Planning (Ch. 10)
Announcements

Test Wednesday:  
Covers Ch 5-7 and 17.5

HW3 has been graded  
Minor correction in solution to Problem 3(1)

Writing 2 has been graded  
Resubmission due Monday, April 18

Last day to talk to Gabriel, April 15
Planning

Planning is doing a sequence of actions to reach a goal state.

This differs from search in that there are often objectives along the way that must be done.

In search reaching the goal is everything, while planning can be thought of deciding what sequence to reach multiple goals.
Search

Search: How to get from point A to point B quickly?
Planning

Planning: multiple tasks/subtasks need to be done, what order to do them in?
The book uses Planning Domain Definition Language (PDDL) to represent states/actions.

PDDL is very similar to first order logic in terms of notation (states are now similar to what our knowledge base was).

The large difference is that we need to define actions to move between states.
Planning: assumptions

We make the same 3 assumptions as FO logic:
1. Objects are unique (i.e. $\neg Bob = Jack$)

2. All un-said sentences are false
   
   Thus if I only say: $\text{Brother}(James, Bob)$
   
   I also imply: $\neg \text{Brother}(James, Jack)$

3. Only objects I have specified exists
   
   (i.e. There is no $Davis$ object unless I explicitly use it at some point)
Planning: state

A state is all of the facts ANDed together in FO logic, but are not allowed to have:
1. Variables (otherwise it would not be specific)
2. Functions (just replace them with objects)
3. Negations (as we assume everything not mentioned is false)

State = B\text{Knight}(D, 8) \land B\text{Pawn}(C, 7) \\
\land B\text{King}(D, 7) \land W\text{Pawn}(B, 6) \\
\land W\text{Knight}(C, 6) \land W\text{Rook}(E, 6) \land ... \\
\land \text{Turn}(\text{Black})
Actions have three parts:
1. Name (similar to a function call)
2. Precondition (requirements to use action)
3. Effect (unmentioned states do not change)

For example:

Action( MoveB Knight1(x, y),
Precondition: B Knight(x, y) ∧ Turn(Black),
Effect: ¬B Knight(x, y) ∧ B Knight(x + 2, y − 1)
∧¬Turn(Black) ∧ Turn(White)

remove black's turn
Planning: actions

State = \( BKnight(D, 8) \land BPawn(C, 7) \land BKing(D, 7) \land WPawn(B, 6) \land WKnight(C, 6) \land WRook(E, 6) \land \ldots \land Turn(Black) \)

Apply: MoveBK\(Knight(D, 8)\)

State = \( BKnight(F, 7) \land BPawn(C, 7) \land BKing(D, 7) \land WPawn(B, 6) \land WKnight(C, 6) \land WRook(E, 6) \land \ldots \land Turn(White) \)
Planning: example

Let's look at a grocery store example:
Objects = store locations and food items
Goal = At(Checkout) \land Cart(Milk) \land Cart(Apples) \\
\land Cart(Eggs) \land Cart(ToiletPaper) \land Cart(Bananas) \\
\land Cart(Bread) \land \neg Cart(Candy)

Aisle 1 = Milk, Eggs
Aisle 2 = Apples, Bananas
Aisle 3 = Bread, Candy, ToiletPaper
### Planning: example

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GoTo(x, y)</code></td>
<td><code>At(x)</code></td>
<td><code>¬At(x) ∧ At(y)</code></td>
</tr>
<tr>
<td><code>AddApples()</code></td>
<td><code>At(Aisle2)</code></td>
<td><code>Cart(Apples)</code></td>
</tr>
<tr>
<td><code>AddMilk()</code></td>
<td><code>At(Aisle1)</code></td>
<td><code>Cart(Milk)</code></td>
</tr>
<tr>
<td><code>AddBananas()</code></td>
<td><code>At(Aisle2)</code></td>
<td><code>Cart(Bananas)</code></td>
</tr>
<tr>
<td><code>AddEggs()</code></td>
<td><code>At(Aisle1)</code></td>
<td><code>Cart(Eggs)</code></td>
</tr>
<tr>
<td><code>AddBread()</code></td>
<td><code>At(Aisle3)</code></td>
<td><code>Cart(Bread)</code></td>
</tr>
<tr>
<td><code>AddCandy()</code></td>
<td><code>At(Aisle3)</code></td>
<td><code>Cart(Candy)</code></td>
</tr>
<tr>
<td><code>AddToiletPaper()</code></td>
<td><code>At(Aisle3)</code></td>
<td><code>Cart(ToiletPaper)</code></td>
</tr>
</tbody>
</table>
Planning: example

Initial state = At(Door)
A possible solution:
1. GoTo(Aisle1)       2. Add(Milk)
3. Add(Eggs)          4. GoTo(Aisle2)
5. Add(Apples)        5. GoTo(Aisle3)
6. Add(Bread)         7. Add(ToiletPaper)
8. GoTo(Aisle2)       8. Add(Bananas)
9. GoTo(Checkout)

Not most efficient, but goal reached
Planning: actions

If we treat the current state like a knowledge base and actions with $\forall$s for every variable...

“state $\models$ Precondition(A)” means action A's preconditions are met from the current state

Thus if each action uses $v$ variables, each with $k$ possible values, there are $O(k^v)$ actions (we can ignore actions that do not change the current state in some cases)
Since our planning is similar to FO logic, it is unsurprisingly semi-decidable as well.

Thus, in general you will be able to find a solution if it exists, but possibly be unable to tell if a solution does not exist.

If there are no functions or we know the goal can be found in a finite number of steps, then it is decidable.
Planning: difficulty

PlanSAT tells whether a solution exists or not, but takes PSPACE to tell

If negative preconditions are not allowed, we find a solution in P, and optimal in NP-hard
Planning: algorithms

Again similar to FO logic, there are two basic algorithms you can use to try and plan:

1. Forward search - similar to BFS and check all states you can find in 1 action, then 2 actions, then 3... until you find the goal state

2. Backward search - start at goal and try to work backwards to initial state
Forward search

Forward search is a brute force search that finds all possible states you can end up in.

Each action is tested on each state currently known and is repeated until the goal is found.

This can be quite costly, as actions that do not lead to the goal could be repeatedly explored (we will see a way to improve this).
Forward search

GoTo(Door)

At(Door)
At(Aisle1)
At(Aisle2)
At(Aisle3)
At(Checkout)

AddMilk()
At(Aisle1)
Cart(Milk)

can ignore

GoTo(Checkout)

At(Checkout)

...
Forward search

You try it!

Initial: \( \text{At(UPSD)} \land \text{Package(UPSD, P1)} \land \text{Package(UPSD, P2)} \)

Goal: \( \text{Package(P1, H1)} \land \text{Package(P2, H2)} \)

Action( \( \text{GoTo}(x, y) \),
   Precondition: \( \text{At}(x) \),
   Effect: \( \neg \text{At}(x) \land \text{At}(y) \) )

Action( \( \text{Load}(m, x, y) \),
   Precondition: \( \text{At}(y) \land \text{Package}(y, x) \),
   Effect: \( \neg \text{Package}(y, x) \land \text{Package}(m, x) \) )

Action( \( \text{Deliver}(m, x, y) \),
   Precondition: \( \text{At}(y) \),
   Effect: \( \neg \text{Package}(m, x) \land \text{Package}(y, x) \) )
Backward search

Backward search is also similar to FO's backward search

Start at goal and do actions in reverse (swap effect and precondition), except substitute:

Example:

Goal = At(Home)
Initial = At(Class)

Unify: \{x/\text{Home}, y/\text{Class}\} ... done
Backward search

We also need some easy definition of what is “away from the goal” (some problems this is hard, for example n-queens)

Want to keep unification general if possible

The branching factor is much smaller due to not considering irrelevant actions, but it is more difficult to come up with a heuristic (as this considers sets of states)
Backward search

We also need some easy definition of what is “away from the goal”

Some problems, such as n-queens do not have a great “away from goal” definition

While stepping backwards, you need to find any “swapped” preconditions that are applicable and find a valid substitution
Heuristics for planning

In “search” we had no generalize-able heuristics as each problem could be different.

Heuristics in planning are also the same, we want an admissible one found from relaxing the problem and solving that optimally.

There are two ways to always do this:
1. Add more actions
2. Reduce number of states
Heuristics: add actions

Multiple ways to add actions (to goal faster):

1. Ignore preconditions completely - also ignore any effects not related to goals

   This becomes set-covering problem, which is NP-hard but has P approximations

2. Ignore any deletions in effects (i.e. anything with \(\neg\)), also NP-hard but P approximation
Heuristics: group states

Group similar states together into “super states” and solve the problem within “super states” separately (divide & conquer)

A admissible but bad heuristic would be the maximum of all “super states” individual solutions (but this is often poor)

A possibly non-admissible would be the sum of all “super states” (need independence)