

# Fundamentals of Artificial Intelligence

## Chapter 10: **Classical Planning**

**Roberto Sebastiani**

DISI, Università di Trento, Italy – [roberto.sebastiani@unitn.it](mailto:roberto.sebastiani@unitn.it)

[https://disi.unitn.it/rseba/DIDATTICA/fai\\_2024/](https://disi.unitn.it/rseba/DIDATTICA/fai_2024/)

Teaching assistants:

**Mauro Dragoni**, [dragoni@fbk.eu](mailto:dragoni@fbk.eu), <https://www.maurodragoni.com/teaching/fai/>

**Paolo Morettin**, [paolo.morettin@unitn.it](mailto:paolo.morettin@unitn.it), <https://paolomorettin.github.io/>

M.S. Course “Artificial Intelligence Systems”, academic year 2024-2025

Last update: Thursday 5<sup>th</sup> September, 2024, 18:59

Copyright notice: Most examples and images displayed in the slides of this course are taken from [Russell & Norvig, “Artificial Intelligence, a Modern Approach”, 3<sup>rd</sup> ed., Pearson], including explicitly figures from the above-mentioned book, so that their copyright is detained by the authors. A few other material (text, figures, examples) is authored by (in alphabetical order): Pieter Abbeel, Bonnie J. Dorr, Anca Dragan, Dan Klein, Nikita Kitaev, Tom Lenaerts, Michela Milano, Dana Nau, Maria Simi, who detain its copyright. ▶ ☰ 🔍 ↻

These slides cannot be displayed in public without the permission of the author.

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Outline

- 1 **Basics on Planning**
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Automated Planning (aka “Planning”)

## Automated Planning

Synthesize a sequence of actions (plan) to be performed by an agent leading from an initial state of the world to a set of target states (goal)

- Planning is both:
  - an application per se
  - a common activity in many applications  
(e.g. design & manufacturing, scheduling, robotics,...)
- Similar to problem-solving agents (Ch.03), with factored/structured representation of states
- “Classical” Planning (this chapter):  
fully observable, deterministic, static environments with single agents

# Automated Planning (aka “Planning”)

## Automated Planning

Synthesize a sequence of actions (plan) to be performed by an agent leading from an initial state of the world to a set of target states (goal)

- Planning is both:
  - an application per se
  - a common activity in many applications  
(e.g. design & manufacturing, scheduling, robotics,...)
- Similar to problem-solving agents (Ch.03), with factored/structured representation of states
- “Classical” Planning (this chapter):  
fully observable, deterministic, static environments with single agents

# Automated Planning (aka “Planning”)

## Automated Planning

Synthesize a sequence of actions (plan) to be performed by an agent leading from an initial state of the world to a set of target states (goal)

- Planning is both:
  - an application per se
  - a common activity in many applications  
(e.g. design & manufacturing, scheduling, robotics,...)
- Similar to problem-solving agents (Ch.03), with factored/structured representation of states
- “Classical” Planning (this chapter):  
fully observable, deterministic, static environments with single agents

# Automated Planning (aka “Planning”)

## Automated Planning

Synthesize a sequence of actions (plan) to be performed by an agent leading from an initial state of the world to a set of target states (goal)

- Planning is both:
  - an application per se
  - a common activity in many applications  
(e.g. design & manufacturing, scheduling, robotics,...)
- Similar to problem-solving agents (Ch.03), with factored/structured representation of states
- “Classical” Planning (this chapter):  
fully observable, deterministic, static environments with single agents



# Automated Planning [cont.]

## Automated Planning

- Given:
  - an initial state
  - a set of actions you can perform
  - a (set of) state(s) to achieve (goal)
- Find:
  - a **plan**: a partially- or totally-ordered set of actions needed to achieve the goal from the initial state

# Automated Planning [cont.]

## Automated Planning

- Given:
  - an initial state
  - a set of actions you can perform
  - a (set of) state(s) to achieve (goal)
- Find:
  - a **plan**: a partially- or totally-ordered set of actions needed to achieve the goal from the initial state

# Decidability and Complexity

- **PlanSAT**: the question of whether there exists any plan that solves a planning problem
  - decidable for classical planning
  - with function symbols, the number of states becomes infinite
    - ⇒ undecidable
  - in PSPACE
    - harder than NP, no polynomial-size witness (e.g., Tower of Hanoi)
- **Bounded PlanSAT**: the question of whether there exists any plan of a given length  $k$  or less
  - can be used for optimal-length plan
  - decidable for classical planning
  - decidable even in the presence of function symbols
  - in PSPACE, NP for many problems of interest

# Decidability and Complexity

- **PlanSAT**: the question of whether there exists any plan that solves a planning problem
  - decidable for classical planning
  - with function symbols, the number of states becomes infinite
    - ⇒ undecidable
  - in PSPACE
    - harder than NP, no polynomial-size witness (e.g., Tower of Hanoi)
- **Bounded PlanSAT**: the question of whether there exists any plan of a given length  $k$  or less
  - can be used for optimal-length plan
  - decidable for classical planning
  - decidable even in the presence of function symbols
  - in PSPACE, NP for many problems of interest

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# A Language for Planning: PDDL

## Planning Domain Definition Language (PDDL)

- A **state** is a conjunction of **fluents**: **ground, function-less atoms**
  - ex:  $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - ex of non-fluents:  $At(x, y)$  (non ground),  $\neg Poor$  (negated),  $At(Father(Fred), Sydney)$  (not function-less)
    - **closed-world assumption**: all non-mentioned fluents are false
    - **unique-name assumption**: distinct names refer to distinct objects
- **Actions** are described by a set of **action schemata**
  - concise description: **describe which fluent change**  
 $\implies$  the other fluents implicitly maintain their values
- **Action Schema**: consists in **action name**, a **list of variables** in the schema, the **precondition**, the **effect** (aka **postcondition**)
  - precondition and effect are **conjunctions of literals** (positive or negated atomic sentences)
  - **lifted representation**: variables implicitly **universally quantified**
- Can be instantiated into (ground) actions

# A Language for Planning: PDDL

## Planning Domain Definition Language (PDDL)

- A **state** is a conjunction of **fluents**: **ground, function-less atoms**
  - ex:  $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - ex of non-fluents:  $At(x, y)$  (non ground),  $\neg Poor$  (negated),  $At(Father(Fred), Sydney)$  (not function-less)
  - **closed-world assumption**: all non-mentioned fluents are false
  - **unique-name assumption**: distinct names refer to distinct objects
- **Actions** are described by a set of **action schemata**
  - concise description: **describe which fluent change**  
 $\implies$  the other fluents implicitly maintain their values
- **Action Schema**: consists in **action name**, a **list of variables** in the schema, the **precondition**, the **effect** (aka **postcondition**)
  - precondition and effect are **conjunctions of literals** (positive or negated atomic sentences)
  - **lifted representation**: variables implicitly **universally quantified**
- Can be instantiated into (ground) actions

# A Language for Planning: PDDL

## Planning Domain Definition Language (PDDL)

- A **state** is a conjunction of **fluents**: **ground, function-less atoms**
  - ex:  $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - ex of non-fluents:  $At(x, y)$  (non ground),  $\neg Poor$  (negated),  $At(Father(Fred), Sydney)$  (not function-less)
  - **closed-world assumption**: all non-mentioned fluents are false
  - **unique-name assumption**: distinct names refer to distinct objects
- **Actions** are described by a set of **action schemata**
  - concise description: **describe which fluent change**  
 $\implies$  the other fluents implicitly maintain their values
- **Action Schema**: consists in **action name**, a **list of variables** in the schema, the **precondition**, the **effect** (aka **postcondition**)
  - precondition and effect are **conjunctions of literals** (positive or negated atomic sentences)
  - **lifted representation**: variables implicitly **universally quantified**
- Can be instantiated into (ground) actions



# A Language for Planning: PDDL

## Planning Domain Definition Language (PDDL)

- A **state** is a conjunction of **fluents**: **ground, function-less atoms**
  - ex:  $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - ex of non-fluents:  $At(x, y)$  (non ground),  $\neg Poor$  (negated),  $At(Father(Fred), Sydney)$  (not function-less)
  - **closed-world assumption**: all non-mentioned fluents are false
  - **unique-name assumption**: distinct names refer to distinct objects
- **Actions** are described by a set of **action schemata**
  - concise description: **describe which fluent change**  
 $\implies$  the other fluents implicitly maintain their values
- **Action Schema**: consists in **action name**, a **list of variables** in the schema, the **precondition**, the **effect** (aka **postcondition**)
  - precondition and effect are **conjunctions of literals** (positive or negated atomic sentences)
  - **lifted representation**: variables implicitly **universally quantified**
- Can be instantiated into (ground) actions

# A Language for Planning: PDDL

## Planning Domain Definition Language (PDDL)

- A **state** is a conjunction of **fluents**: **ground, function-less atoms**
  - ex:  $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - ex of non-fluents:  $At(x, y)$  (non ground),  $\neg Poor$  (negated),  $At(Father(Fred), Sydney)$  (not function-less)
  - **closed-world assumption**: all non-mentioned fluents are false
  - **unique-name assumption**: distinct names refer to distinct objects
- **Actions** are described by a set of **action schemata**
  - concise description: **describe which fluent change**  
 $\implies$  the other fluents implicitly maintain their values
- **Action Schema**: consists in **action name**, a **list of variables** in the schema, the **precondition**, the **effect** (aka **postcondition**)
  - precondition and effect are **conjunctions of literals** (positive or negated atomic sentences)
  - **lifted representation**: variables implicitly **universally quantified**
- Can be instantiated into (ground) actions

# A Language for Planning: PDDL

## Planning Domain Definition Language (PDDL)

- A **state** is a conjunction of **fluents**: **ground, function-less atoms**
  - ex:  $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - ex of non-fluents:  $At(x, y)$  (non ground),  $\neg Poor$  (negated),  $At(Father(Fred), Sydney)$  (not function-less)
  - **closed-world assumption**: all non-mentioned fluents are false
  - **unique-name assumption**: distinct names refer to distinct objects
- **Actions** are described by a set of **action schemata**
  - concise description: **describe which fluent change**  
 $\implies$  the other fluents implicitly maintain their values
- **Action Schema**: consists in **action name**, a **list of variables** in the schema, the **precondition**, the **effect** (aka **postcondition**)
  - precondition and effect are **conjunctions of literals** (positive or negated atomic sentences)
  - **lifted representation**: variables implicitly **universally quantified**
- Can be instantiated into (ground) actions

# PDDL: Example

- Action schema:

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

*PRECOND* :  $\text{Plane}(p) \wedge \text{Airport}(\text{from}) \wedge \text{Airport}(\text{to}) \wedge \text{At}(p, \text{from})$

*EFFECT* :  $\neg \text{At}(p, \text{from}) \wedge \text{At}(p, \text{to})$ )

- Action instantiation:

*Action*(Fly( $P_1$ , SFO, JFK),

*PRECOND* :  $\text{Plane}(P_1) \wedge \text{Airport}(\text{SFO}) \wedge \text{Airport}(\text{JFK}) \wedge \text{At}(P_1, \text{SFO})$

*EFFECT* :  $\neg \text{At}(P_1, \text{SFO}) \wedge \text{At}(P_1, \text{JFK})$ )

# PDDL: Example

- Action schema:

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

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

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

- Action instantiation:

*Action*(*Fly*(*P*<sub>1</sub>, *SFO*, *JFK*),

*PRECOND* : *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(*SFO*)  $\wedge$  *Airport*(*JFK*)  $\wedge$  *At*(*P*<sub>1</sub>, *SFO*)

*EFFECT* :  $\neg$ *At*(*P*<sub>1</sub>, *SFO*)  $\wedge$  *At*(*P*<sub>1</sub>, *JFK*)

# A Language for Planning: PDDL [cont.]

- **Precondition**: must hold to ensure the action can be executed
  - defines the states in which the action can be executed
  - action is **applicable** in state  $s$  if the preconditions are satisfied by  $s$
- **Effect**: represent the effects of the action on the world
  - defines the result of executing the action
- **Add list (ADD(a))**: (the fluents in) the positive literals in the action's effects
  - ex:  $\{At(p, to)\}$
- **Delete list (DEL(a))**: (the fluents in) the negative literals in the action's effects
  - ex:  $\{At(p, from)\}$
- **Result of action a in state s**:  $RESULT(s,a) \stackrel{\text{def}}{=} (s \setminus DEL(a) \cup ADD(a))$ 
  - start from  $s$
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex:  $Fly(P_1, SFO, JFK) \implies$  remove  $At(P_1, SFO)$ , add  $At(P_1, JFK)$

# A Language for Planning: PDDL [cont.]

- **Precondition**: must hold to ensure the action can be executed
  - defines the states in which the action can be executed
  - action is **applicable** in state  $s$  if the preconditions are satisfied by  $s$
- **Effect**: represent the effects of the action on the world
  - defines the result of executing the action
- **Add list (ADD(a))**: (the fluents in) the positive literals in the action's effects
  - ex:  $\{At(p, to)\}$
- **Delete list (DEL(a))**: (the fluents in) the negative literals in the action's effects
  - ex:  $\{At(p, from)\}$
- **Result of action a in state s**:  $RESULT(s,a) \stackrel{\text{def}}{=} (s \setminus DEL(a) \cup ADD(a))$ 
  - start from  $s$
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex:  $Fly(P_1, SFO, JFK) \implies$  remove  $At(P_1, SFO)$ , add  $At(P_1, JFK)$

# A Language for Planning: PDDL [cont.]

- **Precondition**: must hold to ensure the action can be executed
  - defines the states in which the action can be executed
  - action is **applicable** in state  $s$  if the preconditions are satisfied by  $s$
- **Effect**: represent the effects of the action on the world
  - defines the result of executing the action
- **Add list (ADD(a))**: (the fluents in) the positive literals in the action's effects
  - ex:  $\{At(p, to)\}$
- **Delete list (DEL(a))**: (the fluents in) the negative literals in the action's effects
  - ex:  $\{At(p, from)\}$
- **Result of action  $a$  in state  $s$** :  $RESULT(s,a) \stackrel{\text{def}}{=} (s \setminus DEL(a) \cup ADD(a))$ 
  - start from  $s$
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex:  $Fly(P_1, SFO, JFK) \implies$  remove  $At(P_1, SFO)$ , add  $At(P_1, JFK)$



# A Language for Planning: PDDL [cont.]

- **Precondition**: must hold to ensure the action can be executed
  - defines the states in which the action can be executed
  - action is **applicable** in state  $s$  if the preconditions are satisfied by  $s$
- **Effect**: represent the effects of the action on the world
  - defines the result of executing the action
- **Add list (ADD(a))**: (the fluents in) the positive literals in the action's effects
  - ex:  $\{At(p, to)\}$
- **Delete list (DEL(a))**: (the fluents in) the negative literals in the action's effects
  - ex:  $\{At(p, from)\}$
- **Result of action a in state s**:  $RESULT(s,a) \stackrel{\text{def}}{=} (s \setminus DEL(a) \cup ADD(a))$ 
  - start from  $s$
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex:  $Fly(P_1, SFO, JFK) \implies$  remove  $At(P_1, SFO)$ , add  $At(P_1, JFK)$

# A Language for Planning: PDDL [cont.]

- **Precondition**: must hold to ensure the action can be executed
  - defines the states in which the action can be executed
  - action is **applicable** in state  $s$  if the preconditions are satisfied by  $s$
- **Effect**: represent the effects of the action on the world
  - defines the result of executing the action
- **Add list (ADD(a))**: (the fluents in) the positive literals in the action's effects
  - ex:  $\{At(p, to)\}$
- **Delete list (DEL(a))**: (the fluents in) the negative literals in the action's effects
  - ex:  $\{At(p, from)\}$
- **Result of action a in state s**:  $RESULT(s,a) \stackrel{\text{def}}{=} (s \setminus DEL(a) \cup ADD(a))$ 
  - start from  $s$
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex:  $Fly(P_1, SFO, JFK) \implies$  remove  $At(P_1, SFO)$ , add  $At(P_1, JFK)$

## PDDL: Example [cont.]

- Action schema:

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

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

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

- Action instantiation:

*Action*(Fly(*P*<sub>1</sub>, SFO, JFK),

*PRECOND* : *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  *At*(*P*<sub>1</sub>, SFO)

*EFFECT* :  $\neg$ *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *At*(*P*<sub>1</sub>, JFK)

- $s$  : *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

$\implies$   $s'$  : *At*(*P*<sub>1</sub>, JFK)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

Sometimes we want to **propositionalize** a PDDL problem:  
replace each action schema with a set of ground actions.

- Ex: ...*At\_P1\_SFO*  $\wedge$  *Plane\_P1*  $\wedge$  *Airport\_SFO*  $\wedge$  *Airport\_JFK*...

## PDDL: Example [cont.]

- Action schema:

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

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

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

- Action instantiation:

*Action*(Fly(*P*<sub>1</sub>, SFO, JFK),

*PRECOND* : *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  *At*(*P*<sub>1</sub>, SFO)

*EFFECT* :  $\neg$ *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *At*(*P*<sub>1</sub>, JFK)

- *s* : *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

$\Rightarrow$  *s'* : *At*(*P*<sub>1</sub>, JFK)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

Sometimes we want to **propositionalize** a PDDL problem:  
replace each action schema with a set of ground actions.

- Ex: ...*At*\_*P*<sub>1</sub>\_SFO  $\wedge$  *Plane*\_*P*<sub>1</sub>  $\wedge$  *Airport*\_SFO  $\wedge$  *Airport*\_JFK)...

## PDDL: Example [cont.]

- Action schema:

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

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

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

- Action instantiation:

*Action*(Fly(*P*<sub>1</sub>, SFO, JFK),

*PRECOND* : *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  *At*(*P*<sub>1</sub>, SFO)

*EFFECT* :  $\neg$ *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *At*(*P*<sub>1</sub>, JFK)

- $s$  : *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

$\Rightarrow$   $s'$  : *At*(*P*<sub>1</sub>, JFK)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

Sometimes we want to **propositionalize** a PDDL problem:  
replace each action schema with a set of ground actions.

- Ex: ...*At\_P1\_SFO*  $\wedge$  *Plane\_P1*  $\wedge$  *Airport\_SFO*  $\wedge$  *Airport\_JFK*...

## PDDL: Example [cont.]

- Action schema:

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

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

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

- Action instantiation:

*Action*(Fly(*P*<sub>1</sub>, SFO, JFK),

*PRECOND* : *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  *At*(*P*<sub>1</sub>, SFO)

*EFFECT* :  $\neg$ *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *At*(*P*<sub>1</sub>, JFK)

- *s* : *At*(*P*<sub>1</sub>, SFO)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

$\Rightarrow$  *s'* : *At*(*P*<sub>1</sub>, JFK)  $\wedge$  *Plane*(*P*<sub>1</sub>)  $\wedge$  *Airport*(SFO)  $\wedge$  *Airport*(JFK)  $\wedge$  ...

Sometimes we want to **propositionalize** a PDDL problem:  
replace each action schema with a set of ground actions.

- Ex: ...*At\_P1\_SFO*  $\wedge$  *Plane\_P1*  $\wedge$  *Airport\_SFO*  $\wedge$  *Airport\_JFK*...

# A Language for Planning: PDDL [cont.]

## Time in PDDL

- Fluents do not explicitly refer to time
- Times and states are **implicit** in the action schemata:
  - the precondition always refers to time  $t$
  - the effect to time  $t+1$ .

## PDDL Problem

- A set of action schemata defines a planning domain
- PDDL problem: a planning domain, an initial state and a goal
  - the initial state is a conjunction of ground atoms (positive literals)
    - closed-world assumption: any not-mentioned atoms are false
  - the goal is a conjunction of literals (positive or negative)
    - may contain variables, which are implicitly existentially quantified
    - a goal  $g$  may represent a set of states (the set of states entailing  $g$ )
- Ex: goal:  $At(p, SFO) \wedge Plane(p)$ :
  - variable "p" implicitly means "for some plane p"
  - the state  $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge \dots$  entails  $g$

# A Language for Planning: PDDL [cont.]

## Time in PDDL

- Fluents do not explicitly refer to time
- Times and states are **implicit** in the action schemata:
  - the precondition always refers to time  $t$
  - the effect to time  $t+1$ .

## PDDL Problem

- A set of action schemata defines a planning domain
- PDDL problem: a planning domain, an initial state and a goal
  - the initial state is a conjunction of ground atoms (positive literals)
    - closed-world assumption: any not-mentioned atoms are false
  - the goal is a conjunction of literals (positive or negative)
    - may contain variables, which are implicitly existentially quantified
    - a goal  $g$  may represent a set of states (the set of states entailing  $g$ )
- Ex: goal:  $At(p, SFO) \wedge Plane(p)$ :
  - variable "p" implicitly means "for some plane p"
  - the state  $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge \dots$  entails  $g$



# A Language for Planning: PDDL [cont.]

## Time in PDDL

- Fluents do not explicitly refer to time
- Times and states are **implicit** in the action schemata:
  - the precondition always refers to time  $t$
  - the effect to time  $t+1$ .

## PDDL Problem

- A set of action schemata defines a **planning domain**
- **PDDL problem**: a **planning domain**, an **initial state** and a **goal**
  - the **initial state** is a **conjunction of ground atoms** (positive literals)
    - closed-world assumption: any not-mentioned atoms are false
  - the **goal** is a **conjunction of literals (positive or negative)**
    - **may contain variables**, which are implicitly **existentially quantified**
    - a goal  $g$  may represent a **set of states** (the set of states entailing  $g$ )
- Ex: goal:  $At(p, SFO) \wedge Plane(p)$ :
  - variable “ $p$ ” implicitly means “for some plane  $p$ ”
  - the state  $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge \dots$  entails  $g$

# A Language for Planning: PDDL [cont.]

## Time in PDDL

- Fluents do not explicitly refer to time
- Times and states are **implicit** in the action schemata:
  - the precondition always refers to time  $t$
  - the effect to time  $t+1$ .

## PDDL Problem

- A set of action schemata defines a **planning domain**
- **PDDL problem**: a **planning domain**, an **initial state** and a **goal**
  - the **initial state** is a **conjunction of ground atoms** (positive literals)
    - closed-world assumption: any not-mentioned atoms are false
  - the **goal** is a **conjunction of literals (positive or negative)**
    - **may contain variables**, which are implicitly **existentially quantified**
    - a goal  $g$  may represent a **set of states** (the set of states entailing  $g$ )
- Ex: goal:  $At(p, SFO) \wedge Plane(p)$ :
  - variable “ $p$ ” implicitly means “for some plane  $p$ ”
  - the state  $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge \dots$  entails  $g$

# A Language for Planning: PDDL [cont.]

## Time in PDDL

- Fluents do not explicitly refer to time
- Times and states are **implicit** in the action schemata:
  - the precondition always refers to time  $t$
  - the effect to time  $t+1$ .

## PDDL Problem

- A set of action schemata defines a **planning domain**
- **PDDL problem**: a **planning domain**, an **initial state** and a **goal**
  - the **initial state** is a **conjunction of ground atoms** (positive literals)
    - closed-world assumption: any not-mentioned atoms are false
  - the **goal** is a **conjunction of literals (positive or negative)**
    - **may contain variables**, which are implicitly **existentially quantified**
    - a goal  $g$  may represent a **set of states** (the set of states entailing  $g$ )
- Ex: **goal**:  $At(p, SFO) \wedge Plane(p)$ :
  - variable “ $p$ ” implicitly means “for some plane  $p$ ”
  - the state  $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge \dots$  entails  $g$

# A Language for Planning: PDDL [cont.]

## Planning as a search problem

All components of a search problem

- an **initial state**
- an **ACTIONS** function
- a **RESULT** function
- and a **goal test**

## Example: Air Cargo Transport

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$   
 $\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$   
 $\wedge Airport(JFK) \wedge Airport(SFO))$

$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$

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

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

(© S. Russell & P. Norwig, AIMA)

One solution:  $[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$   
 $Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)]$

## Example: Air Cargo Transport

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$   
 $\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$   
 $\wedge Airport(JFK) \wedge Airport(SFO))$

$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$

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

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

(© S. Russell & P. Norwig, AIMA)

One solution:  $[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$   
 $Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)]$

## Example: 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:  
    EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$   
             $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

(© S. Russell & P. Norwig, AIMA)

(We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.)

One solution:  $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$

## Example: 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:  
    EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$   
             $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

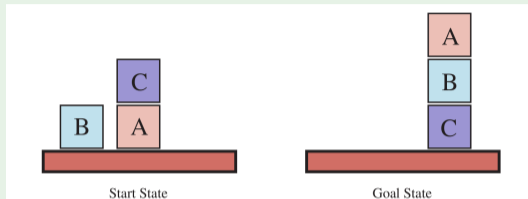
(© S. Russell & P. Norwig, AIMA)

(We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.)

One solution:  $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$



# Example: Blocks World

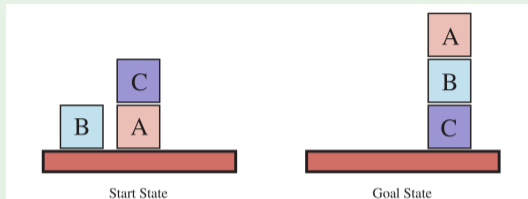


$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)$   
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C))$   
 $Goal(On(A, B) \wedge On(B, C))$   
 $Action(Move(b, x, y),$   
    PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$   
             $(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$   
    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:  $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$

(© S. Russell & P. Norwig, AIMA)

One solution:  $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$

# 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:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$   
             $(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$   
    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:  $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$

© S. Russell & P. Norwig, AIMA)

One solution:  $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics**
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics**
  - Forward and Backward Search**
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

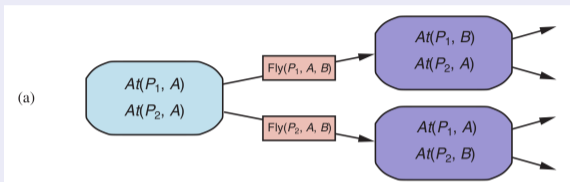
# Two Main Approaches

## (a) Forward search (aka progression search)

- start in the initial state
- use actions to search forward for a goal state

## (b) Backward search (aka regression search)

- start from goals
- use reverse actions to search forward for the initial state



(© S. Russell & P. Norwig, AIMA)

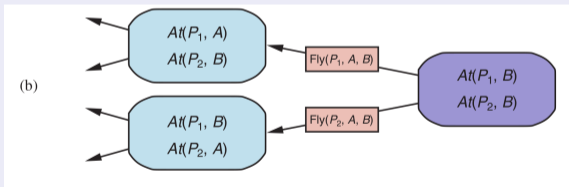
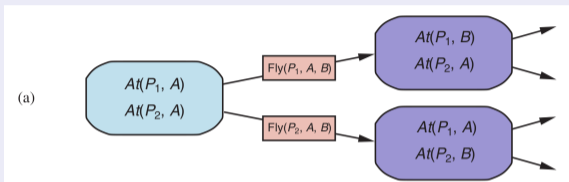
# Two Main Approaches

## (a) Forward search (aka progression search)

- start in the initial state
- use actions to search forward for a goal state

## (b) Backward search (aka regression search)

- start from goals
- use reverse actions to search forward for the initial state



(© S. Russell & P. Norwig, AIMA)

# Forward Search

- **Forward search** (aka **progression search**)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative

- Goal test: does the state satisfy the goal?
- Step cost: each action costs 1

⇒ We can use any of the search algorithms from Ch. 03, 04

- need keeping track of the actions used to reach the goal
- **Breadth-first** and **best-first**
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - **Require exponential memory** wrt. solution length! ⇒ **unpractical**
- **Depth-first search** and **greedy search**
  - **Sound**
  - **Not complete**
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - **Require linear memory** wrt. solution length

# Forward Search

- **Forward search** (aka **progression search**)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative
- **Goal test**: does the state satisfy the goal?
- **Step cost**: each action costs 1

⇒ We can use any of the search algorithms from Ch. 03, 04

- need keeping track of the actions used to reach the goal
- **Breadth-first** and **best-first**
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - **Require exponential memory** wrt. solution length! ⇒ **unpractical**
- **Depth-first search** and **greedy search**
  - **Sound**
  - **Not complete**
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - **Require linear memory** wrt. solution length



# Forward Search

- **Forward search** (aka **progression search**)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative
- **Goal test**: does the state satisfy the goal?
- **Step cost**: each action costs 1

⇒ We can use any of the search algorithms from Ch. 03, 04

- need keeping track of the actions used to reach the goal
- **Breadth-first** and **best-first**
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - **Require exponential memory** wrt. solution length! ⇒ **unpractical**
- **Depth-first search** and **greedy search**
  - **Sound**
  - **Not complete**
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - **Require linear memory** wrt. solution length

# Forward Search

- **Forward search** (aka **progression search**)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative
- Goal test: does the state satisfy the goal?
- Step cost: each action costs 1

⇒ We can use any of the search algorithms from Ch. 03, 04

- need keeping track of the actions used to reach the goal
- **Breadth-first** and **best-first**
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - **Require exponential memory** wrt. solution length! ⇒ **unpractical**
- **Depth-first search** and **greedy search**
  - **Sound**
  - **Not complete**
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - **Require linear memory** wrt. solution length

# Forward Search

- **Forward search** (aka **progression search**)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative
- Goal test: does the state satisfy the goal?
- Step cost: each action costs 1

⇒ We can use any of the search algorithms from Ch. 03, 04

- need keeping track of the actions used to reach the goal
- **Breadth-first** and **best-first**
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - **Require exponential memory** wrt. solution length! ⇒ **unpractical**
- **Depth-first search** and **greedy search**
  - **Sound**
  - **Not complete**
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - **Require linear memory** wrt. solution length

# Forward Search

- **Forward search** (aka **progression search**)
  - choose actions whose preconditions are satisfied
  - add positive effects, delete negative
- Goal test: does the state satisfy the goal?
- Step cost: each action costs 1

⇒ We can use any of the search algorithms from Ch. 03, 04

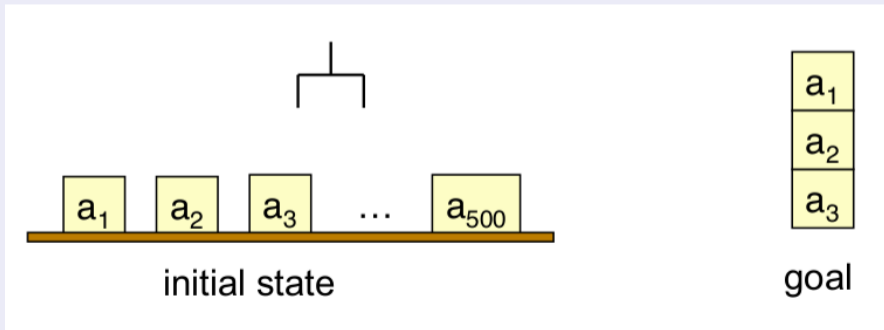
- need keeping track of the actions used to reach the goal
- **Breadth-first** and **best-first**
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - **Require exponential memory** wrt. solution length! ⇒ **unpractical**
- **Depth-first search** and **greedy search**
  - **Sound**
  - **Not complete**
    - may enter in infinite loops
    - (classical planning only): made complete by loop-checking
  - **Require linear memory** wrt. solution length

# Branching Factor of Forward Search

- Planning problems can have huge state spaces
- Forward search can have a very large branching factor
  - ex:  $pickup(a_1), pickup(a_2), \dots, pickup(a_{500})$

⇒ Forward-search can waste time trying lots of irrelevant actions

⇒ Need a good heuristic to guide the search

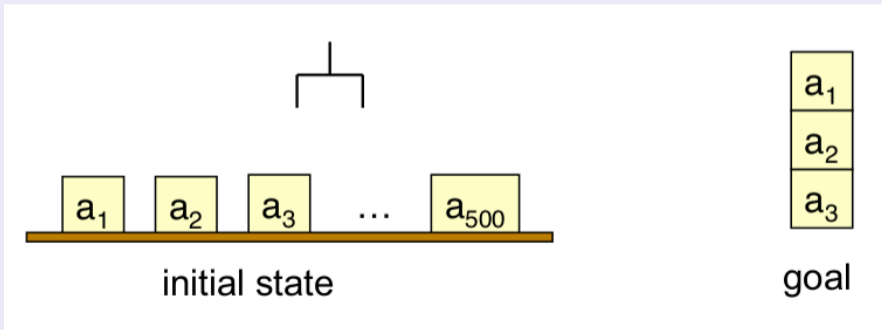


# Branching Factor of Forward Search

- Planning problems can have huge state spaces
- Forward search can have a very large branching factor
  - ex:  $\text{pickup}(a_1), \text{pickup}(a_2), \dots, \text{pickup}(a_{500})$

⇒ Forward-search can waste time trying lots of irrelevant actions

⇒ Need a good heuristic to guide the search

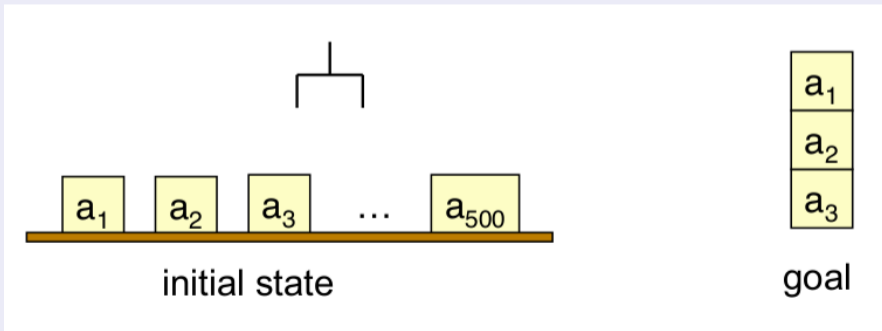


# Branching Factor of Forward Search

- Planning problems can have huge state spaces
- Forward search can have a very large branching factor
  - ex:  $\text{pickup}(a_1), \text{pickup}(a_2), \dots, \text{pickup}(a_{500})$

⇒ Forward-search can waste time trying lots of irrelevant actions

⇒ Need a good heuristic to guide the search

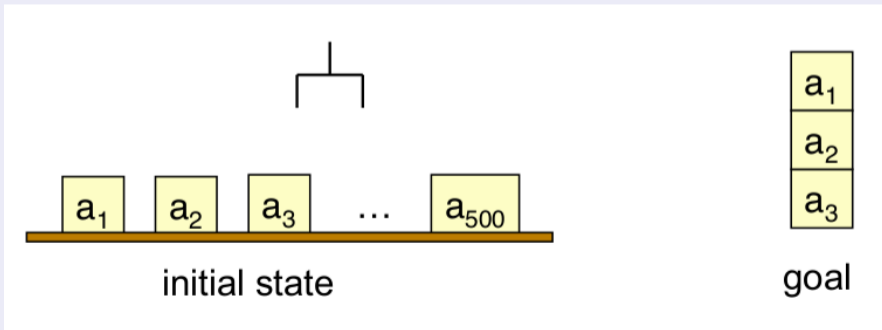


# Branching Factor of Forward Search

- Planning problems can have huge state spaces
- Forward search can have a very large branching factor
  - ex: *pickup*( $a_1$ ), *pickup*( $a_2$ ), ..., *pickup*( $a_{500}$ )

⇒ Forward-search can waste time trying lots of irrelevant actions

⇒ **Need a good heuristic to guide the search**





# Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal  $g'$  of ground goal  $g$  via ground action  $a$ :

$$Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$$

$$Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$$

- Note: Both  $g$  and  $g'$  represent many states
  - irrelevant ground atoms unassigned

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- Consider the ground action:

$Action(Unload(C_1, P_1, SFO),$

$PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$

$EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$

- This produces the sub-goal  $g'$ :

$In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$

- Both  $g'$  and  $g$  represent many states
  - e.g. truth value of  $In(C_3, P_2)$  irrelevant

# Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal  $g'$  of ground goal  $g$  via ground action  $a$ :

$$Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$$

$$Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$$

- Note: Both  $g$  and  $g'$  represent many states
  - irrelevant ground atoms unassigned

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- Consider the ground action:

$Action(Unload(C_1, P_1, SFO),$

$PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$

$EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$

- This produces the sub-goal  $g'$ :

$In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$

- Both  $g'$  and  $g$  represent many states
  - e.g. truth value of  $In(C_3, P_2)$  irrelevant

# Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal  $g'$  of ground goal  $g$  via ground action  $a$ :

$$Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$$

$$Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$$

- Note: Both  $g$  and  $g'$  represent many states
  - irrelevant ground atoms unassigned

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- Consider the ground action:

$Action(Unload(C_1, P_1, SFO),$

$PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$

$EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$

- This produces the sub-goal  $g'$ :

$In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$

- Both  $g'$  and  $g$  represent many states
  - e.g. truth value of  $In(C_3, P_2)$  irrelevant

# Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal  $g'$  of ground goal  $g$  via ground action  $a$ :

$$Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$$

$$Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$$

- Note: Both  $g$  and  $g'$  represent many states
  - irrelevant ground atoms unassigned

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- Consider the ground action:

$Action(Unload(C_1, P_1, SFO),$

$PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$

$EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$

- This produces the sub-goal  $g'$ :

$In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$

- Both  $g'$  and  $g$  represent many states
  - e.g. truth value of  $In(C_3, P_2)$  irrelevant

# Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal  $g'$  of ground goal  $g$  via ground action  $a$ :

$$Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$$

$$Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$$

- Note: Both  $g$  and  $g'$  represent many states
  - irrelevant ground atoms unassigned

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- Consider the ground action:

$Action(Unload(C_1, P_1, SFO),$

$PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$

$EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$

- This produces the sub-goal  $g'$ :

$In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$

- Both  $g'$  and  $g$  represent many states
  - e.g. truth value of  $In(C_3, P_2)$  irrelevant

# Backward Search (aka Regression or Relevant-States)

- Predecessor (sub)goal  $g'$  of ground goal  $g$  via ground action  $a$ :

$$Pos(g') \stackrel{\text{def}}{=} (Pos(g) \setminus Add(a)) \cup Pos(Precond(a))$$

$$Neg(g') \stackrel{\text{def}}{=} (Neg(g) \setminus Del(a)) \cup Neg(Precond(a))$$

- Note: Both  $g$  and  $g'$  represent many states
  - irrelevant ground atoms unassigned

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- Consider the ground action:

$Action(Unload(C_1, P_1, SFO),$

$PRECOND : In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO)$

$EFFECT : At(C_1, SFO) \wedge \neg In(C_1, P_1))$

- This produces the sub-goal  $g'$ :

$In(C_1, P_1) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO) \wedge At(C_2, JFK)$

- Both  $g'$  and  $g$  represent many states
  - e.g. truth value of  $In(C_3, P_2)$  irrelevant

## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations
- ⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes
- ⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$

## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations

⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes  
⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$



## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations

⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes  
⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$

## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations
    - ⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes
  - ⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$

## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations

⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes  
⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$

## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations
- ⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes
- ⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$

## Backward Search [cont.]

- Idea: deal with partially un-instantiated actions and states
  - avoid unnecessary instantiations
- ⇒ no need to produce a goal for every possible instantiation
- use the most general unifier ⇒ compute weakest precondition
- standardize action schemata first (rename vars into fresh ones)

- Consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$
- Consider the partially-instantiated action:  
 $Action(Unload(C_1, p', SFO),$   
 $PRECOND : In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$   
 $EFFECT : At(C_1, SFO) \wedge \neg In(C_1, p'))$
- This produces the sub-goal  $g'$ :  
 $In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO) \wedge At(C_2, JFK)$
- Represents states with all possible planes
- ⇒ no need to produce a subgoal for every plane  $P_1, P_2, P_3, \dots$

# Backward Search [cont.]

## Which action to choose?

- **Relevant action:** could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal  
(see AIMA book for formal definition)
- **Consistent action:** must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $fly(C_1, SFO, LAX)$  is relevant (positive effect)
  - $fly(C_2, SFO, LAX)$  is not relevant
  - $fly(C_1, SFO, LAX)$  is not consistent  $\rightarrow$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

# Backward Search [cont.]

## Which action to choose?

- **Relevant action:** could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action:** must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $At(C_1, SFO)$  is relevant (positive literal)
  - $At(C_2, JFK)$  is not relevant
  - $At(C_1, JFK)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action:** could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action:** must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

● Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

●  $fly(C_1, SFO, LAX)$  is relevant (positive effect)

●  $fly(C_2, LAX, JFK)$  is not relevant

●  $fly(C_1, SFO, JFK)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics



## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

● Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - ⇒ makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant
- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant
- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant
- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant
- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

# Backward Search [cont.]

## Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics

## Backward Search [cont.]

### Which action to choose?

- **Relevant action**: could be the last step in a plan for goal  $g$ 
  - at least one of the action's effects (positive or negative) must unify with an element of the goal (see AIMA book for formal definition)
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

- Ex: consider the goal  $At(C_1, SFO) \wedge At(C_2, JFK)$ 
  - $Action(Unload(C_1, p', SFO), \dots)$  is relevant (previous example)
  - $Action(Unload(C_3, p', SFO), \dots)$  is not relevant
  - $Action(Load(C_2, p', JFK), \dots)$  is not consistent  $\implies$  is not relevant

- + B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  - $\implies$  makes it harder to come up with good heuristics
- Most planners work with forward search plus heuristics



# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics**
  - Forward and Backward Search
  - Heuristics**
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Heuristics for (Forward-Search) Planning

## A\* for Planning

- Recall: A\* is a best-first algorithm which
  - uses an **evaluation function**  $f(s) = g(s) + h(s)$ ,
  - $g(s)$ : (exact) cost to reach  $s$
  - $h(s)$ : **admissible (optimistic) heuristics**  
(never overestimates the distance to the goal)
- A technique for admissible heuristics: **problem relaxation**  
⇒  $h(s)$ : the exact cost of a solution to the relaxed problem
- Forms of problem relaxation exploiting problem structure
  - Add arcs to the search graph ⇒ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) ⇒ reduce search space
    - ignore less-relevant fluents

# Heuristics for (Forward-Search) Planning

## A\* for Planning

- Recall: A\* is a best-first algorithm which
  - uses an **evaluation function**  $f(s) = g(s) + h(s)$ ,
  - $g(s)$ : (exact) cost to reach  $s$
  - $h(s)$ : **admissible (optimistic) heuristics**  
(never overestimates the distance to the goal)
- A technique for admissible heuristics: **problem relaxation**  
⇒  $h(s)$ : the exact cost of a solution to the relaxed problem
- Forms of problem relaxation exploiting problem structure
  - **Add arcs to the search graph** ⇒ make it easier to search
    - **ignore-preconditions** heuristics
    - **ignore-delete-lists** heuristics
  - **Clustering nodes** (aka **state abstraction**) ⇒ reduce search space
    - **ignore less-relevant fluents**

# Heuristics for (Forward-Search) Planning

## A\* for Planning

- Recall: A\* is a best-first algorithm which
  - uses an **evaluation function**  $f(s) = g(s) + h(s)$ ,
  - $g(s)$ : (exact) cost to reach  $s$
  - $h(s)$ : **admissible (optimistic) heuristics**  
(never overestimates the distance to the goal)
- A technique for admissible heuristics: **problem relaxation**  
 $\implies h(s)$ : the exact cost of a solution to the relaxed problem
- Forms of problem relaxation exploiting problem structure
  - **Add arcs to the search graph**  $\implies$  make it easier to search
    - **ignore-preconditions** heuristics
    - **ignore-delete-lists** heuristics
  - **Clustering nodes** (aka **state abstraction**)  $\implies$  reduce search space
    - **ignore less-relevant fluents**

# Heuristics for (Forward-Search) Planning

## A\* for Planning

- Recall: A\* is a best-first algorithm which
  - uses an **evaluation function**  $f(s) = g(s) + h(s)$ ,
  - $g(s)$ : (exact) cost to reach  $s$
  - $h(s)$ : **admissible (optimistic) heuristics**  
(never overestimates the distance to the goal)
- A technique for admissible heuristics: **problem relaxation**  
 $\implies h(s)$ : the exact cost of a solution to the relaxed problem
- Forms of problem relaxation exploiting problem structure
  - **Add arcs to the search graph**  $\implies$  make it easier to search
    - **ignore-preconditions** heuristics
    - **ignore-delete-lists** heuristics
  - **Clustering nodes** (aka **state abstraction**)  $\implies$  reduce search space
    - **ignore less-relevant fluents**

# Ignore (some) Preconditions Heuristics

- **Ignore all preconditions** drops all preconditions from actions
  - every action is applicable in any state
  - any single goal literal can be satisfied in one step (or there is no solution)
  - fast, but over-optimistic
- **Ignore some selected (less relevant) preconditions**
  - relevance based on heuristics or domain-dependent criteria

# Ignore (some) Preconditions Heuristics

- **Ignore all preconditions** drops all preconditions from actions
  - every action is applicable in any state
  - any single goal literal can be satisfied in one step (or there is no solution)
  - fast, but over-optimistic
- **Ignore some selected (less relevant) preconditions**
  - relevance based on heuristics or domain-dependent criteria

# Ignore-Preconditions Heuristics: Example

## Sliding tiles

Action(*Slide*( $t, s_1, s_2$ ),

*PRECOND* :  $\text{Tile}(t) \wedge \text{Blank}(s_2) \wedge \text{On}(t, s_1) \wedge \text{Adjacent}(s_1, s_2)$

*EFFECT* :  $\text{On}(t, s_2) \wedge \text{Blank}(s_1) \wedge \neg \text{On}(t, s_1) \wedge \neg \text{Blank}(s_2)$ )

- Remove the preconditions  $\text{Blank}(s_2) \wedge \text{Adjacent}(s_1, s_2)$

⇒ we get the **number-of-misplaced-tiles** heuristics

- Remove the precondition  $\text{Blank}(s_2)$

⇒ we get the **Manhattan-distance** heuristics

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State



# Ignore-Preconditions Heuristics: Example

## Sliding tiles

Action(*Slide*( $t, s_1, s_2$ ),

*PRECOND* :  $\text{Tile}(t) \wedge \text{Blank}(s_2) \wedge \text{On}(t, s_1) \wedge \text{Adjacent}(s_1, s_2)$

*EFFECT* :  $\text{On}(t, s_2) \wedge \text{Blank}(s_1) \wedge \neg \text{On}(t, s_1) \wedge \neg \text{Blank}(s_2)$ )

- Remove the preconditions  $\text{Blank}(s_2) \wedge \text{Adjacent}(s_1, s_2)$

⇒ we get the **number-of-misplaced-tiles** heuristics

- Remove the precondition  $\text{Blank}(s_2)$

⇒ we get the **Manhattan-distance** heuristics

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

# Ignore-Preconditions Heuristics: Example

## Sliding tiles

Action(*Slide*( $t, s_1, s_2$ ),

*PRECOND* :  $Tile(t) \wedge Blank(s_2) \wedge On(t, s_1) \wedge Adjacent(s_1, s_2)$

*EFFECT* :  $On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2)$ )

- Remove the preconditions  $Blank(s_2) \wedge Adjacent(s_1, s_2)$

⇒ we get the **number-of-misplaced-tiles** heuristics

- Remove the precondition  $Blank(s_2)$

⇒ we get the **Manhattan-distance** heuristics

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

# Ignore Delete-list Heuristics

- Assumption: goals & preconditions contain only positive literals
  - reasonable in many domains
- Idea: Remove the delete lists from all actions
  - No action will ever undo the effect of actions,  
⇒ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

# Ignore Delete-list Heuristics

- Assumption: goals & preconditions contain only positive literals
  - reasonable in many domains
- Idea: Remove the delete lists from all actions
  - No action will ever undo the effect of actions,  
⇒ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

# Ignore Delete-list Heuristics

- Assumption: goals & preconditions contain only positive literals
  - reasonable in many domains
- Idea: Remove the delete lists from all actions
  - No action will ever undo the effect of actions,  
⇒ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

# Ignore Delete-list Heuristics

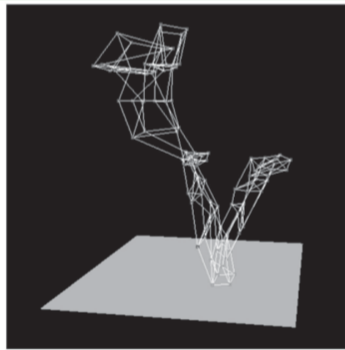
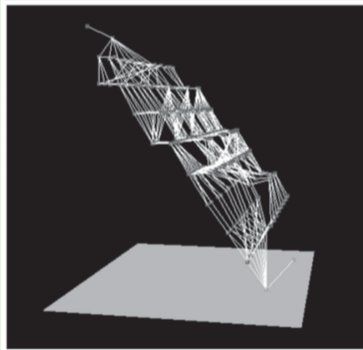
- Assumption: goals & preconditions contain only positive literals
  - reasonable in many domains
- Idea: Remove the delete lists from all actions
  - No action will ever undo the effect of actions,  
⇒ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

# Ignore Delete-list Heuristics: Example (Hoffmann'05)

- Planning state spaces with ignore-delete-lists heuristic
  - height above the bottom plane is the heuristic score of a state
  - states on the bottom plane are goals

⇒ No local minima, non dead-ends, non backtracking

⇒ Search for the goal is straightforward for hill-climbing

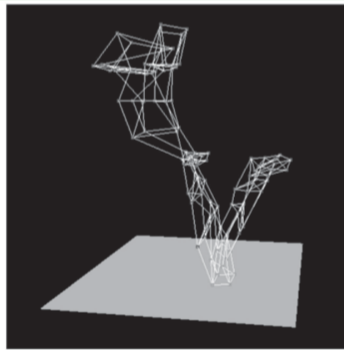
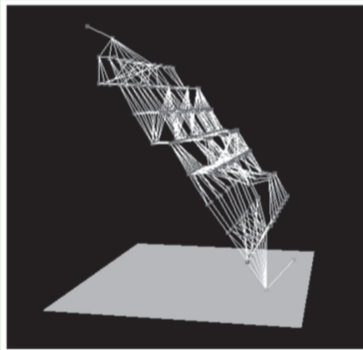


# Ignore Delete-list Heuristics: Example (Hoffmann'05)

- Planning state spaces with ignore-delete-lists heuristic
  - height above the bottom plane is the heuristic score of a state
  - states on the bottom plane are goals

⇒ No local minima, non dead-ends, non backtracking

⇒ Search for the goal is straightforward for hill-climbing





# State Abstractions

- Many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation
  - drastically reduces the number of states
- Common strategy: ignore some (less-relevant) fluents
  - drop  $k$  fluents  $\implies$  reduce search space by  $2^k$  factors
  - relevance based on (heuristic) evaluation or domain knowledge

- Air cargo problem: 10 airports, 50 planes, 200 pieces of cargo  
 $\implies 10^{50} \cdot (50 + 10)^{200} \approx 10^{405}$  states (\*)
- Consider particular problem in that domain
  - all packages are at 5 airports
  - all packages at a given airport have the same destination
- Abstraction: drop all "At" fluents except for these involving one plane and one package at each of the the 5 airports  
 $\implies 10^5 \cdot (5 + 10)^5 \approx 10^{11}$  states (\*)
  - abstract solution shorter than ground solutions  $\implies$  admissible
  - abstract solution easy to extend: add Load and Unload actions

(\*) wrong in AIMA III Ed, corrected in later editions

# State Abstractions

- Many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation
  - drastically reduces the number of states
- Common strategy: **ignore some (less-relevant) fluents**
  - drop  $k$  fluents  $\implies$  reduce search space by  $2^k$  factors
  - relevance based on (heuristic) evaluation or domain knowledge

- Air cargo problem: 10 airports, 50 planes, 200 pieces of cargo  
 $\implies 10^{50} \cdot (50 + 10)^{200} \approx 10^{405}$  states (\*)
- Consider particular problem in that domain
  - all packages are at 5 airports
  - all packages at a given airport have the same destination
- Abstraction: drop all "At" fluents except for these involving one plane and one package at each of the the 5 airports  
 $\implies 10^5 \cdot (5 + 10)^5 \approx 10^{11}$  states (\*)
  - abstract solution shorter than ground solutions  $\implies$  admissible
  - abstract solution easy to extend: add Load and Unload actions

(\*) wrong in AIMA III Ed, corrected in later editions

# State Abstractions

- Many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation
  - drastically reduces the number of states
- Common strategy: **ignore some (less-relevant) fluents**
  - drop  $k$  fluents  $\implies$  reduce search space by  $2^k$  factors
  - relevance based on (heuristic) evaluation or domain knowledge

- Air cargo problem: 10 airports, 50 planes, 200 pieces of cargo

$$\implies 10^{50} \cdot (50 + 10)^{200} \approx 10^{405} \text{ states (*)}$$

- Consider particular problem in that domain

- all packages are at 5 airports
- all packages at a given airport have the same destination

- Abstraction: drop all "At" fluents except for these involving one plane and one package at each of the the 5 airports

$$\implies 10^5 \cdot (5 + 10)^5 \approx 10^{11} \text{ states (*)}$$

- abstract solution shorter than ground solutions  $\implies$  admissible
- abstract solution easy to extend: add Load and Unload actions

(\*) wrong in AIMA III Ed, corrected in later editions

# State Abstractions

- Many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation
  - drastically reduces the number of states
- Common strategy: **ignore some (less-relevant) fluents**
  - drop  $k$  fluents  $\implies$  reduce search space by  $2^k$  factors
  - relevance based on (heuristic) evaluation or domain knowledge

- Air cargo problem: 10 airports, 50 planes, 200 pieces of cargo

$$\implies 10^{50} \cdot (50 + 10)^{200} \approx 10^{405} \text{ states (*)}$$

- Consider particular problem in that domain

- all packages are at 5 airports
- all packages at a given airport have the same destination

- Abstraction: drop all "At" fluents except for these involving one plane and one package at each of the the 5 airports

$$\implies 10^5 \cdot (5 + 10)^5 \approx 10^{11} \text{ states (*)}$$

- abstract solution shorter than ground solutions  $\implies$  admissible
- abstract solution easy to extend: add Load and Unload actions

(\*) wrong in AIMA III Ed, corrected in later editions

# State Abstractions

- Many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation
  - drastically reduces the number of states
- Common strategy: **ignore some (less-relevant) fluents**
  - drop  $k$  fluents  $\implies$  reduce search space by  $2^k$  factors
  - relevance based on (heuristic) evaluation or domain knowledge

- Air cargo problem: 10 airports, 50 planes, 200 pieces of cargo

$$\implies 10^{50} \cdot (50 + 10)^{200} \approx 10^{405} \text{ states (*)}$$

- Consider particular problem in that domain

- all packages are at 5 airports
- all packages at a given airport have the same destination

- Abstraction: drop all “At” fluents except for these involving one plane and one package at each of the the 5 airports

$$\implies 10^5 \cdot (5 + 10)^5 \approx 10^{11} \text{ states (*)}$$

- abstract solution shorter than ground solutions  $\implies$  admissible
- abstract solution easy to extend: add Load and Unload actions

(\*) wrong in AIMA III Ed, corrected in later editions

# Other Strategies for Planning

## Other strategies to define heuristics

- Problem decomposition
  - “divide & conquer” problem into subproblem
  - solve subproblems independently
- Using a data structure called “**planning graph**” (next section)

# Other Strategies for Planning

## Other strategies to define heuristics

- Problem decomposition
  - “divide & conquer” problem into subproblem
  - solve subproblems independently
- Using a data structure called “**planning graph**” (next section)

# Other Strategies for Planning

## Other strategies to define heuristics

- Problem decomposition
  - “divide & conquer” problem into subproblem
  - solve subproblems independently
- Using a data structure called “**planning graph**” (next section)



# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan**
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan**
  - Planning Graphs**
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

## Planning Graph

- A data structure which is a rich source of information:
  - can be used to give **better heuristic estimates  $h(s)$**
  - can drive an algorithm called **Graphplan**
- A **polynomial-size over-approximation to the (exponential) search tree**
  - can be constructed very quickly
- cannot answer definitively if goal  $g$  is reachable from initial state
- + may discover that the goal is not reachable
- + can estimate the most-optimistic step # to reach  $g$ 
  - ⇒ it can be used to derive an admissible heuristic  $h(s)$

## Planning Graph

- A data structure which is a rich source of information:
    - can be used to give **better heuristic estimates  $h(s)$**
    - can drive an algorithm called **Graphplan**
  - A **polynomial-size over-approximation to the (exponential) search tree**
    - can be constructed very quickly
- cannot answer definitively if goal  $g$  is reachable from initial state
- + may discover that the goal is not reachable
- + can estimate the most-optimistic step # to reach  $g$
- ⇒ it can be used to derive an admissible heuristic  $h(s)$

## Planning Graph

- A data structure which is a rich source of information:
  - can be used to give **better heuristic estimates  $h(s)$**
  - can drive an algorithm called **Graphplan**
- A **polynomial-size over-approximation to the (exponential) search tree**
  - can be constructed very quickly
- cannot answer definitively if goal  $g$  is reachable from initial state
- + may discover that the goal is not reachable
- + can estimate the most-optimistic step # to reach  $g$ 
  - ⇒ it can be used to derive an admissible heuristic  $h(s)$

## Planning Graph

- A data structure which is a rich source of information:
  - can be used to give **better heuristic estimates  $h(s)$**
  - can drive an algorithm called **Graphplan**
- A **polynomial-size over-approximation to the (exponential) search tree**
  - can be constructed very quickly
- cannot answer definitively if goal  $g$  is reachable from initial state
- + may discover that the goal is not reachable
- + can estimate the most-optimistic step # to reach  $g$ 
  - ⇒ it can be used to derive an admissible heuristic  $h(s)$

## Planning Graph

- A data structure which is a rich source of information:
  - can be used to give **better heuristic estimates  $h(s)$**
  - can drive an algorithm called **Graphplan**
- A **polynomial-size over-approximation to the (exponential) search tree**
  - can be constructed very quickly
- cannot answer definitively if goal  $g$  is reachable from initial state
- + may discover that the goal is not reachable
- + can estimate the most-optimistic step # to reach  $g$ 
  - $\implies$  it can be used to derive an admissible heuristic  $h(s)$

# Planning Graph: Definition

- A directed graph, built **forward** and organized into **levels**
  - **level  $S_0$** : contain each ground fluent that holds in the initial state
  - **level  $A_0$** : contains each ground action with preconditions in  $S_0$  (i.e. applicable in  $S_0$ )
  - ...
  - **level  $A_i$** : contains all ground actions with preconditions in  $S_i$
  - **level  $S_{i+1}$** : all the effects of all the actions in  $A_i$ 
    - each  $S_i$  may contain both  $P_j$  and  $\neg P_j$

until  $S_N = S_{N-1}$  (“leveled off”).

- Contains **persistence actions** (aka **maintenance actions, no-ops**)
  - say that a literal  $l$  persists if no action negates it
- **Mutual exclusion links (mutex)** connect
  - incompatible pairs of actions
  - incompatible pairs of literals

Deals with ground states and actions only



# Planning Graph: Definition

- A directed graph, built **forward** and organized into **levels**
  - **level  $S_0$** : contain each ground fluent that holds in the initial state
  - **level  $A_0$** : contains each ground action with preconditions in  $S_0$  (i.e. applicable in  $S_0$ )
  - ...
  - **level  $A_i$** : contains all ground actions with preconditions in  $S_i$
  - **level  $S_{i+1}$** : all the effects of all the actions in  $A_i$ 
    - each  $S_i$  may contain both  $P_j$  and  $\neg P_j$

until  $S_N = S_{N-1}$  (“leveled off”).

- Contains **persistence actions** (aka **maintenance actions**, **no-ops**)
  - say that a literal  $l$  persists if no action negates it
- **Mutual exclusion links (mutex)** connect
  - incompatible pairs of actions
  - incompatible pairs of literals

Deals with ground states and actions only

# Planning Graph: Definition

- A directed graph, built **forward** and organized into **levels**
  - **level  $S_0$** : contain each ground fluent that holds in the initial state
  - **level  $A_0$** : contains each ground action with preconditions in  $S_0$  (i.e. applicable in  $S_0$ )
  - ...
  - **level  $A_i$** : contains all ground actions with preconditions in  $S_i$
  - **level  $S_{i+1}$** : all the effects of all the actions in  $A_i$ 
    - each  $S_i$  may contain both  $P_j$  and  $\neg P_j$

until  $S_N = S_{N-1}$  (“leveled off”).

- Contains **persistence actions** (aka **maintenance actions**, **no-ops**)
  - say that a literal  $l$  persists if no action negates it
- **Mutual exclusion links (mutex)** connect
  - incompatible pairs of actions
  - incompatible pairs of literals

Deals with ground states and actions only

# Planning Graph: Definition

- A directed graph, built **forward** and organized into **levels**
  - **level  $S_0$** : contain each ground fluent that holds in the initial state
  - **level  $A_0$** : contains each ground action with preconditions in  $S_0$  (i.e. applicable in  $S_0$ )
  - ...
  - **level  $A_i$** : contains all ground actions with preconditions in  $S_i$
  - **level  $S_{i+1}$** : all the effects of all the actions in  $A_i$ 
    - each  $S_i$  may contain both  $P_j$  and  $\neg P_j$

until  $S_N = S_{N-1}$  (“leveled off”).

- Contains **persistence actions** (aka **maintenance actions**, **no-ops**)
  - say that a literal  $l$  persists if no action negates it
- **Mutual exclusion links (mutex)** connect
  - incompatible pairs of actions
  - incompatible pairs of literals

Deals with ground states and actions only

# Planning Graph: 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)*

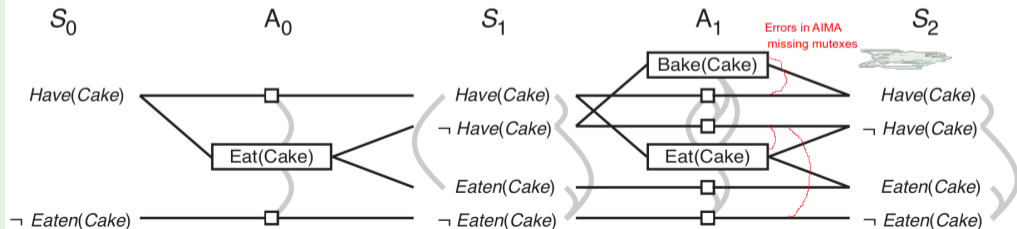
*You would like to eat your cake and still have a cake.  
Fortunately, you can bake a new one.*

Rectangles indicate actions

Small squares persistence actions (**no-ops**)

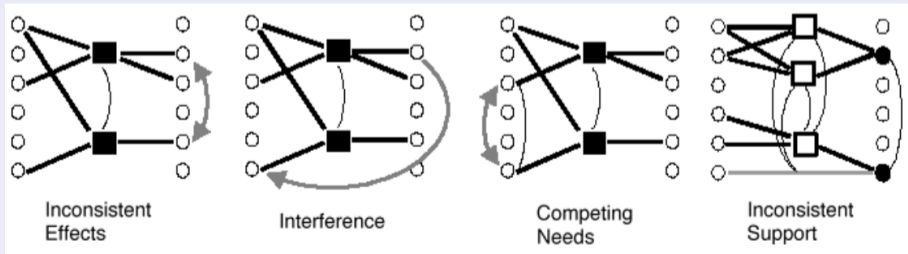
Straight lines indicate preconditions  
and effects

Mutex links are shown as curved gray lines



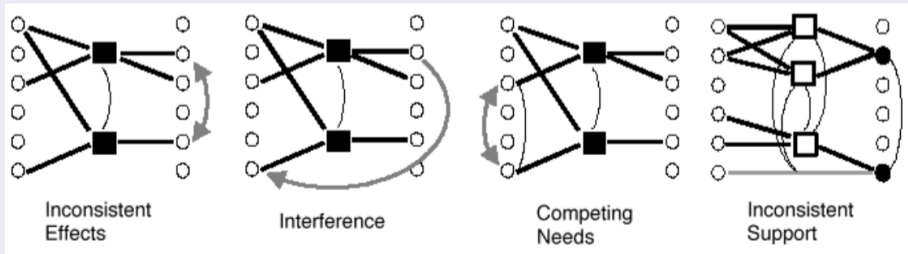
# Mutex Computation

- Two **actions** at the same action-level have a mutex relation if
  - **Inconsistent effects**: an effect of one negates an effect of the other
  - **Interference**: one deletes a precondition of the other
  - **Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions
- Otherwise they don't interfere with each other  
⇒ both may appear in a solution plan
- Two **literals** at the same state-level have a mutex relation if
  - **inconsistent support**: one is the negation of the other
  - all ways of achieving them (including no-ops) are pairwise mutex



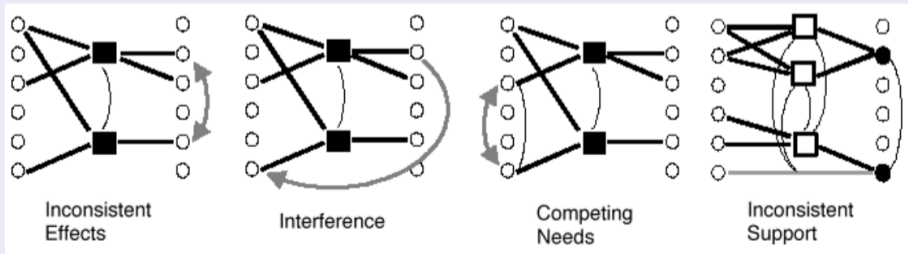
# Mutex Computation

- Two **actions** at the same action-level have a mutex relation if
  - **Inconsistent effects**: an effect of one negates an effect of the other
  - **Interference**: one deletes a precondition of the other
  - **Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions
- Otherwise they don't interfere with each other  
⇒ both may appear in a solution plan
- Two **literals** at the same state-level have a mutex relation if
  - **inconsistent support**: one is the negation of the other
  - all ways of achieving them (including no-ops) are pairwise mutex



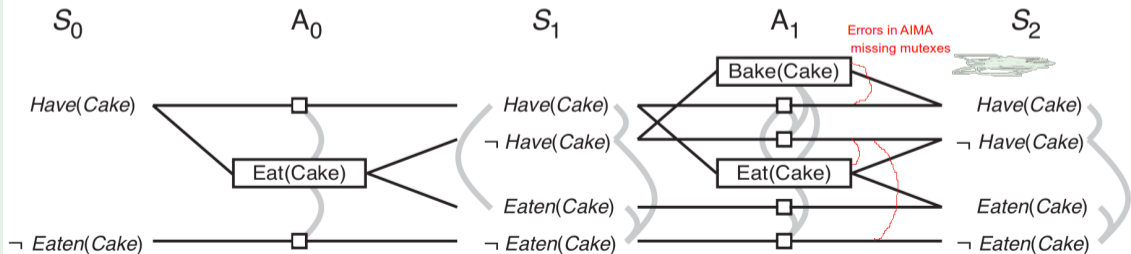
# Mutex Computation

- Two **actions** at the same action-level have a mutex relation if
  - **Inconsistent effects**: an effect of one negates an effect of the other
  - **Interference**: one deletes a precondition of the other
  - **Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions
- Otherwise they don't interfere with each other  
⇒ both may appear in a solution plan
- Two **literals** at the same state-level have a mutex relation if
  - **inconsistent support**: one is the negation of the other
  - all ways of achieving them (including no-ops) are pairwise mutex



# Mutex Computation: Example

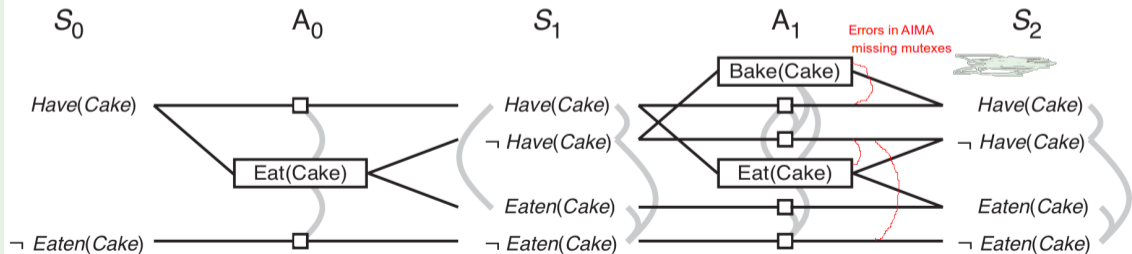
- Two **actions** at the same action-level have a mutex relation if
  - **Inconsistent effects**: an effect of one negates an effect of the other  
ex: persistence of  $Have(Cake)$ ,  $Eat(Cake)$  have competing effects  
ex:  $Bake(Cake)$ ,  $Eat(Cake)$  have competing effects
  - **Interference**: one deletes a precondition of the other  
ex:  $Eat(Cake)$  interferes with the persistence of  $Have(Cake)$
  - **Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions  
ex:  $Bake(Cake)$  and  $Eat(Cake)$





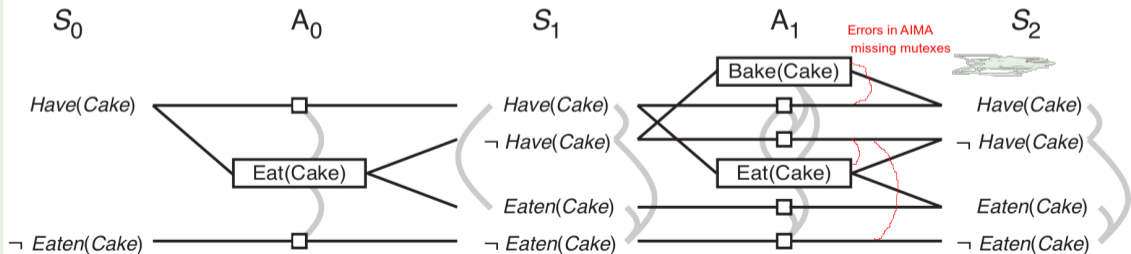
# Mutex Computation: Example

- Two **actions** at the same action-level have a mutex relation if
  - **Inconsistent effects**: an effect of one negates an effect of the other  
ex: *persistence of Have(Cake), Eat(Cake)* have competing effects  
ex: *Bake(Cake), Eat(Cake)* have competing effects
  - **Interference**: one deletes a precondition of the other  
ex: *Eat(Cake)* interferes with the persistence of *Have(Cake)*
  - **Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions  
ex: *Bake(Cake)* and *Eat(Cake)*



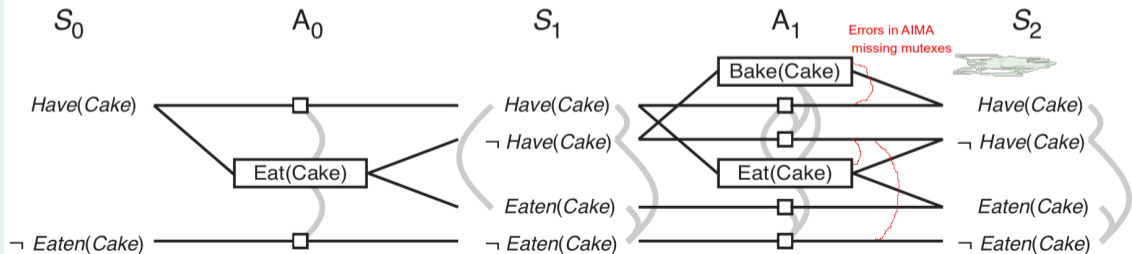
# Mutex Computation: Example

- Two **actions** at the same action-level have a mutex relation if
  - Inconsistent effects**: an effect of one negates an effect of the other  
ex: *persistence of Have(Cake), Eat(Cake)* have competing effects  
ex: *Bake(Cake), Eat(Cake)* have competing effects
  - Interference**: one deletes a precondition of the other  
ex: *Eat(Cake)* interferes with the persistence of *Have(Cake)*
  - Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions  
ex: *Bake(Cake)* and *Eat(Cake)*



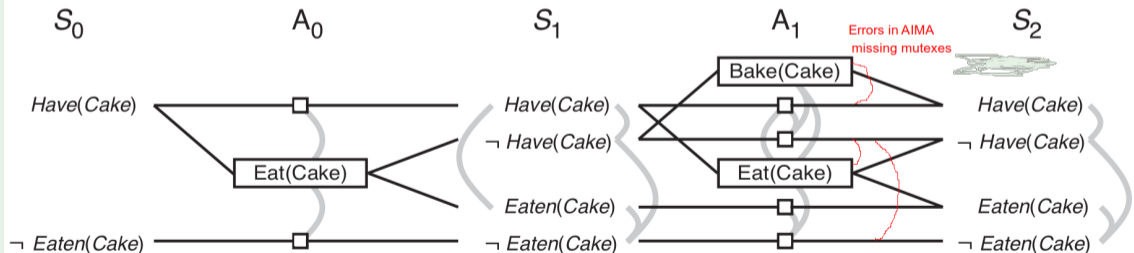
# Mutex Computation: Example

- Two **actions** at the same action-level have a mutex relation if
  - **Inconsistent effects**: an effect of one negates an effect of the other  
ex: *persistence of Have(Cake), Eat(Cake)* have competing effects  
ex: *Bake(Cake), Eat(Cake)* have competing effects
  - **Interference**: one deletes a precondition of the other  
ex: *Eat(Cake)* interferes with the persistence of *Have(Cake)*
  - **Inconsistent preconditions** (aka **competing needs**): they have mutually exclusive preconditions  
ex: *Bake(Cake)* and *Eat(Cake)*



# Mutex Computation: Example [cont.]

- Two **literals** at the same state-level have a mutex relation if
  - **inconsistent support**: one is the negation of the other  
ex.:  $Have(Cake)$ ,  $\neg Have(Cake)$
  - all ways of achieving them are pairwise mutex  
ex.:  $(S_1)$ :  $Have(Cake)$  in mutex with  $Eaten(Cake)$   
because persistence of  $Have(Cake)$ ,  $Eat(Cake)$  are mutex



(© S. Russell & P. Norwig, AIMA)

# Building of the Planning Graph

## Create initial layer $S_0$ :

- 1 insert into  $S_0$  all literals in the initial state

Repeat for increasing values of  $i = 0, 1, 2, \dots$ :

## Create action layer $A_i$ :

- 1 for each action schema, for each way to unify its preconditions to **non-mutually exclusive** literals in  $S_i$ , enter an action node into  $A_i$
- 2 for every literal in  $S_i$ , enter a no-op action node into  $A_i$
- 3 add mutexes between the newly-constructed action nodes

## Create state layer $S_{i+1}$ :

- 1 for each action node  $a$  in  $A_i$ ,
  - add to  $S_{i+1}$  the fluents in his Add list, linking them to  $a$
  - add to  $S_{i+1}$  the negated fluents in his Del list, linking them to  $a$
- 2 for every "no-op" action node  $a$  in  $A_i$ ,
  - add the corresponding literal to  $S_{i+1}$
  - link it to  $a$
- 3 add mutexes between literal nodes in  $S_{i+1}$

... until  $S_{i+1} = S_i$  (aka "graph leveled off") or **bound reached** (if any)

# Building of the Planning Graph

## Create initial layer $S_0$ :

- 1 insert into  $S_0$  all literals in the initial state

## Repeat for increasing values of $i = 0, 1, 2, \dots$ :

### Create action layer $A_i$ :

- 1 for each action schema, for each way to unify its preconditions to **non-mutually exclusive** literals in  $S_i$ , enter an action node into  $A_i$
- 2 for every literal in  $S_i$ , enter a no-op action node into  $A_i$
- 3 add mutexes between the newly-constructed action nodes

### Create state layer $S_{i+1}$ :

- 1 for each action node  $a$  in  $A_i$ ,
  - add to  $S_{i+1}$  the fluents in his Add list, linking them to  $a$
  - add to  $S_{i+1}$  the negated fluents in his Del list, linking them to  $a$
- 2 for every "no-op" action node  $a$  in  $A_i$ ,
  - add the corresponding literal to  $S_{i+1}$
  - link it to  $a$
- 3 add mutexes between literal nodes in  $S_{i+1}$

... until  $S_{i+1} = S_i$  (aka "graph leveled off") or **bound reached** (if any)

# Building of the Planning Graph

## Create initial layer $S_0$ :

- 1 insert into  $S_0$  all literals in the initial state

## Repeat for increasing values of $i = 0, 1, 2, \dots$ :

### Create action layer $A_i$ :

- 1 for each action schema, for each way to unify its preconditions to **non-mutually exclusive** literals in  $S_i$ , enter an action node into  $A_i$
- 2 for every literal in  $S_i$ , enter a no-op action node into  $A_i$
- 3 add mutexes between the newly-constructed action nodes

### Create state layer $S_{i+1}$ :

- 1 for each action node  $a$  in  $A_i$ ,
  - add to  $S_{i+1}$  the fluents in his Add list, linking them to  $a$
  - add to  $S_{i+1}$  the negated fluents in his Del list, linking them to  $a$
- 2 for every "no-op" action node  $a$  in  $A_i$ ,
  - add the corresponding literal to  $S_{i+1}$
  - link it to  $a$
- 3 add mutexes between literal nodes in  $S_{i+1}$

... until  $S_{i+1} = S_i$  (aka "graph leveled off") or bound reached (if any)

# Building of the Planning Graph

## Create initial layer $S_0$ :

- 1 insert into  $S_0$  all literals in the initial state

## Repeat for increasing values of $i = 0, 1, 2, \dots$ :

### Create action layer $A_i$ :

- 1 for each action schema, for each way to unify its preconditions to **non-mutually exclusive** literals in  $S_i$ , enter an action node into  $A_i$
- 2 for every literal in  $S_i$ , enter a no-op action node into  $A_i$
- 3 add mutexes between the newly-constructed action nodes

### Create state layer $S_{i+1}$ :

- 1 for each action node  $a$  in  $A_i$ ,
  - add to  $S_{i+1}$  the fluents in his Add list, linking them to  $a$
  - add to  $S_{i+1}$  the negated fluents in his Del list, linking them to  $a$
- 2 for every "no-op" action node  $a$  in  $A_i$ ,
  - add the corresponding literal to  $S_{i+1}$
  - link it to  $a$
- 3 add mutexes between literal nodes in  $S_{i+1}$

... until  $S_{i+1} = S_i$  (aka "graph leveled off") or bound reached (if any)



# Building of the Planning Graph

## Create initial layer $S_0$ :

- 1 insert into  $S_0$  all literals in the initial state

Repeat for increasing values of  $i = 0, 1, 2, \dots$ :

## Create action layer $A_i$ :

- 1 for each action schema, for each way to unify its preconditions to **non-mutually exclusive** literals in  $S_i$ , enter an action node into  $A_i$
- 2 for every literal in  $S_i$ , enter a no-op action node into  $A_i$
- 3 add mutexes between the newly-constructed action nodes

## Create state layer $S_{i+1}$ :

- 1 for each action node  $a$  in  $A_i$ ,
  - add to  $S_{i+1}$  the fluents in his Add list, linking them to  $a$
  - add to  $S_{i+1}$  the negated fluents in his Del list, linking them to  $a$
- 2 for every "no-op" action node  $a$  in  $A_i$ ,
  - add the corresponding literal to  $S_{i+1}$
  - link it to  $a$
- 3 add mutexes between literal nodes in  $S_{i+1}$

... until  $S_{i+1} = S_i$  (aka "graph leveled off") or bound reached (if any)

# Planning Graphs: Properties

- **Literals and actions increase monotonically and are finite**  
⇒ we eventually reach a level where they stabilize
  - **Mutexes decrease monotonically** (and cannot become less than zero)  
⇒ they too eventually must level off
- ⇒ When we reach this stable state, **if one of the goal literals is missing or is mutex with another goal literal, then it will remain so**  
⇒ we can stop

# Planning Graphs: Complexity

- A planning graph is polynomial in the size of the problem:
  - a graph with  $n$  levels,  $a$  actions,  $l$  literals, has size  $O(n(a + l)^2)$
  - time complexity is also  $O(n(a + l)^2)$

⇒ The process of constructing the planning graph is very fast

- does not require choosing among actions

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan**
  - Planning Graphs
  - Heuristics Driven by Planning Graphs**
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving

# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_i$  represents a set of possible belief states
  - two literals connected by a mutex belong to different belief state
- A literal not appearing in the final level of the graph cannot be achieved by any plan
  - ⇒ if a goal literal is not in the final level, the problem is unsolvable
- The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan
  - $j$  is called the level cost of literal  $l$
- the level cost of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )
  - this estimate is admissible
  - ex: from  $s_0$  Have(cake) has cost 0 and Eaten(cake) has cost 1
- Planning graph admits several actions per level
  - ⇒ inaccurate estimate
- **Serialization**: enforcing only one action per level (adding mutex)
  - ⇒ better estimate

# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_j$  represents **a set of possible belief states**
  - two literals connected by a mutex belong to different belief state
- A literal not appearing in the final level of the graph cannot be achieved by any plan
  - ⇒ if a goal literal is not in the final level, the problem is unsolvable
- The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan
  - $j$  is called the **level cost** of literal  $l$
- the **level cost** of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )
  - this estimate is **admissible**
  - ex: from  $s_0$  Have(cake) has cost 0 and Eaten(cake) has cost 1
- Planning graph admits several actions per level
  - ⇒ inaccurate estimate
- **Serialization**: enforcing only one action per level (adding mutex)
  - ⇒ better estimate

# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_j$  represents **a set of possible belief states**
  - two literals connected by a mutex belong to different belief state
- **A literal not appearing in the final level of the graph cannot be achieved by any plan**  
⇒ if a goal literal is not in the final level, the problem is unsolvable
- The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan
  - $j$  is called the **level cost** of literal  $l$
- the **level cost** of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )
  - this estimate is **admissible**
  - ex: from  $s_0$  Have(cake) has cost 0 and Eaten(cake) has cost 1
- Planning graph admits several actions per level  
⇒ inaccurate estimate
- **Serialization**: enforcing only one action per level (adding mutex)  
⇒ better estimate

# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_j$  represents **a set of possible belief states**
  - two literals connected by a mutex belong to different belief state
- **A literal not appearing in the final level of the graph cannot be achieved by any plan**  
⇒ if a goal literal is not in the final level, the problem is unsolvable
- **The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan**
  - $j$  is called the **level cost** of literal  $l$
- the **level cost** of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )
  - this estimate is **admissible**
  - ex: from  $s_0$  Have(cake) has cost 0 and Eaten(cake) has cost 1
- Planning graph admits several actions per level  
⇒ inaccurate estimate
- **Serialization**: enforcing only one action per level (adding mutex)  
⇒ better estimate



# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_j$  represents a set of possible belief states
  - two literals connected by a mutex belong to different belief state
- A literal not appearing in the final level of the graph cannot be achieved by any plan
  - ⇒ if a goal literal is not in the final level, the problem is unsolvable
- The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan
  - $j$  is called the level cost of literal  $l$
- the level cost of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )
  - this estimate is admissible
  - ex: from  $s_0$  Have(cake) has cost 0 and Eaten(cake) has cost 1
- Planning graph admits several actions per level
  - ⇒ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)
  - ⇒ better estimate

# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_j$  represents **a set of possible belief states**
  - two literals connected by a mutex belong to different belief state
- **A literal not appearing in the final level of the graph cannot be achieved by any plan**
  - ⇒ if a goal literal is not in the final level, the problem is unsolvable
- **The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan**
  - $j$  is called the **level cost** of literal  $l$
- **the level cost of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )**
  - this estimate **is admissible**
  - ex: from  $s_0$  **Have(cake)** has cost 0 and **Eaten(cake)** has cost 1
- **Planning graph admits several actions per level**
  - ⇒ inaccurate estimate
- **Serialization**: enforcing only one action per level (adding mutex)
  - ⇒ better estimate

# Planning Graphs for Heuristic Estimation

## Information provided by Planning Graphs

- Each level  $S_j$  represents **a set of possible belief states**
  - two literals connected by a mutex belong to different belief state
- **A literal not appearing in the final level of the graph cannot be achieved by any plan**  
⇒ if a goal literal is not in the final level, the problem is unsolvable
- **The level  $S_j$  a literal  $l$  appears first is never greater than the level it can be achieved in a plan**
  - $j$  is called the **level cost** of literal  $l$
- **the level cost of a literal  $g_j$  in the graph constructed starting from state  $s$ , is an estimate of the cost to achieve it from  $s$  (i.e.  $h(g)$ )**
  - this estimate **is admissible**
  - ex: from  $s_0$  **Have(cake)** has cost 0 and **Eaten(cake)** has cost 1
- Planning graph admits several actions per level  
⇒ inaccurate estimate
- **Serialization**: enforcing only one action per level (adding mutex)  
⇒ better estimate

# Planning Graphs for Heuristic Estimation [cont.]

## Estimating the heuristic cost of a conjunction of goal literals

- **Max-level heuristic:** the maximum level cost of the sub-goals
  - admissible
- **Level-sum heuristic:** the sum of the level costs of the goals
  - inadmissible only if goals are independent,
  - it may work well in practice
- **Set-level heuristic:** the level at which all goal literals appear together, without pairwise mutexes
  - admissible, more accurate

## Estimating the heuristic cost of a conjunction of goal literals

- **Max-level heuristic:** the maximum level cost of the sub-goals
  - admissible
- **Level-sum heuristic:** the sum of the level costs of the goals
  - inadmissible only if goals are independent,
  - it may work well in practice
- **Set-level heuristic:** the level at which all goal literals appear together, without pairwise mutexes
  - admissible, more accurate

# Planning Graphs for Heuristic Estimation [cont.]

## Estimating the heuristic cost of a conjunction of goal literals

- **Max-level heuristic:** the maximum level cost of the sub-goals
  - admissible
- **Level-sum heuristic:** the sum of the level costs of the goals
  - inadmissible only if goals are independent,
  - it may work well in practice
- **Set-level heuristic:** the level at which all goal literals appear together, without pairwise mutexes
  - admissible, more accurate

# Planning Graphs for Heuristic Estimation [cont.]

## Estimating the heuristic cost of a conjunction of goal literals

- **Max-level heuristic:** the maximum level cost of the sub-goals
  - admissible
- **Level-sum heuristic:** the sum of the level costs of the goals
  - inadmissible only if goals are independent,
  - it may work well in practice
- **Set-level heuristic:** the level at which all goal literals appear together, without pairwise mutexes
  - admissible, more accurate

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan**
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm**
- 4 Other Approaches (hints)
  - Planning as SAT Solving



# The Graphplan Algorithm

- A strategy for extracting a plan from the planning graph
- Repeatedly adds a level to a planning graph (EXPAND-GRAPH)
- If all the goal literals occur in last level and are non-mutex
  - search for a plan that solves the problem (EXTRACT-SOLUTION)
  - if that fails, expand another level and try again (and add  $\langle goal, level \rangle$  as nogood)
- If graph and nogoods have both leveled off then return failure
- Depends on EXPAND-GRAPH & EXTRACT-SOLUTION

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

*graph*  $\leftarrow$  INITIAL-PLANNING-GRAPH(*problem*)

*goals*  $\leftarrow$  CONJUNCTS(*problem*.GOAL)

*nogoods*  $\leftarrow$  an empty hash table

**for**  $t = 0$  **to**  $\infty$  **do**

typo in  
AIMA  
book **if** *goals* all non-mutex in  $S_t$  of *graph* **then**

*solution*  $\leftarrow$  EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)

**if** *solution*  $\neq$  failure **then return** *solution*

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

*graph*  $\leftarrow$  EXPAND-GRAPH(*graph*, *problem*)

# The Graphplan Algorithm

- A strategy for extracting a plan from the planning graph
- Repeatedly adds a level to a planning graph (**EXPAND-GRAPH**)
- If all the goal literals occur in last level and are non-mutex
  - search for a plan that solves the problem (**EXTRACT-SOLUTION**)
  - if that fails, expand another level and try again (and add  $\langle goal, level \rangle$  as nogood)
- If graph and nogoods have both leveled off then return failure
- Depends on **EXPAND-GRAPH** & **EXTRACT-SOLUTION**

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

*graph*  $\leftarrow$  INITIAL-PLANNING-GRAPH(*problem*)

*goals*  $\leftarrow$  CONJUNCTS(*problem*.GOAL)

*nogoods*  $\leftarrow$  an empty hash table

**for**  $t = 0$  **to**  $\infty$  **do**

typo in  
AIMA  
book **if** *goals* all non-mutex in  $S_t$  of *graph* **then**

*solution*  $\leftarrow$  EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)

**if** *solution*  $\neq$  failure **then return** *solution*

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

*graph*  $\leftarrow$  EXPAND-GRAPH(*graph*, *problem*)

# The Graphplan Algorithm

- A strategy for extracting a plan from the planning graph
- Repeatedly adds a level to a planning graph (**EXPAND-GRAPH**)
- If all the goal literals occur in last level and are non-mutex
  - search for a plan that solves the problem (**EXTRACT-SOLUTION**)
  - if that fails, expand another level and try again (and add  $\langle goal, level \rangle$  as nogood)
- If graph and nogoods have both leveled off then return failure
- Depends on **EXPAND-GRAPH** & **EXTRACT-SOLUTION**

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

*graph*  $\leftarrow$  INITIAL-PLANNING-GRAPH(*problem*)

*goals*  $\leftarrow$  CONJUNCTS(*problem*.GOAL)

*nogoods*  $\leftarrow$  an empty hash table

**for**  $t = 0$  **to**  $\infty$  **do**

typo in  
AIMA  
book **if** *goals* all non-mutex in  $S_t$  of *graph* **then**

*solution*  $\leftarrow$  EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)

**if** *solution*  $\neq$  failure **then return** *solution*

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

*graph*  $\leftarrow$  EXPAND-GRAPH(*graph*, *problem*)

# The Graphplan Algorithm

- A strategy for extracting a plan from the planning graph
- Repeatedly adds a level to a planning graph (**EXPAND-GRAPH**)
- If all the goal literals occur in last level and are non-mutex
  - search for a plan that solves the problem (**EXTRACT-SOLUTION**)
  - if that fails, expand another level and try again (and add  $\langle goal, level \rangle$  as nogood)
- If graph and nogoods have both leveled off then return failure
- Depends on **EXPAND-GRAPH** & **EXTRACT-SOLUTION**

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

*graph*  $\leftarrow$  INITIAL-PLANNING-GRAPH(*problem*)

*goals*  $\leftarrow$  CONJUNCTS(*problem*.GOAL)

*nogoods*  $\leftarrow$  an empty hash table

**for**  $t = 0$  **to**  $\infty$  **do**

typo in  
AIMA  
book **if** *goals* all non-mutex in  $S_t$  of *graph* **then**

*solution*  $\leftarrow$  EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)

**if** *solution*  $\neq$  failure **then return** *solution*

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

*graph*  $\leftarrow$  EXPAND-GRAPH(*graph*, *problem*)

# The Graphplan Algorithm

- A strategy for extracting a plan from the planning graph
- Repeatedly adds a level to a planning graph (**EXPAND-GRAPH**)
- If all the goal literals occur in last level and are non-mutex
  - search for a plan that solves the problem (**EXTRACT-SOLUTION**)
  - if that fails, expand another level and try again (and add  $\langle goal, level \rangle$  as nogood)
- If graph and nogoods have both leveled off then return failure
- Depends on **EXPAND-GRAPH** & **EXTRACT-SOLUTION**

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

*graph*  $\leftarrow$  INITIAL-PLANNING-GRAPH(*problem*)

*goals*  $\leftarrow$  CONJUNCTS(*problem*.GOAL)

*nogoods*  $\leftarrow$  an empty hash table

**for**  $t = 0$  **to**  $\infty$  **do**

typo in  
AIMA  
book **if** *goals* all non-mutex in  $S_t$  of *graph* **then**

*solution*  $\leftarrow$  EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)

**if** *solution*  $\neq$  failure **then return** *solution*

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

*graph*  $\leftarrow$  EXPAND-GRAPH(*graph*, *problem*)

## [Recall] Example: 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:  
    EFFECT:  $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$   
             $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk)$ )

(© S. Russell & P. Norwig, AIMA)

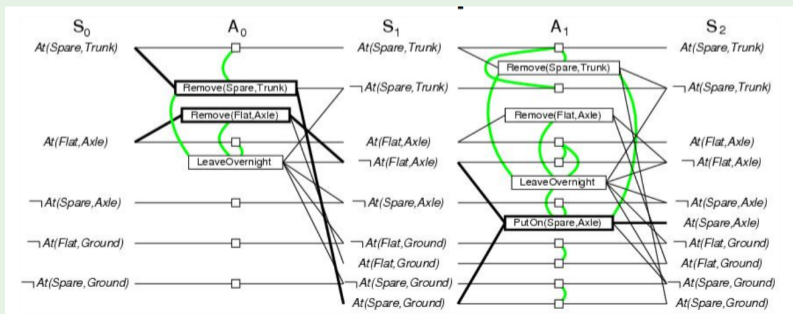
(We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.)

One solution:  $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$

# Graphplan: Example

## Spare Tire problem

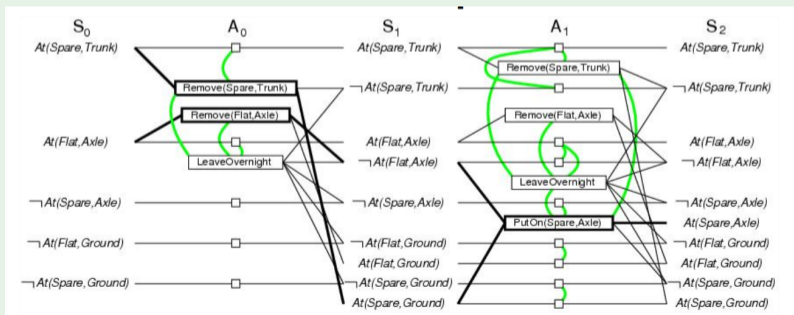
- Initial plan 5 literals from initial state and the Closed-World-Assumption literals ( $S_0$ ).
  - fixed literals (e.g.  $Tire(Flat)$ ) ignored here
  - irrelevant literals ignored here
- Goal  $At(Spare, Axle)$  not present in  $S_0$   
⇒ no need to call EXTRACT-SOLUTION
- Graph and nogoods not leveled off ⇒ invoke EXPAND-GRAPH



# Graphplan: Example [cont.]

## Spare Tire problem

- Invoke EXPAND-GRAPH
  - add actions  $A_0$ , persistence actions and mutexes
  - add fluents  $S_1$  and mutexes
- Goal  $At(Spare, Axle)$  not present in  $S_1$   
⇒ no need to call EXTRACT-SOLUTION
- Graph and nogoods not leveled off ⇒ invoke EXPAND-GRAPH

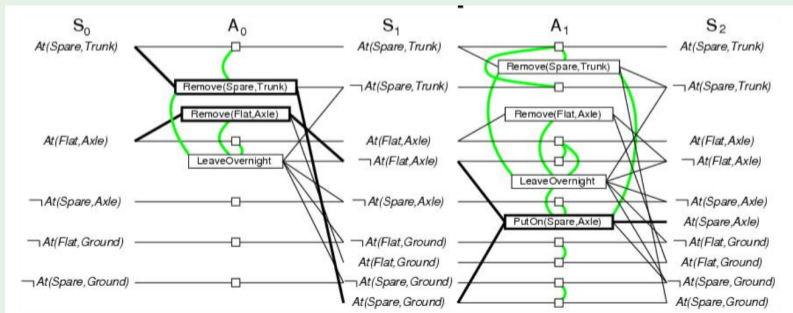




# Graphplan: Example [cont.]

## Spare Tire problem

- Invoke EXPAND-GRAPH
  - add actions  $A_1$ , persistence actions and mutexes
  - add fluents  $S_2$  and mutexes
- Goal  $At(Spare, Axle)$  present in  $S_2$ 
  - call EXTRACT-SOLUTION
- **Solution found!**



## Exercise

- Consider the following variant of the Spare Tire problem:  
add  $At(Flat, Trunk)$  to the goal
- Write the (non-serialized) planning graph
- Extract a plan from the graph
- Do the same with the serialized planning graph

# The Graphplan Algorithm [cont.]

Graphplan “family” of algorithms, depending on approach used in EXTRACT-SOLUTION(...)

## About EXTRACT-SOLUTION(...)

- Can be formulated as an (incremental) **SAT problem**
  - one proposition for each ground action and fluent
  - clauses represent preconditions, effects, no-ops and mutexes
- Can be formulated as a **backward search problem**
- Planning problem **restricted to planning graph**
  - mutexes found by EXPAND-GRAPH prune paths in the search tree
  - ⇒ **much faster than unrestricted planning**
- (if P.G. not serialized) may produce **partial order plans**
  - ⇒ may be later serialized into a total-order plan

# The Graphplan Algorithm [cont.]

Graphplan “family” of algorithms, depending on approach used in EXTRACT-SOLUTION(...)

## About EXTRACT-SOLUTION(...)

- Can be formulated as an (incremental) **SAT problem**
  - one proposition for each ground action and fluent
  - clauses represent preconditions, effects, no-ops and mutexes
- Can be formulated as a **backward search problem**
- Planning problem **restricted to planning graph**
  - mutexes found by EXPAND-GRAPH prune paths in the search tree
  - ⇒ **much faster than unrestricted planning**
- (if P.G. not serialized) may produce **partial order plans**
  - ⇒ may be later serialized into a total-order plan

# The Graphplan Algorithm [cont.]

Graphplan “family” of algorithms, depending on approach used in EXTRACT-SOLUTION(...)

## About EXTRACT-SOLUTION(...)

- Can be formulated as an (incremental) **SAT problem**
  - one proposition for each ground action and fluent
  - clauses represent preconditions, effects, no-ops and mutexes
- Can be formulated as a **backward search problem**
- Planning problem **restricted to planning graph**
  - mutexes found by EXPAND-GRAPH prune paths in the search tree
  - ⇒ **much faster than unrestricted planning**
- (if P.G. not serialized) may produce **partial order plans**
  - ⇒ may be later serialized into a total-order plan

# The Graphplan Algorithm [cont.]

Graphplan “family” of algorithms, depending on approach used in EXTRACT-SOLUTION(...)

## About EXTRACT-SOLUTION(...)

- Can be formulated as an (incremental) **SAT problem**
  - one proposition for each ground action and fluent
  - clauses represent preconditions, effects, no-ops and mutexes
- Can be formulated as a **backward search problem**
- Planning problem **restricted to planning graph**
  - mutexes found by EXPAND-GRAPH prune paths in the search tree
  - ⇒ **much faster than unrestricted planning**
- (if P.G. not serialized) may produce **partial order plans**
  - ⇒ may be later serialized into a total-order plan

# The Graphplan Algorithm [cont.]

Graphplan “family” of algorithms, depending on approach used in EXTRACT-SOLUTION(...)

## About EXTRACT-SOLUTION(...)

- Can be formulated as an (incremental) **SAT problem**
  - one proposition for each ground action and fluent
  - clauses represent preconditions, effects, no-ops and mutexes
- Can be formulated as a **backward search problem**
- Planning problem **restricted to planning graph**
  - mutexes found by EXPAND-GRAPH prune paths in the search tree
  - ⇒ **much faster than unrestricted planning**
- (if P.G. not serialized) may produce **partial order plans**
  - ⇒ may be later serialized into a total-order plan

# Partial-Order Plans

## Partial-Order vs. Total-Order Plans

- **Total-order plans:** **strictly linear sequences of actions**
  - disregards the fact that some action are mutually independent
- **Partial-order plans:** **set of precedence constraints between action pairs**
  - form a directed acyclic graph
  - longest path to goal may be much shorter than total-order plan
  - **easily converted into (possibly many) distinct total-order plans**  
(any possible interleaving of independent actions)



# Partial-Order Plans

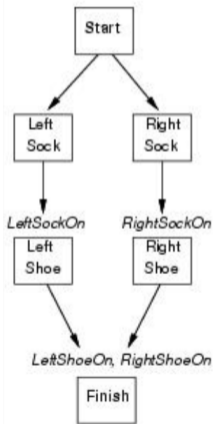
## Partial-Order vs. Total-Order Plans

- **Total-order plans:** strictly linear sequences of actions
  - disregards the fact that some action are mutually independent
- **Partial-order plans:** set of precedence constraints between action pairs
  - form a directed acyclic graph
  - longest path to goal may be much shorter than total-order plan
  - easily converted into (possibly many) distinct total-order plans  
(any possible interleaving of independent actions)

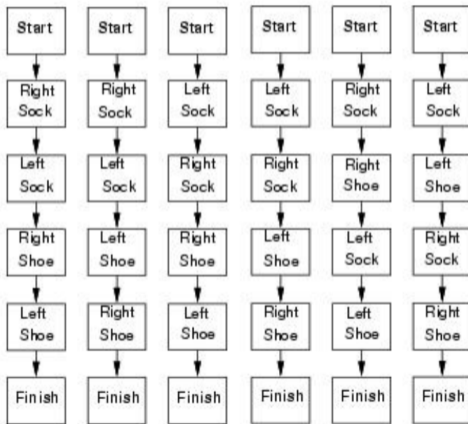
# Partial-Order Plans: Example

## Socks & Shoes Examples

**Partial Order Plan:**



**Total Order Plans:**



# Termination of Graphplan

- Theorem: If the graph and the no-goods have both leveled off, and no solution is found we can safely terminate with failure
- Intuition (proof sketch):
  - Literals and actions increase monotonically and are finite  
⇒ we eventually reach a level where they stabilize
  - Mutexes and no-goods decrease monotonically (and cannot become less than zero)  
⇒ they too eventually must level off
- ⇒ When we reach this stable state, if one of the goal literals is missing or is mutex with another goal literal, then it will remain so  
⇒ we can stop

# Termination of Graphplan

- **Theorem:** If the graph and the no-goods have both leveled off, and no solution is found we can safely terminate with failure
- Intuition (proof sketch):
  - Literals and actions increase monotonically and are finite  
⇒ we eventually reach a level where they stabilize
  - Mutexes and no-goods decrease monotonically (and cannot become less than zero)  
⇒ they too eventually must level off
- ⇒ When we reach this stable state, if one of the goal literals is missing or is mutex with another goal literal, then it will remain so  
⇒ we can stop

# Termination of Graphplan

- Theorem: If the graph and the no-goods have both leveled off, and no solution is found we can safely terminate with failure
- Intuition (proof sketch):
  - Literals and actions increase monotonically and are finite  
⇒ we eventually reach a level where they stabilize
  - Mutexes and no-goods decrease monotonically (and cannot become less than zero)  
⇒ they too eventually must level off
- ⇒ When we reach this stable state, if one of the goal literals is missing or is mutex with another goal literal, then it will remain so  
⇒ we can stop

# Exercise

- Socks & Shoes example:
  - 1 Formalize the Socks & Shoes example in PDDL
  - 2 Write the non-serialized planning graph
  - 3 Compute the level cost for every fluent
  - 4 Choose some states, compute  $h(s)$  using the three heuristics
  - 5 Extract a plan from the graph in (2)
  - 6 Compare  $h(s)$  with the level they occur in the plan
  - 7 Write the serialized planning graph
  - 8 Repeat steps (3)-(6) with the serialized graph
- Do same steps (1)-(8) for the Air Cargo Transport example

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 **Other Approaches (hints)**
  - Planning as SAT Solving

# Outline

- 1 Basics on Planning
  - The Problem
  - The PDDL Language
- 2 Search Strategies and Heuristics
  - Forward and Backward Search
  - Heuristics
- 3 Planning Graphs, Heuristics and Graphplan
  - Planning Graphs
  - Heuristics Driven by Planning Graphs
  - The Graphplan Algorithm
- 4 Other Approaches (hints)
  - Planning as SAT Solving



# Planning as SAT Solving

- Encode bounded planning problem into a propositional formula
- ⇒ Solve it by (incremental) calls to a SAT solver
- A model for the formula (if any) is a plan of length  $t$
  - Many variants in the encoding
  - Extremely efficient with many problems of interest

**function** SATPLAN(*init*, *transition*, *goal*,  $T_{\max}$ ) **returns** solution or failure

**inputs:** *init*, *transition*, *goal*, constitute a description of the problem

$T_{\max}$ , an upper limit for plan length

**for**  $t = 0$  **to**  $T_{\max}$  **do**

$cnf \leftarrow$  TRANSLATE-TO-SAT(*init*, *transition*, *goal*,  $t$ )

$model \leftarrow$  SAT-SOLVER( $cnf$ )

**if**  $model$  is not null **then**

**return** EXTRACT-SOLUTION( $model$ )

**return** *failure*

# Planning as SAT Solving

- Encode bounded planning problem into a propositional formula
- ⇒ Solve it by (incremental) calls to a SAT solver
- A model for the formula (if any) is a plan of length  $t$
  - Many variants in the encoding
  - Extremely efficient with many problems of interest

**function** SATPLAN(*init*, *transition*, *goal*,  $T_{\max}$ ) **returns** solution or failure

**inputs:** *init*, *transition*, *goal*, constitute a description of the problem

$T_{\max}$ , an upper limit for plan length

**for**  $t = 0$  **to**  $T_{\max}$  **do**

*cnf*  $\leftarrow$  TRANSLATE-TO-SAT(*init*, *transition*, *goal*,  $t$ )

*model*  $\leftarrow$  SAT-SOLVER(*cnf*)

**if** *model* is not null **then**

**return** EXTRACT-SOLUTION(*model*)

**return** *failure*

# Planning as SAT Solving

- Encode bounded planning problem into a propositional formula
- ⇒ Solve it by (incremental) calls to a SAT solver
- A model for the formula (if any) is a plan of length  $t$
- Many variants in the encoding
- Extremely efficient with many problems of interest

**function** SATPLAN(*init*, *transition*, *goal*,  $T_{\max}$ ) **returns** solution or failure

**inputs:** *init*, *transition*, *goal*, constitute a description of the problem

$T_{\max}$ , an upper limit for plan length

**for**  $t = 0$  **to**  $T_{\max}$  **do**

$cnf \leftarrow$  TRANSLATE-TO-SAT(*init*, *transition*, *goal*,  $t$ )

$model \leftarrow$  SAT-SOLVER( $cnf$ )

**if**  $model$  is not null **then**

**return** EXTRACT-SOLUTION( $model$ )

**return** *failure*

# Planning as SAT Solving

- Encode bounded planning problem into a propositional formula
- ⇒ Solve it by (incremental) calls to a SAT solver
- A model for the formula (if any) is a plan of length  $t$
  - Many variants in the encoding
  - Extremely efficient with many problems of interest

**function** SATPLAN(*init*, *transition*, *goal*,  $T_{\max}$ ) **returns** solution or failure

**inputs:** *init*, *transition*, *goal*, constitute a description of the problem

$T_{\max}$ , an upper limit for plan length

**for**  $t = 0$  **to**  $T_{\max}$  **do**

$cnf \leftarrow$  TRANSLATE-TO-SAT(*init*, *transition*, *goal*,  $t$ )

$model \leftarrow$  SAT-SOLVER( $cnf$ )

**if**  $model$  is not null **then**

**return** EXTRACT-SOLUTION( $model$ )

**return** *failure*

# Planning as SAT Solving [cont.]

- **TRANSLATE-TO-SAT**(INIT, TRANSITION, GOAL, T):
  - ground fluents & actions at each step **are propositionalized**
    - ex:  $\langle At(P_1, SFO), 3 \rangle \implies At\_P1\_SFO\_3$
    - ex:  $\langle Fly(P_1, SFO, JFK), 3 \rangle \implies Fly\_P1\_SFO\_JFK\_3$
  - returns propositional formula:  $Init^0 \wedge (\bigwedge_{i=1}^{t-1} Transition^{i,i+1}) \wedge Goal^t$
- $Init^0$  and  $Goal^t$ : conjunctions of literals at step 0 and t resp.
  - ex:  $Init^0: At\_P1\_SFO\_0 \wedge At\_P2\_JFK\_0$
  - ex:  $Goal^3: At\_P1\_JFK\_3 \wedge At\_P2\_SFO\_3$
- $Transition^{i,i+1}$ : encodes transition from steps  $i$  to  $i + 1$ 
  - Actions:  $Action^i \rightarrow (Precond^i \wedge Effects^{i+1})$   
ex:  $Fly\_P1\_SFO\_JFK\_2 \rightarrow (At\_P1\_SFO\_2 \wedge At\_P1\_JFK\_3)$
  - No-Ops: for each fluent  $F$  and step  $i$ :
$$F^{i+1} \leftrightarrow \bigvee_k ActionCausingF_k^i \vee (F^i \wedge \bigwedge_j \neg ActionCausingNotF_j^i)$$
  - Mutex constraints:  $\neg Action_1^i \vee \neg Action_2^i$   
ex:  $\neg Fly\_P1\_SFO\_JFK\_2 \vee \neg Fly\_P1\_SFO\_Newark\_2$
  - If serialized: add mutex between each pair of actions at each step

# Planning as SAT Solving [cont.]

- **TRANSLATE-TO-SAT**(INIT, TRANSITION, GOAL, T):
  - ground fluents & actions at each step **are propositionalized**
    - ex:  $\langle At(P_1, SFO), 3 \rangle \implies At\_P_1\_SFO\_3$
    - ex:  $\langle Fly(P_1, SFO, JFK), 3 \rangle \implies Fly\_P_1\_SFO\_JFK\_3$
  - returns propositional formula:  $Init^0 \wedge (\bigwedge_{i=1}^{t-1} Transition^{i,i+1}) \wedge Goal^t$
- $Init^0$  and  $Goal^t$ : conjunctions of literals at step 0 and t resp.
  - ex:  $Init^0: At\_P_1\_SFO\_0 \wedge At\_P_2\_JFK\_0$
  - ex:  $Goal^t: At\_P_1\_JFK\_3 \wedge At\_P_2\_SFO\_3$
- $Transition^{i,i+1}$ : encodes transition from steps  $i$  to  $i + 1$ 
  - Actions:  $Action^i \rightarrow (Precond^i \wedge Effects^{i+1})$   
ex:  $Fly\_P_1\_SFO\_JFK\_2 \rightarrow (At\_P_1\_SFO\_2 \wedge At\_P_1\_JFK\_3)$
  - No-Ops: for each fluent  $F$  and step  $i$ :
$$F^{i+1} \leftrightarrow \bigvee_k ActionCausingF_k^i \vee (F^i \wedge \bigwedge_j \neg ActionCausingNotF_j^i)$$
  - Mutex constraints:  $\neg Action_1^i \vee \neg Action_2^i$   
ex:  $\neg Fly\_P_1\_SFO\_JFK\_2 \vee \neg Fly\_P_1\_SFO\_Newark\_2$
  - If serialized: add mutex between each pair of actions at each step

# Planning as SAT Solving [cont.]

- TRANSLATE-TO-SAT(INIT, TRANSITION, GOAL, T):
  - ground fluents & actions at each step are propositionalized
    - ex:  $\langle At(P_1, SFO), 3 \rangle \implies At\_P_1\_SFO\_3$
    - ex:  $\langle Fly(P_1, SFO, JFK), 3 \rangle \implies Fly\_P_1\_SFO\_JFK\_3$
  - returns propositional formula:  $Init^0 \wedge (\bigwedge_{i=1}^{t-1} Transition^{i,i+1}) \wedge Goal^t$
- $Init^0$  and  $Goal^t$ : conjunctions of literals at step 0 and t resp.
  - ex:  $Init^0: At\_P_1\_SFO\_0 \wedge At\_P_2\_JFK\_0$
  - ex:  $Goal^3: At\_P_1\_JFK\_3 \wedge At\_P_2\_SFO\_3$
- $Transition^{i,i+1}$ : encodes transition from steps  $i$  to  $i + 1$ 
  - Actions:  $Action^i \rightarrow (Precond^i \wedge Effects^{i+1})$   
ex:  $Fly\_P_1\_SFO\_JFK\_2 \rightarrow (At\_P_1\_SFO\_2 \wedge At\_P_1\_JFK\_3)$
  - No-Ops: for each fluent  $F$  and step  $i$ :
$$F^{i+1} \leftrightarrow \bigvee_k ActionCausingF_k^i \vee (F^i \wedge \bigwedge_j \neg ActionCausingNotF_j^i)$$
  - Mutex constraints:  $\neg Action_1^i \vee \neg Action_2^i$   
ex:  $\neg Fly\_P_1\_SFO\_JFK\_2 \vee \neg Fly\_P_1\_SFO\_Newark\_2$
  - If serialized: add mutex between each pair of actions at each step

# Exercise

Consider the socks & shoes example

- Translate it into SAT for  $t=0,1,2$ 
  - non serialized
  - no need to propositionalize: treat ground atoms as propositions
  - no need to CNF-ize here (human beings don't like CNFs)
- Find a model for the formula
- Convert it back to a plan



# Exercise

Consider the socks & shoes example

- Translate it into SAT for  $t=0,1,2$ 
  - non serialized
  - no need to propositionalize: treat ground atoms as propositions
  - no need to CNF-ize here (human beings don't like CNFs)
- Find a model for the formula
- Convert it back to a plan

# Exercise

Consider the socks & shoes example

- Translate it into SAT for  $t=0,1,2$ 
  - non serialized
  - no need to propositionalize: treat ground atoms as propositions
  - no need to CNF-ize here (human beings don't like CNFs)
- Find a model for the formula
- Convert it back to a plan