

Using logic in searching problems

## Lecture 13 Planning

Now it is finally time to do things **really** funny with logic.

In the beginning of the course, we study how to do searching. And later we start working on logic, in order to represent more complicated worlds.

Now we will **actually use logic in searching problems**, and find some heuristics that makes searching faster.

**Reference:**

- Chapter 11, up to page 345.

AI(0270)

AI(0270)-13.1

Simplifying our problem

- We will need to focus on a **changing world**. So **let's make the logic system simpler first**.  
It is possible to add them back later. But in this course, this "add them back" will not happen.
- We assume that the world is always represented by **some predicates**. The semantics is that the predicate holds at that state.
- There are some **operators** which brings us from one state to another. These operators may add or delete some predicates from our state.
- Some operators can be applied only if some "**pre-conditions**" hold.
- The initial state has **certain known predicates** that holds. The goal state requires that **some** predicates hold.
- So our task is to **find a sequence of operators** that will achieve the required goal when applied.

AI(0270)-13.2

STRIP representation

STRIP: a standard **format** for planning problems. It includes three parts:

- **State representation**. It is a set of predicates, perhaps negated. The objects in the predicates must be constants.  
E.g.,  $At(Home) \wedge \neg Have(Milk) \wedge \neg Have(Bananas) \dots$
- **Goal representation**. It is a set of predicates, perhaps negated. The object maybe variables, and if there are, they are existentially quantified.  
E.g.,  $At(Home) \wedge Have(Milk) \wedge Have(Bananas) \dots$
- **Operator representation**. It includes 3 parts:
  1. **Action description**: a name of the operator (action).
  2. **Precondition**: a conjunction of positive predicates that must be true before the action is applicable.
  3. **Effect**: a conjunction of predicates that the action adds to the state.

AI(0270)-13.3

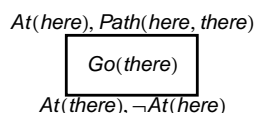
The operator

An operator is of the following form:

$Op(ACTION : Go(there), PRECOND : At(there) \wedge Path(there, there), EFFECT : At(there) \wedge \neg At(there))$

In many language, the positive terms of the effect are grouped together to form an **add list**, and the negated terms forms a **delete list**.

It can be graphically represented (we will soon use this extensively):

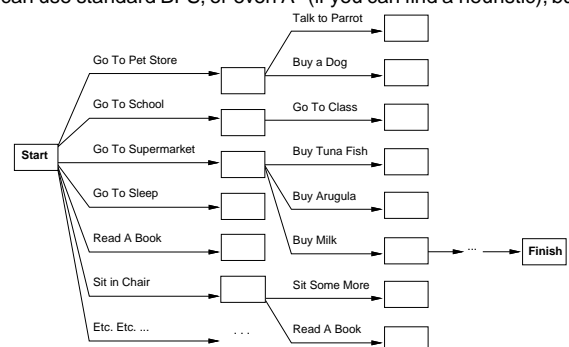


The plan: a sequence of operators with variables instantiated, e.g.,  $[Go(Home, Supermarket), Buy(Banana), Go(Supermarket, Home)]$ .

AI(0270)-13.4

Use standard searching?

You can use standard BFS, or even A\* (if you can find a heuristic), but...



the branching factor is simply too big...

AI(0270)-13.5

### One way to try limiting branch factor: Regression

- If our goal contains only a few predicates that **can be independently established**, then we can reduce branching factor by **regression**, i.e., search backwards:
  - Add the goal predicates as the list of **open predicates**.
  - At each step, **choose** an open predicate.  
We use the word "choose" to mean that it should be possible to back-track and choose another. The word "pick" means that no backtracking is needed.
  - **Choose** an action that **adds** the predicate and do not **delete** any other **satisfied** predicate. This can also be the **null** action that does nothing if the predicate is already satisfied.
  - Add the operator to the **end** of the plan.
- This looks like backward chaining. The difference is that now **the world changes through the operators**. (In chaining, no changing world.)

AI(0270)-13.6

### But is it good enough...

The strategy does not improve the situation **that** much.

- Suppose our goal is  $At(Home) \wedge Have(Milk) \wedge Have(Drill)$
- It does help us to avoid **having to consider irrelevant actions**.  
We won't need to consider things like go to sleep.
- But yet we need to search a lot to know that  $At(Home)$  eventually **needs to be established**.  
Very counter-intuitive: originally  $At(Home)$  is satisfied! And of course, any operator which can create  $Have(Milk)$  (e.g.,  $Buy(Milk)$ ) have a precondition like  $At(Supermarket)$ , which can be established only if  $At(Home)$  is deleted.
- So we need  $Go(x, Home)$ . But what's  $x$ ?  
So we need to consider every possibility here: from the cinema, from the hardware store, from the supermarket, from the school, from the pet store, ...
- It is clear that **we have too much searching to do**.

AI(0270)-13.7

### Solving the problem: first step

So **what we can do?**

- At some time we **need to make**  $Have(Milk)$  satisfied. We know that we need something like  $Buy(Milk)$  which establish it.
- But **why we stick it to the end of our plan?** This is unjustified, apart from that **one** of the establishment of  $Have(Milk)$ ,  $Have(Drill)$  and  $At(Home)$  will be the last operator.
- This means it is possible that **only one succeed**, while we have to waste time trying out **all** possibilities!
- So is it possible to choose an operator, but **allow it to situate anywhere** in the plan?

This is indeed the first of a sequence of **common lines of attack**: try **not to commit** too much when we don't really know that much.

AI(0270)-13.8

### New algorithm

1. Initialize the list of **protected** predicates to empty.  
What is protected? See below.
2. If no unsatisfied predicate, return the empty action list as plan.
3. **Choose a predicate** that is **not satisfied**.
4. **Choose** an operator that **adds** the predicate and do not **delete** any other **protected** predicate.
5. **Recursively** make a plan with the goal being the **pre-condition** and with the same starting state. (Retain continuations!)
6. Execute the found plan and find the resulting **intermediate** state.
7. Mark the chosen predicate as **protected**, and recursively plan from the intermediate state to the goal state.  
Protection is needed so that the achieved predicate will not be removed.

Implementing this in Prolog is a very good (but not very difficult) exercise.

AI(0270)-13.9

### How it plans our problem

- At the beginning  $Have(Milk)$  is an unachieved goal.
- $Have(Milk)$  is added by  $Buy(Milk)$ , with precondition  $At(SuperMarket)$ .
- **Recursively** plan from  $At(Home) \wedge \neg Have(Milk) \wedge \neg Have(Drill)$  to  $At(SuperMarket)$ :
  - $At(SuperMarket)$  is an unachieved goal.
  - This is added by  $Go(x, SuperMarket)$ .
  - If we choose  $x = Home$ , all preconditions are satisfied.
  - All needed clauses are there, so return.
- Recursively run planner, starting at  $At(SuperMarket) \wedge Have(Milk) \wedge \neg Have(Drill)$ , goal  $At(Home) \wedge Have(Milk) \wedge Have(Drill)$ , and with  $Have(Milk)$  protected.

AI(0270)-13.10

### How it plan our problem: cont'd

- We find that  $Have(Drill)$  is not satisfied yet, and  $Buy(Drill)$  achieve it.
- $Buy(Drill)$  have the precondition  $At(HardwareStore)$ .
- **Recursively** call the planner, starting with  $At(SuperMarket) \wedge Have(Milk) \wedge \neg Have(Drill)$ , with goal  $At(HardwareStore)$ , and with the predicate  $Have(Milk)$  protected.
  - $At(HardwareStore)$  is an unachieved goal.
  - This is added by  $Go(x, HardwareStore)$ .
  - If we choose  $x = HardwareStore$ , all preconditions are satisfied.
  - All needed clauses are there, so return.
- Recursively run planner, starting at  $At(HardwareStore) \wedge Have(Milk) \wedge Have(Drill)$ , goal  $At(Home) \wedge Have(Milk) \wedge Have(Drill)$ , and with  $Have(Milk)$  and  $Have(Drill)$  protected.

AI(0270)-13.11

### Concluding our example...

- The final recursion find that  $At(Home)$  is the only thing unsatisfied, so it find  $Go(HardwareStore, Home)$  as a plan.
- After this, we can combine the plans and get  $Go(Supermarket)$ ,  $Buy(Milk)$ ,  $Go(HardwareStore)$ ,  $Buy(Drill)$ ,  $Go(Home)$ .
- Is it **really** good? How fast a planner can do this depends on the number of choices it has to make.
- In our example, not too many, but not few either:
  - It has to choose to **try buying milk first**.  
Why not drill first? It is also okay, but it has to choose, making planning slow.
  - It has to choose to **buy drill before going home**.
  - It has to choose to **go directly** from home to supermarket, from supermarket to hardware store, and from hardware store to home.  
Yes, they all add to the branching factor if we do IDFS.

AI(0270)-13.12

### Why it improves the situations? Applicability of planning.

- So **why this improves the situation** over searching?
- We still have decisions to make, but **the number of choices** per decision is not large. This gives us a small branching factor.  
... and thus allow a larger search depth.
- In contrast, if we do it in with regression (or pure searching), the branching factor will be very large (usually in the 10000 for real applications).
- This is achieved by allowing us to make a decision **before** we can **attach** it to (the start or end of) the plan.
- Why these decisions are worthy? Because in most easier problems, **the world usually don't change that much** by the operators.
- This allows us to break up a big problem into **many small** ones.  
In contrast, if we solve the 8-puzzle with this approach, we won't gain much because after we can have a subtask to bring 1 and 2 to the right place, but moving 3 right needs to destroy what we achieved.

AI(0270)-13.13

### Some deficiencies

There are some deficiencies of our algorithm. We have seen one:

- We have to commit on an ordering** even though it really doesn't matter (until the end of the plan).

In our example, we have to commit that **Have(Milk)** is done before **Have(Drill)**, and whether to achieve **Have(Drill)** or **At(Home)** first is complete guess-work.

- We have to commit on a variable binding** for the action before it really matter.

E.g., if we have other stores that sell milk (perhaps  $Sells(FastWood, Milk)$ ), we have to commit on one.

Both increases branching factor.

AI(0270)-13.14

### What's so bad about commitment?

What's so bad committing into something that we don't need to commit, or committing too early?

Basically, we increases our branching factor.

- If we want the best plan, we do **BFS** or **IDFS**. The larger branching factor directly translate to more **search time**.
- If it turns out that we have made the wrong choice, but is discovered only much later, we must **forget everything we have planned**, and backtrack to this "reason of failure".
- And in the middle, we can have more **wrong** backtracking that can be done, and we will try all of them before undoing the reason of failure.
- Even outputting all possible solutions can be very time consuming.  
If we have  $n$  independent actions, we have  $n!$  ordering possibilities.

These become more serious when we have larger planning problems.

AI(0270)-13.15

### But the problem is even more serious...

It turns out that our planner is not complete. Let's see an example...

- Consider the following planning problem in a *block-world*: we have some **blocks**, and we want to **move them to some configuration**.
- In the state, we have knowledge about whether a block is **clear on the top** so that it can be moved and other blocks can be moved into it.
- There are some **places** where the bottom block can be placed.

- Our initial position:
 

- Our goal: **A is on B, and B is on C**. We don't care where C sits on.  
It turns out that it would be easier if we want that C sits on 2 or 4.  
(Question: What is the optimal plan?)

AI(0270)-13.16

### STRIP representation

- Starting state:  $On(C, A)$ ,  $On(A, 1)$ ,  $On(B, 3)$ ,  $Clear(C)$ ,  $Clear(2)$ ,  $Clear(B)$ ,  $Clear(4)$ .
- Goal:  $On(A, B)$ ,  $On(B, C)$
- Operator: only 1 operator

$Clear(obj)$ ,  $On(obj, here)$ ,  $Clear(there)$

$Move(obj, here, there)$

$On(obj, there)$ ,  $Clear(here)$ ,  $\neg Clear(there)$

It might seems that this problem is much easier than the supermarket problem. But it is not.

Because there are 3 variables in the Move operator, and usually many are applicable at the same time, the computational complexity is very large.

AI(0270)-13.17

**Example: cont'd**

- If you run the above algorithm, we will get a plan like this:
  1. Move C to 2.
  2. Move B to A. (!!)
  3. Move B to C.
  4. Move A to B.
- Why we don't directly move B to C in step 2 instead?
  - If we start by thinking of ways to satisfy **On(A,B)**, we will never try moving B from 3 to C: it **won't add anything unsatisfied**.
  - If we start by thinking of ways to satisfy **On(B,C)**, we will never try moving C from A to either 2 or 4 with the same reason!
- So in our planning process, we **messed out some important choices**.  
We need to consider all goals together, not one by one!