

Artificial Intelligence Planning

State-Space Search

Artificial Intelligence Planning

- **State-Space Search**

Forward State-Space Search Algorithm

```
function fwdSearch( $O, s_i, g$ )  
   $state \leftarrow s_i$   
   $plan \leftarrow \langle \rangle$   
  loop  
    if  $state.satisfies(g)$  then return  $plan$   
     $applicables \leftarrow \{\text{ground instances from } O \text{ applicable in } state\}$   
    if  $applicables.isEmpty()$  then return failure  
     $action \leftarrow applicables.chooseOne()$   
     $state \leftarrow \gamma(state, action)$   
     $plan \leftarrow plan \bullet \langle action \rangle$ 
```

Forward State-Space Search Algorithm

Overview

- States with Internal Structure
- Operators with Structure
- Planning Domains and Problems
- Forward State-Space Search
- Backward State-Space Search

Overview

➤ States with Internal Structure

- now: the STRIPS representation for world states

• Operators with Structure

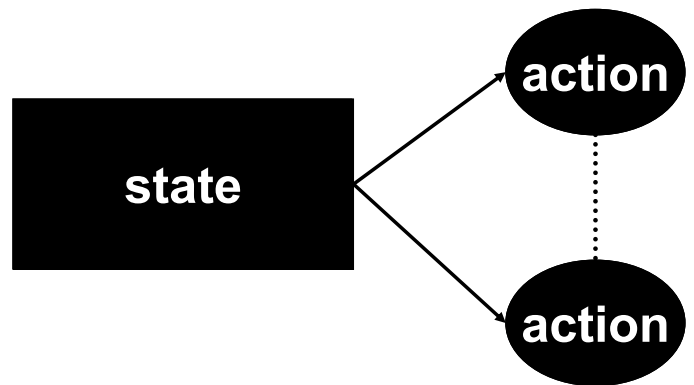
• Planning Domains and Problems

• Forward State-Space Search

• Backward State-Space Search

States as Black Boxes

- access to states in search problems:
 - goal test
 - applicable actions
 - successor states
 - equality test
 - hash function
 - heuristic estimate
- aim: reason about states and actions



States as Black Boxes

- **access to states in search problems:**
 - goal test
 - applicable actions
 - successor states
 - equality test
 - hash function
 - heuristic estimate
 - no other access: states and actions are black boxes
- **aim: reason about states and actions**
 - need to know about internal structure of states and actions

Objects in the DWR Domain

- robots {robot1, robot2, ...}:
 - container carrier carts for one container
 - can move between adjacent locations
- cranes {crane1, crane2, ...}:
 - belongs to a single location
 - can move containers between robots and piles at same location
- containers {cont1, cont2, ...}:
 - stacked in some pile on some pallet, loaded onto robot, or held by crane
- locations {loc1, loc2, ...}:
 - storage area, dock, docked ship, or parking or passing area
- piles {pile1, pile2, ...}:
 - attached to a single location
 - pallet at the bottom, possibly with containers stacked on top of it
- pallet:
 - at the bottom of a pile

Objects in the DWR Domain

• robots {robot1, robot2, ...}:

- **container carrier carts for one container**
- **can move between adjacent locations**
- can be loaded/unloaded by cranes at the same location
- at most one robot at one location at any one time

• cranes {crane1, crane2, ...}:

- **belongs to a single location**
- **can move containers between robots and piles at same location**
- can load/unload containers onto/from robots, or take/put containers from/onto top of piles, all at the same location
- possibly multiple cranes per location

• containers {cont1, cont2, ...}:

- **stacked in some pile on some pallet, loaded onto robot, or held by crane**
- pile on pallet means bottom container on pallet and other on container below them

• locations {loc1, loc2, ...}:

- **storage area, dock, docked ship, or parking or passing area**
- do not necessarily have piles, e.g. parking or passing areas

• piles {pile1, pile2, ...}:

- **attached to a single location**
- locations with piles must also have cranes
- **pallet at the bottom, possibly with containers stacked on top of it**
- zero or more (unlimited number of) containers in a pile
- possibly multiple piles per location

• pallet:

- **at the bottom of a pile**
- need only one symbol as pallets are identified by their pile

Example: DWR Types in PDDL Syntax

```
(define (domain dock-worker-robot)

  (:requirements :strips :typing )

  (:types
    location ;there are several connected locations
    pile     ;is attached to a location,
             ;it holds a pallet and a stack of containers
    robot    ;holds at most 1 container,
             ;only 1 robot per location
    crane    ;belongs to a location to pickup containers
    container )

  ...)
```

Example: DWR Types in PDDL Syntax

•(define (domain dock-worker-robot)

- defines a named domain (running example)

•(:requirements :strips :typing)

- simple requirements: STRIPS actions and typing (to make domain more readable)

•(:types

•location ;there are several connected locations

- first type: set of objects that belong to this type
- note: semicolon is beginning of comment

•pile ;is attached to a location, it holds a pallet and a stack of containers

•robot ;holds at most 1 container, only 1 robot per location

•crane ;belongs to a location to pickup containers

•container)

•...)

- remaining domain omitted here

Example: DWR Predicates (PDDL)

```
(:predicates
  (adjacent ?l1 ?l2 - location) ;location ?l1 is adjacent to ?l2
  (attached ?p - pile ?l - location) ;pile ?p attached to location ?l
  (belong ?k - crane ?l - location) ;crane ?k belongs to location ?l

  (at ?r - robot ?l - location) ;robot ?r is at location ?l
  (occupied ?l - location) ;there is a robot at location ?l
  (loaded ?r - robot ?c - container) ;robot ?r is loaded with container ?c
  (unloaded ?r - robot) ;robot ?r is empty

  (holding ?k - crane ?c - container) ;crane ?k is holding a container ?c
  (empty ?k - crane) ;crane ?k is empty

  (in ?c - container ?p - pile) ;container ?c is within pile ?p
  (top ?c - container ?p - pile) ;container ?c is on top of pile ?p
  (on ?c1 - container ?c2 - container) ;container ?c1 is on container ?c2
)
```

Example: DWR Predicates

•(:predicates

•(adjacent ?l1 ?l2 - location) ;location ?l1 is adjacent to ?l2

- predicate name: adjacent
- two arguments represented by variables: ?l1 and ?l2
- type of both variables must be location

•(attached ?p - pile ?l - location) ;pile ?p attached to location ?l

- arguments of two different types

•(belong ?k - crane ?l - location) ;crane ?k belongs to location ?l

•(at ?r - robot ?l - location) ;robot ?r is at location ?l

•(occupied ?l - location) ;there is a robot at location ?l

•(loaded ?r - robot ?c - container) ;robot ?r is loaded with container ?c

•(unloaded ?r - robot) ;robot ?r is empty

•(holding ?k - crane ?c - container) ;crane ?k is holding a container ?c

•(empty ?k - crane) ;crane ?k is empty

•(in ?c - container ?p - pile) ;container ?c is within pile ?p

•(top ?c - container ?p - pile) ;container ?c is on top of pile ?p

•(on ?c1 - container ?c2 - container) ;container ?c1 is on container ?c2

- always use comments!

•)

States in the STRIPS Representation

- Let \mathcal{L} be a first-order language with finitely many predicate symbols, finitely many constant symbols, and no function symbols.
- A state in a STRIPS planning domain is a set of ground atoms of \mathcal{L} .
 - (ground) atom p holds in state s iff $p \in s$
 - s satisfies a set of (ground) literals g (denoted $s \models g$) if:
 - every positive literal in g is in s and
 - every negative literal in g is not in s .

States in the STRIPS Representation

• Let \mathcal{L} be a first-order language with finitely many predicate symbols, finitely many constant symbols, and no function symbols.

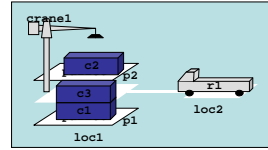
- terms in \mathcal{L} are either constants or variables
- extensions of \mathcal{L} will follow later

• A state in a STRIPS planning domain is a set of ground atoms of \mathcal{L} .

- note: number of different states is finite
- (ground) atom p holds in state s iff $p \in s$
 - closed-world assumption
- s satisfies a set of (ground) literals g (denoted $s \models g$) if:
 - literals: atoms and negated atoms
 - every positive literal in g is in s and
 - every negative literal in g is not in s .
- definitions for “holds” and “satisfies” may be generalized using substitutions

DWR Example: STRIPS States

```
state = {
  adjacent(loc1,loc2), adjacent(loc2, loc1),
  attached(p1,loc1), attached(p2,loc1),
  belong(crane1,loc1),
  occupied(loc2),
  empty(crane1),
  at(r1,loc2),
  unloaded(r1),
  in(c1,p1),in(c3,p1),
  on(c3,c1), on(c1,pallet),
  top(c3,p1),
  in(c2,p2),
  on(c2,pallet),
  top(c2,p2)}
```



DWR Example: STRIPS States

- predicate symbols: relations for DWR domain
- constant symbols: for objects in the domain {loc1, loc2, r1, crane1, p1, p2, c1, c2, c3, pallet}
- state = {attached(p1,loc1), attached(p2,loc1), in(c1,p1),in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2, loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}

Overview

- States with Internal Structure
- Operators with Structure
- Planning Domains and Problems
- Forward State-Space Search
- Backward State-Space Search

Overview

•States with Internal Structure

- just done: the STRIPS representation for world states

➤Operators with Structure

- now: the STRIPS representation for operators

•Planning Domains and Problems

•Forward State-Space Search

•Backward State-Space Search

Operators and Actions in STRIPS Planning Domains

- A planning operator in a STRIPS planning domain is a triple $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$ where:
 - the name of the operator $\text{name}(o)$ is a syntactic expression of the form $n(x_1, \dots, x_k)$ where n is a (unique) symbol and x_1, \dots, x_k are all the variables that appear in o , and
 - the preconditions $\text{precond}(o)$ and the effects $\text{effects}(o)$ of the operator are sets of literals.
- An action in a STRIPS planning domain is a ground instance of a planning operator.

Operators and Actions in STRIPS Planning Domains

• A planning operator in a STRIPS planning domain is a triple $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$ where:

• the name of the operator $\text{name}(o)$ is a syntactic expression of the form $n(x_1, \dots, x_k)$ where n is a (unique) symbol and x_1, \dots, x_k are all the variables that appear in o , and

• unique: no two operators in the same domain must have the same name symbol

• the preconditions $\text{precond}(o)$ and the effects $\text{effects}(o)$ of the operator are sets of literals.

• only variables mentioned in the name are allowed to appear in these literals

• An action in a STRIPS planning domain is a ground instance of a planning operator.

• actions are also called operator instances

• note: rigid relation must not appear in the effects of an operator, only in the preconditions

DWR Example: STRIPS Operators

- $\text{move}(r, l, m)$
 - precondition: $\text{adjacent}(l, m), \text{at}(r, l), \neg \text{occupied}(m)$
 - effects: $\text{at}(r, m), \text{occupied}(m), \neg \text{occupied}(l), \neg \text{at}(r, l)$
- $\text{load}(k, l, c, r)$
 - precondition: $\text{belong}(k, l), \text{holding}(k, c), \text{at}(r, l), \text{unloaded}(r)$
 - effects: $\text{empty}(k), \neg \text{holding}(k, c), \text{loaded}(r, c), \neg \text{unloaded}(r)$
- $\text{put}(k, l, c, d, p)$
 - precondition: $\text{belong}(k, l), \text{attached}(p, l), \text{holding}(k, c), \text{top}(d, p)$
 - effects: $\neg \text{holding}(k, c), \text{empty}(k), \text{in}(c, p), \text{top}(c, p), \text{on}(c, d), \neg \text{top}(d, p)$

DWR Example: STRIPS Operators

• $\text{move}(r, l, m)$

- robot r moves from location l to an adjacent location m

• **precond:** $\text{adjacent}(l, m), \text{at}(r, l), \neg \text{occupied}(m)$

• **effects:** $\text{at}(r, m), \text{occupied}(m), \neg \text{occupied}(l), \neg \text{at}(r, l)$

• $\text{load}(k, l, c, r)$

- crane k at location l loads container c onto robot r

• **precond:** $\text{belong}(k, l), \text{holding}(k, c), \text{at}(r, l), \text{unloaded}(r)$

• **effects:** $\text{empty}(k), \neg \text{holding}(k, c), \text{loaded}(r, c), \neg \text{unloaded}(r)$

• $\text{put}(k, l, c, d, p)$

- crane k at location l puts container c onto d in pile p

• **precond:** $\text{belong}(k, l), \text{attached}(p, l), \text{holding}(k, c), \text{top}(d, p)$

• **effects:** $\neg \text{holding}(k, c), \text{empty}(k), \text{in}(c, p), \text{top}(c, p), \text{on}(c, d), \neg \text{top}(d, p)$

• similar: unload and take operators

• action: just substitute variables with values consistently

Example: DWR Operator (PDDL)

```
;; moves a robot between two adjacent locations
(:action move
 :parameters (?r - robot ?from ?to - location)
 :precondition (and
  (adjacent ?from ?to) (at ?r ?from)
  (not (occupied ?to)))
 :effect (and
  (at ?r ?to) (occupied ?to)
  (not (occupied ?from)) (not (at ?r ?from)) ))
```

Example: DWR Action

•;; moves a robot between two adjacent locations

- Lisp convention: double semicolon not strictly necessary

•(:action move

•:parameters (?r - robot ?from ?to - location)

- typed parameters: “?r” of type robot and “?from” and “?to” of type location

•:precondition (and

- conjunction
- (adjacent ?from ?to) (at ?r ?from)
- (not (occupied ?to)))

•:effect (and

- (at ?r ?to) (occupied ?to)
- (not (occupied ?from)) (not (at ?r ?from))))

- note: common to find negated fluent preconditions as effects, but not always

Applicability and State Transitions

- Let L be a set of literals.
 - \underline{L}^+ is the set of atoms that are positive literals in L and
 - \underline{L}^- is the set of all atoms whose negations are in L .
- Let a be an action and s a state. Then a is applicable in s iff:
 - $\text{precond}^+(a) \subseteq s$; and
 - $\text{precond}^-(a) \cap s = \{\}$.
- The state transition function γ for an applicable action a in state s is defined as:
 - $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$

Applicability and State Transitions

- Let L be a set of literals.
 - \underline{L}^+ is the set of atoms that are positive literals in L and
 - \underline{L}^- is the set of all atoms whose negations are in L .
 - specifically, for operators: $\text{precond}^+(a)$, $\text{precond}^-(a)$, $\text{effects}^+(a)$, and $\text{effects}^-(a)$ are defined in this way
- Let a be an action and s a state. Then a is applicable in s iff:
 - $\text{precond}^+(a) \subseteq s$; and
 - $\text{precond}^-(a) \cap s = \{\}$.
- The state transition function γ for an applicable action a in state s is defined as:
 - $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$
 - note implicit frame axioms: what is not mentioned as an effect persists

Finding Applicable Actions: Algorithm

```
function addApplicables(A, op, precs,  $\sigma$ , s)  
  if precs.isEmpty() then  
    for every np in precs do  
      if s.falsifies( $\sigma(np)$ ) then return  
      A.add( $\sigma(op)$ )  
  else  
    pp  $\leftarrow$  precs.chooseOne()  
    for every sp in s do  
       $\sigma' \leftarrow \sigma$ .extend(sp, pp)  
      if  $\sigma'$ .isValid() then  
        addApplicables(A, op, (precs - pp),  $\sigma'$ , s)
```

•function addApplicables(*A*, *op*, *precs*, σ , *s*)

•Parameters: set of actions, operator, set of remaining preconditions, partial substitution, state

•if *precs*.isEmpty() then

•Note: σ should now be complete

•for every *np* in *precs* do

•if *s*.falsifies($\sigma(np)$) then return

•*A*.add($\sigma(op)$)

•test for inconsistent effects before adding!

•else

•*pp* \leftarrow *precs*.chooseOne()

•Heuristics: nr of atoms in state; nr of unbound variables

•for every *sp* in *s* do

• $\sigma' \leftarrow \sigma$.extend(*sp*, *pp*)

•if σ' .isValid() then

•addApplicables(*A*, *op*, (*precs* - *pp*), σ' , *s*)

Example: Applicable Actions

```
{adjacent(loc1,loc2),
 adjacent(loc2, loc1),
 attached(p1,loc1),
 attached(p2,loc1),
 belong(crane1,loc1),
 occupied(loc2),
 empty(crane1),
 at(r1,loc2),
 unloaded(r1),
 in(c1,p1),in(c3,p1),
 on(c3,c1), on(c1,pallet),
 top(c3,p1),
 in(c2,p2),
 on(c2,pallet),
 top(c2,p2)}
```

```
(:action move :parameters (?r - robot ?from ?to - location)
:precondition (and (adjacent ?from ?to) (at ?r ?from)
(not (occupied ?to)))
:effect (and (at ?r ?to) (occupied ?to)
(not (occupied ?from)) (not (at ?r ?from)) ))
```

$pp \leftarrow (\text{adjacent } ?\text{from } ?\text{to})$

$sp \leftarrow (\text{adjacent loc1 loc2})$

$\sigma' \leftarrow \{?\text{from}=\text{loc1}, ?\text{to}=\text{loc2}\}$

$pp \leftarrow (\text{at } ?r ?\text{from})$

$sp \leftarrow (\text{at r1 loc2})$

σ' not valid

$sp \leftarrow (\text{adjacent loc2 loc1})$

$\sigma' \leftarrow \{?\text{from}=\text{loc2}, ?\text{to}=\text{loc1}\}$

$pp \leftarrow (\text{at } ?r ?\text{from})$

$sp \leftarrow (\text{at r1 loc2})$

$\sigma' \leftarrow \{?\text{from}=\text{loc2}, ?\text{to}=\text{loc1}, ?r=r1\}$

$\sigma(np) = (\text{not (occupied loc1)})$

A.add((move r1 loc2 loc1))

Overview

- States with Internal Structure
- Operators with Structure
- Planning Domains and Problems
- Forward State-Space Search
- Backward State-Space Search

Overview

•States with Internal Structure

•Operators with Structure

- just done: the STRIPS representation for operators

➤Planning Domains and Problems

- now: a formal definition of STRIPS planning problems and solutions

•Forward State-Space Search

•Backward State-Space Search

Classical Planning

- task: find solution for planning problem
- planning problem
 - initial state
 - atoms (relations, objects)
 - planning domain
 - operators (name, preconditions, effects)
 - goal
- solution (plan)

Classical Planning

•task: find solution for planning problem

•planning problem

•initial state

•state is a set of **atoms (relations, objects)**

•difference between representations: what constitutes an atom

•planning domain

•operators (name, preconditions, effects)

•goal

•solution (plan)

STRIPS Planning Domains

- Let \mathcal{L} be a function-free first-order language. A STRIPS planning domain on \mathcal{L} is a restricted state-transition system $\Sigma=(S,A,\gamma)$ such that:
 - S is a set of STRIPS states, i.e. sets of ground atoms
 - A is a set of ground instances of some STRIPS planning operators O
 - $\gamma:S \times A \rightarrow S$ where
 - $\gamma(s,a)=(s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if a is applicable in s
 - $\gamma(s,a)=\text{undefined}$ otherwise
 - S is closed under γ

STRIPS Planning Domains

- Let \mathcal{L} be a function-free first-order language. A STRIPS planning domain on \mathcal{L} is a restricted state-transition system $\Sigma=(S,A,\gamma)$ such that:
 - **S is a set of STRIPS states, i.e. sets of ground atoms**
 - STRIPS vs. propositional domains: ground atoms instead of propositions
 - **A is a set of ground instances of some STRIPS planning operators O**
 - abstraction in operator descriptions due to variables; action effectively same as propositional actions
 - **$\gamma:S \times A \rightarrow S$ where**
 - **$\gamma(s,a)=(s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if a is applicable in s**
 - **$\gamma(s,a)=\text{undefined}$ otherwise**
 - **S is closed under γ**

Example: Domain

```
(define (domain dock-worker-robot)
  (:requirements :strips :typing)
  (:types location pile robot crane container)
  (:constants pallet - container)
  (:predicates
    (adjacent ?l1 ?l2 - location)
    (attached ?p - pile ?l - location)
    (belong ?k - crane ?l - location)
    (at ?r - robot ?l - location)
    (occupied ?l - location)
    (loaded ?r - robot ?c - container)
    (unloaded ?r - robot)
    (holding ?k - crane ?c - container)
    (empty ?k - crane)
    (in ?c - container ?p - pile)
    (top ?c - container ?p - pile)
    (on ?k1 - container ?k2 - container));;

    (:action move :parameters (?r - robot ?from ?to)
      :precondition (and (adjacent ?from ?to)
        (not (occupied ?from) (occupied ?to)))
      :effect (and (at ?r ?to) (not (occupied ?to))))

    (:action load :parameters (?k - crane ?l - location ?c - container)
      :precondition (and (at ?r ?l) (belong ?k ?l) (not (loaded ?k ?c)))
      :effect (and (loaded ?k ?c) (not (unloaded ?k))))

    (:action unload :parameters (?k - crane ?l - location ?c - container)
      :precondition (and (belong ?k ?l) (at ?k ?l) (loaded ?k ?c))
      :effect (and (unloaded ?k) (holding ?k ?c)))

    (:action take :parameters (?k - crane ?l - location ?c - container)
      :precondition (and (belong ?k ?l) (attached ?k ?l) (empty ?c) (not (on ?k ?c)))
      :effect (and (holding ?k ?c) (top ?k ?c)))

    (:action put :parameters (?k - crane ?l - location ?c - container)
      :precondition (and (belong ?k ?l) (attached ?k ?l) (holding ?k ?c) (not (empty ?c)))
      :effect (and (in ?c ?p) (top ?c ?p) (on ?k ?c)))
```

Automated Planning: Theory and Practice

AUTOMATED PLANNING

theory and practice

Malik Ghallab
LAAS-CNRS
Toulouse, France

Dana Nau
University of Maryland
College Park, Md., USA

Paolo Traverso
ITC-IRST
Trento, Italy

Morgan Kaufmann Publishers, May 2004, 663 pages. ISBN 1-55860-066-7

"With this wonderful text, three established leaders in the field of automated planning have met a long-standing need for a detailed, comprehensive book that can be used both as a text for college classes—complete with student exercises—and as a reference book for researchers and practitioners."
— Martha Pollack, University of Michigan

About the book

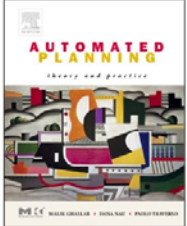
- Endorsements
- Publisher's page for the book
 - North America
 - elsewhere
- Material from the book
 - About the authors
 - Foreword by Malik Ghallab
 - Preface
 - Table of contents
 - Blurb from the back cover

Lecture slides

- Dana Nau's lecture slides (in English)
- Fredheuer's lecture slides (in French)

Auxiliary materials

- The domain
 - PDDL specification
 - Problem with 1 robot and 2 locations
 - Problem with 3 robots and 8 locations
- Errata list



The cover art is a reproduction of *La grande répartition* (The large repartition), painted by Fernand Léger in 1923. The original is at the Musée national Fernand Léger.

STRIPS Planning Problems

- A STRIPS planning problem is a triple $\mathcal{P}=(\Sigma, s_i, g)$ where:
 - $\Sigma=(S, A, \gamma)$ is a STRIPS planning domain on some first-order language \mathcal{L}
 - $s_i \in S$ is the initial state
 - g is a set of ground literals describing the goal such that the set of goal states is: $S_g = \{s \in S \mid s \text{ satisfies } g\}$

STRIPS Planning Problems

- A STRIPS planning problem is a triple $\mathcal{P}=(\Sigma, s_i, g)$ where:

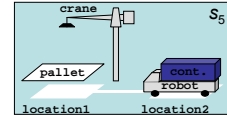
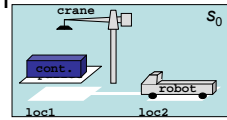
- $\Sigma=(S, A, \gamma)$ is a STRIPS planning domain on some first-order language \mathcal{L}
- $s_i \in S$ is the initial state
- g is a set of ground literals describing the goal such that the set of goal states is: $S_g = \{s \in S \mid s \text{ satisfies } g\}$
 - note: g may contain positive and negated ground atoms (no closed world assumption for goals)

DWR Example: STRIPS Planning Problem

- Σ : STRIPS planning domain for DWR domain
- s_i : any state

– example: $s_0 = \{\text{attached}(\text{pile}, \text{loc1}), \text{in}(\text{cont}, \text{pile}), \text{top}(\text{cont}, \text{pile}), \text{on}(\text{cont}, \text{pallet}), \text{belong}(\text{crane}, \text{loc1}), \text{empty}(\text{crane}), \text{adjacent}(\text{loc1}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc1}), \text{at}(\text{robot}, \text{loc2}), \text{occupied}(\text{loc2}), \text{unloaded}(\text{robot})\}$

- g : any subset of L
 - example: $g = \{\neg \text{unloaded}(\text{robot}), \text{at}(\text{robot}, \text{loc2})\}$, i.e. $S_g = \{s_5\}$



DWR Example: STRIPS Planning Problem

• Σ : STRIPS planning domain for DWR domain

• see previous slides

• s_i : any state

• example: $s_0 = \{\text{attached}(\text{pile}, \text{loc1}), \text{in}(\text{cont}, \text{pile}), \text{top}(\text{cont}, \text{pile}), \text{on}(\text{cont}, \text{pallet}), \text{belong}(\text{crane}, \text{loc1}), \text{empty}(\text{crane}), \text{adjacent}(\text{loc1}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc1}), \text{at}(\text{robot}, \text{loc2}), \text{occupied}(\text{loc2}), \text{unloaded}(\text{robot})\}$

• note: s_0 is not necessarily initial state

• g : any subset of L

• example: $g = \{\neg \text{unloaded}(\text{robot}), \text{at}(\text{robot}, \text{loc2})\}$, i.e. $S_g = \{s_5\}$

• other relations will hold, but they are not mentioned in the goal = partial specification of a state

Example: DWR Problem (PDDL)

```

;; a simple DWR problem with 1 robot and 2 locations
(define (problem dwrbp1)
  (:domain dock-worker-robot)
  (:objects
    r1 - robot
    l1 l2 - location
    k1 k2 - crane
    p1 q1 p2 q2 - pile
    ca cb cc cd ce cf pallet - container)
  (:init
    (adjacent l1 l2)
    (adjacent l2 l1)
    (attached p1 l1)
    (attached q1 l1)
    (attached p2 l2)
    (attached q2 l2)
    (belong k1 l1)
    (belong k2 l2)

    (in ca p1) (in cb p1) (in cc p1)
    (on ca pallet) (on cb ca) (on cc cb)
    (top cc p1)

    (in cd q1) (in ce q1) (in cf q1)
    (on cd pallet) (on ce cd) (on cf ce)
    (top cf q1)

    (top pallet p2)
    (top pallet q2)

    (at r1 l1)
    (unloaded r1)
    (occupied l1)

    (empty k1)
    (empty k2))
  ;; task is to move all containers to locations l2
  ;; ca and cc in pile p2, the rest in q2
  (:goal (and
    (in ca p2) (in cc p2)
    (in cb q2) (in cd q2) (in ce q2) (in cf q2))))

```

<http://projects.laas.fr/planning/DWR-pb1>

Example: DWR Problem

•;; a simple DWR problem with 1 robot and 2 locations

•(define (problem dwrbp1)

•(:domain dock-worker-robot)

•(:objects r1 - robot l1 l2 - location k1 k2 - crane p1 q1 p2 q2 - pile ca cb cc cd ce cf pallet - container)

•(:init

•(adjacent l1 l2) (adjacent l2 l1) (attached p1 l1) (attached q1 l1)
(attached p2 l2) (attached q2 l2) (belong k1 l1) (belong k2 l2)

•rigid relations

•(in ca p1) (in cb p1) (in cc p1) (on ca pallet) (on cb ca) (on cc cb)
(top cc p1)

•(in cd q1) (in ce q1) (in cf q1) (on cd pallet) (on ce cd) (on cf ce)
(top cf q1)

•the two piles of containers at location l1

•(top pallet p2)

•(top pallet q2)

•no containers at location l2

•(at r1 l1) (unloaded r1) (occupied l1)

•(empty k1) (empty k2))

•;; task is to move all containers to locations l2 ;; ca and cc in pile p2, the rest in q2

•(:goal (and

•(in ca p2) (in cc p2)

•(in cb q2) (in cd q2) (in ce q2) (in cf q2))))

•note: many solutions as order of containers is undefined

Classical Plans

- A plan is any sequence of actions $\pi = \langle a_1, \dots, a_k \rangle$, where $k \geq 0$.
 - The length of plan π is $|\pi| = k$, the number of actions.
 - If $\pi_1 = \langle a_1, \dots, a_k \rangle$ and $\pi_2 = \langle a'_1, \dots, a'_j \rangle$ are plans, then their concatenation is the plan $\pi_1 \bullet \pi_2 = \langle a_1, \dots, a_k, a'_1, \dots, a'_j \rangle$.
 - The extended state transition function for plans is defined as follows:
 - $\gamma(s, \pi) = s$ if $k = 0$ (π is empty)
 - $\gamma(s, \pi) = \gamma(\gamma(s, a_1), \langle a_2, \dots, a_k \rangle)$ if $k > 0$ and a_1 applicable in s
 - $\gamma(s, \pi) = \text{undefined}$ otherwise

Classical Plans

- note: classical definitions apply to all representations
- A plan is any sequence of actions $\pi = \langle a_1, \dots, a_k \rangle$, where $k \geq 0$.
 - $k = 0$ means no actions in the empty plan
 - The length of plan π is $|\pi| = k$, the number of actions.
 - If $\pi_1 = \langle a_1, \dots, a_k \rangle$ and $\pi_2 = \langle a'_1, \dots, a'_j \rangle$ are plans, then their concatenation is the plan $\pi_1 \bullet \pi_2 = \langle a_1, \dots, a_k, a'_1, \dots, a'_j \rangle$.
 - The extended state transition function for plans is defined as follows:
 - $\gamma(s, \pi) = s$ if $k = 0$ (π is empty)
 - $\gamma(s, \pi) = \gamma(\gamma(s, a_1), \langle a_2, \dots, a_k \rangle)$ if $k > 0$ and a_1 applicable in s
 - $\gamma(s, \pi) = \text{undefined}$ otherwise
- plan corresponds to a path through the state space

Classical Solutions

- Let $\mathcal{P}=(\Sigma, s_i, g)$ be a planning problem. A plan π is a solution for \mathcal{P} if $\gamma(s_i, \pi)$ satisfies g .
 - A solution π is redundant if there is a proper subsequence of π is also a solution for \mathcal{P} .
 - π is minimal if no other solution for \mathcal{P} contains fewer actions than π .

Classical Solutions

• Let $\mathcal{P}=(\Sigma, s_i, g)$ be a propositional planning problem. A plan π is a solution for \mathcal{P} if $g \subseteq \gamma(s_i, \pi)$.

• A solution π is redundant if there is a proper subsequence of π is also a solution for \mathcal{P} .

• π is minimal if no other solution for \mathcal{P} contains fewer actions than π .

• note: a minimal solution cannot be redundant

• solution is a path through the state space that leads from the initial state to a state that satisfies the goal

Classical Representations

- propositional representation
 - world state is set of propositions
 - action consists of precondition propositions, propositions to be added and removed
- STRIPS representation
 - like propositional representation, but first-order literals instead of propositions
- state-variable representation
 - state is tuple of state variables $\{x_1, \dots, x_n\}$
 - action is partial function over states

Classical Representations

• propositional representation

- world state is set of propositions
- action consists of precondition propositions, propositions to be added and removed

• STRIPS representation

- named after STRIPS planner
- like propositional representation, but first-order literals instead of propositions
- most popular for restricted state-transitions systems

• state-variable representation

- state is tuple of state variables $\{x_1, \dots, x_n\}$
 - action is partial function over states
 - useful where state is characterized by attributes over finite domains
- equally expressive: planning domain in one representation can also be represented in the others

Overview

- States with Internal Structure
- Operators with Structure
- Planning Domains and Problems
- Forward State-Space Search
- Backward State-Space Search

Overview

•States with Internal Structure

•Operators with Structure

•Planning Domains and Problems

- just doe: a formal definition of STRIPS planning problems and solutions

➤Forward State-Space Search

- now: applying search algorithms to solve planning problems

•Backward State-Space Search

State-Space Search

- idea: apply standard search algorithms (breadth-first, depth-first, A*, etc.) to planning problem:
 - search space is subset of state space
 - nodes correspond to world states
 - arcs correspond to state transitions
 - path in the search space corresponds to plan

State-Space Search

•idea: apply standard search algorithms (breadth-first, depth-first, A*, etc.) to planning problem:

•**search space is subset of state space**

- subset: generate only reachable states until a goal state has been found

•**nodes correspond to world states**

•**arcs correspond to state transitions**

- arcs are labelled with actions

•**path in the search space corresponds to plan**

- path from initial state to goal state is solution

State-Space Planning as a Search Problem

- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
 - initial state: s_i
 - goal test for state s : s satisfies g
 - path cost function for plan π : $|\pi|$
 - successor function for state s : $\Gamma(s)$

State-Space Planning as a Search Problem

- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
 - initial state: s_i
 - goal test for state s : s satisfies g
 - path cost function for plan π : $|\pi|$
 - simplification: plan length = path cost
 - successor function for state s : $\Gamma(s)$
 - to be defined next

Reachable Successor States

- The successor function $\Gamma^m: 2^S \rightarrow 2^S$ for a STRIPS domain $\Sigma = (S, A, \gamma)$ is defined as:
 - $\Gamma(s) = \{\gamma(s, a) \mid a \in A \text{ and } a \text{ applicable in } s\} \quad \text{for } s \in S$
 - $\Gamma(\{s_1, \dots, s_n\}) = \bigcup_{k \in [1, n]} \Gamma(s_k)$
 - $\Gamma^0(\{s_1, \dots, s_n\}) = \{s_1, \dots, s_n\}$
 - $\Gamma^m(\{s_1, \dots, s_n\}) = \Gamma(\Gamma^{m-1}(\{s_1, \dots, s_n\}))$
- The transitive closure of Γ defines the set of all reachable states:
 - $\Gamma^>(s) = \bigcup_{k \in [0, \infty]} \Gamma^k(\{s\}) \quad \text{for } s \in S$

Reachable Successor States

- The successor function $\Gamma^m: 2^S \rightarrow 2^S$ for a STRIPS domain $\Sigma = (S, A, \gamma)$ is defined as:

- $\Gamma(s) = \{\gamma(s, a) \mid a \in A \text{ and } a \text{ applicable in } s\} \text{ for } s \in S$
 - all states that can be reached by applying exactly one applicable action
- $\Gamma(\{s_1, \dots, s_n\}) = \bigcup_{k \in [1, n]} \Gamma(s_k)$
 - union of all states that can be reached by applying exactly one applicable action
- $\Gamma^0(\{s_1, \dots, s_n\}) = \{s_1, \dots, s_n\}$
 - identity function; the states themselves
- $\Gamma^m(\{s_1, \dots, s_n\}) = \Gamma(\Gamma^{m-1}(\{s_1, \dots, s_n\}))$
 - union of all states that can be reached by applying exactly m applicable actions

- The transitive closure of Γ defines the set of all reachable states:

- $\Gamma^>(s) = \bigcup_{k \in [0, \infty]} \Gamma^k(\{s\}) \text{ for } s \in S$
 - pronounce: gamma forward
 - all states that can be reached by applying any number of applicable actions

Solution Existence

- **Proposition:** A STRIPS planning problem $\mathcal{P}=(\Sigma, s_i, g)$ (and a statement of such a problem $P=(O, s_i, g)$) has a solution iff $S_g \cap \Gamma^>(\{s_i\}) \neq \{\}$.

Solution Existence

• **Proposition:** A STRIPS planning problem $\mathcal{P}=(\Sigma, s_i, g)$ (and a statement of such a problem $P=(O, s_i, g)$) has a solution iff $S_g \cap \Gamma^>(\{s_i\}) \neq \{\}$.

- ... iff there is a goal state that is also a reachable state
- enumerate all reachable states from the initial state (in some good order) and we will generate a goal state eventually = forward search

Forward State-Space Search Algorithm

```
function fwdSearch(O, si, g)  
  state  $\leftarrow$  si  
  plan  $\leftarrow$   $\langle \rangle$   
  loop  
    if state.satisfies(g) then return plan  
    applicables  $\leftarrow$  {ground instances from O applicable in state}  
    if applicables.isEmpty() then return failure  
    action  $\leftarrow$  applicables.chooseOne()  
    state  $\leftarrow$   $\gamma$ (state, action)  
    plan  $\leftarrow$  plan •  $\langle$ action $\rangle$ 
```

•function fwdSearch(*O*, *s_i*, *g*)

- given: statement of a STRIPS planning problem; return a solution plan (or failure)

- non-deterministic version

•*state* \leftarrow *s_i*

- start with the initial state

•*plan* \leftarrow $\langle \rangle$

- initialize solution with empty plan (partial plan: prefix of the solution)

•loop

•**if** *state*.satisfies(*g*) **then return** *plan*

•*applicables* \leftarrow {ground instances from *O* applicable in *state*}

•**if** *applicables*.isEmpty() **then return** failure

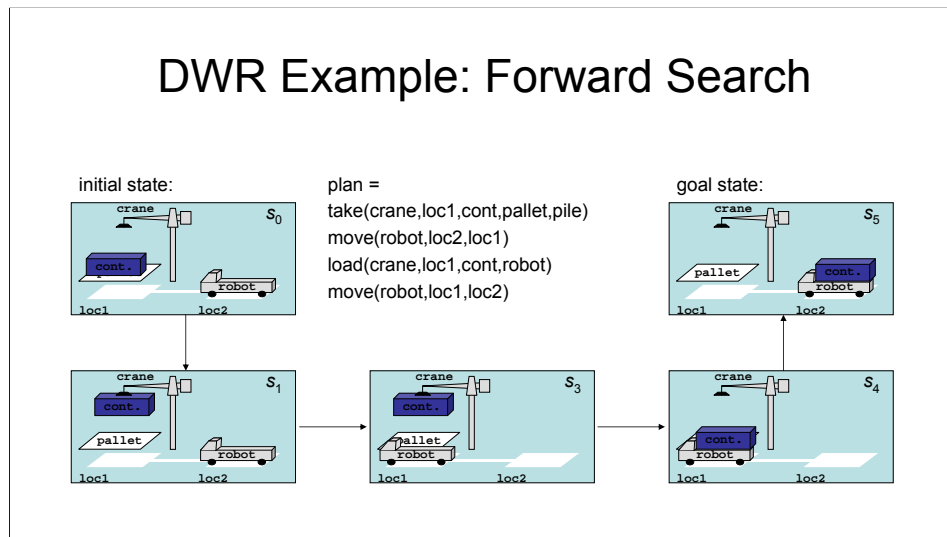
•*action* \leftarrow *applicables*.chooseOne()

- non-deterministically choose an applicable action

•*state* \leftarrow γ (*state*, *action*)

•*plan* \leftarrow *plan* • \langle *action* \rangle

DWR Example: Forward Search



•DWR Example: Forward Search

- goal state available at start
- choose action; (non-deterministic; alternative would be “move” action)
- compute successor state
- chose action; (again non-deterministic; alternative would be “put” returning to S_0)
- compute successor state
- chose action
- compute successor state
- chose action
- compute successor state; goal state!

Properties of Forward Search

- **Proposition:** fwdSearch is sound, i.e. if the function returns a plan as a solution then this plan is indeed a solution.
 - proof idea: show (by induction) $state = \gamma(s_i, plan)$ at the beginning of each iteration of the loop
- **Proposition:** fwdSearch is complete, i.e. if there exists a solution plan then there is an execution trace of the function that will return this solution plan.
 - proof idea: show (by induction) there is an execution trace for which $plan$ is a prefix of the sought plan

Properties of Forward Search

• **Proposition:** fwdSearch is sound, i.e. if the function returns a plan as a solution then this plan is indeed a solution.

• **proof idea:** show (by induction) $state = \gamma(s_i, plan)$ at the beginning of each iteration of the loop

• variable $state$ always contains STRIPS state that is result of applying $plan$ (variable) in initial state

• hence: when $state$ contains goal state $plan$ contains solution plan

• **Proposition:** fwdSearch is complete, i.e. if there exists solution plan then there is an execution trace of the function that will return this solution plan.

• **proof idea:** show (by induction) there is an execution trace for which $plan$ is a prefix of the sought plan

• given a solution plan, the variable $plan$ contains a prefix of that plan starting with the initial empty plan

• chooseOne(...) can always choose the next step in the solution plan we are looking for

Overview

- States with Internal Structure
- Operators with Structure
- Planning Domains and Problems
- Forward State-Space Search
- Backward State-Space Search

Overview

•States with Internal Structure

•Operators with Structure

•Planning Domains and Problems

•Forward State-Space Search

- just done: applying search algorithms to solve planning problems

➤Backward State-Space Search

- now: searching through the space of sub-goals (really: goal-space search)

Relevance and Regression Sets

- Let $\mathcal{P}=(\Sigma, s_i, g)$ be a STRIPS planning problem. An action $a \in A$ is relevant for g if
 - $g \cap \text{effects}(a) \neq \{\}$ and
 - $g^+ \cap \text{effects}^-(a) = \{\}$ and $g^- \cap \text{effects}^+(a) = \{\}$.
- The regression set of g for a relevant action $a \in A$ is:
 - $\gamma^{-1}(g, a) = (g - \text{effects}(a)) \cup \text{precond}(a)$

Relevance and Regression Sets

- Let $\mathcal{P}=(\Sigma, s_i, g)$ be a STRIPS planning problem. An action $a \in A$ is relevant for g if
 - $g \cap \text{effects}(a) \neq \{\}$ and
 - a 's effects contribute to g
 - $g^+ \cap \text{effects}^-(a) = \{\}$ and $g^- \cap \text{effects}^+(a) = \{\}$.
 - a 's effects do not conflict with g
- The regression set of g for a relevant action $a \in A$ is:
 - $\gamma^{-1}(g, a) = (g - \text{effects}(a)) \cup \text{precond}(a)$
 - subtract all effects, not just positive ones
 - note: goal and regression set $(\gamma^{-1}(g, a))$ are sets of ground literals
 - regression set can be seen as sub-goal

Regression Function

- The regression function Γ^{-m} for a STRIPS domain $\Sigma=(S,A,\gamma)$ on L is defined as:
 - $\Gamma^{-1}(g)=\{\gamma^{-1}(g,a) \mid a \in A \text{ is relevant for } g\}$ for $g \in 2^L$
 - $\Gamma^{-0}(\{g_1, \dots, g_n\}) = \{g_1, \dots, g_n\}$
 - $\Gamma^{-1}(\{g_1, \dots, g_n\}) = \bigcup_{k \in [1, n]} \Gamma^{-1}(g_k)$
 - $\Gamma^{-m}(\{g_1, \dots, g_n\}) = \Gamma^{-1}(\Gamma^{-(m-1)}(\{g_1, \dots, g_n\}))$
- The transitive closure of Γ^{-1} defines the set of all regression sets:
 - $\Gamma^{<}(g) = \bigcup_{k \in [0, \infty]} \Gamma^{-k}(\{g\})$ for $g \in 2^L$

Regression Function

- The regression function Γ^{-m} for a STRIPS domain $\Sigma=(S,A,\gamma)$ on L is defined as:

- $\Gamma^{-1}(g)=\{\gamma^{-1}(g,a) \mid a \in A \text{ is relevant for } g\}$ for $g \in 2^L$
 - regression set for a single set of (goal) propositions
- $\Gamma^{-0}(\{g_1, \dots, g_n\}) = \{g_1, \dots, g_n\}$
 - as for successors
- $\Gamma^{-1}(\{g_1, \dots, g_n\}) = \bigcup_{k \in [1, n]} \Gamma^{-1}(g_k)$
 - union of individual regression sets
- $\Gamma^{-m}(\{g_1, \dots, g_n\}) = \Gamma^{-1}(\Gamma^{-(m-1)}(\{g_1, \dots, g_n\}))$
 - minimal sets of propositions that must hold in a state s from which m actions lead to a state in which one of g_1, \dots, g_n is satisfied

- The transitive closure of Γ^{-1} defines the set of all regression sets:

- $\Gamma^{<}(g) = \bigcup_{k \in [0, \infty]} \Gamma^{-k}(\{g\})$ for $g \in 2^L$
 - pronounce: gamma backward

State-Space Planning as a Search Problem

- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
 - initial search state: g
 - goal test for state s : s_i satisfies s
 - path cost function for plan π : $|\pi|$
 - successor function for state s : $\Gamma^{-1}(s)$

State-Space Planning as a Search Problem

- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
 - initial search state: g
 - search backwards from the goal
 - goal test for state s : s satisfies s_i
 - initial state satisfies regression set (sub-goal)
 - path cost function for plan π : $|\pi|$
 - successor function for state s : $\Gamma^{-1}(s)$
 - as defined in previous slide

Example: Regression with Operators

- goal: $\text{at}(\text{robot}, \text{loc1})$
- operator: $\text{move}(r, l, m)$
 - precondition: $\text{adjacent}(l, m), \text{at}(r, l), \neg \text{occupied}(m)$
 - effects: $\text{at}(r, m), \text{occupied}(m), \neg \text{occupied}(l), \neg \text{at}(r, l)$
- actions: $\text{move}(\text{robot}, l, \text{loc1})$
 - $l = ?$
 - many options increase branching factor
- lifted backward search: use partially instantiated operators instead of actions

Example: Regression with Operators

- goal: $\text{at}(\text{robot}, \text{loc1})$
- operator: $\text{move}(r, l, m)$
 - precond: $\text{adjacent}(l, m), \text{at}(r, l), \neg \text{occupied}(m)$
 - effects: $\text{at}(r, m), \text{occupied}(m), \neg \text{occupied}(l), \neg \text{at}(r, l)$
 - operator may achieve or undo goal depending on variable bindings
- actions: $\text{move}(\text{robot}, l, \text{loc1})$
 - $l = ?$
 - to contribute to goal, r must bound to robot and m to loc1; l can remain unbound
 - many options increase branching factor
 - keeping variables unbound can significantly reduce the branching factor (as opposed to using actions)
- lifted backward search: use partially instantiated operators instead of actions
 - essentially same as ground version, but need to maintain appropriate variable substitutions

Overview

- States with Internal Structure
- Operators with Structure
- Planning Domains and Problems
- Forward State-Space Search
- Backward State-Space Search

Overview

- **States with Internal Structure**
- **Operators with Structure**
- **Planning Domains and Problems**
- **Forward State-Space Search**
- **Backward State-Space Search**

- just done: searching through the space of sub-goals (really: goal-space search)