

Vorlesung

Grundlagen der

Künstlichen Intelligenz

Reinhard Lafrenz / Prof. A. Knoll

Robotics and Embedded Systems
Department of Informatics – I6
Technische Universität München

www6.in.tum.de

lafrenz@in.tum.de

089-289-18136

Room 03.07.055



Wintersemester 2012/13

17.12.2012



Chapter 10 (3rd ed.)

Classical Planning

Planning

- The Planning problem
- Planning with State-space search
- Partial-order planning
- Planning graphs
- Planning with propositional logic
- Analysis of planning approaches



What is Planning

- Generate sequences of actions to perform tasks and achieve objectives.
 - States, actions and goals
- Search for solution over abstract space of plans.
- Assists humans in practical applications
 - design and manufacturing
 - games
 - space exploration
 - Rescue operation (see also RoboCup rescue league)



Difficulty of real world problems

- Assume a problem-solving agent using some search method ...
 - Which actions are relevant?
 - Exhaustive search vs. backward search
 - What is a good heuristic functions?
 - Good estimate of the cost of the state?
 - Problem-dependent vs. -independent
 - How to decompose the problem?
 - Most real-world problems are *nearly* decomposable.



Planning language

- What is a good language?
 - Expressive enough to describe a wide variety of problems.
 - Restrictive enough to allow efficient algorithms to operate on it.
 - Planning algorithm should be able to take advantage of the logical structure of the problem.
- STRIPS, ADL, and PDDL



General language features

- Representation of states
 - Decompose the world in logical conditions and represent a state as a *conjunction of positive literals*.
 - Propositional literals: $Poor \wedge Unknown$
 - First Order-literals (grounded and function-free): $At(Plane1, Melbourne) \wedge At(Plane2, Sydney)$
 - Closed world assumption
- Representation of goals
 - Partially specified state and represented as a *conjunction of positive ground literals*
 - A goal is *satisfied* if the state contains all literals in goal.



General language features

- Representations of actions

- **Action = PRECOND + EFFECT**

- Action(Fly(p,from, to),*

- PRECOND: At(p,from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)*

- EFFECT: \neg At(p,from) \wedge At(p,to)*

- = action schema (*p, from, to* need to be instantiated)**

- Action name and parameter list
 - Precondition (conj. of function-free literals)
 - Effect (conj. of function-free literals)

- Add-list vs. delete-list in Effect



Language semantics?

How do actions affect states?

- An action is applicable in any state that satisfies the precondition.
- For FO action schema applicability involves a substitution θ for the variables in the PRECOND.

$At(P1, JFK) \wedge At(P2, SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge$
 $Airport(JFK) \wedge Airport(SFO)$

Satisfies : $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge$
 $Airport(to)$

With $\theta = \{p/P1, from/JFK, to/SFO\}$

Thus the action is applicable.



Language semantics?

- The result of executing action a in state s is the state s'
 - s' is same as s except
 - Any positive literal P in the effect of a is added to s'
 - Any negative literal $\neg P$ is removed from s'

$At(P1,SFO) \wedge At(P2,SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge$
 $Airport(JFK) \wedge Airport(SFO)$

- STRIPS assumption: (avoids representational frame problem)

every literal NOT in the effect remains unchanged



Expressiveness and extensions

- STRIPS is simplified
 - Important limit: function-free literals
 - Allows for propositional representation
 - Closed-world assumption
- Function symbols lead to infinitely many states and actions
- Open-world extension: Action Description language (ADL)
 - Action(Fly(p:Plane, from: Airport, to: Airport),*
 - PRECOND: At(p,from) \wedge (from \neq to)*
 - EFFECT: \neg At(p,from) \wedge At(p,to)*

Standardization : *Planning domain definition language (PDDL)*

- Developed for 1998/2000 International Planning Competition (IPC)



Example: air cargo transport

$Init(At(C1, SFO) \wedge At(C2, JFK) \wedge At(P1, SFO) \wedge At(P2, JFK) \wedge Cargo(C1) \wedge$
 $Cargo(C2) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO))$

$Goal(At(C1, JFK) \wedge At(C2, SFO))$

$Action(Load(c, p, a))$

PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$

EFFECT: $\neg At(c, a) \wedge In(c, p)$

$Action(Unload(c, p, a))$

PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$

EFFECT: $At(c, a) \wedge \neg In(c, p)$

$Action(Fly(p, from, to))$

PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$

EFFECT: $\neg At(p, from) \wedge At(p, to)$

$[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Load(C2, P2, JFK),$
 $Fly(P2, JFK, SFO)]$



Example: Spare tire problem

Init(*At*(*Flat*, *Axle*) \wedge *At*(*Spare*, *trunk*))

Goal(*At*(*Spare*, *Axle*))

Action(*Remove*(*Spare*, *Trunk*))

PRECOND: *At*(*Spare*, *Trunk*)

EFFECT: \neg *At*(*Spare*, *Trunk*) \wedge *At*(*Spare*, *Ground*)

Action(*Remove*(*Flat*, *Axle*))

PRECOND: *At*(*Flat*, *Axle*)

EFFECT: \neg *At*(*Flat*, *Axle*) \wedge *At*(*Flat*, *Ground*)

Action(*PutOn*(*Spare*, *Axle*))

PRECOND: *At*(*Spare*, *Ground*) \wedge \neg *At*(*Flat*, *Axle*)

EFFECT: *At*(*Spare*, *Axle*) \wedge \neg *At*(*Spare*, *Ground*)

Action(*LeaveOvernight*()), PRECOND: <none>

EFFECT: \neg *At*(*Spare*, *Ground*) \wedge \neg *At*(*Spare*, *Axle*) \wedge \neg *At*(*Spare*, *trunk*)
 \wedge \neg *At*(*Flat*, *Ground*) \wedge \neg *At*(*Flat*, *Axle*))

13 This example goes beyond STRIPS: negative literal in pre-condition



Example: Blocks world

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A) \wedge Block(A) \wedge Block(B) \wedge$
 $Block(C) \wedge Clear(B) \wedge Clear(C))$

$Goal(On(A, B) \wedge On(B, C))$

$Action(Move(b, x, y))$

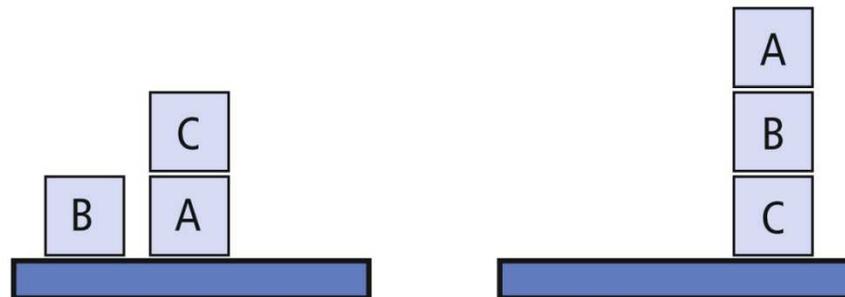
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge (b \neq x) \wedge (b \neq y)$
 $\wedge (x \neq y)$

EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$

$Action(MoveToTable(b, x))$

PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x)$

EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$

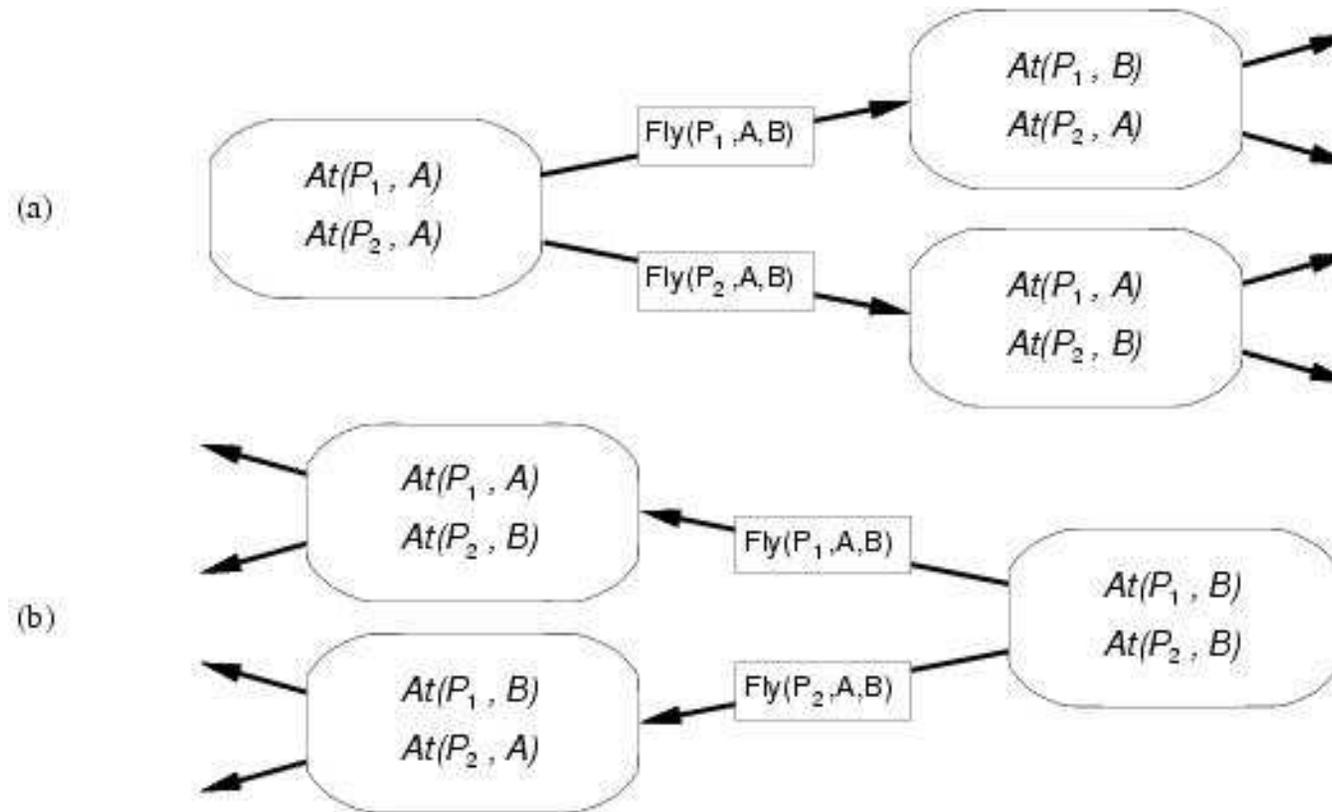


Planning with state-space search

- Both forward and backward search possible
- Progression planners
 - forward state-space search
 - Consider the effect of all possible actions in a given state
- Regression planners
 - backward state-space search
 - To achieve a goal, what must have been true in the previous state.



Progression and regression



Progression algorithm

- Formulation as state-space search problem:
 - Initial state = initial state of the planning problem
 - Literals not appearing are false
 - Actions = those whose preconditions are satisfied
 - Add positive effects, delete negative
 - Goal test = does the state satisfy the goal
 - Step cost = each action costs 1
- No functions ... any graph search that is complete is a complete planning algorithm.
- Inefficient: (1) irrelevant action problem (2) good heuristic required for efficient search



Regression algorithm

- How to determine predecessors?
 - What are the states from which applying a given action leads to the goal?
 - Goal state = $At(C1, B) \wedge At(C2, B) \wedge \dots \wedge At(C20, B)$
 - Relevant action for first conjunct: $Unload(C1, p, B)$
 - Works only if pre-conditions are satisfied.
 - Previous state = $In(C1, p) \wedge At(p, B) \wedge At(C2, B) \wedge \dots \wedge At(C20, B)$
 - Subgoal $At(C1, B)$ should not be present in this state.
- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
 - Often much lower branching factor than forward search.



Regression algorithm

- General process for predecessor construction
 - Give a goal description G
 - Let A be an action that is relevant and consistent
 - The predecessors is as follows:
 - Any positive effects of A that appear in G are deleted.
 - Each precondition literal of A is added , unless it already appears.
- Any standard search algorithm can be added to perform the search.
- Termination when predecessor satisfied by initial state.
 - In FO case, satisfaction might require a substitution.



Heuristics for state-space search

- Neither progression or regression are very efficient without a good heuristic.
 - How many actions are needed to achieve the goal?
 - Exact solution is NP hard, find a good estimate
- Approaches to find admissible heuristics: Find optimal solution to relaxed problems
 - Heuristic: Remove all preconditions from actions
 - Heuristic: Ignore Delete-List
 - Use the subgoal independence assumption:
 - The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving the subproblems independently.



Partial-order planning

- Progression and regression planning are *totally ordered plan search* forms.
 - They cannot take advantage of problem decomposition.
 - Decisions must be made on how to sequence actions on all the subproblems
- Least commitment strategy:
 - Delay choice during search



Shoe example

Goal(RightShoeOn \wedge LeftShoeOn)

Init()

Action(RightShoe, PRECOND: RightSockOn
 EFFECT: RightShoeOn)

Action(RightSock, PRECOND: <none>
 EFFECT: RightSockOn)

Action(LeftShoe, PRECOND: LeftSockOn
 EFFECT: LeftShoeOn)

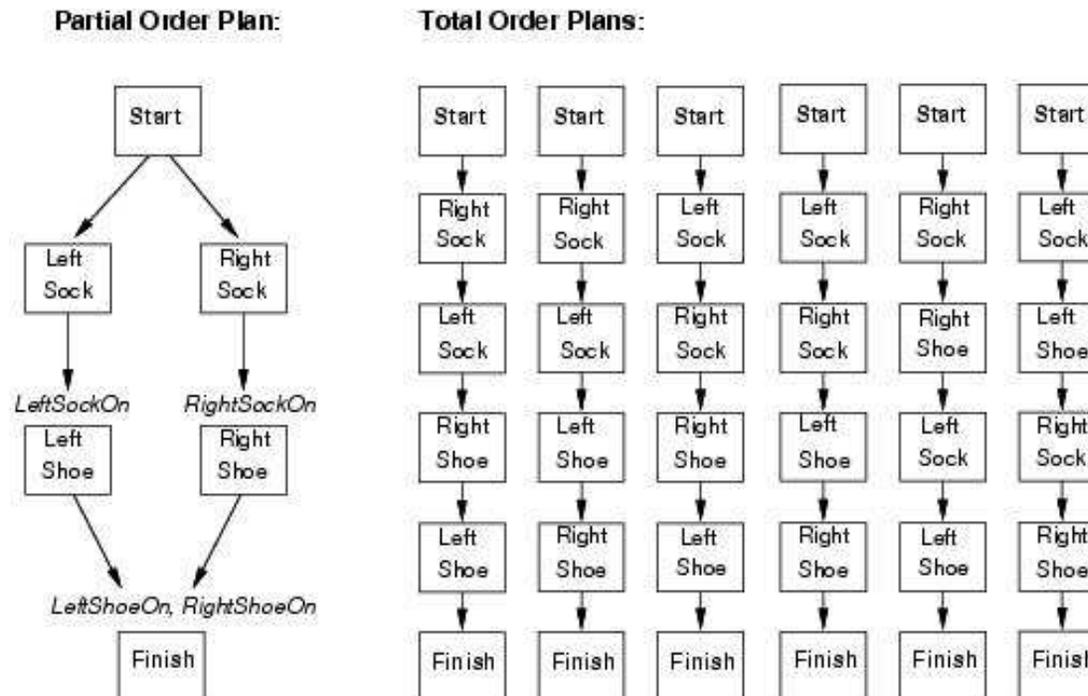
Action(LeftSock, PRECOND: <none>
 EFFECT: LeftSockOn)

Planner: combine two action sequences (1)leftsock, leftshoe
(2)rightsock, rightshoe



Partial-order planning

- Any planning algorithm that can place two actions into a plan without which comes first is a Partially Ordered Plan.



Partial-order planning as a search problem

- States are (mostly unfinished) plans.
 - The empty plan contains only start and finish actions.

- Each plan has 4 components:
 1. A set of actions (steps of the plan)
 2. A set of ordering constraints: $A < B$
 - Cycles represent contradictions
 3. A set of causal links
 - The plan may not be extended by adding a new action C that conflicts with the causal link. (if the effect of C is $\neg p$ and if C could come after A and before B)
 4. A set of open preconditions.
 - If precondition is not achieved by action in the plan.



Partial-order planning as a search problem

- A plan is *consistent* iff there are no cycles in the ordering constraints and no conflicts with the causal links.
- A consistent plan with no open preconditions is a *solution*.
- A partial order plan is executed by repeatedly choosing *any* of the possible next actions.
 - This flexibility is a benefit in non-cooperative environments.



Solving Partial-order planning

Assume propositional planning problems:

- The initial plan contains *Start* and *Finish*, the ordering constraint $Start < Finish$, no causal links, all the preconditions in *Finish* are open.
- Successor function :
 - picks one open precondition p on an action B and
 - generates a successor plan for every possible consistent way of choosing action A that achieves p .
- Test goal



Enforcing consistency

When generating successor plan:

- The causal link $A \xrightarrow{p} B$ and the ordering constraint $A < B$ is added to the plan.
 - If A is new also add $start < A$ and $A < B$ to the plan
- Resolve conflicts between new causal link and all existing actions
- Resolve conflicts between action A (if new) and all existing causal links.



Process summary

- Operators on partial plans
 - Add link from existing plan to open precondition.
 - Add a step to fulfill an open condition.
 - Order one step w.r.t 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.



Example: Spare tire problem

Init(*At*(*Flat*, *Axle*) \wedge *At*(*Spare*, *trunk*))

Goal(*At*(*Spare*, *Axle*))

Action(*Remove*(*Spare*, *Trunk*))

PRECOND: *At*(*Spare*, *Trunk*)

EFFECT: \neg *At*(*Spare*, *Trunk*) \wedge *At*(*Spare*, *Ground*)

Action(*Remove*(*Flat*, *Axle*))

PRECOND: *At*(*Flat*, *Axle*)

EFFECT: \neg *At*(*Flat*, *Axle*) \wedge *At*(*Flat*, *Ground*)

Action(*PutOn*(*Spare*, *Axle*))

PRECOND: *At*(*Spare*, *Ground*) \wedge \neg *At*(*Flat*, *Axle*)

EFFECT: *At*(*Spare*, *Axle*) \wedge \neg *Ar*(*Spare*, *Ground*)

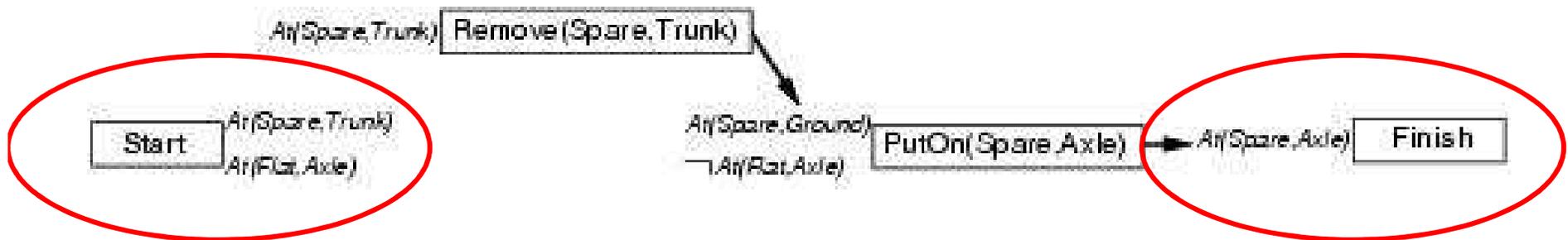
Action(*LeaveOvernight*

PRECOND:

EFFECT: \neg *At*(*Spare*, *Ground*) \wedge \neg *At*(*Spare*, *Axle*) \wedge \neg *At*(*Spare*, *trunk*)
 \wedge \neg *At*(*Flat*, *Ground*) \wedge \neg *At*(*Flat*, *Axle*))



Solving the problem



- Initial plan: Start with EFFECTS and Finish with PRECOND.



Solving the problem



- Initial plan: Start with EFFECTS and Finish with PRECOND.
- Pick an open precondition: $At(Spare, Axle)$
- Only $PutOn(Spare, Axle)$ is applicable
- Add causal link: $PutOn(Spare, Axle) \xrightarrow{At(Spare, Axle)} Finish$
- Add constraint : $PutOn(Spare, Axle) < Finish$



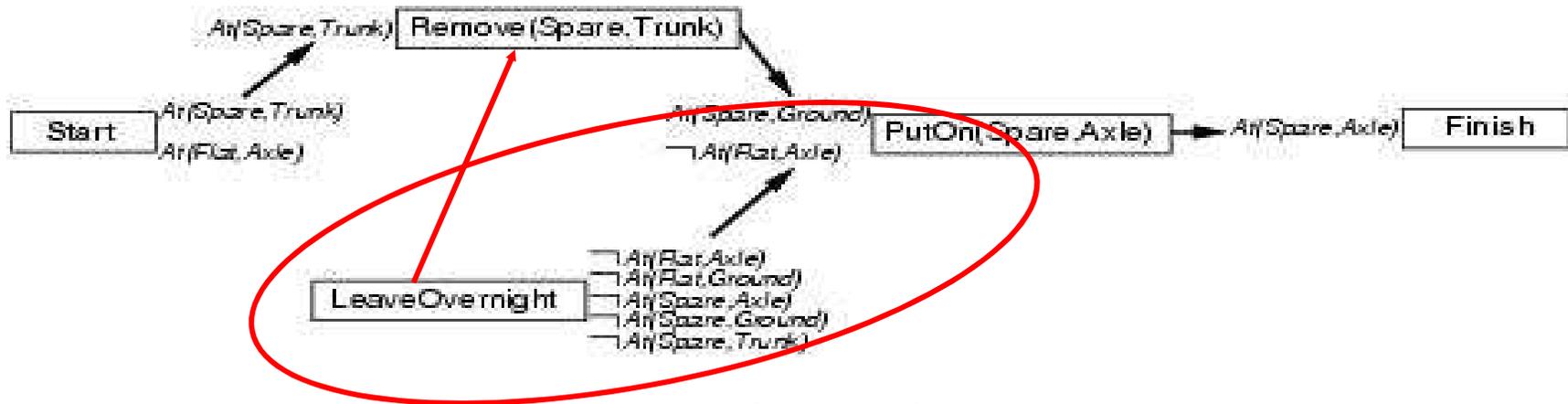
Solving the problem



- Pick an open precondition: $At(Spare, Ground)$
- Only $Remove(Spare, Trunk)$ is applicable
- Add causal link: $Remove(Spare, Trunk) \xrightarrow{At(Spare, Ground)} PutOn(Spare, Axle)$
- Add constraint : $Remove(Spare, Trunk) < PutOn(Spare, Axle)$



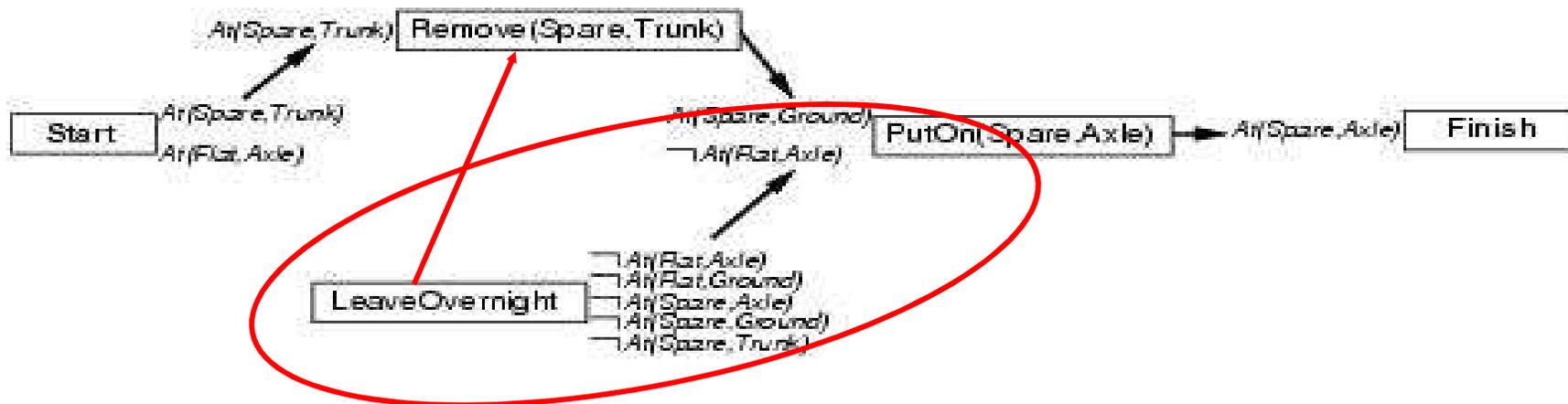
Solving the problem



- Pick an open precondition: $At(Spare, Ground)$
- *LeaveOverNight* is applicable
- conflict: $Remove(Spare, Trunk) \xrightarrow{At(Spare, Ground)} PutOn(Spare, Axle)$
- To resolve, add constraint: $LeaveOverNight < Remove(Spare, Trunk)$



Solving the problem

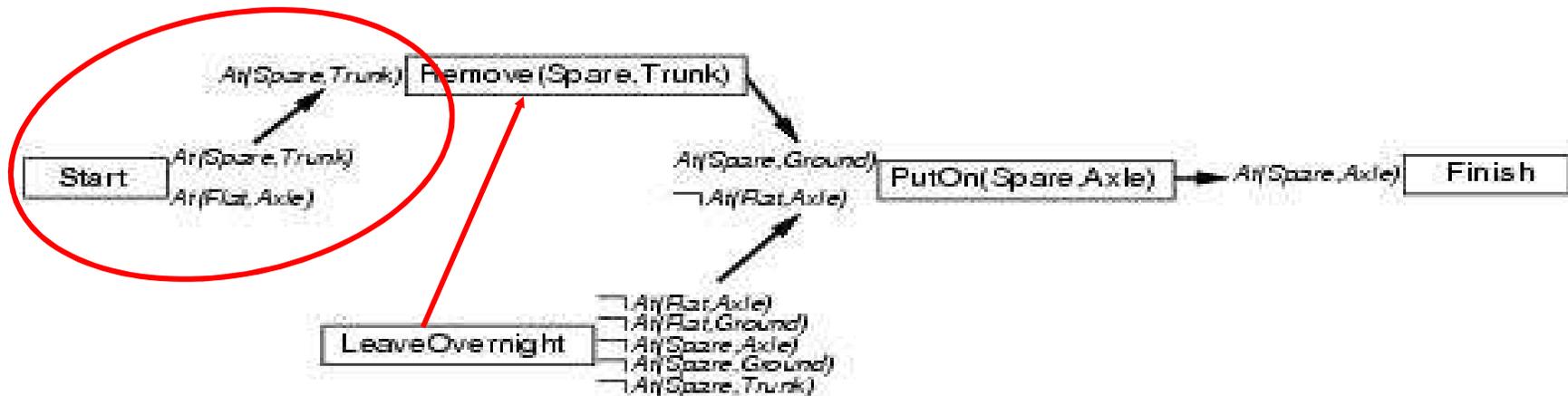


- Pick an open precondition: $At(Spare, Ground)$
- *LeaveOverNight* is applicable
- conflict: $Remove(Spare, Trunk) \xrightarrow{At(Spare, Ground)} PutOn(Spare, Axle)$
- To resolve, add constraint: $LeaveOverNight < Remove(Spare, Trunk)$
- Add causal link:

$$LeaveOverNight \xrightarrow{\neg At(Spare, Ground)} PutOn(Spare, Axle)$$



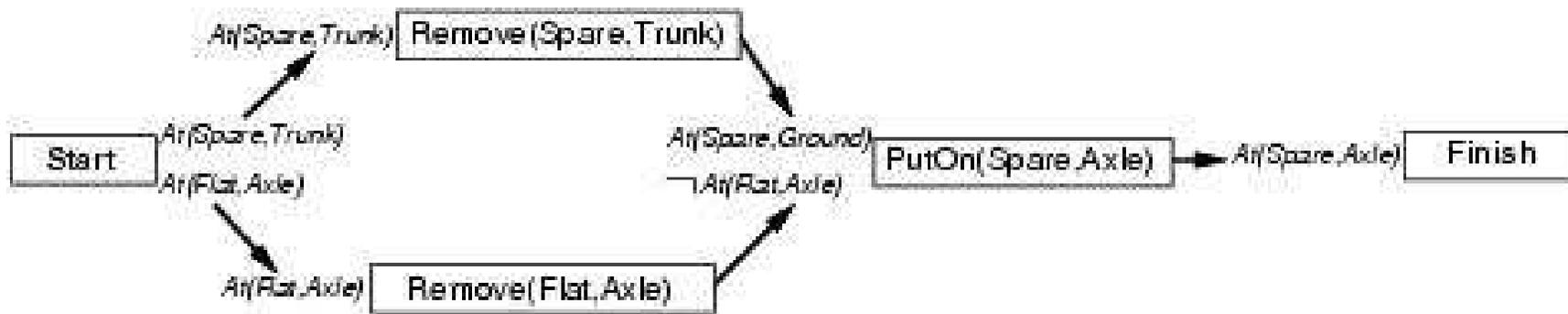
Solving the problem



- Pick an open precondition: $At(Spare, Trunk)$
- Only *Start* is applicable
- Add causal link: $Start \xrightarrow{At(Spare, Trunk)} Remove(Spare, Trunk)$
- Conflict: of causal link with effect $At(Spare, Trunk)$ in *LeaveOverNight*
 - *No re-ordering solution possible.*
- backtrack



Solving the problem



- Remove *LeaveOverNight*, *Remove(Spare, Trunk)* and causal links
- Repeat step with *Remove(Spare, Trunk)*
- Add also *Remove(Flat, Axle)* and finish



Some details ...

- What happens when a first-order representation that includes variables is used?
 - Complicates the process of detecting and resolving conflicts.
 - Can be resolved by introducing inequality constraints.
- CSP's most-constrained-variable constraint can be used for planning algorithms to select a PRECOND.



Planning graphs

- Used to achieve better heuristic estimates.
 - A solution can also directly extracted using GRAPHPLAN.
- Consists of a sequence of levels that correspond to time steps in the plan.
 - Level 0 is the initial state.
 - Each level consists of a set of literals and a set of actions.
 - *Literals* = all those that *could* be true at that time step, depending upon the actions executed at the preceding time step.
 - *Actions* = all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold.

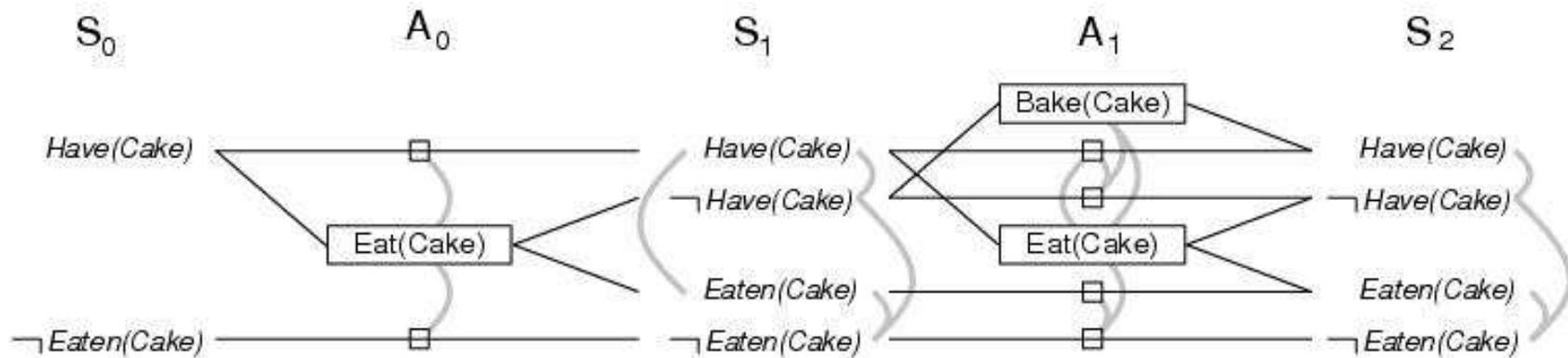


Planning graphs

- “Could”?
 - Records only a restricted subset of possible negative interactions among actions.
- They work only for propositional problems.
- Example:
 - Init(Have(Cake))
 - Goal(Have(Cake) \wedge Eaten(Cake))
 - Action(Eat(Cake), PRECOND: Have(Cake)
EFFECT: \neg Have(Cake) \wedge Eaten(Cake))
 - Action(Bake(Cake), PRECOND: \neg Have(Cake)
EFFECT: Have(Cake))



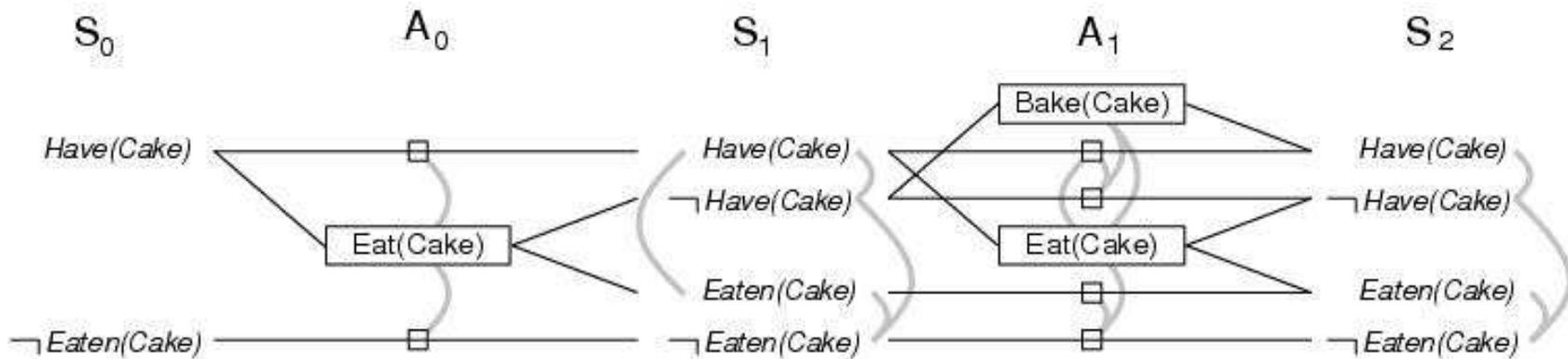
Cake example



- Start at level S_0 and determine action level A_0 and next level S_1 .
 - $A_0 \gg$ all actions whose preconditions are satisfied in the previous level.
 - Connect precond and effect of actions $S_0 \rightarrow S_1$
 - Inaction is represented by persistence actions.
- Level A_0 contains the actions that could occur
 - Conflicts between actions are represented by *mutex* links



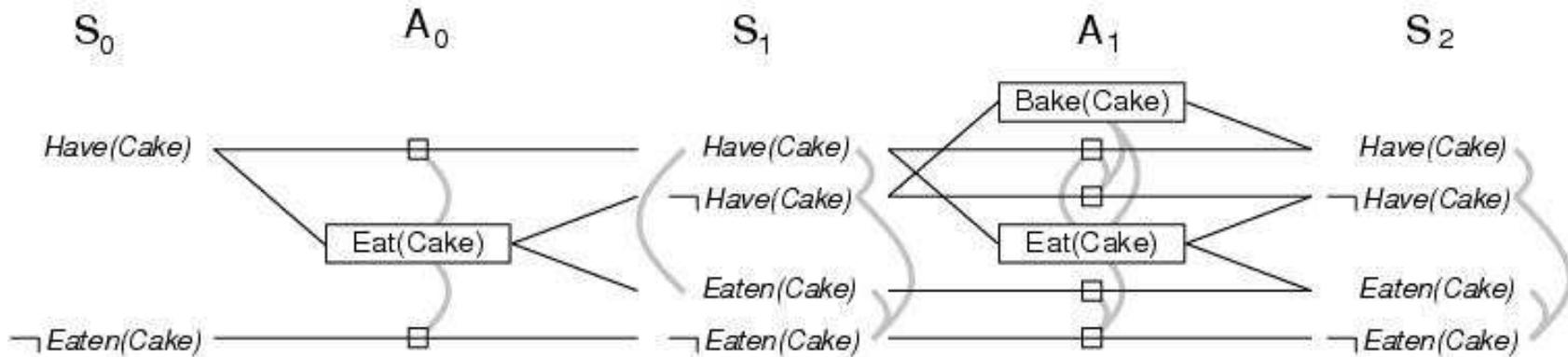
Cake example



- Level S_1 contains all literals that could result from picking any subset of actions in A_0
 - Conflicts between literals that can not occur together are represented by mutex links.
 - S_1 defines multiple states and the mutex links are the constraints that define this set of states.
- Continue until two consecutive levels are identical: *leveled off*
 - Or contain the same amount of literals (explanation follows later)



Cake example



- A mutex relation holds between **two actions** when:
 - *Inconsistent effects*: one action negates the effect of another.
 - *Interference*: one of the effects of one action is the negation of a precondition of the other.
 - *Competing needs*: one of the preconditions of one action is mutually exclusive with the precondition of the other.
- A mutex relation holds between **two literals** when (*inconsistent support*):
 - If one is the negation of the other OR
 - if each possible action pair that could achieve the literals is mutex.



PG and heuristic estimation

PGs provide information about the problem

- A literal that does not appear in the final level of the graph cannot be achieved by any plan.
 - Useful for backward search (cost = inf).
- Level of appearance can be used as cost estimate of achieving any goal literals = *level cost*.
- Small problem: several actions can occur
 - Restrict to one action using serial PG (add mutex links between every pair of actions, except persistence actions).
- Max-level, sum-level and set-level heuristics.

PG is a relaxed problem.



The GRAPHPLAN Algorithm

- How to extract a solution directly from the PG

function GRAPHPLAN(*problem*) **return** *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*)

