CPS331 Lecture: Planning

last revised September 19, 2018

Objectives:

1. To introduce the STRIPS-style planning and the STRIPS assumption
2. To introduce Means-Ends Analysis
3. To note other issues in planning - nonlinear (partial order) planning, hierarchical planning, reactive planning.

Materials:

1. Projectable of Russell and Norvig (2nd ed) figure 11.2
2. Prolog blocks world program - both character graphics frame axiom version and Java graphics STRIPS version
3. Projectable of frame axioms from first version
4. Excerpt from Dennett article in Boden to read (pp 147-148)
5. Handout Exercise 2 on page 96 as class exercise (same as search)
6. Projectable of STRIPS-style operators for the above + solution

I. Introduction

A. Planning is defined in Cawsey as “the problem of finding a sequence of primitive actions to achieve some goal.” Planning crops up in many places.

ASK

1. The use of a GPS or a tool like mapquest or google to plan how to drive from one place to another.

2. Planning an airline trip from one airport to another.

3. Manufacturing scheduling.

4. Planning complex tasks (e.g. the launch of a space shuttle)

5. Robotics

B. Planning is closely related to search, but is considered a distinct area of AI for a number of reasons

1. Goals are often composite - i.e. there is more than one things we want to achieve.

   Example: One text (Russell and Norvig) develops its examples in terms of a scenario where we want to go shopping and purchase milk, an electric drill and bananas (which presumably entails visiting two separate stores). That is, the goal can be described as

   at(home) ^ have(milk) ^ have(drill) ^ have(bananas)

2. While a plan needs to ultimately be <u>executed</u> in a linear order from start to finish, it is often expedient to <u>develop</u> the plan in some other order.

   Example: In the above, once we have gone to the grocery store we can buy both milk and bananas, but we have to buy the drill at the hardware store, but we may not discover that we can put buying milk and bananas together until after we have "thought about" the drill given the order in which the subgoals are listed.

3. Plans are often developed in a context where there may be myriads of possible operations.

   Example: PROJECT Russell/Norvig figure 11.2

4. It is often desirable to think about operations hierarchically

   Example: In the above, the primitive operations I can do are actually things like go to the milk display in the store, pick up a bottle of milk, go to the check out stand, pay for my order ... but it is helpful to first think hierarchically in terms of "buying milk" and later decompose this into more primitive operations, at least one of which can be shared with "buying bananas".

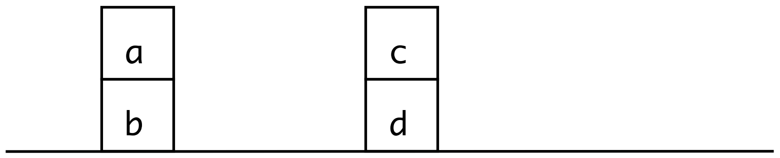
5. Moreover, it is desirable to develop general algorithms that do not depend on problem-specific heuristics.

C. Planning has been a subject of research in AI since the earliest days. The goal has been to develop generalized approaches to planning that can be used with any problem, rather than developing planning approaches to specific problems.

D. As was the case with search, we will develop the basic ideas using a toy problem, in this case based on a <u>microworld</u> called the blocks world. A microworld is very limited domain used as a "test bed" for basic ideas. This world we will use is one in which the only objects are children's blocks, and the operations relate to stacking and unstacking them.

1. We will use a particularly primitive example in which all blocks are cubical and of the same size. Further, we will assume each block has a unique label so we can talk about it.

   Example:



   A possible goal might be "put block a on block d" - which, of course, necessitates moving c out of the way.

2. This problem can be described using predicate calculus as:

```
Initial state:   on(a, b)
                 on(b, table)
                 on(c, d)
                 on(d, table)
                 clear(a)
                 clear(c)

Goal state:      on(a, d)
```
(and probably lots of other things we don't care about)

a) There are two possible operations we can perform: stacking one block on top of another block, and unstacking a block that is currently on top of another block. Traditionally, these operations have been called `puton` and `putontable`. Each starts from some state and produces a new "result" state.

(1) Each of these operators has certain preconditions. For example, we cannot put one block on another unless both are clear to begin with; and we cannot put a block on the table unless it is clear to begin with.

(2) Therefore, a plan for achieving the above goal can be represented by

```
putontable(c)
puton(a, d)
```

(Of course, we figured this out by hand; the goal is to automate this!)

(3) This is not the only possible plan, of course (though it is the simplest)

ASK for other possible plans

b) Demonstrate frame axiom version Prolog program with natural language front end - NOTE: Run from command line, not from PrologJ window, due to use of VT100-style graphics. (Note: the natural language aspects of this will be discussed in a future lecture)

3. A more general form of the blocks world has been used in conjunction with work on natural language. In this case, the blocks often have differing sizes, shapes, and colors, but are not labelled. This allows dialogs like this:

"Pick up the large red block next to the small blue block"
"Put it on the table."

(but that's not our subject now)

4. Critics of AI have criticized the use of microworlds by arguing that techniques developed in a microworld break down when confronted with the complexity of the real world. While this claim has a lot of validity, it remains the case that microworlds like the blocks world are useful for understanding basic concepts.

II. **The Frame Problem**

A. One of the crucial issues in planning is dealing with the impact of operations on the world. In general, any operation changes some things and leaves others unchanged.

1. One writer has illustrated the general problem this way:

   READ Excerpt from Dennett article in Boden (p 147- first two paragraphs on 148)

2. Example: In the blocks world, when we put one block on top of another

   a) The intended effect of the operation is for it to be "on" that block

   b) But certain other things change as well

   ASK

   (1) The block it is put on is no longer clear.

   (2) It is no longer "on" the place (another block or the table) it once was.

   (3) If it was previously on some other block, that block is now clear.

   c) But the "on" and "clear" status of other blocks does not change

B. An early approach to handling this was by using "frame axioms". A frame axiom is a statement to the effect that a specific thing does not change when a particular operation is done.

1. Example: In our blocks world, we might have the axiom:

   If block A is put on block B, then the status of clear(X) does not change for any X different from B

   PROJECT: Frame axioms from frame axiom version of blocks world program

2. A significant problem with this approach is that the number of frame axioms needed can be very large. In particular, we need one frame axiom for each (operation, sort of description) pair - e.g. we need one for (puton, on), another for (puton, clear), another for (putontable, on) and another for (putontable, clear). If we had 10 operations and 10 descriptions we would need 100 axioms!

C. A better approach is to expand our operator specifications to be explicit about what <u>does</u> change, rather than using frame axioms to specify what does not change. In principle, this should be simpler, since a given operator normally only affects one or two things.

1. This approach was pioneered in a planning system called STRIPS (Stanford Research Institute Problem Solver). Though this particular system has since been superseded (and Stanford Research Institute now goes by the name SRI), the name STRIPS has stuck as a description of this particular approach.

   In particular, the term "STRIPS assumption" is used to describe the assumption most planning systems make that an operator only changes the things that it is specifically stated as changing; so anything that is not stated remains unchanged.

2. The basic idea is this:

a) We maintain a list of things which are true at the current point in the planning process.

b) When describing an operator, we specify not only what it makes true, but also what it makes no longer true - both things that become true and things that cease to be true. This is used to add and delete things from the description of the state.

Thus, our blocks world operators might be specified in STRIPS-style this way:

(1)putontable(Block)

Precondition:	on(Block,X), clear(Block),
Add:	clear(X), on(Block, table),
Delete	on(Block, X)

(Note how the first precondition simply serves to facilitate additions and deletions)

(2)puton(Block1, Block2)

Preconditions:	on(Block1, X) where X is table, clear(Block1), clear(Block2),
Add:	on(Block1, Block2)
Delete	on(Block1, X), clear(Block2)

or

Preconditions:	on(Block1, X) where X is not table, clear(Block1), clear(Block2),
Add:	clear(X), on(Block1, Block2)
Delete	on(Block1, X), clear(Block2)

(Note how two versions are needed - one for the case where the block being moves starts out on the table, and one for the case where the block being moved starts out on some other block, because clear(table) is never included in the state since there it is assumed that there is always some clear spot on the table.)

3. Example: A STRIPS-style solution to our blocks world problem

a) Initial and goal states

```
Current:    on(a, b)
            on(b, table)
            on(c, d)
            on(d, table)
            clear(a)
            clear(c)

Goal:       on(a, d)
```

b) State after putontable(c):

Current: on(a, b)
on(b, table)
~~on(c, d)~~
on(d, table)
clear(a)
clear(c)
on(c, table)
clear(d)

Goal: on(a, d)

c) State after puton(a, d):

Current: ~~on(a, b)~~
on(b, table)
~~on(c, d)~~
on(d, table)
clear(a)
clear(c)
on(c, table)
~~clear(d)~~
on(a, d)
clear(b)

Goal: on(a, d)

d) Since our goal `on(a, d)` is now part of the state, we're done. (We don't care about the other aspects of the description)

III. Means-Ends Analysis

A. So far, we have begged the question of how do we decide what operator to apply (which is, after all, the interesting question!)

B. STRIPS and most other planners make use of a strategy called <u>means-ends-analysis.</u>, which is a form of backward-chaining.

The STRIPS approach is this: to repeat the following process until the current state matches the goal:

1. Look at the goal (which is typically composite) and select some component that is unsatisfied in the current state

2. Find an operator which (by appropriate instantiation of variables) has this component in its "Add" list.

   a) Add any preconditions of the operator that are not satisfied in the current state to the goal.

   b) Add the things that are on the "Add" list for the operator to the current state, and delete the things on the "Delete" list for the operator from the current state if they are there, or record that they should be deleted when they become true later.

   c) Mark the things that are on the "Add" list for the operator which appear in the goal as satisfied.

C. Example: blocks world problem

1. Initial and goal states:

   Current: `on(a, b)`
   `on(b, table)`
   `on(c, d)`
   `on(d, table)`
   `clear(a)`
   `clear(c)`

   Goal: `on(a, d)`

2. Fulfill by `on(a, d)` using the puton operator with B1 instantiated to a, B2 to d, and B3 to b

   First precondition satisfied by `on(a, b)`
   Second precondition satisfied by `clear(a)`
   Add third precondition to goal (`clear(d)`)
   Add: `on(a, d)`, `clear(b)`
   Delete: `on(a, b)`
   Can't delete `clear(d)` since not there yet, but note that it should be deleted later

   New conditions

   Current: `on(a, d)`
   `on(b, table)`
   `on(c, d)`
   `on(d, table)`
   `clear(a)`
   `clear(b)`
   `clear(c)`

   Goal: `clear(d)`

3. Fulfill `clear(d)` by using the putontable operator with B1 instantiated to c and B2 to d

   First precondition satisfied by `clear(c)`
   Second precondition satisfied by `on(c, d)`
   Add: `on(c, table) [ clear(d) deleted ]`
   Delete: `on(c, d)`

   New conditions

```
Current:  on(a, d)
          on(b, table)
          on(c, table)
          on(d, table)
          clear(a)
          clear(b)
          clear(c)

Goal:     (empty)
```

D. Demo: STRIPS Blocks world planner program. (This one doesn't have a natural language front end, but will be used for a homework problem)

E. PROJECT operators.pro

F. Means-Ends Analysis is an example of what is sometimes called "weak method" problem solving.

1. It was first introduced in 1963 as part of an AI program known as General Problem Solver (GPS). It was the claim of the program's authors that this approach could be used to solve any problem, but it has since been recognized that this was a gross overstatement.

2. "Weak method" refers to something quite distinct from "weak AI". It is used in contrast to "strong method" - which is a problem solving method that uses knowledge about the specific problem to control search. (For example, when we used heuristics for the eight-puzzle we were using a strong method.)

G. Class Exercise - redo exercise 2 on page 96 of book (previously done as a search problem) using STRIPS

HANDOUT

1. Develop STRIPS style operators, then PROJECT

2. Solve using forward chaining (not means-ends analysis) - with the following restrictions:

   a) Never select an operator whose preconditions are not true

   b) Never select an operator which produces a cycle.

   c) If multiple operators are available, select first.

3. PROJECT solution

IV. Some Other Issues

A. Planning is an active research field, and there are many important issues that arise. We are just going to mention a few. Modern planning systems are built on the basic ideas of STRIPS, with significant improvements.

1. The use of the "STRIPS assumption" to keep track of the changes an operation makes on the world.

   a) Each operator explicitly lists the things in the world that it changes (the Add and Delete lists of STRIPS).

   b) If something is not listed, it is assumed not to change.

2. The use of Means-Ends Analysis, perhaps augmented with heuristics.

B. We noted at the outset that goals are often composite. Sometimes, the goals are more or less independent of one another, in the sense that an overall plan can be constructed by combining pieces for each part. However, this is not always the case.

1. Sometimes, the relative order in which the goals are tackled is critical. Consider the following examples from the simple planner we just looked at:

   on(black, orange), on(yellow, black) DEMO

   The reverse order

   on(yellow, black), on(black, orange) Reset, then DEMO

   The simple planner messes up on the second case because it takes a very simplistic approach to planning composite goals: plan the first, then plan the second, then put the pieces together- which breaks if the plan for the second undoes what the first did, as in this case.

   In this particular case, we can find a correct solution if we order the subgoals correctly.

2. Sometimes, though, we need to overlap the achievement of the goals.

   a) Recall the example we looked at at the start of our discussion - the composite goal at(home) ^ have(milk) ^ have(drill) ^ have(bananas). If both milk and bananas come from the same store, then it makes sense to have the plans overlap - i.e. for the sake of efficiency we want:

      go to the grocery store
      go to the dairy department and pick up milk
      go to the produce department and pick up bananas
      go to the checkout counter

      rather than

go to the grocery store
go to the dairy department and pick up milk
go to the checkout counter
...
go to the hardware store
...
go to the grocery store
go to the produce department and pick up bananas
go to the checkout counter

b) In the above example, the overlap was needed as a matter of efficiency - the second plan would still work. There are times, though, when overlap is essential. Consider the following blocks world problem

on(yellow, orange), on(pink, yellow). DEMO

(1)Here, reversing the order doesn't help. DEMO

(2)There is no way to formulate a plan for this goal in terms of two independent pieces - the only possible plan requires overlap

(3)This problem has come to be known as the Sussman anomaly (after Gerald Sussman, who first discovered it)

3. Problems like these are addressed by a strategy known as non-linear or partial order planning. The key ideas are these:

a) We search through a space of partial plans for achieving our goals.

(1)Each partial plan represents an incomplete piece of our overall plan; eventually, we will combine them into a complete plan.

(a) Each partial plan may have as-yet unsatisfied preconditions.

(b) Initially, there are just two partial plans:

Start - which represents the initial conditions
Finish - which has all the components of the goal as preconditions

(c) Means-ends analysis is used to build up the partial plans by creating new partial plans to satisfy the preconditions of existing ones.

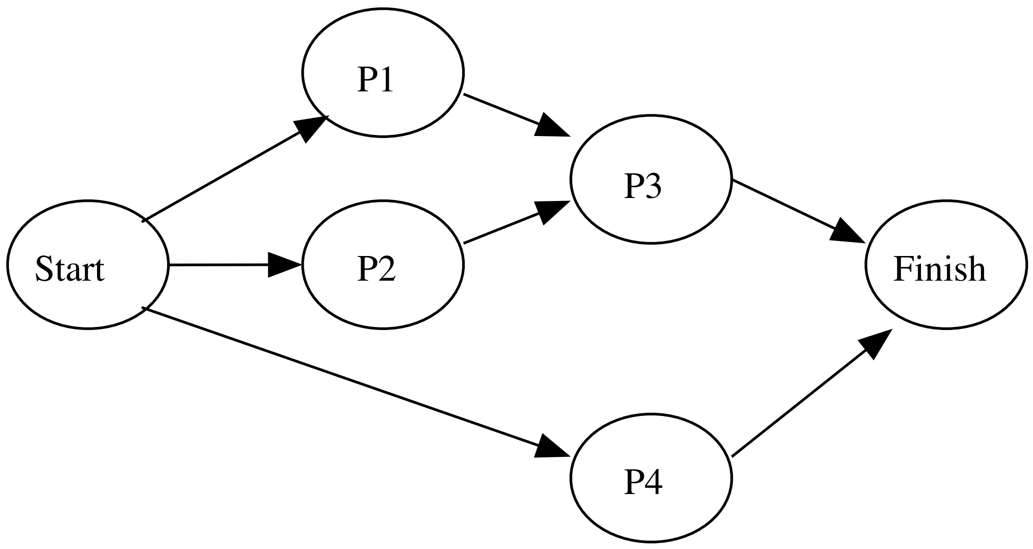
(2) Partial planning uses the principle of deferred commitment - which says that, in formulating a plan, we defer choices about specifics as long as possible

Example; if we were formulating a partial plan for buying milk, we might defer the decision as to exactly which grocery store to go to.

(3) We maintain information about relationships between partial plans - eg.

(a) Ordering constraints, that specify that partial plan must be done before another

i) We call the technique "partial order" planning because, in general, these constraints don't specify a total ordering between plans. For example, at some point we may have ordering constraints that specify the following



This says that both P3 and P4 must be done before we are finished, but they can be done in either order. Both P1 and P2 must be done before P3 can be done, though they can be done in either order

ii) Ordering constraints arise because sometimes one partial plan will undo the work done by another. If some plan achieves a component of our final goal, it must be done after any plan that would undo its result.

(b) Causal links, that record information about partial plans needed to accomplish preconditions of other plan. These become necessary because some partial plan may undo a precondition of some other plan. In this case, it must either be done before or after both plans, but cannot be done in between them.

(This is known as goal-protection)

(4) The ultimate goal is an overall plan in which the partial plans have been merged into a single plan that satisfies all preconditions

C. Another component of an overall strategy may be hierarchical planning. Instead of operators being restricted to just being primitive actions, many operators are composite - i.e. they are templates for collections of primitive actions

Example: we may have an operator buy(milk) which stands for the sequence of operations go to store, go to dairy section, pick up milk, go to checkout counter, checkout. This is a general template which must ultimately be made more specific by specifying the store; but (following the principle of deferred commitment) may be used in a partial plan with the store being specified later.

D. Sometimes, planning is used in a context where the environment is changing, or it is not certain that a given operator will necessarily accomplish its goal. A reactive planner continually updates the plan as it is being carried out in light of actual conditions in the environment.

Appendix: STRIPS-style solution to Farmer, Rabbit, Dog, Lettuce

Represent state as FP (farmer position) etc - with values E, W

<u>Initial state:</u>

FP = E
RP = E
DP = E
LP = E

<u>(Goal state is all four = W)</u>

<u>Rules:</u>

Row with Dog

Preconditions:	FP = X, DP = X, RP != LP
Delete:	FP = X, DP = X
Add:	FP = opposite(X), DP = opposite(X)

Row with Rabbit:

Preconditions:	FP = X, RP = X
Delete:	FP = X, RP = X
Add:	FP = opposite(X), RP = opposite(X)

Row with Lettuce:

Preconditions:	FP = X, LP = X, RP != DP
Delete:	FP = X, LP = X
Add:	FP = opposite(X), LP = opposite(X)

Row Alone:

Preconditions:	FP = X, RP != DP, RP != LP
Delete:	FP = X
Add	FP = opposite(X)

Only rule whose preconditions are satisfied is Row with Rabbit

FP = ~~E~~ W
RP = ~~E~~ W
DP = E
LP = E

Only rule whose preconditions are satisfied that does not lead to a cycle is Row Alone

FP = ~~E~~ ~~W~~ E
RP = ~~E~~ W
DP = E
LP = E

Only rules whose preconditions are satisfied that do not lead to a cycle are Row with Dog, Row with Lettuce. Choose Row with Dog

FP = ~~E~~ ~~W~~ ~~E~~ W
RP = ~~E~~ W
DP = ~~E~~ W
LP = E

Only rule whose preconditions are satisfied that does not lead to a cycle is Row with Rabbit

FP = ~~E~~ ~~W~~ ~~E~~ ~~W~~ E
RP = ~~E~~ ~~W~~ E
DP = ~~E~~ W
LP = E

Only rule whose preconditions are satisfied that does not lead to a cycle is Row with Lettuce

FP = ~~E~~ ~~W~~ ~~E~~ ~~W~~ ~~E~~ W
RP = ~~E~~ ~~W~~ E
DP = ~~E~~ W
LP = ~~E~~ W

Two rules satisfy the preconditions and do not lead to a cycle: Row Alone, Row with Dog
If we choose Row Alone

FP = ~~E~~ ~~W~~ ~~E~~ ~~W~~ ~~E~~ ~~W~~ E
RP = ~~E~~ ~~W~~ E
DP = ~~E~~ W
LP = ~~E~~ W

Now the only rule whose preconditions are satisfied that does not lead to a cycle is Row with Rabbit

FP = ~~E~~ ~~W~~ ~~E~~ ~~W~~ ~~E~~ ~~W~~ E
RP = ~~E~~ ~~W~~ E
DP = ~~E~~ W
LP = ~~E~~ W

We're at Goal!