Artificial Intelligence Foundations and Applications

Partha Pratim Chakrabarti
Plaban Bhowmik
Sudeshna Sarkar
Centre of Excellence in Artificial Intelligence
IIT Kharagpur

Planning

Deterministic Planning

Planning

- **Planning** is one of the classic AI problems
- E.g. how can a robot figure out a sequence of actions to solve some problem?
- It is part of the logic-oriented tradition of AI

Autonomous vehicle navigation
Process control
Assembly line
Military operations
Travel planning
design and manufacturing environments
military operations, games, space exploration

- Military operations
- Autonomous space operations
- Construction tasks
- Machining tasks
- Mechanical assembly
- Design of experiments in genetics
- Command sequences for satellite

What is Planning?

The task of finding a course of action to achieve goals

- Given
  - a logical description of the **world states**
  - a logical description of a set of **possible actions**
  - a logical description of the **initial situation**, and
  - a logical description of the **goal conditions**
- Find
  - a **sequence of actions** (a **plan of actions**) that brings us from the initial situation to a situation in which the goal conditions hold.

Classical planning:
Environment
–Fully observable
–Deterministic
–Static
–Discrete

Example: A Robot that Makes Tea

1. put water in the kettle
2. heat the kettle
3. get a cup
4. pour hot water into the cup (after the water is hot enough)
5. get a tea bag
6. leave the tea bag in the water for enough time
7. remove the tea bag
8. add milk
9. add sugar
10. stir until mixed

Example: A Robot that Makes Tea

1. put water in the kettle
2. heat the kettle
3. get a cup
4. pour hot water into the cup (after the water is hot enough)
5. get a tea bag
6. leave the tea bag in the water for enough time
7. remove the tea bag
8. add milk
9. add sugar
10. stir until mixed

- **Sub-actions**, are often needed
- e.g. “get a cup” might consist of the actions
  - A. Move to the cupboard
  - B. Open the cupboard door
  - C. Grasp a cup
  - D. Take it out of the cupboard
  - E. Close the cupboard
- Any of these actions could further broken down into smaller actions
  - What are the exact muscle and finger movements needed to grasp a cup?

Example: A Robot that Makes Tea

1. put water in the kettle
2. heat the kettle
3. get a cup
4. pour hot water into the cup (after the water is hot enough)
5. get a tea bag
6. leave the tea bag in the water for enough time
7. remove the tea bag
8. add milk
9. add sugar
10. stir until mixed

- **Exceptional** situations might trigger entire **sub-plans**
  - E.g. if there’s no milk, the “get milk” action might trigger one of the following sub-plans
    - Go to the store to buy milk
    - Borrow some milk from a neighbor
    - Find a substitute for milk
    - Make something other than tea
    - Etc.
- Actions might also **fail**, e.g. the cup could slip and fall to the floor, or opening the sugar could spill the sugar on the floor
  - Must fix, or re-do, failed actions

Example: A Robot that Makes Tea

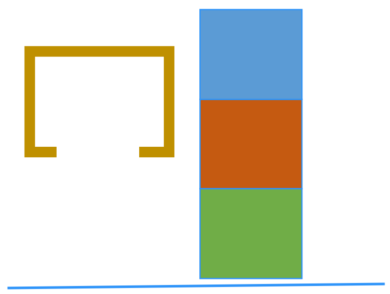
- **Timing** and **sensing** are also important
  - The kettle can’t be heated for too short a time or too long a time.
  - Does the order in which milk and sugar are added matter? Could they be added at the same time?
  - The robot might need to taste the tea to decide if more milk/sugar is needed.
  - How does the robot know when the mixing is done?

Example: Dialog Planning

- Supermarket conversation:
  - Customer: Can you tell me where I can find the bread?
  - Employee: It’s in aisle 2.
  - Customer: Thanks!
- A dialog like this can be modelled as a planning problem
  - The customer’s goal is to get bread
  - He could do that in various ways, e.g. walking around the store searching for bread, finding a map that says where bread is, asking someone for help, etc.
  - In this dialog, the customer has chosen to try the “ask someone for help” action

Represent this Blocks World

- A robot arm (yellow) can pick up and put down blocks to form stacks.
- It cannot pick up a block that has another block on top of it.
- It cannot pick up more than one block at a time.
- Any number of blocks can sit on the table.

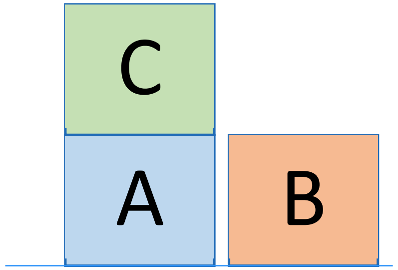


- How would you represent this block world to use logic to find a plan?
- You need to represent the states, actions, goals, transitions.

Classical Planning:

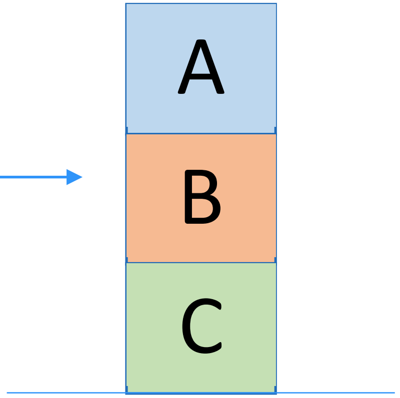
- + concise object representation and clearer action definitions
- – only works for deterministic fully observable worlds

Blocks World



Starting state

Task: Find a (short) sequence of block moves that transforms the starting state into the goal state.

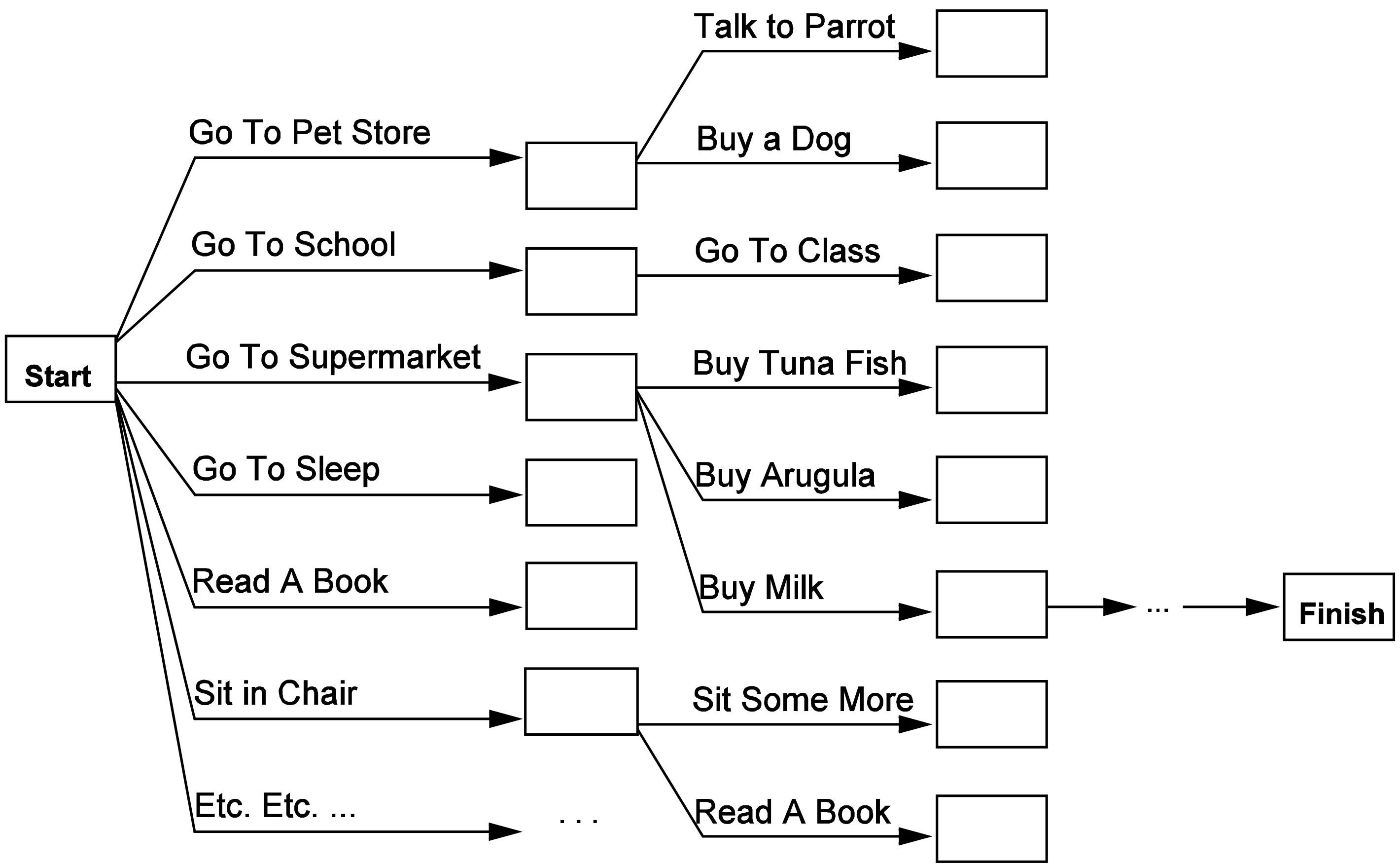


Goal state

Planning vs. General Search

- Basic difference: Explicit, logic-based representation
- **States/Situations**: descriptions of the world by logical formulae
  - $\rightarrow$ Agent can explicitly reason about and communicate with the world.
- **Operators/Actions**: Axioms or transformation on formulae in a logical form
  - $\rightarrow$ Agent can gain information about the effects of actions by inspecting the operators.
- **Goal conditions** as logical formulae vs. goal test (black box)
  - $\rightarrow$ Agent can reflect on its goals.

Challenges in Planning – too many choices!



Languages for Planning Problems

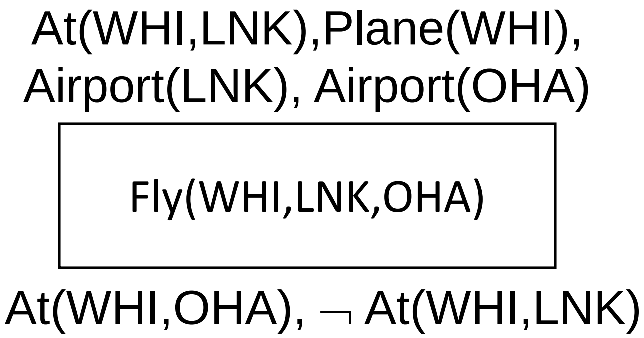
- STRIPS
  - Stanford Research Institute Problem Solver
  - Historically important
- ADL
  - Action Description Languages
- PDDL
  - Planning Domain Definition Language
  - Revised & enhanced for the needs of the International Planning Competition

State of the world (STRIPS language)

- State of the world = conjunction of positive, ground, function-free literals
- At(Home) AND IsAt(Umbrella, Home) AND CanBeCarried(Umbrella) AND IsUmbrella(Umbrella) AND HandEmpty AND Dry
- Not OK as part of the state:
  - NOT(At(Home)) (negative)
  - At(x) (not ground)
  - At(Bedroom(Home)) (uses the function Bedroom)
- Any literal not mentioned is assumed false
  - Other languages make different assumptions, e.g., negative literals part of state, unmentioned literals unknown

Action Representation

- Action Schema
  - Action name
  - Preconditions
  - Effects



- Example

  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))

- Sometimes, Effects are split into ADD list and DELETE list

Action TakeObject

TakeObject(location, x)

Preconditions:

- HandEmpty
- CanBeCarried(x)
- At(location)
- IsAt(x, location)

Effects ("NOT something" means that that something should be removed from state):

- Holding(x)
- NOT(HandEmpty)
- NOT(IsAt(x, location))

WalkWithUmbrella (location1, location2, umbr)

- Preconditions:
  - At(location1)
  - Holding(umbr)
  - IsUmbrella(umbr)
- Effects:
  - At(location2)
  - NOT(At(location1))

WalkWithoutUmbrella (location1, location2)

- Preconditions:
  - At(location1)
- Effects:
  - At(location2)
  - NOT(At(location1))
  - NOT(Dry)

A goal and a plan

Goal: At(Work) AND Dry

Initial state:

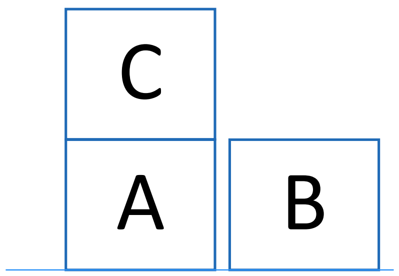
- At(Home) AND IsAt(Umbrella, Home) AND CanBeCarried(Umbrella) AND IsUmbrella(Umbrella) AND HandEmpty AND Dry

- TakeObject(Home, Umbrella)
  - At(Home) AND CanBeCarried(Umbrella) AND IsUmbrella(Umbrella) AND Dry AND Holding(Umbrella)
- WalkWithUmbrella(Home, Work, Umbrella)
  - At(Work) AND CanBeCarried(Umbrella) AND IsUmbrella(Umbrella) AND Dry AND Holding(Umbrella

Classical Planning Model

- Classical planning model is a tuple S = <S, s_0 , S_G , A, f, c>
  - Finite and discrete state space S
  - A known initial state $s_0 \in$ S
  - A set $S_G \subseteq$ S of goal states
  - Actions A(s) ⊆ A applicable in each s ∈ S
  - A deterministic transition function
    $s' = f(a, s)$
  - Non-negative action costs c(a, s)

Input Representation



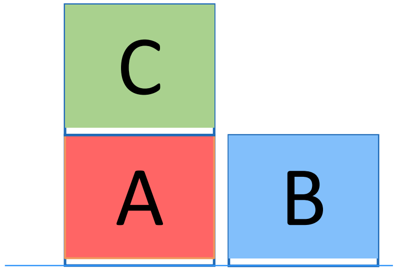
- Description of initial state of world
  - Conjunction of propositions:

block (a), block (b), block (c),
on-table (a), on-table(b), clear (a), clear (b), clear (c), arm-empty()

Generalize with variables

- **block(x)** means object x is a block
  - the table is an object, but not a block
- **on(x, y)** means object x is on top of object y
  - on(x,x) is not allowed, i.e. an object can’t be on top of itself
  - on(x,y) and on(y,x) cannot *both* be true at the same time
- **clear(x)** means there is nothing on top of object x
  - without this, we’d have to use quantified statements like “for all blocks y, on(y,x) is false

Blocks World Representation

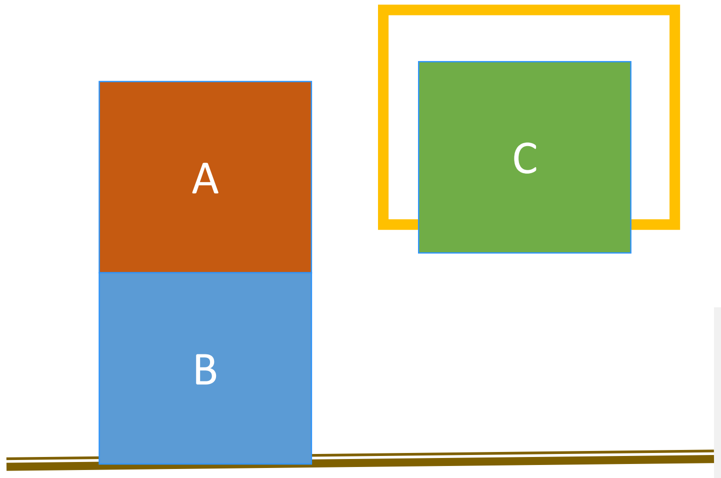


Representation of this State

- block(A), block(B), block(C)
- on(C, A)
- on-table (A), on-table(B)
- clear(C), clear(B)

All terms are and-ed together

How to represent this state?



- block(A), block(B), block(C)
- on-block (A, B)
- on-table(B)
- clear(A)
- In-hand(C)

Goal Description

Description of goal: i.e. set of worlds

- E.g., Logical conjunction
- Any world satisfying conjunction is a goal

and (on-block (a, b) , on-block (b ,c))

Actions / Operators

Operators change the state by adding/deleting predicates

Preconditions:

Actions can be applied only if all precondition predicates are true in the current state

Effects:

New state is a copy of the current predicates with the addition or deletion of specified predicates

Pickup Block C from Table (State Transition)

Instances:

Blocks A, B, C

Possible Predicates:

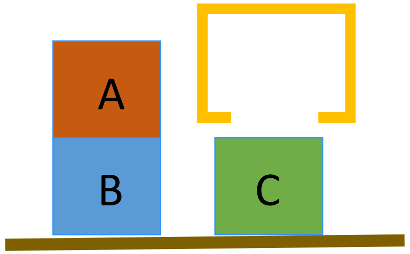
HandEmpty()

On-Table(block)

On-Block(b1,b2)

Clear(block)

In-Hand(block)



<u>State:</u>

HandEmpty()

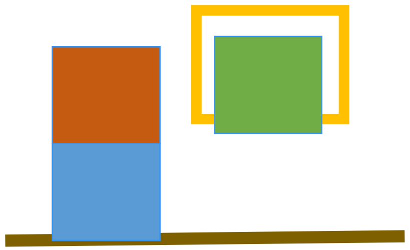
On-Table(B)

On-Table(C)

On-Block(A,B)

Clear(A)

Clear(C)



<u>State:</u>

In-Hand(C)

On-Table(B)

On-Block(A,B)

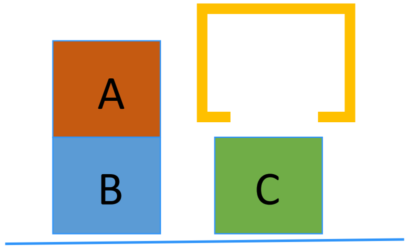
Clear(A)

Clear(C)

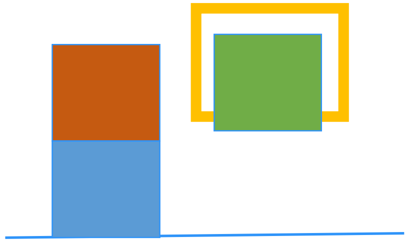
Pickup Block C from Table (Preconditions, Effects)

Instances:
Blocks A, B, C

Possible Predicates:
HandEmpty()
On-Table(block)
On-Block(b1,b2)
Clear(block)
In-Hand(block)

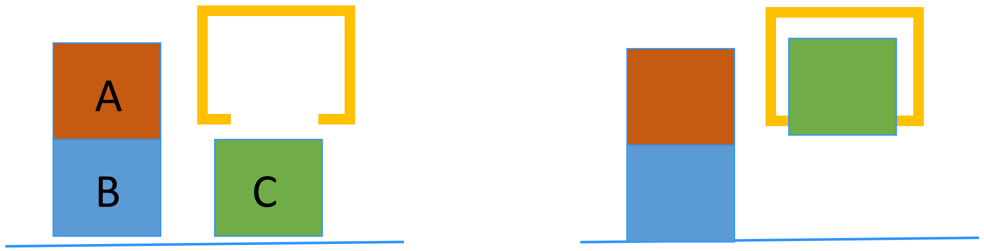


State:
HandEmpty()
On-Table(B)
On-Table(C)
On-Block(A,B)
Clear(A)
Clear(C)



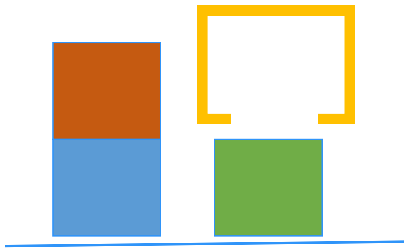
State:
In-Hand(C)
On-Table(B)
On-Block(A,B)
Clear(A)
Clear(C)
Delete HandEmpty()
Delete On-Table(C)

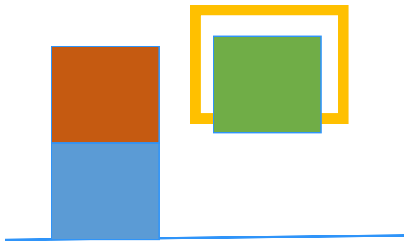
Operator: Pickup-Block-C from Table



Preconditions	Effects	
HandEmpty()	Add	In-Hand(C)
Clear(C)	Delete	HandEmpty()
On-Table(C)		On-Table(C)

Operator: Pickup-Block from Table

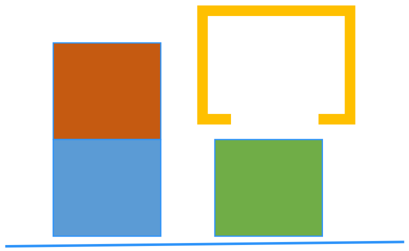


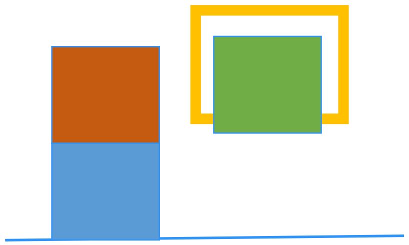


Preconditions	Effects
HandEmpty()	Add In-Hand(block)
Clear(block)	Delete HandEmpty()
On-Table(block)	On-Table(block)

Create a variable that takes on the value of a particular instance for all times it appears in an operator.

Operator: PutDown-Block on Table





<u>Preconditions</u>

In-Hand(block)

<u>Effects</u>

Add HandEmpty()

On-Table(block)

Delete In-Hand(block)

Why do we not need to check if ~HandEmpty() is true?

Operators for Block Stacking

Pickup_Table(b):

Pre: HandEmpty(), Clear(b), On-Table(b)

Add: In-Hand(b)

Delete: HandEmpty(), On-Table(b)

Putdown_Table(b):

Pre: In-Hand(b)

Add: HandEmpty(), On-Table(b)

Delete: In-Hand(b)

Pickup_Block(b,c):

Pre: HandEmpty(), On-Block(b,c), b!=c

Add: In-Hand(b), Clear(c)

Delete: HandEmpty(), On-Block(b,c)

Putdown_Block(b,c):

Pre: In-Hand(b), Clear(c)

Add: HandEmpty(), On-Block(b,c)

Delete: Clear(c), In-Hand(b)

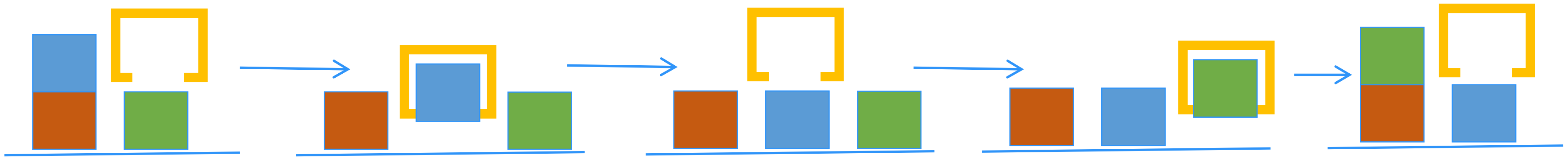
Why do we need separate operators for table vs on a block?

Example Matching Operators

HandEmpty() & On-Table(O) & On-Block(B,O) & Clear(B) & On-Table(G) & Clear(G)

Pickup_Block(b,c):
Pre: HandEmpty(), On-Block(b,c), b!=c
Add: In-Hand(b), Clear(c)
Delete: HandEmpty(), On(b,c)

Pickup_Table(b):
Pre: HandEmpty, Clear(b), On-Table(b)
Add: In-Hand(b)
Delete: HandEmpty(), On-Table(b)



Finding Plans with Symbolic Representations

Breadth-First Search

Sound? Yes

Complete? Yes

Optimal? Yes

Soundness - all solutions found are legal plans

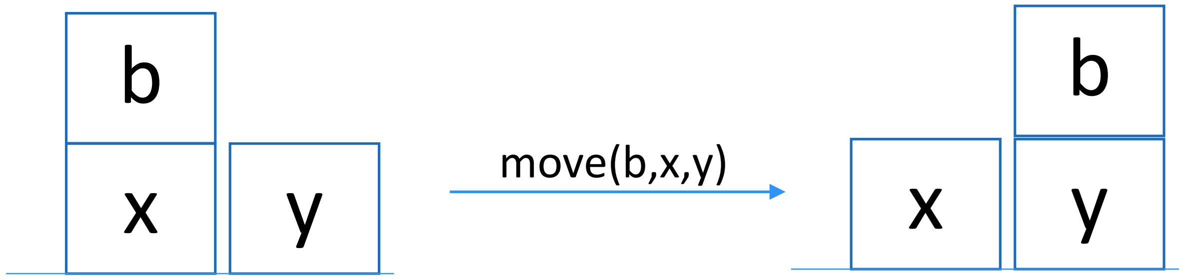
Completeness - a solution can be found whenever one actually exists

Optimality - the order in which solutions are found is consistent with some measure of plan quality

Linear Planning

- Since we have a conjunction of goal predicates, let’s try to solve one at a time
  - Maintain a stack of achievable goals
  - Use BFS (or anything else) to find a plan to achieve that single goal
  - Add a goal back on the stack if a later change makes it violated

Blocks World: Action Schemas



Pre-condition: Facts that must be true before the action can be applied.

Name

Move block b from x to y.

Move(b, x, y)
Pre-condition: on-block (b,x) & clear(b) & clear(y)
Effect: on(b,y) & clear(x) & !on(b,x) & !clear(y)

Effect: What facts have changed after the action has been applied to a state.

This is called a **schema** because b, x, and y are variables, and for a concrete action they must each be assigned to an object.

Notice that we must explicitly say that on(b,x) and clear(y) are no longer true.

Blocks World

MoveToTable(b,x) moves block b from the top of x onto the table

MoveToTable(b, x)
Pre-condition: on(b,x) & clear(b)
Effect: on(b,table) & clear(x) & !on(b,x)

The need for these two action schemas shows that coming up with the right actions in a planning problem can be tricky, even for relatively simple domains like blocksworld.

Move(b, x, y)
Pre-condition: on(b,x) & clear(b) & clear(y)
Effect: on(b,y) & clear(x) & !on(b,x) & !clear(y)

A General-purpose Planner

Planning to write a paper

Goal: Paper AND Contributed(You)

LearnAbout(x,y)

Preconditions: HasTimeForStudy(x)

Effects: Knows(x,y),
NOT(HasTimeForStudy(x))

HaveNewIdea(x)

Preconditions: Knows(x,AI),
Creative(x)

Effects: Idea, Contributed(x)

FindExistingOpenProblem(x)

Preconditions: Knows(x,AI)

Effects: Idea

ProveTheorems(x)

Preconditions: Knows(x,AI),
Knows(x,Math), Idea

Effect: Theorems, Contributed(x)

PerformExperiments(x)

Preconditions: Knows(x,AI),
Knows(x,Coding), Idea

Effect: Experiments, Contributed(x)

WritePaper(x)

Preconditions: Knows(x,AI),
Knows(x,Writing), Idea, Theorems,
Experiments

Effect: Paper, Contributed(x)

Some start states

Start1: HasTimeForStudy(You) AND Knows(You,Math) AND Knows(You,Coding) AND Knows(You,Writing)

Start2: HasTimeForStudy(You) AND Creative(You) AND Knows(Advisor,AI) AND Knows(Advisor,Math) AND Knows(Advisor,Coding) AND Knows(Advisor,Writing)

(Good luck with that plan…)

Start3: Knows(You,AI) AND Knows(You,Coding) AND Knows(OfficeMate,Math) AND HasTimeForStudy(OfficeMate) AND Knows(Advisor,AI) AND Knows(Advisor,Writing)

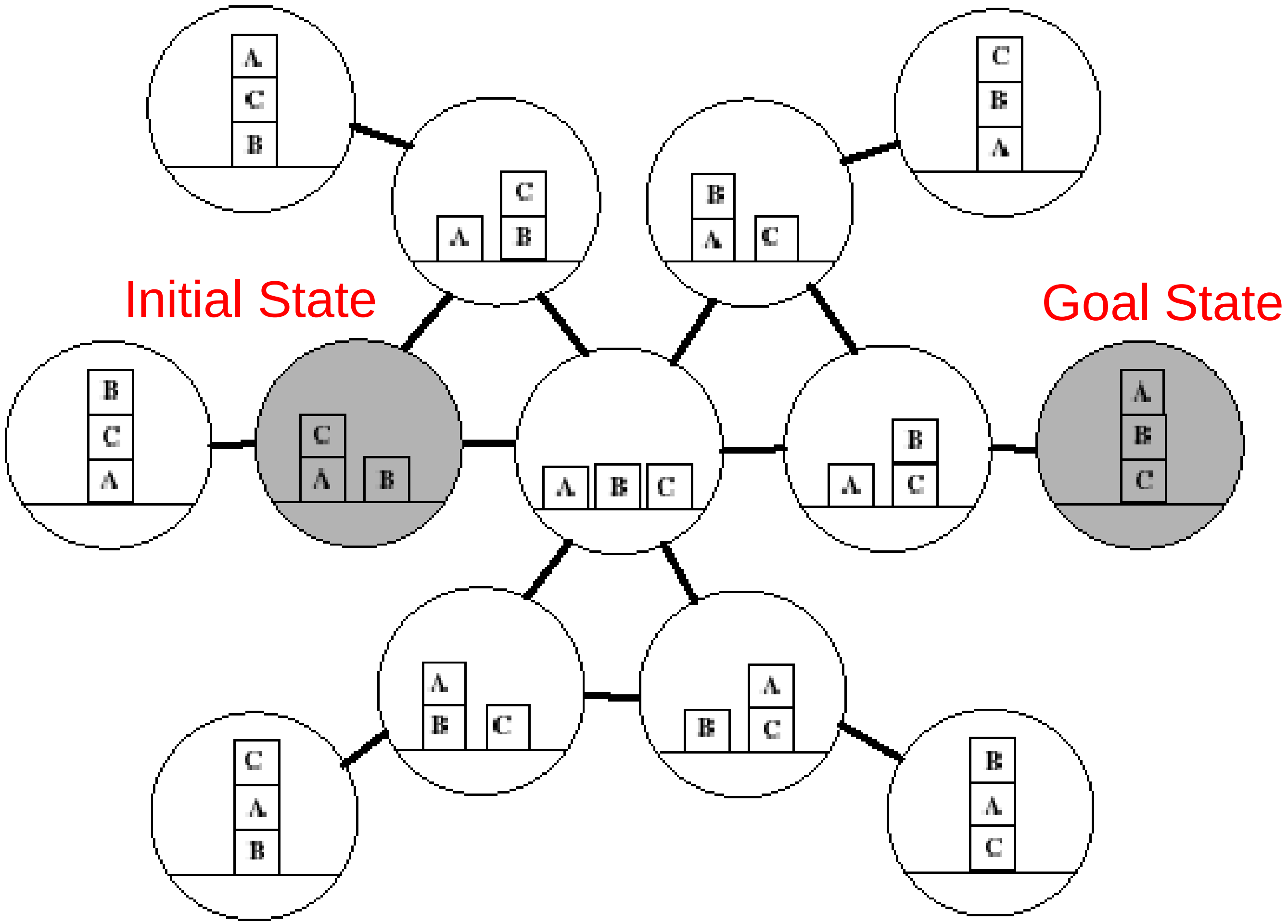
Start4: HasTimeForStudy(You) AND Knows(Advisor,AI) AND Knows(Advisor,Math) AND Knows(Advisor,Coding) AND Knows(Advisor,Writing)

Planning as Graph Search

- It is easy to view planning as a graph search problem
- Nodes/vertices = possible states
- Directed Arcs = STRIPS actions
- Solution: path from the initial state (i.e. vertex) to one state/vertices that satisfies the goal

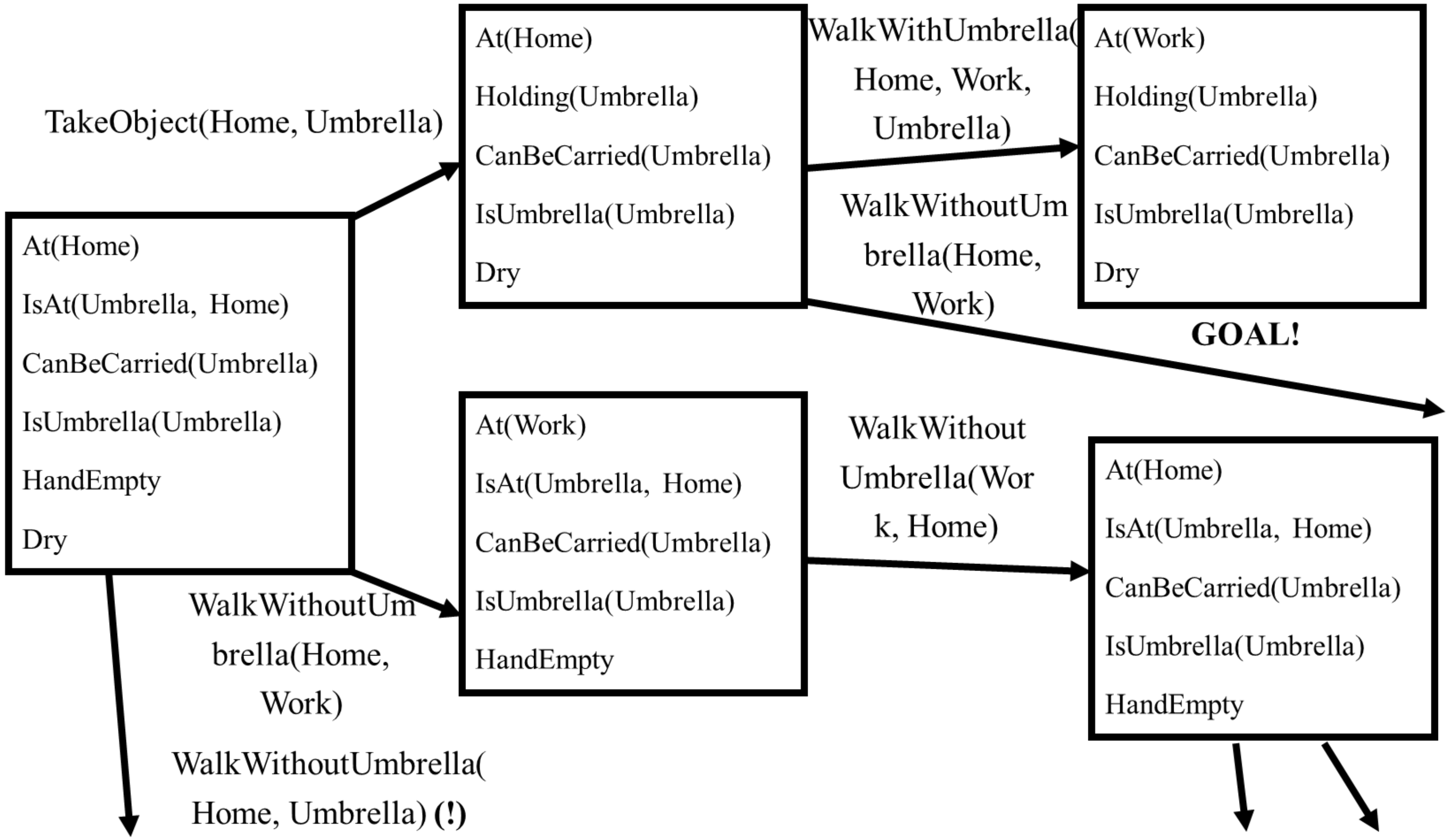
Search Space: Blocks World

Graph is finite



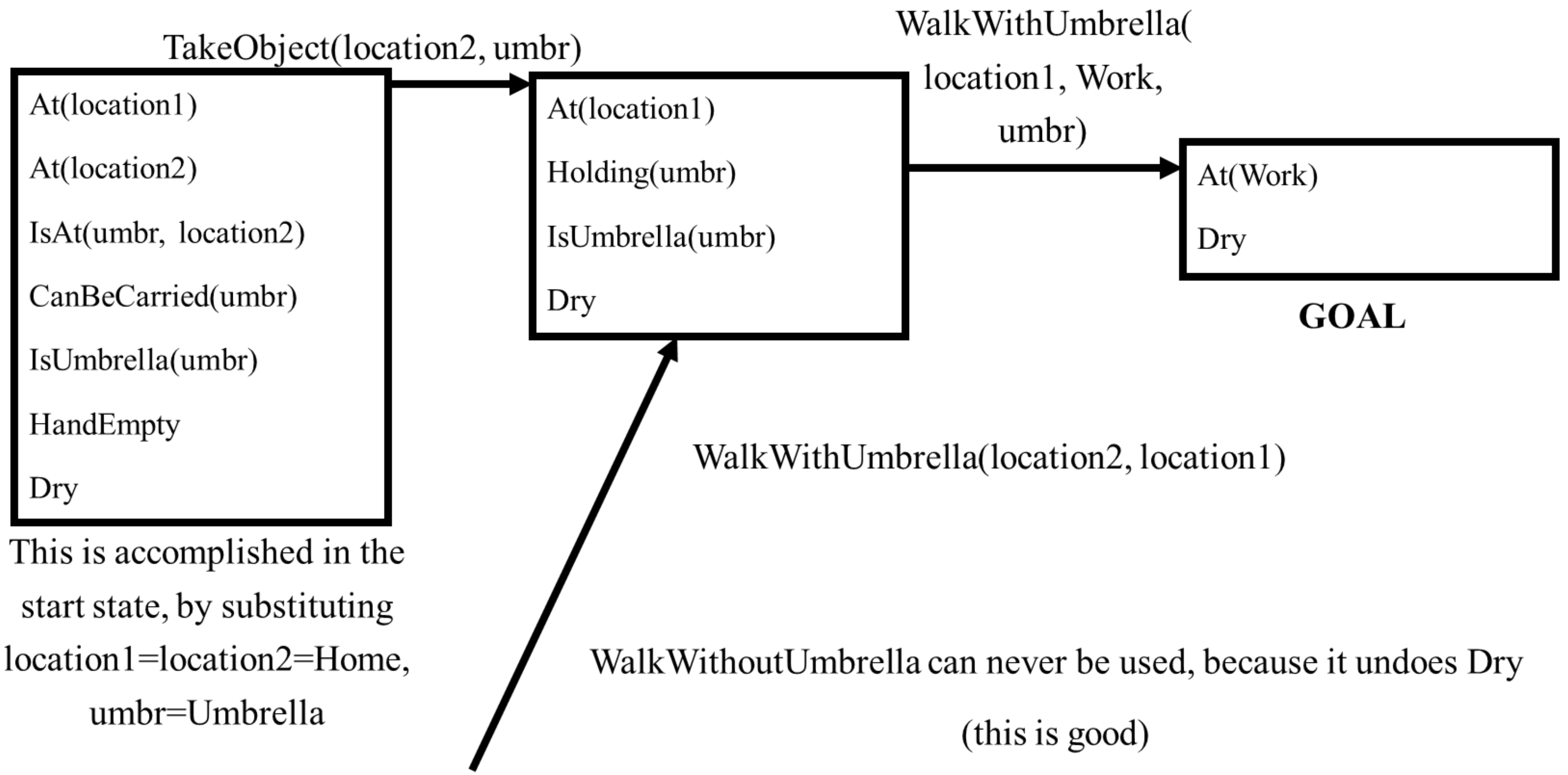
Forward state-space search (progression planning)

- Successors: all states that can be reached with an action whose preconditions are satisfied in current state



Backward state-space search (regression planning)

Predecessors: for every action that accomplishes one of the literals (and does not undo another literal), remove that literal and add all the preconditions



Other Kinds of Planners

- **Graphplan** is an influential planner from the 1990s that converts a planning problem (the actions and initial/goal state) into a kind of graph that can be efficiently searched
- **SATplan** is another influential planner that converts a planning problem into a SAT problem, and then uses a SAT solver to find a plan
  - The textbook describes how to do this transformation if you are curious
- **Partial Order Planners** are a kind of planner that searches the space of *plans* instead of the space of *states*
  - A partial order planner starts with an empty plan, and then inserts actions into the plan to make it better
  - For some applications (like dialog planning), this kind of planning makes a lot of sense, since it reasons directly about the actions, In general, though, they get out-performed by forward planners

Planning

Deterministic Planning

GraphPlan: Basic idea

- Construct a planning graph that encodes constraints on possible plans
- Use graph to constrain search for a valid plan
- Planning graph can be built for each problem in a relatively short time
- Extract a solution from planning graph

Planning as Graph Search

- Planning is just finding a path in a graph
  - Why not just use standard graph algorithms for finding paths?
- **Answer:** graphs are exponentially large in the problem encoding size (i.e. size of STRIPS problems).
  - But, standard algorithms are poly-time in graph size
  - So standard algorithms would require exponential time

- Can we do better than this?

Graphplan

Planning graph

- Directed, leveled graph with alternating layers of nodes
- Odd layers (state levels) represent candidate propositions that could possibly hold at step i
- Even layers (action levels) represent candidate actions that could possibly be executed at step i, including maintenance actions [do nothing]
- Arcs represent preconditions, adds and deletes
- Can only execute one real action at a step, but the data structure keeps track of all actions & states that are possible

Simple planning problem

Initial state: Have(cake)

Goal: Have(cake), Eaten(cake)

Action Eat(cake):
Preconditions: Have(cake)
Effects: ¬Have(cake), Eaten(cake)

Action Bake(cake):
Preconditions: ¬Have(cake)
Effects: Have(cake)

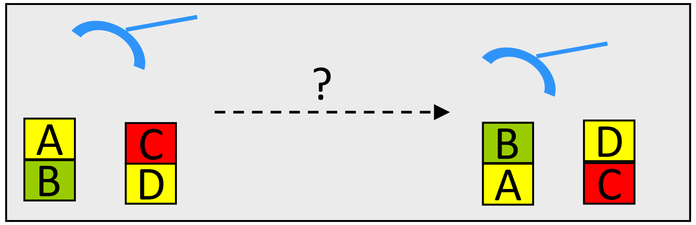
Solution:

Eat(cake)
Bake(cake)

- Phase 1 – Create a Planning Graph
  - built from initial state
  - contains actions and propositions that are possibly reachable from initial state
  - does not include unreachable actions or propositions
- Phase 2 - Solution Extraction
  - Backward search for the solution in the planning graph
    - backward from goal

Layered Plans

- Graphplan searches for **layered plans** (often called parallel plans)
- A layered plan is a sequence of **sets** of actions
  - actions in the same set must be **compatible**
    - a1 and a2 are compatible iff a1 does not delete preconditions or positive effects of a2 (and vice versa)
  - all sequential orderings of compatible actions gives same result



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

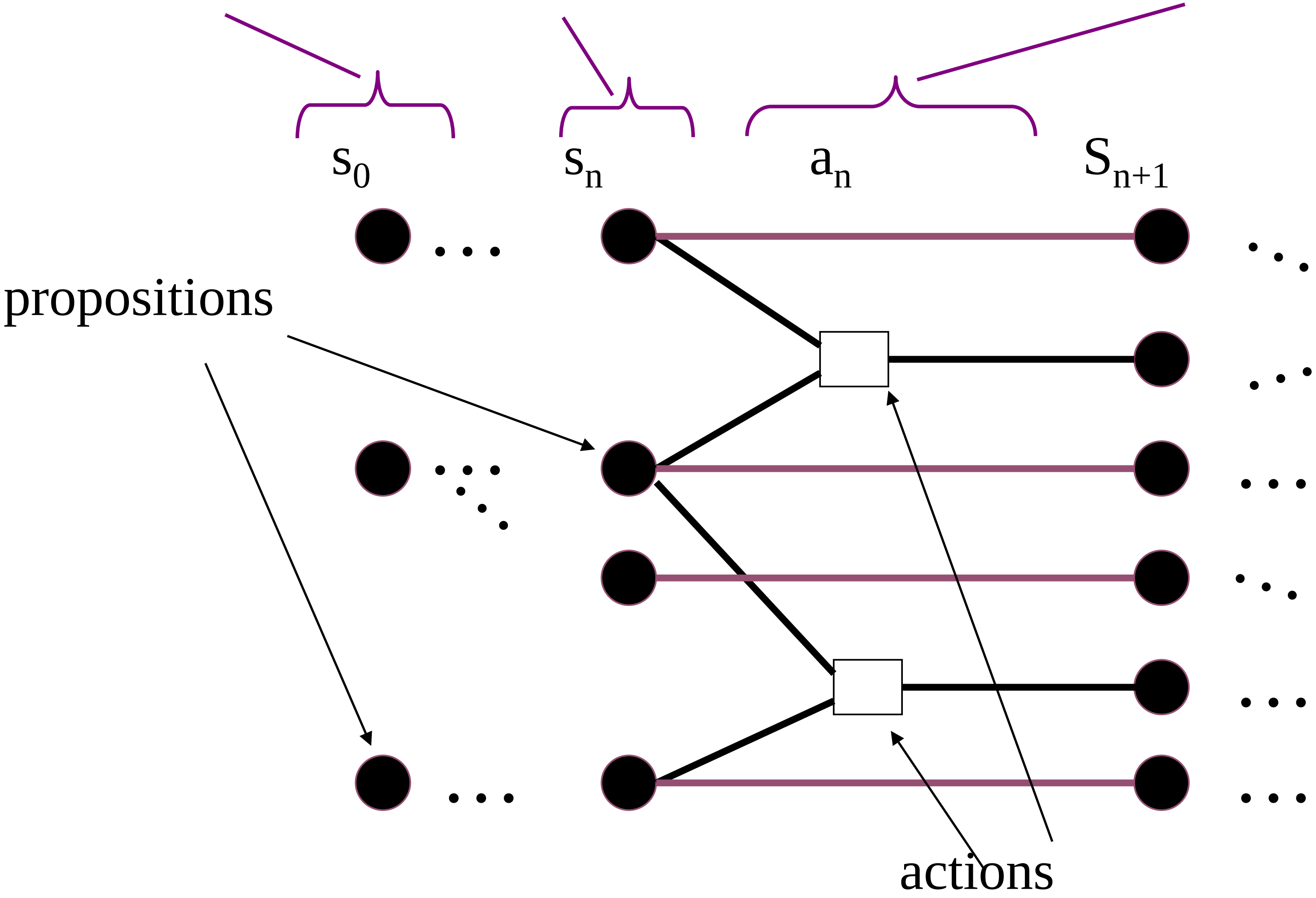
$$\left\{\begin{array}{l}\text{move(A,B,TABLE)}\\ \text{move(C,D,TABLE)}\end{array}\right\} ; \left\{\begin{array}{l}\text{move(B,TABLE,A)}\\ \text{move(D,TABLE,C)}\end{array}\right\}$$

Planning Graph

state-level 0: propositions true in s_0

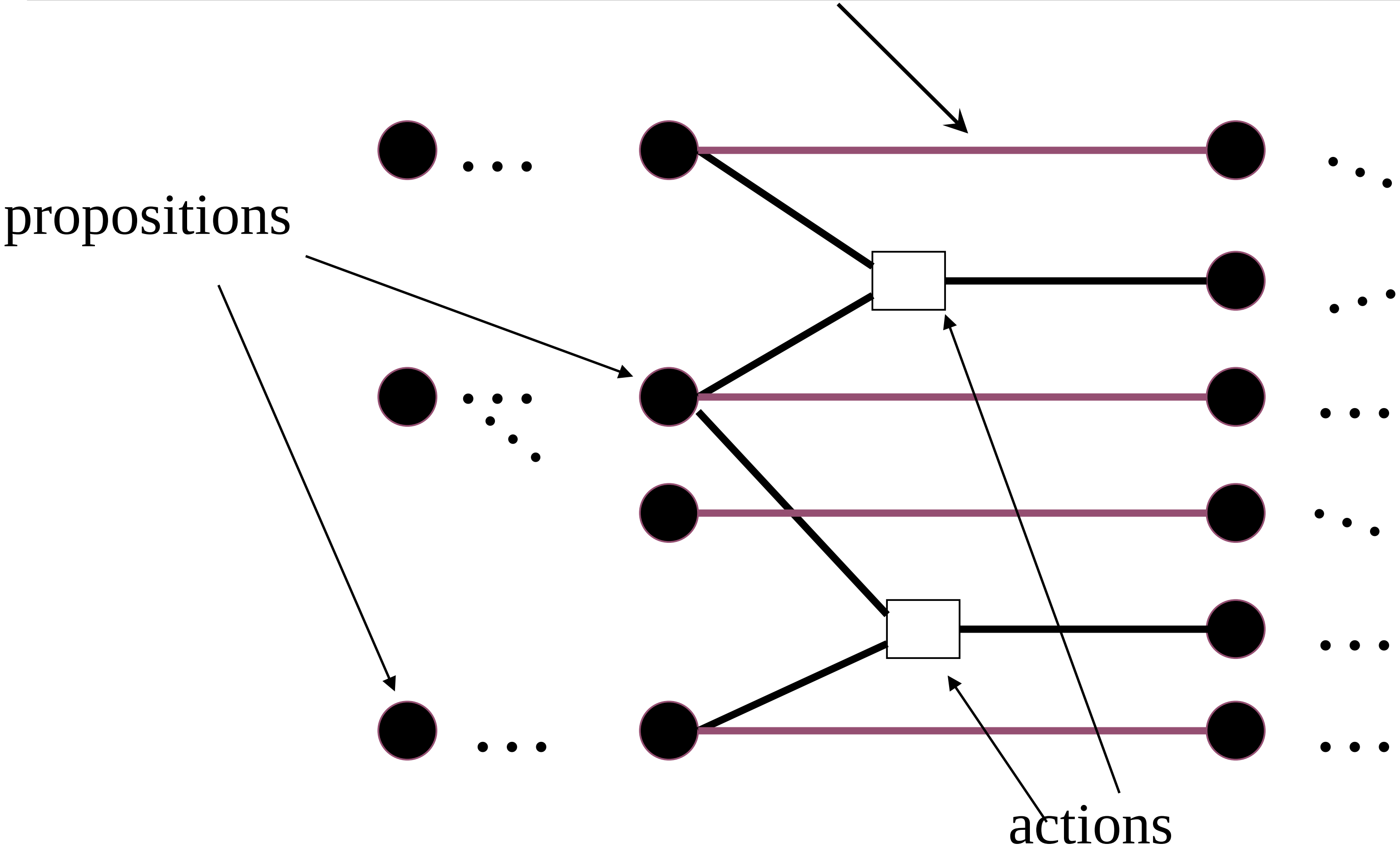
state-level *n*: literals that may possibly be true after some *n* level plan

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



Planning Graph

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

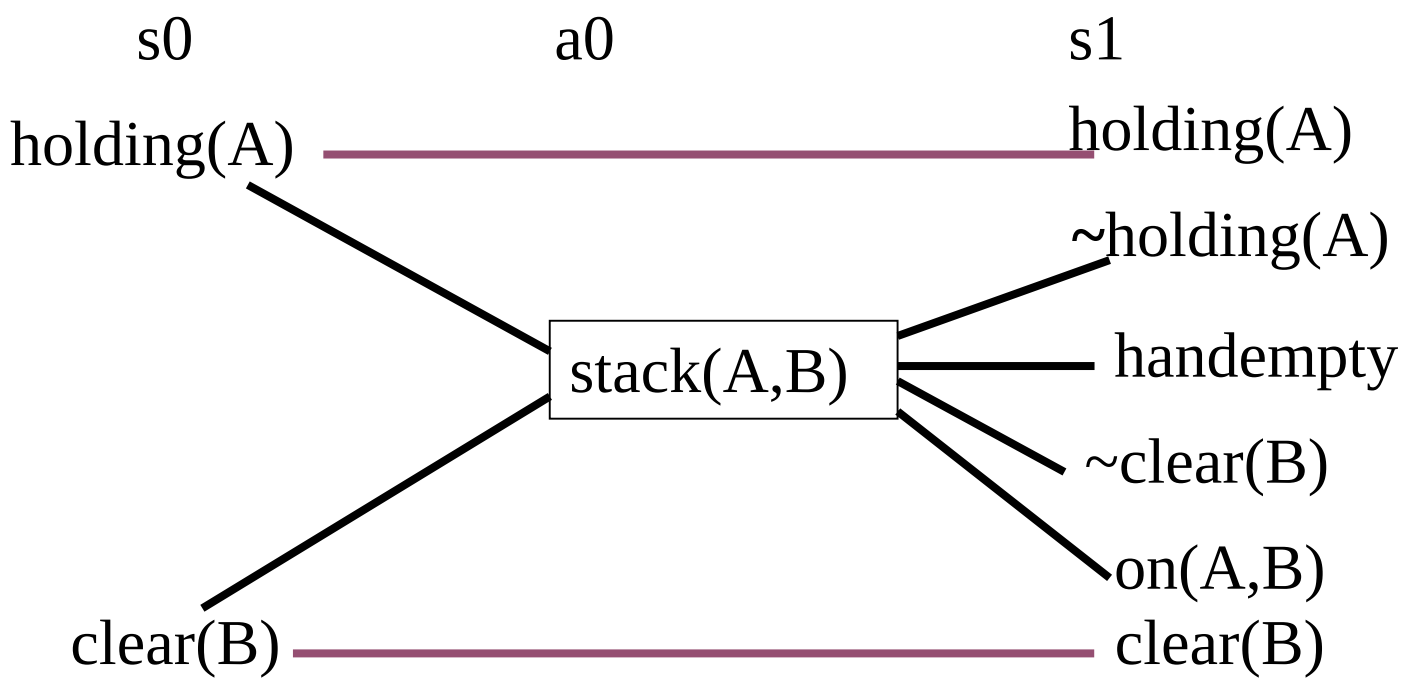


Graph expansion

- Initial proposition layer
  - Just the propositions in the initial state
- Action layer n
  - If all of an action's preconditions are in proposition layer n, then add action to layer *n*
- Proposition layer *n*+1
  - For each action at layer *n* (including persistence actions)
  - Add all its effects (both positive and negative) at layer **n**+1
    (Also allow propositions at layer *n* to persist to *n*+1)
- Propagate mutex information
  (we'll talk about this in a moment)

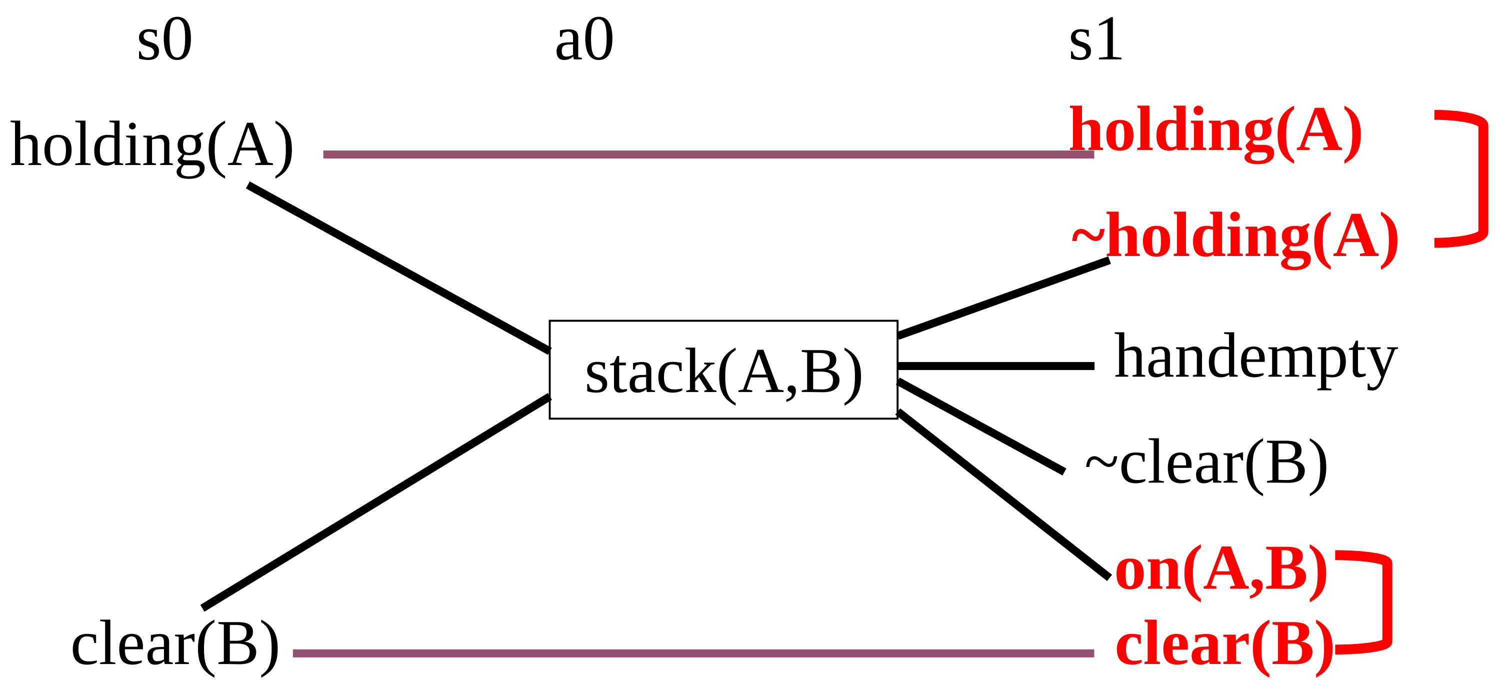
Example

stack(A,B)
precondition: holding(A), clear(B)
effect: ~holding(A), ~clear(B), on(A,B), clear(B), handempty



Example

stack(A,B)
precondition: holding(A), clear(B)
effect: ~holding(A), ~clear(B), on(A,B), clear(B), handempty



Notice that not all literals in s1 can be made true simultaneously after 1 level:
e.g. holding(A), ~holding(A) and on(A,B), clear(B)

Mutual Exclusion (Mutex)

- Mutex between pairs of actions at layer *n* means
  - no valid plan could contain both actions at layer *n*
  - E.g., stack(a,b), unstack(a,b)
- Mutex between pairs of literals at layer *n* means
  - no valid plan could produce both at layer *n*
  - E.g., clear(a), ~clear(a)
    on(a,b), clear(b)
- GraphPlan checks pairs only
  - mutex relationships can help rule out possibilities during search in phase 2 of Graphplan

Mutual exclusion links between actions

- **Inconsistent effects:** one action negates an effect of another.
- **Interference:** one of the effects of one action is the negation of the precondition for another.
- **Competing needs:** one of the preconditions of one action is mutually exclusive with a precondition of the other.
- **Negation:** one proposition is the negation of the other.
- **Inconsistent support:** all the actions for establishing one proposition are mutually exclusive with the actions of establishing the other proposition.

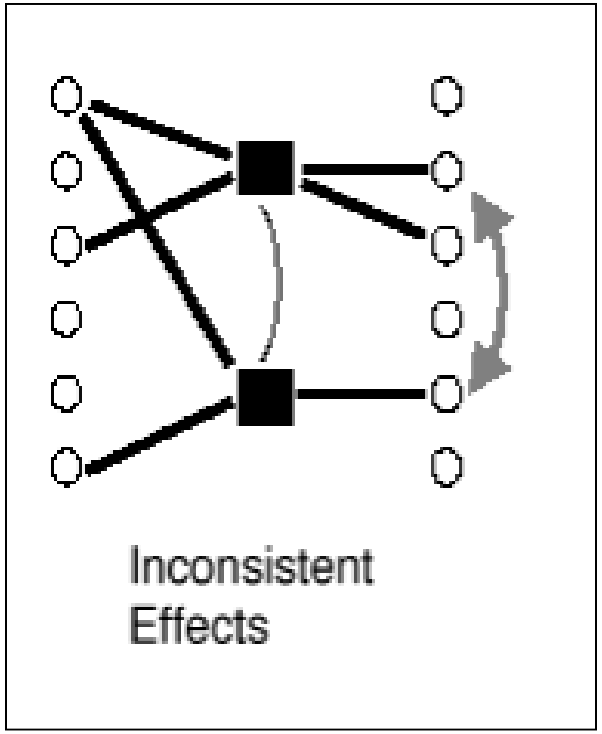
Action Mutex: condition 1

- Inconsistent effects
  - an effect of one negates an effect of the other

- E.g., stack(a,b) & unstack(a,b)

add handempty

delete handempty
(add ~handempty)

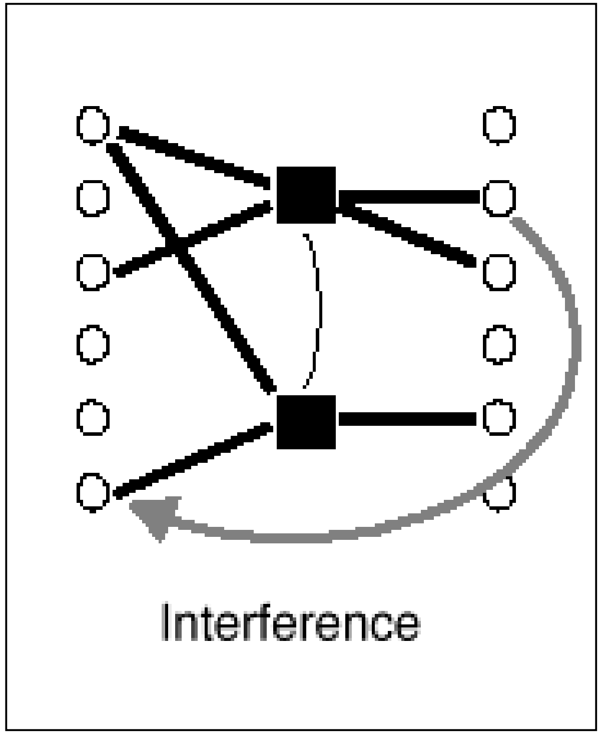


Action Mutex: condition 2

- *Interference :*
  - one deletes a precondition of the other

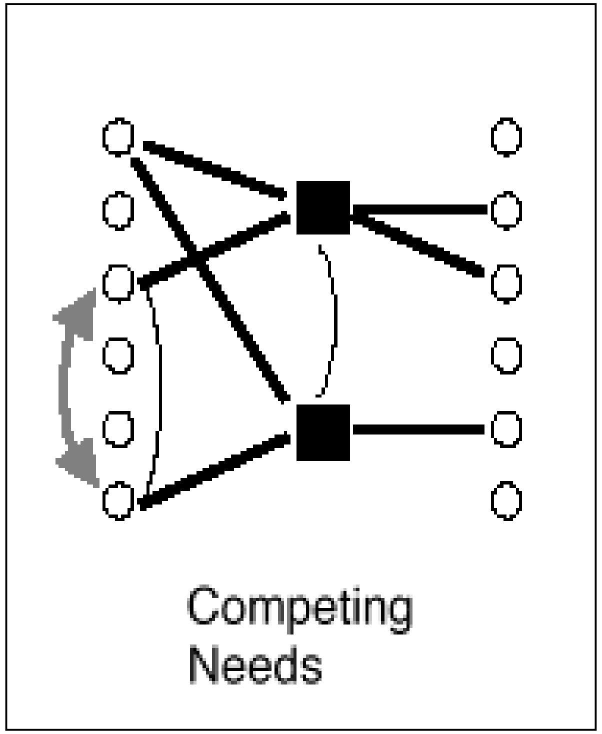
- E.g., stack(a,b) & putdown(a)

deletes holdindg(a) needs holding(a)



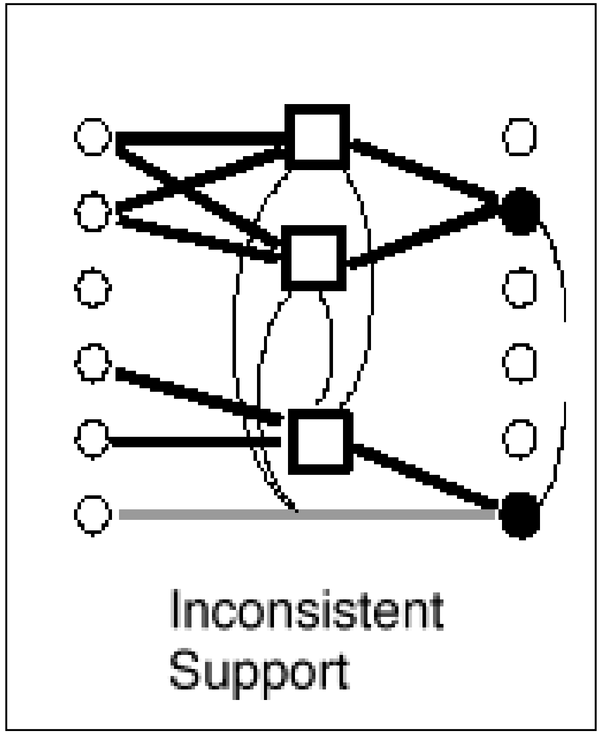
Action Mutex: condition 3

- ***Competing needs:***
  - they have mutually exclusive preconditions
  - Their preconditions can't be true at the same time



Literal Mutex: two conditions

- *Inconsistent support :*
  - one is the negation of the other
    E.g., handempty and ~handempty
  - or all ways of achieving them via actions are are pairwise mutex



- **Inconsistent effects:** one action negates an effect of another.
    - Eat(cake) deletes Have(cake) so is inconsistent with persistence of Have(cake)
    - Eat(cake) adds Eaten(cake) so is inconsistent with persistence of ¬Eaten(Cake)
- **Interference:** one of the effects of one action is the negation of the precondition for another.
    - Eat(cake) negates the preconditions of the persistence actions for Have(cake) and ¬Eaten(Cake)
- **Competing needs:** one of the preconditions of one action is mutually exclusive with a precondition of the other.
    - Bake(cake) requires ¬Have(cake) while Eat(cake) requires Have(cake)
- **Negation:** one proposition is the negation of the other.
    - Have(cake) and ¬Have(cake) are mutually exclusive.
- **Inconsistent support:** all the actions for establishing one proposition are mutually exclusive with the actions of establishing the other proposition.
    - Have(cake) and Eaten(cake) are mutex in S_1 because all their establishing actions are mutex
    - Have(cake) and Eaten(cake) are not mutex in S_2

Simple planning problem

Initial state: Have(cake)

Goal: Have(cake), Eaten(cake)

Action Eat(cake):
- Preconditions: Have(cake)
- Effects: ¬Have(cake), Eaten(cake)

Action Bake(cake):
- Preconditions: ¬Have(cake)
- Effects: Have(cake)

Solution:

Eat(cake)
Bake(cake)

Planning Graph for Cake Example

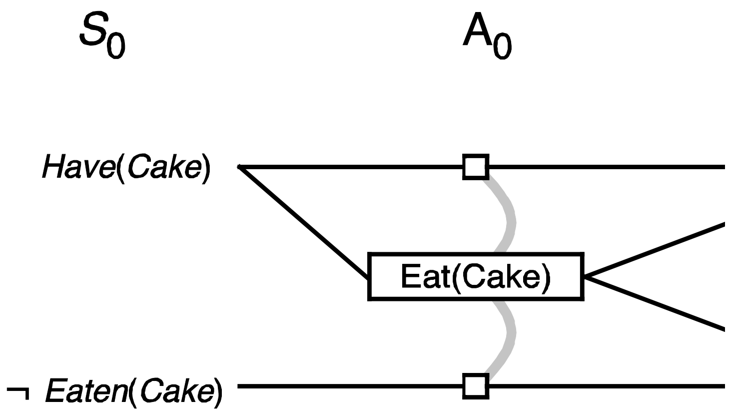
S_0

Have(*Cake*)

¬ *Eaten*(*Cake*)

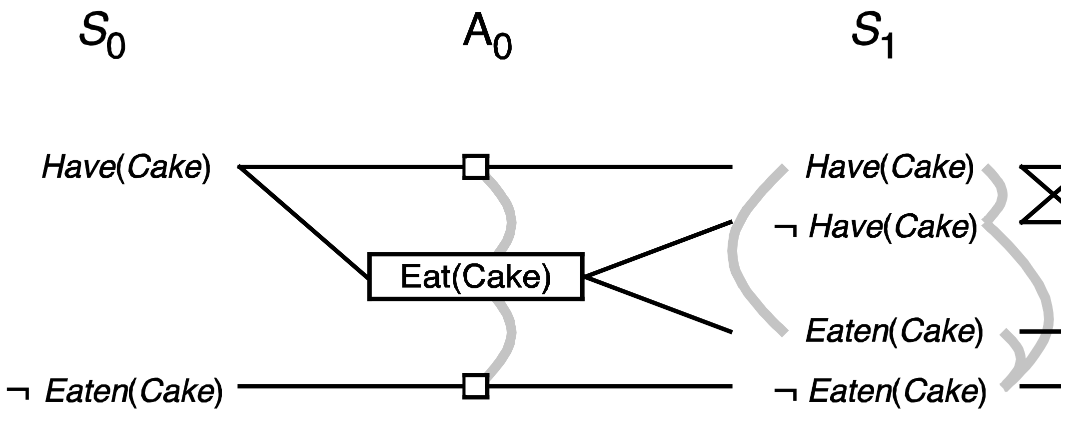
- Level S_0 has all literals from initial state

Planning Graph for Cake Example



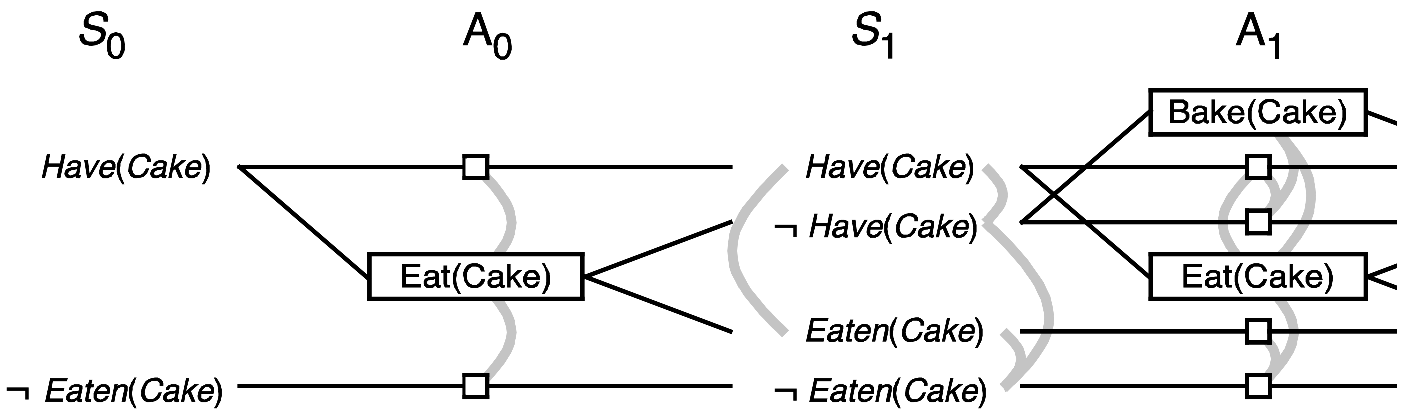
- Level S_0 has all literals from initial state
- **Level A_0 has all actions whose preconditions are satisfied in S_0, including no-ops**

Planning Graph for Cake Example



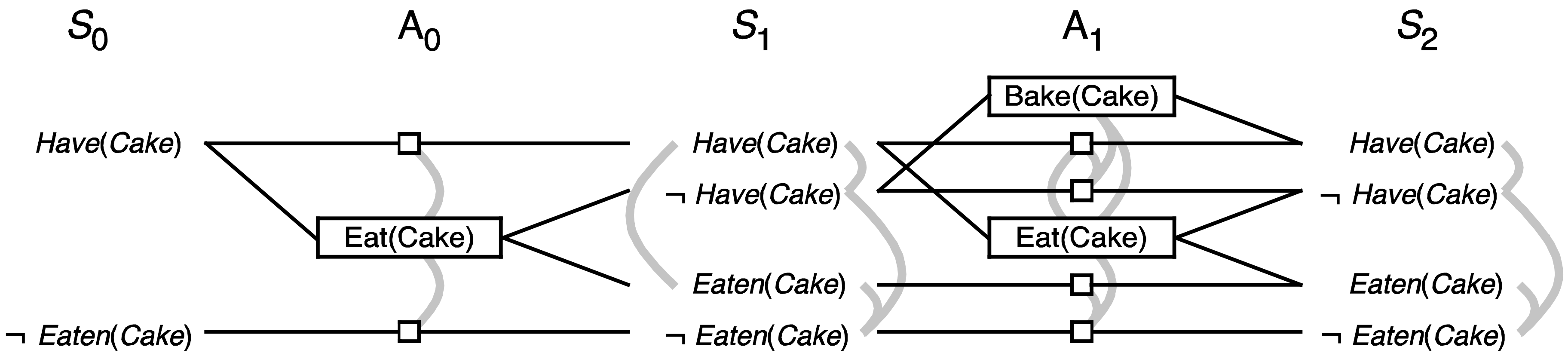
- Level S_0 has all literals from initial state
- Level A_0 has all actions whose preconditions are satisfied in S_0, including no-ops
- **Actions connect preconditions to effects**
- **Gray arcs connect propositions that are mutex (mutually exclusive) & actions that are mutex**

Planning Graph for Cake Example



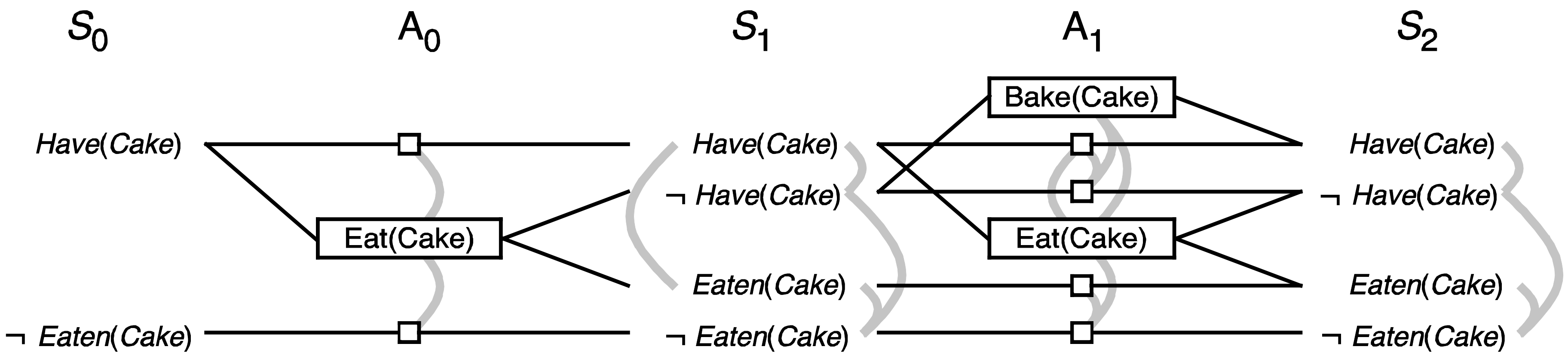
- Actions connect preconditions to effects
- Gray arcs connect propositions that are mutex
- **Actions at level A_i must have support from a set of literals in state S_i that have no mutex relations among themselves**

Planning Graph for Cake Example



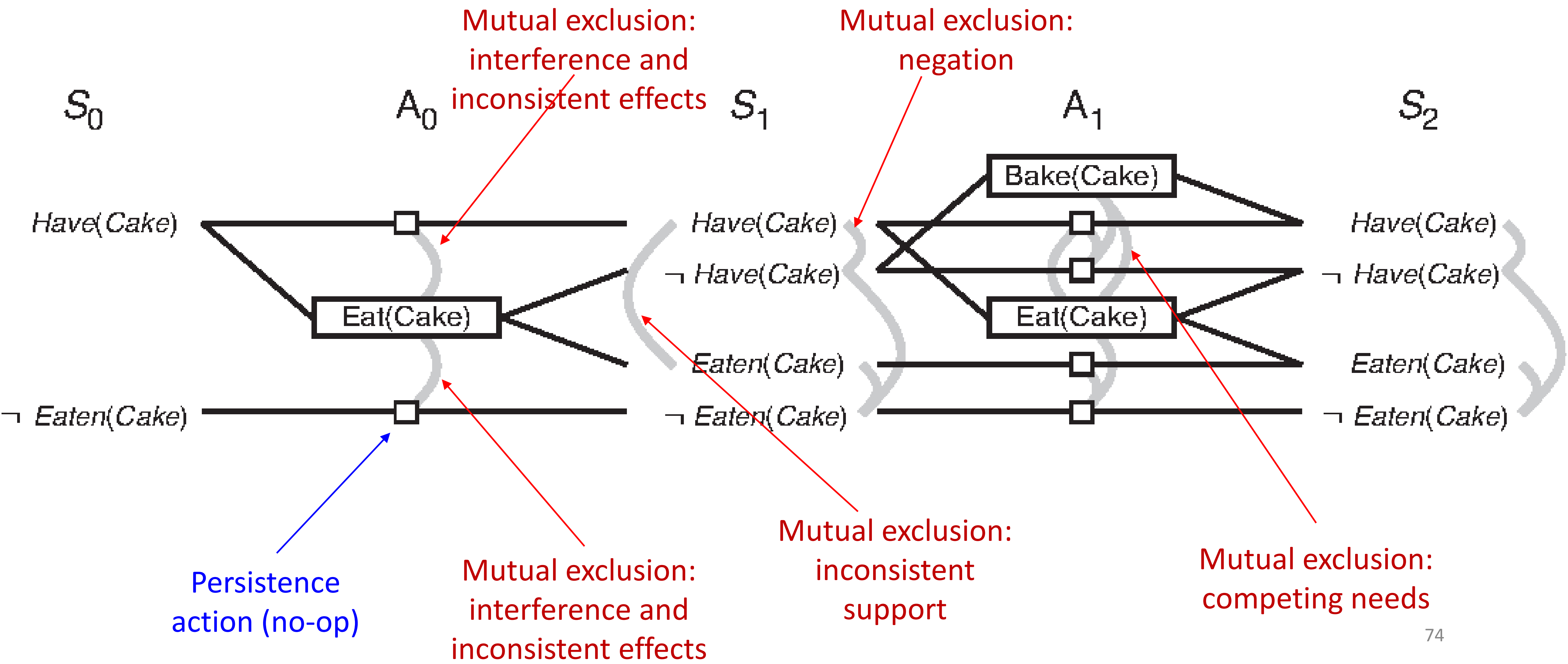
- Actions at level A_i must have support from a set of literals in state S_i that have no mutex relations among themselves
- **Stop when the set of literals does has not changed**

Planning Graph for Cake Example



- If all of the literals in the goal are in the final state and are non-mutex …
- We can try to extract a plan from the plan graph

Planning graph representation of cake problem



GraphPlan

function GRAPHPLAN(*problem*) **returns** solution or failure

graph ← INITIAL-PLANNING-GRAPH(*problem*)

goals ← CONJUNCTS(*problem*.GOAL)

nogoods ← an empty hash table

for $t = 0$ **to** ∞ **do**

if *goals* all non-mutex in S_t of *graph* **then**

solution ← EXTRACT-SOLUTION(*graph, goals,* NUMLEVELS(*graph*), *nogoods*)

if *graph* and *nogoods* have both leveled off **then return** *failure*

graph ← EXPAND-GRAPH(*graph*, *problem*)

From Fig. 10.9, p. 383

Solution Extraction: Backward Search

Search problem:

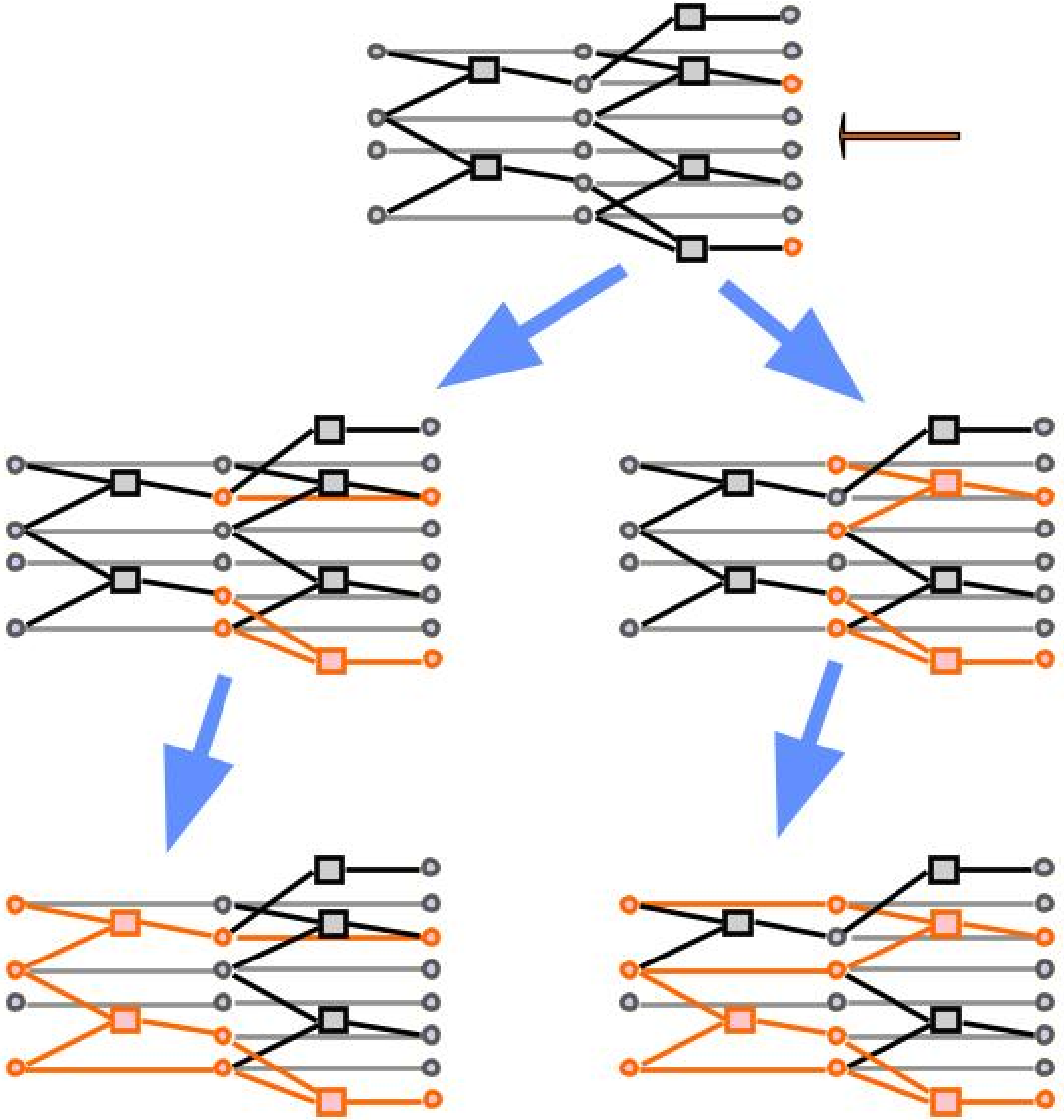
Start state: goal set at last level

Actions: conflict-free ways of achieving the current goal set

Terminal test: at S0 with goal set entailed by initial planning state

Note: may need to start much deeper than the leveling-off point!

Caching, good ordering is important



Important Ideas

- Plan graph construction is polynomial time
  - Though construction can be expensive when there are many “objects” and hence many propositions
- The plan graph captures important properties of the planning problem
  - Necessarily unreachable literals and actions
  - Possibly reachable literals and actions
  - Mutually exclusive literals and actions
- Significantly prunes search space compared to previously considered planners
- Plan graphs can also be used for deriving admissible (and good non-admissible) heuristics

Spare Tire Problem

Init(Tire(Flat) $\wedge$ Tire(Spare) $\wedge$ At(Flat,Axle) $\wedge$ At(Spare,Trunk))

Goal(At(Spare,Axle))

Action(Remove(obj,loc),
PRECOND: At(obj,loc),
EFFECT: $\neg$At(obj,loc) $\wedge$ At(obj,Ground))

Action(PutOn(t, Axle),
PRECOND: Tire(t) $\wedge$ At(t,Ground) $\wedge$ $\neg$At(Flat,Axle),
EFFECT: $\neg$At(t,Ground) $\wedge$ At(t,Axle))

Action(LeaveOvernight,
PRECOND: $\varnothing$,
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) $\wedge$ $\neg$At(Flat,Trunk))

From Fig. 10.2, p. 370

SATPlan

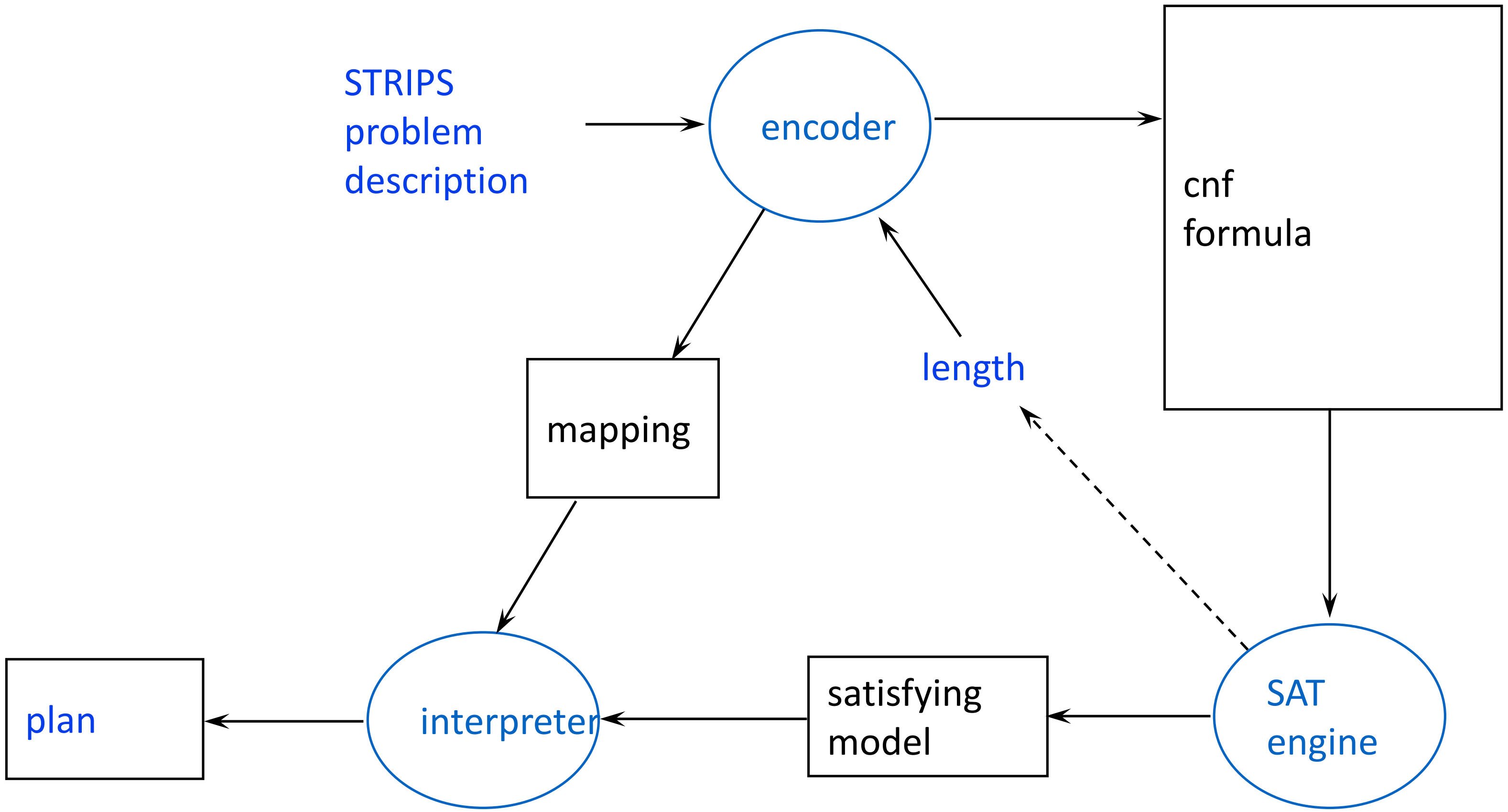
- Formulate the planning problem as a CSP
- Assume that the plan has k actions
- Create a binary variable for each possible action a:
  - Action(a,i) (TRUE if action a is used at step i)
- Create variables for each proposition that can hold at different points in time:
  - Proposition(p,i) (TRUE if proposition p holds at step i)

Constraints

- Only one action can be executed at each time step (XOR constraints)
- Constraints describing effects of actions
- Persistence: if an action does not change a proposition p, then p’s value remains unchanged
- A proposition is true at step i only if some action (possibly a maintain action) made it true
- Constraints for initial state and goal state

Now apply our favorite CSP solver!

SATPLAN


STRIPS
problem
description
encoder
cnf
formula
mapping
length
plan
interpreter
satisfying
model
SAT
engine