Planning

CSE-345:Aritificial Intelligence

Objective

- Advanced Problem Solving Approaches
- How Planning works
- Components of a Planning System
- Block World Problem
- STRIPS Mechanism
- Simple planning using a Goal Stack
- Sussman anomaly problem

Advanced Problem Solving Approaches

- In order to solve nontrivial problems, it is necessary to combine
  - Basic problem solving strategies
  - Knowledge representation mechanisms
  - Partial solutions and at the end combine into complete problem solution (decomposition)
- Planning refers to the process of computing several steps of a problem solving before executing any of them.
- Planning is useful as a problem solving technique for non decomposable problem.

How Planning works?

1. Avoid having to recomputed the entire problem state when move from one state to the next. Instead, consider only that part of the state that may have changed.

Ex: guiding robot around an ordinary house. The description of a single state is very large since it must describe the location of each object in the house as well as that of the robot. A given action on the part of the robot will change only a small part of the total state. If the robot pushes a table across the room, then the locations of the table & all of the objects that were on it will change. But the locations of the other objects in the house will not.

-Instead of writing rules that describe transformations of one entire state into another, we would like to write rules that describe only the affected parts of the state description.

-The rest of the description can then be assumed to stay constant.

How Planning works?

2. **Problem decomposition:**

-the division of a single difficult problem into several sub problems

-can make the solution of hard problems easier

-sometimes possible, often not

Nearly decomposable: problems may be divided into sub problems that have only a small amount of interaction.

Ex: suppose to move all furniture's out of room. This problem can be decomposed into a set of smaller problems, each involving moving one piece of furniture out of the room. Within each of these sub problems, considerations such as removing drawers can be addressed separately for each piece of furniture. But if there is a bookcase behind a couch, then we must move the couch before we can move the bookcase.

-To solve such nearly decomposable problems, we would like a method that enables us to work on each sub problem separately.

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search often proceeds through *plan space* rather than *state space* (though there are also state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Planning

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of subgoals to be achieved

Components of a Planning System

- In any general problem solving systems, elementary techniques to perform following functions are required
  - Choose the best rule (based on heuristics) to be applied
  - Apply the chosen rule to get new problem state
  - Detect when a solution has been found
  - Detect dead ends so that new directions are explored.

Choose Rules to apply

- Most widely used technique for selecting appropriate rules is to
  - first isolate a set of differences between the desired goal state and current state,
  - identify those rules that are relevant to reducing these difference,
  - if more rules are found then apply heuristic information to choose out of them.

Apply Rules

- In simple problem solving system,
  - applying rules was easy as each rule specifies the problem state that would result from its application.
  - In complex problem we deal with rules that specify only a small part of the complete problem state.

Example: Block World Problem

- Block world problem assumptions
  - Square blocks of same size
  - Blocks can be stacked one upon another.
  - Flat surface (table) on which blocks can be placed.
  - Robot arm that can manipulate the blocks. It can hold only one block at a time.
- In block world problem, the state is described by a set of predicates representing the facts that were true in that state.
- One must describe for every action, each of the changes it makes to the state description.
- In addition, some statements that everything else remains unchanged is also necessary.

Actions (Operations) done by Robot

- UNSTACK (X, Y) : **[US (X, Y)]**
  - Pick up X from its current position on block Y. The arm must be empty and X has no block on top of it.
- STACK (X, Y): **[S (X, Y)]**
  - Place block X on block Y. Arm must holding X and the top of Y is clear.
- PICKUP (X): **[PU (X)]**
  - Pick up X from the table and hold it. Initially the arm must be empty and top of X is clear.
- PUTDOWN (X): **[PD (X)]**
  - Put block X down on the table. The arm must have been holding block X.

Contd..

- Predicates used to describe the state

ON(X, Y)	-	Block X on block Y.
ONT(X)	-	Block X on the table.
CL(X)	-	Top of X clear.
HOLD(X)	-	Robot-Arm holding X.
AE	-	Robot-arm empty.

- Logical statements true in this block world.
  - Holding X means, arm is not empty

    $(\exists X)\ HOLD(X) \rightarrow \sim AE$
  - X is on a table means that X is not on the top of any block

    $(\forall X)\ ONT(X) \rightarrow \sim (\exists Y)\ ON(X, Y)$
  - Any block with no block on has clear top

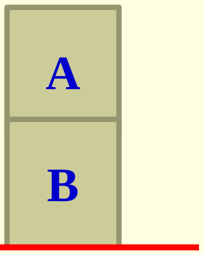
    $(\forall X)\ (\sim (\exists Y)\ ON(Y,X)) \rightarrow CL(X)$

Effect of Unstack operation

- The effect of US(X, Y) is described by the following axiom

  [CL(X, State) Λ ON(X, Y, State)] →
  [HOLD(X, DO(US (X, Y), State)) Λ
  CL(Y, DO(US(X, Y), State))]

  - DO is a function that generates a new state as a result of given action and a state.
- For each operator, set of rules (called frame axioms) are defined where the components of the state are
  - affected by an operator
    - If US(A, B) is executed in state S0, then we can infer that HOLD (A, S1) Λ CL (B, S1) holds true, where S1 is new state after Unstack operation is executed.
  - not affected by an operator
    - If US(A, B) is executed in state S0, B in S1 is still on the table but we can't derive it. So frame rule stating this fact is defined as ONT(Z, S) → ONT(Z, DO(US (A, B), S))



A simple Blocks World Description

ON(A,B,S0)∧ ONTABLE(B,S0) ∧ CLEAR(A,S0)

Contd..

- Advantage of this approach is that
  - simple mechanism of resolution can perform all the operations that are required on the state descriptions.
- Disadvantage is that
  - number of axioms becomes very large for complex problem such as COLOR of block also does not change.
  - So we have to specify rule for each attribute.

    COLOR(X, red, S) $\rightarrow$

    COLOR(X, red, DO(US(Y, Z), s))
- To handle complex problem domain, there is a need of mechanism that does not require large number of explicit frame axioms.

STRIPS Mechanism

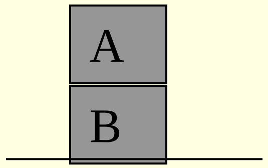
- One such mechanism was used in early robot problem solving system named STRIPS (developed by Fikes, 1971).
- Stands for Stanford Research Institute Problem Solver
- In this approach, each operation is described by three lists.
  - Pre_Cond list contains predicates which have to be true before operation.
  - ADD list contains those predicates which will be true after operation
  - DELETE list contain those predicates which are no longer true after operation
- Predicates not included on either of these lists are assumed to be unaffected by the operation.
- Frame axioms are specified implicitly in STRIPS which greatly reduces amount of information stored.

STRIPS – Style Operators

- S (X, Y)
  - Pre: CL (Y) Λ HOLD (X)
  - Del: CL (Y) Λ HOLD (X)
  - Add: AE Λ ON (X, Y)
- US (X, Y)
  - Pre: ON (X, Y) Λ CL (X) Λ AE
  - Del: ON (X, Y) Λ AE
  - Add: HOLD (X) Λ CL (Y)
- PU (X)
  - Pre: ONT (X) Λ CL (X) Λ AE
  - Del: ONT (X) Λ AE
  - Add: HOLD (X)
- PD (X)
  - Pre: HOLD (X)
  - Del: HOLD (X)
  - Add: ONT (X) Λ AE

Example

ON(A,B)∧ ONTABLE(B) ∧ CLEAR(A) ∧ AE

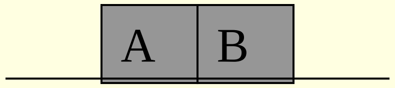


1

UNSTACK(A,B)

2

PUTDOWN(A)



3 Global Database at this point

ONTABLE(B)^CLEAR(A)^CLEAR(B)^ONTABLE(A)

Detecting a solution

- A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transforms the initial problem state into the goal state.
- How will it know when this has been done?
- In simple problem-solving systems, this question is easily answered by a straightforward match of the state descriptions.
- One of the representative systems for planning systems is, predicate logic. Suppose that as a part of our goal, we have the predicate P(x). To see whether P(x) is satisfied in some state, we ask whether we can prove P(x) given the assertions that describe that state and the axioms that define the world model.

Detecting Dead Ends

- As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution.
- The same reasoning mechanisms that can be used to detect a solution can often be used for detecting a dead end.
- If the search process is reasoning forward from the initial state, it can prune any path that leads to a state from which the goal state cannot be reached.
- If search process is reasoning backward from the goal state, it can also terminate a path either because it is sure that the initial state cannot be reached or because little progress is being made.

Repairing an Almost Correct Solution

- The kinds of techniques we are discussing are often useful in solving nearly decomposable problems.
- One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the sub problems separately, and then check that when the sub solutions are combined, they do infect yield a solution to the original problem.

Goal Stack Planning

- One of the earliest techniques is planning using goal stack.
- Problem solver uses single stack that contains
  - sub goals and operators both
  - sub goals are solved linearly and then finally the conjoined sub goal is solved.
- Plans generated by this method will contain
  - complete sequence of operations for solving one goal followed by complete sequence of operations for the next etc.
- Problem solver also relies on
  - A database that describes the current situation.
  - Set of operators with precondition, add and delete lists.

Goal Stack Planning

- To start with goal stack is simply:

ON(C,A)^ON(B,D)^ONTABLE(A)^ONTABLE (D)

- This problem is separate into four sub problems, one for each component of the goal.
- Two of the sub problems ONTABLE(A) and ONTABLE(D) are already true in the initial state.
- Alternative 1: Goal Stack:

  ON(C,A)

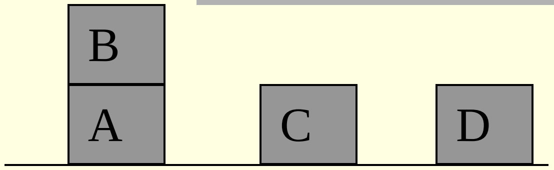
  ON(B,D)

  ON(C,A)^ON(B,D)^OTAD
- Alternative 2: Goal stack:

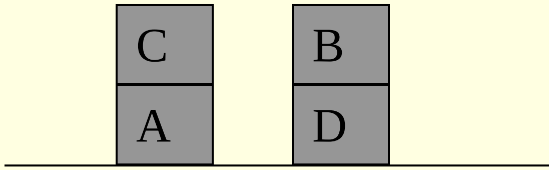
  ON(B,D)

  ON(C,A)

  ON(C,A)^ON(B,D)^OTAD



Start: ON(B,A)^ONTABLE(A) ^ ONTABLE(C) ^ONTABLE(D) ^ARMEMPTY



Goal: ON(C,A)^ON(B,D)^ ONTABLE(A)^ONTABLE(D)

Exploring Operators

- Pursuing alternative 1, we check for operators that could cause ON(C,A)
- Of the 4 operators, there is only one STACK. So it yields:

  STACK(C,A)
  ON(B,D)
  ON(C,A)^ON(B,D)^OTAD
- Preconditions for STACK(C,A) should be satisfied, we must establish them as sub goals:

  CLEAR(A)
  HOLDING(C)
  CLEAR(A)^HOLDING(C)
  STACK(C,A)
  ON(B,D)
  ON(C,A)^ON(B,D)^OTAD
- Here we exploit the Heuristic that if HOLDING is one of the several goals to be achieved at once, it should be tackled last.

Goal stack Planning contd

- Next we see if CLEAR(A) is true. It is not. The only operator that could make it true is UNSTACK(B,A). This produces the goal stack:

  ON(B,A)

  CLEAR(B)

  ARMEMPTY

  ON(B,A)^CLEAR(B)^ARMEMPTY

  UNSTACK(B,A)

  HOLDING(C)

  CLEAR(A)^HOLDING(C)

  STACK(C,A)

  ON(B,D)

  ON(C,A)^ON(B,D)^OTAD

- We see that we can pop predicates on the stack till we reach HOLDING(C) for which we need to find suitable operator

- The operators that might make HOLDING(C) true : PICKUP(C) and UNSTACK(C,x). Without looking ahead, since we cannot tell which of these operators is appropriate, we create two branches of the search tree corresponding to the following goal stacks:

ALT1:	**ALT2:**
ONTABLE(C)	**ON(C,x)**
CLEAR(C)	**CLEAR(C)**
ARMEMPTY	**ARMEMPTY**
ONTABLE(C)^CLEAR(C)^ARMEMPTY	**ON(C,x)^CLEAR(C)^ARMEMPTY**
PICKUP(C)	**UNSTACK(C,x)**
CLEAR(A)^HOLDING(C)	**CLEAR(A)^HOLDING(C)**
STACK(C,A)	**STACK(C,A)**
ON(B,D)	**ON(B,D)**
ON(C,A)^ON(B,D)^OTAD	**ON(C,A)^ON(B,D)^OTAD**

Choosing Alternative

- How should our program choose now between alternative 1 and alternative 2?
  - We can tell that picking up C (alt 1) is better than unstacking it because it is not currently on anything. So to unstack it, we would first have to stack it.This would be waste of effort.
- But how could the program know that?
  - If we pursue the alternative 1, the top element on the goal stack is ONTABLE(C) which is already satisfied, so we pop it off. CLEAR(C) is also satisfied and is popped off.
- The remaining precondition of PICKUP(C) is ARMEMPTY which is not satisfied since HOLDING(B) is true.So we apply the operator STACK(B,D). This makes the Goal stack:

CLEAR(D)
HOLDING(B)
CLEAR(D)^HOLDING(B)
STACK(B,D)
ONTABLE(C)^CLEAR(C)^ARMEMPTY
PICKUP(C)
CLEAR(A)^HOLDING(C)
STACK(C,A)
ON(B,D)
ON(C,A)^ON(B,D)^OTAD

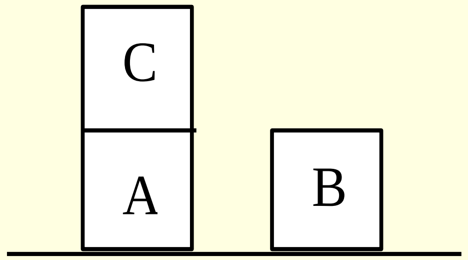
Complete plan

1. UNSTACK(B,A)
2. STACK(B,D)
3. PICKUP(C)
4. STACK(C,A)

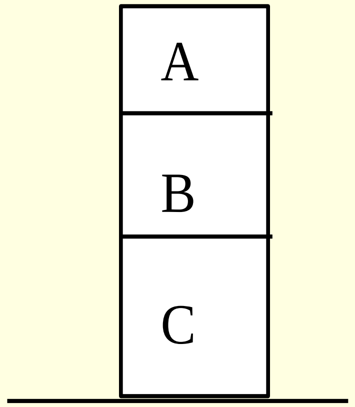
Difficult Problem

- The Goal stack method is not efficient for difficult problems such as Sussman anomaly problem.
- It fails to find good solution.
- Let us consider the Sussman anomaly problem

Initial State (State0)



Goal State



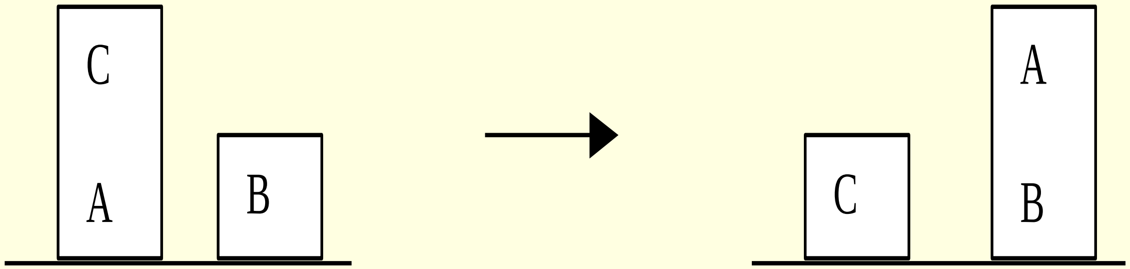
Cont…

Initial State: ON(C, A) Λ ONT(A) Λ ONT(B)

Goal State: ON(A, B) Λ ON(B, C)

- Remove CL and AE predicates for the sake of simplicity.
- To satisfy ON(A, B), following operators are applied

US(C, A) , PD(C), PU(A) and S(A, B)



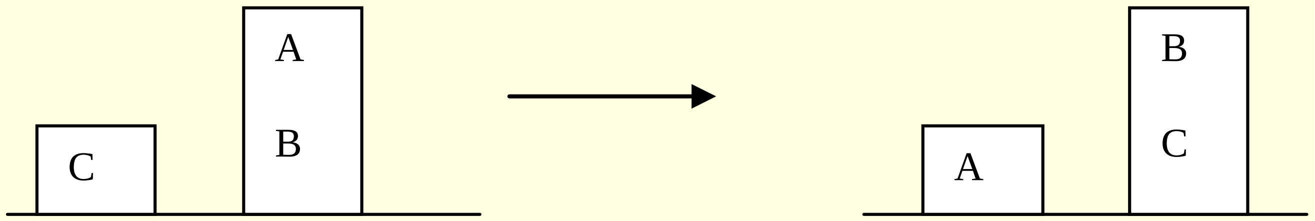
Cont…

State_1: ON(B, A) Λ ONT(C)

- To satisfy ON(B, C), following operators are applied

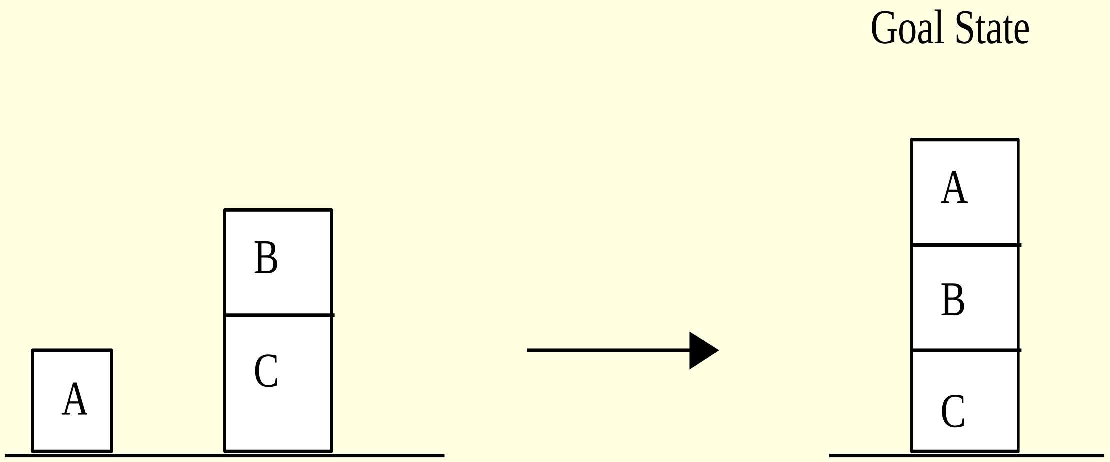
  US(A, B) , PD(A), PU(B) and S(B, C)

State_2: ON(B, C) Λ ONT(A)



Cont…

- Finally satisfy combined goal ON(A, B) Λ ON(B, C).
- Combined goal fails as while satisfying ON(B, C), we have undone ON(A, B).
- Difference in goal and current state is ON(A, B).
- Operations required are PU(A) and S(A, B)



Final Solution

- The complete plan for solution is as follows:

1.	US(C, A)
2.	PD (C)
3.	**PU(A)**
4.	**S(A, B)**
5.	**US(A, B)**
6.	**PD(A)**
7.	PU(B)
8.	S(B, C)
9.	PU(A)
10.	S(A, B)

- Although this plan will achieve the desired goal, but it is not efficient.

Cont…

- In order to get efficient plan, either repair this plan or use some other method.
- Repairing is done by looking at places where operations are done and undone immediately, such as S(A, B) and US(A, B).
- By removing them, we get
    1. US(C, A)
    2. PD (C)
    3. PU(B)
    4. S(B, C)
    5. PU(A)
    6. S(A, B)

Self

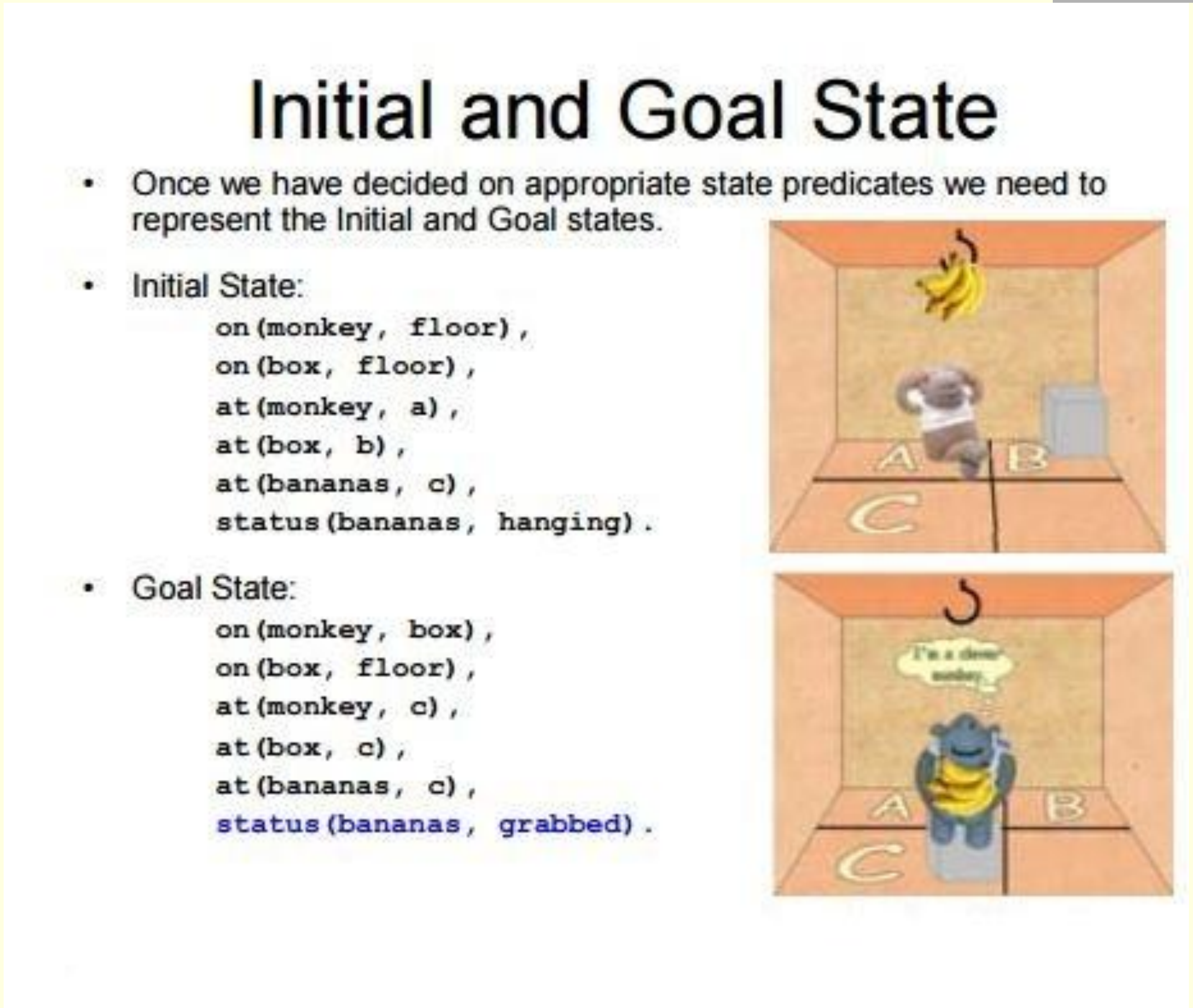
- Nonlinear Planning using Constraint Posting
- Hierarchical Planning
- Reactive System

Example

The monkey-and-bananas problem is faced by a monkey in a laboratory with some bananas hanging out of reach from the ceiling. A box is available that will enable the monkey to reach the bananas if he climbs on it. There are few labelled positions at which monkey, bananas and box can be. Moreover each object can have an height (e.g. bananas on the ceiling are high and box is low).

Initially, the monkey is at A, the bananas at B, and the box at C. The monkey and box have height *Low*, but if the monkey climbs onto the box he will have height *High*, the same as the bananas.

The actions available to the monkey include *Go* from one place to another, *Push* an object from one place to another, *ClimbUp* onto or *ClimbDown* from an object, and *Grasp* or *Ungrasp* an object. Grasping results in holding the object if the monkey and object are in the same place at the same height.



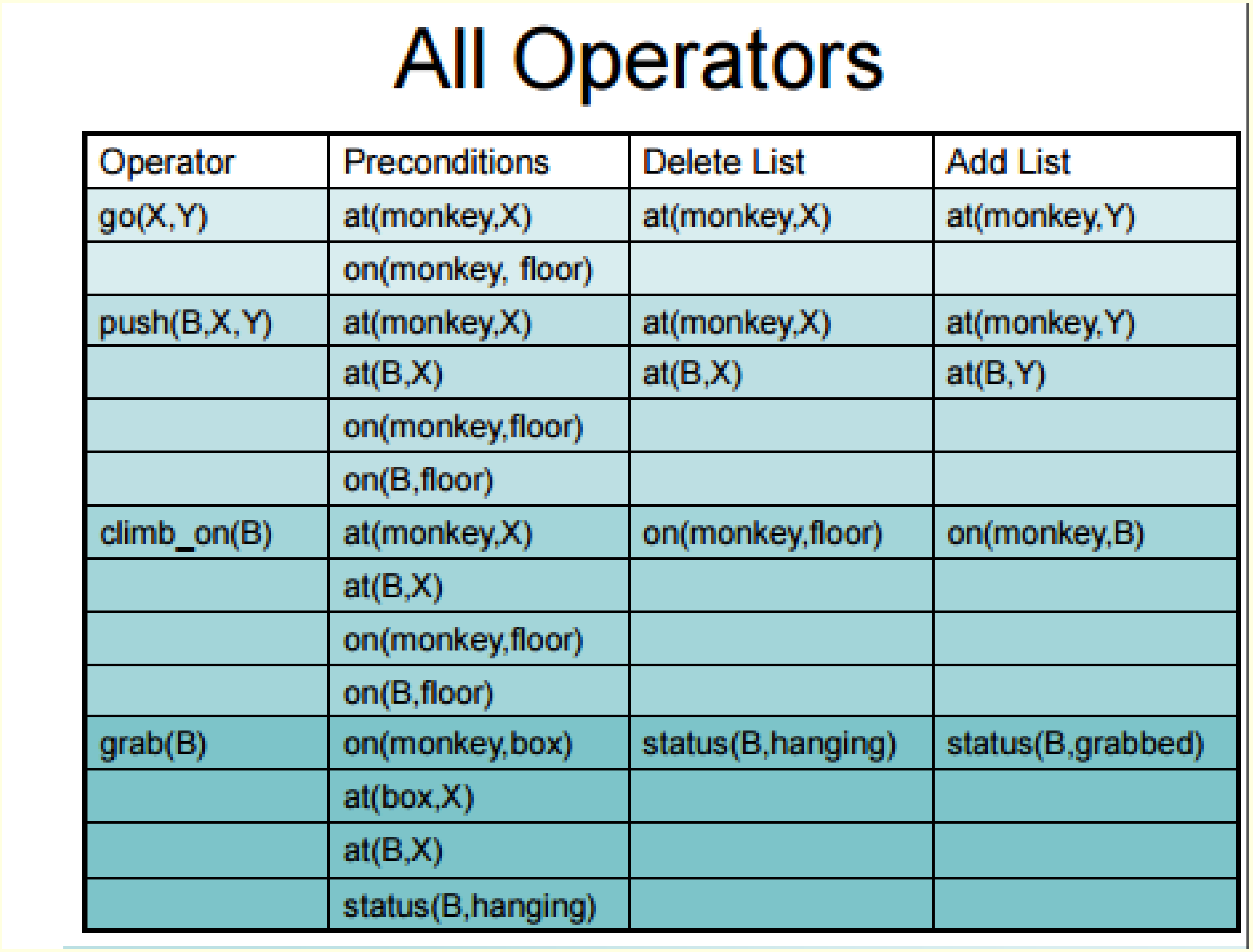
Initial and Goal State

- Once we have decided on appropriate state predicates we need to represent the Initial and Goal states.
- Initial State:

```
on(monkey, floor),
on(box, floor),
at(monkey, a),
at(box, b),
at(bananas, c),
status(bananas, hanging).
```

- Goal State:

```
on(monkey, box),
on(box, floor),
at(monkey, c),
at(box, c),
at(bananas, c),
status(bananas, grabbed).
```



All Operators

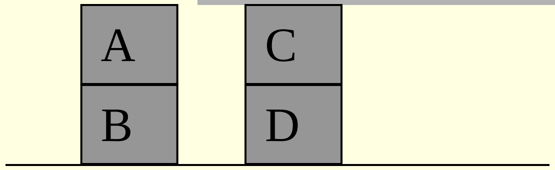
Operator	Preconditions	Delete List	Add List
go(X,Y)	at(monkey,X)	at(monkey,X)	at(monkey, Y)
	on(monkey, floor)		
push(B,X,Y)	at(monkey,X)	at(monkey,X)	at(monkey, Y)
	at(B,X)	at(B,X)	at(B,Y)
	on(monkey,floor)		
	on(B,floor)		
climb_on(B)	at(monkey,X)	on(monkey,floor)	on(monkey,B)
	at(B,X)		
	on(monkey,floor)		
	on(B,floor)		
grab(B)	on(monkey,box)	status(B,hanging)	status(B,grabbed)
	at(box,X)		
	at(B,X)		
	status(B,hanging)		

Solution

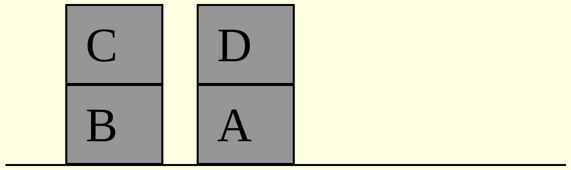
- go(A,B)
- push(Box,B,C)
- climb_on(Box,C)
- grab(Bananas)

Problem

- Show how the STRIPS would solve this problem?



Start:
ON(A,B)^ON(C,D)^ONTABLE(B) ^ ONTABLE(A) ^ARMEMPTY



Goal:
ON(C,B)^ON(D,A)^ONTABLE(B)^ ONTABLE(A)