50.021 -AI

Alex

Week 08: Search

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

The idea of this lecture is

- to describe planning problems in a simple formalism - that is finding a sequence of actions.
- introduce the PDDL planning language
- discuss planning as a search problem with heuristics
- discuss a few simple heuristics.

STRIPS

originally a planner software, today mostly used to name a formal language to describe planing problems.

Planning is about finding a sequence of actions to achieve a goal. how to describe such things?

Suppose you know facts. There are 2 rooms. Robot is in room A, ball is in room B. You can model such facts – in the simplest form – by variables that are true or false.

Variables that are true or false are called **propositional variables**.

so you can have a fact $inA(?ball)$ which just encodes the question whether the ball is in room A. It will be true or false.

Same you can have a fact $inA(?robot)$.

How to model actions using facts encoded as boolean variables?

Actions change facts – here they make them true or false.

- an action may make some fact true
- an action may make some other fact false.

Action may require some facts to be true or false in order to be able to execute an action.

Example: a robot wants to kick the ball from room B to room A.

What are the necessary requirements?
$inB(robot), inB(ball), dooropen(roomA, roomB)$

What does the action "kick ball from room B to room A" do in terms of changing facts?
$not\ inB(ball), inA(ball)$. does it change any other facts? No, the robot will still be in room B, the door between the rooms will be open.

Lets consider another example – the simple blocksworld:

Blocks world

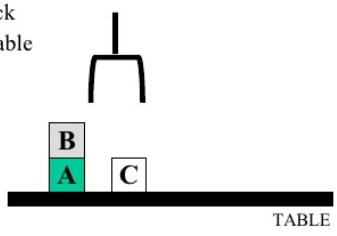
The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

Some domain constraints:

- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

ontable(a)
ontable(c)
on(b,a)
handempty
clear(b)
clear(c)



5

There is a table, and, yep, blocks, and a gripper.
facts you can have in this world as true-false propositional variables:

- ontable(someblock)
- on(block1,block2) - block1 stacked on block2
- clear(block) - no block on top of that, if true
- handempty() - the gripper is free

See the description of the world state above.

Lets put this all together formally:
A STRIPS instance is composed of:

- An initial state;

- The specification of the goal states – situations which the planner is trying to reach
- A set of actions. For each action, the following are included:
  - preconditions: true and false facts that must hold so that an action can be performed
  - postcondition: facts that change when an action is performed.

  Definition: Formally! Strips is a 4-tuple (P, O, I, G)
- P is a set of propositional variables – the facts that describe the state of the world
- O is a set of operators (that is: actions). Each operator itself is a 3-tuple (pre_a,add_a,del_a) of
  - pre_a - facts that must be true before the action can be performed [1]
  - add_a - facts that will change to true when/after the action can be performed
  - del_a - facts that will change to false when/after the action can be performed
  - requirement: $add_a \cap del_a = \emptyset$
- I - the initial state of the world, true or false assignments to variables from P
- G - the goal state of the world

Solving a planning problem means to find a sequence of actions which transform I into G. Makes sense?

Example:

Traveling in Australia - Planning slide 27-42

What is the difference to search? Search can be used to find such a sequence.

Simplifications made by this? all is just true or false, black or white. no grey zones. The perfect world that does almost never exist. No probabilities that an event is true or false.

The uncertainty principle of quantum mechanics:

There are fundamental limits to the precision to which pairs of physical properties of a particle can be known.

[1] (in a stricter version not covered one could require also that facts must be false, but that changes a lot, makes no-delete relaxations impossible)

Example: it is not possible to know at the same time the position and momentum of an electron to arbitrary precision. The product of standard deviations of those is *lower*-bounded by:

$$\sigma_x \sigma_p \geq \frac{h}{4\pi}, h = 6.626 \cdot 10^{-34} J \cdot s$$

Probabilities are at small scales the basic building blocks in this world.

PDDL

A language to describe planning problems that can be used as input to planner software. Quasi-Standard for classical planning.

See Malte Helmert's slides.

Components of a PDDL planning task:

- Objects: Things in the world that interest us.
- Predicates: Properties of objects that we are interested in; can be true or false
- Initial state: The state of the world that we start in.
- Goal specification: Things that we want to be true.
- Actions/Operators: Ways of changing the state of the world.

Planning tasks specified in PDDL are separated into two files:

- A domain file for predicates (= facts) and actions.
- A problem file for objects, initial state and goal specification.

example Domain file - but see the slides, I cannot make it better than Malte Helmert:

Note: the facts are instantiated with objects e.g. (ball ?b) - so that we can inquire for every object whether it is a ball!

```
(define (domain gripper-strips)
   (:predicates (room ?r)
(ball ?b)
(gripper ?g)
(at-robby ?r)
(at ?b ?r)
(free ?g)
(carry ?o ?g))
```

```
   (:action move
       :parameters  (?from ?to)
       :precondition (and  (room ?from) (room ?to) (at-robby ?from))
       :effect (and  (at-robby ?to)
     (not (at-robby ?from))))



   (:action pick
       :parameters (?obj ?room ?gripper)
       :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
    (at ?obj ?room) (at-robby ?room) (free ?gripper))
       :effect (and (carry ?obj ?gripper)
    (not (at ?obj ?room))
    (not (free ?gripper))))



   (:action drop
       :parameters  (?obj  ?room ?gripper)
       :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
    (carry ?obj ?gripper) (at-robby ?room))
       :effect (and (at ?obj ?room)
    (free ?gripper)
(not (carry ?obj ?gripper)))))
```

example Problem file:

```
(define (problem strips-gripper4)
   (:domain gripper-strips)
   (:objects rooma roomb ball1 ball2 ball3 ball4 left right)
   (:init (room rooma)
          (room roomb)
          (ball ball1)
          (ball ball2)
          (ball ball3)
          (ball ball4)
          (gripper left)
          (gripper right)
          (at-robby rooma)
          (free left)
          (free right)
          (at ball1 rooma)
          (at ball2 rooma)
          (at ball3 rooma)
```

```
          (at ball4 rooma))
   (:goal (and (at ball1 roomb)
               (at ball2 roomb)
(at ball3 roomb))))
```

search Heuristics: h_+

Caveat for this all to work: We never require the absence/negation of a fact as precondition of an action to be executable.

consider a relaxed problem: a setup with no deletes at all:

$$\forall \textit{actions } a: \ del_a = \emptyset \ .$$

We can transform any problem into a delete-relaxed problem by setting above.

Why a delete-relaxed problem is an easier problem? Every plan that solves the problem with deletes, also solves the problem without deletes - because the goal state is simply to have some facts to be set to true, and it does not matter if other additional facts are true, too.

This implies: **the minimal number of steps to solve a problem with delete-relaxation can be not larger than the minimal number of steps to solve the original problem (bcs the solution of the original problem solve the delete-relaxed prob. It can exist in the delete relaxed prob faster solutions though).**

Therefore we can use the minimal number of actions needed to solve a problem with delete-relaxation as heuristic for a state in the original problem – bcs it never overestimates the cost in the original problem!

definition: The minimal number of actions in a problem without deletes is called h_+ heuristic. Admissible by design as heuristic for every state of the original problem.

The problem with delete relaxation is an easier problem – but any heuristic needs to be computed in for every state that is generated. So every time we apply an action to generate a new state, every time we have to solve a planning problem to compute h_+ for this one state as start state – expensive.

Need to approximate h_+.

Faster Planner Search Heuristics: h_{add}, h_{max}, h_{FF}

All Derived from the no delete relaxation. Not admissible, but easier to compute than h_+.

h_{add}, h_{max} needs only a forward pass. h_{FF} needs the same forward pass, and an additional backward pass.

What is the idea:

Every action a can be applied only if all facts in its precondition set pre_a are true. Given a set of true facts, we can check what action are applicable, apply them, and create a new set of facts. With this new set of facts, since we have no deletes, some facts are added/made true, so that new actions can be applied now (bcs we have more true facts now).

- F_0 is the initial set of true facts
- Some actions can be applied to it. Apply all applicable actions with no-delete-relaxation. Lets call this set A_0. This will result in F_0 plus some added facts. Define $F_1 = F_0+$ plus the added facts that come from applying every action $a \in A_0$.
- now F_1 is again a set of facts. apply all applicable actions that could not be applied before. Lets call this set of newly applicable actions A_1.
- iterate this: one has a set F_i, apply all applicable actions A_i to yield F_{i+1}.
- terminate at iteration number M when the set F_M of facts contains all goal facts.
- define for every fact a level. The level of a fact g is the index i of the set F_i in which the fact g appeared/was added for the first time.
- define for every action that was applied in the action sets A0,A1,A2 a level. The level of an action a is the index i of the set A_i in which the action a was used / was applicable **for the first time** because all facts in the precondition set of this actyion became true.).

  execute such a set forward pass as an example
- What for is this useful? The level of a fact is the minimal number of actions needed to make a fact become true. facts in F_1 need exactly one action. Facts in F_2 need 2, and so on. So: **the index of the F_i set is a measure for the cost to make a fact in F_i true.** These costs can be used to define heuristics!

- can define h_{max} now: h_{max} is the maximum over all levels of all facts in the goal set.

  h_{max} is the cost of the single most costly goal fact - the max number of actions needed for achieving one of the goal facts.

- can define h_{add} now: h_{add} is the sum over all levels of all facts in the goal set.

  h_{add} is the summed cost of all goal facts.

  why it is a heuristic that can overestimate the true cost? Because summing the costs makes an assumption: making each fact true happens by independent paths of actions.

  In reality: one action in set A_{i-1} can add multiple facts to set F_i. so if two facts are added by the same action, then summing their cost counts all necessary actions twice. So h_{add} is really loose and overestimating.

- h_{max} is the other extreme: it looks at one longest path of actions, and ignores that achieving multiple goals can have partially independent/non-overlapping paths of actions leading to all those multiple goals.

- reality is in between: when achieving all goals in the delete relaxed problem, paths of actions can be partially independent, and partially overlapping/shared

- h_{FF} tries to find a common set of shared actions by using a backward pass (in a simple manner). shared applies to: the paths to make all goal facts true.

Explaining h_{FF}

For h_{FF} need a backward pass, in which a set of selected actions alias shared actions is created.

Rough idea: maintain sets G_t - this is initialized with only the goal facts that have level equal to t. Goal fact means: a fact that is part of the goal state.

Let G_t **be at the start** the set of goal facts at level t:

$$\begin{aligned} G_t &= \{g \in G \cap (F_t \setminus F_{t-1})\} \\ &= \{g \in G \mid level(g) = t\} \end{aligned}$$

We want to find a set of selected actions which are needed to make all these goals facts, at all levels t, become true.

Suppose M is the maximal index of F_i.

Lets start at the very end: We can ask which actions $a \in A_{M-1}$ can be used to make the goal facts in G_M true? If we have found such actions, then we will add these actions to the selected set. But the selected set will need to contain more actions! Why?

The above added actions have preconditions pre_a which are sets of facts. We must have all the facts in the preconditions pre_a to be true, in order to be able to execute action a.

Therefore: all the facts in pre_a must be made true by **preceding other actions** – and these preceding other actions will need to be added to the selected set, too.

The preconditions pre_a are facts which were added in the forward pass to at least one of the sets $F_t,\ t < M$ (otherwise how could the action a be in A_{M-1}?? It is in A_{M-1} because all preconditions were met)

During the backward pass of the h_{FF} algorithm, the set G_t will be enlarged by adding those non-goal facts

- which have level equal to t and
- which are necessary to achieve a goal fact at some higher level $t' > t$.

  necessary means here because they are part of pre_a for some action used

  - either to make a goal fact true
  - or to make another fact true on the way to some goal fact (even further above: $t' > t$).

Why adding those? Our goal: We want to find a common set of shared actions that is necessary to make all goal facts true. If we need a non-goal fact to be true on the way to reaching a goal fact, and this non-goal fact is in F_t, then we will need to add an action in A_{t-1} to the selected set that makes this non-goal fact true.

Suppose M is the maximal index of F_i.

We start the algorithm by trying to find actions that make all the goals in $G_t, t = M$ become true:

selected= $\{\}$ for $t = M, \ldots, 1$ do:
  for all $f \in G_t$ do:
    select $a, level(a) = t - 1$ such that $f \in add_a$
    #(this action a will be added to the selected set)
    selected= $selected \cup \{a\}$
    for all $p \in pre_a$ do:
      $G_{level(p)} = G_{level(p)} \cup \{p\}$
      # the preconditions $p \in pre_a$ of the chosen action must be made true, before the chosen action a can be used, so we add them at their level, so that they will be satisfied by another action from $A_{level(p)-1}$
    end for
  end for
end for
h_{FF} = number of selected actions $a = |\#selected|$

$h_{FF} =$ can be an overestimate: we select for every fact in g_t simply one action which can make it true. We do not care if there is a smart choice of an action that creates multiple facts at the same time.

Also: suppose we can choose at a high level t between two actions that make the same fact true. We do not care whether these actions needs more or less preconditions in lower levels t. So we could choose an action that needs preconditions which need many other actions further down in order to achieve them.

- h_{max} is admissible, but often too loose, too low
- h_{add} is inadmissible, sometimes way too large compared to h_+ which we want to approximate
- h_{FF} is inadmissible, but in practice useful