CSC2542
Representations
for Classical Planning

Sheila McIlraith
Department of Computer Science
University of Toronto
Winter 2009

Acknowledgements

Some the slides used in this course are modifications of Dana Nau's lecture slides for the textbook *Automated Planning,* licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License: http://creativecommons.org/licenses/by-nc-sa/2.0/

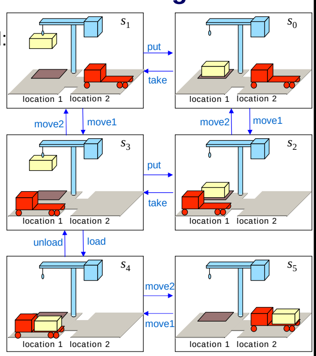
Other slides are modifications of slides developed by Malte Helmert, Bernhard Nebel, and Jussi Rintanen.

I have also used some material prepared by P@trick Haslum and Rao Kambhampati.

I would like to gratefully acknowledge the contributions of these researchers, and thank them for generously permitting me to use aspects of their presentation material.

Quick Review of Classical Planning

8 restrictive assumptions req'd:
- A0: Finite
- A1: Fully observable
- A2: Deterministic
- A3: Static
- A4: Attainment goals
- A5: Sequential plans
- A6: Implicit time
- A7: Offline planning



Representation: Motivation for Approach

Default view:
- represent state explicitly
- represent actions as a transition system (e.g., as an incidence matrix)

Problem:
- explicit graph corresponding to transition system is huge
- direct manipulation of transition system is cumbersome

Solution:

Provide **compact representation** of transition system & induced graph
1. Explicate the structure of the "states"
   - e.g., states specified in terms of state variables
2. Represent actions not as transition system/incidence matrices but as functions (e.g., operators) specified in terms of the state variables
   - An action is applicable to a state when some state variables have certain values. When applicable, it will change the values of certain (other) state variables
3. To plan,
   - Just give the initial state
   - Use the operators to generate the other states as needed

Why is this more compact?

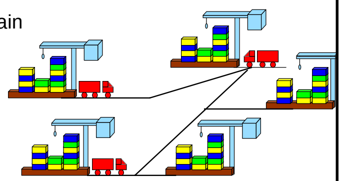
Why is this more compact than an explicit transition system?
- In an explicit transition system, actions are represented as state-to-state transitions. Ech action will be represented by an incidence matrix of size |S|x|S|
- In the proposed model, actions are represented only in terms of state variables 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.

Outline

- Representation schemes
  1. Classical representation
  2. Set-theoretic representation
  3. State-variable representation
- Examples: DWR and the Blocks World
- Comparisons

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, …



Classical Representation

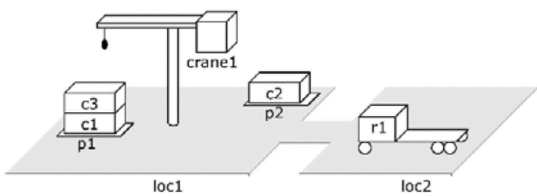
- *Atom*: predicate symbol and args
  - Use these to represent both fixed and dynamic ("fluent") relations

adjacent(*l*,*l'*)	attached(*p*,*l*)	belong(*k*,*l*)
occupied(*l*)	at(*r*,*l*)	
loaded(*r*,*c*)	unloaded(*r*)	
holding(*k*,*c*)	empty(*k*)	
in(*c*,*p*)	on(*c*,*c'*)	
top(*c*,*p*)	top(pallet,*p*)	

- *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 θ 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 Σ's states
  - Only finitely many ground atoms, so only finitely many possible states



{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,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

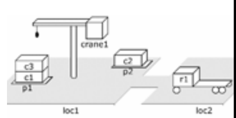
Operators

- *Operator*: a triple *o*=(name(*o*), precond(*o*), effects(*o*))
  - 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*
  - precond(*o*): *preconditions*
    - literals that must be true in order to use the operator
  - effects(*o*): *effects*
    - literals the operator will make true

take(k,l,c,d,p)
;; crane k at location l takes c off of d in pile p
precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)
effects: holding(k,c), $\neg$empty(k), $\neg$in(c,p), $\neg$top(c,p), $\neg$on(c,d), top(d,p)

Actions

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



take(k,l,c,d,p)
;; crane k at location l takes c off of d in pile p
precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)
effects: holding(k,c), $\neg$empty(k), $\neg$in(c,p), $\neg$top(c,p), $\neg$on(c,d), top(d,p)

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)
effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

Notation

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

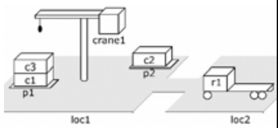
E.g.,

take(k,l,c,d,p)
;; crane k at location l takes c off of d in pile p
precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)
effects: holding(k,c), $\neg$empty(k), $\neg$in(c,p), $\neg$top(c,p), $\neg$on(c,d), top(d,p)

- effects^+(take(*k*,*l*,*c*,*d*,*p*) = {holding(*k*,*c*), top(*d*,*p*)}
- effects^-(take(*k*,*l*,*c*,*d*,*p*) = {empty(*k*), in(*c*,*p*), top(*c*,*p*), on(*c*,*d*)}

Applicability

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

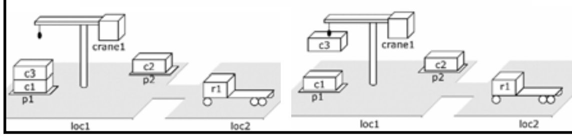


take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)
effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

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

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)
effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)



Operators for the DWR Domain

move(r, l, m)
;; robot r moves from location l to location m
precond: adjacent(l, m), at(r, l), ¬occupied(m)
effects: at(r, m), occupied(m), ¬occupied(l), ¬at(r, l)

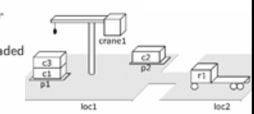
load(k, l, c, r)
;; crane k at location l loads container c onto robot r
precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
effects: empty(k), ¬holding(k, c), loaded(r, c), ¬unloaded(r)

unload(k, l, c, r)
;; crane k at location l takes container c from robot r
precond: belong(k, l), at(r, l), loaded(r, c), empty(k)
effects: ¬empty(k), holding(k, c), unloaded(r), ¬loaded

put(k, l, c, d, p)
;; crane k at location l puts c onto d in pile p
precond: belong(k, l), attached(p, l), holding(k, c), top(d, p)
effects: ¬holding(k, c), empty(k), in(c, p), top(c, p), on(c, d), ¬top(d, p)

take(k, l, c, d, p)
;; crane k at location l takes c off of d in pile p
precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
effects: holding(k, c), ¬empty(k), ¬in(c, p), ¬top(c, p), ¬on(c, d), top(d, p)

- Planning domain: language & operators
- Operators corresponds to a set of state-transition systems



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*: $\mathcal{P} = (\Sigma, s_0, g)$
  - s_0 and g are as above
  - $\Sigma = (S,A,\gamma)$ is a state-transition system
  - S = {all sets of ground atoms in L}
  - A = {all ground instances of operators in O}
  - γ = state-transition function determined by the operators

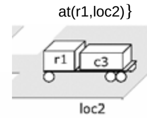
Plans and Solutions

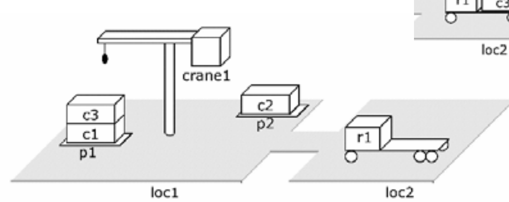
- *Plan*: any sequence of actions $\sigma = \langle a_1, a_2, \dots, a_n\rangle$ such that each a_i is a ground instance of an operator in *O*
- The plan is a *solution* for $P=(O,s_0,g)$ if it is executable and achieves *g*
  - i.e., if there are states $s_0, s_1, \dots, s_n$ such that
    - $\gamma(s_0,a_1)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

Example

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

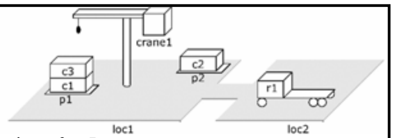
g_1={loaded(r1,c3), at(r1,loc2)}



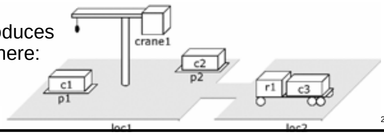


s_1={attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

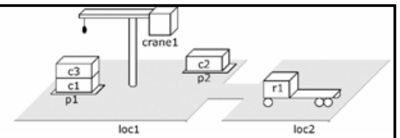
Example (cont.)



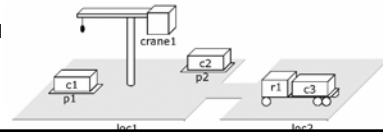
- Here are three solutions for P_1:
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), move(r1,loc1,loc2), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
- Each of them produces the state shown here:



Example (cont.)

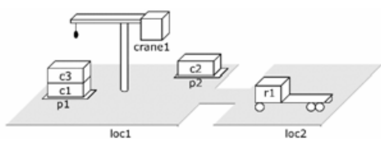


- First is *redundant*: can remove actions and still have a solution
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), move(r1,loc1,loc2), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
- The 2nd and 3rd are *irredundant* and *shortest*



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,

take(k,l,c,d,p)
;; crane k at location l takes c off of d in pile p
precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)
effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)

take all of the operator instances,

e.g., this one

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)
effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

and rewrite ground atoms as propositions

take-crane1-loc1-c3-c1-p1
precond: belong-crane1-loc1, attached-p1-loc1, empty-crane1, top-c3-p1, on-c3-c1
delete: empty-crane1, in-c3-p1, top-c3-p1, on-c3-p1
add: holding-crane1-c3, top-c1-p1

Comparison

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

Problem: Exponential blowup

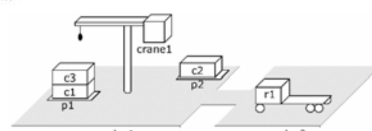
- If a classical operator contains *n* atoms and each atom has arity *k*, then it corresponds to c^{nk} actions where *c* = |{constant symbols}|

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 rep'ns take similar amounts of space
  each can be translated into the other in low-order polynomial time

move(r,l,m)
;; robot r at location l moves to an adjacent location m
precond: rloc(r) = l, adjacent(l,m)
effects: rloc(r) ← m

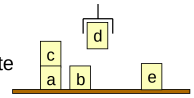
{top(p1)=c3,
cpos(c3)=c1,
cpos(c1)=pallet,
holding(crane1)=nil,
rloc(r1)=loc2,
loaded(r1)=nil, …}



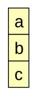
Example: The Blocks World

- 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



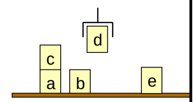
goal



- 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



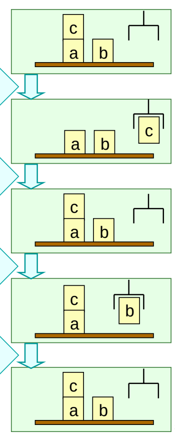
Classical Operators

unstack(*x*,*y*)
Precond: on(*x*,*y*), clear(*x*), handempty
Effects: ~on(*x*,*y*), ~clear(*x*), ~handempty, holding(*x*), clear(*y*)

stack(*x*,*y*)
Precond: holding(*x*), clear(*y*)
Effects: ~holding(*x*), ~clear(*y*), on(*x*,*y*), clear(*x*), handempty

pickup(*x*)
Precond: ontable(*x*), clear(*x*), handempty
Effects: ~ontable(*x*), ~clear(*x*), ~handempty, holding(*x*)

putdown(*x*)
Precond: holding(*x*)
Effects: ~holding(*x*), ontable(*x*), clear(*x*), handempty

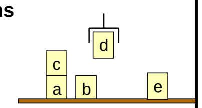


2. Example Set-Theoretic Representation

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
holding-d - the robot hand is holding block d
handempty - the robot hand isn't holding anything
… (31 more)



Set-Theoretic Actions

E.g.,

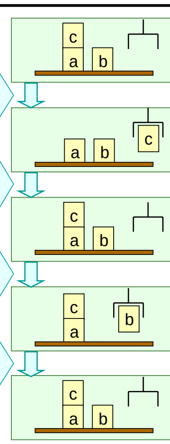
unstack-c-a
Pre: on-c,a, clear-c, handempty
Del: on-c,a, clear-c, handempty
Add: holding-c, clear-a

stack-c-a
Pre: holding-c, clear-a
Del: holding-c, clear-a
Add: on-c-a, clear-c, handempty

pickup-c
Pre: ontable-c, clear-c, handempty
Del: ontable-c, clear-c, handempty
Add: holding-c

putdown-c
Pre: holding-c
Del: holding-c
Add: ontable-c, clear-c, handempty

. . . (46 more)



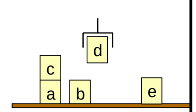
3. Example State-Variable Representation

- 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* is being held 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



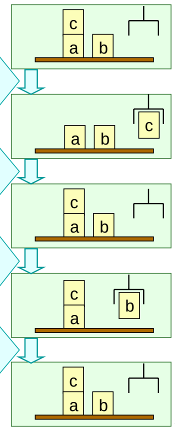
State-Variable Operators

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

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

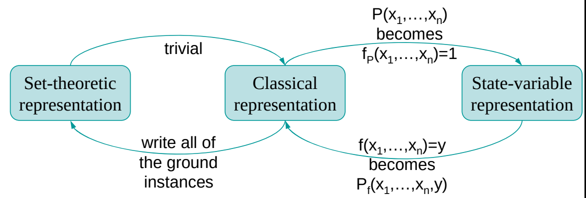
pickup(x : block)
Precond: pos(x)=table, clear(x)=1, holding=nil
Effects: pos(x)=nil, clear(x)=0, holding=x

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