Planning

18 December 2020 17:12

1. Knowledge bases
2. Reasoning using knowledge and inference
3. Search vs. Planning
4. Partial-order Planning
5. Conditional Planning
6. Monitoring and Replanning

Search Vs. Planning

In most situations, the branch factor of the search problem is so large that search becomes virtually useless.

General search problems..

1. In search, we must specify an initial state, operators, and optionally a heuristic function
2. Branching factor may be huge depending on how we specify these operators
3. Path length may be very long, and thus there are too many states to consider
4. The agent is forced to construct a full sequence of actions and must decide what to do in initial state first.

Difficulties with heuristics...

1. Heuristics can only choose which state is closer to a goal, but cannot eliminate actions from consideration
2. Evaluation function ranks these guesses, but must still consider them all
3. We need to work on the appropriate part of the sequence.

The main problem with basic problem solving agents is that they consider actions in sequence, starting from the initial state. Until the agent has worked out HOW to obtain the items we're looking for, it cannot really decide where to go. The agent therefore needs a more flexible way of structuring its deliberations so that it can in a non-linear fashion.

3 Key Ideas Behind Planning

The **first key idea** in planning is that we 'open up' the representation of states, goals and actions.

1. Planning algorithms use descriptions in some formal language - usually first-order logic
2. States and goals are represented by sets of sentences
3. Actions are represented by logical descriptions of preconditions and effects

This allows the planner to make direct connections between states and actions.
e.g. if the agent knows that the goal is a conjunction that includes have(milk), and it knows that buy(x) achieves have(x) then the agent knows that it is worthwhile to consider a plan that includes buy(milk). It need not consider other irrelevant actions such as buy(orange)

The **second key idea** is that the planner is free to add actions to the plan wherever they are needed, rather than in an incremental sequence starting at the initial state. For example, the agent may decide it needs to buy(milk) even before it has decided how to do such a thing.

There is no necessary *connection between the order of planning and the order of execution*.

The **third key idea** is that most parts of the world are independent of most other parts, and thus it makes it feasible to take a conjunctive goal and solve it using a divide-and-conquer strategy. A subplan involving going to the supermarket can be used to achieve the first two conjuncts and another subplan involving going to the hardware store can be used to achieve the third. The supermarket subplan can be further divided into a milk subplan and a bananas subplan. We can then put all the subplans together to solve the whole problem.

Planning Systems

Open up action, state and goal representations to allow selection - represent in first-order logic
- States and goals = sets of sentences
- actions = description of preconditions and effects

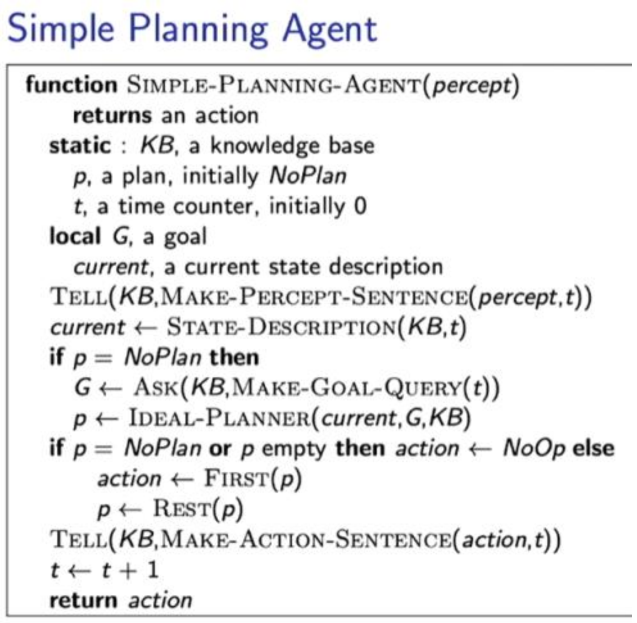
Planning systems allow a planner to make direct connections between states and actions

- We can divide-and-conquer by subgoaling
- Planner can consider several smaller easier problems, and then combine solutions
- Works because little interaction between subplans, otherwise cost of combining solution outweighs the gain e.g. no help for 8 puzzle to consider each tile separately
- Relax requirement for sequential construction of solutions
- Allows planner to add actions where needed, so can make obvious or important decisions first to reduce branching factor

There is no connection between the order of planning and execution. We can do this because of logic - At(Supermarket) represents a class of states, but search requires a complete state description, and so we could not do this.

In the real world, planning tends to do better than search

SPA Algorithm
- Update KB
- if not already executing a plan, generate a goal and construct a plan to achieve it
- Agent must be able to cope if the goal is infeasible or achieved (set action to NoOp)
- Once agent has a plan, it will execute to completion
- Minimal interaction with environment: perceive to determine initial state but then just execute plan - no relevance checks



Simple Planning Agent

```
function SIMPLE-PLANNING-AGENT(percept)
    returns an action
  static : KB, a knowledge base
    p, a plan, initially NoPlan
    t, a time counter, initially 0
  local G, a goal
    current, a current state description
  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  if p = NoPlan then
    G ← ASK(KB, MAKE-GOAL-QUERY(t))
    p ← IDEAL-PLANNER(current, G, KB)
  if p = NoPlan or p empty then action ← NoOp else
      action ← FIRST(p)
      p ← REST(p)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Situation Calculus

- A way of describing change in first-order logic
- World viewed as a sequence of situations, snapshots of the state of the world
- Situations generated from previous situations by actions
- **Fluent**: Functions and predicates that change with time given a situation argument
- Those that do not change are called **eternal** or **atemporal**
- Change represented by function Result(action, situation) which denotes the result of performing action in situation

- *Possibility axioms* — Describes when it is possible to execute an action (Precondition $\implies$ Poss(a,s)) e.g. $At(Agent, x, s) \land Adjacent(x, y) \implies Poss(Go(x, y), s)$
- *Effect axioms* — changes due to action (Poss(a,s) $\implies$ changes), e.g. $Poss(Go(x, y), s) \implies At(Agent, y, Result(Go(x, y), s))$

Planning In Situation Calculus

- Planning can be seen as a logical inference problem using situation calculus
- Logical sentences to describe initial state, goal and operators
- Initial state - sentence about the situation - At home, no milk, no bananas etc
- Goal state - logical query for suitable situations - At home and have milk and have bananas etc

- *Given*
  - Initial State: $At(Agent, [1, 1], S_0) \land At(G, [1, 2], S_0) \land Gold(G)$
  - Possibility Axioms
    - ⋆ $At(Agent, x, s) \land Adjacent(x, y) \implies Poss(Go(x, y), s)$
    - ⋆ $Gold(g) \land At(agent, x, s) \land At(g, x, s) \implies Poss(Grab(g), s)$
  - Effect Axioms
    - ⋆ $Poss(Go(x, y), s) \implies At(Agent, y, Result(Go(x, y), s))$
    - ⋆ $Poss(Grab(g), s) \implies Holding(g, Result(Grab(g), s))$
- *Goal State*: $\exists seq,\ At(G, [1, 1], Result(seq, S_0))$
- From first Possibility Axiom, $Poss(Go(x, y), s)$
- From first Effect Axiom, $At(Agent, y, Result(Go(x, y), s))$
- So can Agent grab the gold?

Nothing in the KB base says that the location of the gold remains unchanged

Frame Axioms

These tell us how the non-changes due to action.
If some object is at some state, and is not the agent and is not being held by the agent, then the object is still in situation S when the agent moves.

If we have f fluents (things that can change) and a actions, it requires O(AF) frame axioms

The Frame Problem

- representational - proliferation of frame axioms (original frame problem)
- representation problem now largely solved
- inferential - having to carry properties through inference steps, even if remain unchanged
- inferential problem avoided by planning; we do not address it for inference systems.

We solve the representational frame problem with **successor state axioms**
- Each axiom is about a predicate (not an action per se)
- General form: p true afterwards = (an action made P true OR P true already and no action made p false)
- We need a successor state axiom for each predicate that can change over time
- Axiom must list all ways the predicate can become true or false

Practicality of Planning

With first order logic, predicates and situational axioms and calculus, we theoretically have all that is required - but this is unpractical (time, space, semi-decidability) etc.

Thus, we need a restricted language. This reduces the number of possible solutions to search through.

Actions represented in a restricted language, allows creation of efficient planning algorithms
So we need a language and a planning algorithm for that language.

STRIPS is a restricted language that lends itself to efficient planning algorithms, while retaining much of the expressiveness of situation calculus representations

Basic Representations for Planning

Representing States and Goals

States are represented by conjunctions of function-free ground literals - or predicates applied to constant symbols, possible negated.

These state descriptions do not need to be complete. An incomplete description, corresponds to a set of possible complete states for which the agent would like to obtain a successful plan.

Many systems adopt the negation as failure convention - if a state description does not mention a given positive literal,
Remember - a goal given to a planner asks for a sequences of actions that makes the goal true if executed, while a query given to a theorem proves asks whether the query is true given the KB

State example: At(Home AND !Have(milk) AND !Have(bananas)
Goal example - we want to be at a shop that sells milk: At(x) AND Sells(x, Milk)

Representing Actions

STRIPS operators have 3 components

- **action description** - what an agent actually returns to the environment in order to do something
- **precondition** - conjunction of atoms (positive literals) that says what must be true before the operator can be applied
- **effect** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied

Op(ACTION:Go(There), PRECONDITION:At(here) AND Path(here, there), EFFECT:At(there) AND !At(here))

An operator with variables is known as an operator schema, because it does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables.

We say that an operator is applicable in a state s if there is some way to instantiate the variables in o so that every one of the preconditions of o is true in s.

For ecample, if the initial state includes the literals At(home), path(home, supermarket) then the action Go(supermarket) is applicable and the resulting situation contains the literals !at(home), At(Supermarket), Path(Home, Supermarket)

Representation for Plans

If we're going to search through plan space, we need to eb able to represnt them. We can settle on a good represntation for plans by considering partial plans for a simple problem: putting on a pair of shoes.

least-commitment - only make choices about things you care about currently, leaving the other choices to be worked out later. This is good for search since you are likely to make the wrong choice and have to backtrack later. A least commitment planner could leave the ordering of the two steps unspecified. This helps us avoid bad plans, by delaying the decision making process until we have more information, allowing the agent to make better choices.

A planner that can represent plans in which some steps are ordered with respect to each other and other steps are unordered is called **partial order planner.**
The alternative planner that plans with a simple list of steps, is a **total-order planner.** A totally ordered plan that is derived from a plan P by adding ordering constraints is called a linearization.

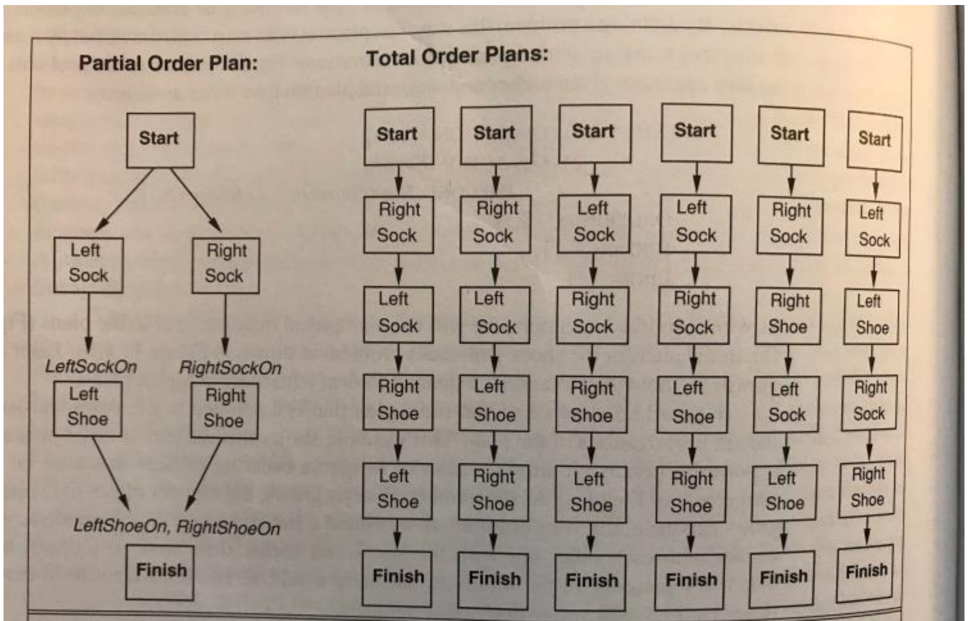

Partial Order Plan:
Total Order Plans:
Start
Left Sock
Right Sock
LeftSockOn
RightSockOn
Left Shoe
Right Shoe
LeftShoeOn, RightShoeOn
Finish
Start → Right Sock → Left Sock → Right Shoe → Left Shoe → Finish
Start → Right Sock → Left Sock → Left Shoe → Right Shoe → Finish
Start → Left Sock → Right Sock → Right Shoe → Left Shoe → Finish
Start → Left Sock → Right Sock → Left Shoe → Right Shoe → Finish
Start → Right Sock → Right Shoe → Left Sock → Left Shoe → Finish
Start → Left Sock → Left Shoe → Right Sock → Right Shoe → Finish

Figure 11.5 A partial-order plan for putting on shoes and socks (including preconditions on steps), and the six possible linearizations of the plan.

Situation Space vs Plan Space

There are a lot of situations that can occur in a world. A path through this space constitutes a plan for a problem. If we wanted, we could take a problem described in the STRIPS languages and solve it by starting at the initial state and applying operators one at a time until we reached a state that includes all the literals in the goal. We could use standard search methods for this

An algorithm that did this would be a problem solver, but it could also be considered a planner. The algorithm would operate in **situation space**. It would be a **progression planner** because it would search forward from the initial situation to the goal situation. Obviously, the branching factor for this method makes it problematic.

One way to cut the branching factor is to search backwards, from a goal state to the initial state. This is a **regression planner**. This approach is possible because the operators contain enough information to regress from a partial description of a result state to a partial description of the state before an operator is applied. We cannot get complete descriptions of states this way, but luckily we don't need to.

This regression approach is desirable because usually, the initial state has many applicable operators that could potentially be used, whereas to move back from the goal state there are typically only a few conjuncts, each of which will only have a few operators.

Searching backwards is hard because we have to achieve a *conjunction of goals*, rather than just one.

Alternatively, we can search through the **space of plans**, rather than the space of plans rather than the space of situations. That is, *we start with a simple, incomplete plan*, which we call a partial plan. Then we *consider ways of expanding the partial plan* until we come up with a complete plan that solves the problem . The operators in this search are operators on plans:

- Adding a step
- Imposing an ordering that puts one step before another
- Instantiating a previously unbound variable.

The solution is the final plan, and the path taken to achieve it is irrelevant.

Refinement operators on plans – take a partial plan and add a constraint. These eliminate plans from the set, but never add new ones
Modification operators – anything that's not a refinement operator is a modification operator. Some planners work by creating an incorrect plan then debugging it with modifications operators.

Plan

- A plan is a data structure consisting of:
- A set of plan steps - each step is one of the operators for the problem
- A set of step ordering constraints e.g. s must occur before x
- A set of variable binding constraints - in the form v = x, where v is a var in some step, and x is either a constant or another variable
- A set of causal links - s_i --c--> s_j, read as "si achieves c for sj". Causal links serve to record the purpose of steps in the plan: here a purpose of s_i is to achieve the precondition of s_j

The initial plan before any refinements take place, simple describes the unsolved problem. It has two steps - Start and Finish and the constraint that start is before finish. There are not links or bindings.

Plan Solution

A solution is a plan that an agent can execute, that guarantees achievement of the goal.
To check a plan is valid, it makes sense to insist on a fully instantiated, totally ordered plan.
However, this is unsatisfactory:

1. Agents can perform tasks in parallel so it makes sense to allow solutions with parallel actions
2. There are many linearisation's of a plan – it is more natural to just return the PO plan than arbitrarily pick a plan
3. If we're creating plans that will be combined with larger plans, it makes sense to maintain flexibility

Therefore, we accept plans that are partially ordered, and

- Complete – every precondition of very step is achieved by some other step – A step achieves a condition if the condition is one of the effects of the step, and if no other step can possible cancel out the condition.
- Consistent – a consistent plan is one in which there are no constrictions in the ordering or binding constraints. A contradiction occurs when both Si must be before Sj and Sj must be before Si – remember plans are transitive.