Advanced Topics Data Science and AI
Automated Planning and Acting

Deterministic Models

Tanya Braun
1. Planning and Acting with **Deterministic** Models
   a. State-variable representation
   b. Forward State-Search Space
   c. Heuristic Functions
   d. Backward Search
   e. Plan-Space Search

2. Planning and Acting with **Refinement** Methods

3. Planning and Acting with **Temporal** Models

4. Planning and Acting with **Nondeterministic** Models

5. Making Simple Decisions

6. Making Complex Decisions

7. Planning and Acting with **Probabilistic** Models

8. Provably Beneficial AI
   - Other: open world, perceiving, learning
     - If time permits
Outline per the Book

2.1 **State-variable representation**
   • State = \{values of variables\}; action = changes to those values

2.2 **Forward state-space search**
   • Start at initial state, look for sequence of actions that achieve goal

2.3 **Heuristic functions**
   • How to guide a forward state-space search

2.6 **Incorporating planning into an actor**
   • Online lookahead, unexpected events

2.4 **Backward search**
   • Start at goal state, go backwards toward initial state

2.5 **Plan-space search**
   • Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan
Motivation

• How to model a complex environment?
  • Generally need simplifying assumptions

• Classical planning
  • Finite, static world
    • Change occurs only when the actor causes it
  • No concurrent actions, no explicit time
    • Just a sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$
  • Determinism, no uncertainty
    • Can predict exactly what each action will do
    • No accidents, no “chance” outcomes

• Avoids a lot of complications
  • But most real-world environments don’t satisfy the assumptions

• Errors in prediction
  • OK if they’re infrequent and don’t have severe consequences
Domain Model

- **State-transition system** *(or classical planning domain)*
  - $\Sigma = (S, A, \gamma, \text{cost})$; cost is optional
    - $S$ - finite set of states that the system may be in
    - $A$ - finite set of actions: things the actor can do
    - $\gamma : S \times A \rightarrow S$ - prediction function *(or state-transition function)*
      - partial function: $\gamma(s, a)$ isn’t defined unless $a$ is applicable in $s$
      - $Dom(a) = \{s \in S | \gamma(s, a) \text{ is defined}\} = \{s \in S | a \text{ applicable in } s\}$
      - $Range(a) = \{\gamma(s, a) | s \in Dom(a)\}$
    - $\text{cost} : S \times A \rightarrow \mathbb{R}^+$ - or - $\text{cost} : A \rightarrow \mathbb{R}^+$
      - Could be monetary cost, time required, something else
      - Often omitted from $\Sigma$; default is $\text{cost}(a) = 1$

- Classical planning problem: $P = (\Sigma, s_0, S_g)$
  - (planning domain, initial state, set of goal states)
  - $s_0 \in S, S_g \subseteq S$

- Solution for $P$: a sequence of actions called plan that will produce a state in $S_g$
Representing $\Sigma$

- If $S$ and $A$ are small enough
  - Give each state and action a name
  - For each $s$ and $a$, store $\gamma(s, a)$ in a lookup table

- In larger domains, don’t represent all states explicitly
  - Language for describing properties of states
  - Language for describing how each action changes those properties
  - Start with initial state, use actions to produce other states
Domain-specific Representation

- Made to order for a specific environment
- State: arbitrary data structure

Action: (head, preconditions, effects, cost)
  - head: name and parameter list
    - Get actions by instantiating the parameters
  - preconditions:
    - Computational tests to predict whether an action can be performed in a state $s$
    - Should be necessary/sufficient for the action to run without error
  - effects:
    - Procedures that modify the current state
  - cost: procedure that returns a number
    - Can be omitted, default is cost $\equiv 1$
Example

• Drilling holes in a metal workpiece
  • A state
    • Geometric model of the workpiece, information about its location and orientation
    • Capabilities and status of drilling machine and drill bit
  • Several actions
    • Putting the workpiece onto the drilling machine
    • Clamping it
    • Loading a drill bit
    • Drilling (next slide)
Drilling

• Name and parameters:
  • drill-hole(*machine*, *drill-bit*, *workpiece*, *geometry*, *machining-tolerances*)

• Preconditions
  • Can the drilling machine and drill bit produce a hole having the desired geometry and machining tolerances?
  • Is the drill bit installed? Is the workpiece clamped onto the drilling platform? Etc.

• Effects
  • Geometric model of new workpiece geometry, annotated with tolerances

• Cost
  • Estimate of time or monetary cost
Discussion

• Advantage of domain-specific representation:
  • Can choose whatever works best for that particular domain

• Disadvantage:
  • For each new domain, need new representation and deliberation algorithms

• Alternative: domain-independent representation
  • Try to create a “standard format” that can be used for many different planning domains
  • Deliberation algorithms that work for anything in this format

• State-variable representation
  • Simple formats for describing states and actions
  • Limited representational capability
    • But easy to compute, easy to reason about
  • Domain-independent search algorithms and heuristic functions that can be used in all state-variable planning problems
State-Variable Representation

- $E$ : environment that we want to represent
- $B$ : set of objects
  - Names for objects in $E$, mathematical constants, ...
  - Only needs to include objects that matter at current level of abstraction
- Example (slightly different from the book)
  - $B = \text{Robots} \cup \text{Containers} \cup \text{Locs} \cup \{nil\}$
    - $\text{Robots} = \{r1, r2\}$
    - $\text{Containers} = \{c1, c2\}$
    - $\text{Locs} = \{d1, d2, d3\}$
- Can omit lots of details
  - E.g., physical characteristics of robots, containers, loading docks, roads
Properties of Objects

• Define ways to represent properties of objects
  • Two kinds of properties: rigid and varying
  • Sets of rigid properties $R$ and varying properties $X$

• Rigid property: $n$-ary relation $r$ over $B$
  • Stays the same in every state
  • Representation
    • As a mathematical relation
      • $adj = \{(d1, d2), (d2, d1), (d1, d3), (d3, d1)\}$
    • As a set of ground atoms
      • $adj(d1, d2), adj(d2, d1), adj(d1, d3), adj(d3, d1)$
Varying Properties

- **Varying property** $x$ (or *fluent*)
  - May differ in different states
  - Represent it using a state variable to assign a value to

- Set of state variables $X = \{loc(r1), loc(r2), loc(c1), loc(c2), cargo(r1), cargo(r2)\}$

- Each state variable $x \in X$ has a range
  $\mathcal{R}(x) = \{\text{all values that can be assigned to } x\}$
  - $\mathcal{R}(loc(r1)) = \mathcal{R}(loc(r2)) = \text{Locs}$
  - $\mathcal{R}(loc(c1)) = \mathcal{R}(loc(c2)) = \text{Robots} \cup \text{Locs}$
  - $\mathcal{R}(cargo(r1)) = \mathcal{R}(cargo(r2)) = \text{Containers} \cup \{\text{nil}\}$
    - $cargo(r)$: robot $r$ has a cargo
States as Functions

• Represent each state as a variable-assignment function
  • Function that maps each $x \in X$ to a value in $\mathcal{R}(x)$
    $$s_1(\text{loc}(r1)) = d1, \quad s_1(\text{loc}(r2)) = d2,$$
    $$s_1(\text{cargo}(r1)) = \text{nil}, \quad s_1(\text{cargo}(r2)) = \text{nil},$$
    $$s_1(\text{loc}(c1)) = d1, \quad s_1(\text{loc}(c2)) = d2$$
  • Mathematically, a function is a set of ordered pairs
    $$s_1 = \{(\text{loc}(r1), d1), (\text{cargo}(r1), \text{nil}), (\text{loc}(c1), d1), \ldots \}$$

• Write it as a set of ground positive literals (or ground atoms):
  $$s_1 = \{\text{loc}(r1) = d1, \quad \text{cargo}(r1) = \text{nil}, \quad \text{loc}(c1) = d1,$$
  $$\text{loc}(r2) = d2, \quad \text{cargo}(r2) = \text{nil}, \quad \text{loc}(c2) = d2\}$$
States as Functions

• Let $s$ be a variable-assignment function
  • $s$ is a state only if it has a sensible meaning in our intended environment $E$
  • Interpretation: a function $I$
    • Maps each $b \in B$ to an object in $E$
    • Maps each $r \in R$ to a rigid property in $E$
    • Maps each $x \in X$ to a varying property in $E$

• State: a variable-assignment function $s$ such that $I(s)$ can occur in $E$
  • State space $S = \{ \text{all possible states} \}$
Action Templates

• **Action template** \(\alpha\): a parameterized set of actions
  \[\alpha = (\text{head}(\alpha), \text{pre}(\alpha), \text{eff}(\alpha), \text{cost}(\alpha))\]

• **head** (\(\alpha\)): name, parameters
  • Each parameter has a range \(\subseteq B\), e.g., \(\mathcal{R}(r) = \text{Robots}\)

• **pre** (\(\alpha\)): precondition literals
  • \(\text{rel}(t_1, \ldots, t_k)\)
  • \(\text{var}(t_1, \ldots, t_k) = t_0\)
  • \(\neg \text{rel}(t_1, \ldots, t_k)\)
  • \(\neg \text{var}(t_1, \ldots, t_k) = t_0\)
  • Each \(t_i\) is a parameter or an element of \(B\)

• **eff** (\(\alpha\)): effect literals
  • \(\text{var}(t_1, \ldots, t_k) \leftarrow t_0\)

• **cost** (\(\alpha\)): a number
  • Optional
  • Default = 1

• **Example**
  • head: \(\text{move}(r, l, m)\)
    • pre: \(\text{loc}(r) = l, \text{adj}(l, m)\)
    • eff: \(\text{loc}(r) \leftarrow m\)
  • head: \(\text{take}(r, l, c)\)
    • pre: \(\text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l\)
    • eff: \(\text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r\)
  • head: \(\text{put}(r, l, c)\)
    • pre: \(\text{loc}(r) = l, \text{loc}(c) = r\)
    • eff: \(\text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l\)

• Ranges
  • \(\mathcal{R}(r) = \text{Robots} = \{r1, r2\}\)
  • \(\mathcal{R}(l) = \mathcal{R}(m) = \text{Locs} = \{d1, d2, d3\}\)
  • \(\mathcal{R}(c) = \text{Containers} = \{c1, c2\}\)
**Actions**

- $\mathcal{A}$ = set of action templates
- Example $\mathcal{A}$
  - Contains three action templates
    - head: $\text{move}(r, l, m)$
      - pre: $\text{loc}(r) = l, \text{adj}(l, m)$
      - eff: $\text{loc}(r) \leftarrow m$
    - head: $\text{take}(r, l, c)$
      - pre: $\text{cargo}(r) = \text{nil}$, $\text{loc}(r) = l, \text{loc}(c) = l$
      - eff: $\text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r$
    - head: $\text{put}(r, l, c)$
      - pre: $\text{loc}(r) = l, \text{loc}(c) = r$
      - eff: $\text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l$
- Ranges
  - $\mathcal{R}(r) = \text{Robots} = \{r1, r2\}$
  - $\mathcal{R}(l) = \mathcal{R}(m) = \text{Locs} = \{d1, d2, d3\}$
  - $\mathcal{R}(c) = \text{Containers} = \{c1, c2\}$

- **Action $a$: ground instance of an action template $\alpha \in \mathcal{A}$**
  - Replace each parameter $t$ occurring in $\alpha$ with a value from $\mathcal{R}(t)$

- Example action:
  - $\text{move}(r1, d1, d2)$
    - pre: $\text{loc}(r1) = d1, \text{adj}(d1, d2)$
    - eff: $\text{loc}(r1) \leftarrow d2$

- Action space $\mathcal{A}$
  - $\mathcal{A} = \{\text{all actions we can get from } \mathcal{A}\}$
  - $\mathcal{A} = \{\text{all ground instances of elements of } \mathcal{A}\}$

---

How many move actions exist?
Applicability

• Action $a$ is **applicable** in state $s$ if
  • For every positive literal $l \in \text{pre}(\alpha)$, $l$ is in $s$ or in one of the rigid relations
  • For every negative literal $\neg l \in \text{pre}(\alpha)$, $l$ is not in $s$ nor in any rigid relations

• Example
  • Rigid relation: $\text{adj} = \{(d1, d2), (d2, d1), (d1, d3), (d3, d1)\}$
  • State $s_1 = \{\text{cargo}(r1) = \text{nil}, \text{cargo}(r2) = \text{nil}, \text{loc}(r1) = d1, \text{loc}(r2) = d2, \text{loc}(c1) = d1, \text{loc}(c2) = d2\}$
  • Action template $\text{move}(r, l, m)$
    • pre: $\text{loc}(r) = l, \text{adj}(l, m)$
    • eff: $\text{loc}(r) \leftarrow m$
  • Ranges
    • $\mathcal{R}(r) = \text{Robots}$
    • $\mathcal{R}(l) = \mathcal{R}(m) = \text{Locs}$

• Applicable action in $s_1$
  • $\text{move}(r1, d1, d2)$
    • pre: $\text{loc}(r1) = d1$, $\text{adj}(d1, d2)$
    • eff: $\text{loc}(r1) \leftarrow d2$

• Not applicable action in $s_1$
  • $\text{move}(r1, d2, d1)$
    • pre: $\text{loc}(r1) = d2$, $\text{adj}(d2, d1)$
    • eff: $\text{loc}(r1) \leftarrow d1$
Computing Prediction Function $\gamma$

- If action $a$ is applicable in state $s$
  - $\gamma(s, a) = \{(x, w) | eff(a) \text{ contains the effect } x \leftarrow w\}$
    $\cup \{(x, w) \in s | x \text{ isn’t the target of any effect in } eff(a)\}$

- Example
  - State $s_1 = \{\text{cargo}(r1) = \text{nil}, \text{cargo}(r2) = \text{nil},$
    $\text{loc}(r1) = d1, \text{loc}(r2) = d2, \text{loc}(c1) = d1, \text{loc}(c2) = d2\}$
  - Action $\text{take}(r2, d2, c2)$
    - pre: $\text{cargo}(r2) = \text{nil}, \text{loc}(r2) = d2, \text{loc}(c2) = d2$
    - eff: $\text{cargo}(r2) \leftarrow c2, \text{loc}(c2) \leftarrow r2$
  - $s_2 = \gamma(s_1, \text{take}(r2, d2, c2)) = \{\text{cargo}(r1) = \text{nil}, \text{cargo}(r2) = c2,$
    $\text{loc}(r1) = d1, \text{loc}(r2) = d2, \text{loc}(c1) = d1, \text{loc}(c2) = r2\}$
State-Variable Planning Domain

• Let
  
  \( B = \) finite set of objects
  \( R = \) finite set of rigid relations over \( B \)
  \( X = \) finite set of state variables
    • for every state variable \( x, R(x) \subseteq B \)
  \( S = \) state space over \( X \)
    = \{all variable-assignment functions that have sensible interpretations\}
  \( A = \) finite set of action templates
    • for every parameter \( t, R(t) \subseteq B \)
  \( A = \) \{all ground instances of action templates in \( A \)\}
  \( \gamma(s, a) = \{ (x, w)| eff(a) \text{ contains the effect } x \leftarrow w \} \)
    \( \cup \{ (x, w) \in s| x \text{ isn’t the target of any effect in } eff(a) \} \)

• Then \( \Sigma = (S, A, \gamma) \) is a state-variable planning domain
**Plans**

- **Plan**: sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$
  - $\text{cost}(\pi) = \sum_i \text{cost}(a_i)$
  - Length of $\pi = n$
- $\pi$ is **applicable** in $s_0$ if the actions in $\pi$ can be applied in the order given,
  - i.e., there are states $s_1, s_2, \ldots, s_n$ such that $\gamma(s_0, a_1) = s_1, \gamma(s_1, a_2) = s_2, \ldots, \gamma(s_{n-1}, a_n) = s_n$
  - If so, then define $\gamma(s_0, \pi) = s_n$

- **Example**
  - $s_0 = \{ \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \}$
  - $\pi = \langle \text{move}(r1, d3, d1), \text{take}(r1, d1, c1), \text{move}(r1, d1, d3) \rangle$
    - $\text{cost}(\pi) = 3$ (default)
    - $\gamma(s_0, \pi) = \{ \text{loc}(r1) = d3, \text{cargo}(r1) = c1, \text{loc}(c1) = r1 \}$
State Space

- Directed graph
  - Nodes = states of the world
  - Edges: $\gamma$
- If $\pi = \langle a_1, a_2, ..., a_n \rangle$ is applicable in $s_0$, it produces a path $\langle s_1, s_2, ..., s_n \rangle$
  - $\gamma(s_0, a_1) = s_1,$
  - $\gamma(s_1, a_2) = s_2, ... ,$
  - $\gamma(s_{n-1}, a_n) = s_n$
Planning Problems

- **State-variable planning problem** \( P = (\Sigma, s_0, g) \)
  - State-variable representation of a classical planning problem
  - \( \Sigma = (S, A, \gamma) \) is a state-variable planning domain
  - \( s_0 \in S \) is the initial state
  - \( g \) is a set of ground literals called the goal

- \( S_g = \{ \text{all states in } S \text{ that satisfy } g \} = \{ s \in S \mid s \cup R \text{ contains every positive literal in } g, \text{ and none of the negative literals in } g \} \)

- If \( \gamma(s_0, \pi) \in S_g \), then \( \pi \) is a solution for \( P \)

- Example
  - \( s_0 = \{ \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
  - \( \text{adj} = \{(d1, d2), (d2, d1), (d1, d3), (d3, d1)\} \)
  - \( g = \{ \text{cargo}(r1) = c1 \} \)
  - \( \pi = \{ \text{move}(r1, d2, d1), \text{take}(r1, d1, c1) \} \)

**How many solutions of length 3 exist?**
Classical Representation

• Motivation
  • The field of AI planning started out as automated theorem proving
  • It still uses a lot of that notation

• Classical representation is equivalent to state-variable representation
  • Represents both rigid and varying properties using logical predicates
    • \( \text{adj}(l, m) \) - location \( l \) is adjacent to location \( m \)
    • \( \text{loc}(r) = l \) \( \rightarrow \) \( \text{loc}(r, l) \) - robot \( r \) is at location \( l \)
    • \( \text{loc}(c) = r \) \( \rightarrow \) \( \text{loc}(c, r) \) - container \( c \) is on robot \( r \)
    • \( \text{cargo}(r) = c \) \( \rightarrow \) \( \text{loaded}(r) \) - robot \( r \) is loaded with a container

• State \( s = \) a set of ground atoms
  • \( s_0 = \{ \text{adj}(d_1, d_2), \text{adj}(d_2, d_1), \text{adj}(d_1, d_3), \text{adj}(d_3, d_1), \text{loc}(c_1, d_1), \text{loc}(r_1, d_2) \} \)
**Classical Representation**

- **Equivalent expressive power**
  - Each can be converted to the other in linear time and space
  
  \[ p(t_1, \ldots, t_k) \] becomes \[ x_p(t_1, \ldots, t_k) = 1 \]

- **Planning operator** \( \equiv \text{action template (next slide)} \)

**Worst case complexity: EXPSPACE**

- Time needed to solve a classical planning problem may be exponential in the size of the problem description
Classical Planning Operators

- Operator \( o = (\text{head}(o), \text{pre}(o), \text{eff}(o)) \)
  - \( \text{pre}(o), \text{eff}(o) \) are sets of literals

- Action: a ground instance

- Translation from \( \alpha \) to \( o \)
  - Precondition \( x(t_1, \ldots, t_k) = v \)
    - \( p_x(t_1, \ldots, t_k, v) \)
  - Precondition \( x(t_1, \ldots, t_k) \neq v \)
    - \( \neg p_x(t_1, \ldots, t_k, v) \)
  - Effect \( x(t_1, \ldots, t_k) \leftarrow v' \)
    - \( p_x(t_1, \ldots, t_k, v') \)
    - If \( p_x(t_1, \ldots, t_k, v) \in \text{pre}(o) \) for some \( v \):
      - Add new effect \( \neg p_x(t_1, \ldots, t_k, v) \)
    - Otherwise
      - Add new parameter \( u \) to \( \text{head}(o) \)
      - Add new precondition \( p_x(t_1, \ldots, t_k, u) \)
      - Add new effect \( \neg p_x(t_1, \ldots, t_k, u) \)

- May have twice as many effects and parameters as action template
  - From operator to template: same number

- Action templates
  - head: \( \text{move}(r, l, m) \)
    - pre: \( \text{loc}(r) = l, \text{adj}(l, m) \)
    - eff: \( \text{loc}(r) \leftarrow m \)
  - head: \( \text{take}(r, l, c) \)
    - pre: \( \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l \)
    - eff: \( \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r \)
  - head: \( \text{put}(r, l, c) \)
    - pre: \( \text{loc}(r) = l, \text{loc}(c) = r \)
    - eff: \( \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l \)

- Classical planning operators
  - head: \( \text{move}(r, l, m) \)
    - pre: \( \text{loc}(r, l), \text{adj}(l, m) \)
    - eff: \( \neg \text{loc}(r, l), \text{loc}(r, m) \)
  - head: \( \text{take}(r, l, c) \)
    - pre: \( \neg \text{loaded}(r), \text{loc}(r, l), \text{loc}(c, l) \)
    - eff: \( \text{loaded}(r), \text{loc}(c, r), \neg \text{loc}(c, l) \)
  - head: \( \text{put}(r, l, c) \)
    - pre: \( \text{loc}(r, l), \text{loc}(c, r) \)
    - eff: \( \neg \text{loaded}(r), \text{loc}(c, l), \neg \text{loc}(c, r) \)
PDDL

• Language for defining planning domains and problems
  • LISP-like syntax
• Original version ≈ 1996
  • Just classical planning
• Multiple revisions and extensions
  • Different subsets accommodate different kinds of planning

• We’ll discuss the classical-planning subset
  • Chapter 2 of the PDDL book
Example domain

• Classical planning operators
  • $move(r, l, m)$
    • pre: $loc(r, l), adj(l, m)$
    • eff: $\neg loc(r, l), loc(r, m)$
  • $take(r, l, c)$
    • pre: $\neg loaded(r), loc(r, l), loc(c, l)$
    • eff: $loaded(r), loc(c, r), \neg loc(c, l)$
  • $put(r, l, c)$
    • pre: $loc(r, l), loc(c, r)$
    • eff: $\neg loaded(r), loc(c, l), \neg loc(c, r)$

(define (domain example-domain-1)
  (requirements :negative-preconditions)

  (:action move
    :parameters (?r ?l ?m)
    :precondition (and (loc ?r ?l)
                       (adj ?l ?m))
    :effect (and (not (loc ?r ?l))
                (loc ?r ?m)))

  (:action take
    :parameters (?r ?l ?c)
    :precondition (and (loc ?r ?l)
                       (loc ?c ?l)
                       (not (loaded ?r)))
    :effect (and (not (loc ?r ?l))
                (loc ?r ?m)))

  (:action put
    :parameters (?r ?l ?c)
    :precondition (and (loc ?r ?l)
                       (loc ?c ?r))
    :effect (and (loc ?c ?l)
                (not (loc ?c ?r))
                (not (loaded ?r))))
Example problem

• Initial state $s_0 = \{\text{adj}(d1, d2), \text{adj}(d2, d1), \text{adj}(d1, d3), \text{adj}(d3, d1), \text{loc}(c1, d1), \text{loc}(r1, d2)\}$

• Goal state $g = \{\text{loc}(c1, r1)\}$
Example typed domain

(define (domain example-domain-2)
  (:requirements
   :negative-preconditions
   :typing)

  (:types
   location movable-obj - object
   robot container - movable-obj)

  (:predicates
   (loc ?r - movable-obj
        ?l - location)
   (loaded ?r - robot)
   (adjacent ?l ?m - location))

  (:action move
   :parameters (?r - robot
                 ?l ?m - location)
               "<as before>"

  (:action take
   :parameters (?r - robot
                 ?l - location
                 ?c - container)
               "<as before>"

  (:action put
   :parameters (?r - robot
                 ?l - location
                 ?c - container)
               "<as before>"
Example typed problem

(define (problem example-problem-1) (:domain example-domain-1))

(:init
  (adj d1 d2)
  (adj d2 d1)
  (adj d1 d3)
  (adj d3 d1)
  (loc c1 d1)
  (loc r1 d2)

(:goal (loc c1 r1)))

(define (problem example-problem-2) (:domain example-domain-2))

(:objects
  r1 - robot
  c1 - container
  loc1 loc2 loc3 - location)

(:init
  (adjacent d1 d2)
  (adjacent d2 d1)
  (adjacent d1 d3)
  (adjacent d3 d1)
  (loc c1 d1)
  (loc r1 d2)

(:goal (loc c1 r1)))
Intermediate Summary

• State-variable representation
  • State-transition systems, classical planning assumptions
  • Classical planning problems, plans, solutions
  • Objects, rigid properties
  • Varying properties, state variables, states as functions
  • Action templates, actions, applicability, $\gamma$
  • State-variable planning domains, plans, problems, solutions
  • Comparison with classical representation

• Classical fragment of PDDL
  • Planning domains, planning problems
  • untyped, typed
Outline per the Book

2.1 State-variable representation
   • State = \{values of variables\}; action = changes to those values

2.2 Forward state-space search
   • Start at initial state, look for sequence of actions that achieve goal

2.3 Heuristic functions
   • How to guide a forward state-space search

2.4 Backward search
   • Start at goal state, go backwards toward initial state

2.5 Plan-space search
   • Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan
Planning as Search

- Nearly all planning procedures are search procedures
  - **Search tree**: the data structure the procedure uses to keep track of which paths it has explored

Search-Tree Terminology

- **Node**: a pair $\nu = (\pi, s)$
  - $s = \gamma(s_0, \pi)$
  - In practice, $\nu$ may contain other things
    - pointer to parent, $cost(\pi)$, ...
  - $\pi$ not always stored explicitly, can be computed from the parent pointers

- **Children** of $\nu = \{(\pi \cdot a, \gamma(s, a)) \mid a \text{ is applicable in } s\}$
  - $\pi \cdot a$: concatenation of $\pi$ and $a$

- **Successors** of $\nu$
  - Children, children of children, etc.

- **Ancestors** of $\nu$
  - Nodes that have $\nu$ as a successor

- **Initial/starting node**: $\nu_0 = (\langle \rangle, s_0)$
  - Root of the search tree

- **Path** in the search space
  - Sequence $\langle \nu_0, \nu_1, \ldots, \nu_n \rangle$ s.t. each $\nu_i$ is a child of $\nu_{i-1}$

- **Height** of search space
  - Length of longest acyclic path from $\nu_0$

- **Depth** of $\nu$
  - Length of path from $\nu_0$ to $\nu$, $length(\pi)$

- **Branching factor** of $\nu$
  - Number of children

- **Branching factor** of search tree
  - max branching factor of the nodes

- **Expand** $\nu$
  - Generate all children

![Diagram](image-url)
Forward Search

- **Nondeterministic algorithm**
  - *Sound*: if an execution trace returns a plan $\pi$, it’s a solution
  - *Complete*: if the planning problem is solvable, at least one of the possible execution traces will return a solution

- **Represents a class of deterministic search algorithms**
  - Depends on how you implement the nondeterministic choice
    - Which leaf node to expand next, which nodes to prune
  - Won’t necessarily be complete

---

**Forward-search**($\Sigma, s_0, g$)

1. $s \leftarrow s_0$
2. $\pi \leftarrow \langle \rangle$
3. loop
   - if $s$ satisfies $g$ then
     return $\pi$
   - $A' \leftarrow \{a \in A \mid a \text{ is applicable in } s\}$
   - if $A' = \emptyset$ then
     return failure
   - nondeterministically choose $a \in A'$
   - $s \leftarrow \gamma(s, a)$
   - $\pi \leftarrow \pi \cdot a$

---

```
Arad
   /   \
Sibiu  Fagaras Oradea Rimnicu Vilcea

Arad
  /   \
Sibiu  Bucharest

Arad
  /   \
Sibiu

Fagaras
  /   \
Sibiu

Oradea
  /   \
Sibiu

Rimnicu Vilcea
  /   \
Pitești

Timisoara
  /   \
Sibiu

Zerind
  /   \
Sibiu
```

- Arad to Zerind: $449=75+374$
- Arad to Sibiu: $447=118+329$
- Bucharest to Craiova: $526=366+160$
- Bucharest to Pitesti: $417=317+100$
- Fagaras to Craiova: $526=366+160$
- Fagaras to Pitesti: $553=300+253$
- Oradea to Pitesti: $417=317+100$
- Sibiu to Bucharest: $450=450+0$
- Sibiu to Craiova: $591=338+253$
- Sibiu to Fagaras: $591=338+253$
- Sibiu to Pitesti: $553=300+253$
- Sibiu to Sibiu: $447=118+329$
- Timisoara to Sibiu: $446=280+366$
Deterministic Version

- Special cases
  - depth-first
  - breadth-first
  - A*

- Classify by
  - how they select nodes (step i)
  - how they prune nodes (step ii)

- Cycle checks during pruning:
  - Remove from children every node \((\pi, s)\)
    that has an ancestor \((\pi', s')\)
    s.t. \(s' = s\)
  - In classical planning, \(S\) is finite
    - Cycle-checking will guarantee termination
Breadth-first search (BFS)

- (i) select $(\pi, s) \in \text{Frontier}$ with smallest $\text{length}(\pi)$
  - tie-breaking rule: select oldest
- (ii) remove every $(\pi, s) \in \text{Children} \cup \text{Frontier}$ s.t. $s$ is in Expanded
  - thus expand states at most once
- Properties
  - Terminates
  - Returns solution if one exists
    - Shortest, but not least-cost (except if shortest=least-cost)
  - Worst-case complexity:
    - memory $O(|S|)$
    - running time $O(b|S|)$
where
- $b = \text{max branching factor}$
- $|S| = \text{number of states in } S$

Deterministic-Search($\Sigma, s_0, g$)

```
Frontier ← {((\emptyset), s_0)}
Expanded ← \emptyset

while Frontier ≠ \emptyset do
    select a node $v = (\pi, s) \in \text{Frontier}$ (i)
    remove $v$ from Frontier
    add $v$ to Expanded
    if $s$ satisfies $g$ then
        return $\pi$
    Children ←
        \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\}
    prune 0 or more nodes from
    Children, Frontier, Expanded (ii)
    Frontier ← Frontier \cup Children

return failure
```
Depth-First Search (DFS)

- (i) Select \((\pi, s) \in \text{Children}\) that has largest \(\text{length}(\pi)\)
  - Possible tie-breaking rules: left-to-right, smallest \(\text{height}(s)\)

- (ii) do cycle-checking, then prune all nodes that recursive DFS would discard
  - Repeatedly remove from \(\text{Expanded}\) any node that has no children in \(\text{Children} \cup \text{Frontier} \cup \text{Expanded}\)

- Properties
  - Terminates
  - Returns solution if there is one
    - No guarantees on quality
  - Worst-case complexity
    - Running time \(O(b^l)\)
    - Memory \(O(bl)\)

Where
- \(b = \text{max branching factor}\)
- \(l = \text{max depth of any node}\)

\[
\text{Deterministic-Search}(\Sigma, s_0, g)
\]

\[
\begin{align*}
\text{Frontier} & \leftarrow \{(\langle \rangle, s_0)\} \\
\text{Expanded} & \leftarrow \emptyset \\
\text{while} \ & \text{Frontier} \neq \emptyset \ 	ext{do} \\
& \text{select a node } v = (\pi, s) \in \text{Frontier} \ (i) \\
& \text{remove } v \text{ from } \text{Frontier} \\
& \text{add } v \text{ to } \text{Expanded} \\
& \text{if } s \text{ satisfies } g \text{ then} \\
& \quad \text{return } \pi \\
& \text{Children} \leftarrow \\
& \quad \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\} \\
& \text{prune 0 or more nodes from} \\
& \quad \text{Children}, \text{Frontier}, \text{Expanded} \ \ (ii) \\
\text{Frontier} & \leftarrow \text{Frontier} \cup \text{Children} \\
\text{return} & \text{failure}
\end{align*}
\]
Uniform-Cost Search

- (i) Select \((\pi, s) \in Children\) that has smallest \(\text{cost}(\pi)\)
- (ii) Prune every \((\pi, s) \in Children \cup Frontier\) such that \(Expanded\) already contains a node \((\pi', s)\)
  - \(\text{cost}(\pi') \leq \text{cost}(\pi)\), so we only keep the least-cost path to \(s\)

Properties
- Terminates
- Finds optimal solution if one exists
- Worst-case complexity
  - Time \(O(b|S|)\)
  - Memory \(O(|S|)\)
  where
  - \(b = \text{max branching factor}\)
  - \(|S| = \text{number of states in } S\)

Deterministic-Search\((\Sigma, s_0, g)\)

- \(\text{Frontier} \leftarrow \{()\}, s_0\}\)
- \(\text{Expanded} \leftarrow \emptyset\)
- while \(\text{Frontier} \neq \emptyset\) do
  - select a node \(v = (\pi, s) \in \text{Frontier} \) (i)
  - remove \(v\) from \(\text{Frontier}\)
  - add \(v\) to \(\text{Expanded}\)
  - if \(s\) satisfies \(g\) then
    - return \(\pi\)
  - \(\text{Children} \leftarrow \{((\pi, a), \gamma(s, a)) \mid s\text{ satisfies } \text{pre}(a)\}\)
  - prune 0 or more nodes from \(\text{Children, Frontier, Expanded} \) (ii)
  - \(\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}\)
- return failure
Heuristic Function

- **Motivation**: get to a solution quickly by selecting nodes close to the goal
  - Compare: A*
- **Let** \( h^*(s) = \min\{\text{cost}(\pi) \mid \gamma(s, \pi) \text{ satisfies } g\} \)
  - Note that \( h^*(s) \geq 0 \) for all \( s \)
- **Heuristic function** \( h(s) \):
  - Returns an estimate of \( h^*(s) \)
    - Assume \( h(s) \geq 0 \) for all \( s \)
  - Properties
    - \( h \) is admissible if for every \( s, h(s) \leq h^*(s) \)
    - \( h \) is \( \varepsilon \)-admissible if for every \( s, h(s) \leq h^*(s) + \varepsilon \)

- **Let** \( \nu = (\pi, s) \) be a node
  - \( f^*(\nu) = \text{cost}(\pi) + h^*(s) \)
    - Min cost of all paths to goal that start with \( \pi \)
  - \( f(\nu) = \text{cost}(\pi) + h(s) \)
    - Estimate of \( f^*(\nu) \)
Example

• State $s$ = what city you are in
• Action: follow road from $s$ to a neighboring city
• $h^*(s) =$ length of shortest sequence of roads from $s$ to Bucharest
• $h(s) =$ straight-line distance from $s$ to Bucharest
  • *domain-specific*; later we will discuss *domain-independent*
• $f^*( (\pi, s)) =$ length of $(\pi +$ shortest sequence of roads from $s$ to Bucharest)
A*

• (i) Select a node \( \nu = (\pi, s) \) in \( \text{Frontier} \) that has smallest value of \( f(\nu) = \text{cost}(\pi) + h(s) \)
  - Tie-breaking rule: choose oldest

• (ii) for every node \( \nu = (\pi, s) \) in \( \text{Children} \)
  - if \( \text{Children} \cup \text{Frontier} \cup \text{Expanded} \) contains more than one node for \( s \)
    - then it has multiple paths to \( s \)
    - Keep only the one with the lowest f-value
  - Tie-breaking rule: keep oldest

• Properties
  • After upcoming example

Deterministic-Search(\( \Sigma, s_0, g \))

\[
\text{Frontier} \leftarrow \{ (\emptyset, s_0) \}
\]
\[
\text{Expanded} \leftarrow \emptyset
\]

\text{while} \ \text{Frontier} \neq \emptyset \ \text{do}

  \text{select a node} \ \nu = (\pi, s) \in \text{Frontier} \ (i)
  \text{remove} \ \nu \ \text{from} \ \text{Frontier}
  \text{add} \ \nu \ \text{to} \ \text{Expanded}

  \text{if} \ s \ \text{satisfies} \ g \ \text{then}
    \text{return} \ \pi

\text{Children} \leftarrow
\{(\pi.a, \gamma(s, a)) \mid s \ \text{satisfies} \ \text{pre}(a)\}

\text{prune} \ 0 \ \text{or more nodes from} \ \text{Children}, \ \text{Frontier}, \ \text{Expanded} \ \ (ii)

\text{Frontier} \leftarrow \text{Frontier} \cup \text{Children}

\text{return} \ \text{failure}
straight-line dist. from s to Bucharest:

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobreta: 242
- Fagaras: 176
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamt: 234
- Oradea: 380
- Pitesti: 100
- Rimnicul Vilcea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from $s$ to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad  366
Bucharest  0
Craiova  160
Dobreta  242
Fagaras  176
Iasi  226
Lugoj  244
Mehadia  241
Neamt  234
Oradea  380
Pitesti  100
Rimnicu Vilcea  193
Sibiu  253
Timisoara  329
Urziceni  80
Vaslui  199
Zerind  374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobrogea 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
Properties of A*

• In classical planning problems, A* will always terminate
• Completeness: if the problem is solvable, A* will return a solution
  • If $h$ is admissible, then the solution will be optimal (least cost)
  • If $h$ is $\varepsilon$-admissible, then the solution will be $\varepsilon$-optimal
• If $h$ is monotone then
  • $f(\nu) \leq f(\nu')$ for every child $\nu'$ of a node $\nu$
  • A* will expand nodes in non-decreasing order of $f$ values
  • A* will never prune any nodes from Expanded
  • A* will expand no state more than once
• Definition: $h$ dominates $h'$ if $h'(s) \leq h(s) \leq h^*(s)$ for every $s$
  • If $h$ dominates $h'$ then (assuming ties are always resolved in favor of the same node)
    • A* will never expand more nodes with $h$ than with $h'$
    • In most cases A* will expand fewer nodes with $h$ than with $h'$
• A* needs to store every node it visits
  • Running time and memory both $O(b|S|)$ in worst case
  • With good heuristic function, usually much smaller
Greedy Best-First Search (GBFS)

- Find a solution as quickly as possible, even if it isn’t optimal
  - Select nodes that are likely to be on the least-cost path from where you are now
- (i) Select a node \((\pi, s) \in \text{Frontier}\) that has smallest \(h(s)\)
- (ii) same as in A*: for every node \(v = (\pi, s)\) in \(\text{Children}\)
  - if \(\text{Children} \cup \text{Frontier} \cup \text{Expanded}\) contains more than one node for \(s\)
    - then it has multiple paths to \(s\)
    - Keep only the one with the lowest \(f\)-value
  - Tie-breaking rule: keep oldest
- Properties
  - Terminates
  - Returns a solution if one exists
    - Often near-optimal
    - will usually find it quickly

Deterministic-Search\((\Sigma, s_0, g)\)

\[
\begin{align*}
\text{Frontier} & \leftarrow \{(\langle \rangle, s_0)\} \\
\text{Expanded} & \leftarrow \emptyset \\
\text{while} \ \text{Frontier} \neq \emptyset \ \text{do} \\
& \text{select a node } v = (\pi, s) \in \text{Frontier} (i) \\
& \text{remove } v \text{ from } \text{Frontier} \\
& \text{add } v \text{ to } \text{Expanded} \\
& \text{if } s \text{ satisfies } g \ \text{then} \\
& \hspace{1cm} \text{return } \pi \\
& \text{Children} \leftarrow \\
& \hspace{1cm} \{(\pi.a, \gamma(s,a)) \mid s \text{ satisfies } \text{pre}(a)\} \\
& \text{prune } 0 \text{ or more nodes from} \\
& \hspace{1cm} \text{Children, Frontier, Expanded} \ \text{(ii)} \\
& \text{Frontier} \leftarrow \text{Frontier} \cup \text{Children} \\
\text{return failure}
\end{align*}
\]
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad       366
Bucharest  0
Craiova    160
Dobrogea   242
Fagaras    176
Iasi        226
Lugoj       244
Mehadia    241
Neamț      234
Oradea     380
Pitești    100
Rimnița-Vilcea 193
Sibiu      253
Timișoara  329
Urziceni    80
Vaslui     199
Zerind     374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobretja 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

• expanded 4 nodes instead of 6
• solution cost 450 instead of 418
Depth-First Branch and Bound (DFBB)

- (i) same as DFS
  - Select $v = (\pi, s) \in \text{Children}$ that has largest $\text{length}(\pi)$
  - Tie-breaking: smallest $\text{height}(s)$

- (ii) Prune
  - Like DFS
    - do cycle-checking and prune what recursive DFS would discard
  - Additional pruning during node expansion:
    - If $f(v) \geq c^*$, then discard $v$

- Properties
  - Termination, completeness, optimality same as A*
  - Usually less memory than A*, but more time
  - Worst-case like DFS:
    - $O(bl)$ memory
    - $O(bl)$ running time
straight-line dist. from $s$ to Bucharest

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobroșe: 242
- Făgăraș: 176
- Iași: 226
- Lugoj: 244
- Mehadia: 241
- Neamț: 234
- Oradă: 380
- Pitesti: 100
- Rimnicu Vîlcea: 193
- Sibiu: 253
- Timișoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374
straight-line dist. from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist.
from s to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
straight-line dist. from s to Bucharest
Arad  366
Bucharest  0
Craiova  160
Dobreta  242
Fagaras  176
Iasi  226
Lugoj  244
Mehadia  241
Neamt  234
Oradea  380
Pitesti  100
Rimnicu Vilcea  193
Sibiu  253
Timisoara  329
Urziceni  80
Vaslui  199
Zerind  374

\[ c^* = 418 \]
\[ \pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle \]
straight-line dist. from $s$ to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobroța 242
Făgărăș 176
Iasi 226
Lugoj 244
Mehadia 241
Neamț 234
Oradea 380
Pitesti 100
Rimnicu Vîlcea 193
Sibiu 253
Timișoara 329
Urziceni 80
Vaslui 199
Zerind 374

$c^* = 418$

$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$
straight-line dist. from $s$ to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobreta 242
Fagaras 176
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 100
Rimnicu Vilcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374

c* = 418
$\pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle$
\[ c^* = 418 \]
\[ \pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle \]
\[ c^* = 418 \]
\[ \pi^* = \langle a_{AS}, a_{SR}, a_{RP}, a_{PB} \rangle \]
Iterative Deepening Search (IDS)

• Example:
  - Expand $a$
    - $(k = 1)$
  - Expand $a, b, c$
    - $(k = 2)$
  - Expand $a, b, c, d, e, f, g$
    - $(k = 3)$
  - Expand $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o$
    - $(k = 4)$
  - Solution path $\langle a, c, g, o \rangle$
  - Total number of node expansions:
    - $1 + 3 + 7 + 15 = 26$

• If goal is at depth $d$ and branching factor is 2:

$$
\sum_{i=1}^{d} (2^i - 1) = \left( \sum_{i=1}^{d} 2^i \right) - d
$$

$$
= 2^{d+1} - 2^d - d = O(2^d)
$$

IDS($\Sigma, s_0, g$)

for $k = 1$ to $\infty$ do
  $\pi^* \leftarrow$ do DFS, backtracking at every node of depth $k$
  if $\pi^* \neq$ failure then
    return $\pi^*$
  if the search generated no nodes of depth $k$ then
    return failure
Iterative Deepening Search (IDS)

• If goal is at depth \( d \) and branching factor is \( b \):

\[
\sum_{i=1}^{d} (b^i - 1) = \left(\sum_{i=1}^{d} b^i\right) - d
\]

\[= b^{d+1} - b - d = O(b^d)\]

• Properties
  • Termination, completeness, optimality
  • same as BFS
  • Worst-case complexity
  • Memory requirement \( O(bd) \)
    • vs. \( O(b^d) \) with BFS
  • Worst-case running time \( O(b^d) \)
    • vs. \( O(b^i) \) for DFS
    • If the number of nodes at depth \( d \) grows exponentially with \( d \)

where

• \( b = \text{max branching factor} \)
• \( d = \text{min solution depth if there is one, otherwise max depth of any node} \)

```
IDS(\Sigma, s_0, g)
for k = 1 to \( \infty \) do
  \( \pi^* \leftarrow \) do DFS, backtracking at every node of depth \( k \)
  if \( \pi^* \neq \text{failure} \) then
    return \( \pi^* \)
  if the search generated no nodes of depth \( k \) then
    return failure
```
IDA*

• Properties
  • Termination, completeness, and optimality same as A*
  • Worst-case complexity
    • If $h$ is admissible, memory requirement $O(bd)$ rather than $O(b^d)$
    • If the number of nodes grows exponentially with $c$, running time $O(b^d)$
      • Can be much worse if the number of nodes grows subexponentially
        • e.g., real-valued costs
  • IDA* is not much used in practice

IDA*($\Sigma, s_0, g$)
\[ c \leftarrow 0 \]
\[ \text{loop} \]
\[ \pi^* \leftarrow \text{do DFS, backtracking whenever } f(v) > c \]
\[ \text{if } \pi^* \neq \text{failure then} \]
\[ \text{return } \pi^* \]
\[ \text{if DFS didn’t generate an } f(v) > c \text{ then} \]
\[ \text{return failure} \]
\[ c \leftarrow \text{the smallest } f(v) > c \text{ where backtracking occurred} \]
Discussion

• If $h$ is admissible, both A* and DFBB will return optimal solutions
  • Usually DFBB takes more time, A* takes more memory
  • A* better than DFBB in highly connected graphs (many paths to states)
    • DFBB can have exponentially worse running time than A*
  • DFBB best in problems where $S$ is a tree of uniform height, all solutions at the bottom (e.g., constraint satisfaction)
    • DFBB and A* have similar running time, A* takes exponentially more memory than DFBB

• DFS returns the first solution it finds
  • Less backtracking than DFBB, but solution can be very far from optimal

• GBFS returns the first solution it finds
  • With a good heuristic function, usually near-optimal without much backtracking
  • Used by most classical planners nowadays
Intermediate Summary

- Forward-search, Deterministic-Search
- Cycle-checking
- Breadth-first, depth-first, uniform-cost search
- A*, GBFS, DFBB
- IDS, IDA*
Outline per the Book

2.1 *State-variable representation*
   • State = \{values of variables\}; action = changes to those values

2.2 *Forward state-space search*
   • Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*
   • How to guide a forward state-space search

2.6 *Incorporating planning into an actor*
   • Online lookahead, unexpected events

2.4 *Backward search*
   • Start at goal state, go backwards toward initial state

2.5 *Plan-space search*
   • Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan
Heuristic Functions

• Planning problem $P$ in domain $\Sigma$
• Creating a heuristic function:
  • Weaken some of the constraints that
    • restrict what the states, actions, and plans are
    • restrict when an action or plan is applicable, what goals it achieves
    • increase the costs of actions and plans
• **Relaxed** planning domain $\Sigma' = (S', A', \gamma')$ and problem $P' = (\Sigma', s'_0, g')$
  • for every solution $\pi$ for $P$, $P'$ has a solution $\pi'$ with $cost'(\pi') \leq cost(\pi)$
• Suppose we have an algorithm $A$ for solving planning problems in $\Sigma'$
  • Heuristic function $h^A(s)$ for $P$:
    • Find a solution $\pi'$ for $(\Sigma', s, g')$; return $cost(\pi')$
    • If $A$ runs quickly, then $h^A$ may be a useful heuristic function
    • If $A$ always finds optimal solutions, then $h^A$ is admissible
Example from A*

- Relaxation: let vehicle travel in a straight line between any pair of cities
  - straight-line-distance ≤ distance by road
Domain-independent Heuristics

• Heuristic functions that can be used work in any classical planning problem
  • Additive-cost heuristic
  • Max-cost heuristic
  • Delete-relaxation heuristics
    • Optimal relaxed solution
    • Fast-forward heuristic
  • Landmark heuristics

In the book, but I’ll skip them
Delete-Relaxation

• Relaxation:
  • A state variable can have more than one value at the same time
  • When assigning a new value, keep the old one too

• Suppose state $s$ includes an atom $x = v$, action $a$ has effect $x \leftarrow w$
  • $\gamma^+(s, a)$ is a relaxed state
  • Includes both $x = v$ and $x = w$

• Example
  • $s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
  • $\text{move}(r1, d3, d1)$
    • Pre: $\text{loc}(r1) = d3$
    • Eff: $\text{loc}(r1) \leftarrow d1$
  • $\hat{s}_1 = \gamma^+(s_0, \text{move}(r1, d3, d1))$
    $= \{\text{loc}(r1) = d3, \text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
Relaxed States

- **Relaxed state** (or *r-state*):
  - Set \( \hat{s} \) of ground atoms that includes at least 1 value for each state variable
  - Represents \( \{ \text{all states that are subsets of } \hat{s} \} \)
  - Note: every state \( s \) is also a relaxed state that represents \( \{ s \} \)

- **Examples**
  - \( \hat{s}_1 = \{ \text{loc}(r1) = d1, \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
  - \( \hat{s}_2 = \gamma^+(\hat{s}_1, \text{take}(r1, d1, c1)) = \{ \text{loc}(r1) = d1, \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = r1, \text{loc}(c1) = d1, \text{cargo}(r1) = c1 \} \)
R-applicability

• An r-state $\hat{s}$ r-satisfies a set of literals $g$ if a set $s \subseteq \hat{s}$ satisfies $g$

• Action $a$ is r-applicable in $\hat{s}$ if $\hat{s}$ r-satisfies $\text{pre}(a)$
  • i.e., $\hat{s}$ contains a subset $s$ that satisfies the preconditions of $a$
  • If $a$ is r-applicable, then $\gamma^+(\hat{s}, a) = \hat{s} \cup \gamma(s, a)$

• $\pi = \langle a_1, ..., a_n \rangle$ is r-applicable in $\hat{s}_0$ if there are r-states $\hat{s}_1, \hat{s}_2, ..., \hat{s}_n$ such that
  • $a_1$ is r-applicable in $\hat{s}_0$ and $\gamma^+(\hat{s}_0, a_1) = \hat{s}_1$
  • $a_2$ is r-applicable in $\hat{s}_1$ and $\gamma^+(\hat{s}_1, a_2) = \hat{s}_2$
  • ...
  • $a_n$ is r-applicable in $\hat{s}_{n-1}$ and $\gamma^+(\hat{s}_{n-1}, a_n) = \hat{s}_n$

• In this case, $\gamma^+(\hat{s}_{n-1}, \pi) = \hat{s}_n$
Example

- $s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
- $move(r1, d3, d1)$
  - Pre: $\text{loc}(r1) = d3$
  - Eff: $\text{loc}(r1) \leftarrow d1$
- $\hat{s}_1 = \gamma^+(\hat{s}_1, move(r1, d1, c1)) = \{\text{loc}(r1) = d1, \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
- $take(r, l, c)$
  - pre: $\text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l$
  - eff: $\text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r$
- $\hat{s}_2 = \gamma^+(\hat{s}_1, take(r1, d1, c1)) = \{\text{loc}(r1) = d1, \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = r1, \text{loc}(c1) = d1, \text{cargo}(r1) = c1\}$
Relaxed Solution

- Planning problem $P = (\Sigma, s_0, g)$
- Plan $\pi$ is a relaxed solution for $P$ if $\gamma^+(\hat{s}_0, \pi)$ r-satisfies $g$
- Example:
  - Initial $s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
  - Goal states $g = \{\text{loc}(r1) = d3, \text{loc}(c1) = r1\}$
  - Plan $\pi = \langle \text{move}(r1, d3, d1), \text{take}(r1, c1, d1) \rangle$

- End state $\gamma^+(s_0, \pi) = \{\text{loc}(r1) = d1, \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = r1, \text{loc}(c1) = d1, \text{cargo}(r1) = c1\}$
Optimal Relaxed Solution Heuristic

• Given a planning problem \( P = (\Sigma, s_0, g) \)
• Optimal relaxed solution heuristic:
  • \( h^+(s) = \) minimum cost of all relaxed solutions for \( P \)
• Example:
  • Initial \( s_0 = \{ \text{loc}(r1) = d3, \ text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
  • Goal states \( g = \{ \text{loc}(r1) = d3, \text{loc}(c1) = r1 \} \)
  • \( \pi = \langle \text{move}(r1, d3, d1), \text{take}(r1, c1, d1) \rangle \)
    • \( \text{cost}(\pi) = 2 \)
  • No less-costly relaxed solution, so \( h^+(s_0) = 2 \)
Example

- \( s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\} \)
- In \( s_0 \), two applicable actions
  - \( a_1 = \text{move}(r1, d3, d1) \)
  - \( s_1 = \{\text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\} \)
  - \( a_2 = \text{move}(r1, d3, d2) \)
  - \( s_2 = \{\text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\} \)
- GBFS evaluates \( h^+(s_1) \) and \( h^+(s_2) \), and chooses to move to whichever is smaller.

\[
g = \{\text{loc}(r1) = d3, \text{loc}(c1) = r1\}
\]
Fast-Forward Heuristic

• Every state is also a relaxed state
• Every solution is also a relaxed solution

• $h^+(s) = \text{minimum cost of all relaxed solutions}$
  • Thus $h^+$ is admissible
  • Problem: computing it is NP-hard

• Fast-Forward Heuristic $h^{FF}$
  • An approximation of $h^+$ that is easier to compute
    • Upper bound on $h^+$
  • Name comes from a planner called *Fast Forward*
Preliminaries

- Let $A_1$ be a set of actions that are $r$-applicable in $\hat{s}$
  - Can apply them in any order and get same result
  - Define result of applying $A_1$ in $\hat{s}$ as

$$\gamma^+(\hat{s}, A_1) = \hat{s} \cup \bigcup_{a \in A_1} \text{eff}(a)$$

- Let $\hat{s}_1 = \gamma^+(\hat{s}_0, A_1)$
  - Suppose $A_2$ is a set of actions that are $r$-applicable in $\hat{s}_1$
  - Define $\gamma^+(\hat{s}_0, \langle A_1, A_2 \rangle) = \gamma^+(\hat{s}_1, A_2)$
  - ...
  - Define $\gamma^+(\hat{s}_0, \langle A_1, A_2, \ldots, A_n \rangle)$ in the obvious way
Fast-Forward Heuristic

\[ \text{HFF}(\Sigma, s, g) \]

// construct a relaxed solution \( \langle A_1, A_2, \ldots, A_k \rangle \):
\[ \hat{s}_0 \leftarrow s \]
for \( k = 1; k++ \); a subset of \( \hat{s}_k \) r-satisfies \( g \) do
\[ A_k = \{ \text{all actions r-applicable in } \hat{s}_{k-1} \} \]
\[ \hat{s}_k = \gamma^+(s_{k-1}, A_k) \]
if \( k > 1 \) and \( \hat{s}_k = \hat{s}_{k-1} \) then
\[ \text{return } \infty \] // there’s no solution

// extract minimal relaxed solution \( \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle \):
\[ \hat{g}_k = g \]
for \( i = k \) down to 1 do
\[ \hat{a}_i = \text{any minimal subset of } A_i \text{ such that } \gamma^+(\hat{s}_{i-1}, \hat{a}_i) \text{ r-satisfies } \hat{g}_i \]
\[ \hat{g}_{i-1} = (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i) \]
\[ \hat{h} \leftarrow \langle \hat{a}_1, \ldots, \hat{a}_k \rangle \]
\[ \text{return } \sum_{a \text{ is an action in } \hat{h}} \text{cost}(a) \] // upper bound on \( h^* \)

• Find a minimal relaxed solution and return its cost
  • Generates a sequence of successively larger r-states and sets of applicable actions until \( \hat{s}_k \) r-satisfies \( g \):
    \[ \hat{s}_0, A_1, \hat{s}_1, A_2, \hat{s}_2, \ldots, A_{k-1}, \hat{s}_{k-1}, A_k, \hat{s}_k \]
  • Extract minimal relaxed solution from that sequence
Fast-Forward Heuristic

\[ HFF(\Sigma, s, g) \]

// construct a relaxed solution \( \langle A_1, A_2, \ldots, A_k \rangle \):
\[ \hat{s}_0 \leftarrow s \]
for \( k = 1; k++ \) a subset of \( \hat{s}_k \) r-satisfies \( g \) do
\[ A_k = \{ \text{all actions } r \text{-applicable in } \hat{s}_{k-1} \} \]
\[ \hat{s}_k = \gamma^+(s_{k-1}, A_k) \]
if \( k > 1 \) and \( \hat{s}_k = \hat{s}_{k-1} \) then
\[ \text{return } \infty \] // there’s no solution
\[ \]
// extract minimal relaxed solution \( \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle \):
\[ \hat{g}_k = g \]
for \( i = k \text{ down to } 1 \) do
\[ \hat{a}_i = \text{any minimal subset of } A_i \text{ such that } \gamma^+(\hat{s}_{i-1}, \hat{a}_i) \text{ r-satisfies } \hat{g}_i \]
\[ \hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i) \]
\[ \hat{h} \leftarrow \langle \hat{a}_1, \ldots, \hat{a}_k \rangle \]
\[ \text{return } \sum_a \text{ is an action in } \pi^* \text{ cost}(a) \] // upper bound on \( h^+ \)

• Find a minimal relaxed solution and return its cost
• Define \( h^{FF} = \) the value returned by \( HFF(\Sigma, s, g) \)
  • Return value is ambiguous
    • Each \( \hat{a}_i \) in \( h^{FF}(s) \) is a minimal set of actions s.t. \( \gamma^+(\hat{s}_{i-1}, \hat{a}_i) \) r-satisfies \( \text{pre}(\hat{a}_i) \)
    • Depends on which minimal subsets we choose
Example (as before)

- \( s_0 = \{ \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
- In \( s_0 \), two applicable actions
  - \( a_1 = \text{move}(r1, d3, d1) \)
    - \( s_1 = \{ \text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
  - \( a_2 = \text{move}(r1, d3, d2) \)
    - \( s_2 = \{ \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
- GBFS using \( h^{FF} \)
  - Compute \( h^{FF}(s_1) \) and \( h^{FF}(s_2) \)
  - Move to whichever is smaller

\[ g = \{ \text{loc}(r1) = d3, \text{loc}(c1) = r1 \} \]
Example

Relaxed Planning Graph (RPG) from $\hat{s}_0 = s_1$ to $g$
(solid lines indicate preconditions/effects):

Atoms in $\hat{s}_0 = s_1$:  
- $\text{loc}(r1) = d1$
- $\text{loc}(c1) = d1$
- $\text{cargo}(r1) = \text{nil}$

Actions in $A_1$:  
- $\text{move}(r1, d1, d2)$
- $\text{move}(r1, d1, d3)$
- $\text{take}(r1, c1, d1)$

Atoms in $\hat{s}_1$:  
- $\text{loc}(r1) = d2$
- $\text{loc}(r1) = d3$
- $\text{loc}(c1) = r1$
- $\text{cargo}(r1) = c1$

$\langle A_1 \rangle$ is a relaxed solution

$\gamma^+(s_0, A_1)$ r-satisfies $g$

$g = \{\text{loc}(r1) = d3, \text{loc}(c1) = r1\}$

$s_1 = \{\text{loc}(r1) = d1, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}$
Example

Relaxed Planning Graph (RPG) from $\hat{s}_0 = s_1$ to $g$
(follow lines from atoms in $g$: all if precond.; one if eff.)

Atoms in $\hat{s}_0 = s_1$:
- $\text{loc(r1)} = d1$
- $\text{loc(c1)} = d1$
- $\text{cargo(r1)} = \text{nil}$

Actions in $A_1$:
- $\text{move(r1, d1, d2)}$
- $\text{move(r1, d1, d3)}$
- $\text{take(r1, c1, d1)}$

Atoms in $\hat{s}_1$:
- $\text{loc(r1)} = d1$
- $\text{loc(r1)} = d3$
- $\text{loc(c1)} = r1$
- $\text{cargo(r1)} = c1$

Cost of each action is 1, so $h_{FF}(s_1) = 2$

• $\langle \hat{a}_1 \rangle$ is a minimal relaxed solution

\[
\hat{g}_k = g \\
\text{for } i = k \text{ down to 1 do} \\
\hat{a}_i = \text{minimal subset of } A_i \text{ s.t. } \gamma^+(\hat{s}_{i-1}, \hat{a}_i) \text{ r-satisfies } \hat{g}_i \\
\hat{g}_{i-1} \leftarrow (\hat{g}_i \setminus \text{eff}(\hat{a}_i)) \cup \text{pre}(\hat{a}_i)
\]

$g = \{\text{loc(r1)} = d3, \text{loc(c1)} = r1\}$

$s_1 = \{\text{loc(r1)} = d1, \text{cargo(r1)} = \text{nil}, \text{loc(c1)} = d1\}$
Example

// construct a relaxed solution \( \langle A_1, A_2, ..., A_k \rangle \):
\( \hat{s}_0 \leftarrow s \)
for \( k = 1; k++ \); subset of \( \hat{s}_k \) r-satisfies g do
  \( A_k = \{ \text{all actions r-applicable in } \hat{s}_{k-1} \} \)
  \( \hat{s}_k = \gamma^+(s_{k-1}, A_k) \)
if \( k > 1 \) and \( \hat{s}_k = \hat{s}_{k-1} \) then
  return \( \infty \)  // there's no solution

RPG from \( \hat{s}_0 = s_2 \) to \( g \)

Atoms in \( \hat{s}_0 = s_2 \):  
\( \text{loc}(r1) = d2 \)  
\( \text{loc}(c1) = d1 \)  
\( \text{cargo}(r1) = \text{nil} \)

Actions in \( A_1 \):  
\( \text{move}(r1, d2, d3) \)
\( \text{move}(r1, d2, d1) \)

Atoms in \( \hat{s}_1 \):  
\( \text{loc}(r1) = d3 \)
\( \text{loc}(r1) = d2 \)
\( \text{loc}(c1) = d1 \)
\( \text{cargo}(r1) = \text{nil} \)

Actions in \( A_2 \):  
\( \text{move}(r1, d3, d2) \)
\( \text{move}(r1, d1, d2) \)
\( \text{move}(r1, d2, d1) \)
\( \text{move}(r1, d3, d1) \)
\( \text{move}(r1, d1, d3) \)
\( \text{move}(r1, d2, d3) \)
\( \text{take}(r1, c1, d1) \)
\( \text{cargo}(r1) = c1 \)
\( \text{loc}(c1) = r1 \)

Atoms in \( \hat{s}_2 \):  
\( \text{loc}(r1) = d2 \)
\( \text{loc}(c1) = d1 \)
\( \text{cargo}(r1) = \text{nil} \)

from \( \hat{s}_1 \):
\( \text{loc}(r1) = d3 \)
\( \text{loc}(r1) = d2 \)
\( \text{loc}(c1) = d1 \)
\( \text{cargo}(r1) = \text{nil} \)

\( \langle A_1, A_2 \rangle \) is a relaxed solution

\( g = \{ \text{loc}(r1) = d3, \text{loc}(c1) = r1 \} \)

\( s_2 = \{ \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \} \)
Example

RPG from $\hat{s}_0 = s_2$ to $g$

Atoms in $\hat{s}_0 = s_2$:  
- $\text{loc}(r1) = d2$
- $\text{loc}(c1) = d1$
- $\text{cargo}(r1) = \text{nil}$

Actions in $A_1$:  
- move($r1, d2, d3$)  
- move($r1, d1, d2$)  

Atoms in $\hat{s}_1$:  
- $\text{loc}(r1) = d3$
- $\text{loc}(c1) = d1$
- $\text{cargo}(r1) = \text{nil}$

Actions in $A_2$:  
- move($r1, d3, d2$)
- move($r1, d1, d2$)

Atoms in $\hat{s}_2$:  
- $\text{loc}(r1) = d2$
- $\text{loc}(c1) = d1$
- $\text{cargo}(r1) = \text{nil}$

$g = \{\text{loc}(r1) = d3, \text{loc}(c1) = r1\}$

$\langle \hat{a}_1 \rangle$

$\langle \hat{a}_2 \rangle$

Could have followed other eff. line (would have lead to $h^{FF}(s_1) = 2$)

- $\langle \hat{a}_1, \hat{a}_2 \rangle$ is a minimal relaxed solution
- Cost of each action is 1, so $h^{FF}(s_1) = 3$

\[ s_2 = \{\text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\} \]
Properties

• Running time is polynomial in $|A| + \sum_{x \in X} |\mathcal{R}(x)|$

• **Minimal solution** doesn’t mean *smallest cost*
  • A solution $\pi$ to $P$ is *minimal* if
    • no subsequence of $\pi$ is also a solution for $P$.
  • A solution $\pi$ to $P$ is *shortest* if
    • there is no solution $\pi'$ such that $|\pi'| < |\pi|$.
  • A solution $\pi$ to $P$ is *cost-optimal* if
    • $\text{cost}(\pi) = \min\{\text{cost}(\pi') | \pi' \text{ is a solution for } P\}$.

• $h^{FF}(s) =$ value returned by $HFF(\Sigma, s, g)$
  • $h^{FF}(s) = \sum \text{costs of } \hat{a}_1, \ldots, \hat{a}_k$
  • $h^{FF}(s) \geq h^+(s) =$ *smallest* cost of any relaxed plan from $s$ to goal
  • $h^{FF}$ not admissible
Example

• Suppose the goal atoms are c7, c8, c9. How many minimal solutions are there?
  • Assume default cost of 1
Landmark Heuristics

• \( P = (\Sigma, s_0, g) \) be a planning problem
• Let \( \varphi = \varphi_1 \lor \cdots \lor \varphi_m \) be a disjunction of ground atoms
• \( \varphi \) is a landmark for \( P \) if \( \varphi \) is true at some point in every solution for \( P \)
• Example landmarks
  • \( \text{loc}(r1) = d1 \)
  • \( \text{loc}(r1) = d3 \lor \text{loc}(r1) = d2 \)
  • \( \text{loc}(r1) = d3 \)

\[
g = \{ \text{loc}(r1) = d3, \text{loc}(c1) = r1 \}
\]

\[
s_0 = \{ \text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1 \}
\]
Why are Landmarks Useful?

• Breaks down a problem into smaller subproblems

• Suppose $m_1, m_2, m_3$ are landmarks
  • Every solution to $P$ must achieve $m_1, m_2, m_3$

• Possible strategy:
  • find a plan to go from $s_0$ to any state $s_1$ that satisfies $m_1$
  • find a plan to go from $s_1$ to any state $s_2$ that satisfies $m_2$
  • ...
Computing Landmarks

• Worst-case complexity:
  • Deciding whether $\varphi$ is a landmark is PSPACE-hard
  • As hard as solving the planning problem itself

• But there are often useful landmarks that can be found more easily
  • Polynomial time
  • Going to see one such procedure based on RPGs
    • Why RPGs?
      • Solving relaxed planning problems easier
        • Computing landmarks for relaxed planning problems easier
      • A landmark for a relaxed planning problem is a landmark for the original planning problem as well
RPG-based Landmark Computation

• Main intuition:
  • if $\varphi$ is a landmark, can get new landmarks from the preconditions of the actions that achieve $\varphi$

• Example:
  • goal $g$
  • $\{a_1, a_2\} =$ all actions that achieve $g$
  • $pre(a_1) = \{p_1, q\}$
  • $pre(a_2) = \{q, p_2\}$
  • To achieve $g$, must achieve $(p_1 \land q) \lor (p_2 \land q)$
    • same as $q \land (p_1 \lor p_2)$
  • Landmarks:
    • $q$
    • $p_1 \lor p_2$
RPG-based Landmark Computation

- Suppose goal is \( g = \{g_1, g_2, \ldots, g_k\} \)
  - Trivially, every \( g_i \) is a landmark

- Suppose \( g_1 = (\text{loc}(r1) = d1) \)
  - Two actions can achieve \( g_1 \):
    - \( \text{move}(r1, d3, d1) \)
    - \( \text{move}(r1, d2, d1) \)
  - Preconditions
    - \( \text{loc}(r1) = d3 \)
    - \( \text{loc}(r1) = d2 \)

- New landmark: \( \phi' = (\text{loc}(r1) = d3 \lor \text{loc}(r1) = d2) \)

- \( \text{move}(r, l, m) \)
  - pre: \( \text{loc}(r) = l \)
  - eff: \( \text{loc}(r) \leftarrow m \)

- \( \text{take}(r, l, c) \)
  - pre: \( \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l \)
  - eff: \( \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r \)

- \( \text{put}(r, l, c) \)
  - pre: \( \text{loc}(r) = l, \text{loc}(c) = r \)
  - eff: \( \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l \)

\[ s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\} \]
RPG-based Landmark Computation

RPG-Landmarks \( (s_0, g = \{g_1, g_2, \ldots, g_k\}) \)

```plaintext
queue ← \{g_i ∈ g \mid s_0 doesn't satisfy g_i\};
Landmarks ← ∅
A ← all actions
while queue ≠ ∅ do
    remove a g_i from queue
    Landmarks ← Landmarks ∪ g_i
    R ← \{actions whose effects include g_i\}
    if \( s_0 \) satisfies pre(a) for some \( a ∈ R \) then
        return Landmarks
    generate RPG from \( s_0 \) and \( A \setminus R \), stop when \( \hat{s}_k = \hat{s}_{k-1} \)
    N ← \{a ∈ R \mid a \text{ r-applicable in } \hat{s}_k\}
    if \( N = ∅ \) then
        return failure
    Pre ← \( ∪ \{\text{pre}(a) \mid a ∈ N\}\) \setminus \( s_0 \)
    \( \Phi \) ← \{p_1 ∨ p_2 ∨ \ldots ∨ p_m \mid m ≤ 4, ∀ a ∈ N \exists i: p_i \in \text{pre}(a), ∀ i: p_i ∈ \text{Pre}\}
    for each \( \varphi ∈ \Phi \) do
        add \( \varphi \) to queue
    return Landmarks
```
RPG-based Landmark Computation

RPG-Landmarks($s_0$, $g = \{g_1, g_2, \ldots, g_k\}$)
queue ← \{$g_i \in g$ | $s_0$ doesn’t satisfy $g_i$\};
Landmarks ← \emptyset
A ← all actions
while queue ≠ \emptyset do
    remove a $g_i$ from queue
    Landmarks ← Landmarks $\cup$ $g_i$
    $R$ ← \{actions whose effects include $g_i$\}
    if $s_0$ satisfies $\text{pre}(a)$ for some $a \in R$ then
        return Landmarks
    generate RPG from $s_0$ and $A \setminus R$, stop when $\hat{s}_k = \hat{s}_{k-1}$
    $N$ ← \{a ∈ $R$ | a $r$-applicable in $\hat{s}_k$\}
    if $N = \emptyset$ then
        return failure
    $Pre$ ← $\bigcup\{\text{pre}(a) | a \in N\} \setminus s_0$
    $\Phi$ ← \{$p_1 \lor p_2 \lor \ldots \lor p_m | m \leq 4, \forall a \in N \exists i: p_i \in \text{pre}(a), \forall i: p_i \in Pre$\}
    for each $\varphi \in \Phi$ do
        add $\varphi$ to queue
return Landmarks
RPG-based Landmark Computation

RPG-Landmarks($s_0, g = \{g_1, g_2, \ldots, g_k\}$)
queen ← \{gi ∈ g | $s_0$ doesn’t satisfy $g_i\};
Landmarks ← ∅
A ← all actions
while queue ≠ ∅ do
  remove a $g_i$ from queue
  Landmarks ← Landmarks $\cup$ $g_i$
  $R$ ← \{actions whose effects include $g_i$\}
  if $s_0$ satisfies $pre(a)$ for some $a$ $\in$ $R$ then
    return Landmarks
  generate RPG from $s_0$ and $A \setminus R$, stop when $\hat{s}_k = \hat{s}_{k-1}$
  $N$ ← \{a $\in$ $R$ | a r-applicable in $\hat{s}_k$\}
  if $N$ = ∅ then
    return failure
  $Pre ← \cup\{pre(a) | a$ $\in$ $N\} \setminus s_0$
  $\Phi ← \{p_1 \lor p_2 \lor \ldots \lor p_m | m ≤ 4, \forall a ∈ N \exists i : p_i ∈ pre(a), \forall i : p_i ∈ Pre\}$
  for each $\varphi$ $∈$ $\Phi$ do
    add $\varphi$ to queue
  return Landmarks
RPG-based Landmark Computation

RPG-Landmarks($s_0$, $g = \{g_1, g_2, \ldots, g_k\}$)

queue ← \{gi ∈ g | $s_0$ doesn’t satisfy $g_i$$\};
Landmarks ← \emptyset
A ← all actions

while queue ≠ ∅ do
    remove a $g_i$ from queue
    Landmarks ← Landmarks U $g_i$
    R ← \{actions whose effects include $g_i$$\}
    if $s_0$ satisfies pre($a$) for some $a ∈ R$ then
        return Landmarks
    generate RPG from $s_0$ and $A \setminus R$, stop when $\hat{s}_k = \hat{s}_{k-1}$

N ← \{a ∈ R | a $r$-applicable in $\hat{s}_k$$\}

if N = ∅ then
    return failure

Pre ← $\bigcup$\{pre($a$) | $a ∈ N$$\}\setminus s_0$
$\Phi$ ← \{p_1\lor p_2\lor \ldots \lor p_m | m ≤ 4, $\forall a ∈ N$ $\exists i : p_i ∈ pre(a) , \forall i : p_i ∈ Pre$$\}

for each $\varphi ∈ \Phi$ do
    add $\varphi$ to queue

return Landmarks

“necessary” actions

$N$: the only ones that can be $r$-applied and achieve $g_i$
RPG-based Landmark Computation

**RPG-Landmarks**($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

- queue ← $\{g_i \in g \mid s_0$ doesn’t satisfy $g_i\}$;
- Landmarks ← $\emptyset$
- $A$ ← all actions
- while queue ≠ $\emptyset$ do
  - remove a $g_i$ from queue
  - Landmarks ← Landmarks $\cup$ $g_i$
  - $R$ ← $\{$actions whose effects include $g_i$}$
  - if $s_0$ satisfies $\text{pre}(a)$ for some $a \in R$ then
    - return Landmarks
  - generate RPG from $s_0$ and $A \setminus R$, stop when $\hat{s}_k=\hat{s}_{k-1}$
  - $N$ ← $\{a \in R \mid a$ r-applicable in $\hat{s}_k\}$
  - if $N = \emptyset$ then
    - return failure
  - $\text{Pre} ← \text{U}\{\text{pre}(a) \mid a \in N\} \setminus s_0$
  - $\Phi ← \{p_1 \lor p_2 \lor \ldots \lor p_m \mid m \leq 4, \forall a \in N \exists i:p_i \in \text{pre}(a), \forall i:p_i \in \text{Pre}\}$
  - for each $\varphi \in \Phi$ do
    - add $\varphi$ to queue
  - return Landmarks
Example

RPG-Landmarks\(^1\)(\(s_0, \ g = \{g_1, g_2, \ldots, g_k\}\))

\[
\begin{align*}
\text{queue} & \leftarrow \{g_i \in g \mid s_0 \text{ doesn’t satisfy } g_i\}; \\
\text{Landmarks} & \leftarrow \emptyset \\
A & \leftarrow \text{all actions} \\
\text{while } \text{queue} \neq \emptyset \text{ do} & \\
& \ldots
\end{align*}
\]

\(\text{queue} = \{\text{loc}(c1) = r1\}\)
\(\text{Landmarks} = \emptyset\)

\(g = \{\text{loc}(r1) = d3, \text{loc}(c1) = r1\}\)
\(s_0 = \{\text{loc}(r1) = d3, \text{cargo}(r1) = \text{nil}, \text{loc}(c1) = d1\}\)

- \textit{move}(r, l, m)
  - \textit{pre}: \text{loc}(r) = l
  - \textit{eff}: \text{loc}(r) \leftarrow m

- \textit{take}(r, l, c)
  - \textit{pre}: \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l
  - \textit{eff}: \text{cargo}(r) \leftarrow c,
    \quad \text{loc}(c) \leftarrow r

- \textit{put}(r, l, c)
  - \textit{pre}: \text{loc}(r) = l, \text{loc}(c) = r
  - \textit{eff}: \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l

\(^1\text{true in } s_0\)
Example

RPG-Landmarks($s_0$, $g = \{g_1, g_2, \ldots, g_k\}$)

... while queue ≠ ∅ do

remove a $g_i$ from queue

Landmarks ← Landmarks ∪ $g_i$

$R ← \{\text{actions whose effects include } g_i\}$

if $s_0$ satisfies pre($a$) for some $a ∈ R$ then

return Landmarks

queue = ∅

Landmarks = \{loc(c1) = r1\}

$R = \{\text{take}(r1,d1,c1),$
\text{take}(r1,d2,c1),$
\text{take}(r1,d3,c1)\}$

true in $s_0$

g = \{loc(r1) = d3, loc(c1) = r1\}

s_0 = \{loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1\}

- move($r, l, m$)
  - pre: loc($r$) = $l$
  - eff: loc($r$) ← $m$

- take($r, l, c$)
  - pre: cargo($r$) = nil,
    loc($r$) = $l$, loc($c$) = $l$
  - eff: cargo($r$) ← $c$,
    loc($c$) ← $r$

- put($r, l, c$)
  - pre: loc($r$) = $l$,
    loc($c$) = $r$
  - eff: cargo($r$) ← nil,
    loc($c$) ← $l$
Example

RPG-Landmarks($s_0,\ g = \{g_1,\ g_2,\ldots,\ g_k\}$)

...  
while queue ≠ ∅ do

...  
generate RPG from $s_0$ and $A \setminus R$, stop when $\hat{s}_k = \hat{s}_{k-1}$

$N \leftarrow \{a \in R \mid a$-applicable in $\hat{s}_k\}$

if $N = \emptyset$ then  

return failure

queue = ∅  
Landmarks = {$loc(c1) = r1$}

$R = \{\text{take}(r1,d1,c1),\ \text{take}(r1,d2,c1),\ \text{take}(r1,d3,c1)\}$

$N = \{\text{take}(r1,d1,c1)\}$

• $move(r,l,m)$
  - pre: $loc(r) = l$
  - eff: $loc(r) \leftarrow m$

• $\text{take}(r,l,c)$
  - pre: $cargo(r) = \text{nil}$, $loc(r) = l, loc(c) = l$
  - eff: $cargo(r) \leftarrow c$, $loc(c) \leftarrow r$

• $\text{put}(r,l,c)$
  - pre: $loc(r) = l$, $loc(c) = r$
  - eff: $cargo(r) \leftarrow \text{nil}$, $loc(c) \leftarrow l$

$\hat{s}_0$:  
$\begin{align*}
loc(c1) &= d1 \\
loc(r1) &= d3 \\
cargo(r1) &= \text{nil}
\end{align*}$

$A_1$:  
$\begin{align*}
move(r1,d3,d1) &- loc(r1) = d1 \\
move(r1,d3,d2) &- loc(r1) = d2
\end{align*}$

both $\hat{s}_1$ and $\hat{s}_2$:
$\begin{align*}
loc(c1) &= d1 \\
loc(r1) &= d3 \\
cargo(r1) &= \text{nil}
\end{align*}$

From $\hat{s}_0$:
$\begin{align*}
loc(c1) &= d1 \\
loc(r1) &= d3 \\
cargo(r1) &= \text{nil}
\end{align*}$

RPG using $A \setminus R$
Example

RPG-Landmarks($s_0, g = \{g_1, g_2, \ldots, g_k\}$)

\[
\text{while} \ queue \neq \emptyset \ do \\
\quad \quad Pre \leftarrow \bigcup \{ \text{pre}(a) \mid a \in N \} \setminus s_0 \\
\quad \phi \leftarrow \{p_1 \lor p_2 \lor \cdots \lor p_m \mid m \leq 4, \forall a \in N \exists i: p_i \in \text{pre}(a), \forall i: p_i \in Pre \} \\
\quad \text{for each } \phi \in \phi \ do \\
\quad \quad \text{add } \phi \text{ to queue}
\]

\[
\text{queue} = \{\text{loc}(c1) = d1\} \\
\text{Landmarks} = \{\text{loc}(c1) = r1\} \\
R = \{\text{take}(r1, d1, c1)\}, \\
\quad \text{take}(r1, d2, c1), \\
\quad \text{take}(r1, d3, c1)\} \\
\text{N} = \{\text{take}(r1, d1, c1)\}
\]

- \textit{move}(r, l, m)
  - \textit{pre}: \text{loc}(r) = l
  - \textit{eff}: \text{loc}(r) \leftarrow m

- \textit{take}(r, l, c)
  - \textit{pre}: \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l
  - \textit{eff}: \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r

- \textit{put}(r, l, c)
  - \textit{pre}: \text{loc}(r) = l, \text{loc}(c) = r
  - \textit{eff}: \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l

\text{take}(r1, d1, c1)
\quad \text{pre}: \text{cargo}(r1) = \text{nil}, \text{loc}(r1) = d1, \text{loc}(c1) = d1

\text{satisfied in } \hat{s}_0

\text{add to queue}
Landmark Heuristic

• Every solution to the problem needs to achieve all the computed landmarks

• One possible heuristic:
  • $h^{sl}(s) =$ number of landmarks returned by RPG-Landmarks
  • Is this heuristic admissible?
    • No

\[ g = \{g_1, g_2\} \]
Two landmarks: $g_1, g_2$
Optimal plan: $\langle a_1 \rangle$, length = 1

• There are other more-advanced landmark heuristics
  • Some of them are admissible
  • Check textbook for references
Intermediate Summary

• Heuristic functions
  • Straight-line distance example
  • Delete-relaxation heuristics
    • relaxed states, $\gamma^+$, $h^+$, HFF, $h^{FF}$
  • Disjunctive landmarks, RPG-Landmark, $h^{sl}$
    • Get necessary actions by making RPG for all non-relevant actions
Outline per the Book

2.1 State-variable representation
   • State = \{values of variables\}; action = changes to those values

2.2 Forward state-space search
   • Start at initial state, look for sequence of actions that achieve goal

2.3 Heuristic functions
   • How to guide a forward state-space search

2.6 Incorporating planning into an actor
   • Online lookahead, unexpected events

2.4 Backward search
   • Start at goal state, go backwards toward initial state

2.5 Plan-space search
   • Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan
Incorporating Planning into an Actor

- Plans are abstract
  - Need additional refinement
  - (Chapter 3)

*The best laid schemes o’ mice an’ men,*

*Gang aft agley.*

–Robert Burns

- Plans don’t always work
  - What to do about it?
• $s_0 = \{\text{loc}(r1) = \text{loc}3, \text{loc}(o7) = \text{loc}1, \text{cargo}(r1) = \text{nil}\}$
• $g = \{\text{loc}(o7) = \text{loc}2\}$
• $\pi = \langle a1, a2, a3, a4, a5 \rangle$
  • $a1 = \text{go}(r1, \text{loc}3, \text{hall})$
  • $a2 = \text{navigate}(r1, \text{hall}, \text{loc}1)$
  • $a3 = \text{take}(r1, \text{loc}1, o7)$
  • $a4 = \text{navigate}(r1, \text{loc}1, \text{loc}2)$
  • $a5 = \text{put}(r1, \text{loc}2, o7)$
• $\text{go}(r, l, m)$
  • pre: $\text{adj}(l, m), \text{loc}(r) = l$
  • eff: $\text{loc}(r) \leftarrow m$
• $\text{navigate}(r, l, m)$
  • pre: $\neg \text{adj}(l, m), \text{loc}(r) = l$
  • eff: $\text{loc}(r) \leftarrow m$
• $\text{take}(r, l, o)$
  • pre: $\text{loc}(r) = l, \text{loc}(o) = l, \text{cargo}(r) = \text{nil}$
  • eff: $\text{loc}(o) \leftarrow r, \text{cargo}(r) \leftarrow o$

What are possible issues?

- respond to user requests
- bring o7 to loc2
- go to hallway
- navigate to loc1
- fetch o7
- navigate to loc2
- deliver o7
- move to door
- open door
- get out
- close door
- identify type of door
- move close to knob
- grasp knob
- turn knob
- maintain
- pull monitor
- move back
- pull
- monitor
- ungrasp
Using Planning in Acting

- Lookahead is the planner
- Receding horizon:
  - Call Lookahead, obtain $\pi$, perform 1st action, call Lookahead again ...
  - Like game-tree search (chess, checkers, etc.)
- Useful when unpredictable things are likely to happen
  - Re-plans immediately
- Potential problem:
  - May pause repeatedly while waiting for Lookahead to return
  - What if $\xi$ changes during the wait?

```
Run-Lookahead($\Sigma, g$)
while $s \leftarrow$ abstraction of observed state $\xi \neq g$ do
    $\pi \leftarrow$ Lookahead($\Sigma, s, g$)
    if $\pi = \text{failure}$ then
        return failure
    $a \leftarrow$ pop-first-action($\pi$)
    perform $a$
```

Planning stage
Acting stage
Using Planning in Acting

• Call Lookahead, execute the plan as far as possible, don’t call Lookahead again unless necessary

• Simulate tests whether the plan will execute correctly
  • Could just compute $\gamma(s, \pi)$, or could do something more detailed
    • Lower-level refinement, physics-based simulation

• Potential problems
  • May might miss opportunities to replace $\pi$ with a better plan
  • Without Simulate, may not detect problems until it is too late

Run-Lazy-Lookahead($\Sigma, g$)

\[
s \leftarrow \text{abstraction of observed state } \xi
\]

while $s \not\equiv g$

\[
\pi \leftarrow \text{Lookahead}(\Sigma, s, g)
\]

if $\pi = \text{failure}$ then

return failure

while $\pi \not= \langle \rangle$ and $s \not= g$ and \n
\[
\text{Simulate}(\Sigma, s, g, \pi)
\]

\not= \text{failure}

\[
a \leftarrow \text{pop-first-action}(\pi)
\]

perform $a$

\[
s \leftarrow \text{abstraction of observed state } \xi
\]
Using Planning in Acting

• May detect opportunities earlier than Run-Lazy-Lookahead
  • But may miss some that Run-Lazy-Lookahead would find
• Without Simulate, may fail to detect problems until it is too late
  • Not as bad at this as Run-Lazy-Lookahead
  • Possible work-around: restart Lookahead each time \( s \) changes

---

\[
\text{Run-Concurrent-Lookahead}(\Sigma, g) \\
\quad \pi \leftarrow \langle \rangle \\
\quad s \leftarrow \text{abstraction of observed state } \xi \\
\quad // \text{ thread 1 + 2 run concurrently} \\
\text{thread 1:} \\
\quad \text{loop} \\
\quad \quad \pi \leftarrow \text{Lookahead}(\Sigma, s, g) \\
\text{thread 2:} \\
\quad \text{loop} \\
\quad \quad \text{if } s \models g \text{ then} \\
\quad \quad \quad \text{return success} \\
\quad \quad \text{else if } \pi = \text{failure} \text{ then} \\
\quad \quad \quad \text{return failure} \\
\quad \quad \text{else if } \pi \neq \langle \rangle \text{ and } s \not\models g \text{ and} \\
\quad \quad \quad \quad \text{Simulate}(\Sigma, s, g, \pi) \\
\quad \quad \quad \quad \neq \text{failure} \text{ then} \\
\quad \quad \quad \quad \quad a \leftarrow \text{pop-first-action}(\pi) \\
\quad \quad \quad \quad \text{perform } a \\
\quad \quad \quad s \leftarrow \text{abstraction of observed state } \xi
\]
How to do Lookahead

- **Subgoaling**
  - Instead of planning for $g$, plan for a subgoal $g'$
  - Once $g'$ is achieved, plan for next subgoal

- **Receding horizon**
  - Return a plan that goes just part-way to $g'$
  - E.g., cut off search at
    - every plan whose cost exceeds some value $c_{max}$
    - or whose length exceeds some value $l_{max}$
    - or when no time is left
  - Horizon recedes on the actor’s successive calls to the planner

- **Sampling**
  - Try a few (e.g., randomly chosen) depth-first rollouts, take the one that looks best

- Can use combinations of these
Receding-Horizon Search

- After line (i), put something like these:
  - **cost-based cutoff:**
    
    \[
    \text{if } \text{cost}(\pi) + h(s) > c_{\text{max}} \text{ then}
    \]
    
    return $\pi$
  
  - **length-based cutoff:**
    
    \[
    \text{if } |\pi| > l_{\text{max}} \text{ then}
    \]
    
    return $\pi$
  
  - **time-based cutoff:**
    
    \[
    \text{if } \text{time-left}() = 0 \text{ then}
    \]
    
    return $\pi$

---

```latex
\begin{align*}
\text{Deterministic-Search}(\Sigma, s_0, g) \\
\text{Frontier} & \leftarrow \{\langle \emptyset, s_0 \rangle \} \\
\text{Expanded} & \leftarrow \emptyset \\
\text{while } \text{Frontier} \neq \emptyset \text{ do} \\
\quad \text{select a node } v = (\pi, s) \in \text{Frontier} \text{ (i)} \\
\quad \text{remove } v \text{ from } \text{Frontier} \\
\quad \text{add } v \text{ to } \text{Expanded} \\
\quad \text{if } s \text{ satisfies } g \text{ then} \\
\quad \quad \text{return } \pi \\
\quad \text{Children} & \leftarrow \{ (\pi.a, s) | s \text{ satisfies } \text{pre}(a) \} \\
\quad \text{prune 0 or more nodes from } \text{Frontier, Expanded} \text{ (ii)} \\
\quad \text{Frontier} & \leftarrow \text{Frontier} \cup \text{Children} \\
\text{return } \text{failure}
\end{align*}
```
Partial or Non-optimal Plans

- **Sampling**
  - Planner is a modified version of greedy algorithm
  - Make randomized choice at (i)
  - Run several times, get several solutions
  - Return best one

- **Actor calls the planner repeatedly as it acts**
  - An analogous technique is used in the game of Go

---

**Greedy**($\Sigma, s_0, g, \text{Visited}$)

1. if $s$ satisfies $g$ then
   - return $\pi$
2. $Act \leftarrow \{a \in A | s \text{ satisfies } pre(a) \text{ and } \gamma(s,a) \notin \text{Visited}\}$
3. if $Act = \emptyset$ then
   - return failure
4. $a \leftarrow \text{argmin}_{a \in Act} h(\gamma(s,a))$ (i)
5. $\pi \leftarrow \text{Greedy}(\Sigma, \gamma(s,a), g, \text{Visited} \cup \{s\})$
6. if $\pi \neq \text{failure}$ then
   - return $a.\pi$
7. return failure
Intermediate Summary

• Incorporating Planning into an actor
  • Things that can go wrong while acting
  • Algorithms
    • Run-Lookahead
    • Run-Lazy-Lookahead
    • Run-Concurrent-Lookahead

• Lookahead
  • Subgoaling
  • Receding-horizon search
  • Sampling
2.1 *State-variable representation*  
  • State = \{values of variables\}; action = changes to those values

2.2 *Forward state-space search*  
  • Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*  
  • How to guide a forward state-space search

2.6 *Incorporating planning into an actor*  
  • Online lookahead, unexpected events

2.4 *Backward search*  
  • Start at goal state, go backwards toward initial state

2.5 *Plan-space search*  
  • Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan
Backward Search

- Forward search starts at the initial state
  - Choose applicable action
  - Compute state transition $s' = \gamma(s, a)$
- Backward search starts at the goal
  - Chooses relevant action
    - A possible “last action” before the goal
  - Computes inverse state transition $g' = \gamma^{-1}(g, a)$
    - $g'$ = properties a state $s'$ should satisfy in order for $\gamma(s', a)$ to satisfy $g$
- Sometimes has a lower branching factor
  - Forward: 7 applicable actions
    - five load actions, two move actions
  - Backward: $g = \{loc(r1) = d3\}$
    - two relevant actions: $move(r1, d1, d3), move(r2, d1, d3)$
Relevance

• Idea: when can \( a \) be useful as the last action of a plan \( \pi \) for achieving \( g \)?
  • \( a \) can make at least one atom in \( g \) true that wasn’t true already
  • \( a \) doesn’t make any part of \( g \) false

• \( a \) is **relevant** for \( g = \{ x_1 = c_1, \ldots, x_k = c_k \} \) if
  • at least one atom in \( g \) is also in \( \text{eff} (a) \)
    • i.e., \( g \) contains \( x = c \) and \( \text{eff} (a) \) contains \( x \leftarrow c \)
  • for every atom \( x = c \) in \( g \)
    • \( a \) doesn’t make \( x = c \) false
      • i.e., \( \text{eff} (a) \) doesn’t contain \( x \leftarrow c' \) for some \( c' \neq c \)
    • if \( \text{pre} (a) \) requires \( x = c \) to be false, then \( \text{eff} (a) \) makes it true
      • i.e., if \( \text{pre} (a) \) contains \( x \neq c \) or \( x = c' \), then \( \text{eff} (a) \) contains \( x \leftarrow c \)
Relevance

- \( \text{adj} = \{(d1, d2), (d1, d3), (d2, d1), (d2, d3), (d3, d1), (d3, d2)\} \)
- \( s = \{\text{loc}(c1) = d1, \text{loc}(c2) = d1, \text{loc}(c3) = d1, \text{loc}(r1) = d2, \text{cargo}(r1) = \text{nil}, \text{loc}(r2) = d2, \text{cargo}(r2) = \text{nil}\} \)
- \( g = \{\text{loc}(c1) = r1, \text{loc}(r1) = d3\} \)
- For each action below, is it relevant for \( g \)?
  - \( \text{take}(r1, d1, c1) \)
  - \( \text{take}(r1, d2, c1) \)
  - \( \text{put}(r2, d3, c1) \)
  - \( \text{move}(r1, d1, d3) \)
  - \( \text{move}(r1, d3, d1) \)
  - \( \text{move}(r1, d2, d3) \)

- \( \text{move}(r, l, m) \)
  - pre: \( \text{loc}(r) = l, \text{adj}(l, m) \)
  - eff: \( \text{loc}(r) \leftarrow m \)
- \( \text{take}(r, l, c) \)
  - pre: \( \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l \)
  - eff: \( \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r \)
- \( \text{put}(r, l, c) \)
  - pre: \( \text{loc}(r) = l, \text{loc}(c) = r \)
  - eff: \( \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l \)

- Ranges
  - \( \mathcal{R}(r) = \text{Robots} = \{r1, r2\} \)
  - \( \mathcal{R}(l) = \mathcal{R}(m) = \text{Locs} = \{d1, d2, d3\} \)
  - \( \mathcal{R}(c) = \text{Containers} = \{c1, c2, c3\} \)
Inverse State Transitions

- If \( a \) is relevant for \( g \), then
  \[
  \gamma^{-1}(g, a) = \text{pre}(a) \cup (g - \text{eff}(a))
  \]
- If \( a \) isn’t relevant for \( g \), then \( \gamma^{-1}(g, a) \) is undefined
- Example:
  - \( g = \{\text{loc}(c1) = r1\} \)
  - What is \( \gamma^{-1}(g, \text{take}(r1, d3, c1)) \)?
  - What is \( \gamma^{-1}(g, \text{take}(r2, d1, c1)) \)?

- \textit{move}(r, l, m)
  - \text{pre}: \text{loc}(r) = l, \text{adj}(l, m)
  - \text{eff}: \text{loc}(r) \leftarrow m

- \textit{take}(r, l, c)
  - \text{pre}: \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(c) = l
  - \text{eff}: \text{cargo}(r) \leftarrow c, \text{loc}(c) \leftarrow r

- \textit{put}(r, l, c)
  - \text{pre}: \text{loc}(r) = l, \text{loc}(c) = r
  - \text{eff}: \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(c) \leftarrow l
Backward Search

• Cycle checking:
  • After line (i), put \( \text{Solved} \leftarrow \{g\} \)
  • After line (ii), put
    • either this:
      if \( g \in \text{Solved} \) then
        return failure
      \( \text{Solved} \leftarrow \text{Solved} \cup \{g\} \)
    • or this:
      if \( \exists g' \in \text{Solved} \text{ s.t. } g \subseteq g' \) then
        return failure
      \( \text{Solved} \leftarrow \text{Solved} \cup \{g\} \)

• With cycle checking, sound and complete
  • If \((\Sigma, s_0, g_0)\) is solvable, then at least one of the execution traces will find a solution

Backward-search(\(\Sigma, s_0, g_0\))

\[
\begin{align*}
g &\leftarrow g_0 \\
\pi &\leftarrow \langle \rangle \\
\text{loop} &\quad \text{(i)} \\
\quad &\quad \text{if } s_0 \text{ satisfies } g \text{ then} \\
\quad &\quad \quad \text{return } \pi \\
\quad &\quad A' \leftarrow \{a \in A \mid a \text{ is relevant for } g\} \\
\quad &\quad \text{if } A' = \emptyset \text{ then} \\
\quad &\quad \quad \text{return failure} \\
\quad &\quad \text{nondeterministically choose } a \in A' \\
\quad &\quad g \leftarrow \gamma^{-1}(g, a) \\
\quad &\quad \pi \leftarrow a.\pi \\
\text{end loop} &\quad \text{(ii)}
\end{align*}
\]
Motivation for Backward-search was to reduce the branching factor

As written, doesn’t accomplish that

Solve this by lifting:

When possible, leave variables uninstantiated

\[
\begin{align*}
\text{move}(r1, d1, d3) & \quad \gamma^{-1} \quad g = \{\text{loc}(r1) = d3\} \\
\text{move}(r1, d2, d3) & \\
\text{move}(r1, d4, d3) & \\
\vdots & \\
\text{move}(r1, d7, d3) & \quad \gamma^{-1} \quad g = \{\text{loc}(r1) = d3\}
\end{align*}
\]
Lifted Backward Search

- Like Backward-search but much smaller branching factor
  - Must keep track of what values were substituted for which parameters
  - I won’t discuss the details
  - Plan-space planning (later) does something similar

Backward-search($\Sigma, s_0, g_0$)

```latex
\begin{align*}
g &\leftarrow g_0 \\
\pi &\leftarrow \langle \rangle \\
\text{loop} & \\
\text{if } s_0 \text{ satisfies } g &\text{ then} \\
&\quad \text{return } \pi \\
A' &\leftarrow \{ a \in A \mid a \text{ is relevant for } g \} \\
\text{if } A' = \emptyset &\text{ then} \\
&\quad \text{return } \text{failure} \\
&\quad \text{nondeterministically choose } a \in A' \\
g &\leftarrow \gamma^{-1}(g, a) \\
\pi &\leftarrow a \cdot \pi
\end{align*}
```

Lifted-Backward-search($\mathcal{A}, s_0, g$)

```latex
\begin{align*}
\pi &\leftarrow \langle \rangle \\
\text{loop} & \\
\text{if } s_0 \text{ satisfies } g &\text{ then} \\
&\quad \text{return } \pi \\
A &\leftarrow \{(a, \theta) \mid a \text{ is a standardisation of an action template in } \mathcal{A}, \\
&\quad \theta \text{ is an mgu for an atom of } g \text{ and an atom of } \text{eff}^+(a), \text{ and} \\
&\quad \gamma^{-1}(\theta(g), \theta(a)) \text{ is defined}\} \\
\text{if } A = \emptyset &\text{ then} \\
&\quad \text{return } \text{failure} \\
&\quad \text{nondeterministically choose } (a, \theta) \in A \\
g &\leftarrow \gamma^{-1}(\theta(g), \theta(a))
\end{align*}
```
Intermediate Summary

• Backward State-space Search
  • Relevance, inverse state transition $\gamma^{-1}$
  • Backward search, cycle checking
  • Lifted backward search (briefly)
2.1 State-variable representation
   • State = {values of variables}; action = changes to those values

2.2 Forward state-space search
   • Start at initial state, look for sequence of actions that achieve goal

2.3 Heuristic functions
   • How to guide a forward state-space search

2.6 Incorporating planning into an actor
   • Online lookahead, unexpected events

2.4 Backward search
   • Start at goal state, go backwards toward initial state

2.5 Plan-space search
   • Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan
Plan-Space Search

• Formulate planning as a constraint satisfaction problem
  • Use constraint-satisfaction techniques to produce solutions that are more flexible than ordinary plans
    • E.g., plans in which the actions are partially ordered
    • Postpone ordering decisions until the plan is being executed
      • the actor may have a better idea about which ordering is best

• First step toward temporal planning (Chapter 4 in book)

• Basic idea:
  • Backward search from the goal
  • Each node of the search space is a partial plan that contains flaws
    • Remove the flaws by making refinements
  • If successful, we will get a partially ordered solution
Definitions

• Partially ordered plan
  • partially ordered set of nodes
  • each node contains an action

• Partially ordered solution
  • partially ordered plan $\pi$ such that every total ordering of $\pi$ is a solution

• Partial plan
  • partially ordered set of nodes that contain partially instantiated actions
  • inequality constraints
    • e.g. $z \neq x$ or $w \neq p1$
  • causal links (dashed arcs)
    • use action $a$ to establish precondition $p$ of action $b$

```
foo(y) → pre: loc(y) = p1
<table>
<thead>
<tr>
<th></th>
<th>pre: loc(z) = p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar(y)</td>
<td>baz(z)</td>
</tr>
</tbody>
</table>

eff: loc(y) = p1
z ≠ x
```

```
move(d, a, p1) move(c, b, p4)
move(a, p3, d) move(b, p4, c)
```

```
foo(y) → bar(y) → baz(z)

p1
p2

a
b
c
d

p3
p4

ϕ

g

s0
```
Flaws: 1. Open Goals

- A precondition $p$ of an action $b$ is an **open goal** if there is no causal link for $p$

- Resolve the flaw by creating a causal link
  - Find an action $a$ (either already in $\pi$, or can add it to $\pi$) that can establish $p$
    - can precede $b$
    - can have $p$ as an effect
  - Do substitutions on variables to make $a$ assert $p$
    - e.g., replace $x$ with $y$
  - Add an ordering constraint $a < b$
  - Create a causal link from $a$ to $p$

\[
\begin{align*}
\text{pre: } & \text{loc}(y) = p1 \\
\text{foo}(x) & \quad \text{bar}(y) \\
\text{eff: } & \text{loc}(x) = p1
\end{align*}
\]

\[
\begin{align*}
\text{foo}(y) & \quad \text{bar}(y) \\
\text{pre: } & \text{loc}(y) = p1 \\
\text{eff: } & \text{loc}(y) = p1 \\
\text{replace } x \text{ with } y
\end{align*}
\]
Flaws: 2. Threats

• Suppose we have a causal link from action $a$ to precondition $p$ of action $b$

• Action $c$ threatens the link if $c$ may affect $p$ and may come between $a$ and $b$
  • $c$ is a threat even if it makes $p$ true rather than false
    • Causal link means $a$, not $c$, is supposed to establish $p$ for $b$
    • The plan in which $c$ establishes $p$ will be generated on another path in the search space

• Three possible ways to resolve the flaw:
  • Make $c < a$
  • Make $b < c$
  • Add inequality constraints to prevent $c$ from affecting $p$

\[\text{eff: } \text{loc}(z) = p2 \quad \text{pre: } \text{loc}(y) = p1\]

\[\text{clobber}(z)\]

\[\text{eff: } \text{loc}(y) = p1\]

\[\text{foo}(y) \quad \text{bar}(y)\]
PSP Algorithm

• Initial plan is always \{Start, Finish\} with \textit{Start} < \textit{Finish}
  
  • Start
    • No preconditions
    • Effects: atoms in \( s_0 \)
  
  • Finish
    • Preconditions: atoms in \( g \)
    • No effects

• PSP is sound and complete
  
  • Returns a partially ordered plan \( \pi \) s.t. any total ordering of \( \pi \) will achieve \( g \)
  
  • In some environments, could execute actions in parallel

\[
\text{PSP}(\Sigma, \pi)
\]

\[
\text{loop}
\]

\[
\quad \text{if Flaws}(\pi) = \emptyset \text{ then}
\]

\[
\quad \text{return } \pi
\]

\[
\quad \text{arbitrarily select } f \in \text{Flaws}(\pi)
\]

\[
\quad R \leftarrow \{ \text{all feasible resolvers for } f \}
\]

\[
\quad \text{if } R = \emptyset \text{ then}
\]

\[
\quad \text{return } \text{failure}
\]

\[
\quad \text{nondeterministically choose } \rho \in R
\]

\[
\quad \pi \leftarrow \rho(\pi)
\]

\[
\quad \text{return } \pi
\]
Example

- Finish has two open goals: pos(a)=d, pos(b)=c

```plaintext
move(c, y, z)
pre: pos(c)=y, clear(c)=T, clear(z)=T
eff: pos(c)←z, clear(y)←T, clear(z)←F pos(a)=d, pos(b)=c

ℛ(c) = Containers
ℛ(y) = ℛ(z) = Container ∪ pallets
```

```plaintext
loop
if Flaws(π) = ∅ then
    return π
arbitrarily select f ∈ Flaws(π)
R ← {all feasible resolvers for f}
if R = ∅ then
    return failure
nondeterministically choose ρ ∈ R
π ← ρ(π)
return π
```

clear(p1)=T clear(p2)=T clear(p3)=F clear(p4)=F
clear(a)=F clear(b)=F clear(c)=T clear(d)=T
pos(a)=p3 pos(b)=p4 pos(c)=b pos(d)=a
Example

- For each open goal, add a new action
  - Every new action $a$ must have $\text{Start} < a < \text{Finish}$

$\mathcal{R}(c) = \text{Containers}$
$\mathcal{R}(y) = \mathcal{R}(z) = \text{Container} \cup \text{pallets}$
Example

- Resolve four open goals using the Start action
  - substitute $y_1=p3$, $y_2=p4$

\[
\begin{align*}
\text{move}(c, y, z) & \quad \text{pre: } \text{pos}(c)=y, \text{clear}(c)=T, \text{clear}(z)=T \\
& \quad \text{eff: } \text{pos}(c)\leftarrow z, \text{clear}(y)\leftarrow T, \text{clear}(z)\leftarrow F \quad \text{pos}(a)=d \\
\mathcal{R}(c) &= \text{Containers} \\
\mathcal{R}(y) &= \mathcal{R}(z) = \text{Container} \cup \text{pallets}
\end{align*}
\]
Example

- New action to resolve open goal
- 1\textsuperscript{st} threat has one resolver: $z_3 \neq d$
- 2\textsuperscript{nd} threat has two resolvers:
  - $\text{move}(b, p4, c) < \text{move}(x_3, a, z_3)$
  - $z_3 \neq c$

\begin{align*}
\text{clear}(z_3) &= T \\
\text{clear}(x_3) &= T \\
\text{pos}(x_3) &= a
\end{align*}

\begin{align*}
\text{move}(x_3, a, z_3) \\
\text{clear}(d) &= T \\
\text{clear}(a) &= T \\
\text{pos}(a) &= p3
\end{align*}

\begin{align*}
\text{move}(a, p3, d) \\
\text{pos}(a) &= p3 \\
\text{pos}(b) &= p4 \\
\text{pos}(c) &= b \\
\text{pos}(d) &= a
\end{align*}

\begin{align*}
\text{move}(c, y, z) \\
\text{pre: } &\text{pos}(c) = y, \text{clear}(c) = T, \text{clear}(z) = T \\
\text{eff: } &\text{pos}(c) \leftarrow z, \text{clear}(y) \leftarrow T, \text{clear}(z) \leftarrow F \\
\text{pos}(a) &= d
\end{align*}

$\mathcal{R}(c) = \text{Containers}$

$\mathcal{R}(y) = \mathcal{R}(z) = \text{Container} \cup \text{pallets}$
Example

- Threats resolved

\[
\begin{align*}
\text{clear}(z_3) &= T \\
\text{clear}(x_3) &= T \\
\text{pos}(x_3) &= a \\
\text{move}(x_3,a,z_3) \\
\end{align*}
\]

\[
\begin{align*}
\text{clear}(d) &= T \\
\text{clear}(a) &= T \\
\text{pos}(a) &= p3 \\
\text{move}(a,p3,d) \\
\end{align*}
\]

\[
\begin{align*}
\text{clear}(c) &= T \\
\text{clear}(x_3) &= T \\
\text{pos}(x_3) &= a \\
\text{move}(x_3,a,z_3) \\
\end{align*}
\]

\[
\begin{align*}
\text{clear}(b) &= T \\
\text{clear}(c) &= T \\
\text{move}(b,p4,c) \\
\end{align*}
\]

\[
\begin{align*}
\text{clear}(x_3) &= T \\
\text{clear}(z_3) &= T \\
\text{pos}(x_3) &= a \\
\text{move}(x_3,a,z_3) \\
\end{align*}
\]

\[
\begin{align*}
\text{clear}(p1) &= T \\
\text{clear}(p2) &= T \\
\text{clear}(p3) &= F \\
\text{clear}(p4) &= F \\
\text{pos}(a) &= p3 \\
\text{pos}(b) &= p4 \\
\text{pos}(c) &= b \\
\text{pos}(d) &= a \\
\end{align*}
\]

\[
\begin{align*}
\text{move}(c, y, z) \\
\text{pre: pos}(c) = y, \text{clear}(c) = T, \text{clear}(z) = T \\
\text{eff: pos}(c) \leftarrow z, \text{clear}(y) \leftarrow T, \text{clear}(z) \leftarrow F \\
\text{pos}(a) = d \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{R}(c) &= \text{Containers} \\
\mathcal{R}(y) &= \mathcal{R}(z) = \text{Container} \cup \text{pallets} \\
\end{align*}
\]
Example

- 1st threat has two resolvers:
  - An ordering constraint, and $z_4 \neq d$
- 2nd threat has three resolvers:
  - Two ordering constraints, and $z_4 \neq a$
- 3rd threat has one: $z_4 \neq c$

clear(z_3) = T  clear(x_3) = T  pos(x_3) = a

move(x_3, a, z_3)

pos(x_4) = b  clear(x_4) = T  clear(z_4) = T

move(x_4, b, z_4)

clear(d) = T  clear(a) = T  pos(a) = p3

move(a, p3, d)

pos(b) = p4  clear(b) = T  clear(c) = T

move(b, p4, c)

move(c, y, z)
pre: pos(c) = y, clear(c) = T, clear(z) = T
eff: pos(c) \leftarrow z, clear(y) \leftarrow T, clear(z) \leftarrow F  pos(a) = d

pos(b) = c

$\mathcal{R}(c) = \text{Containers}$

$\mathcal{R}(y) = \mathcal{R}(z) = \text{Container } \cup \text{ pallets}$
Example

- Resolve the three threats using inequality constraints

\[
\begin{align*}
\text{clear}(z_3) &= T \\
\text{clear}(x_3) &= T \\
\text{pos}(x_3) &= a \\
\text{move}(x_3, a, z_3) &\rightarrow \text{pos}(a) = p3 \\
\text{move}(a, p3, d) &\rightarrow \text{pos}(b) = c \\
\text{move}(b, p4, c) &\rightarrow \text{pos}(d) = b \\
\text{pos}(x_4) &= b \\
\text{clear}(x_4) &= T \\
\text{clear}(z_4) &= T \\
\text{move}(x_4, b, z_4) &\rightarrow \text{pos}(b) = p4 \\
\text{move}(b, p4, c) &\rightarrow \text{pos}(c) = T \\
\text{clear}(c) &= T \\
\text{move}(c, y, z) &\rightarrow \text{pos}(a) = d \\
\text{pos}(a) &= d \\
\text{clear}(d) &= T \\
\text{clear}(a) &= T \\
\text{move}(a, p3, d) &\rightarrow \text{pos}(b) = c \\
\text{move}(b, p4, c) &\rightarrow \text{pos}(b) = c \\
\end{align*}
\]

\(\mathcal{R}(c) = \text{Containers}\)

\(\mathcal{R}(y) = \mathcal{R}(z) = \text{Container} \cup \text{pallets}\)
Example

• Resolve five open goals using the Start action
  • substitute
    \( x_3 = d, \ x_4 = c, \ z_3 = p1 \)

\( \text{move}(a, p_3, d) \)
\( \text{clear}(d) = T \)
\( \text{clear}(c) = T \)
\( \text{clear}(z_4) = T \)
\( \text{pos}(a) = p_3 \)
\( \text{pos}(b) = c \)
\( \text{pos}(d) = a \)

\( \text{move}(c, b, z_4) \)
\( \text{pos}(c) = b \)
\( \text{clear}(c) = T \)
\( \text{clear}(z_4) = T \)
\( \text{pos}(b) = p_4 \)
\( \text{pos}(d) = a \)
\( \text{pos}(a) = p_3 \)
\( \text{clear}(d) = T \)
\( \text{clear}(a) = T \)
\( \text{clear}(z_4) = T \)
\( \text{x_4} \)

\( \text{move}(a, p_3, d) \)
\( \text{clear}(d) = T \)
\( \text{clear}(a) = T \)
\( \text{pos}(a) = p_3 \)
\( \text{pos}(b) = c \)
\( \text{pos}(d) = a \)
\( \text{clear}(d) = T \)

\( \text{move}(d, a, p_1) \)
\( \text{clear}(p_1) = T \)
\( \text{clear}(d) = T \)
\( \text{pos}(d) = a \)
\( \text{pos}(a) = p_3 \)
\( \text{pos}(b) = c \)

\( \text{move}(c, b, z_4) \)
\( \text{pos}(c) = b \)
\( \text{clear}(c) = T \)
\( \text{clear}(z_4) = T \)
\( \text{pos}(b) = p_4 \)
\( \text{pos}(d) = a \)
\( \text{pos}(a) = p_3 \)
\( \text{clear}(d) = T \)
\( \text{clear}(a) = T \)
\( \text{clear}(z_4) = T \)

\( \text{move}(c, y, z) \)
\( \text{pre: pos}(c) = y, \ clear(c) = T, \ clear(z) = T \)
\( \text{eff: pos}(c) \leftarrow z, \ clear(y) \leftarrow T, \ clear(z) \leftarrow F \)
\( \text{pos}(a) = d \)
\( \text{pos}(b) = c \)

\( \mathcal{R}(c) = \text{Containers} \)
\( \mathcal{R}(y) = \mathcal{R}(z) = \text{Container} \cup \text{pallets} \)
Example

• Threatened causal link
• Resolvers:
  • $\text{move}(d, a, p1) < \text{move}(c, b, z_4)$
  • $z_4 \neq p1$

\[
\begin{align*}
\text{clear}(p1) &= T, \text{clear}(d) = T, \text{pos}(d) = a \\
\text{clear}(d) &= T, \text{clear}(a) = T, \text{pos}(a) = p3 \\
\text{clear}(p1) &= T, \text{clear}(c) = T, \text{clear}(z_4) = T \\
\text{pos}(c) &= b, \text{clear}(c) = T, \text{clear}(z_4) = T \\
\end{align*}
\]

$\text{move}(c, y, z)$

pre: $\text{pos}(c) = y, \text{clear}(c) = T, \text{clear}(z) = T$

eff: $\text{pos}(c) \leftarrow z, \text{clear}(y) \leftarrow T, \text{clear}(z) \leftarrow F$

$\text{pos}(a) = d, \text{pos}(b) = c$

$\mathcal{R}(c) = \text{Containers}$

$\mathcal{R}(y) = \mathcal{R}(z) = \text{Container} \cup \text{pallets}$
Example

• Threat resolved

\[ \text{move}(c, y, z) \]
pre: pos(c) = y, clear(c) = T, clear(z) = T

\[ \text{eff: pos}(c) \leftarrow z, \text{clear}(y) \leftarrow T, \text{clear}(z) \leftarrow F \]
\[ \text{pos}(a) = d \]
\[ \text{pos}(b) = c \]

\[ \mathcal{R}(c) = \text{Containers} \]
\[ \mathcal{R}(y) = \mathcal{R}(z) = \text{Container} \cup \text{pallets} \]
Example

- Resolve open goal using the Start action
  - substitute $z_4 = p2$
- No more flaws, so we’re done!

\[
\begin{align*}
\text{move}(c, y, z) & \quad \text{pre:} \ pos(c)=y, \ clear(c)=T, \ clear(z)=T \\
\text{eff:} \ & pos(c)\leftarrow z, \ clear(y)\leftarrow T, \ clear(z)\leftarrow F \ pos(a)=d \ pos(b)=c \\
\mathcal{R}(c) & = \text{Containers} \\
\mathcal{R}(y) & = \mathcal{R}(z) = \text{Container} \cup \text{pallets}
\end{align*}
\]
Example

- PSP returns this solution:

```
move(d,a,p1)
moves(a,p3,d)
moves(c,b,p2)
moves(b,p4,c)
```
Example 2

- Go back to the last threat, choose the other resolver:
  - $\text{move}(d,a,p1) < \text{move}(c,b,z_4)$
  - $z_4 \neq p1$

\[
\begin{align*}
\text{clear}(p1) &= T \\
\text{clear}(d) &= T \\
\text{pos}(d) &= a \\
\text{move}(d,a,p1) \\
\text{clear}(c) &= T \\
\text{move}(c,b,z_4) \\
\text{clear}(p1) &= T \\
\text{clear}(d) &= T \\
\text{pos}(d) &= a \\
\text{pos}(a) &= p3 \\
\text{move}(a,p3,d) \\
\text{clear}(z_4) &= T \\
\text{pos}(c) &= b \\
\text{clear}(c) &= T \\
\text{move}(b,p4,c) \\
\text{clear}(z_4) &= T \\
\text{pos}(b) &= c \\
\text{pos}(a) &= d \\
\text{move}(b,p4,c) \\
\text{Finish}
\end{align*}
\]

- $z_4 \neq a$
- $z_4 \neq c$
- $z_4 \neq d$
Example 2

- Threat resolved
Example 2

- Resolve open goal
  - substitute $z_4 = p2$
- No more flaws, so we’re done

- clear($d$) = $T$
- pos($a$) = $p3$
- clear($c$) = $T$
- pos($b$) = $p4$

- clear($p1$) = $T$
- pos($d$) = $a$
- clear($p2$) = $T$
- pos($c$) = $b$
- clear($b$) = $T$
- pos($d$) = $a$
- clear($p3$) = $T$
- pos($a$) = $p3$
- clear($d$) = $T$
- pos($a$) = $p3$
- clear($c$) = $T$
- pos($b$) = $p4$
- pos($a$) = $d$

- satisfied
Example 2

- Like previous solution, but has another ordering constraint

```
move(a,p3,d)  move(b,p4,c)
```

Start

```
move(d,a,p1)  move(c,b,p2)
```

move(a,p3,d)  move(b,p4,c)

Finish
Node-selection Heuristics

• Analogy to constraint-satisfaction problems (CSPs)
  • Resolving a flaw in PSP
    ≈ assigning a value to a variable in a CSP

• What flaw to work on next?
  • Fewest Alternatives First (FAF)
    • Choose a flaw having the fewest resolvers
      ≈ Minimum Remaining Values (MRV) heuristic for CSPs

• To resolve the flaw, which resolver to try first?
  • Least Constraining Resolver (LCR)
    • Choose a resolver that rules out the fewest resolvers for the other flaws
      ≈ Least Constraining Value (LCV) heuristic for CSPs
Example

- Fewest Alternatives First:
  - 1\textsuperscript{st} threat has two resolvers: an ordering constraint, and $z_4 \neq d$
  - 2\textsuperscript{nd} threat has three resolvers: 2 ordering constraints, and $z_4 \neq a$
  - 3\textsuperscript{rd} threat has one resolver: $z_4 \neq c$

- So resolve the 3\textsuperscript{rd} threat first
Node-selection Heuristics

• In PSP, introducing a new action introduces new flaws to resolve
  • The plan can get arbitrarily large; want it to be as small as possible
    • Not like CSPs, where the search tree always has a fixed depth
  • Avoid introducing new actions unless necessary

• To choose between actions $a$ and $b$, estimate distance from $s_0$ to $Pre(a)$ and $Pre(b)$
  • Can use the heuristic functions we discussed earlier
Discussion

• Problem: how to prune infinitely long paths in the search space?
  • Loop detection is based on recognizing states or goals we have seen before

\[ \ldots \rightarrow s \rightarrow s' \rightarrow s \]

• In a partially ordered plan, we do not know the states

• Can we prune if \( \pi \) contains the same action more than once?
  • \( \langle a_1, a_2, \ldots, a_1, \ldots \rangle \)
  • No. Sometimes we might need the same action several times in different states of the world
  • E.g., Towers of Hanoi problem
    • Do this action many times:
      • stack disk1 onto disk2
A Weak Pruning Technique

• Can prune all partial plans of $n$ or more actions, where $n = |S|$
  • Not very helpful

“...I’m not sure whether there’s a good pruning technique for plan-space planning.”

Dana Nau
Intermediate Summary

- Plan-space Search
  - Partially ordered plans and solutions
  - partial plans, causal links
  - flaws: open goals, threats, resolvers
  - PSP algorithm, long example, node-selection heuristics
Summary

2.1 *State-variable representation*  
- State = {values of variables}; action = changes to those values

2.2 *Forward state-space search*  
- Start at initial state, look for sequence of actions that achieve goal

2.3 *Heuristic functions*  
- How to guide a forward state-space search

2.6 *Incorporating planning into an actor*  
- Online lookahead, unexpected events

2.4 *Backward search*  
- Start at goal state, go backwards toward initial state

2.5 *Plan-space search*  
- Start with incomplete plan for getting from initial state to goal state, make transformations to fix flaws in the plan

⇒ Next: Planning and Acting with Refinement Methods