# Introduction to AI

## Lecture 9
## Planning with State Space Search

**Dr. Tamal Ghosh**
**Department of CSE**
**Adamas University**

# Planning with State-Space Search

- The most straightforward approach of planning algorithm, is state-space search
  - Forward state-space search (Progression)
  - Backward state-space search (Regression)
- The descriptions of actions in a planning problem, and specify both preconditions and effects
- It is possible to search in both direction: either forward from the initial state or backward from the goal
- We can also use the explicit action and goal representations, to derive effective heuristics automatically.

# FSSS: Problem Formulation

- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

# FSSS Progression

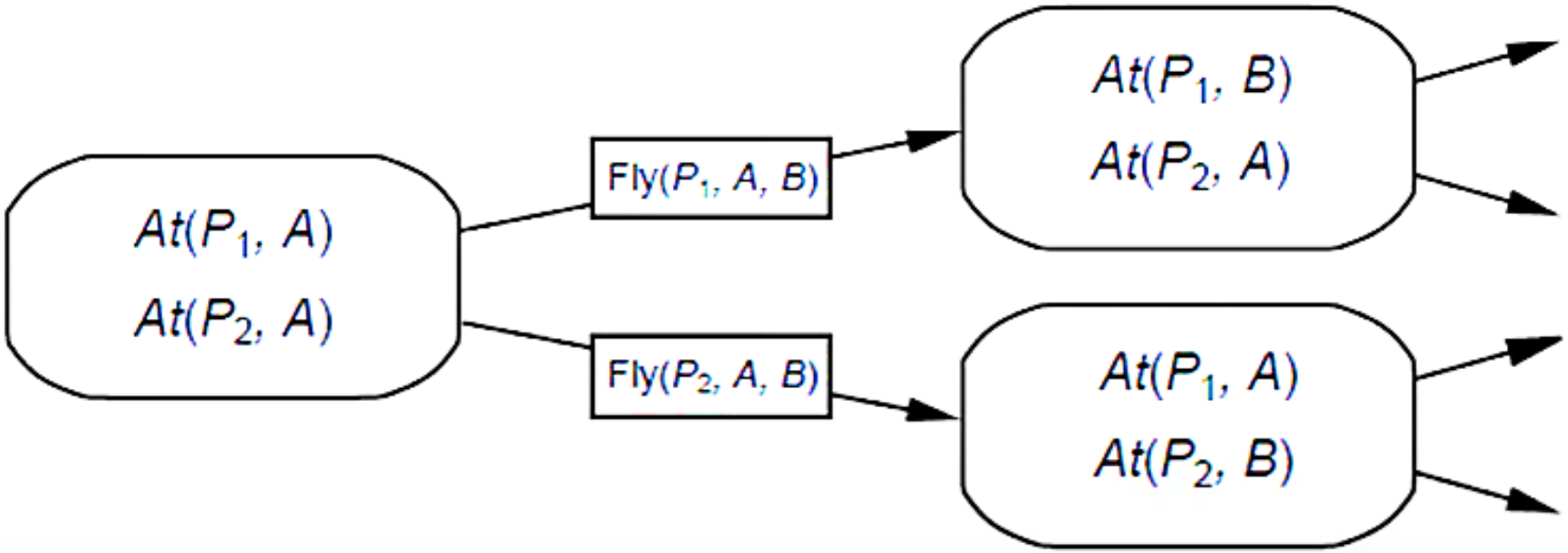
- From initial state, search forward by selecting operators whose preconditions can be unified with literals in the state
- New state includes positive literals of effect; the negated literals of effect are deleted
- Search forward until goal unifies with resulting state
- This is forward state-space search using STRIPS operators

# Example

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

# FSSS Flow

- Starting from the initial state and using the problem's actions to search forward for the goal state.



# Algorithm FSSS

- (1) compute whether or not a state is a goal state,
- (2) find the set of all actions that are applicable to a state, and
- (3) compute a successor state, that is the result of applying an action to a state
- Algorithm takes as input the statement P = (O, s0, g) of a planning problem P. (O contains a list of actions)
- If P is solvable, then Forward-search(O, s0, g) returns a solution plan; otherwise it returns failure.

# Algorithm FSSS

1. Forward-search(O, s0, g)
2. $s \leftarrow s0$
3. $\pi \leftarrow$ the empty plan
4. loop
    1. if s satisfies g then return $\pi$
    2. applicable $\leftarrow$ {a | a is a ground instance of an operator in O, and precond(a) is true in s}
    3. if applicable = $\emptyset$ then return failure
    4. Non-deterministically choose an action a $\in$ applicable
    5. $s \leftarrow \gamma(s, a)$
    6. $\pi \leftarrow \pi.a$

# Problems with FSSS

- Forward state-space search is too inefficient to be practical.
- First, forward search does not address the irrelevant action problem—all applicable actions are considered from each state.
- Second, the approach quickly bogs down without a good heuristic.
- Consider an air cargo problem with 10 airports, Each airport has 5 planes and 20 cargo. The goal is to move all the cargo from Airport A to Airport B. There is a simple solution: load the 20 pieces of cargo into one plane at A, fly the plane to B, and unload the cargo. But, finding the solution can be difficult because the avg. branching factor is: that each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded). On average, let's say there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about $1000^{41}$ nodes.

# Problem Formulation for Regression/BSSS

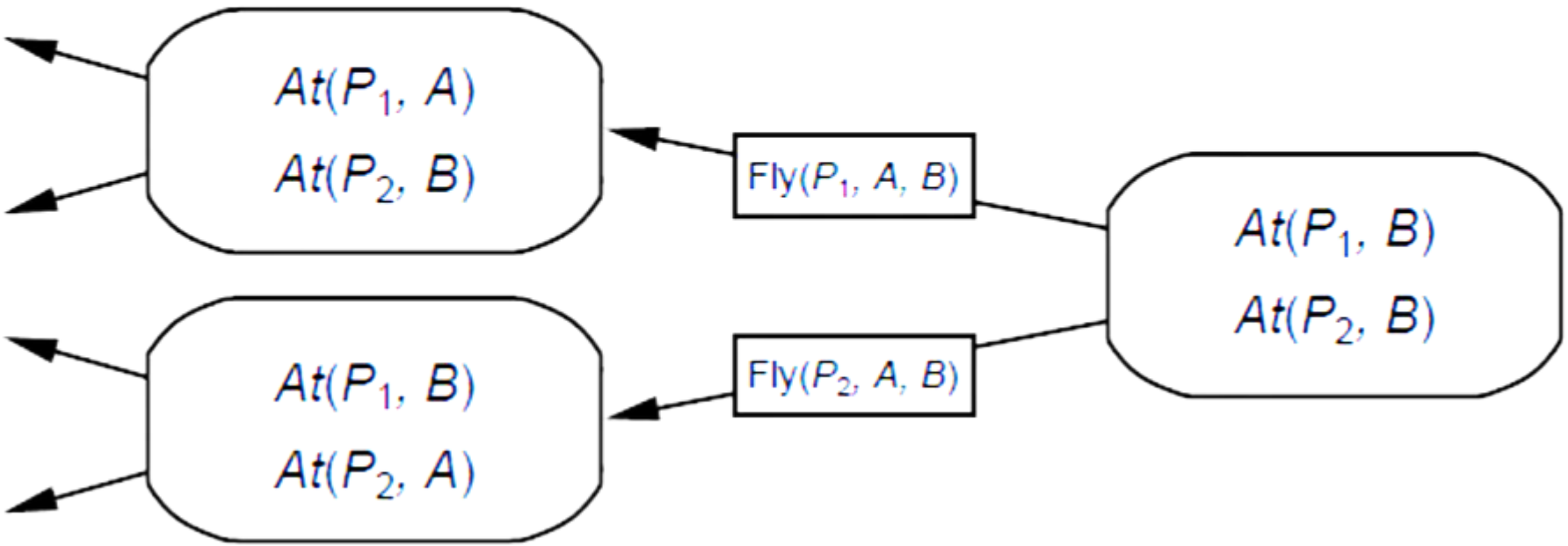
- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

# Regression BSSS

- The goal state must unify with at least one of the positive literals in the operator's effect
- Its preconditions must hold in the previous situation, and these become subgoals which might be satisfied by the initial conditions
- Perform backward chaining from goal
- Again, this is just state-space search using STRIPS operators

# BSSS

- Backward state search starting from the goal state(s)
- Using the inverse of the actions to search backward for the initial state.



# BSSS Algorithm

- Start at the goal,
- Test the goal is initial state, otherwise
- Apply inverses of the planning operators to produce subgoals,
- The algorithm will stop, if we produce a set of subgoals that satisfies the initial state.

# BSSS Algorithm

1. Backward-search(*O*, *s*0, *g*)
2. $\pi$ the empty plan
3. loop
    1. if *s*0 satisfies *g* then return $\pi$
    2. *applicable* $\leftarrow$ {*a* | *a* is a ground instance of an operator in *O* that is relevant for *g*}
    3. if *applicable* = $\emptyset$ then return failure
    4. Non-deterministically choose an action *a* $\epsilon$ *applicable*
    5. $\pi \leftarrow a.\ \pi$
    6. $g \leftarrow \gamma\text{-}1(g, a)$

# Advantages of BSSS

- Backward planning is sometimes better as we start with the goal and thus the visibility of the goal helps us choose the moves. It is preferred when the branching factor, in the reverse direction, is less than in the forward direction.
- It is better to move from a smaller to a larger number of states. It is easier to start from an unknown goal state to an intermediate state from where the path to the start state is known.
- A database or a big data inquiry requires us to reason back from a query to a solution.
- If there is a need for an explanation facility, backward reasoning is a must. E.g. if the expert system suggests that the engine of the car is to be replaced, the mechanic would like to learn why. The program can only explain if it can reason back from the answer it gave.

# Heuristics for State-Space Search

- How to find an admissible heuristic estimate?
  - Distance from a state to the goal?
  - Look at the effects of the actions and at the goals and guess how many actions are needed
- NP-hard
- Relaxed problem
- Subgoal independence assumption:
- The cost of solving a conjunction of subgoals, is approximated by the sum of the costs of solving each subgoal independently

# Relaxation Problem

- Idea: removing all preconditions from the actions

- Which implies, the number of steps required to solve a conjunction of goals, and the number of unsatisfied goals
  - There may be two actions,
  - Some actions deletes the goal literal achieved by the other actions
  - some action may achieve multiple goals
- Combining relaxation with subgoal $\rightarrow$ exact # of unsatisfied goals

# Removing Negative Effects

- Generate a relaxed problem by removing negative effects:
  - Empty-delete-list heuristic
  - Quite accurate, but computing required, it involves running a simple planning algorithm