A0: Finite
A1: Fully observable
A2: Deterministic
A3: Static
A4: Attainment goals
A5: Sequential plans
A6: Implicit time
A7: Offline planning

Why is this more compact than an explicit transition system?
- In an explicit transition system, actions are represented as state-to-state transitions. Each action will be represented by an incidence matrix of size |S|x|S|.
- In the proposed model, actions are represented only in terms of state variables whose values they care about, and whose value they affect. (It exploits the structure of the problem!)
- Consider a state space of 1024 states. It can be represented by log_2(1024)=10 state variables. If an action needs variable v1 to be true and makes v7 to be false, it can be represented by just 2 bits (instead of a 1024x1024 matrix).
- Of course, if the action has a complicated mapping from states to states, in the worst case the action rep will be just as large.
- The assumption being made here is that the actions will have effects on a small number of state variables.
1. Classical Representation

- Start with a function-free first-order language
- Finitely many predicate symbols and constant symbols, but no function symbols

Example: the DWR domain
- Locations: l1, l2, ...
- Containers: c1, c2, ...
- Piles: p1, p2, ...
- Robot carts: r1, r2, ...
- Cranes: k1, k2, ...

Atom: predicate symbol and args
- Use these to represent both fixed and dynamic (“fluent”) relations
  - adjacent(l,l')
  - attached(p,l)
  - belong(k,l)
  - occupied(l)
  - in(p,c)
  - on(c,c')
  - top(c)
  - loaded(r)
  - unloaded(r)
  - holding(k,c)
  - empty(k)
  - in(c1,p3)
  - empty(k)

Ground expression: contains no variable symbols — e.g., in(c1,p3)
Unground expression: at least one variable symbol — e.g., in(c1,x)
Substitution: \( \theta = \{x_1 \leftarrow v_1, x_2 \leftarrow v_2, \ldots, x_n \leftarrow v_n\} \)
- Each \( x_i \) is a variable symbol; each \( v_i \) is a term

Instance of \( e \): result of applying a substitution \( \theta \) to \( e \)
- Replace variables of \( e \) simultaneously, not sequentially

States

- State: a set \( s \) of ground atoms
- The atoms represent the things that are true in one of \( \Sigma \)'s states
- Only finitely many ground atoms, so only finitely many possible states

Operators

- Operator: a triple \( \tau(\text{name}(o), \text{precond}(o), \text{effects}(o)) \)
  - \( \text{name}(o) \) is a syntactic expression of the form \( n(x_1, \ldots, x_k) \)
  - \( n \): operator symbol - must be unique for each operator
  - \( x_1, \ldots, x_k \): variable symbols (parameters)
    - must include every variable symbol in \( o \)
  - \( \text{precond}(o) \): preconditions
    - literals that must be true in order to use the operator
  - \( \text{effects}(o) \): effects
    - literals the operator will make true

Actions

- Action: ground instance (via substitution) of an operator

Notation

- Let \( a \) be an operator or action. Then
  - \( \text{precond}^+(a) = \{ \text{atoms that appear positively in } a\text{'s preconditions} \} \)
  - \( \text{precond}^-(a) = \{ \text{atoms that appear negatively in } a\text{'s preconditions} \} \)
  - \( \text{effects}^+(a) = \{ \text{atoms that appear positively in } a\text{'s effects} \} \)
  - \( \text{effects}^-(a) = \{ \text{atoms that appear negatively in } a\text{'s effects} \} \)

E.g.,

\( \text{take}(k,c,d,p) \)
- \( \text{precond} = \{ \text{belong}(k,l), \text{attached}(p,l), \text{empty}(k), \text{top}(c,p), \text{on}(c,d) \} \)
- \( \text{effects} = \{ \text{holding}(k,c), \text{empty}(k), \text{empty}(p), \text{top}(c,p), \text{on}(c,d), \text{top}(d,p) \} \)
Applicability

- An action \( a \) is applicable to a state \( s \) if \( s \) satisfies \( \text{precond}(a) \), i.e., if \( \text{precond}^+(a) \subseteq s \) and \( \text{precond}^-(a) \cap s = \emptyset \).
- Here are an action and a state that it's applicable to:

\[
\begin{align*}
\text{take(crane, loc1, c3, c1, p1)} & : \text{crane} \text{ crane at location loc1 takes c3 off c1 in pile p1} \\
\text{precond:} & \text{belong(crane, loc1), attached(p1,loc1), empty(crane), top(c3,p1), on(c3,c1)} \\
\text{effects:} & \text{holding(crane, c3), empty(crane), \neg\text{in(c3,p1)}, \neg\text{top(c3,p1)}, \neg\text{on(c3,c1)}, \text{top(c1,p1)}}
\end{align*}
\]

Result of Performing an Action

- If \( a \) is applicable to \( s \), the result of performing it is \( \gamma(s,a) = (s - \text{effects}(a)) \cup \text{effects}^+(a) \)
- Delete negative effects, and add positive ones

\[
\begin{align*}
\text{take(crane, loc1, c3, c1, p1)} & : \text{crane} \text{ crane at location loc1 takes c3 off c1 in pile p1} \\
\text{precond:} & \text{belong(crane, loc1), attached(p1,loc1), empty(crane), top(c3,p1), on(c3,c1)} \\
\text{effects:} & \text{holding(crane, c3), \neg\text{empty(crane)}, \neg\text{in(c3,p1)}, \neg\text{top(c3,p1)}, \neg\text{on(c3,c1)}, \text{top(c1,p1)}}
\end{align*}
\]

Planning Problems

Given a planning domain (language \( L \), operators \( O \))

- **Statement** of a planning problem: a triple \( P=(O,s_0,g) \)
  - \( O \) is the collection of operators
  - \( S_0 \) is a state (the initial state)
  - \( g \) is a set of literals (the goal formula)
- The actual planning problem: \( P=(S,A,g) \)
  - \( S \) and \( A \) are as above
  - \( S = (S,A,f) \) is a state-transition system
  - \( A = (all \ ground \ instances \ of \ operators \ in \ O) \)
  - \( f \) is state-transition function determined by the operators

Plans and Solutions

- **Plan**: any sequence of actions \( \sigma = \langle a_1, a_2, \ldots, a_n \rangle \) such that each \( a_i \) is a ground instance of an operator in \( O \)
- The plan is a **solution** for \( P=(O,s_0,g) \) if it is executable and achieves \( g \)
  - i.e., if there are states \( s_0, s_1, \ldots, s_n \) such that
    - \( \gamma(s_0,\sigma) = s_1 \)
    - \( \gamma(s_1,\sigma) = s_2 \)
    - \[ \vdots \]
    - \( \gamma(s_{n-1},\sigma) = s_n \)
    - \( s_n \) satisfies \( g \)

Example

- Let \( P_1 = (O, s_1, g_1) \), where
  - \( O \) is the set of operators given earlier

\[
\begin{align*}
\text{g_1} & : (\text{loaded(r1,c3)}, \text{at(r1,loc2)}) \\
\text{s_1} & : (\text{attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc3), at(1,loc2), occupied(loc2), unloaded(r1)})}
\end{align*}
\]
Here are three solutions for $P_1$:

\begin{align*}
\langle &\text{take} (\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2}), \\
&\text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{load} (\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2})\rangle \\
\langle &\text{take} (\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2}), \\
&\text{load} (\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2})\rangle \\
\langle &\text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{take} (\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2})\rangle
\end{align*}

Each of them produces the state shown here:

Example (cont.)

First is redundant: can remove actions and still have a solution

\begin{align*}
\langle &\text{take} (\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2}), \\
&\text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{load} (\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2})\rangle \\
\langle &\text{take} (\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{load} (\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2})\rangle \\
\langle &\text{move} (\text{r1}, \text{loc2}, \text{loc1}), \text{take} (\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{load} (\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move} (\text{r1}, \text{loc1}, \text{loc2})\rangle
\end{align*}

The 2nd and 3rd are irredundant and shortest

Example (cont.)

2. Set-Theoretic Representation

Like classical representation, but restricted to propositional logic

States:

- Instead of a collection of ground atoms ... 
  \{on(c1, pallet), on(c1, r1), on(c1, c2), ... at(r1, l1), at(r1, l2), ...\}

... use a collection of propositions (boolean variables):
  \{on-c1-pallet, on-c1-r1, on-c1-c2, ... at-r1-l1, at-r1-l2, ...\}

Instead of operators like this one.

\begin{align*}
\text{take} (\text{c1}, \text{c2}, \text{c3}, \text{c4}, \text{p1}) \\
\text{precond:} \text{belong(c1, f), attached(p1, l), empty(k), top(k, r), on(r, d)} \\
\text{effects:} \text{\textcolor{red}{holding(c2, c1), \textcolor{cyan}{empty(c1), \textcolor{green}{in(c3, p1), \textcolor{blue}{top(c2, r1)}}}}
\end{align*}

and rewrite ground atoms as propositions

Comparison

A set-theoretic representation is equivalent to a classical representation in which all of the atoms are ground

Problem: Exponential blowup

- If a classical operator contains $n$ atoms and each atom has arity $k$,
  then it corresponds to $c^n$ actions where $c = |\{\text{constant symbols}\}|$

3. State-Variable Representation

- Non-fluents (properties that don't change) are ground relations: e.g., adjacent(loc1, loc2)

- Fluents are functions:
  i.e., for properties that can change, assign values to state variables

- Classical and state-variable representations take similar amounts of space each can be translated into the other in low-order polynomial time
Example: The Blocks World

- Infinitely wide table, finite number of children’s blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- Want to move blocks from one configuration to another
  - e.g.,
    - initial state
    - goal

- Classical, set-theoretic, and state-variable formulations for the case of FIVE BLOCKS follow.

1. Example Classical Representation

- Constant symbols:
  - The blocks: a, b, c, d, e
- Predicates:
  - ontable(x) - block x is on the table
  - on(x,y) - block x is on block y
  - clear(x) - block x has nothing on it
  - holding(x) - the robot hand is holding block x
  - handempty - the robot hand isn’t holding anything

2. Example Set-Theoretic Representation

For five blocks, 36 propositions, 50 actions

E.g.,
- ontable-a - block a is on the table
- on-c-a - block c is on block a
- clear-c - block c has nothing on it
- clear-d - block d has nothing on it
- holding-d - the robot hand is holding block d
- handempty - the robot hand isn’t holding anything

... (31 more)

3. Example State-Variable Representation

- Constant symbols:
  - a, b, c, d, e of type block
  - 0, 1, table, nil of type other
- State variables:
  - pos(x) = y if block x is on block y
  - pos(x) = table if block x is on the table
  - pos(x) = nil if block x is being held
  - clear(x) = 1 if block x has nothing on it
  - clear(x) = 0 if block x has nothing or has a block on it
  - holding = x if the robot hand is holding block x
  - holding = nil if the robot hand is holding nothing

... (46 more)
State-Variable Operators

unstack(x : block, y : block)
Precond: pos(x)=y, clear(x)=0, clear(y)=1, holding=nil
Effects: pos(x)=nil, clear(x)=0, clear(y)=1, holding=x

stack(x : block, y : block)
Precond: holding=x, clear(x)=0, clear(y)=1
Effects: holding=nil, clear(y)=0, pos(x)=y, clear(x)=1

pickup(x : block)
Precond: pos(x)=table, clear(x)=1, holding=nil
Effects: pos(x)=nil, clear(x)=0, holding=x

putdown(x : block)
Precond: holding=x
Effects: holding=nil, pos(x)=table, clear(x)=1

Representational Equivalence

- Any problem that can be represented in one representation can also be represented in the other two
- Can convert in linear time and space, except when converting to set-theoretic (where we get an exponential blowup)

Comparison

- Classical representation
  - Most popular for classical planning, basis of PDDL
- Set-theoretic representation
  - Can take much more space than classical representation
  - Useful in algorithms that manipulate ground atoms directly
    - e.g., planning graphs, SAT
  - Useful for certain kinds of theoretical studies
- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers
  - Useful in non-classical planning problems as a way to handle numbers, functions, time

Richer Specification Languages: ADL

- The previous representations were so-called “STRIPS” representations.
- ADL is a richer, and thus more compact, representation language that allows for
  - Disjunction and Quantification in preconditions and goals
  - Effects that are Quantified, and/or Conditional (effect is conditioned on state)
- PDDL supports STRIPS and ADL, but not all planners support ADL, and not all planners even support a so-called Classical Representation

Compiling to Canonical Action Rep’n

PROS & CONS:
It is possible to compile down ADL actions into STRIPS actions
- Quantification is written as conjunctions/disjunctions over finite universes
- Actions with conditional effects are compiled into multiple (exponentially more) actions without conditional effects
- Actions with disjunctive effects are compiled into multiple actions, each of which take one of the disjuncts as their preconditions
- (Domain axioms can be compiled down into the individual effects of the actions; so all actions satisfy STRIPS assumption)
- Compilation is not always a win-win.
  - By compiling down to canonical form, we can concentrate on highly efficient planning for canonical actions
  - However, often compilation leads to an exponential blowup and makes it harder to exploit the structure of the domain
  - By leaving actions in non-canonical form, we can often do more compact encoding of the domains as well as more efficient search
  - However, we will have to continually extend planning algorithms to handle these representations