EECS 496: Sequential Decision Making

Soumya Ray

sray@case.edu

Office: Olin 516

Office hours : T 4-5:30 or by appointment

Announcements

- Exam
  - Will be third week of Nov
  - In class, 75 minutes
  - No project
  - Grade distribution: 35% written, 35% programming, 30% exam

Recap

- Forward state space search has problems with ____ actions. Actions are _____ if __________.
- In regression planning, the initial state is the ____.
- We must invert STRIPS operators to get the next state. We do this by ________.
- Search stops when _________.
- State space planners return ____ ____ plans. This means every _____ of actions has a ____ _____ _____.
- If some actions don't, this is called a _____ _____ ____.
- What are the dummy actions START and END in a POP?
- POPs are better than TOPs because (i)_____, (ii)______, (iii)______.
- States in the POP search contain ______, ______ and _____.
- The second element is ________.
- The third element is ________.
- What is a conflict? How do we resolve it?
- What is a consistent plan?
- To generate the next state, we pick an ___ ___ and find and ____ that has this as an ____.
- Then we add ____ constraints, ___ ___ and resolve _____.
- If we cannot resolve a conflict, the planner must _______.

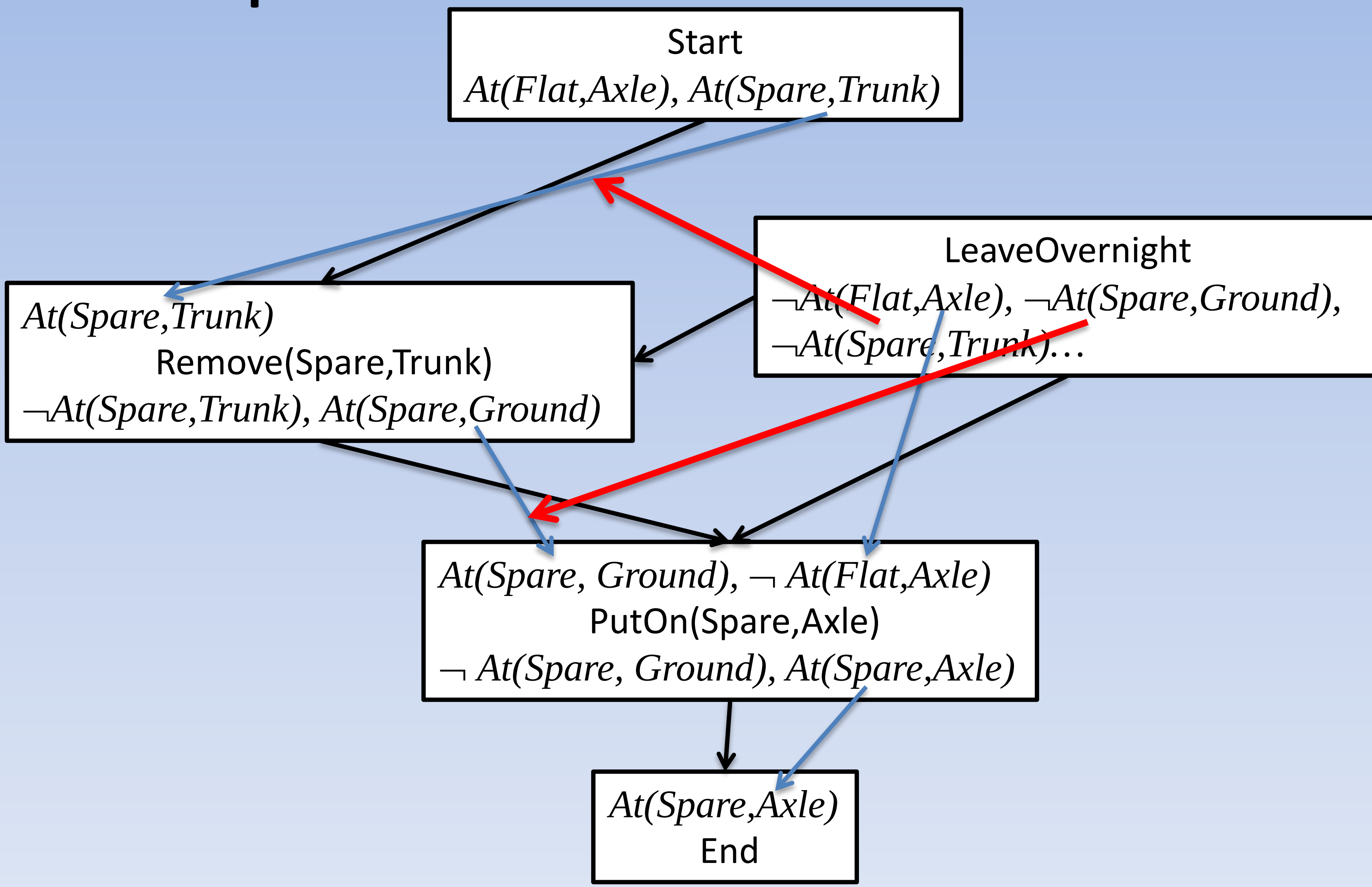
Today

- Automated Planning (Ch 10 R&N)

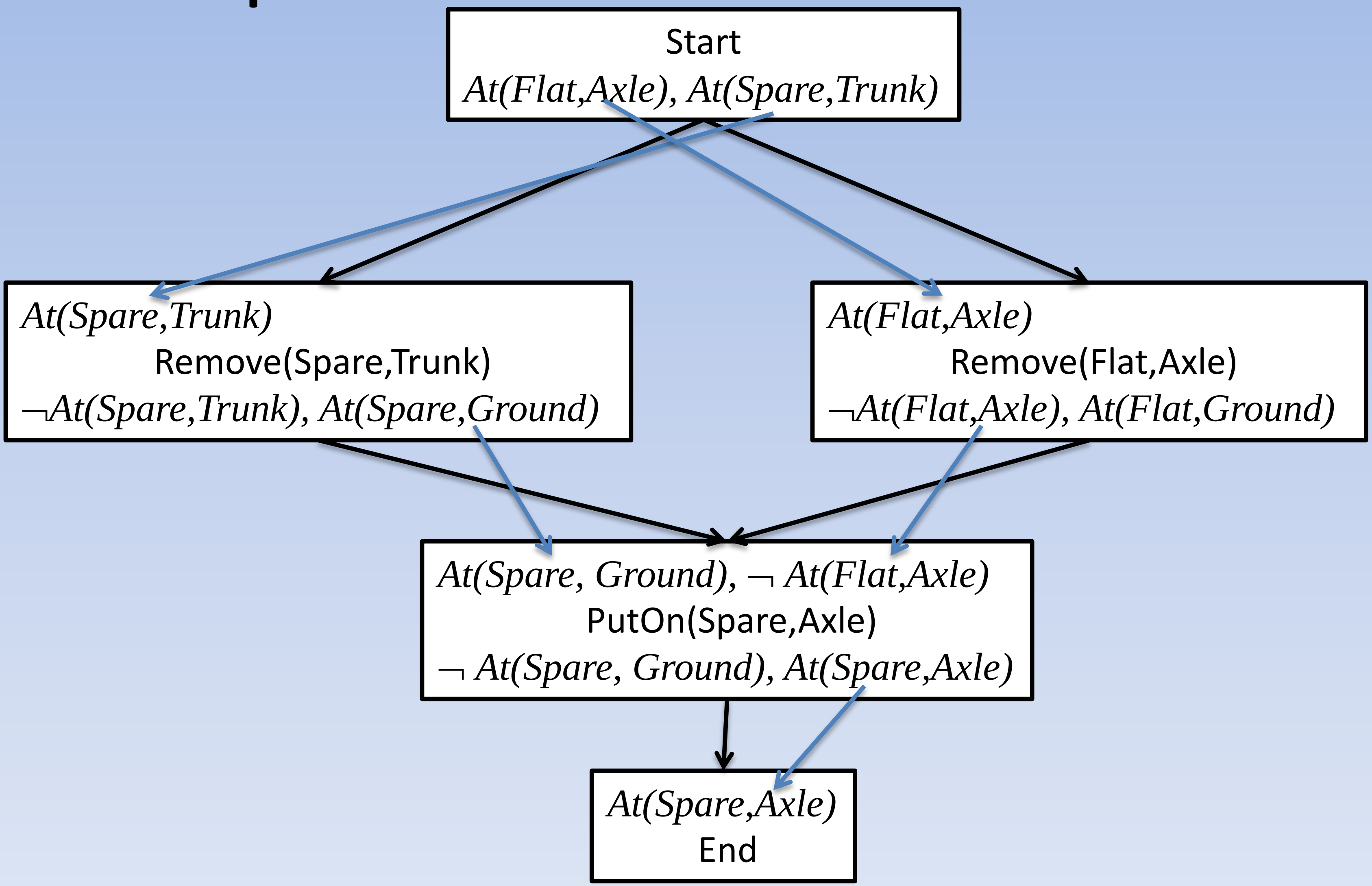
Example

- Init: $At(Flat,Axle), At(Spare,Trunk)$
- Goal: $At(Spare,Axle)$
- $Remove(Spare,Trunk)$
  - PRE: $At(Spare,Trunk)$
  - EFF: $\neg At(Spare,Trunk), At(Spare,Ground)$
- $Remove(Flat,Axle)$
  - PRE: $At(Flat,Axle)$
  - EFF: $\neg At(Flat,Axle), At(Flat,Ground)$
- $PutOn(Spare,Axle)$
  - PRE: $At(Spare, Ground), \neg At(Flat,Axle)$
  - EFF: $\neg At(Spare, Ground), At(Spare,Axle)$
- $LeaveOvernight$
  - PRE:
  - EFF: $\neg At(Spare,Trunk), \neg At(Spare, Ground), \neg At(Spare,Axle), \neg At(Flat,Axle), \neg At(Flat,Ground)$

Example


Start
At(Flat,Axle), At(Spare,Trunk)
LeaveOvernight
¬At(Flat,Axle), ¬At(Spare,Ground), ¬At(Spare,Trunk)…
At(Spare,Trunk)
Remove(Spare,Trunk)
¬At(Spare,Trunk), At(Spare,Ground)
At(Spare, Ground), ¬ At(Flat,Axle)
PutOn(Spare,Axle)
¬ At(Spare, Ground), At(Spare,Axle)
At(Spare,Axle)
End

Example


Start
At(Flat,Axle), At(Spare,Trunk)
At(Spare,Trunk)
Remove(Spare,Trunk)
¬At(Spare,Trunk), At(Spare,Ground)
At(Flat,Axle)
Remove(Flat,Axle)
¬At(Flat,Axle), At(Flat,Ground)
At(Spare, Ground), ¬ At(Flat,Axle)
PutOn(Spare,Axle)
¬ At(Spare, Ground), At(Spare,Axle)
At(Spare,Axle)
End

Properties of POP

- Very flexible, able to produce “least commitment” plans
- But not very efficient
  - Generating the next state in a POP requires many complex operations
    - Check for threats, resolve threats, check for cyclic temporal orderings
    - Wouldn’t it be nice if we could somehow *efficiently generate POPs in state-space*?

Graphplan (Blum and Furst 1994)

- A state-space planner that combines three key ideas:
  1. Layered Plans
  2. The Planning Graph
  3. Action Independence
- To create a fast general purpose planning algorithm
  – A basis for many state-of-the-art general purpose planners today

1. Layered Plans

- Key insight: May not need to produce arbitrary POPs
  - Can solve all planning problems with a smaller subset

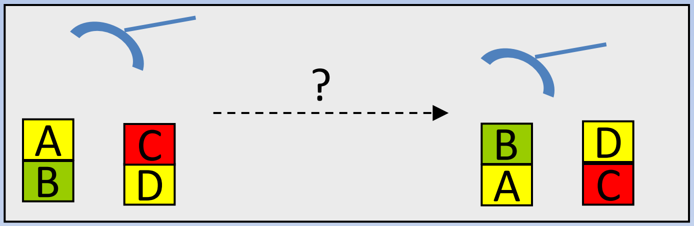
- Gives up some parallelism and flexibility but gains efficiency in search

1. Layered Plans

- A layered plan is a sequence of **sets** of actions
  - {a1,a2,a3},{a4,a5},…
  - Actions in one set can be done in parallel
  - Actions in different sets need to be done in sequence

1. Layered Plans

- Graphplan produces “layered plans”



<u>Layered Plan:</u> (a two layer plan)

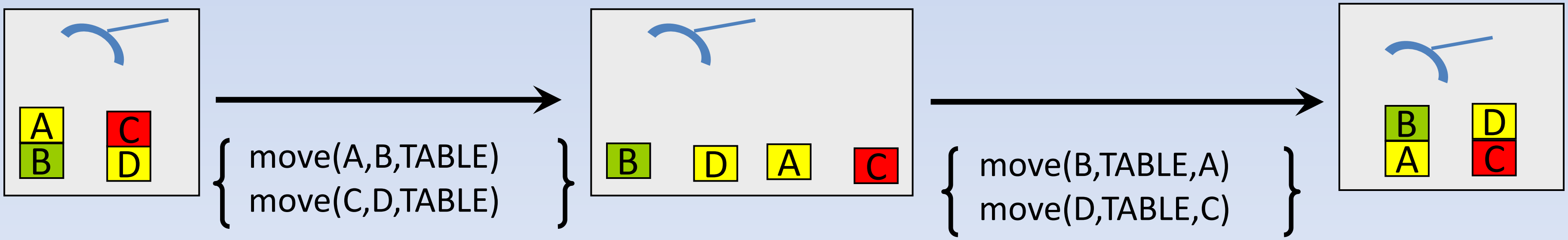
$$\left\{\begin{array}{l}\text{move(A,B,TABLE)}\\ \text{move(C,D,TABLE)}\end{array}\right\};\left\{\begin{array}{l}\text{move(B,TABLE,A)}\\ \text{move(D,TABLE,C)}\end{array}\right\}$$

1. Layered Plans

- For a valid layered plan, actions in the same set must be such that all sequential orderings of actions gives same result
  - Sufficient condition: actions are **independent** (later)

1. Executing a Layered Plan

- A set of actions is applicable in a state if all the actions are applicable
- Executing an applicable set of actions yields a new state that results from executing each individual action (order does not matter)



1. Layered Plans

- Observe:
  - Every totally ordered plan is a layered plan
  - Every layered plan can be linearized
  - Every layered plan is a POP
  - Is every POP layered?
    - NO
    - (Find an example!)
  - Does every solvable planning problem have a layered plan solution?

2. The Planning Graph

- Key insight: Create a data structure that summarizes the effects of multiple plans at once
  - The Planning graph
- Works by encoding a partial *reachability analysis* on the state space
- Can be built efficiently! (in time polynomial in the size of the planning problem)

2. Planning as Reachability

- Start at the initial state and perform all applicable actions; repeat
- The resulting structure is the state space graph (explored by state space planners)
- If s_1 is reachable from s_0 in this graph, a plan exists to go from s_0 to s_1
- So planning can be thought of as a reachability analysis problem on the state space graph

2. Planning as Reachability

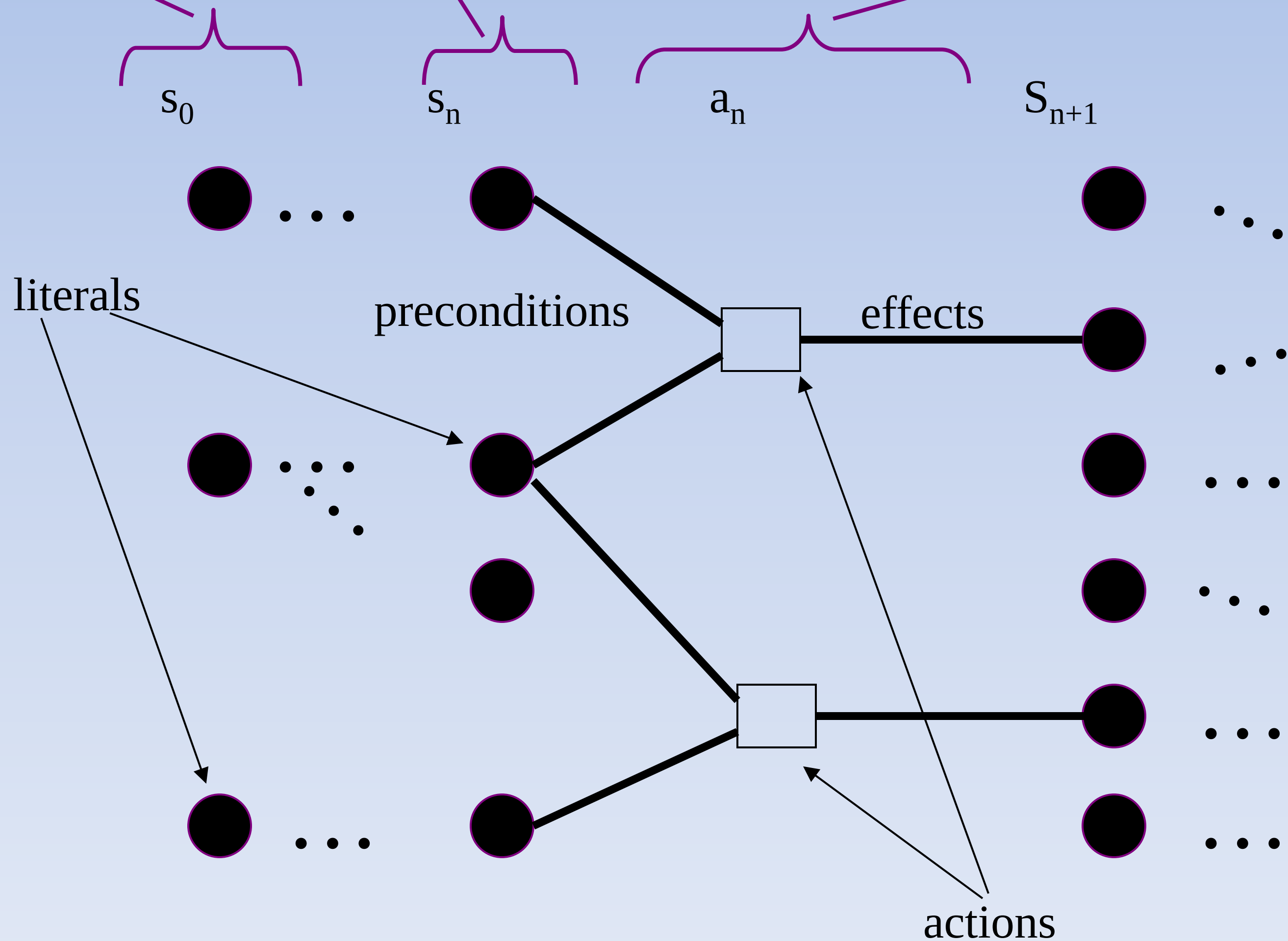
- Of course, one cannot actually construct the state-space graph efficiently
- But, we can construct a version of it where each state represents a *set* of ground states
  - More specifically, make each "state" correspond to everything that could *possibly* be true after action sequences of a fixed length
  - This is the "planning graph," the primary data structure in Graphplan

Planning Graph

state-level 0: literals true in s_0

state-level *n*: literals that may *possibly* be true after some *n* level plan

action-level *n*: actions that may *possibly* be applicable after some *n* level plan



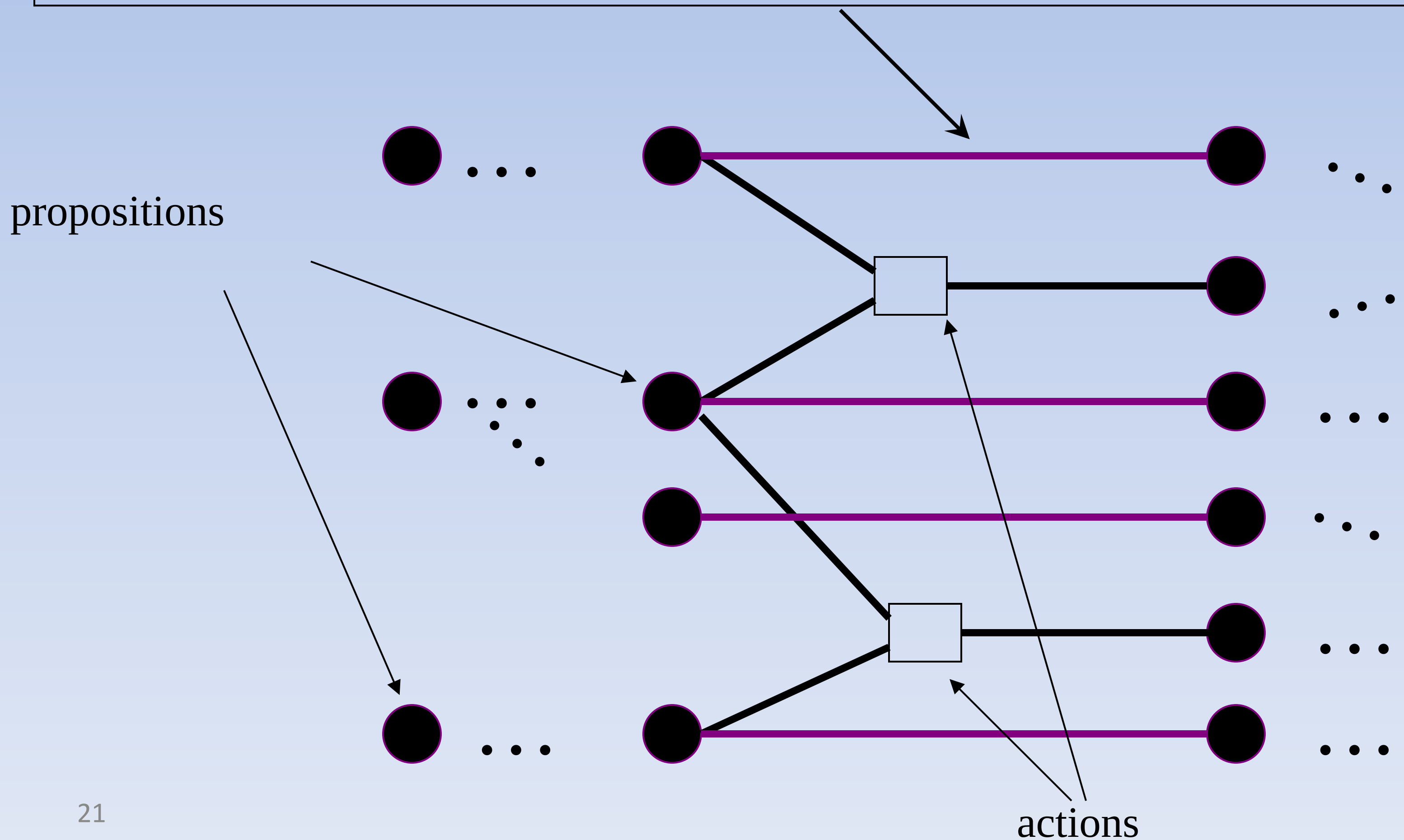
2. Planning Graph

- Idea: construct set of literals that *could be possibly achieved* after an *n*-level layered plan
  - Gives a compact (but approximate) representation of states that are reachable by *n*-level plans

- Sequence of levels correspond to time-steps in the plan
  - Each level contains a set of literals and a set of actions
  - Literals are those that could *possibly* be true at the time step
  - Actions are those for which the preconditions could *possibly* be satisfied at the time step

Planning Graph

- Maintenance action (persistence actions/no-ops)
  - represents what happens if no action affects the literal
  - include action with precondition *c* and effect *c*, for each literal *c*

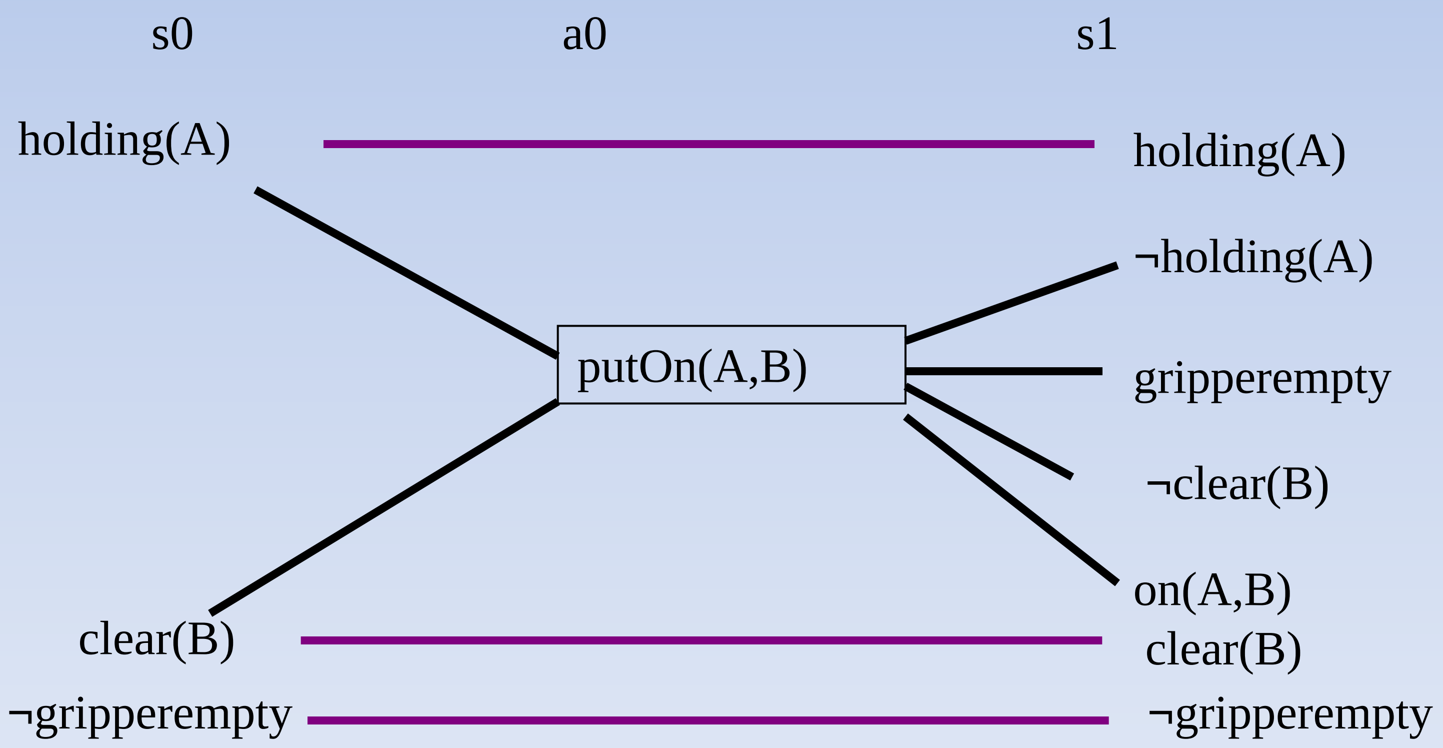


Example

putOn(A,B)

precondition: holding(A), clear(B)

effect: ¬holding(A), ¬clear(B), on(A,B), clear(A), gripperempty



Planning Graph Properties

- Construction of the planning graph can be done efficiently

- The planning graph can be used to establish *necessary conditions* for a solution to exist at any level