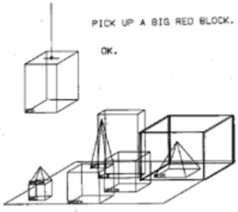
DD2380 Artificial Intelligence

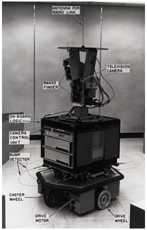
Lecture 12
Classical Planning

Jana Tumova

Planning

Deliberating a plan of action to achieve one's goal





Real-world planning

- Limited resources: time, cost, capacity,...
- Uncertainty
- Multiple agents
- Different criteria: optimality,...
- Robotics: dynamical constraints
- Integration into context, integration with other AI methods

Real-world planning

- Limited resources: time, cost, capacity,. . .
- Uncertainty
- Multiple agents
- Different criteria: optimality,. . .
- Robotics: dynamical constraints
- Integration into context, integration with other AI methods

Today's lecture: only fully observable, deterministic, static, environments.

Planning problem

Planning problem:

- Initial state
- Actions available in a state $ACTIONS(s)$
- Results of applying action $RESULT(s, a)$
- Goal test

Planning vs. problem solving

- Problem solving (search and games):
  - Explicit atomic representations
  - Need good domain-specific heuristics
- Classical planning:
  - Factored representation
  - Domain-independent algorithms

Challenges

- How do we represent a planning problem?
- How do we solve a planning problem?

Challenges

- How do we represent a planning problem?
  - PDDL, STRIPS, ...
- How do we solve a planning problem?

Planning Domain Definition Language (PDDL)

A careful balance between expressivity and simplicity

States

- $At(P, SFO)$
- $At(P, Arlanda) \wedge Plane(P) \wedge Loaded(Cargo, P)$

Actions (think of them as universally quantified)

$$
\begin{array}{l}
Action(Fly(p, from, to), \\
\quad \text{PRECOND:} At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\
\quad \text{EFFECT:} \neg At(p, from) \wedge At(p, to))
\end{array}
$$

Example: Air Cargo Transport in PDDL

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

PDDL States

- State is a conjunction of ground, functionless atomic fluents
  - $At(P, Arlanda) \wedge Plane(P) \wedge Loaded(Cargo, P)$, but
  - Not $\neg At(P, Bromma)$,
  - Not $At(x, y)$,
  - Not $At(P, TheHomeAirport(SAS))$
- Database semantics: Fluents that are not mentionad are false
- Two equivalent viewpoints:
  - Conjunction of fluents: logical inference
  - Set of fluents: set operations

PDDL Action Schemes

Actions and their results are represented through action schemes

- Action name with the list of all used variables
- Precondition: a conjunction of literals saying when an action is *applicable* in a state s, namely if s entails the precondition

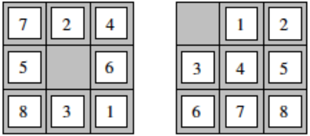
$$ACTIONS(s) = \{a \mid s \models PRECOND(a)\}.$$

- Effect: a conjunction of literals representing the literals that need to be removed and added

$$RESULT(s, a) = (s \setminus DEL(a)) \cup ADD(a),$$

where $DEL(a)$ are the fluents that appear as negative literals in the effect and $ADD(a)$ are the fluents that appear as positive ones.

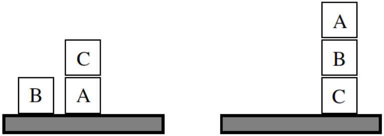
Q: What are the initial state and the goal test?



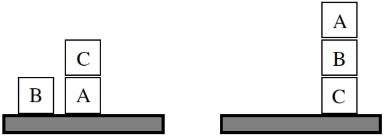
Start State　Goal State

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

Block World in PDDL

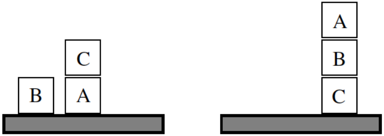


Block World in PDDL



$Init(On(A, Table) \land On(B, Table) \land On(C, A)$
$\quad \land\ Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$
$\qquad (b \neq x) \land (b \neq y) \land (x \neq y),$
$\quad$ EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$
$Action(MoveToTable(b, x),$

Block World in PDDL



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

Challenges

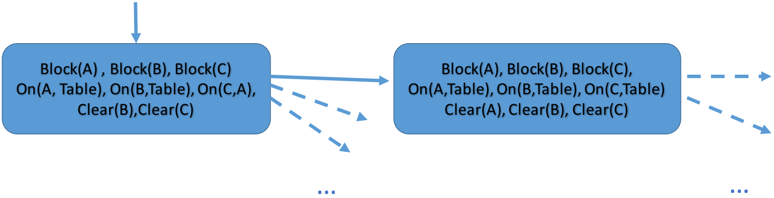
- How do we represent a planning problem?
    - PDDL, STRIPS, ...
- How do we solve a planning problem?

Challenges

- How do we represent a planning problem?
  - PDDL, STRIPS, ...
- How do we solve a planning problem?
  - Via forward search and backward search with heuristics

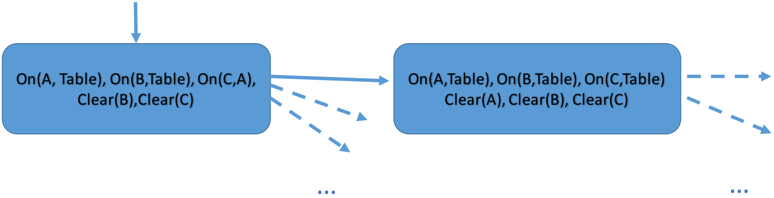
PDDL Planning Problem as a State Space Search

- Description of a planning problem defines a search problem
- States are truth assignments to fluents, actions and results define the transitions



PDDL Planning Problem as a State Space Search

- Description of a planning problem defines a search problem
- States are truth assignments to fluents, actions and results define the transitions



Forward (progression) search

- As expected
  - Start at the initial state
  - Explore applicable actions
- Properties
  - Large branching factor, often explores irrelevant actions
  - Needs a good heuristic, ideally domain-independent

Example: Air Cargo Transport in PDDL

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

Heuristics

In both progression and regression search, we need good a heuristic.

Domain-independent heuristics

- Any planning problem instance
- Define a relaxed, easier problem and a heuristic as a solution to this easier problem
- Use the internal structure of a factored representation of the state space

- Ignore preconditions
- Ignore delete lists
- State abstraction
- Decomposition
- Planning graph

Domain-independent heuristics

- Any planning problem instance
- Define a relaxed, easier problem and a heuristic as a solution to this easier problem
- Use the internal structure of a factored representation of the state space

- Ignore preconditions
- Ignore delete lists
- State abstraction
- Decomposition
- Planning graph

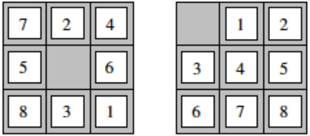
Ignore preconditions

Ignore all preconditions

- Every action becomes applicable in every state
- The number of edges in the graph increases
- The number of steps to solve the relaxed problem is almost the number of unsatisfied fluents in the goal
  - Some actions may achieve multiple goals
  - Some actions may undo the effects of others

Ignore some preconditions

Example: Ignore preconditions

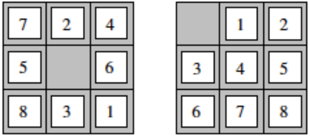


Start State   Goal State

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

$h_1(n)$: number of the misplaced tiles

Example: Ignore preconditions



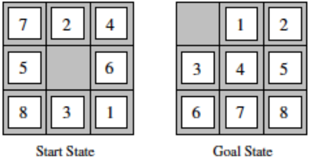
Start State Goal State

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

$h_1(n)$: number of the misplaced tiles

- The relaxed problem assumed we can transfer any tile anywhere
- $PRECOND : On(t, s_1) \wedge Tile(t) \cancel{\wedge Blank(s_2) \wedge Adjacent(s_1, s_2)}$

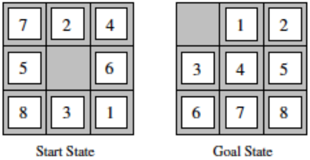
Q: Match ignoring preconditions with the heuristic



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

$h_2(n)$: sum of the distances of the tiles from the goal position

Q: Match ignoring preconditions with the heuristic



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

$h_2(n)$: sum of the distances of the tiles from the goal position

- The relaxed problem assumed we can transfer any tile to an adjacent cell

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

Domain-independent heuristics

- Any planning problem instance
- Define a relaxed, easier problem and a heuristic as a solution to this easier problem
- Use the internal structure of a factored representation of the state space

- Ignore preconditions
- Ignore delete lists
- State abstraction
- Decomposition
- Planning graph

Ignore Delete Lists

- Remove all negative literals from effects
- No action will undo progress by another action towards the goal
- The number of edges in the graph increases

Domain-independent Heuristics

- Any planning problem instance
- Define a relaxed, easier problem and a heuristic as a solution to this easier problem
- Use the internal structure of a factored representation of the state space

- Ignore preconditions
- Ignore delete lists
- State abstraction
- Decomposition
- Planning graph

State Abstraction

- Many-to-one mapping from states in the ground representation to the abstract representation
- The number of states decreases
- Ignore some fluents

Domain-independent Heuristics

- Any planning problem instance
- Define a relaxed, easier problem and a heuristic as a solution to this easier problem
- Use the internal structure of a factored representation of the state space

- Ignore preconditions
- Ignore delete lists
- State abstraction
- Decomposition
- Planning graph

Decomposition

- $G = G_1 \wedge \ldots \wedge G_n$
- Instead of problem P, we solve problems $P_1, \ldots, P_n$
- We can use $\max_i COST(P_i)$ as a heuristic

Decomposition

- $G = G_1 \wedge \ldots \wedge G_n$
- Instead of problem P, we solve problems $P_1, \ldots, P_n$
- We can use $\max_i COST(P_i)$ as a heuristic

- Q: Can we use $COST(P_1) + \ldots + COST(P_n)$ as a heuristic?

Decomposition

- $G = G_1 \wedge \ldots \wedge G_n$
- Instead of problem P, we solve problems $P_1, \ldots, P_n$
- We can use $\max_i COST(P_i)$ as a heuristic

- Q: Can we use $COST(P_1) + \ldots + COST(P_n)$ as a heuristic?

- Q: What generally happens if we use a non-admissible heuristic?

Domain-independent Heuristics

- Any planning problem instance
- Define a relaxed, easier problem and a heuristic as a solution to this easier problem
- Use the internal structure of a factored representation of the state space

- Ignore preconditions
- Ignore delete lists
- State abstraction
- Decomposition
- Planning graph

Planning Graph

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

Planning Graph

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

State level

S_0

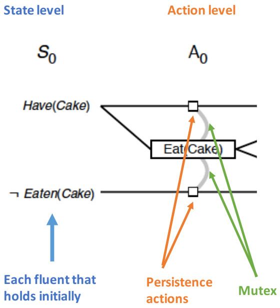
Have(Cake)

$\neg$ *Eaten(Cake)*

Each fluent that holds initially

Planning Graph

Init(*Have*(*Cake*))
Goal(*Have*(*Cake*) ∧ *Eaten*(*Cake*))
Action(*Eat*(*Cake*)
  PRECOND: *Have*(*Cake*)
  EFFECT: ¬ *Have*(*Cake*) ∧ *Eaten*(*Cake*))
Action(*Bake*(*Cake*)
  PRECOND: ¬ *Have*(*Cake*)
  EFFECT: *Have*(*Cake*))

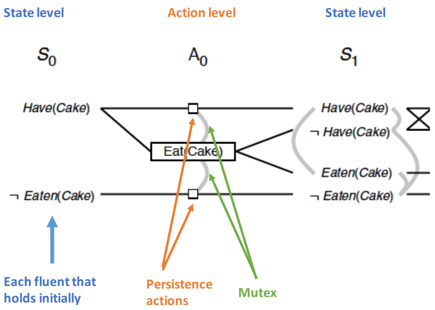


Mutex:

- Inconsistent effects: one action negates the effect of the other
- Interference: one of the effects of one action is the negation of a precondition of the other
- Competing needs: one of the preconditions of one action is the negation of a precondition of the other

Planning Graph

Init(*Have*(*Cake*))
Goal(*Have*(*Cake*) ∧ *Eaten*(*Cake*))
Action(*Eat*(*Cake*)
  PRECOND: *Have*(*Cake*)
  EFFECT: ¬ *Have*(*Cake*) ∧ *Eaten*(*Cake*))
Action(*Bake*(*Cake*)
  PRECOND: ¬ *Have*(*Cake*)
  EFFECT: *Have*(*Cake*))

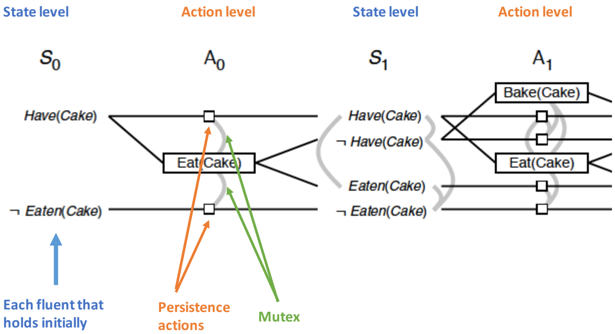


Mutex for literals:

- Negation
- If each possible pair of actions that could achieve the two literals is mutually exclusive

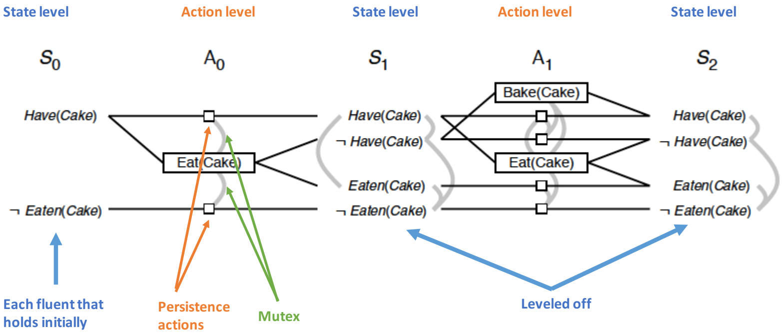
Planning Graph

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



Planning Graph

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



Planning Graph

- Directed graph with alternating state and action levels
- Roughly,
  - S_i level contains literals that could hold at time i
  - A_i level contains actions that could have their precoditions satisfied at time i.
- Persistence actions: a literal can persist if no action negates it
- Mutex:
  - Inconsistent effects: one action negates effect of the other
  - Interference: effect of one action negates a precondition of the other
  - Competing needs: the precondition of one action is mutually exclusive with a precondition of the other

Planning Graph Properties

- Polynomial in the size of the planning problem
- A literal never appears too late, but might appear too early
- If a goal literal does not appear in the final level of the graph, the problem is unsolvable

- We can use it for designing an independent-domain heuristic:
  - Max-level heuristic: admissible, but not always accurate
  - Level-sum heuristic: generally inadmissible, but works well in practice
  - Set-level heuristic: find the level at which all the goal literals appear without being mutually exclusive; admissible and works well

Challenges

- How do we represent a planning problem?
  - PDDL, STRIPS, ...
- How do we solve a planning problem?
  - Via forward search and backward search with heuristics

Challenges

- How do we represent a planning problem?
  - PDDL, STRIPS, ...
- How do we solve a planning problem?
  - Via forward search and backward search with heuristics
  - Via GRAPHPlan

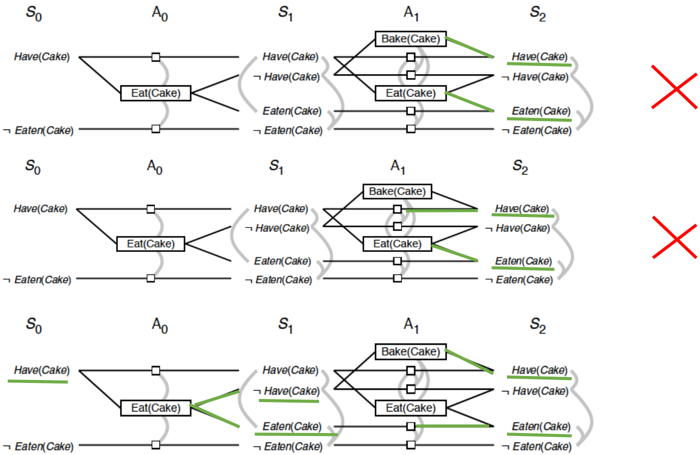
GRAPHPlan

- Using the planning graph to extract a plan directly instead of using it to design a heuristic for search
- EXTRACT-SOLUTION either through CSP or through backward search in the planning graph, not in the state space
- If EXTRACT-SOLUTION does not find a plan, we store (*level*, *goals*) in *nogoods*

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

  graph ← INITIAL-PLANNING-GRAPH(*problem*)
  goals ← CONJUNCTS(*problem*.GOAL)
  nogoods ← an empty hash table
  for *tl* = 0 **to** ∞ **do**
    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*)

GRAPHPlan



S_0
A_0
S_1
A_1
S_2
Have(Cake)
¬ Eaten(Cake)
Eat(Cake)
Have(Cake)
¬ Have(Cake)
Eaten(Cake)
¬ Eaten(Cake)
Bake(Cake)
Eat(Cake)
Have(Cake)
¬ Have(Cake)
Eaten(Cake)
¬ Eaten(Cake)

GRAPHPlan Properties

- Literals increase monotonically
- Actions increase monotonically
- Mutexes decrease monotonically
- No-goods decrease monotonically

- It is not enough to level-off the graph
- Termination when mutexes and no-goods have both leveled off

Challenges

- How do we represent a planning problem?
    - PDDL, STRIPS, ...
- How do we solve a planning problem?
    - Via forward search and backward search with heuristics
    - Via GRAPHPlan
    - As refinement of partially ordered plans
    - As Boolean satisfiability
    - As first-order logical deduction: situation calculus
    - As constraint satisfaction

Think

- 1. What is the point of using planning as opposed to problem solving?
- 2. What are the limitations of PDDL as we saw it today?

Final remarks

Where to learn more

- Artificial Intelligence: A Modern Approach by Stuart J. Russell and Peter Norvig, chapter 10 + the udacity course Intro to AI
- Automated Planning and Acting by Dana S. Nau, Malik Ghallab, and Paolo Traverso

Tools

- http://www.fast-downward.org