15-381: AI
Classical Deterministic Planning – Representation and Search

Fall 2009

Manuela Veloso

(Thanks to Reid Simmons for the blocksworld example run.)

Carnegie Mellon

Outline

- Planning
- Actions, states, goals
- Linear planning
- Beyond linear planning

Problem Solving – Planning

Newell and Simon 1956

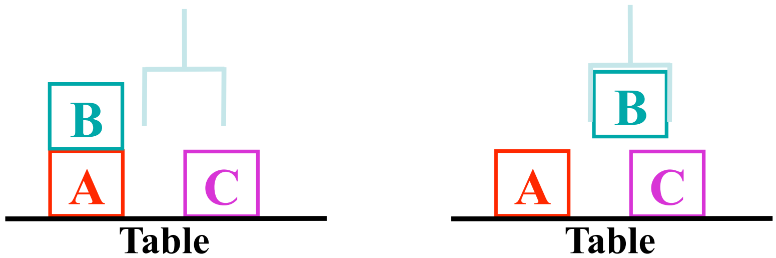
- Given the *actions* available in a task domain.
- Given a problem specified as:
  - an initial *state* of the world,
  - a set of *goals* to be achieved.
- Find a *solution* to the problem, i.e., a *way* **to transform** the initial state into a new state of the world where the goal statement is true.

Action Model, State, Goals

Classical Deterministic Planning

- Action Model:
  - How *to represent* actions
  - Deterministic, correct, rich representation
- State:
  - single initial state, *fully known*
- Goals:
  - *complete* satisfaction

The Blocks World Definition - Actions



- Blocks are picked up and put down by the arm
- Blocks can be picked up only if they are clear, i.e., without any block on top
- The arm can pick up a block only if the arm is empty, i.e., if it is not holding another block, i.e., the arm can be pick up only one block at a time
- The arm can put down blocks on blocks or on the table

Note: assume table is infinite.

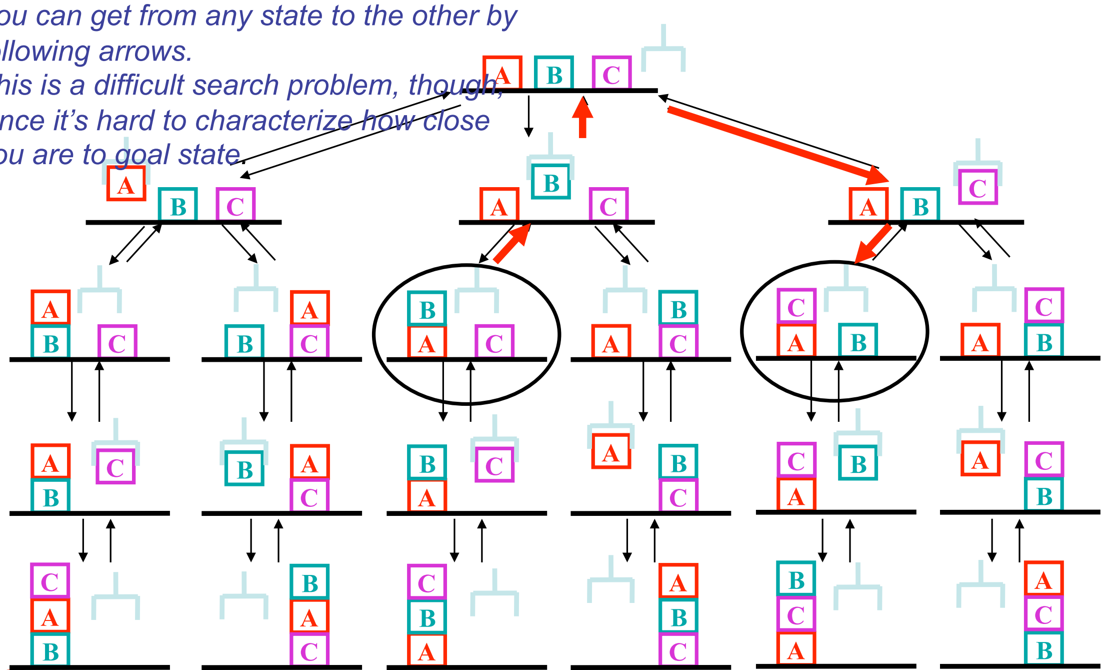
Planning by "Plain" State Search

- **Search from** an ***initial state*** of the world to a ***goal state***
- Enumerate all states of the world
- Connect states with legal actions
- Search for paths between initial and goal states

This isn't great when there are a whole lot of states.

3-BlocksWorld State Transitions

You can get from any state to the other by following arrows.
This is a difficult search problem, though, since it's hard to characterize how close you are to goal state.



Planning - Generation

- Many plan generation algorithms:
  - **Forward from state, backward from goals**
  - **Serial, parallel search**
  - **Logical satisfiability**
  - **Heuristic search**
  - **………..**

Make sure you really get the following slides. For the midterm, you may want to be able to perform DFS, BFS, etc, on block world. And be able to describe how to do all these things.

Planning – Actions and States

- ***Model*** of an action
  - a description of *legal actions* in the domain
    - "move queen", "open door if unlocked", "unstack if top is clear",….
- ***Model*** of the state
  - **Numerical identification (s1, s2,...) – no information**
  - **"Symbolic" description**
    - **objects, predicates**

The representation problem is hard– as humans we often underestimate how difficult it is to precisely describe a problem domain.

The Blocks World - States

- Objects
  - Blocks: *A, B, C*
  - Table: *Table*
- Predicates
  - *(on A B), (on C table), (clear B), (arm-empty), (holding C)*
  - *(on table A), (on A C), (top B),…*
  - *(tower A B C),…*
- States – Conjunctive
  - (on A B) and (on B C) and (clear A) and (on C table) and (arm-empty)

*Note that different representations can have a **huge** impact on the search for solution. Here we choose the conjunctive (binary) representation.*

Model of World States

- Numerical identification (s1, s2,...)
- Symbolic description
  - Features
  - Predicates
  - Conjunctive, enumerative, observable
  - Complete, correct, deterministic
- Probabilistic, approximate, incremental, on-demand

Action Representation - BlocksWorld

STRIPS planner, by Fikes & Nilsson (Stanford)
Work with preconditions and effects (add and delete) on literals

```
(OPERATOR PICK_FROM_TABLE
?ob BLOCK
:preconds
 (and (clear ?ob)
      (on-table ?ob)
      (arm-empty))
:effects
  del (on-table ?ob)
  del (clear ?ob)
  del (arm-empty)
  add (holding ?ob))
```

Preconds: To pick, object must be clear (nothing on it), must be on table, and can't be holding something else.

```
(OPERATOR
PICK_FROM_BLOCK
?uob BLOCK
?uob BLOCK
:preconds
 (and (on ?ob ?uob)
      (clear ?ob)
      (arm-empty))
:effects
   del (on ?ob ?uob)
   del (clear ?ob)
   del (arm-empty)
   add (holding ?ob)
   add (clear ?uob))
```

STRIPS assumption: anything not mentioned in the effect remains unchanged. E.g. one move in sliding puzzle only changes 2 tiles.

Effects: Once you pick it up, it's not on the table, so it's no longer clear, and your arm isn't empty– rather it's holding the object. Note how careful we are, we don't want to leave anything out. We need to keep the state consistent.

Action Representation - BlocksWorld

```
(OPERATOR PUT_ON_BLOCK
  ?ob BLOCK
  ?uob BLOCK
  :preconds
     (and (clear ?uob)
       (holding ?ob))
  :effects
   del (holding ?ob)
   del (clear ?uob)
   add (clear ?ob )
   add (arm-empty)
   add (on ?ob ?uob))
```

```
(OPERATOR PUT_DOWN_ON_TABLE
?ob
BLOCK                    :precon
ds
      (holding ?ob)
 :effects
   del (holding ?ob)
   add (clear ?ob)
   add (arm-empty)
   add (on-table ?ob))
```


Different Representation - Blocksworld

- *MOVE*(*x,y,z*) moves block x from the top of y to the top of z. y and z can be either the table or another block. *MOVE* is applicable only if x and z are clear, and x is on y.

```
(OPERATOR MOVE
 :preconds
   ?block BLOCK
   ?from OBJECT
   ?to OBJECT
  (and (clear ?block)
      (clear ?to)
      (on ?block ?from)
 :effects
  add (on ?block ?to)
  del (on ?block ?from)
  (if (block-p ?from)
      add (clear ?from))
  (if (block-p ?to)
      del (clear ?to)))
```

You'll have to be able to make a representation, either on HW3 or on the midterm.

STRIPS Action Representation

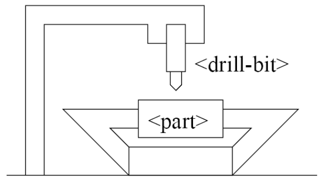
- Actions - operators -- rules -- with:
  - Precondition expression -- must be satisfied before the operator is applied.
  - Set of effects -- describe how the application of the operator changes the state.
- Precondition expression: propositional, typed first order predicate logic, negation, conjunction, disjunction, existential and universal quantification, and functions.
- Effects: add-list and delete-list.
- Conditional effects -- dependent on condition on the state *when action takes place*.

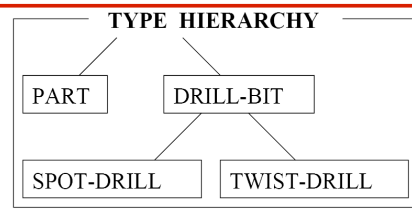
Here's a formal definition– try to identify these things in the previous slides.

Many Planning "Domains"

- Web management agents
- Robot planning
- Manufacturing planning
- Image processing management
- Logistics transportation
- Crisis management
- Bank risk management
- Blocksworld
- Puzzles
- Artificial domains

Example – Action Model





drill-spot (<part>, <drill-bit>)
<part>: type PART
<drill-bit>: type SPOT-DRILL
Pre: (holding-tool <drill-bit>)
(holding-part <part>)
Add: (has-spot <part>)

put-drill-bit (<drill-bit>)
<drill-bit>: type DRILL-BIT
Pre: tool-holder-empty
Add: (holding-tool <drill-bit>)
Del: tool-holder-empty

put-part(<part>)
<part>: type PART
Pre: part-holder-empty
Add: (holding-part <drill-bit>)
Del: part-holder-empty

drill-hole(<part>, <drill-bit>)
<part>: type PART
<drill-bit>: type TWIST-DRILL
Pre: (has-spot <part>)
(holding-tool <drill-bit>)
(holding-part <part>)
Add: (has-hole <part>)

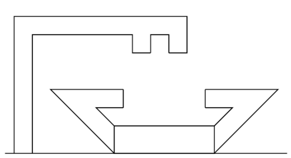
remove-drill-bit(<drill-bit>)
<drill-bit>: type DRILL-BIT
Pre: (holding-tool <drill-bit>)
Add: tool-holder-empty
Del: (holding-tool <drill-bit>)

remove-part(<part>)
<part>: type PART
Pre: (holding-part <drill-bit>)
Add: part-holder-empty
Del: (holding-part <drill-bit>)

Note that drill-hole can go last, since has-hole is not a precondition for anything
Is there anything that needs to be a precondition at start, since nothing adds it?
What is an alg to search for a solution? For starters, if you want an action, trace backwards for an action that causes these preconditions. For ex. Precond to drill-hole is has-spot. However, the tricky part is achieving these preconditions in the right order.

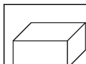
Example – Problem and Plan

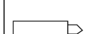
Initial State

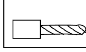


part-holder-empty
drill-holder-empty

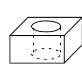
Set of Objects


part-1, part-2 : type PART


drill-1: type SPOT-DRILL


drill-2, drill-3 : type TWIST-DRILL

Goal Statement


has-hole (part-1)

```
put-part(part-1)
put-drill=bit(drill-1)
drill-spot(part-1, drill-1)
remove-drill-bit(drill-1)
put-drill-bit(drill-2)
drill-hole(part-1, drill-2)
```

General problem solver

GPS – Means-ends Analysis

(*Newell and Simon 60s*) (*Ernst and Newell 69*)

GPS Algorithm (*initial-state*, *goals*)

- If *goals* $\subseteq$ *initial-state*, then return *True*
- Choose a difference $d \in$ *goals* between *initial-state* and *goals*
- Choose an operator o to reduce the difference d
- If no more operators, then return *False*
- *State*=**GPS**(*initial-state*, preconditions(o))
- If *State*, then return **GPS**(apply(o, *initial-state*), *goals*)

You keep a search stack, recursively calling GPS

Here's an illustration of the challenge– we have multilple goals– C on B and A on C. This makes it harder to get a good heuristic, as we did in search.

GPS Blocks-World Example

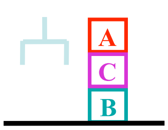
1. Search Stack

On(A, C) On(C, B)

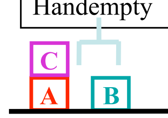
Cyan: Goal.

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty



Goal



Initial State

2. Search Stack

On(A, C) On(C, B)
On(A, C)
On(C, B)

Green: Difference between init and goals

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

3. Search Stack

On(A, C) On(C, B)
On(A, C)
Put_Block(C, B)
Holding(C) Clear(B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

We took all available actions with On(C,B) as postcond and picked one (in magenta). Now we push the precond of Put_Block(C,B) and add to stack as a goal.

4. Search Stack

On(A, C) On(C, B)
On(A, C)
Put_Block(C, B)
Holding(C) Clear(B)
Holding(C)
Clear(B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

We're on state = GPS(init, precond(Put_Block(C,B)) Compute the differences– add the two greens to the stack.

GPS Blocks-World Example

9. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)

State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)
Clear(A)

[Pick_Block(C)]

10. Search Stack

State
- On(A, C) On(C, B)
- On(A, C)

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C); Put_Block(C, B)]

11. Search Stack

State
- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C)
Put_Block(C, B)]

12. Search Stack

State
- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Holding(A)
- Clear(C)

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C)
Put_Block(C, B)]

GPS Blocks-World Example

5. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Holding(C)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

Now take Clear(B) from the stack since it's in the state already. PB(C) is the next operator.

6. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Pick_Block(C)
- Handempty Clear(C) On(C, ?b)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

Call GPS with cyan as new goals.

7. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Pick_Block(C)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

All of Handempty, Clear(C), and C on something are in the state, so return true. Apply PB(C)

8. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)

State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)
Clear(A)

[Pick_Block(C)]

Now call GPS(PB(C)(init-state), current goals). Where current goals are the top of the stack– holding(C) and clear(B). These are already in, return true again.

GPS Blocks-World Example

13. Search Stack

State

- **On(A, C) On(C, B)**
- **Put_Block(A, C)**
- **Holding(A) Clear(C)**
- **Holding(A)**

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C);
Put_Block(C, B)]

Now call GPS on the action, so our new goal diff is Holding(A)

14. Search Stack

State

- **On(A, C) On(C, B)**
- **Put_Block(A, C)**
- **Holding(A) Clear(C)**
- **Pick_Table(A)**
- **Handempty Clear(A) On(A, Table)**

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

Recurse back, try to get to holding(A)

[Pick_Block(C); Put_Block(C, B)]

15. Search Stack

State

- **On(A, C) On(C, B)**
- **Put_Block(A, C)**
- **Holding(A) Clear(C)**
- **Pick_Table(A)**

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C);
Put_Block(C, B)]

Preconds to holding(A) were fulfilled already, so add that to the actions.

16. Search Stack

St

- **On(A, C) On(C, B)**
- **Put_Block(A, C)**
- **Holding(A) Clear(C)**

Clear(C)
On(B, Table)
Clear(A)
On(C, B)
Holding(A)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

Add PT(A) to the actions, and move down to the next goals on the stack. Oh look, they're already in our state.

GPS Blocks-World Example

17. Search Stack

State

- **On(A, C) On(C, B)**
- **Put_Block(A, C)**

Clear(C)
On(B, Table)
Clear(A)
On(C, B)
Holding(A)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

So that means we want our action PB(A,C). Add to our actions.

18. Search Stack

State

- **On(A, C) On(C, B)**

On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

Now we're down to our original goals.

19. Search Stack

State

On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

And since our goals are a subset of our state, we return true. Since our stack's empty we're done.

Properties of Planning Algorithms

- **Soundness**
  - A planning algorithm is ***sound*** if all solutions found are legal plans
    - All preconditions and goals are satisfied
    - No constraints are violated (temporal, variable binding)
- **Completeness**
  - A planning algorithm is ***complete*** if a solution can be found whenever one actually exists
  - A planning algorithm is ***strictly complete*** if all solutions are included in the search space
- **Optimality**
  - A planning algorithm is ***optimal*** if the order in which solutions are found is consistent with some measure of plan quality

Optimality can be complicated. For instance, we may have many factors affecting optimality of a subway trip (stations with escalators, crowds on trains, etc). "Best" is complex.

Why is Planning Hard?

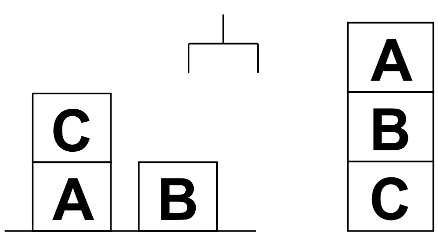
Planning involves a complex search:

- Alternative operators to achieve a goal
- Multiple goals that interact
- Solution optimality, quality
- Planning efficiency, soundness, completeness

Linear Planning: Discussion

- **Advantages**
  – Reduced search space, since goals are solved one at a time
  – Advantageous if goals are (mainly) independent
  – Linear planning is ***sound***
- **Disadvantages**
  – Linear planning may produce ***suboptimal*** solutions (based on the number of operators in the plan)

The Sussman Anomaly



Here you get really stuck because you're only working on one goal at a time.

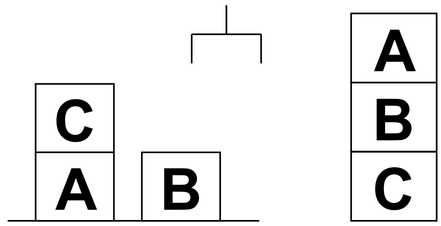
Because "C on table" isn't a goal, we don't do the obvious.

So, when you have multiple goals, the ordering really affects what you do.

Linear Solution:
- *(on B C)*
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Unstack (B, C)***
  - ***Putdown (B)***
  - ***Unstack (C, A)***
  - ***Putdown (C)***
  - ***Stack (A, B)***
- *(on B C)*
  - ***Unstack (A, B)***
  - ***Putdown (A)***
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Pickup (A)***
  - ***Stack (A,B)***

Linear Solution:
- *(on A B)*
  - ***Unstack (C, A)***
  - ***Putdown (C)***
  - ***Stack (A, B)***
- *(on B C)*
  - ***Unstack (A, B)***
  - ***Putdown (A)***
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Pickup (A)***
  - ***Stack (A,B)***

"NonLinear" Solution



NonLinear Solution:

- ***(on A B)***
    - ***Unstack (C, A)***
    - ***Putdown (C)***
- ***(on B C)***
    - ***Pickup (B)***
    - ***Stack (B, C)***
- ***(on A B)***
    - ***Pickup (A)***
    - ***Stack (A, B)***

The only way to solve this is goal switching. When you're on the way to (on A B), you interrupt it and switch to (on B C). Goals are now a set, rather than a stack. Branching factor increases dramatically, but you can actually solve the problem.

Linear Planning – Goal Stack

- Planner can be unoptimal
- Planner's efficiency is sensitive to goal orderings
    - Control knowledge for the "right" ordering
    - Random restarts
    - Iterative deepening
- Planner keeps a small search space by not considering all the possible goal orderings.
- Any other problems/features?

Example: One-Way Rocket (Veloso 89)

```
(OPERATOR LOAD-ROCKET
 :preconds
  ?roc ROCKET
  ?obj OBJECT
  ?loc LOCATION
 (and (at ?obj ?loc)
      (at ?roc ?loc))
 :effects
  add (inside ?obj ?roc)
  del (at ?obj ?loc))
```

```
(OPERATOR UNLOAD-ROCKET
 :preconds
  ?roc ROCKET
  ?obj OBJECT
  ?loc LOCATION
 (and (inside ?obj ?roc)
      (at ?roc ?loc))
 :effects
  add (at ?obj ?loc)
  del (inside ?obj ?roc))
```

```
(OPERATOR MOVE-ROCKET
 :preconds
  ?roc ROCKET
  ?from-l LOCATION
  ?to-l LOCATION
 (and (at ?roc ?from-l)
      (has-fuel ?roc))
 :effects
  add (at ?roc ?to-l)
  del (at ?roc ?from-l)
  del (has-fuel ?roc))
```

Sussman's anomaly showed "unoptimal". But sometimes linear planning can even be not complete. Here, you run out of fuel– we have nonreversible actions, unlike in the block world where you can always return to a past state.

Incompleteness of Linear Planning

```
Initial state:
          (at obj1 locA)
          (at obj2 locA)
          (at ROCKET locA)
          (has-fuel ROCKET)
```

```
Goal statement:
    (and
      (at obj1 locB)
      (at obj2 locB))
```

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	*failure*

State-Space Nonlinear Planning

Extend linear planning [Prodigy4.0]:

- From **stack** to **set** of goals.
- Include in the search space all possible interleaving of goals

State-space nonlinear planning is **complete**.

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA)
(at obj2 locB)	(LOAD-ROCKET obj2 locA)
(at obj1 locB)	(MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	(UNLOAD-ROCKET obj1 locB)

Summary

- **State and Action Representation:** predicates, conjunction of predicates; preconditions, adds, deletes
- **Planning:** selecting one sequence of actions (operators) that transform (apply to) an initial state to a final state where the goal statement is true.
- **Means-ends analysis:** identify and reduce, as soon as possible, *differences* between state and goals.
- **Linear planning:** backward chaining with means-ends analysis using a stack of goals - potentially efficient, possibly unoptimal, incomplete; GPS, STRIPS.
- **Nonlinear planning with means-ends analysis:** backward chaining using a set of goals; reason about *when* “to reduce the differences;” Prodigy4.0.