Planning

- Introduction
  - Planning vs. Problem-Solving
  - Representation in Planning Systems
- Situation Calculus
  - The Frame Problem
- STRIPS representation language
  - Blocks World
- Planning with State-Space Search
  - Progression Algorithms
  - Regression Algorithms
- Planning with Plan-Space Search
  - Partial-Order Planning
  - The Plan Graph and GraphPlan
  - SatPlan

Material from Russell & Norvig, chapters 10.3. and 11

Slides based on Slides by Lise Getoor and Tom Lenaerts

Planning problem

- Planning is the task of coming up with a sequence of actions that will achieve a goal starting from an initial state
  - many search-based problem-solving agents are special cases
- Given:
  - a set of action descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,
- Find a plan, which is
  - a sequence of action instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of subgoals to be achieved

Application Scenario

- Classical planning environment
  - fully observable, deterministic, finite, static, discrete
- Practical Applications
  - design and manufacturing
  - military operations
  - games
  - space exploration

Planning vs. Problem Solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
  - States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Planning can analyze the effects of actions
  - The successor function is a black box: it must be “applied” to a state to know which actions are possible in that state and what are the effects of each one
  - An explicit representation of the possible actions and their effects would help the problem solver
- Subgoals can often be planned independently, reducing the complexity of the planning problem
- Search may be through *plan space* rather than *state space*

Key Problems

- Which actions are relevant?
  - Example: Goal is `have(milk)`
    - the agent may have billions of possible actions
      - e.g., one `buy`-action for each possible product in a store
    - an intelligent planner will know that `buy(X)` will cause `own(X)`, and only consider the action `buy(milk)`
- What is a good heuristic functions?
  - Problem:
    - states are domain-specific data structures, and new heuristics must be supplied for each new problem
  - Example: Goal is buying n different items
    - Number of plans grows exponentially with n

  → Problem-independent heuristics are needed
    - e.g., number of subgoals that have already been reached
- How to decompose a problem?

Decomposable Problems

- Goals are often given as a conjunction of subgoals
  - e.g., `have(milk) & have(bread)`
  - each subgoal can be solved independently

Other problems can be decomposed into subproblems:

- Example: overnight delivery of a set of packages
  - Planning a complete route for all packages at once is very expensive ($O(n!)$ different routes)

  → Better decompose the problem:
  - First distribute the packages according to nearest airport to destination
  - Then plan to distribute the package from each airport separately

    → $O(k \cdot (n/k)!)$ different routes (much less than $O(n!)$)

Nearly Decomposable Problems

- Completely decomposable Problems are rare
  - typically there are interactions between subgoals

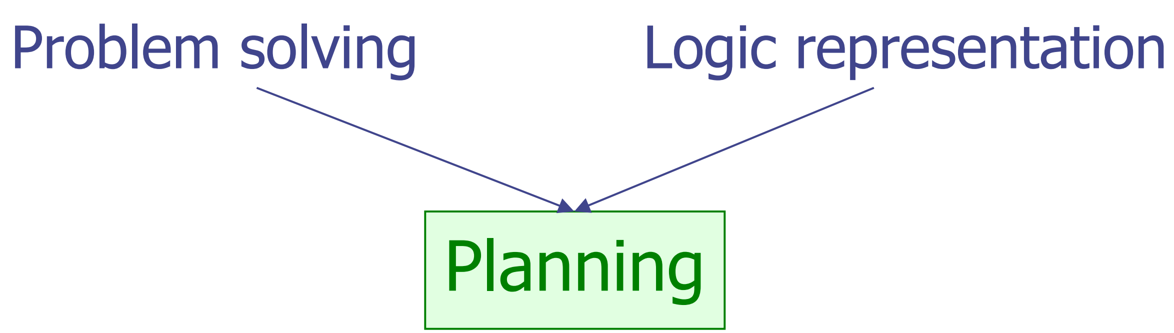
→ Nearly decomposable Problems
  - planning for subgoals is possible
  - but additional work may be required to bring the partial results together
- Example:
  - Independent plans for **`have(milk)`** and **`have(bread)`** may have the result that two different super-markets are visited

Representation in Planning

- In Problem Solving, actions, states, and goals are black boxes
  - each problem has its own representation
  - agent does not understand the representations of actions, states, and goals

  → cannot exploit relations between them

- Planning works with explicit representations of actions, states, and goals
  - typically in some form of logical calculus



Major Approaches to Planning

- Situation calculus
- State space planning
- Partial order planning
- Planning graphs
- Planning with Propositional Logic
- Hierarchical decomposition (HTN planning)
- Reactive planning

Planning in First-Order Logic

Principal Idea:

- Formulate planning problem in First-Order Logic (FOL)
  - states (and goals) are conjunctions of literals
  - actions are logical rules
- Use theorem prover to find a proof for the goal
  - the actions used in this proof are the plan
  - e.g., use PROLOG

Key Problem:

- How to represent change?
  a) add and delete sentences from the KB to reflect changes
  b) all facts are indexed by a situation variable → situation calculus

PROLOG-like Logical Notation

- **Constant:** represents some objects
  - starts with a number or a lower-case letter
    - e.g., `pam, bob, liz, 1, pi, true,` etc.
  - functions are like constants, but complex expressions
- **Variable:** denotes some unknown object/constant
  - starts with an upper-case letter or an underscore
    - e.g. `X, Person, Nummer, _42,` etc.
  - within a conjunction of literals, same variables refer to same objects
  - but may be different objects in different conjunctions / rules
- **Predicate:** denotes a relation between two objects
  - starts with a lower-case letter
    - e.g., `parent, male, female`
- **Literal:** a predicate symbol with some arguments
  - e.g., `parent(pam,bob), at(pam,X), airport(X)`
- **Rule:** an implication, typically written Head :- Cond1, Cond2, ...
  - e.g., `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`

Situation Calculus

- A situation is a snapshot of the world at some instant in time
- Every true or false statement is made with respect to a particular situation
  - Add situation variables to every predicate.
  - `at(agent,1,1)` becomes `at(agent,1,1,s0)`: `at(agent,1,1)` is true in situation (i.e., state) `s0`.
- Add a new function, `result(a,s)`, that maps a situation `s` into a new situation as a result of performing action `a`.
  - For example, `result(forward,s)` is a function that returns the successor state (situation) to `s` after performing action `a`
  - Note that this is just notation!
    - Logical functions are not implemented or evaluated!
    - They are used in pattern matching

Situation Calculus

- Actions can be respresented as logical rules that describe which states can be valid
- Example:
  - The action *agent-walks-to-location-y* could be represented by the PROLOG rule

```
at(A,Y,result(walk(Y),S)) :- at(A,X,S).
```

agent `A` is at location `Y` in state `result(walk(Y),S)`
if it was at location `X` in state `S` (and performed action `walk(Y)`)

- Action sequences are also useful: `results(l,s)` is the result of executing the list of actions `l` starting in `s`:
  - corresponding rules could be included as short-hand notation into inference engine

```
results([],S) = S
results([A|P],S) = results(P,result(A,S))
```

Situation Calculus Planning

- Initial state
  - a logical sentence that describes current situation S_0

```
at(home,s0), not(have(milk,s0)), not(have(bread,s0)),
not(have(drill,s0))
```

- Goal state
  - a logical sentence that describes the goal state

```
at(home,G), have(milk,G), have(bread,G), have(drill,G)
```

- Actions (Operators)
  - logical rules that describe the effects of actions

```
have(milk,result(A,S)) :- at(grocery,S),
                          A = buy(milk).
have(milk,result(A,S)) :- have(milk,S),
                          A != drop(milk).
```

etc.

Situation Calculus Planning

- Solution
  - A sequence of actions `P` (a plan) that, when applied to the initial state, yields a situation satisfying the goal query

    ```
    at(home,G), have(milk,G), have(bread,G), have(drill,G)
    ```

    with

    ```
    G = results(P,s0)
    ```

  - `P` could, for example, be something like

    ```
    P = [go(grocery), buy(milk), buy(bread),
         go(hardwareStore), buy(drill), go(home)]
    ```

- Projection
  - determine the effect of a sequence of actions
- Planning
  - find the sequence of action with the desired effect

The Frame Problem

- the action rules only specify what aspects change when an action is performed

```
have(milk,result(A,S)) :- at(grocery,S),
                          A = buy(milk).
```

- we also need rules that describe what does *not* change!

```
at(grocery,result(A,S)) :- at(grocery,S),
                           A = buy(milk).
```

If we are in a grocery store and buy milk, we remain in the grocery store.

- such frame axioms are necessary for all possible combination of state predicates and actions
- representational frame problem:
  - we do not want to represent each such possible combination
- inferential frame problem:
  - most of the work will be spent in deriving that nothing changes

SC Planning: More Problems

- Qualification problem:
  - difficulty in specifying all the conditions that must hold in order for an action to work
  - e.g., `go` action might fail for various reasons (locked doors, hit by a truck while crossing the street, ...)
- Ramification problem:
  - difficulty in specifying all of the effects that will hold after an action is taken
  - e.g., if the agent carries something, a `go` action will move that thing too...
- Complexity:
  - problem solving (search) is exponential in the worst case
- Optimality:
  - resolution theorem proving can only find *a* proof (plan), not necessarily a *good* plan

Representation Languages for Planning

- Some of the afore-mentioned problems can be solved by better knowledge representation
  - some of them will necessarily remain (e.g., qualification and ramification problems)
- Alternative approach
  - we restrict the language
  - use a special-purpose algorithm (a planner) rather than general theorem prover
- Criteria for a good representation language
  - Expressive enough to describe a wide variety of problems
  - Restrictive enough to allow efficient algorithm
  - Planning algorithm should be able to take advantage of the logical structure of the problem.

The STRIPS Language

- STRIPS (STanford Research Institute Problem Solver)
  - classical planning system (Fikes & Nilsson, 1971)
  - representation of states and actions quite influential

STRIPS: Representation of States

- Decompose the world in logical conditions and represent a state as a conjunction of positive literals.
  - Propositional literals
    - e.g., `poor ∧ unknown`
  - First-Order literals
    - e.g., `at(plane1, melbourne) ∧ at(plane2, sydney)`
    - grounded (contain no variables)
    - function-free (contain no function symbols)
- Closed world assumption
  - what is not known to be true, is assumed to be false

STRIPS: Representation of Goals

- like any other state, a goal is a conjunction of positive ground literals
  - e.g. `rich ∧ famous`
- may be partially instantiated:
  - e.g., `at(P,paris) ∧ plane(P)`
    (some plane should be in Paris)
- A goal is satisfied if the state contains all literals in goal
  - e.g. `rich ∧ famous ∧ miserable` satisfies goal
- In the case of partially instantiated first-order predicates, the state must contain some instantiation of the literals
  - e.g., `at(spirit_of_st_louis,paris) ∧ plane(spirit_of_st_louis)`

  satisfies the goal with the substitution

  $\theta = \{P/spirit\_of\_st\_louis\}$

STRIPS: Representation of Actions

Preconditions: determine the applicability of an action

- conjunction of function-free literals
- all variables that occur here, must also occur in the effects
- the action is applicable if the preconditions match the current state (similar to goals)

Effects: describe the state change after executing an action

- conjunction of function-free literals
- typically divided into:
- ADD-list:
  - facts that become true after executing the action
- DELETE-list
  - facts that become false after executing the action

```
Action( fly(P, From, To),
PRECOND: at(P,From),
         plane(P),
         airport(From),
         airport(To)
ADD:     at(P,To)
DELETE:  at(P,From)
)
```

Semantics of the STRIPS Language

- What actions are applicable in a state?
  - An action is applicable in any state that satisfies the precondition.
  - For First-Order action schema applicability involves a substitution θ for the variables in the PRECOND.
- Example:

```
at(p1,jfk), at(p2,sfo), plane(p1), plane(p2),
 airport(jfk), airport(sfo)
```

satisfies

```
at(P,From), plane(P), airport(From), airport(To)
```

with

$\theta = \{$`P/p1,From/jfk,To/sfo`$\}$

  - Thus the action `fly(P, From, To)` is applicable.

Semantics of the STRIPS Language

- What effects do the actions have?
  - The result of executing action `a` in state `s` is the state `t`
  - `t` is same as `s` except
    - Any literal P in the ADD-list is added
    - Any literal P in the DELETE-list is removed
- Example

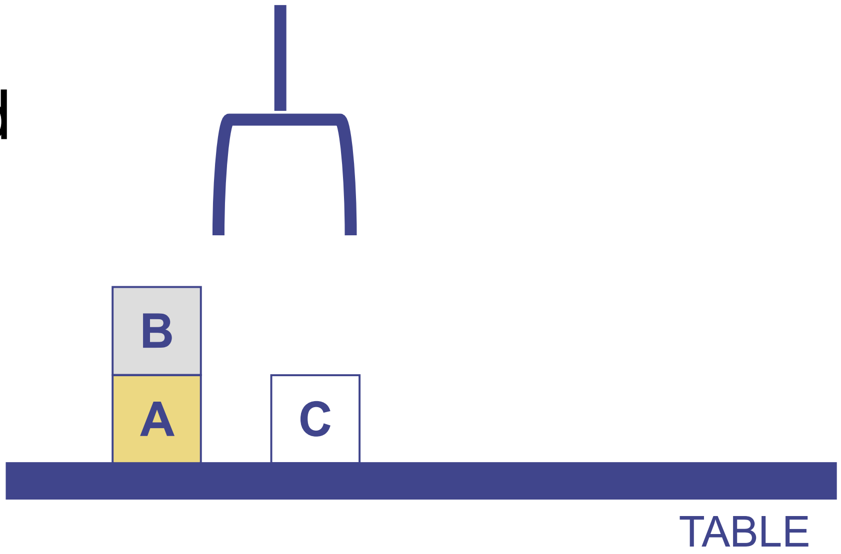
```
ADD:      at(P,To)
DELETE:   at(P,From)
```

with substitution $\theta = \{P/p1, From/jfk, To/sfo\}$ results in state

`at(p1,sfo), at(p2,sfo), plane(p1), plane(p2), airport(jfk), airport(sfo)`

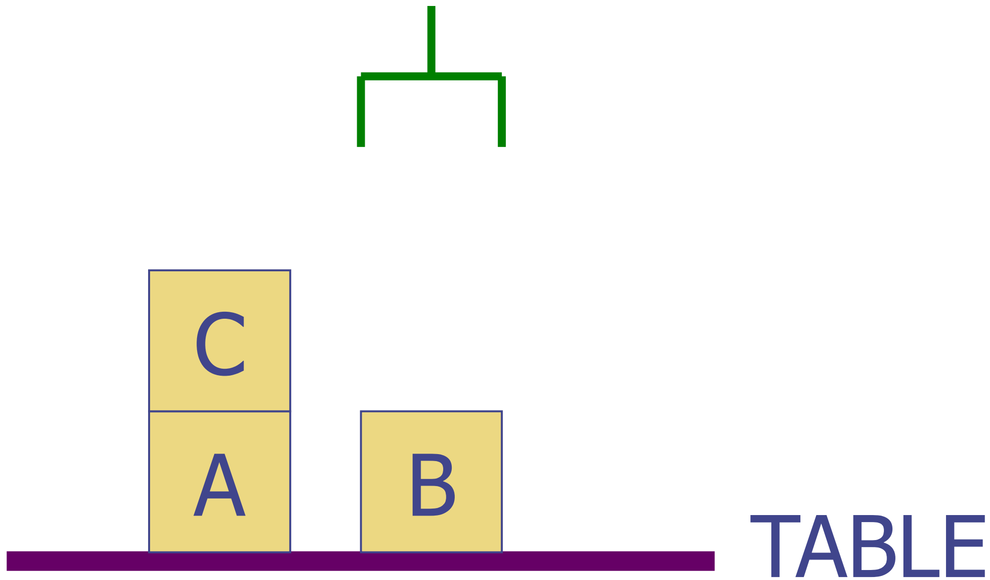
- STRIPS assumption
  - every literal NOT in the effect remains unchanged
  - avoids representational frame problem

Example: Blocks World



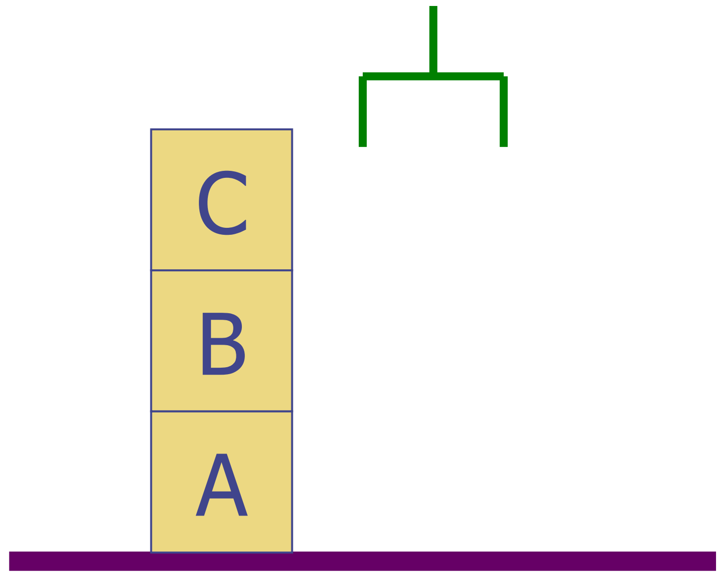
- Very famous AI toy domain
- The blocks world is a micro-world that consists of
  - a table
  - a set of blocks
  - a robot hand
- Operation
  - The robot hand can grasp a single block
  - The robot hand can move over the table (with or without a block)
  - The robot hand can release a block it is holding
  - Blocks can be stacked on top of each other if the top is clear
  - Any number of blocks can be on the table
  - The hand can only hold one block

State Representation



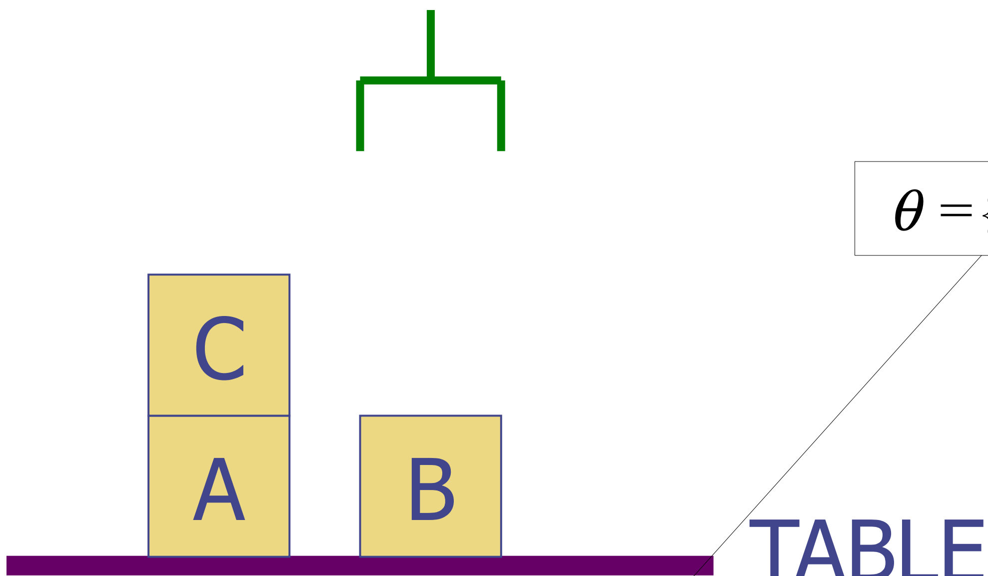

```
block(a), block(b), block(c),
on(a,table), on(b,table), on(c,a),
clear(b), clear(c), handempty
```

Goal Representation



`on(a,table), on(b,a), on(c,b)`

Action Application



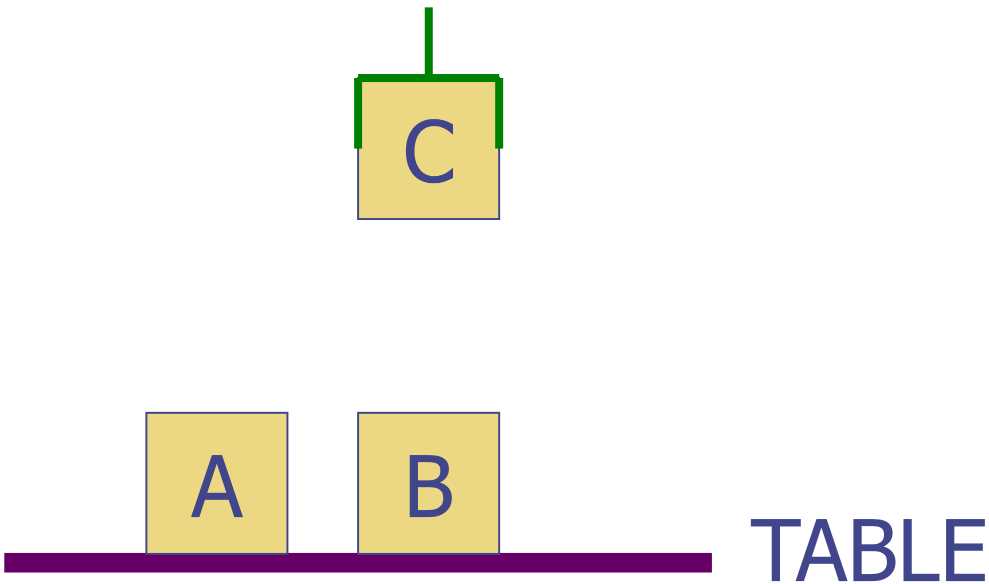

```
Action( unstack(X,Y),
PRECOND:  handempty,
          block(X),
          block(Y),
          clear(X),
          on(X,Y),
ADD:      holding(X),
          clear(Y),
DELETE:   handempty,
          clear(X),
          on(X,Y)
)
```

$\theta = \{X/c,\ Y/a\}$

block(a), block(b), block(c),
on(a,table), on(b,table), ~~on(c,a)~~,
clear(b), ~~clear(c)~~, ~~handempty~~,
holding(c), clear(a)

← unstack(c,a)

Action Application




```
Action( unstack(X,Y),
PRECOND: handempty,
         block(X),
         block(Y),
         clear(X),
         on(X,Y),
ADD:     holding(X),
         clear(Y),
DELETE:  handempty,
         clear(X),
         on(X,Y)
)
```

```
block(a), block(b), block(c),
on(a,table), on(b,table),
clear(b),
holding(c), clear(a)
```

← `unstack(c,a)`

More Blocks-World Actions

```
Action( stack(X,Y),
PRECOND:   holding(X),
           block(X),
           block(Y),
           clear(Y)
ADD:       handempty,
           clear(X),
           on(X,Y),
DELETE:    holding(X),
           clear(Y)
)
```

```
Action( pickup(X),
PRECOND:   handempty,
           block(X),
           clear(X),
           on(X,table),
ADD:       holding(X),
DELETE:    handempty,
           clear(X),
           on(X,table)
)
```

```
Action( putdown(X),
PRECOND:   holding(X)
ADD:       handempty,
           clear(X),
           on(X,table)
DELETE:    holding(X)
)
```

Example: Air Cargo Transport

- Initial state:

```
at(c1,sfo), at(c2,jfk), at(p1,sfo),
at(p2,sfo), cargo(c1), cargo(c2),
plane(p1), plane(p2), airport(jfk),
airport(sfo)
```

- Goal state:

```
at(c1,jfk), at(c2,sfo)
```

```
Action(unload(C,P,A),
PRECOND: in(C,P),
         at(P,A),
         cargo(C),
         plane(P),
         airport(A)
ADD:     at(C,A)
DELETE:  in(C,P)
)
```

```
Action( fly(P,From,To),
PRECOND: at(P,From),
         plane(P),
         airport(From),
         airport(To)
ADD:     at(P,To)
DELETE:  at(P,From)
)
```

```
Action( load(C,P,A),
PRECOND: at(C,A),
         at(P,A),
         cargo(C),
         plane(P),
         airport(A)
ADD:     in(C,P)
DELETE:  at(C,A)
)
```

Expressiveness and Extensions

- The STRIPS language is a very simple subset of FOL
  - Important limitation: function-free literals
    - All such problems can be represented in propositional logic
      - use one proposition for each possible combination of predicate symbol and arguments
    - Function symbols lead to infinitely many states and actions
      - infinitely many arguments can be constructed with function symbols, hence propositionalization is not possible
- Various extensions have been proposed:
  - Action Description language (ADL)
    - recent extension to STRIPS language
    - allows for types, explicit negation (no CWA), relations and conditions in goals, equality predicate built in, ...
  - Planning domain definition language (PDDL)
    - standardization of various AI planning formalisms

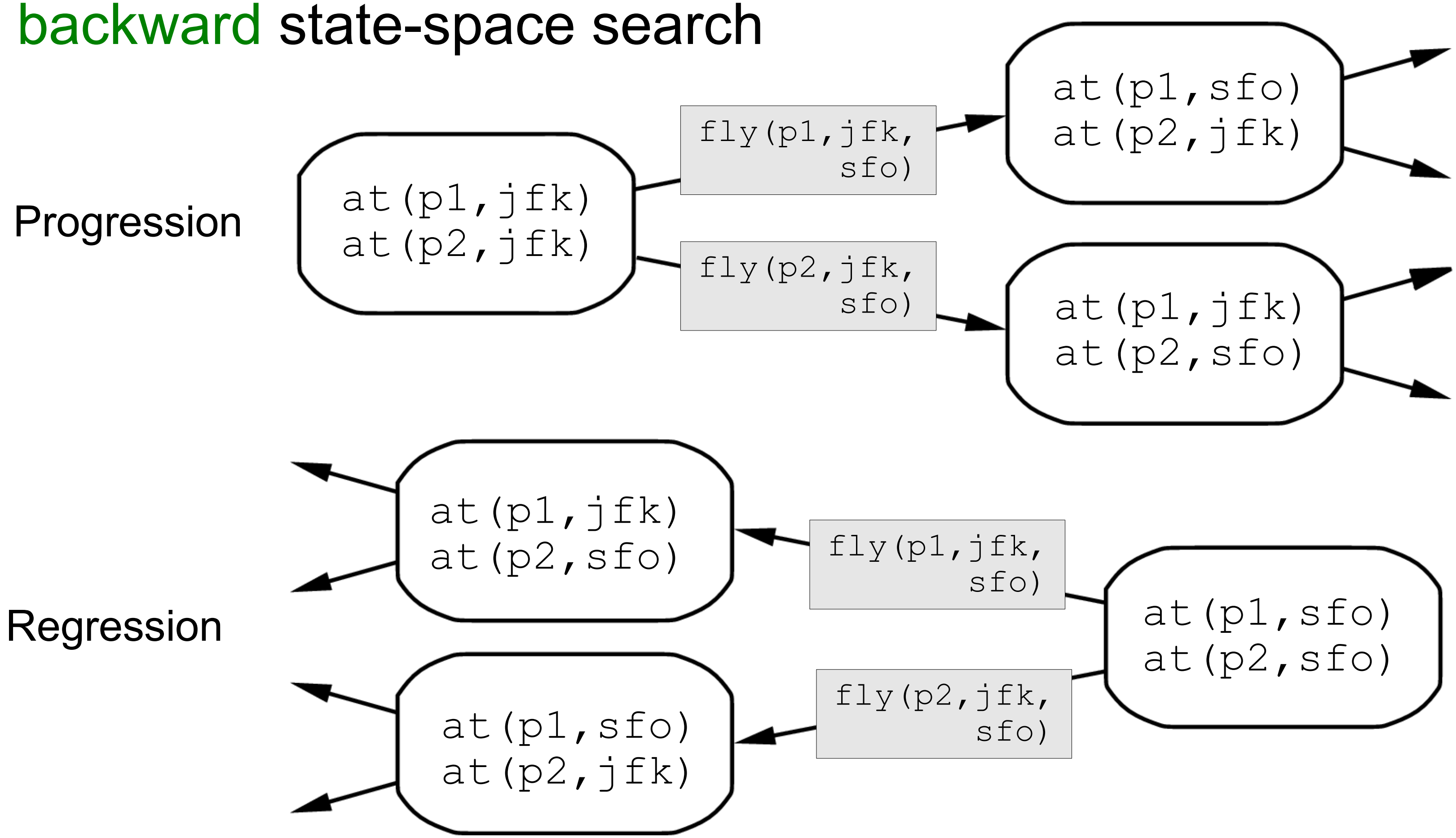
Comparison STRIPS-ADL

STRIPS **Language**	ADL **Language**
Only positive literals in states: $Poor \wedge Unknown$	Positive and negative literals in states: $\neg Rich \wedge \neg Famous$
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect $P \wedge \neg Q$ means add P and delete Q.	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: $Rich \wedge Famous$	Quantified variables in goals: $\exists x\, At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions.	Conditional effects allowed: **when** $P: E$ means E is an effect only if P is satisfied.
No support for equality.	Equality predicate $(x = y)$ is built in.
No support for types.	Variables can have types, as in $(p : Plane)$.

Figure 11.1 Comparison of STRIPS and ADL languages for representing planning problems. In both cases, goals behave as the preconditions of an action with no parameters.

Planning with State-Space Search

- Progression planners
  - forward state-space search
- Regression planners
  - backward state-space search



Progression Algorithm

Formulation as state-space search problem:

- Initial state = initial state of the planning problem
  - Literals not appearing are false
- Actions = those whose preconditions are satisfied
  - Add positive effects, delete negative
- Goal test = does the state satisfy the goal
- Step cost = each action costs 1
  - could be changed if necessary

Search Algorithms

  - function-free → finite → any complete graph search algorithm will yield a complete planner
- Efficiency is a problem
  - irrelevant action problem
  - good heuristic required for efficient search

Regression Algorithm

- In order to be able to use a backward search, we must be able to apply the STRIPS operators backwards
- Relevant actions
  - actions that achieve one of the subgals
    - i.e., the subgoal is on the actions' ADD-list
  - Example:
    - Goal state:
      `at(c1,a), at(c2,a),..., at(c20,a)`
    - Relevant action for first conjunct: `unload(c1,P,a)`
- Consistent actions
  - Actions must not undo subgoals that are already achieved
  - Example:
    - `load(c1,p)` will never appear in a plan for the above task because it will delete the subgoal `at(c1,a)` which has been achieved with the first action

→ How can an action be applied backwards?

Inverse Action Application

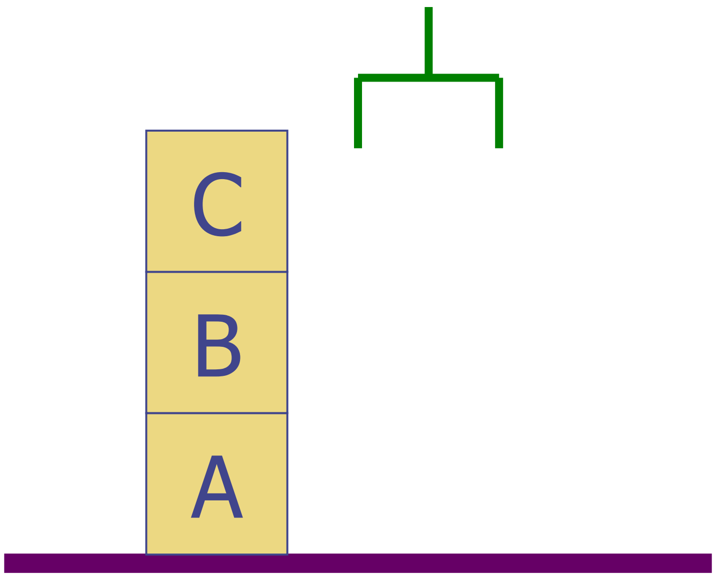
General process for predecessor construction

- Given a goal description G
- Let A be an action that is relevant and consistent
- The predecessor state is determined as follows:
  - Positive effects of A that appear in G are deleted.
    - because they are assumed to have been added by A (otherwise we do not need A in the plan)
  - Each precondition literal of A is added (unless it already appears)
    - because in order to apply A, we must now make find actions that enable the precconditions.

→ New Goal = Old Goal – ADD(A) + PRECOND(A)

Inverse Action Application

- Goal:

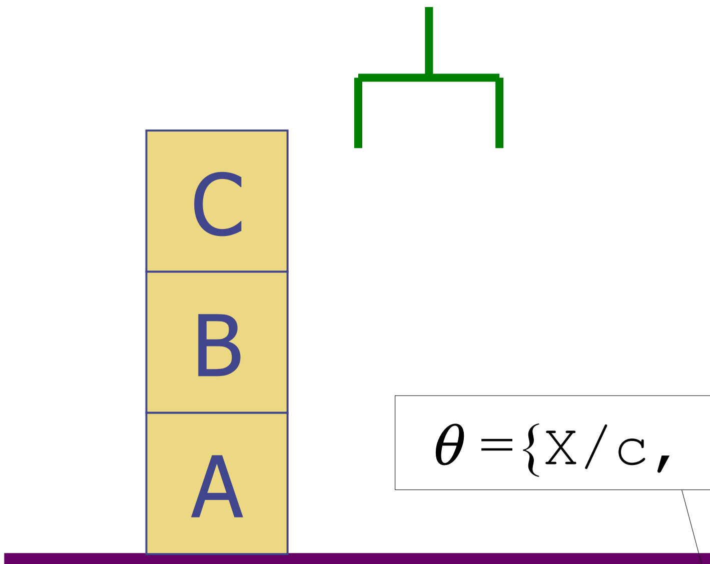



```
Action( stack(X,Y),
PRECOND:  holding(X),
          block(X),
          block(Y),
          clear(Y)
ADD:      handempty,
          clear(X),
          on(X,Y),
DELETE:   holding(X),
          clear(Y)
)
```

`on(a,table), on(b,a), on(c,b)`

Inverse Action Application

- Goal:



$\theta = \{X/c,\ Y/b\}$

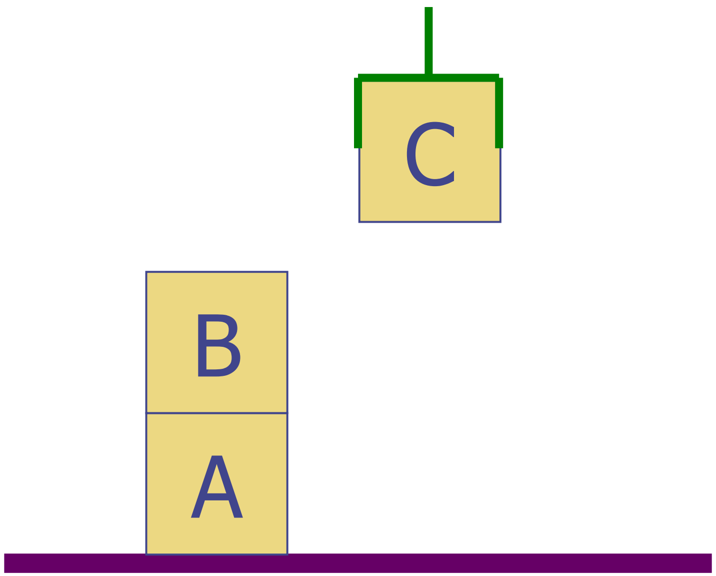
```
Action( stack(X,Y),
PRECOND: holding(X),
         block(X),
         block(Y),
         clear(Y)
ADD:     handempty,
         clear(X),
         on(X,Y),
DELETE:  holding(X),
         clear(Y)
)
```

on(a,table), on(b,a), ~~on(c,b)~~ — stack(c,b)

holding(c), block(c), block(b), clear(b)

Inverse Action Application

- New Goal:




```
Action( stack(X,Y),
PRECOND:  holding(X),
          block(X),
          block(Y),
          clear(Y)
ADD:      handempty,
          clear(X),
          on(X,Y),
DELETE:   holding(X),
          clear(Y)
)
```

`on(a,table), on(b,a),`

`holding(c), block(c), block(b), clear(b)`

Regression Algorithm

Formulation as state-space search problem:

- Initial state = goal state of the planning problem
  - Literals not appearing may be true or false
- Actions = those whose add-list satisfy the current state
  - delete positive effects, add preconditions
- Goal test = is the current state satisfied in the initial state of the planning problem?
- Step cost = each action costs 1
  - could be changed if necessary

Search algorithm

  - again, any standard algorithm can perform the search
- Main Advantage of Regression Planning
  - only relevant actions are considered

  → often much lower branching factor than for forward search

Heuristics for State-Space Search

- Even for regression we need good heuristics
  - How many actions are needed to achieve the goal?
  - Exact solution is NP hard, find a good estimate

Two approaches to find an admissible search heuristic:

- The optimal solution to a relaxed problem
  - remove all preconditions from actions
    - almost identical to the number of open subgoals
  - remove only the delete-list and find a (minimal) set of actions that collectively achieve the goals
    - problem: finding a minimal set cover is NP-hard, and relaxing the constraint looses admissibility of heuristic
- The subgoal independence assumption:
  - The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving them independently
  - is only admissible if co-ordination causes additional complexity (not admissible for the `have(milk) & have(bread)` plan)