Fundamentals of Artificial Intelligence
Chapter 10: **Classical Planning**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it

https://disi.unitn.it/rseba/DIDATTICA/fai_2025/

Teaching assistant:

Paolo Morettin, paolo.morettin@unitn.it, https://paolomorettin.github.io/

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

Last update: Wednesday 10th September, 2025, 16:24

Outline

Outline

Outline

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

    $\Longrightarrow$ 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

    $\Longrightarrow$ 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

A Language for Planning: PDDL

Planning Domain Definition Language (PDDL)

- A state is a conjunction of fluents: ground, function-less atoms
  - ex: $Poor \land Unknown$, $At(Truck_1, Melbourne) \land At(Truck_2, Sydney)$
  - examples 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
  - $\Longrightarrow$ 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: $Poor \land Unknown$, $At(Truck_1, Melbourne) \land At(Truck_2, Sydney)$
  - examples 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

  $\Longrightarrow$ 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: $Poor \land Unknown$, $At(Truck_1, Melbourne) \land At(Truck_2, Sydney)$
  - examples 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

  $\Longrightarrow$ 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: $Poor \land Unknown$, $At(Truck_1, Melbourne) \land At(Truck_2, Sydney)$
  - examples 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

  $\Longrightarrow$ 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: $Poor \land Unknown$, $At(Truck_1, Melbourne) \land At(Truck_2, Sydney)$
  - examples 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

  $\Longrightarrow$ 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: $Poor \wedge Unknown$, $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$
  - examples 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

  $\Longrightarrow$ 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:
  $Action(Fly(p, from, to),$
  $PRECOND : Plane(p) \wedge Airport(from) \wedge Airport(to) \wedge At(p, from)$
  $EFFECT : \neg At(p, from) \wedge At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$
  $EFFECT : \neg At(P_1, SFO) \wedge At(P_1, JFK))$

PDDL: Example

- Action schema:
  $Action(Fly(p, from, to),$
  $\quad PRECOND : Plane(p) \land Airport(from) \land Airport(to) \land At(p, from)$
  $\quad EFFECT \quad : \neg At(p, from) \land At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $\quad PRECOND : Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land At(P_1, SFO)$
  $\quad EFFECT \quad : \neg At(P_1, SFO) \land At(P_1, JFK))$

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 \ 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: $Fly(P_1, SFO, JFK) \Longrightarrow$ 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 \ 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: $Fly(P_1, SFO, JFK) \Longrightarrow$ 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 \ DEL(a) ∪ 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: $Fly(P_1, SFO, JFK) \Longrightarrow$ 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 ($\textsc{Add}(a)$): (the fluents in) the positive literals in the action's effects
  - ex: $\{At(p, to)\}$
- Delete list ($\textsc{Del}(a)$): (the fluents in) the negative literals in the action's effects
  - ex: $\{At(p, from)\}$
- Result of action a in state s: $\textsc{Result}(s,a) \stackrel{\text{def}}{=} (s \setminus \textsc{Del}(a) \cup \textsc{Add}(a))$
  - start from s
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex: $Fly(P_1, SFO, JFK) \Longrightarrow$ 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 ($\textsc{Add}(a)$): (the fluents in) the positive literals in the action's effects
  - ex: $\{At(p, to)\}$
- Delete list ($\textsc{Del}(a)$): (the fluents in) the negative literals in the action's effects
  - ex: $\{At(p, from)\}$
- Result of action a in state s: $\textsc{Result}(s,a) \stackrel{\text{def}}{=} (s \setminus \textsc{Del}(a) \cup \textsc{Add}(a))$
  - start from s
  - remove the fluents that appear as negative literals in effect
  - add the fluents that appear as positive literals in effect
  - ex: $Fly(P_1, SFO, JFK) \Longrightarrow$ 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:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$
  $EFFECT \ : \neg At(P_1, SFO) \wedge At(P_1, JFK))$
- $s : At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge ...$

$\Longrightarrow s' : At(P_1, JFK) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge ...$

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

- Ex: $...At\_P_1\_SFO \wedge Plane\_P_1 \wedge Airport\_SFO \wedge Airport\_JFK)...$

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:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND: Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge At(P_1, SFO)$
  $EFFECT: \neg At(P_1, SFO) \wedge At(P_1, JFK))$
- $s : At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge ...$

$\Longrightarrow s' : At(P_1, JFK) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK) \wedge ...$

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

- Ex: $...At\_P_1\_SFO \wedge Plane\_P_1 \wedge Airport\_SFO \wedge Airport\_JFK)...$

PDL: Example [cont.]

- Action schema:
  $Action(Fly(p, from, to),$
  $PRECOND : Plane(p) \land Airport(from) \land Airport(to) \land At(p, from)$
  $EFFECT \quad : \neg At(p, from) \land At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land At(P_1, SFO)$
  $EFFECT \quad : \neg At(P_1, SFO) \land At(P_1, JFK))$
- $s : At(P_1, SFO) \land Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land ...$

$\Longrightarrow s' : At(P_1, JFK) \land Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land ...$

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

- Ex: $...At\_P_1\_SFO \land Plane\_P_1 \land Airport\_SFO \land Airport\_JFK)...$

PDDL: Example [cont.]

- Action schema:
  $Action(Fly(p, from, to),$
  $PRECOND : Plane(p) \land Airport(from) \land Airport(to) \land At(p, from)$
  $EFFECT \;\; : \neg At(p, from) \land At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land At(P_1, SFO)$
  $EFFECT \;\; : \neg At(P_1, SFO) \land At(P_1, JFK))$
- $s : At(P_1, SFO) \land Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land ...$

$\Longrightarrow$ $s' : At(P_1, JFK) \land Plane(P_1) \land Airport(SFO) \land Airport(JFK) \land ...$

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

- Ex: $...At\_P_1\_SFO \land Plane\_P_1 \land Airport\_SFO \land Airport\_JFK)...$

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: $At(p, SFO) \land Plane(p)$:
  - variable "p" implicitly means "for some plane p"
  - the state $Plane(Plane_1) \land At(Plane_1, SFO) \land ...$ 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: $At(p, SFO) \land Plane(p)$:
  - variable "p" implicitly means "for some plane p"
  - the state $Plane(Plane_1) \land At(Plane_1, SFO) \land \ldots$ 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 $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge \ldots$ 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 $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge ...$ 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 $Plane(Plane_1) \wedge At(Plane_1, SFO) \wedge ...$ 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

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

One solution: [$Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK)$, $Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)$]

Example: Air Cargo Transport

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

One solution: [$Load(C_1, P_1, SFO)$, $Fly(P_1, SFO, JFK)$, $Unload(C_1, P_1, JFK)$, $Load(C2, P2, JFK)$, $Fly(P2, JFK, SFO)$, $Unload(C2, P2, SFO)$]

Example: Spare Tire Problem

$$
\begin{array}{l}
Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk)) \\
Goal(At(Spare, Axle)) \\
Action(Remove(obj, loc), \\
\quad \text{PRECOND: } At(obj, loc) \\
\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground)) \\
Action(PutOn(t, Axle), \\
\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle) \\
\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle)) \\
Action(LeaveOvernight, \\
\quad \text{PRECOND: } \\
\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk) \\
\qquad\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))
\end{array}
$$

(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*)]

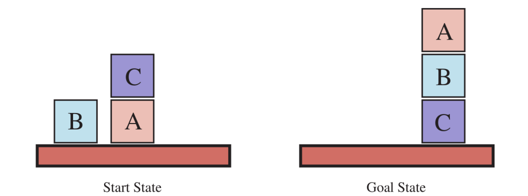
Example: Spare Tire Problem

$$
\begin{array}{l}
Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))\\
Goal(At(Spare, Axle))\\
Action(Remove(obj, loc),\\
\quad \text{PRECOND: } At(obj, loc)\\
\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground))\\
Action(PutOn(t, Axle),\\
\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)\\
\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle))\\
Action(LeaveOvernight,\\
\quad \text{PRECOND:}\\
\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)\\
\qquad\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))
\end{array}
$$

(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*)]

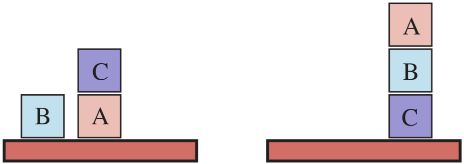
Example: Blocks World



$$
\begin{array}{l}
\mathit{Init}(\mathit{On}(A, \mathit{Table}) \land \mathit{On}(B, \mathit{Table}) \land \mathit{On}(C, A) \\
\quad \land \mathit{Block}(A) \land \mathit{Block}(B) \land \mathit{Block}(C) \land \mathit{Clear}(B) \land \mathit{Clear}(C)) \\
\mathit{Goal}(\mathit{On}(A, B) \land \mathit{On}(B, C)) \\
\mathit{Action}(\mathit{Move}(b, x, y), \\
\quad \text{PRECOND: } \mathit{On}(b, x) \land \mathit{Clear}(b) \land \mathit{Clear}(y) \land \mathit{Block}(b) \land \mathit{Block}(y) \land \\
\qquad\qquad (b \neq x) \land (b \neq y) \land (x \neq y), \\
\quad \text{EFFECT: } \mathit{On}(b, y) \land \mathit{Clear}(x) \land \neg \mathit{On}(b, x) \land \neg \mathit{Clear}(y)) \\
\mathit{Action}(\mathit{MoveToTable}(b, x), \\
\quad \text{PRECOND: } \mathit{On}(b, x) \land \mathit{Clear}(b) \land \mathit{Block}(b) \land (b \neq x), \\
\quad \text{EFFECT: } \mathit{On}(b, \mathit{Table}) \land \mathit{Clear}(x) \land \neg \mathit{On}(b, x))
\end{array}
$$

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

Example: Blocks World



Start State Goal State

$$
\begin{aligned}
&\mathit{Init}(\mathit{On}(A, \mathit{Table}) \land \mathit{On}(B, \mathit{Table}) \land \mathit{On}(C, A) \\
&\quad \land \mathit{Block}(A) \land \mathit{Block}(B) \land \mathit{Block}(C) \land \mathit{Clear}(B) \land \mathit{Clear}(C)) \\
&\mathit{Goal}(\mathit{On}(A, B) \land \mathit{On}(B, C)) \\
&\mathit{Action}(\mathit{Move}(b, x, y), \\
&\quad \text{PRECOND: } \mathit{On}(b, x) \land \mathit{Clear}(b) \land \mathit{Clear}(y) \land \mathit{Block}(b) \land \mathit{Block}(y) \land \\
&\qquad\qquad (b \neq x) \land (b \neq y) \land (x \neq y), \\
&\quad \text{EFFECT: } \mathit{On}(b, y) \land \mathit{Clear}(x) \land \neg \mathit{On}(b, x) \land \neg \mathit{Clear}(y)) \\
&\mathit{Action}(\mathit{MoveToTable}(b, x), \\
&\quad \text{PRECOND: } \mathit{On}(b, x) \land \mathit{Clear}(b) \land \mathit{Block}(b) \land (b \neq x), \\
&\quad \text{EFFECT: } \mathit{On}(b, \mathit{Table}) \land \mathit{Clear}(x) \land \neg \mathit{On}(b, x))
\end{aligned}
$$

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

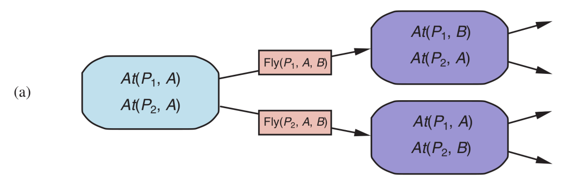
Outline

Outline

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

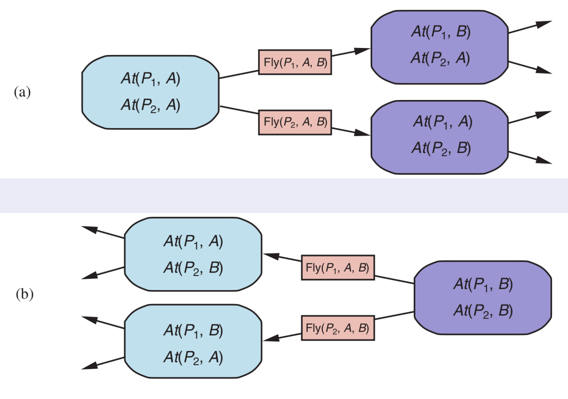


(© S. Russell & P. Norwig, AIMA)

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

$\Longrightarrow$ 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! $\Longrightarrow$ 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

$\Longrightarrow$ 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! $\Longrightarrow$ 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

$\Longrightarrow$ 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! $\Longrightarrow$ 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

$\Longrightarrow$ 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! $\Longrightarrow$ 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

$\Longrightarrow$ 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! $\Longrightarrow$ 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

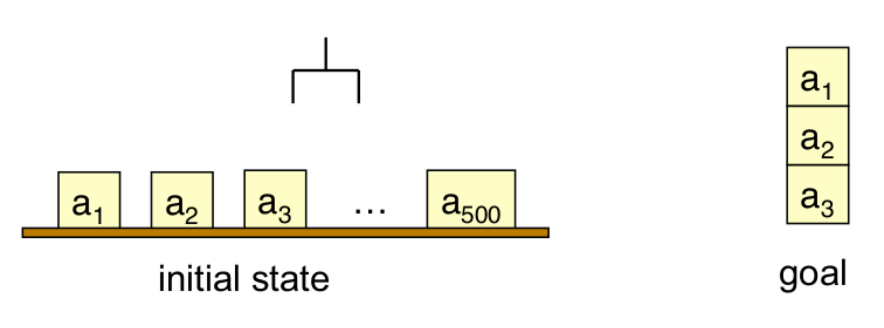
$\Longrightarrow$ We can use any of the search algorithms from Ch. 03, 04
  - need keeping track of the actions used to reach the goal
- Breadth-first and best-first
  - Sound: if they return a plan, then the plan is a solution
  - Complete: if a problem has a solution, then they will return one
  - Require exponential memory wrt. solution length! $\Longrightarrow$ 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: $pickup(a_1)$, $pickup(a_2)$, ..., $pickup(a_{500})$

$\Longrightarrow$ Forward-search can waste time trying lots of irrelevant actions

$\Longrightarrow$ Need a good heuristic to guide the search

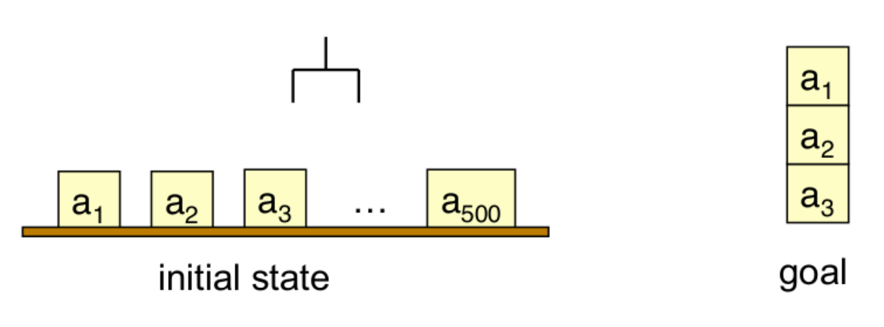


Branching Factor of Forward Search

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

$\Longrightarrow$ Forward-search can waste time trying lots of irrelevant actions

$\Longrightarrow$ Need a good heuristic to guide the search

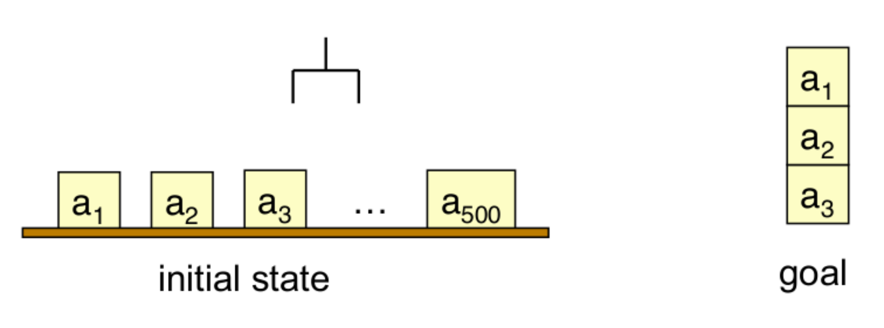


Branching Factor of Forward Search

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

$\Longrightarrow$ Forward-search can waste time trying lots of irrelevant actions

$\Longrightarrow$ Need a good heuristic to guide the search

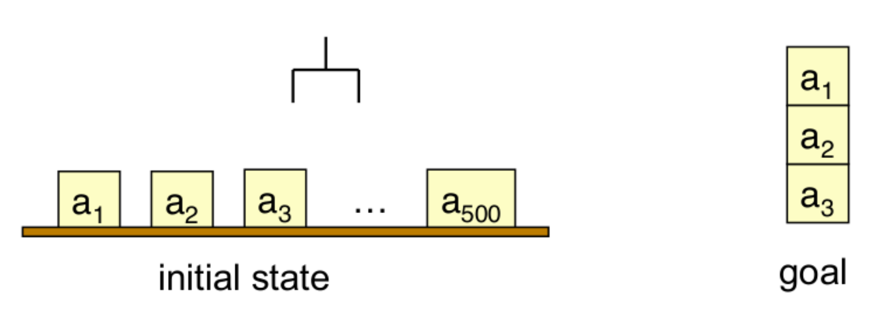


Branching Factor of Forward Search

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

$\Longrightarrow$ Forward-search can waste time trying lots of irrelevant actions

$\Longrightarrow$ 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

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

- Consider the goal $At(C_1, SFO) \land At(C_2, JFK)$
- Consider the partially-instantiated action:

  $Action(Unload(C_1, p', SFO),$
  $PRECOND : In(C_1, p') \land At(p', SFO) \land Cargo(C_1) \land Plane(p') \land Airport(SFO)$
  $EFFECT : At(C_1, SFO) \land \neg In(C_1, p'))$
- This produces the sub-goal g':

  $In(C_1, p') \land At(p', SFO) \land Cargo(C_1) \land Plane(p') \land Airport(SFO) \land At(C_2, JFK)$
- Represents states with all possible planes

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, \ldots$

Backward Search [cont.]

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

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

- Consider the goal $At(C_1, SFO) \land At(C_2, JFK)$
- Consider the partially-instantiated action:

  $Action(Unload(C_1, p', SFO),$

  $PRECOND : In(C_1, p') \land At(p', SFO) \land Cargo(C_1) \land Plane(p') \land Airport(SFO)$

  $EFFECT : At(C_1, SFO) \land \neg In(C_1, p'))$
- This produces the sub-goal g':

  $In(C_1, p') \land At(p', SFO) \land Cargo(C_1) \land Plane(p') \land Airport(SFO) \land At(C_2, JFK)$
- Represents states with all possible planes

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, ...$

Backward Search [cont.]

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

  $\Longrightarrow$ no need to produce a goal for every possible instantiation
- use the most general unifier $\Longrightarrow$ 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

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, \ldots$

Backward Search [cont.]

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

  $\Longrightarrow$ no need to produce a goal for every possible instantiation
- use the most general unifier $\Longrightarrow$ 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

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, \ldots$

Backward Search [cont.]

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

  $\Longrightarrow$ no need to produce a goal for every possible instantiation
- use the most general unifier $\Longrightarrow$ 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

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, \ldots$

Backward Search [cont.]

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

  $\Longrightarrow$ no need to produce a goal for every possible instantiation
- use the most general unifier $\Longrightarrow$ 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

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, \ldots$

Backward Search [cont.]

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

  $\Longrightarrow$ no need to produce a goal for every possible instantiation
- use the most general unifier $\Longrightarrow$ 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:
  $Action(Unload(C_1, p', SFO),$
  $\quad PRECOND: In(C_1, p') \wedge At(p', SFO) \wedge Cargo(C_1) \wedge Plane(p') \wedge Airport(SFO)$
  $\quad EFFECT: At(C_1, SFO) \wedge \neg In(C_1, p'))$
- 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

  $\Longrightarrow$ no need to produce a subgoal for every plane $P_1, P_2, P_3, \ldots$

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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

- Ex: consider the goal $At(C_1, SFO) \land At(C_2, JFK)$

is relevant (previous example)

is not relevant

is not consistent $\Longrightarrow$ is not relevant

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

  is relevant (previous example)

  is not relevant

  is not consistent $\Longrightarrow$ is not relevant

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- **Consistent action**: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

  is relevant (previous example)

  is not relevant

  is not consistent $\Longrightarrow$ is not relevant

+ B.S. typically keeps the branching factor lower than F.S.

- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

is relevant (previous example)

is not relevant

is not consistent $\Longrightarrow$ is not relevant

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

  is relevant (previous example)

  is not relevant

  is not consistent $\Longrightarrow$ is not relevant

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.
- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.

- B.S. reasons with state sets
  $\Longrightarrow$ 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
  - must not undo desired literals of the goal
- Consistent action: must not undo desired literals of the goal
  - inconsistent actions are also non-relevant

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

+ B.S. typically keeps the branching factor lower than F.S.

\- B.S. reasons with state sets
$\Longrightarrow$ makes it harder to come up with good heuristics

- Most planners work with forward search plus heuristics

Outline

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

  $\Longrightarrow$ 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 $\Longrightarrow$ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) $\Longrightarrow$ 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

  $\Longrightarrow$ 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 $\Longrightarrow$ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) $\Longrightarrow$ 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

  $\Longrightarrow$ 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 $\Longrightarrow$ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) $\Longrightarrow$ 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
  $\Longrightarrow$ 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 $\Longrightarrow$ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) $\Longrightarrow$ 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-depended criteria

Ignore (some) Preconditions Heuristics

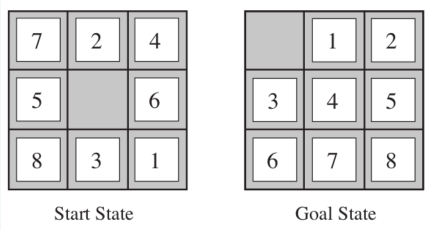
- Ignore all preconditions drops all preconditions from actions
  - every action is applicable in any state
  - any single goal literal can be satisfied in one step (or there is no solution)
  - fast, but over-optimistic
- Ignore some selected (less relevant) preconditions
  - relevance based on heuristics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

Ignore-Preconditions Heuristics: Example

Sliding tiles

$Action(Slide(t, s_1, s_2),$
$\quad PRECOND : Tile(t) \land Blank(s_2) \land On(t, s_1) \land Adjacent(s_1, s_2)$
$\quad EFFECT : On(t, s_2) \land Blank(s_1) \land \neg On(t, s_1) \land \neg Blank(s_2))$

- Remove the preconditions $Blank(s_2) \land Adjacent(s_1, s_2)$
  $\Longrightarrow$ we get the number-of-misplaced-tiles heuristics
- Remove the precondition $Blank(s_2)$
  $\Longrightarrow$ we get the Manhattan-distance heuristics



(Courtesy of Maria Simi, UniPI)

Ignore-Preconditions Heuristics: Example

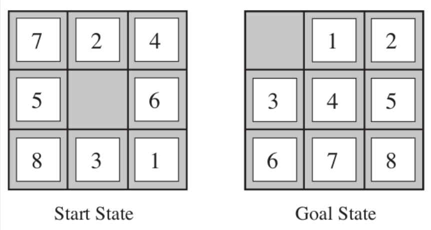
Sliding tiles

$Action(Slide(t, s_1, s_2),$
$\quad PRECOND : Tile(t) \land Blank(s_2) \land On(t, s_1) \land Adjacent(s_1, s_2)$
$\quad EFFECT : On(t, s_2) \land Blank(s_1) \land \neg On(t, s_1) \land \neg Blank(s_2))$

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

  $\Longrightarrow$ we get the number-of-misplaced-tiles heuristics
- Remove the precondition $Blank(s_2)$

  $\Longrightarrow$ we get the Manhattan-distance heuristics



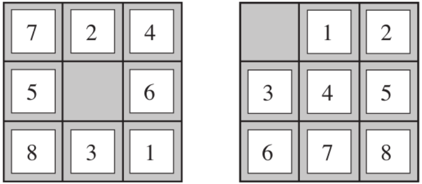
(Courtesy of Maria Simi, UniPI)

Ignore-Preconditions Heuristics: Example

Sliding tiles

$Action(Slide(t, s_1, s_2),$
$\quad PRECOND : Tile(t) \land Blank(s_2) \land On(t, s_1) \land Adjacent(s_1, s_2)$
$\quad EFFECT : On(t, s_2) \land Blank(s_1) \land \neg On(t, s_1) \land \neg Blank(s_2))$

- Remove the preconditions $Blank(s_2) \land Adjacent(s_1, s_2)$
  $\Longrightarrow$ we get the number-of-misplaced-tiles heuristics
- Remove the precondition $Blank(s_2)$
  $\Longrightarrow$ we get the Manhattan-distance heuristics



Start State Goal State

(Courtesy of Maria Simi, UniPI)

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,

  $\Longrightarrow$ 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,

  $\Longrightarrow$ 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,

  $\Longrightarrow$ 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,

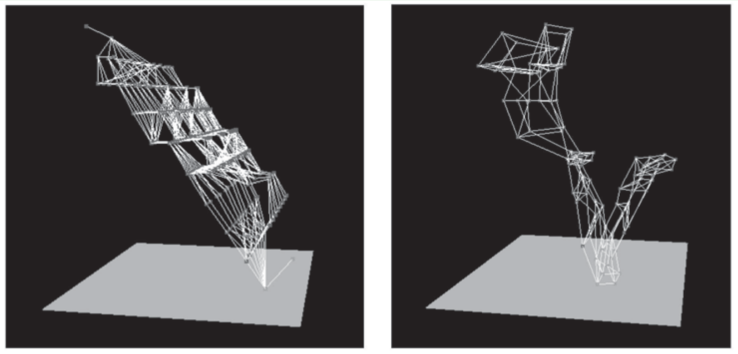
  $\Longrightarrow$ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

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

$\Longrightarrow$ No local minima, non dead-ends, non backtracking

$\Longrightarrow$ Search for the goal is straightforward for hill-climbing



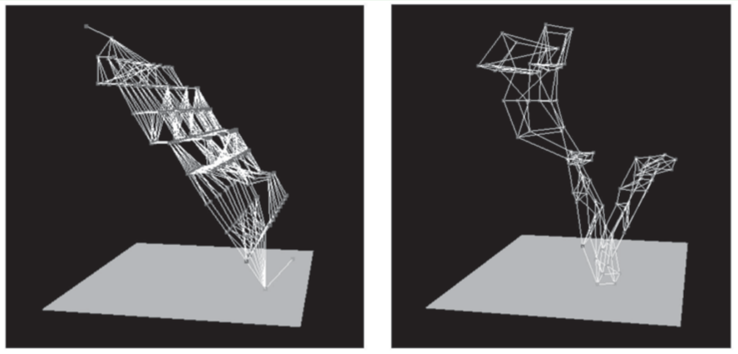
(© S. Russell & P. Norwig, AIMA)

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

$\Longrightarrow$ No local minima, non dead-ends, non backtracking

$\Longrightarrow$ 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 $\Longrightarrow$ 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
  $\Longrightarrow 10^{50} \cdot (50+10)^{200} \approx 10^{405}$ 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
  $\Longrightarrow 10^{5} \cdot (5+10)^{5} \approx 10^{11}$ states (*)
  - abstract solution shorter than ground solutions $\Longrightarrow$ 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 $\Longrightarrow$ 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
  $\Longrightarrow 10^{50} \cdot (50+10)^{200} \approx 10^{405}$ 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
  $\Longrightarrow 10^{5} \cdot (5+10)^{5} \approx 10^{11}$ states (*)
  - abstract solution shorter than ground solutions $\Longrightarrow$ 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 $\Longrightarrow$ 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
  $\Longrightarrow 10^{50} \cdot (50+10)^{200} \approx 10^{405}$ 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
  $\Longrightarrow 10^{5} \cdot (5+10)^{5} \approx 10^{11}$ states (*)
  - abstract solution shorter than ground solutions $\Longrightarrow$ 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 $\Longrightarrow$ 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
  $\Longrightarrow 10^{50} \cdot (50+10)^{200} \approx 10^{405}$ 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
  $\Longrightarrow 10^{5} \cdot (5+10)^{5} \approx 10^{11}$ states (*)
  - abstract solution shorter than ground solutions $\Longrightarrow$ 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 $\Longrightarrow$ 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
  $\Longrightarrow$ $10^{50} \cdot (50+10)^{200} \approx 10^{405}$ 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
  $\Longrightarrow$ $10^{5} \cdot (5+10)^{5} \approx 10^{11}$ states (*)
  - abstract solution shorter than ground solutions $\Longrightarrow$ 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

Outline

Generalities

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

$\Longrightarrow$ it can be used to derive an admissible heuristic $h(s)$

Generalities

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

$\Longrightarrow$ it can be used to derive an admissible heuristic $h(s)$

Generalities

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

$\Longrightarrow$ it can be used to derive an admissible heuristic $h(s)$

Generalities

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

$\Longrightarrow$ it can be used to derive an admissible heuristic $h(s)$

Generalities

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

$\Longrightarrow$ 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 inS_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 inS_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 inS_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 inS_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

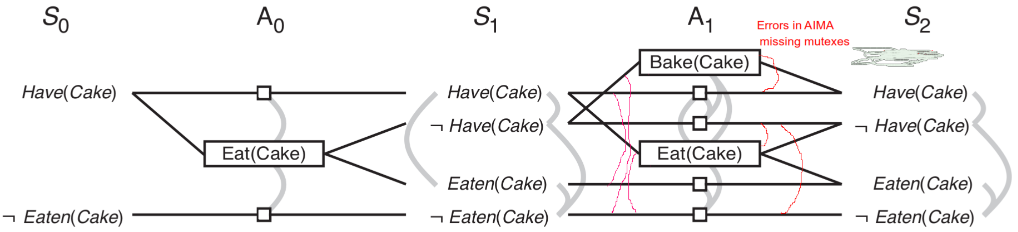
Deals with ground states and actions only

Planning Graph: Example

$Init(Have(Cake))$
$Goal(Have(Cake) \land Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg Have(Cake) \land Eaten(Cake))$
$Action(Bake(Cake)$
  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.

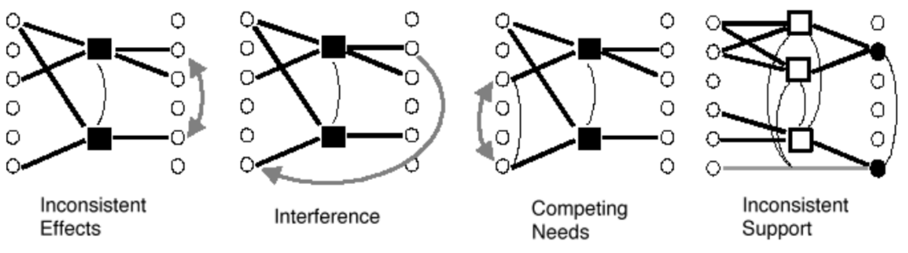
Rectangles indicate actions
Small squares persistence actions (**no-ops**)
Straight lines indicate preconditions and effects
Mutex links are shown as curved gray lines



(© S. Russell & P. Norwig, AIMA)

Mutex Computation

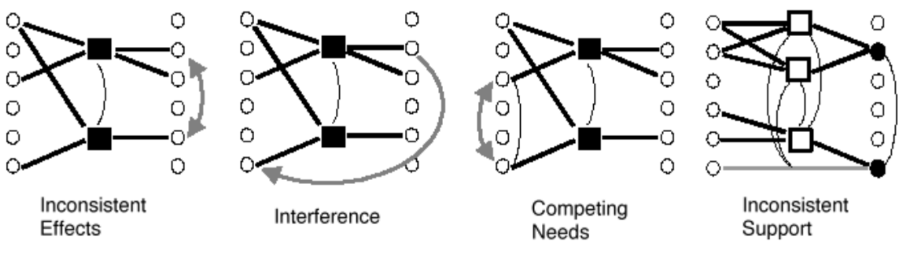
- Two actions at the same action-level have a mutex relation if
  - Inconsistent effects: an effect of one negates an effect of the other
  - Interference: one deletes a precondition of the other
  - Inconsistent preconditions (aka competing needs): they have mutually exclusive preconditions
- Otherwise they don't interfere with each other
  $\Longrightarrow$ 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

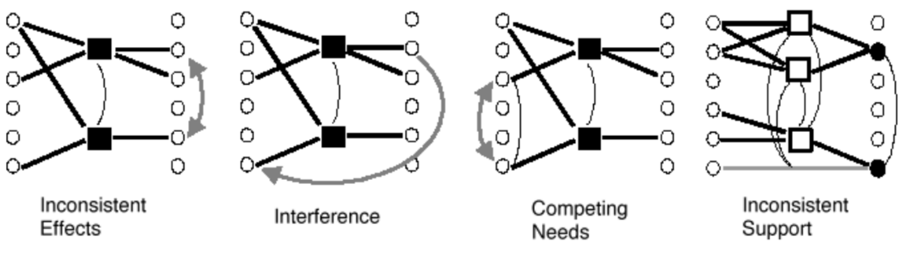
  $\Longrightarrow$ both may appear in a solution plan
- Two literals at the same state-level have a mutex relation if
  - inconsistent support: one is the negation of the other
  - all ways of achieving them (including no-ops) are pairwise mutex



Mutex Computation

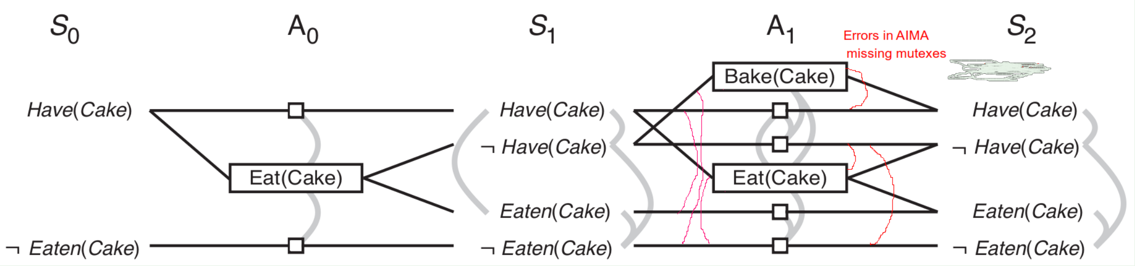
- Two actions at the same action-level have a mutex relation if
  - Inconsistent effects: an effect of one negates an effect of the other
  - Interference: one deletes a precondition of the other
  - Inconsistent preconditions (aka competing needs): they have mutually exclusive preconditions
- Otherwise they don't interfere with each other

  $\Longrightarrow$ 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



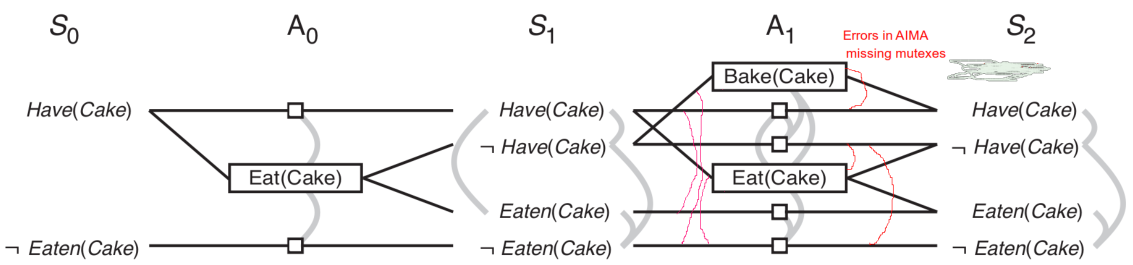
Mutex Computation: Example

- 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*)



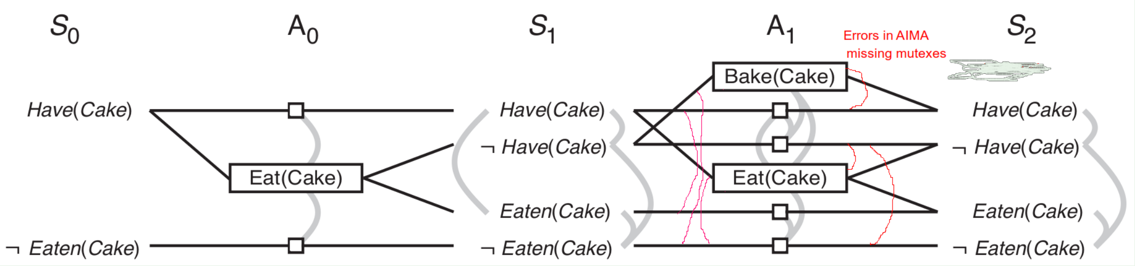
Mutex Computation: Example

- 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

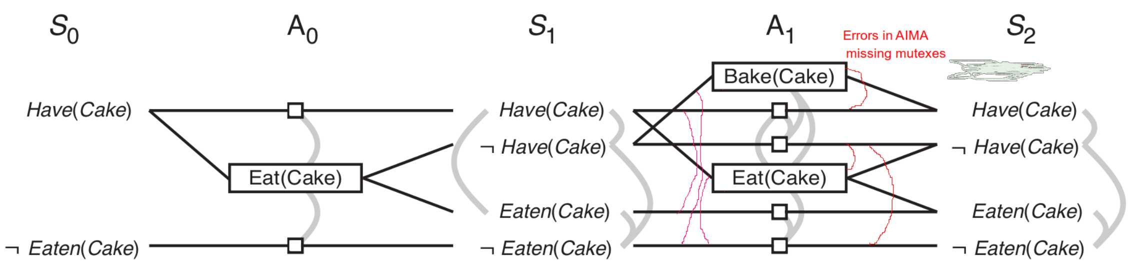
- Two actions at the same action-level have a mutex relation if
  - Inconsistent effects: an effect of one negates an effect of the other
    ex: persistence of *Have*(*Cake*), *Eat*(*Cake*) have competing effects
    ex: *Bake*(*Cake*), *Eat*(*Cake*) have competing effects
  - Interference: one deletes 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*)



(© S. Russell & P. Norwig, AIMA)

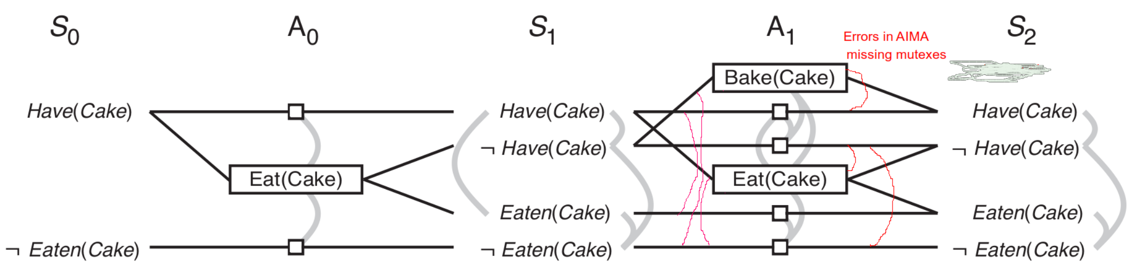
Mutex Computation: Example

- 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)*, ¬*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 $i = 0, 1, 2, ...$:

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 $i = 0, 1, 2, ...$:

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 $i = 0, 1, 2, ...$:

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 $i = 0, 1, 2, ...$:

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 $i = 0, 1, 2, ...$:

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
  $\Longrightarrow$ we eventually reach a level where they stabilize
- Mutexes decrease monotonically (and cannot become less than zero)
  $\Longrightarrow$ they too eventually must level off

$\Longrightarrow$ 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
$\Longrightarrow$ 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 $O(n(a+l)^2)$

$\Longrightarrow$ The process of constructing the planning graph is very fast

  - does not require choosing among actions

Outline

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

  $\Longrightarrow$ 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_i 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

  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)

  $\Longrightarrow$ better estimate

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

  $\Longrightarrow$ 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_i 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

  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)

  $\Longrightarrow$ better estimate

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

  $\Longrightarrow$ 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_i 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

  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)

  $\Longrightarrow$ better estimate

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

  $\Longrightarrow$ 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_i 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

  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)

  $\Longrightarrow$ better estimate

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

  $\Longrightarrow$ 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_i 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

  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)

  $\Longrightarrow$ better estimate

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
  $\Longrightarrow$ 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_i 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
  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)
  $\Longrightarrow$ better estimate

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
  $\Longrightarrow$ 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_i 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
  $\Longrightarrow$ inaccurate estimate
- Serialization: enforcing only one action per level (adding mutex)
  $\Longrightarrow$ 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

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

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

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 ⟨*goal*, *level*⟩ 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 ← INITIAL-PLANNING-GRAPH(*problem*)
  goals ← CONJUNCTS(*problem*.GOAL)
  nogoods ← an empty hash table
  for $t = 0$ **to** ∞ **do**
typo in AIMA book
    if *goals* all non-mutex in S_t of *graph* **then**
      solution ← 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 ← 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 ⟨*goal*, *level*⟩ 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 ← INITIAL-PLANNING-GRAPH(*problem*)
  goals ← CONJUNCTS(*problem*.GOAL)
  nogoods ← an empty hash table
  for $t = 0$ **to** ∞ **do**
typo in AIMA book
    if *goals* all non-mutex in S_t of *graph* **then**
      solution ← 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 ← 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 ← INITIAL-PLANNING-GRAPH(*problem*)
  goals ← CONJUNCTS(*problem*.GOAL)
  nogoods ← an empty hash table
  for $t = 0$ **to** ∞ **do**
typo in AIMA book
    if *goals* all non-mutex in S_t of *graph* **then**
      solution ← EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)
      if *solution* ≠ *failure* **then return** *solution*
    if *graph* and *nogoods* have both leveled off **then return** *failure*
    graph ← 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 ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for t = 0 to ∞ do
typo in  if goals all non-mutex in S_t of graph then
AIMA        solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
book        if solution ≠ failure then return solution
     if graph and nogoods have both leveled off then return failure
     graph ← EXPAND-GRAPH(graph, problem)
```

(© S. Russell & P. Norwig, AIMA)

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 ← INITIAL-PLANNING-GRAPH(*problem*)
goals ← CONJUNCTS(*problem*.GOAL)
nogoods ← an empty hash table
for $t = 0$ **to** ∞ **do**
typo in AIMA book
if *goals* all non-mutex in S_t of *graph* **then**
solution ← 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 ← EXPAND-GRAPH(*graph*, *problem*)

(© S. Russell & P. Norwig, AIMA)

[Recall] Example: Spare Tire Problem

$$
\begin{aligned}
&Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))\\
&Goal(At(Spare, Axle))\\
&Action(Remove(obj, loc),\\
&\quad \textsc{Precond}: At(obj, loc)\\
&\quad \textsc{Effect}: \neg At(obj, loc) \land At(obj, Ground))\\
&Action(PutOn(t, Axle),\\
&\quad \textsc{Precond}: Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)\\
&\quad \textsc{Effect}: \neg At(t, Ground) \land At(t, Axle))\\
&Action(LeaveOvernight,\\
&\quad \textsc{Precond}:\\
&\quad \textsc{Effect}: \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)\\
&\qquad\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))
\end{aligned}
$$

(© 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.)

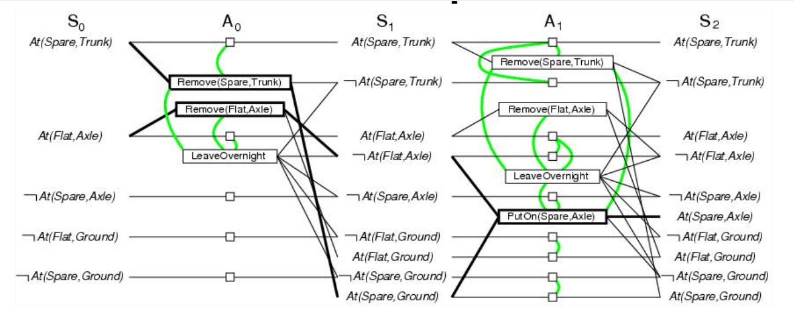
One solution: [*Remove*(*Flat*, *Axle*), *Remove*(*Spare*, *Trunk*), *PutOn*(*Spare*, *Axle*)]

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

  $\Longrightarrow$ no need to call EXTRACT-SOLUTION
- Graph and nogoods not leveled off $\Longrightarrow$ invoke EXPAND-GRAPH



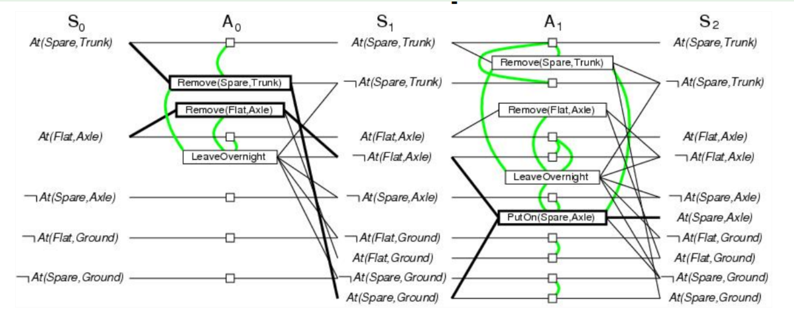
(© S. Russell & P. Norwig, AIMA) (inter-fluent mutexes omitted for readability)

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

  $\Longrightarrow$ no need to call EXTRACT-SOLUTION
- Graph and nogoods not leveled off $\Longrightarrow$ invoke EXPAND-GRAPH

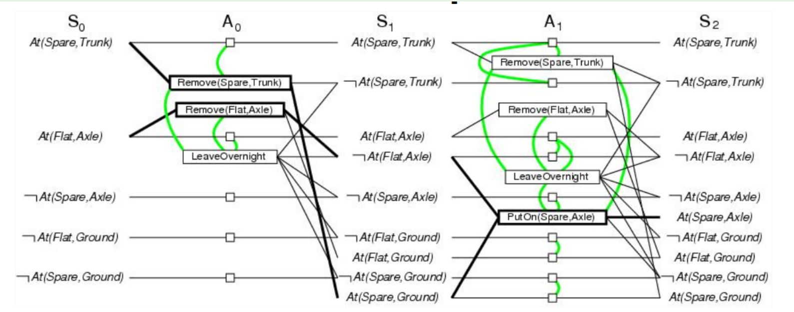


(© S. Russell & P. Norwig, AIMA) (inter-fluent mutexes omitted for readability)

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!



(© S. Russell & P. Norwig, AIMA) (inter-fluent mutexes omitted for readability)

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

  $\Longrightarrow$ much faster than unrestricted planning
- (if P.G. not serialized) may produce partial order plans

  $\Longrightarrow$ 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

  $\Longrightarrow$ much faster than unrestricted planning
- (if P.G. not serialized) may produce partial order plans

  $\Longrightarrow$ 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

  $\Longrightarrow$ much faster than unrestricted planning
- (if P.G. not serialized) may produce partial order plans

  $\Longrightarrow$ 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

  $\Longrightarrow$ much faster than unrestricted planning
- (if P.G. not serialized) may produce partial order plans

  $\Longrightarrow$ 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

  $\Longrightarrow$ much faster than unrestricted planning
- (if P.G. not serialized) may produce partial order plans

  $\Longrightarrow$ 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)

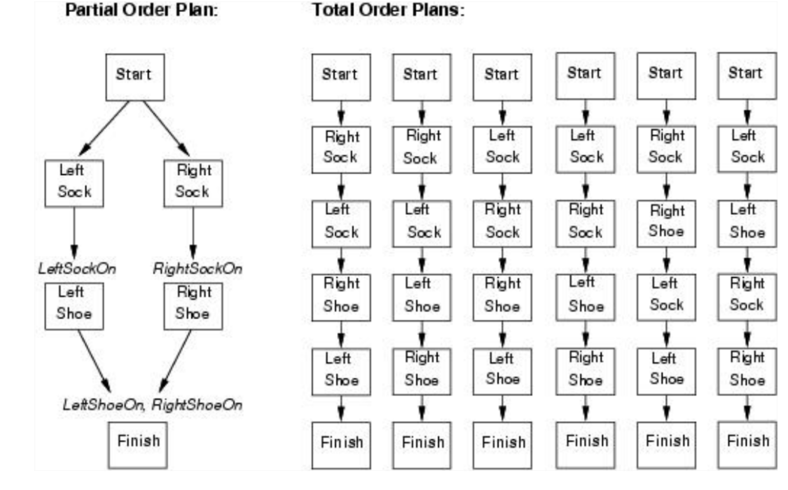
Partial-Order Plans

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



(Courtesy of Michela Milano, UniBO)

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
    $\Longrightarrow$ we eventually reach a level where they stabilize
  - Mutexes and no-goods decrease monotonically (and cannot become less than zero)
    $\Longrightarrow$ they too eventually must level off

  $\Longrightarrow$ 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
    $\Longrightarrow$ 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
    $\Longrightarrow$ we eventually reach a level where they stabilize
  - Mutexes and no-goods decrease monotonically (and cannot become less than zero)
    $\Longrightarrow$ they too eventually must level off

  $\Longrightarrow$ 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
    $\Longrightarrow$ 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
    $\Longrightarrow$ we eventually reach a level where they stabilize
  - Mutexes and no-goods decrease monotonically (and cannot become less than zero)
    $\Longrightarrow$ they too eventually must level off

  $\Longrightarrow$ 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
    $\Longrightarrow$ 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

Outline

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 ← TRANSLATE-TO-SAT(init, transition, goal, t)
     model ← 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

$\Longrightarrow$ 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

$\Longrightarrow$ 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: $\langle At(P_1, SFO), 3\rangle \Longrightarrow At\_P_1\_SFO\_3$
    - ex: $\langle Fly(P_1, SFO, JFK), 3\rangle \Longrightarrow Fly\_P_1\_SFO\_JFK\_3$
  - returns propositional formula: $Init^0 \land (\bigwedge_{i=1}^{t-1} Transition^{i,i+1}) \land Goal^t$
- $Init^0$ and $Goal^t$: conjunctions of literals at step 0 and t resp.
  - ex: $Init^0$: $At\_P_1\_SFO\_0 \land At\_P_2\_JFK\_0$ $(\land \neg At\_P_1\_JFK \land \neg At\_P_2\_SFO \land ...)$
  - ex: $Goal^3$: $At\_P_1\_JFK\_3 \land At\_P_2\_SFO\_3$
- $Transition^{i,i+1}$: encodes transition from steps i to $i+1$
  - Actions: $Action^i \rightarrow (Precond^i \land Effects^{i+1})$
    ex: $Fly\_P_1\_SFO\_JFK\_2 \rightarrow (At\_P_1\_SFO\_2 \land At\_P_1\_JFK\_3)$
  - No-Ops: for each fluent F and step i:

$$F^{i+1} \leftrightarrow \bigvee_k ActionCausingF_k^i \lor (F^i \land \bigwedge_j \neg ActionCausingNotF_j^i)$$

  - Mutex constraints: $\neg Action_1^i \lor \neg Action_2^i$
    ex: $\neg Fly\_P_1\_SFO\_JFK\_2 \lor \neg Fly\_P_1\_SFO\_Newark\_2$
  - 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: $\langle At(P_1, SFO), 3\rangle \Longrightarrow At\_P_1\_SFO\_3$
    - ex: $\langle Fly(P_1, SFO, JFK), 3\rangle \Longrightarrow Fly\_P_1\_SFO\_JFK\_3$
  - 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: $Init^0$: $At\_P_1\_SFO\_0 \wedge At\_P_2\_JFK\_0$ $(\wedge \neg At\_P_1\_JFK \wedge \neg At\_P_2\_SFO \wedge ...)$
  - 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

Planning as SAT Solving [cont.]

- TRANSLATE-TO-SAT(INIT,TRANSITION, GOAL, T):
  - ground fluents & actions at each step are propositionalized
    - ex: $\langle At(P_1, SFO), 3\rangle \Longrightarrow At\_P_1\_SFO\_3$
    - ex: $\langle Fly(P_1, SFO, JFK), 3\rangle \Longrightarrow Fly\_P_1\_SFO\_JFK\_3$
  - 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: $Init^0$: $At\_P_1\_SFO\_0 \wedge At\_P_2\_JFK\_0$ $(\wedge\neg At\_P_1\_JFK \wedge \neg At\_P_2\_SFO \wedge ...)$
  - 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: $Action^i \to (Precond^i \wedge Effects^{i+1})$
    ex: $Fly\_P_1\_SFO\_JFK\_2 \to (At\_P_1\_SFO\_2 \wedge At\_P_1\_JFK\_3)$
  - 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