Artificial Intelligence Programming
Planning

Chris Brooks

Department of Computer Science
University of San Francisco
Planning is the task of selecting a sequence of actions that will achieve a goal.

You’ve already done simple planning in the fox-and-chickens problem.

Challenges:
- Representing actions
- Dealing with large search spaces
- Taking advantage of problem decomposition
Planning combines search and knowledge representation.

Uses a logical formalism to describe states and actions.

Uses heuristic search to select actions to take.

Takes advantage of the fact that planning problems are often decomposable.

For example, a “Travel to Hawaii” plan can be broken into:

- Buying tickets stage
- Packing stage
- Getting to the airport stage
- Flying on the plane stage

These problems are independent - the order in which I pack things shouldn’t affect how I get to the airport.
Planning operators look very much like rules and facts.

States are conjunctions of positive literals and relations between literals.

No variables or functions allowed in states.

- \( \text{At}(Brooks, \text{Airport}) \)
- \( \text{In}(\text{clothing}, \text{suitcase}) \land \text{In}(\text{toothbrush}, \text{suitcase}) \land \text{Has}(Brooks, \text{ticket}) \)
A goal is a partially specified state.

- A conjunction of positive literals, with not every state feature specified.
  - e.g. \( \text{In}(Brooks, \text{Hawaii}) \land \text{Has}(Brooks, \text{suitcase}) \land \text{In}(\text{clothing}, \text{suitcase}) \)

Notice that this is a much more flexible state representation than we saw with fox-and-chickens.

No need to specify locations of all objects.
An action is specified in terms of *preconditions* and *effects*.

Preconditions indicate what must hold in order for the action to be taken.

Effects indicate how the action changes the world.

Both may contain variables.

- **Action**: BuyTicket(person, price, ticket)
  
  *Preconditions*: Costs(ticket, price) ∧ HasMoney(person, price)
  
  *Effects*: Has(ticket, person) ∧ ¬Has(person, price)

- **Action**: DriveToAirport(person, loc)
  
  *Preconditions*: At(person, loc)
  
  *Effects*: ¬At(person, loc) ∧ At(person, Airport)

This representation language is known as STRIPS.
15-5: STRIPS assumptions

- Preconditions are positive conjunctions
- Every literal not mentioned remains unchanged.
- Goals can only contain ground literals
  - Can’t have a goal like $∃locAt(Brooks, loc) \land Warm(loc)$
- Goals are conjunctions
- These assumptions limit the sorts of problems that can be solved, but make planning algorithms simpler and more efficient.
- We’ll look at extensions later on.
15-6: Formulating a Planning Problem

Specify an initial state.

\[ \text{Initial} : \text{On(FlatTire, Axle)} \land \text{In(SpareTire, Trunk)} \]

Specify a goal state.

\[ \text{Goal} : \text{On(SpareTire, Axle)} \land \text{In(FlatTire, Trunk)} \]
15-7: Formulating a Planning Problem

Specify actions

- **RemoveTire**(tire)
  - **Preconditions**: On(tire, Axle)
  - **Effect**: Clear(Axle) ∧ ¬On(tire, Axle) ∧ On(tire, Ground)

- **PutOnTire**(tire)
  - **Preconditions**: On(tire, Ground) ∧ Clear(Axle)
  - **Effect**: On(tire, Axle) ∧ ¬On(tire, Ground) ∧ ¬Clear(Axle)

- **TakeFromTrunk**(tire)
  - **Preconditions**: In(tire, Trunk)
  - **Effect**: On(tire, Ground) ∧ ¬In(tire, Trunk)

- **PutInTrunk**(tire)
  - **Preconditions**: On(tire, Ground)
  - **Effect**: In(tire, Trunk) ∧ ¬On(tire, Ground)

Notice that this looks like a search problem.
Now that we have a formalization for the problem, we can try to apply our standard search techniques.

We can search forward from the initial state to the goal state.

At each state, consider what actions are possible.

Each potential action generates a new state.

This is called progression planning.

- Very similar to forward chaining.

Problem: Lots of irrelevant actions are considered.

Doesn’t scale to complex domains.
We can also work backward from the goal to the initial state.

This is called regression planning.

Look for actions that achieve one or more goal criteria.

Algorithm is similar to backward chaining.

Still doesn’t scale effectively.

Doesn’t allow you to take advantage of problem decomposition.
Partial-order Planning

One problem with progression and regression planning is that they search for linear sequences of actions.

- Often, subgoals can be solved in more than one order.
- It really doesn’t matter whether I buy my ticket before I pack.

Partial-order planning solves subgoals independently (as much as possible) and then combines subplans.

No need to select steps in chronological order.

Example: in planning my trip, I know I’ll need to buy a ticket, pack, and get to the airport. I’ll figure out how to do each of those, then decide on an order later.

This is called a least commitment strategy.
Partial-order planning searches in the space of *partially-completed plans*.

This is different from A*, BFS, regression planning, etc, which search in a space of *states*

We begin with the empty plan

Our successor function generates new, more refined plans.
A plan has four components:

△ A set of actions that make up the steps of the plan.
△ A set of ordering constraints that indicate a sequence on actions.
   - For example, \( \text{PackSuitcase} \prec \text{GoToAirport} \)
△ A set of causal links that indicate one action achieves the precondition of another.
   - For example, \( \text{BuyTicket} \rightarrow^{\text{HasTicket}} \text{TakeFlight} \)
△ A set of open preconditions. These are preconditions not achieved by any action in the current plan.
A consistent plan is one in which there are no cycles in the ordering constraints.

A consistent plan with no open preconditions is a solution.

We can then linearize this plan to get a sequence of actions.

Any linearization of a partial-order plan will reach the goal state.

Some linearizations might be more efficient than others.
6 Initial plan contains $Start, Finish, Start \prec Finish$

6 While (no solution)
   △ Select an open precondition $p$ on an action $B$
   △ Find each action $A$ that satisfies that precondition; generate a new plan for each action.
   △ for each of these plans:
     - Add ordering constraints $A \prec B$, plus causal links $A \rightarrow^p B$
     - Resolve any conflicts between causal links. If no consistent plan exists, discard.
6 Problem Description

- Initial: On(FlatTire, Axle) ∧ In(SpareTire, Trunk)
- Goal: On(SpareTire, Axle) ∧ In(FlatTire, Trunk)
- RemoveTire(tire)
  Preconditions: On(tire, Axle)
  Effect: Clear(Axle) ∧ On(tire, Ground)
- PutOnTire(tire)
  Preconditions: On(tire, Ground) ∧ Clear(Axle)
  Effect: On(tire, Axle) ∧ ¬On(tire, Ground) ∧ ¬Clear(Axle)
- TakeFromTrunk(tire)
  Preconditions: In(tire, Trunk)
  Effect: On(tire, Ground) ∧ ¬In(tire, Trunk)
- PutInTrunk(tire)
  Preconditions: On(tire, Ground)
  Effect: In(tire, Trunk) ∧ ¬On(tire, Ground)
Open preconditions:
\( \text{On} (\text{SpareTire}, \text{Axle}) \wedge \text{In} (\text{FlatTire}, \text{Trunk}) \)

Pick \( \text{On} (\text{SpareTire}, \text{Axle}) \), select \( \text{PutOnTire} \).
\( \triangleright \) Add causal links, ordering constraints between \( \text{PutOnTire} \) and \( \text{Finish} \).

New preconditions:
\( \text{In} (\text{FlatTire}, \text{Trunk}), \text{On} (\text{SpareTire}, \text{Ground}), \text{Clear} (\text{Axle}) \)

Pick \( \text{In} (\text{FlatTire}, \text{Trunk}) \), select \( \text{PutInTrunk} \).
\( \triangleright \) Add causal links, ordering constraints between \( \text{PutInTrunk} \) and \( \text{Finish} \).

New Preconditions:
\( \text{On} (\text{SpareTire}, \text{Ground}), \text{Clear} (\text{Axle}), \text{On} (\text{FlatTire}, \text{Ground}) \)
Pick \( \text{On}(\text{SpareTire}, \text{Ground}) \), select \( \text{TakeFromTrunk} \)

- Add ordering constraints and causal links between \( \text{TakeFromTrunk} \) and \( \text{PutOnTire} \), also between \( \text{Start} \) and \( \text{TakeFromTrunk} \).

New Preconditions: \( \text{Clear}(\text{Axle}), \text{On}(\text{FlatTire}, \text{Ground}) \)

Select \( \text{RemoveTire} \)

- Add ordering constraints and causal links between \( \text{RemoveTire} \) and \( \text{PutInTrunk} \)

We have a solution.
The trickiest part of POP involves choosing a precondition to try to satisfy.

It can be hard to estimate how “close” a plan is to a solution.

A common heuristic:
  - Most-constrained variable - select the open precondition that can be satisfied in the fewest number of ways.
  - Intuition: Locate “bottlenecks” early on.
Basic POP with STRIPS has two weaknesses.
  △ Expressivity of STRIPS.
  △ Lack of abstraction.

There have been many solutions proposed to deal with these problems.
Action Description Language (ADL) is an extension of STRIPS.

- Allows negative literals in preconditions, disjunctive effects, existentially quantified variables in goals, typed variables.

Challenges with these languages:

- Ramifications: describing all the effects of an action. When a plane goes from A to B, so does its contents.
- Quantifying and reasoning about time. (How long do actions take? How to say: “I have to be at home at 2pm, and at the airport at 3pm”?)
- Frame problem: Specifying that things don’t change over time.
Many problems can be hierarchically decomposed.

We often have prebuilt plans for solving simple problems
  ▷ Load samples, moveTo(x,y), upload data, etc.

By decomposing a problem into simpler problems, we can take advantage of decomposition.

Key: Unlike STRIPS, we build *abstract actions* that decompose into combinations of primitive actions.
Our abstract plan $GoToHawaii$ decomposes into partially-ordered plans for $Pack$, $BuyTickets$, $GoToAirport$.

- These plans may have ordering constraints or causal links between them.

Each of these plans can be decomposed into one or more partially-ordered plans.

We continue this process until we reach a point where we are able to solve the problem with primitive operators.

The primitive solution is then cached and stored in a plan library.

By combining pieces of previously solved plans, we can avoid search.
Planning is a common technique for agents that must choose and take action in the real world.

- Used in factory operations, NASA rovers, autopilots, robot assistants, beer factories.

- Takes advantage of problem structure to search in plan space rather than state space.

- Only consistent plans are considered.

- Only productive actions are considered.

- Can scale to large real-world systems