Artificial Intelligence

Planning

MONTANA STATE UNIVERSITY

1

Mountains & Minds

The Planning Problem

- Given:
  - A set of operators
  - An initial state description
  - A goal state description or predicate
- Find
  - A sequence of operator instances such that performing them in order from the initial state will modify the state to achieve the goal.

MONTANA STATE UNIVERSITY

2

Mountains & Minds

Typical Assumptions

- Each action is indivisible (atomic).
- No concurrent actions allowed.
- Actions are deterministic (i.e., no uncertainty in outcome).
- Agent is sole cause of change to world.
- Agent is omniscient.
- *Closed World Assumption*—Everything known to be true is included in state description. Otherwise it's false.

MONTANA STATE UNIVERSITY

3

Mountains & Minds

Possible Approaches

- Situation Calculus:
  – Augment FOL to reason about actions in time.
  – Add situation variables.
  – Define predicates as a function of situation.
  – Add new function, result(a,s), that returns new situation given action and current situation.
  – Example: Definition of action "agent-walks-to-location".
    - $\forall x\ \forall y\ \forall s\ (at(Agent,x,s) \wedge \neg trapped(Agent,s) \Rightarrow at(Agent,y,result(walk(y),s))$

MONTANA STATE UNIVERSITY

4

Mountains & Minds

Possible Approaches

- State-Space Search:
  – We already need an initial state.
  – Goal test checks to see if goal state is achieved.
  – Successor function based on set of operators.
  – Once goal is found, the plan is simply the sequence of operators on path from initial state to goal state.
  – Unfortunately, this approach relies *totally* on algorithm and ignores information inherent in state (esp. goal) and operator descriptions.

MONTANA STATE UNIVERSITY

5

Mountains & Minds

Opening States/Operators

- State/situation representation is a conjunction of ground literals (i.e., facts with no variables).
- Goal is a state where all literals are positive (i.e., all true).
- "STRIPS" Operators have three parts:
  – Operator/action name
  – Preconditions: what needs to be true
  – Effects: what is now true AND false

MONTANA STATE UNIVERSITY

6

Mountains & Minds

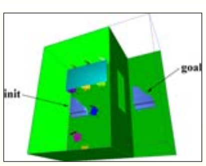
Example Operator Description

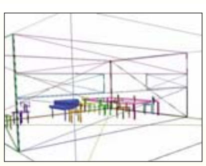
- Suppose we want to describe an operator that picks up an object.
  - **Name:** pickup(x)
  - **Preconditions:** ontable(x), clear(x), handempty
  - **Add List:** holding(x)
  - **Delete List:** ontable(x), clear(x), handempty
- Alternatively, effects could be
  - **Effect List:** holding(x), ¬ontable(x), ¬clear(x), ¬handempty

Action Description Language

- Alternative method for representing actions.
- States include both positive and negative literals (*open* world assumption).
- Quantifiers and disjunction now allowed in goals.
- Equality predicate built in.
- Variables can be typed (reduces search space).

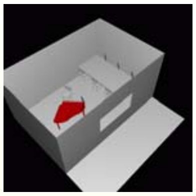
Example: Moving a Piano 1

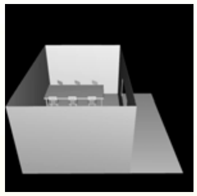




Goal: Move the piano through the window.

Example: Moving a Piano 2





Goal: Move the piano through the window.

Example: Air Cargo

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

Example: Spare Tire

$$
\begin{aligned}
&Init(At(Flat, Axle) \wedge At(Spare, Trunk)) \\
&Goal(At(Spare, Axle)) \\
&Action(Remove(Spare, Trunk), \\
&\quad \text{PRECOND}: At(Spare, Trunk) \\
&\quad \text{EFFECT}: \neg At(Spare, Trunk) \wedge At(Spare, Ground)) \\
&Action(Remove(Flat, Axle), \\
&\quad \text{PRECOND}: At(Flat, Axle) \\
&\quad \text{EFFECT}: \neg At(Flat, Axle) \wedge At(Flat, Ground)) \\
&Action(PutOn(Spare, Axle), \\
&\quad \text{PRECOND}: At(Spare, Ground) \wedge \neg At(Flat, Axle) \\
&\quad \text{EFFECT}: \neg At(Spare, Ground) \wedge At(Spare, Axle)) \\
&Action(LeaveOvernight, \\
&\quad \text{PRECOND}: \\
&\quad \text{EFFECT}: \neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk) \\
&\qquad\qquad \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))
\end{aligned}
$$

Planning as Search

- Generally, planning is regarded as a search problem due to the natural description.
- Two main approaches suggest themselves:
  - **Situation-Space Search:** Search space is space of all possible *states*. Plan is sequence of operators on path from start to goal.
  - **Plan-Space Search:** Search space is space of all possible *plans* (or partial plans).

MONTANA STATE UNIVERSITY

13

Mountains & Minds

Situation-Space Planning

- Two approaches
  - Progression Planning
    - Forward chaining
    - Use standard search techniques (BFS, A*, etc.)
    - State-space search except with STRIPS operators
  - Regression Planning
    - Backward chaining
    - Historically More efficient that progression planning
    - Must consider pre-conditions *and* add-list of effects
- All search methods now benefit from heuristics.

MONTANA STATE UNIVERSITY

14

Mountains & Minds

STRIPS

- Uses a goal stack instead of the regression algorithm.
- Maintains current state throughout.
- Approach:
  - Pick an order for achieving each of the goals.
  - When a goal is popped from stack, push operator that adds goal, followed by its preconditions.
  - Repeat until all preconditions solved.

MONTANA STATE UNIVERSITY

15

Mountains & Minds

STRIPS Algorithm

procedure STRIPS(I, G_1, …, G_n, OPS)
  push(**achieve**(G_1, …, G_n), GoalStack) **do**
    if empty(GoalStack) **then** success
    $N \leftarrow$ pop(GoalStack)
    if N of form 'achieve(g_1, …, g_m)' **then**
      if N true **in** I, **continue**
      if N previously tried, fail
      Mark N attempted
      push(N, GoalStack)
      order N's goals g_i
      push(achieve(g_i), GoalStack) in order
    else if N of form 'achieve(g)' **then**
      if N true **in** I, **continue**
      choose operator O **from** OPS that may add g
      if none, fail
      choose unify(O, I) so O adds g
      push(apply(O), GoalStack)
      push(achieve(preconditions(O)), GoalStack)
    else if N of form 'apply(O)' **then** $I \leftarrow$ apply(O, I)

MONTANA STATE UNIVERSITY

16

Mountains & Minds

Example

- Operators
  - pickup(x)
    - P: ontable(x), clear(x), handempty
    - E: holding(x), ¬ontable(x), ¬clear(x), ¬handempty
  - putdown(x)
    - P: holding(x)
    - E: ontable(x), clear(x), handempty, ¬holding(x)
  - stack(x,y)
    - P: holding(x), clear(y)
    - E: on(x,y), clear(x), handempty, ¬ holding(x), ¬clear(y),
  - unstack(x,y)
    - P: clear(x), on(x,y), handempty
    - E: holding(x), clear(y), ¬clear(x), ¬on(x,y), ¬handempty

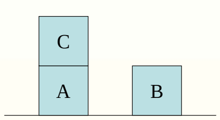
MONTANA STATE UNIVERSITY

17

Mountains & Minds

Example

Initial State:
clear(B)
clear(C)
on(C,A)
ontable(A)
ontable(B)
handempty



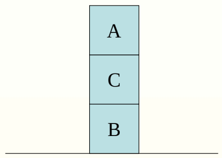
MONTANA STATE UNIVERSITY

18

Mountains & Minds

Example

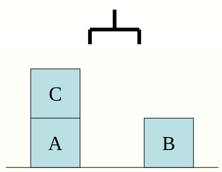
Goal State:
on(A,C)
on(C,B)



Step 1

State:
clear(B)
clear(C)
on(C,A)
ontable(A)
ontable(B)
handempty

Plan:

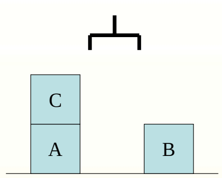


GoalStack:
achieve(on(C,B),on(A,C))

Step 2

State:
clear(B)
clear(C)
on(C,A)
ontable(A)
ontable(B)
handempty

Plan:

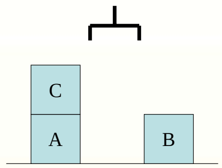


GoalStack:
achieve(on(C,B)) **Subgoals**
achieve(on(A,C))
achieve(on(C,B),on(A,C))

Step 3

State:
clear(B)
clear(C)
on(C,A)
ontable(A)
ontable(B)
handempty

Plan:



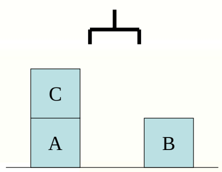
GoalStack:
achieve(clear(B),holding(C)) ← **preconditions**
apply(stack(C,B)) ← **operator**
achieve(on(A,C))
achieve(on(C,B),on(A,C))

22

Step 4

State:
clear(B)
clear(C)
on(C,A)
ontable(A)
ontable(B)
handempty

Plan:



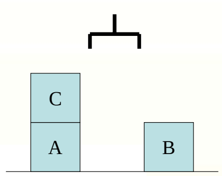
GoalStack:
achieve(holding(C)) ← **Subgoals**
achieve(clear(B)) ←
achieve(clear(B),holding(C))
apply(stack(C,B))
achieve(on(A,C))
achieve(on(C,B),on(A,C))

23

Step 5

State:
clear(B)
clear(C)
on(C,A)
ontable(A)
ontable(B)
handempty

Plan:



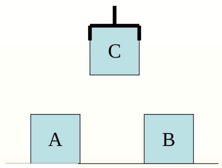
GoalStack:
achieve(handempty,clear(C),on(C,y))
apply(Unstack(C,y))
achieve(clear(B))
achieve(clear(B),holding(C))
apply(stack(C,B))
achieve(on(A,C))
achieve(on(C,B),on(A,C))

24

Step 6

State:
clear(B)
ontable(A)
ontable(B)
holding(C)
clear(A)

Plan:
Unstack(C,A)



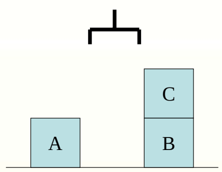
GoalStack:
achieve(clear(B))
achieve(clear(B),holding(C))
apply(stack(C,B))
achieve(on(A,C))
achieve(on(C,B),on(A,C))

MONTANA STATE UNIVERSITY
25
Mountains & Minds

Step 7

State:
ontable(A)
ontable(B)
clear(A)
on(C,B)
clear(C)
handempty

Plan:
Unstack(C,A)
Stack(C,B)



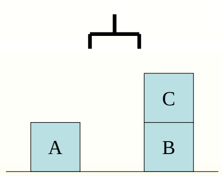
GoalStack:
achieve(on(A,C))
achieve(on(C,B),on(A,C))

MONTANA STATE UNIVERSITY
26
Mountains & Minds

Step 8

State:
ontable(A)
ontable(B)
clear(A)
on(C,B)
clear(C)
handempty

Plan:
Unstack(C,A)
Stack(C,B)



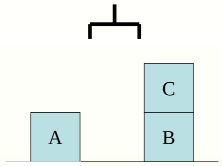
GoalStack:
achieve(clear(C),holding(A))
apply(stack(A,C))
achieve(on(C,B),on(A,C))

MONTANA STATE UNIVERSITY
27
Mountains & Minds

Step 9

State:
ontable(A)
ontable(B)
clear(A)
on(C,B)
clear(C)
handempty

Plan:
Unstack(C,A)
Stack(C,B)

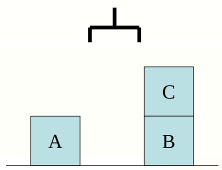


GoalStack:
achieve(holding(A))
achieve(clear(C))
achieve(clear(C),holding(A))
apply(stack(A,C))
achieve(on(C,B),on(A,C))

Step 10

State:
ontable(A)
ontable(B)
clear(A)
on(C,B)
clear(C)
handempty

Plan:
Unstack(C,A)
Stack(C,B)

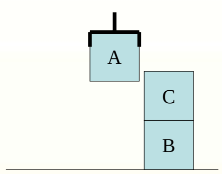


GoalStack:
achieve(ontable(A),clear(A),handempty)
apply(pickup(A))
achieve(clear(C))
achieve(clear(C),holding(A))
apply(stack(A,C))
achieve(on(C,B),on(A,C))

Step 11

State:
ontable(B)
on(C,B)
clear(C)
holding(A)

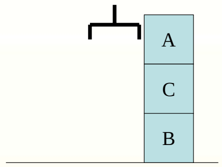
Plan:
Unstack(C,A)
Stack(C,B)
Pickup(A)



GoalStack:
achieve(clear(C))
achieve(clear(C),holding(A))
apply(stack(A,C))
achieve(on(C,B),on(A,C))

Step 12

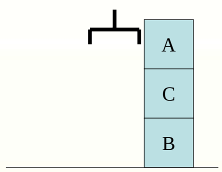
State:	**Plan:**
ontable(B)	Unstack(C,A)
on(C,B)	Stack(C,B)
on(A,C)	Pickup(A)
clear(A)	Stack(A,C)
handempty	



GoalStack:
achieve(on(C,B),on(A,C))

Step 13

State:	**Plan:**
ontable(B)	Unstack(C,A)
on(C,B)	Stack(C,B)
on(A,C)	Pickup(A)
clear(A)	Stack(A,C)
handempty	



GoalStack:

DONE

Progression Planning

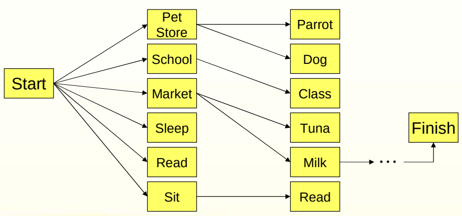
- **Initial State:** The initial state for planning corresponds to the initial search state.
- **Actions:** Application actions are those whose preconditions are met by the current state. Successor state results from applying EFFECTS list.
- **Goal Test:** Checks whether goal state is achieved.
- **Step Cost:** Typically unit cost, but any cost function can be used.
- Any complete search algorithm can be used to solve.

Irrelevant Actions

- Forward (progression) planning considers all applicable actions from a state.
- Many of these actions are likely to be irrelevant for the problem.
- (Potentially) high branching factor can bog down search.
- Can improve the situation with a good heuristic (as in any informed search).

Branching Factor

- Consider task "get milk, bananas, and cordless drill."
- Progression planning tends to have trouble.



Regression Planning

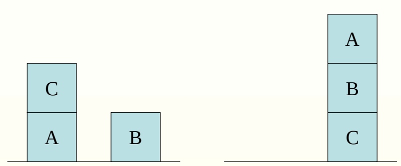
- Helps with the irrelevant action problem.
- Only relevant actions considered based on definition:
  - An action is *relevant* to a conjunctive goal if it achieves one of the conjuncts of the goal.
- Regression involves matching EFFECTS in rules satisfying goals (thus making relevant).
  - First identify states from which applying an action will yield goal (or goal conjunct).
  - Add new actions preconditions to goal description.
  - Delete positive effects from goal description.
- Consistent actions do not *undo* any desired literal.

Goal Interaction

- Most planners assume goals to be achieved are independent.
- Sometimes this is not true.
- Resulting interaction of goals can result in infinite loops or the inability to create a plan.

Sussman Anomaly

Initial State **Goal State**



Solving on(A,B) will be undone when solving on(B,C)
Solving on(B,C) will be undone when solving on(A,B)

Heuristics

- Both progression and regression planning benefit greatly from good heuristics.
- Neither progression nor regression planning are efficient without heuristics.
- In fact, planning is PSPACE-complete unless limited to only positive preconditions and single effects.
- Recent advances in defining heuristics have enabled many practical planning problems to be solved.

Relaxation

- Note that we have explicit representations of both preconditions and effects.
- We will modify these to derive a "simpler" planning problem.
- One approach—remove all preconditions from actions (i.e., make all actions applicable at all times).
  - $h(n)$ = # unsatisfied subgoals? Not quite.
  - Negative interactions—one action canceling out another.
  - Single actions may achieve multiple goals.

MONTANA STATE UNIVERSITY 40 Mountains & Minds

Subgoal Independence

- Assume pure divide-and-conquer will work
- Estimate cost of solving as sum of cost of solving each subgoal independently.
- Note that this can be either optimistic or pessimistic.
  - Optimistic when there are negative interactions between subplans that must be corrected.
  - Pessimistic when subplans contain redundant actions (e.g., actions from two subplans could be replaced with another single action).

MONTANA STATE UNIVERSITY 41 Mountains & Minds

Minimal Set Cover

- First relax the problem by ignoring preconditions.
- Then further relax the problem by removing negative effects.
- Finally, count the minimum number of actions required such that the union of the actions' positive effects satisfies the goal.
  - $Goal(A \wedge B \wedge C)$
  - $Action(X, \text{EFFECT: } A \wedge P)$
  - $Action(Y, \text{EFFECT: } B \wedge C \wedge Q)$
  - $Action(Z, \text{EFFECT: } B \wedge P \wedge Q)$
- Minimal set cover given by $\{X, Y\} = 2$

MONTANA STATE UNIVERSITY 42 Mountains & Minds

Empty Delete List

- Can also relax problems by removing negative effects without removing preconditions.
  - Example: If action has effect $A \wedge \neg B$, it just has effect A in relaxed problem.
- Negative interactions no longer matter since no action can delete literals achieved by another action.
- Requires solving the new planning problem.
- Heuristic is just number of steps to solve the problem.
- In practice, this approach is usually worthwhile.

MONTANA STATE UNIVERSITY

43

Mountains & Minds

Total Order vs Partial Order

- **Total-Order Planner:** a planner that maintains a partial solution as a totally-ordered list of steps found so far.
  - Also known as a "linear planner."
- **Partial-Order Planner:** a planner that only represents partial-order constraints on steps.
  - Also known as a "non-linear planner."
- STRIPS is a total-order planner.

MONTANA STATE UNIVERSITY

44

Mountains & Minds

Principle of Least Commitment

- Never make a choice unless required to do so.
- In planning, never order the steps unless the order is required.
- Thus, ordering constraints define order of steps in a plan.
- Leads to *partial-order* planning.

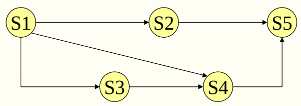
MONTANA STATE UNIVERSITY

45

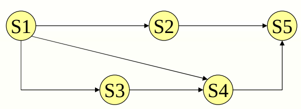
Mountains & Minds

Representing Partial-Order Plans

- Represented as a graph.
- Each node is a step in the plan.
- Each arc represents a temporal constraint between two steps.



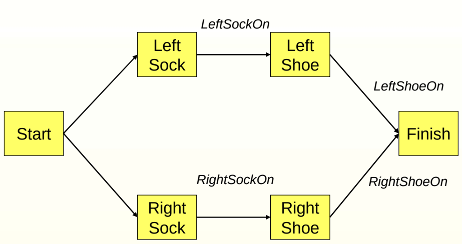
Example



Represents three total-order plans:

1. [S1, S2, S3, S4, S5]
2. [S1, S3, S2, S4, S5]
3. [S1, S3, S4, S2, S5]

Example 2—Partial Order Plan



Example 2—Total Order Plans

Start → Left Sock → Left Shoe → Right Sock → Right Shoe → Finish

Start → Left Sock → Right Sock → Left Shoe → Right Shoe → Finish

Start → Left Sock → Right Sock → Right Shoe → Left Shoe → Finish

Start → Right Sock → Right Shoe → Left Sock → Left Shoe → Finish

Start → Right Sock → Left Sock → Right Shoe → Left Shoe → Finish

Start → Right Sock → Left Sock → Left Shoe → Right Shoe → Finish

Components

- Each partial order plan has the following components:
  - A set of ***actions*** that make up the plan.
  - A set of ***ordering constraints*** of the form "*A before B*" where *A* and *B* are actions. Note this does not require *A* to be *immediately* before *B*.
  - A set of ***causal links*** of the form "*A achieves p for B*" where *p* is a predicate that is now satisfied by performing action *A*. Note that *p* is also a precondition for *B* that is now satisfied.
    - Note: Once *p* is satisfied, a new action *C* cannot be added that conflicts with this causal link.
  - A set of ***open preconditions***, i.e., preconditions not achieved by some action in the plan.

Approach

- Search in *plan-space*.
- Start node in plan-space consists of a plan with two connected "pseudo-nodes."
  - Start
    - P: None
    - E: all positive literals defining initial state
  - Finish
    - P: literals defining the conjunctive goal
    - E: None

Searching in Plan-Space

- There are two main reasons why a plan may not be a solution.
  - Unsatisfied goal
    - There is a goal or subgoal that is not satisfied by the current plan steps.
  - Possible threat
    - A plan step could cause the undoing of a needed goal if that step is done at the wrong time.
- Define plan modification operators to detect and fix these problems.

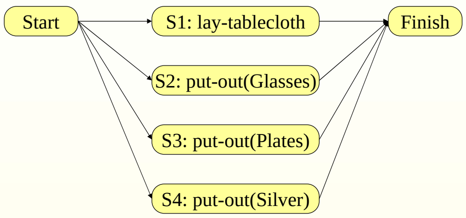
Example–Setting the Table

- Goal: {on(Tablecloth), out(Glasses), out(Plates), out(Silver)}
- Initial State: clear(Table)
- Operators
  - lay-tablecloth
    - P: clear(Table)
    - E: on(Tablecloth), ¬clear(Table)
  - put-out(x)
    - P: None
    - E: out(x), ¬clear(Table)

Step 1

Step 2

Insert the 4 unsolved goals in Finish

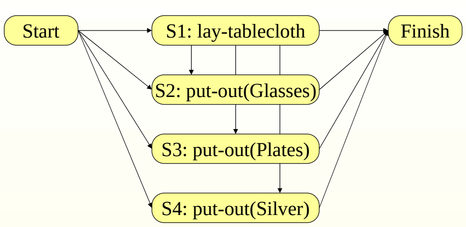


Threat

- Precondition to S1 is clear(Table).
- S2, S3, and S4 do not have this as a precondition.
- Performing S2, S3, or S4 before S1 would prevent S1 from ever being performed.
- Need to add a temporal constraint to prevent this from happening.

Step 3

Add temporal constraints



Any total order satisfying above is legal; therefore, this is a complete plan.

Partial-Order Planning Algorithm

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

    plan ← MakeMinimalPlan(*initial*, *goal*)
    do
        if Solution?(*plan*) **return** *plan*
        S_{need}, *pre-S* ← SelectSubgoal(*plan*)
        ChooseOp(*plan*, *operators*, S_{need}, *pre-S*)
        ResolveThreats(*plan*)
    end_do

Partial-Order Planning Algorithm

function SelectSubgoal(*plan*)
    returns S_{need}, *pre-S*

    pick a plan step S_{need} **from** Steps(*plan*) **with**
        a precondition *pre-S* not yet achieved
    return S_{need}, *pre-S*

Partial-Order Planning Algorithm

procedure ChooseOp(*plan*, *operators*, S_{need}, *pre-S*)

    choose step S_{add} **from** *operators* **or** Steps(*plan*)
        that has *pre-S* as an effect.
    if no such step, fail
    add link S_{add} -(*pre-S*)-> S_{need} **to** Links(*plan*)
    add constraint $S_{add} < S_{need}$ **to** Orderings(*plan*)
    if S_{add} is newly added step from *operators*
        add S_{add} **to** Steps(*plan*)
        add *Start* < S_{add} < *Finish* **to** Orderings(*plan*)
    end_if

Partial-Order Planning Algorithm

procedure ResolveThreats(*plan*)

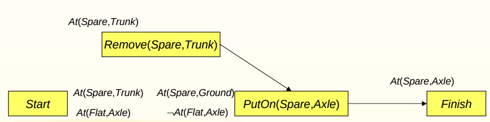
for each S_{threat} threatening link S_i -(*pre-S*)-> S_j
  in Links(*plan*) **do**
  choose either
    Demote: Add $S_{threat} < S_i$ **to** Orderings(*plan*)
    Promote: Add $S_j < S_{threat}$ **to** Orderings(*plan*)
  if not Consistent(*plan*), fail
end_do

Changing a Flat Tire

Init(*At*(*Flat*,*Axle*) ∧ *At*(*Spare*,*Trunk*))
Goal(*At*(*Spare*,*Axle*))
Action(*Remove*(*Spare*,*Trunk*),
  PRECOND: *At*(*Spare*,*Trunk*)
  EFFECT: ¬*At*(*Spare*,*Trunk*) ∧ *At*(*Spare*,*Ground*))
Action(*Remove*(*Flat*,*Axle*),
  PRECOND: *At*(*Flat*,*Axle*)
  EFFECT: ¬*At*(*Flat*,*Axle*) ∧ *At*(*Flat*,*Ground*))
Action(*PutOn*(*Spare*,*Axle*),
  PRECOND: *At*(*Spare*,*Ground*) ∧ ¬*At*(*Flat*,*Axle*)
  EFFECT: ¬*At*(*Spare*,*Ground*) ∧ *At*(*Spare*,*Axle*))
Action(*LeaveOvernight*,
  PRECOND:
  EFFECT: ¬*At*(*Spare*,*Ground*) ∧ ¬*At*(*Spare*,*Axle*) ∧ ¬*At*(*Spare*,*Trunk*)
    ∧ ¬*At*(*Flat*,*Ground*) ∧ ¬*At*(*Flat*,*Axle*))

Changing a Flat Tire

- Start with the initial plan, containing a *Start* action (effects: *At*(*Spare*,*Trunk*) ∧ *At*(*Flat*,*Axle*)), and a *Finish* action (precondition: *At*(*Spare*,*Axle*)).
- First, pick the only open precondition, namely *At*(*Spare*,*Axle*). The only applicable action is *PutOn*(*Spare*,*Axle*).
- Next, pick the *At*(*Spare*,*Ground*) precondition of the *PutOn* action. Again, there is only one applicable action—*Remove*(*Spare,Trunk*).



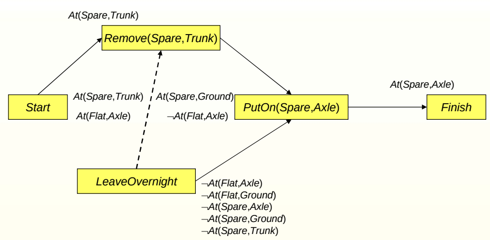
Changing a Flat Tire

- Now pick ¬*At*(*Flat*,*Axle*) precondition of the *PutOn* action.
- Suppose we choose *LeaveOvernight* as the action.
- The effect includes ¬*At*(*Spare*,*Ground*), but this conflicts with the causal link

$$Remove(Spare,Trunk) \xrightarrow{At(Spare,Ground)} PutOn(Spare,Axle)$$

- Now we add ordering constraint *LeaveOvernight before Remove*(*Spare*,*Trunk*).

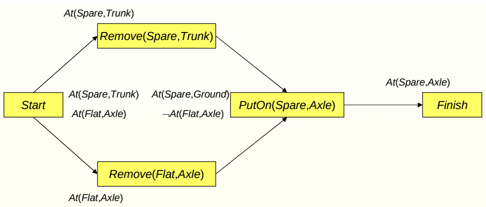
Changing a Flat Tire



Changing a Flat Tire

- Only one open precondition remains—*At*(*Spare*,*Trunk*), which is a precondition of *Remove*(*Spare*,*Trunk*).
- Only one action can achieve this—the *Start* action.
- Unfortunately, we have a conflict between the causal link between *Start* and *Remove* and the ¬*At*(*Spare*,*Trunk*) effect of *LeaveOvernight*.
- Note we cannot demote the *LeaveOvernight* action to precede *Start*, and we cannot promote it to follow *Remove*.
- This leads to a "backtrack" to remove *LeaveOvernight* from the plan altogether.
- Following in this manner yields the final plan.

Changing a Flat Tire



Planning Graphs

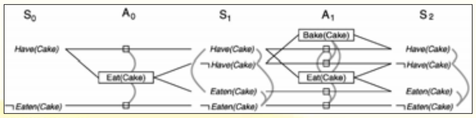
- As we see, planning is a search problem.
- We would like to be able to apply search heuristics to improve the process.
- One approach is to construct a "graph" of the search space and use this to guide search.
- This leads to the concept of a planning graph.

Planning Graphs

- Planning graphs are constructed as a sequence of "levels."
- Each level contains a set of literals and a set of actions.
  - The literals are those that "could" be true at that time step.
  - The actions are those that "could" have their preconditions satisfied at that time step.
- Planning graphs work only for propositional planning problems.

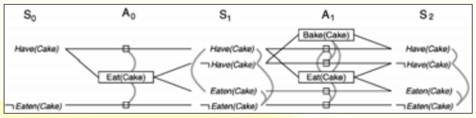
Constructing Planning Graphs

- Start with state level S_0, representing the problem's initial state.
- Insert action level A_0 and place all actions whose preconditions are satisfied in the previous level.
- Connect each action to its preconditions in S_0 and to its effects in S_1.
- This introduces new literals into S_1 not found in S_0.
- Continue until the graph is filled out.



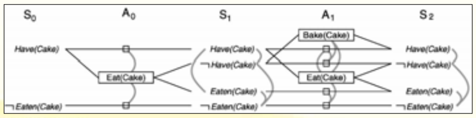
Persistent Actions

- We need to represent persistence (like frame axioms).
- **Persistent actions** are actions that have the same preconditions and effects.
- Thus, persistent action *A* has preconditions *C* and effects *C*.
- These are represented in planning graphs with small boxes.



Mutual Exclusion

- A particular level shows all actions that *could* occur in the corresponding state.
- The level also records *conflicts* between actions that could prevent them from occurring together.
- These conflicts are represented through **mutual exclusion** (**mutex**) links.
- These are shown as gray links in the planning graph.



Mutex Links

- Mutex links exist between two actions at a given level if any of the following holds:
  - *Inconsistent effects*: One action negates an effect of another action.
  - *Interference*: One of the effects of an action is the negation of a precondition of another action.
  - *Competing needs*: One of the preconditions of an action is mutually exclusive with the precondition of another action.
- Mutex links exist between two literals if one is the negation of the other or if each possible pair of actions that could achieve the literals is mutually exclusive (*inconsistent support*).

GRAPHPLAN

- Given a planning graph, it is possible to extract a plan directly from the graph.
- The process follows in two main steps that alternate in a loop:
  - Check whether all the goal literals are present in the current level with no mutex links between any pair of them.
  - If this holds, attempt to extract the plan; otherwise, expand the graph by adding the actions for the current level and the state literals for the next level.

GRAPHPLAN

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← GOALS[problem]
  loop do
    if goals all non-mutex in last level of graph then do
      solution ← EXTRACT-SOLUTION(graph, goals,
                                   LENGTH(graph))
      if solution ≠ failure then return solution
      else if NO-SOLUTION-POSSIBLE(graph)
          then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

Spare Tire Planning Graph