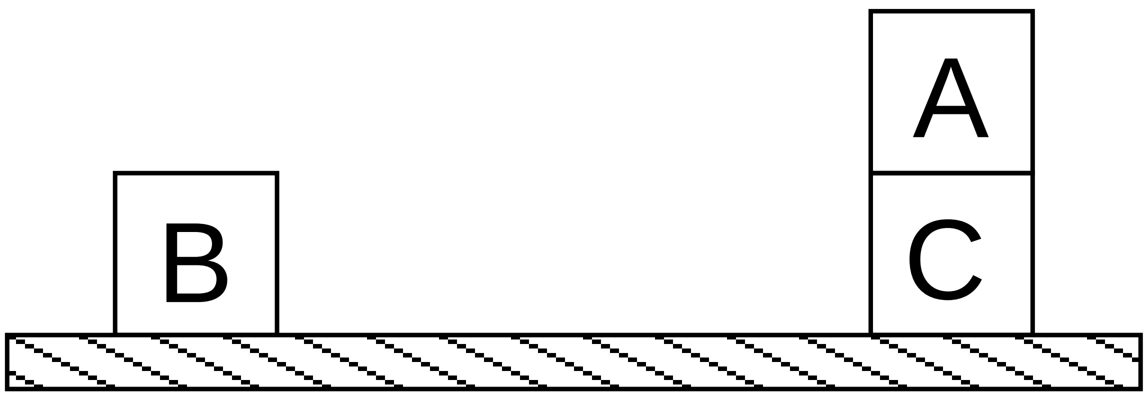
Representations

- Question: How do we *represent*. . .
  - goal to be achieved;
  - state of environment;
  - actions available to agent;
  - plan itself.
- Answer: We use logic, or something that looks a lot like logic.

- We'll illustrate the techniques with reference to the *blocks world*.
- A simple (toy) world, in this case one where we consider toys:

- The blocks world contains a robot arm, 3 blocks (A, B and C) of equal size, and a table-top.

- The aim is to generate a plan for the robot arm to build towers out of blocks.
- For a formal description, we'll clean it up a bit:



- To represent this environment, need an *ontology*.

$On(x, y)$	obj x on top of obj y
$OnTable(x)$	obj x is on the table
$Clear(x)$	nothing is on top of obj x
$Holding(x)$	arm is holding x

- Here is a representation of the blocks world described above:

$$
\begin{aligned}
&Clear(A)\\
&On(A, B)\\
&OnTable(B)\\
&Clear(C)\\
&OnTable(C)
\end{aligned}
$$

- Use the *closed world assumption*
  - Anything not stated is assumed to be *false*.

- A *goal* is represented as a set of formulae.
- Here is a goal:

$$\{OnTable(A), \quad OnTable(B), \quad OnTable(C)\}$$

- *Actions* are represented as follows.

  Each action has:

  - a *name*
    which may have arguments;
  - a *pre-condition list*
    list of facts which must be true for action to be executed;
  - a *delete list*
    list of facts that are no longer true after action is performed;
  - an *add list*
    list of facts made true by executing the action.

  Each of these may contain *variables*.

- The *stack* action occurs when the robot arm places the object x it is holding is placed on top of object y.

$$
\begin{array}{ll}
 & Stack(x, y) \\
\text{pre} & Clear(y) \wedge Holding(x) \\
\text{del} & Clear(y) \wedge Holding(x) \\
\text{add} & ArmEmpty \wedge On(x, y)
\end{array}
$$

- We can think of variables as being universally quantified.
- ArmEmpty is an abbreviation for saying the arm is not holding any of the objects.

- The *unstack* action occurs when the robot arm picks an object x up from on top of another object y.

$$\begin{array}{ll} & UnStack(x,y) \\ \text{pre} & On(x,y) \wedge Clear(x) \wedge ArmEmpty \\ \text{del} & On(x,y) \wedge ArmEmpty \\ \text{add} & Holding(x) \wedge Clear(y) \end{array}$$

Stack and UnStack are *inverses* of one-another.

- The *pickup* action occurs when the arm picks up an object x from the table.

$$
\begin{array}{ll}
 & Pickup(x) \\
\text{pre} & Clear(x) \wedge OnTable(x) \wedge ArmEmpty \\
\text{del} & OnTable(x) \wedge ArmEmpty \\
\text{add} & Holding(x)
\end{array}
$$

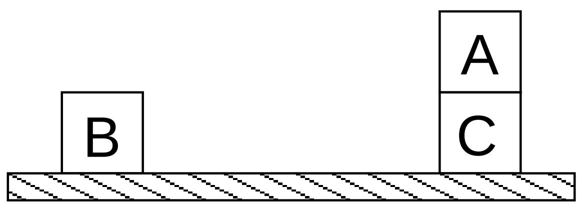
- The *putdown* action occurs when the arm places the object x onto the table.

$$\begin{array}{ll} & PutDown(x) \\ \text{pre} & Holding(x) \\ \text{del} & Holding(x) \\ \text{add} & Clear(x) \wedge OnTable(x) \wedge ArmEmpty \end{array}$$

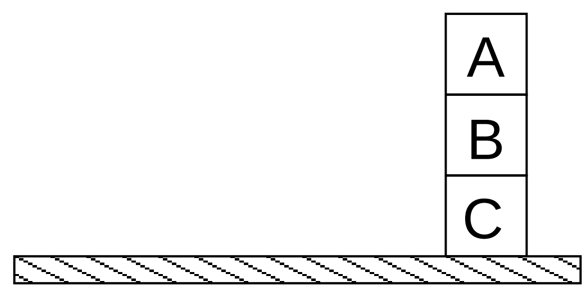
- What is a plan?

  A sequence (list) of actions, with variables replaced by constants.

- So, to get from:



to



- What plan do we need?

- We need the plan:

$$
\begin{gathered}
Unstack(A) \\
Putdown(A) \\
Pickup(B) \\
Stack(B, C) \\
Pickup(A) \\
Stack(A, B)
\end{gathered}
$$

Naive Planner

- In "real life", plans contain conditionals (`IF .. THEN...`) and loops (`WHILE... DO...`), but most simple planners cannot handle such constructs — they construct *linear plans*.
- Simplest approach to planning:

  means-ends analysis.
- Start from where you want to get to (ends) and apply actions (means) that will achieve this state.

- Involves backward chaining from goal to original state.
- Start by finding an action that is consistent with having the *goal* as post-condition.

  Assume this is the *last* action in plan.
- Then figure out what the previous state would have been.

  Try to find action that has *this* state as post-condition.
- *Recurse* until we end up (hopefully!) in original state.
- We say that an action a can be executed in state s if s entails the precondition $pre(a)$ of a.

$$s \models pre(a)$$

- This is true iff every positive literal in $pre(a)$ is in s, and every negative literal in $pre(a)$ is not.

Here's an algorithm for finding a plan:

function *plan*(
  d : WorldDesc, // environment state
  g : Goal, // current goal
  p : Plan, // plan so far
  A : set of actions // actions available)
1. if $d \models g$ then
2. return p
3. else
4. choose some a in A with $g \models add(a)$
5. set $g = (g - add(a)) \cup pre(a)$
6. append a to p
7. return $plan(d, g, p, A)$

- Note that we *ignore* the delete list.

- How does this work on the previous example?

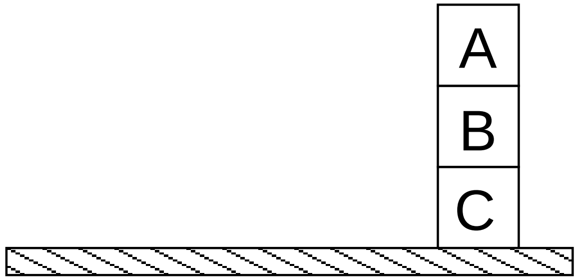
- We start with the goal state:

$$On(A, B)$$
$$On(B, C)$$
$$OnTable(C)$$
$$ArmEmpty$$

- Then pick an action which has an add list that is satisfied by this state:

$$Stack(A, B)$$

- To get the state before this action, delete the add list and add the preconditions.

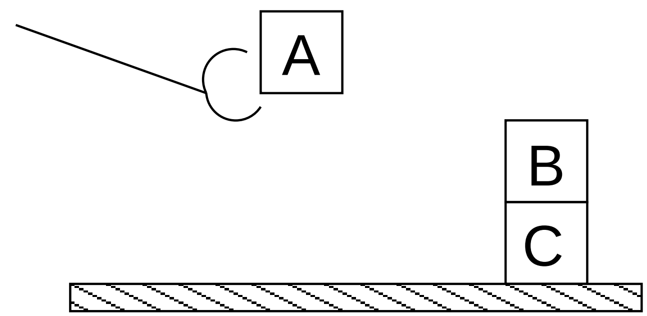


- This gives us:

$$Clear(B)$$
$$On(B, C)$$
$$OnTable(C)$$
$$Holding(A)$$

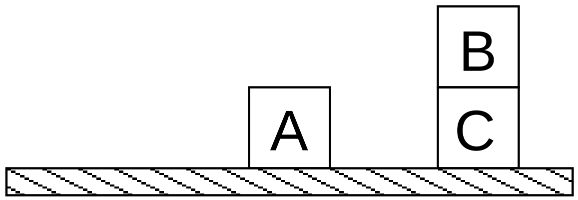
- Pick the previous action in the plan, now it is an action whose add list is satisfied by the above state.

$$Pickup(A)$$



- Now we are here:

$$
\begin{array}{c}
Clear(B) \\
On(B, C) \\
OnTable(C) \\
OnTable(A) \\
ArmEmpty
\end{array}
$$



- And so we go, working backwards until we get to the initial state.

- This algorithm is *not* guaranteed to find a plan to satisfy the goal.
  - Why is that?
- However, this algorithm is *sound*: If it finds the plan is correct.

- Some problems:
  - negative goals;
  - maintenance goals;
  - conditionals & loops;
  - exponential search space;
  - logical consequence tests;

- Negative goals are a problem because?

- Negative goals are a problem because…
- How would you write down:

  Build any tower of blocks where block B is *not* on the table.

  without enumerating all the towers that you could build?

- Maintenance goals are a problem because?

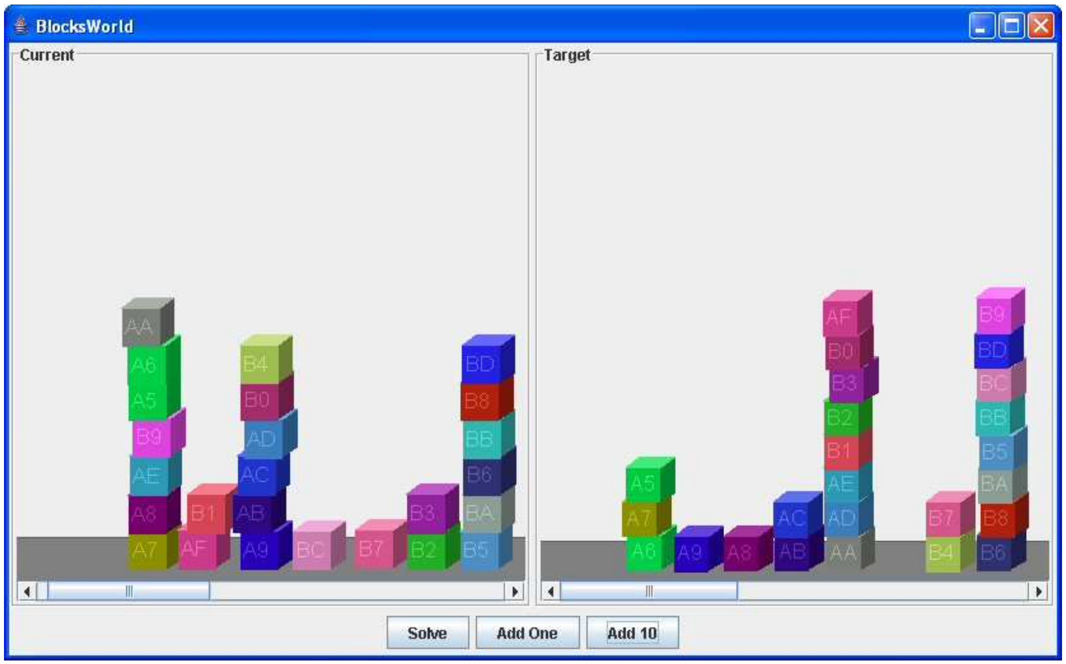
- Maintenance goals are a problem because. . .
- How would you write down:

  Keep moving the bricks around so that there are always at least two bricks on the table.

  without enumerating all the towers that you could build?
- Maintenance goal:

- Exponential search space is a problem because?

- Exponential search space is a problem because:



- Many planning problems have $\sim 10^{100}$*states*.

- Logical consequence tests are a problem because?

- Logical consequence tests are a problem because, to quote Wikipedia:

  > Depending on the underlying logic, the problem of deciding the validity of a formula varies from trivial to *impossible*.
  >
  > For propositional logic, the problem is *decidable* but Co-NP-complete, and hence only *exponential-time* algorithms are believed to exist for general proof tasks.
  >
  > For a first order predicate calculus identifying valid formulas is recursively enumerable: given unbounded resources, any valid formula can *eventually* be proven. However, invalid formulas *cannot* always be recognized.

  (this was heavily cut down, emphasis is mine)

Search space issues

- Another problem with the search space is:
  - how do we pick an action?
- We are just assuming that you can pick a good one.
  - In general, not a good tactic.
- Apply heuristics and use A^*
  - This is just a form of search problem after all

Didn’t you say before that we shouldn’t think of this as search?

Well, yes...

- The difference is that with the factored search operators we can look for *domain independent* heuristics.
  - Ones that will work for planning problems in general.
- Ignore preconditions
  - Just as in search we can establish heuristics that relax the constraints on the problem ensuring that they are *admissable*.
- Ignore selected preconditions.
- Ignore delete lists
  - No action undoes the effect of another action.

- While this gives us a set of heuristics, the state space is still big
  - $\sim 10^{100}$ remember
- State abstraction.
  - plan in a space that groups states together
- The textbook talks about planning for 10 airports with 50 planes and 200 pieces of luggage.
  - Every plane can be at any airport and each package can be on any plane or unloaded at an airport.
  - $50^{10} \times 200^{50+10} \approx 10^{155}$ states

- If all the packages are constrained to be at only 5 of the airports, and all packages at one airport have the same destination, we can reduce the problem to have just 5 airports and and one plane and package at the same airport.
  - $5^{10} \times 5^{50+10} \approx 10^{17}$ states
- Find solution and then expand back to the larger problem, maybe by composing solutions.
- Not optimal but easier.

The Frame Problem

- A general problem with representing properties of actions:

  How do we know exactly what changes as the result of performing an action?

  If I pick up a block, does my hair colour stay the same?

- One solution is to write *frame axioms*.

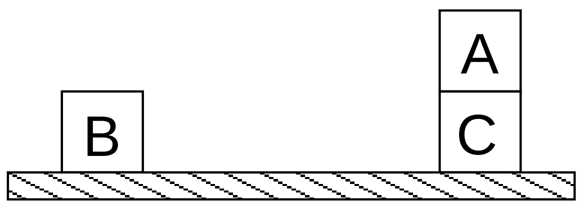
  Here is a frame axiom, which states that my hair colour is the same in all the situations s' that result from performing $Pickup(x)$ in situation s as it is in s.

$$\forall s, s'. Result(SP, Pickup(x), s) = s' \Rightarrow HCol(SP, s) = HCol(SP, s')$$

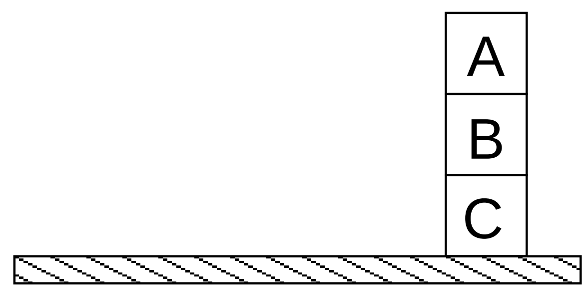
- Stating frame axioms in this way is infeasible for real problems.
- (Think of all the things that we would have to state in order to cover all the possible frame axioms).
- STRIPS solves this problem by assuming that everything not explicitly stated to have changed remains unchanged.
- The price we pay for this is that we lose one of the advantages of using logic:
  - Semantics goes out of the window
- However, more recent work has effectively solved the frame problem (using clever second-order approaches).

Sussman's Anomaly

- Consider again the following initial state and goal state:

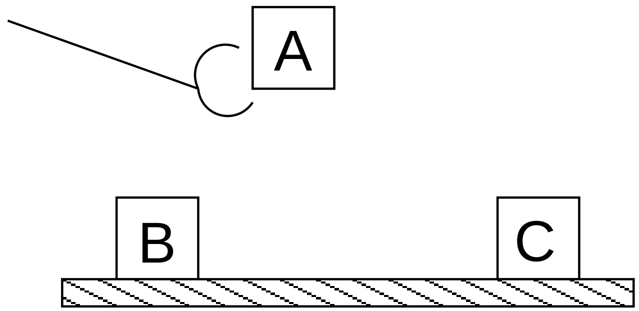


to



- Clearly the first operation is to unstack A from C.

- which gets us to here:

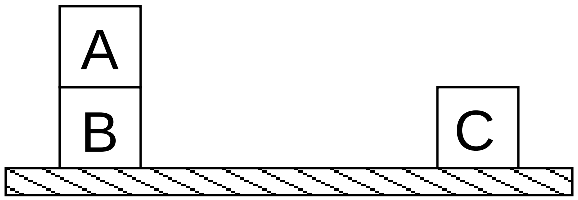


- But what next.
- If the planner considers that the final state is to have:

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

then making the next move $Stack(A, B)$ might seem to be close to the goal.

- We then get to:



which is no closer to our real goal.

- In fact it just means a longer path to the goal which involves going back through the previous state.
- This is a big problem with linear planners
- How could we modify our planning algorithm?

- Modify the middle of the algorithm to be:

1. if $d \models g$ then
2.   return p
3. else
4.   choose some a in A

4a.   if $no\_clobber(a, rest\_of\_plan)$

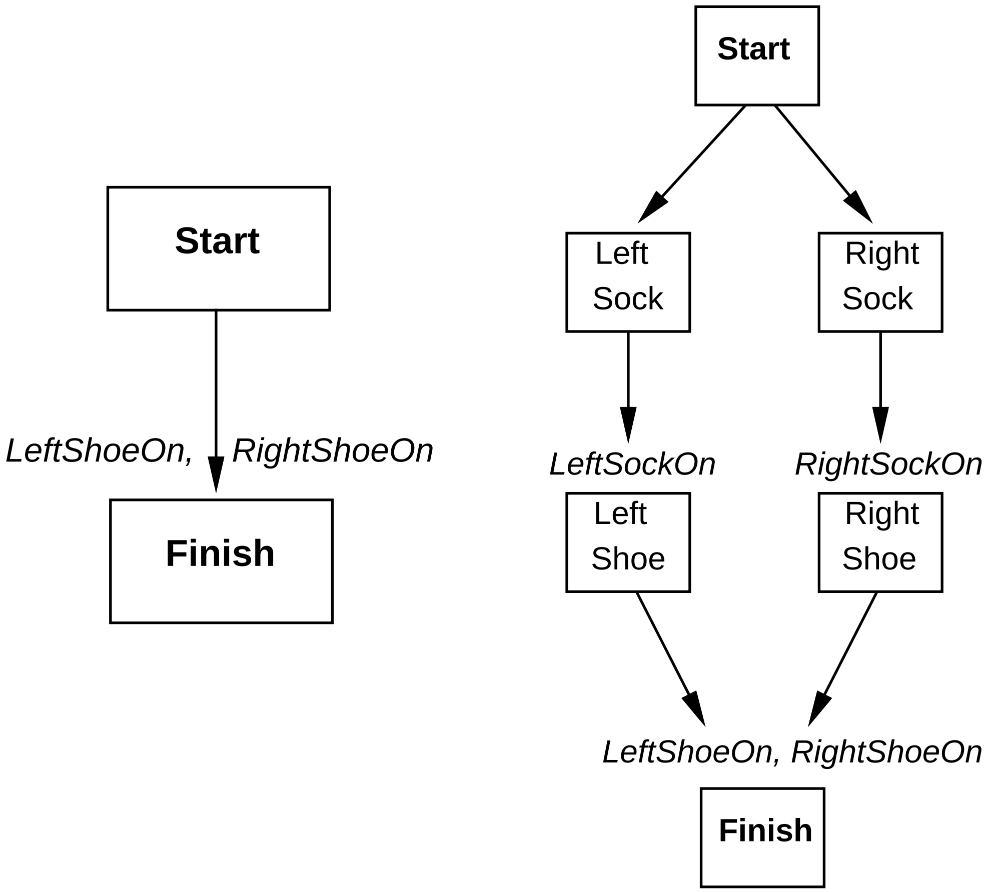
5.   set $g = (g - add(a)) \cup pre(a)$
6.   append a to p
7.   return $plan(d, g, p, A)$

- But how can we do this?

Partial Order Planning

- The answer to the problem on the previous slide is to use *partial order planning*.
- Basically this gives us a way of checking before adding an action to the plan that it doesn't mess up the rest of the plan.
- The problem is that in the recursive process used by STRIPS, we don't know what the "rest of the plan" is.
- Need a new representation *partially ordered plans*.
- This means remembering what a "partial order" is.

Representation



Partially ordered plans

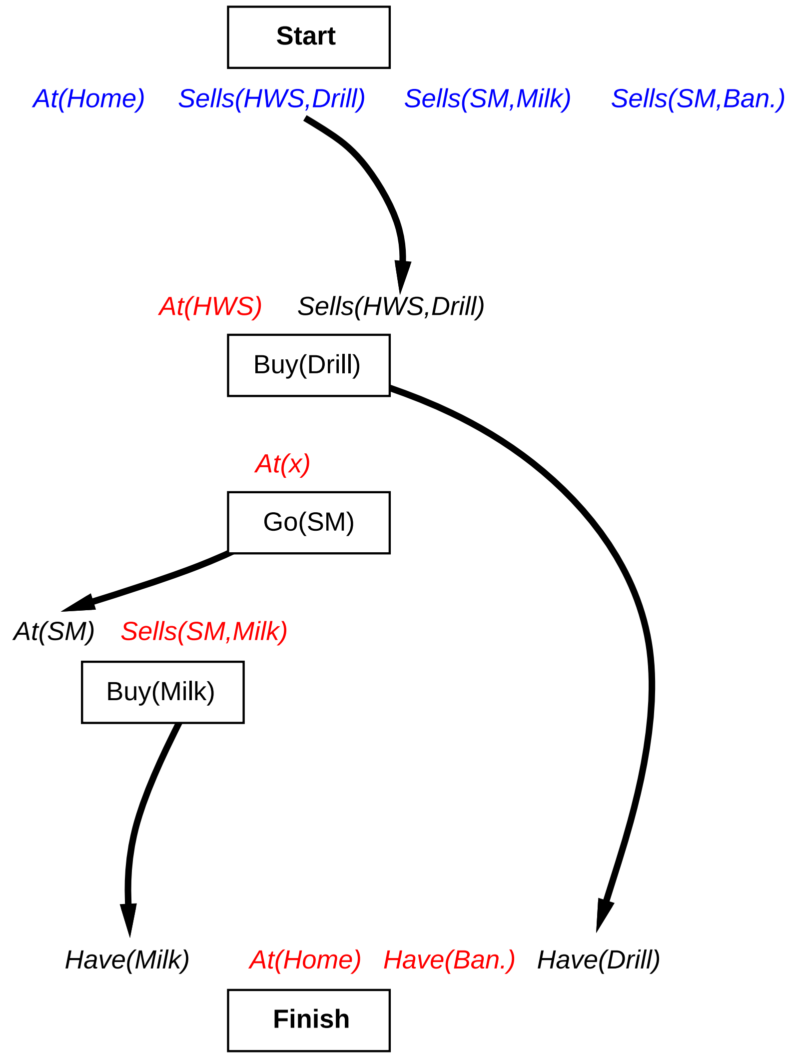
- *Partially ordered* collection of steps with
  - *Start* step has the initial state description as its effect
  - *Finish* step has the goal description as its precondition
  - *causal links* from outcome of one step to precondition of another
  - *temporal ordering* between pairs of steps
- *Open condition* = precondition of a step not yet causally linked
- A plan is *complete* iff every precondition is achieved
- A precondition is *achieved* iff it is the effect of an earlier step and no *possibly intervening* step undoes it

Plan construction

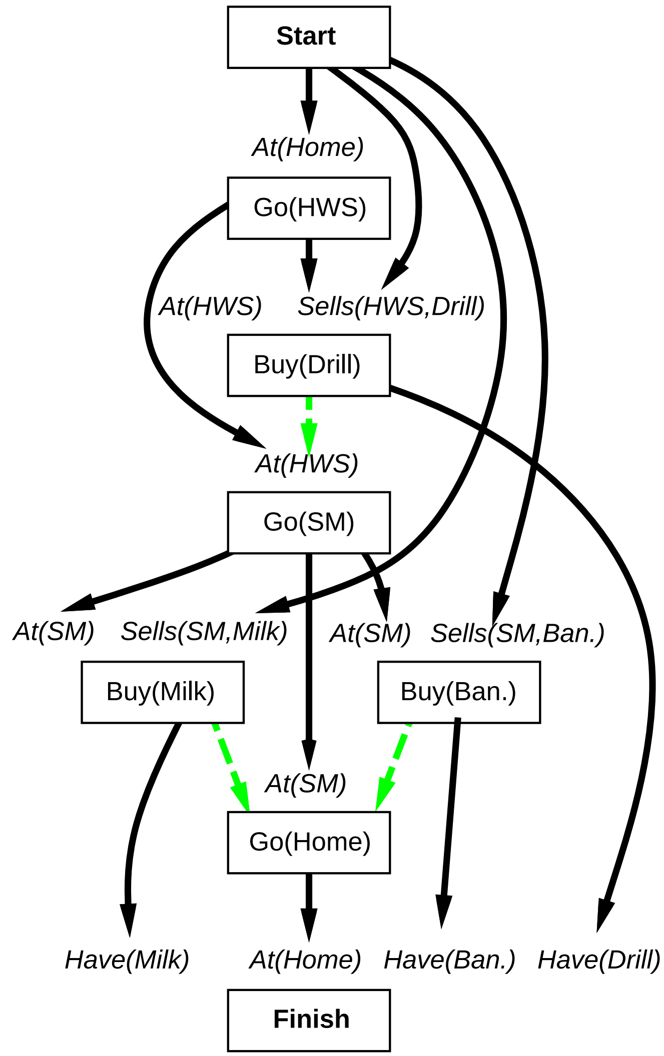
- We start with just the start and end states.


Start
At(Home)
Sells(HWS,Drill)
Sells(SM,Milk)
Sells(SM,Ban.)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

- Then we add in actions, as they seem appropriate.
- We introduce actions that achieve:
  - either the pre-conditions of the final state; or
  - the pre-conditions of actions that were already added.
- Matching pre- and post-conditions are linked.


Start
At(Home)
Sells(HWS,Drill)
Sells(SM,Milk)
Sells(SM,Ban.)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(x)
Go(SM)
At(SM)
Sells(SM,Milk)
Buy(Milk)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

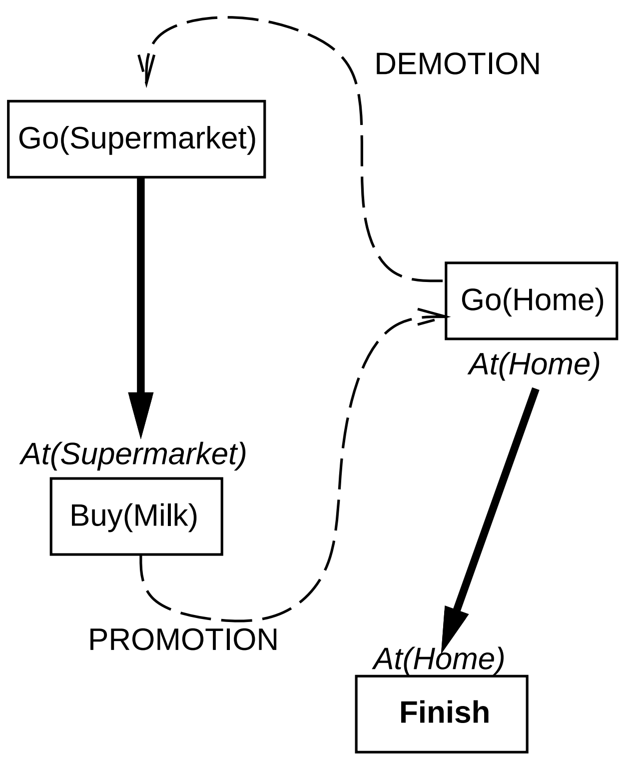
- Some actions will introduce ordering constraints on other actions by having post-conditions that make the pre-conditions of those other actions false.
- These force us to order some actions with respect to each other.
- Thus we don't care what order we buy the milk and bananas in, but we have to do both before we go home.


Start
At(Home)
Go(HWS)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM)
Sells(SM,Milk)
At(SM)
Sells(SM,Ban.)
Buy(Milk)
Buy(Ban.)
At(SM)
Go(Home)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

- The causal links between actions give us a way to detect the "clobbering" mentioned in the previous algorithm.
- This tells us how the steps must be ordered
  - If they need ordering.

Clobbering

- A *clobberer* is a potentially intervening step that destroys the condition achieved by a causal link. E.g., $Go(Home)$ clobbers $At(Supermarket)$:



Demotion: put before $Go(Supermarket)$

Promotion: put after $Buy(Milk)$

Planning process

- Operators on partial plans:
  - *add a link* from an existing action to an open condition
  - *add a step* to fulfill an open condition
  - *order* one step wrt another to remove possible conflicts
- Gradually move from incomplete/vague plans to complete, correct plans
- Backtrack if an open condition is unachievable or if a conflict is unresolvable

function POP(*initial*, *goal*, *operators*) **returns** *plan*

plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)
loop do
  if SOLUTION?(*plan*) **then return** *plan*
  S_{need}, c ← SELECT-SUBGOAL(*plan*)
  CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
  RESOLVE-THREATS(*plan*)
end

---

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c

pick a plan step S_{need} from STEPS(*plan*)
  with a precondition c that has not been achieved
return S_{need}, c

procedure CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)

choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect

if there is no such step **then fail**

add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)

add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)

if S_{add} is a newly added step from *operators* **then**

add S_{add} to STEPS(*plan*)

add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

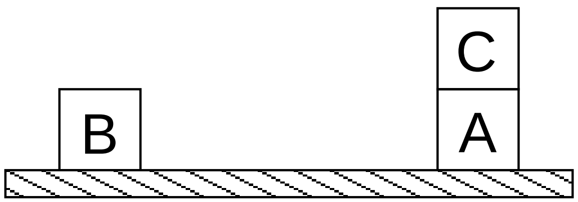
```
procedure RESOLVE-THREATS(plan)
  for each S_threat that threatens a link S_i --c--> S_j in LINKS(plan) do
     choose either
        Demotion: Add S_threat ≺ S_i to ORDERINGS(plan)
        Promotion: Add S_j ≺ S_threat to ORDERINGS(plan)
     if not CONSISTENT(plan) then fail
  end
```

Properties of POP

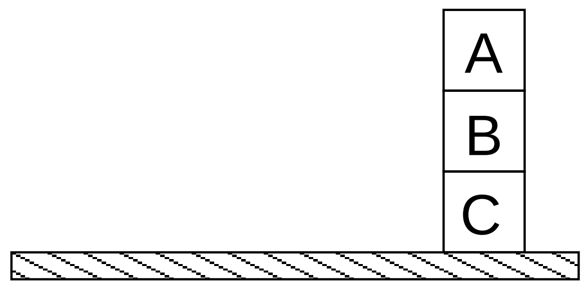
- Nondeterministic algorithm: backtracks at *choice* points on failure:
  - choice of S_{add} to achieve S_{need}
  - choice of demotion or promotion for clobberer
  - selection of S_{need} is irrevocable
- POP is sound, complete, and systematic (no repetition)
- Extensions for disjunction, universals, negation, conditionals
- Can be made efficient with good heuristics derived from problem description
- Particularly good for problems with many loosely related subgoals

Sussman's Anomaly Revisited

- Another version of Sussman's anomaly appears here:

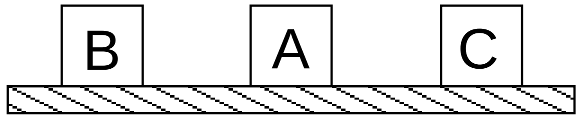


to

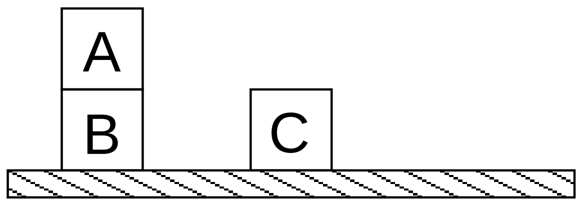


- In this case the problem appears once we have placed all the blocks on the table.

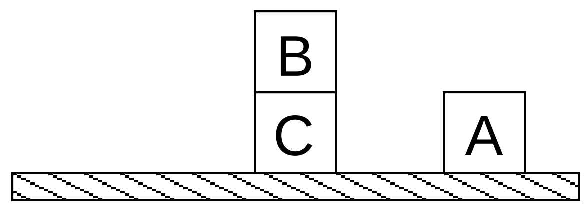
- From here:



this:

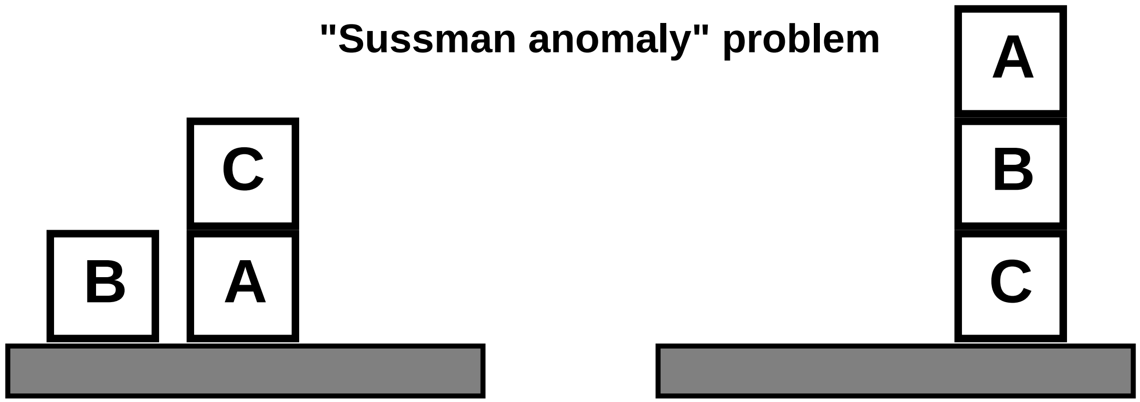


seems as good a move as this:



without some special purpose heuristic.

"Sussman anomaly" problem



Start State | Goal State

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y) Clear(z) On(x,y)

Clear(x) On(x,z)

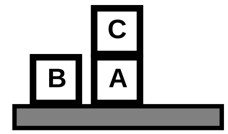
PutOnTable(x)

~On(x,z) Clear(z) On(x,Table)

\+ several inequality constraints

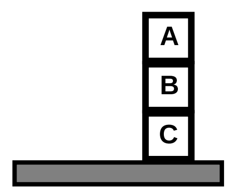
START

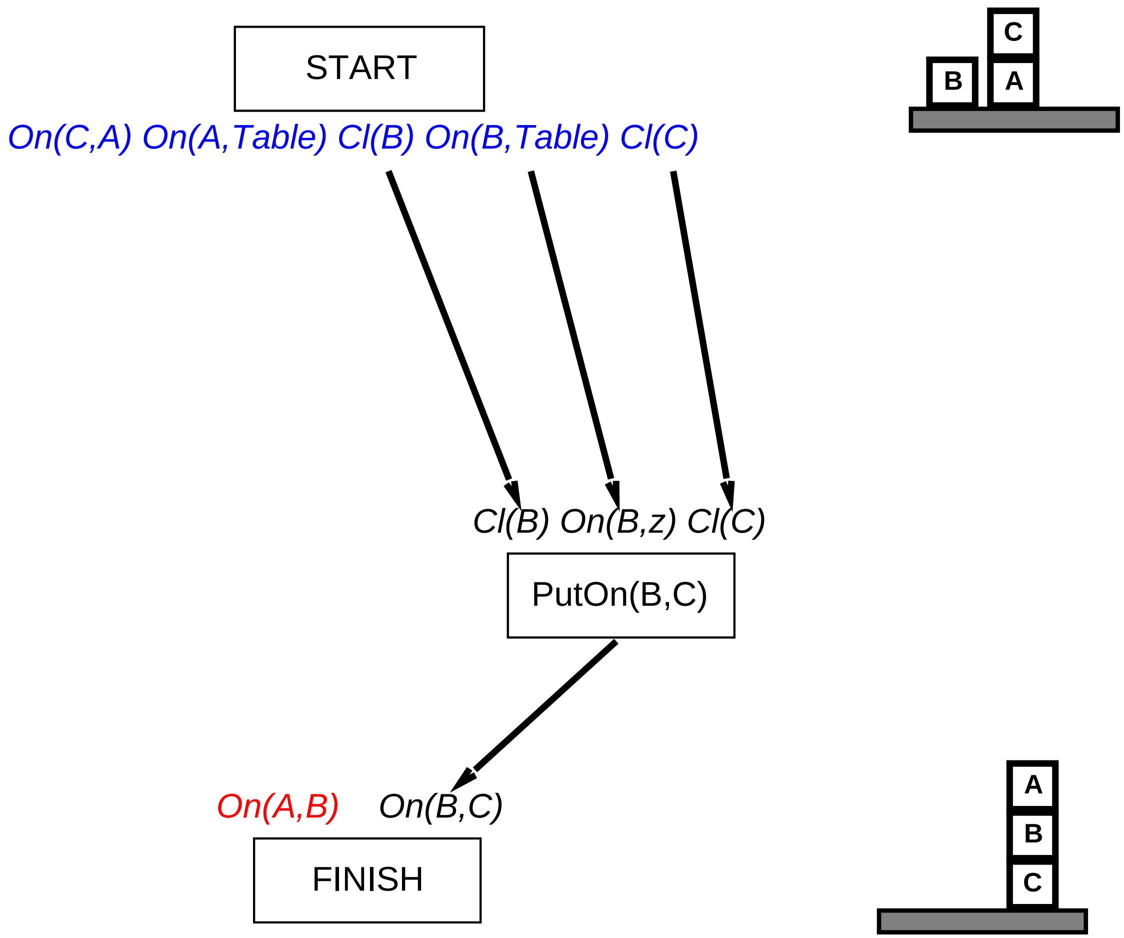
On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

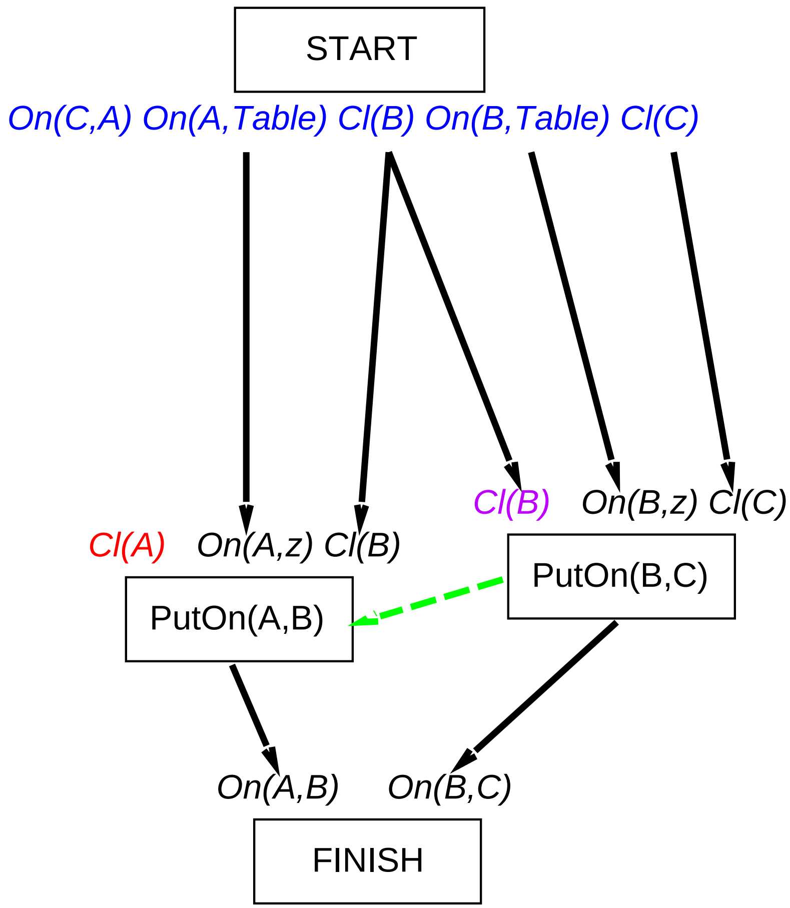


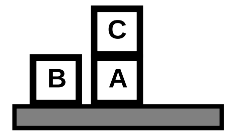
On(A,B) *On(B,C)*

FINISH

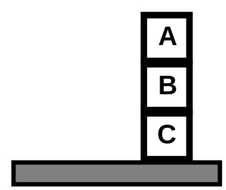


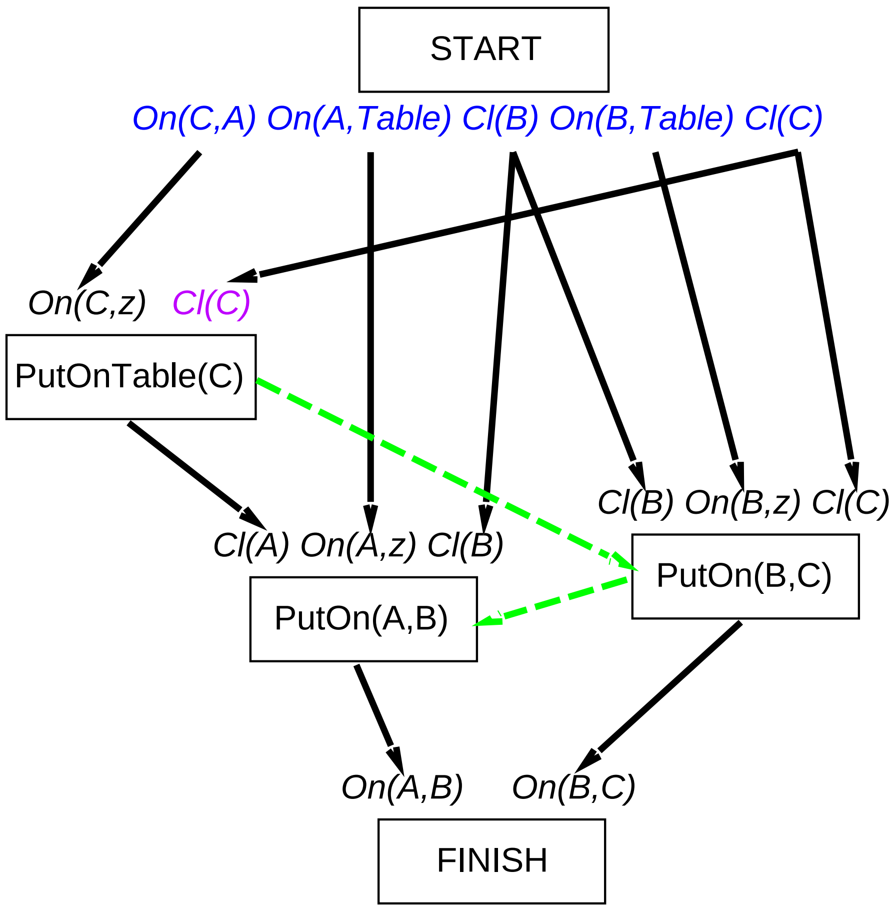

START
On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)
Cl(B) On(B,z) Cl(C)
PutOn(B,C)
On(A,B)
On(B,C)
FINISH
C
B
A
A
B
C

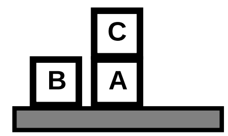




PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

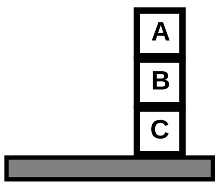






PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)



State of the art

- Though POP is quite intuitive, it isn't the best planner out there any more.
- Currently the hottest planning approaches are the following.
- SATPlan
  - Specify the problem in logic, including all possible transitions.
  - See if there is a satisfying model

  This shifts the computational burden to the creation of all possible sequences, which can then be checked fast for specific goals.
- Search with clever general purpose heuristics.
- GraphPlan
  - Build a graph which approximates the state space.

Summary

- This lecture has looked at planning.
- We started with a logical view of planning, using STRIPS operators.
- We also discussed the frame problem, and Sussman's anomaly.
- Sussman's anomaly motivated some thoughts about partial-order planning.
- We looked at partial order planning in some detail, and then talked about the POP algorithm.