The Planning Problem

Given:

1. an initial state I,
2. a goal state G, and
3. a set of operators O,

produce a plan P such that executing P in state I results in state G.

Example:

- I = 5
- G = 11
- O = { increment, decrement, double, halve }

Plan P_1	Plan P_2	Plan P_3
1. increment	1. double	1. double
2. increment	2. double	2. increment
3. increment	3. decrement	
4. increment	4. decrement	
5. increment	5. halve	
6. increment	6. increment	
	7. increment	

An Exploratory Exercise

Describe the planning problem as a *search* problem.

- What is the initial state?
- What is the goal state?
- How can you recognize goal states?
  That is, what is the goal test?
- What operators can the agent use to change state?
  [OR]
  What successor function can be used to generate the set of states reachable from the current state?
- What deadend states can the agent reach, and how can it recognize deadends?

To make your thinking more concrete, use a "real" problem to ground your discussion. To make your thinking more interesting, use a problem different from the one we just saw. I suggest:

Develop a plan for a person who is getting out of bed in the morning that results in the person being dressed for class.

Planning as Search

What is the initial state?

have right shoe + have left shoe +
have right sock + have left sock +
bare right foot + bear left foot

What is the goal state?

wearing right shoe + wearing left shoe +
wearing right sock + wearing left sock

How can you recognize goal states? That is, what is the goal test?

is the current state equal to the goal state? or
is the goal state a **subset** of the current state?

What operators can the agent use to change state?

put on right shoe put on left shoe
put on right sock put on left sock
take off right shoe take off left shoe
take off right sock take off left sock

What deadend states can the agent reach, and how can it recognize deadends?

Without the “take off” operators, then any state in which I am wearing my right shoe but not my right sock is a deadend. Likewise for the left foot.

How Good is the View of Planning as Search?

Advantages

- The formalism is simple.
- We know a rich set of algorithms for doing search.
- The result of search is a path from the start state to a goal state. Just return that path as the answer, the plan.

Disdvantages

- The combinatorics of the problem are huge.
  - The size of state descriptions grows rapidly.
  - The number of operators becomes infinite if we allow variables, but not allowing variables means having to create customized operators for every planning problem.
  - Even heuristic control of search eliminates only certain states after their generation. Wouldn't we like agents to be able to rule out entire sets of actions in certain circumstances?

.

How Good is the View of Planning as Search?

A major disadvantage:

Search overcommits.

Do I really want my plan to commit me to putting my left shoe before I put on my right shoe???

Plan P_1	Plan P_2	Plan P_3
1. left sock	1. left sock	1. left sock
2. left shoe	2. right sock	2. right sock
3. right sock	3. left shoe	3. right shoe
4. right shoe	4. right shoe	4. left shoe

Planning Viewed as Logical Inference

Write a set of sentences that describe the world:

```
bare( leftFoot )
bare( rightFoot )
...
if   bare( X )       and
     have( Y )       and
     wearOn( Y, X )
then not bare( X )
...
[] p   == p( t ) and [] p( t+1 )
O p    == p( t ) or  O p( t+1 )
```

Use an inference engine to derive the goal state:

```
wearing( leftShoe )
wearing( rightShoe )
...
```

And return the *set of support* for each sentence in the goal state.

Planning Viewed as Logical Inference

Advantages

- States are explicit.
  I and G are sets of sentences.

- Operators are explicit.
  Each operator in O is a set of sentences.

- Predicate logic with only a few extensions gives a rich language for describing states and operators.

Disdvantages

- Inference in predicate logic is very expensive, both in time and space.

- Even restricting the language **severely** doesn't sufficiently offset the costs in time and space.

Planning as its Own Problem

Search and logical inference seem to complement each other, with the strengths of one offsetting the weaknesses of the other.

Can we combine the two to create a planning method that is better?

- Use logic to write state descriptions and operators and to reason about them.

- Use search-style algorithms to build the plans.

The Key Ideas of Planning

1. Planning problems are decomposable and (mostly) independent, so our planner should be able to recognize this and use it to the planner's advantage.

   Example: dressing my feet.

2. If plans are decomposable, then operators and states should be, too.

3. A planner should be able to choose any action that makes sense and add it to the "right place" in the plan at any time.

   Example: putting on shoes and socks

A Simple Form of Planning: Goal-Stack Planning

The earliest work on planning in AI used a natural idea: place goals to be achieved on a stack and then try to achieve each...

To demonstrate so-called **goal-stack planning**, we need to use a knowledge representation that allows us to decompose and recompose states and goals:

States: a state is a set of positive function-free atoms

- Example:

  ```
  { clear(A), on(A,B) . . . }
  ```

- Illegal:

  ```
  not on(A, B)
  on(A, B) V onTable(A)
  V?x: clear(?x) -> NOT ]?y: on(?y, ?x)
  ```

- Absence as negation:

  If e is in the database, it is true.
  If e is not in the database, it is false.

Representing Operators

Operators (example):

```
Name:            pickup(?x)
Preconditions:   clear(?x), on(?x, ?y)
Add:             holding(?x), clear(?y)
Delete:          on(?x, ?y)
```

Execution: If o is an operator and s is a state, then

```
execute(o, s) =

    if  s => precondition(o)
        then   s = [s U add(o)] \ delete(o)
        else      s
```

A Goal-Stack Planning Algorithm

Input: initial-state I
goals g1, g2, ... gn
operator set O

Output: success or failure

Local state: problem-stack
world-state

1. world-state = I

2. Push achieve(g_1,..., G_n) onto problem-stack.

3. Loop:

   (a) If problem-stack is empty, SUCCEED.

   (b) N = pop(problem-stack)

   (c) If N = achieve(g_1, ..., g_n) then
      i. If N is true in the world-state, then go to 3.
      ii. If N has previously been attempted, then fail.
      iii. Mark N as attempted.
      iv. Push N back on problem-stack.
      v. Order the g_i
      vi. Push achieve(g_i) on problem-stack in order.

A Goal-Stack Planning Algorithm (continued)

(d) Else if N = achieve(g) then

i. If N is true in the world-state, then go to 3.
ii. Choose an operator from o from O
that possibly adds g. If none exists, fail.
iii. Choose a set of variable bindings
that makes o grounded and makes o add g
iv. Push apply(o) onto problem-stack
v. Push achieve(precondition(o))
onto the problem-stack

(e) Else if N = apply(o)

world-state = execute(o, world-state)

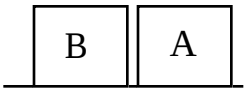
A Demonstration of Goal-Stack Planning

Assume that a robot is given this set of operators:

stack(x, y)	precondition:	clear(y), holding(x)
	add:	armEmpty, on(x, y)
	delete:	clear(y), holding(x)
unstack(x, y)	precondition:	on(x, y), clear(x), armEmpty
	add:	holding(x), clear(y)
	delete:	on(x, y), armEmpty
pickup(x)	precondition:	clear(x), on(x, TABLE), armEmpty
	add:	holding(x)
	delete:	on(x, TABLE), armEmpty
putdown(x)	precondition:	holding(x)
	add:	on(x, TABLE), armEmpty
	delete:	holding(x)

Solve:

Initial state:



Goal state: