CS 5600/6600: F21: Intelligent Systems
Assignment 10
Knowledge Engineering for GPS

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 6, 2021

Learning Objectives

1. Means-Ends Analysis
2. General Problem Solver (GPS)
3. Knowledge Engineering GPS Operators

Problem 1 (2 points)

Read the article "GPS, A Program that Simulates Human Thought" by A. Newell and P. Simon. A scanned pdf of this article is included in the zip. This article influenced multiple generations of AI planning researchers and created an intellectual framework that is still with us today.

GPS was a major breakthrough in AI planning. Its knowledge engineering methodology of desigining problem-specific operators and using a general problem solving method (e.g., means-ends analysis) had a significant impact on subsequent planning systems (e.g., STRIPS, ABSTRIPS, SOAR), and is still used in modern AI planners.

Running GPS

The zip has three files – `auxfuns.lisp`, `gps.lisp`, and `ops.lisp`. The file `auxfuns.lisp` contains some auxiliary functions, the file `gps.lisp` is a Common Lisp implementation of the GPS planner, and `ops.lisp` is where you'll write your GPS operators for Problems 2 and 3 below.

Let's see how we can run GPS on the drive-son-to-school problem we discussed in class this week. Change your directory to where you unzipped `gps.lisp`, `auxfuns.lisp`, and `ops.lisp`, fire up your Lisp, and load `gps.lisp` into it.

```
> (load "gps.lisp")
;; Loading file gps.lisp ...
;;  Loading file auxfuns.lisp ...
;;  Loaded file auxfuns.lisp
;;  Loading file ops.lisp ...
;;  Loaded file ops.lisp
;; Loaded file gps.lisp
T
```

In ops.lisp, the variable *school-ops* is a list of operators we developed and analyzed in class to solve the son-at-school problem.

```
(defparameter *school-ops*
  (list
    (make-op :action 'drive-son-to-school
         :preconds '(son-at-home car-works)
         :add-list '(son-at-school)
         :del-list '(son-at-home))
    (make-op :action 'shop-installs-battery
         :preconds '(car-needs-battery shop-knows-problem
                     shop-has-money)
         :add-list '(car-works))
    (make-op :action 'tell-shop-problem
         :preconds '(in-communication-with-shop)
         :add-list '(shop-knows-problem))
    (make-op :action 'telephone-shop
         :preconds '(know-phone-number)
         :add-list '(in-communication-with-shop))
    (make-op :action 'look-up-number
         :preconds '(have-phone-book)
         :add-list '(know-phone-number))
    (make-op :action 'give-shop-money
         :preconds '(have-money)
         :add-list '(shop-has-money)
         :del-list '(have-money))))
```

The variable *school-world* defines the initial state of the world in which the GPS planner has to work.

```
(defparameter *school-world* '(son-at-home car-needs-battery
                               have-money have-phone-book))
```

You don't have to change anything either in *school-ops* or *school-world* (unless, of course, you want to play with these operators or the world on your own).

Let's use these operators to solve the son-at-school problem with GPS.

First, we tell GPS which operators it needs to use. The integer (i.e., 6) specifies how many operators have been defined. You can verify that the list *school-ops* contains exactly 6 operators.

```
> (use *school-ops*)
6
```

Second, we run GPS on the state of the world in *school-world* and the goal, which in our case is '(son-at-school).

```
> > (gps *school-world* '(son-at-school))
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
(EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
(EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
```

As we can see from the the above output, GPS returns a plan for the agent (a human or a robot) to follow. The plan consists of looking up the auto shop's number, phoning the shop, telling the shop about the battery problem, giving the shop the money, having the shop install the battery, and driving the son to school. If we need to save the plan, we can do it by saving it in a variable.

```
> (setf son-at-school-plan (gps *school-world* '(son-at-school)))
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
 (EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
 (EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
> son-at-school-plan
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
 (EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
 (EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
```

If we want to trace how GPS solves a problem step by step, we can use the function `trace-gps` and then call the function `gps()` on the world's state and the goal to see means-ends analysis goal tree unfolding layer by layer.

As shown below, GPS works by recursively satisfying the preconditions of each operator so that it can be applied to reduce the differences between the current state of the world and the desired state of the world where the goal is satisifed. Each unsatisfied pre-condition, in turn, becomes a goal.

```
> (trace-gps)
(:GPS)
> > (gps *school-world* '(son-at-school))
Goal: SON-AT-SCHOOL
Consider: DRIVE-SON-TO-SCHOOL
  Goal: SON-AT-HOME
  Goal: CAR-WORKS
  Consider: SHOP-INSTALLS-BATTERY
    Goal: CAR-NEEDS-BATTERY
    Goal: SHOP-KNOWS-PROBLEM
    Consider: TELL-SHOP-PROBLEM
      Goal: IN-COMMUNICATION-WITH-SHOP
      Consider: TELEPHONE-SHOP
        Goal: KNOW-PHONE-NUMBER
        Consider: LOOK-UP-NUMBER
          Goal: HAVE-PHONE-BOOK
        Action: LOOK-UP-NUMBER
      Action: TELEPHONE-SHOP
    Action: TELL-SHOP-PROBLEM
    Goal: SHOP-HAS-MONEY
    Consider: GIVE-SHOP-MONEY
      Goal: HAVE-MONEY
    Action: GIVE-SHOP-MONEY
  Action: SHOP-INSTALLS-BATTERY
Action: DRIVE-SON-TO-SCHOOL
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
(EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
(EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
```

If we want to turn the tracer off, we do

```
> (untrace-gps)
```

Problem 2 (1 point)

The Sussman Anomaly is a famous AI planning problem named after its discoverer Dr. Gerald Sussman. Imagine a simple robot that has a camera and a gripper (a camera-arm unit). The robot is supposed to build blocks in a simple block world. A simple version of the Sussman Anomaly includes three blocks A, B, and C, on the table T where C is on A, and A and B are on T.

The file `ops.lisp` defines the initial state of the world as follows.

```
(defparameter *block-world* '(a-on-t b-on-t c-on-a clear-c clear-b))
```

The symbols `clear-c` and `clear-b` indicate that the tops of C and B are cliear and other blocks can be placed on them or they can be grabbed by the robot.

Write the operators that allow the world to build the ABC tower from the initial state of the block world in `*block-world*` and avoid the Sussman Anomaly (i.e., the `PREREQUISITE-CLOBBERS-SIBLING-GOAL` problem). My solution consists of 3 operators. Below are my sample runs.

```
> (use *block-ops*)
3

> (gps *block-world* '(a-on-b b-on-c))
Goal: A-ON-B
Consider: PUT-A-FROM-T-ON-B
  Goal: A-ON-T
  Goal: CLEAR-A
  Consider: PUT-C-FROM-A-ON-T
    Goal: C-ON-A
    Goal: CLEAR-C
    Goal: A-ON-T
  Action: PUT-C-FROM-A-ON-T
  Goal: CLEAR-B
  Goal: B-ON-C
  Consider: PUT-B-FROM-T-ON-C
    Goal: B-ON-T
    Goal: CLEAR-B
    Goal: CLEAR-C
    Goal: C-ON-T
  Action: PUT-B-FROM-T-ON-C
Action: PUT-A-FROM-T-ON-B
Goal: B-ON-C
((START) (EXECUTE PUT-C-FROM-A-ON-T) (EXECUTE PUT-B-FROM-T-ON-C)
 (EXECUTE PUT-A-FROM-T-ON-B))

> (gps *block-world* '(b-on-c a-on-b))
Goal: B-ON-C
Consider: PUT-B-FROM-T-ON-C
  Goal: B-ON-T
  Goal: CLEAR-B
  Goal: CLEAR-C
  Goal: C-ON-T
  Consider: PUT-C-FROM-A-ON-T
    Goal: C-ON-A
    Goal: CLEAR-C
```

```
    Goal: A-ON-T
  Action: PUT-C-FROM-A-ON-T
Action: PUT-B-FROM-T-ON-C
Goal: A-ON-B
Consider: PUT-A-FROM-T-ON-B
  Goal: A-ON-T
  Goal: CLEAR-A
  Goal: CLEAR-B
  Goal: B-ON-C
Action: PUT-A-FROM-T-ON-B
((START) (EXECUTE PUT-C-FROM-A-ON-T) (EXECUTE PUT-B-FROM-T-ON-C)
 (EXECUTE PUT-A-FROM-T-ON-B))
```

Problem 2 (2 points)

Here's another AI planning classic. It's called the Monkey and Bananas Problem. It's classic as the Sussman Anomaly. Imagine the following situation. A hungry monkey is standing at the doorway to a room holding a ball in his hands. In the middle of the room there is a bunch of bananas suspended from the ceiling by a rope, well out of the monkey's reach. There is a chair near the door, which the monkey can push. The chair is tall enough for the monkey to get the bananas after he climbs on it. The monkey cannot grasp the bananas without letting go of the ball in his hands.

Design a set of operators for the monkey to quench his hunger and save your operators in the variable `*banana-ops*` in `ops.lisp`. The initial state of the world for this problem is defined in `*banan-world*` in `ops.lisp` as follows.

```
(defparameter *banana-world* '(at-door on-floor has-ball hungry
                               chair-at-door))
```

Below is one possible six-operator solution to this problem. Of course, your solution may be different in that your knowledge representation may be different.

```
> (use *banana-ops*)
6

> (gps *banana-world* '(not-hungry))
Goal: NOT-HUNGRY
Consider: EAT-BANANAS
  Goal: HAS-BANANAS
  Consider: GRASP-BANANAS
    Goal: AT-BANANAS
    Consider: CLIMB-ON-CHAIR
      Goal: CHAIR-AT-MIDDLE-ROOM
      Consider: PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM
        Goal: CHAIR-AT-DOOR
        Goal: AT-DOOR
      Action: PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM
      Goal: AT-MIDDLE-ROOM
      Goal: ON-FLOOR
    Action: CLIMB-ON-CHAIR
    Goal: EMPTY-HANDED
    Consider: DROP-BALL
      Goal: HAS-BALL
    Action: DROP-BALL
  Action: GRASP-BANANAS
Action: EAT-BANANAS
```

```
((START) (EXECUTE PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM)
 (EXECUTE CLIMB-ON-CHAIR) (EXECUTE DROP-BALL)
 (EXECUTE GRASP-BANANAS) (EXECUTE EAT-BANANAS))
```

What to Submit

Save your operators in `*block-ops*` and `*banana-ops*` and submit `ops.lisp` in Canvas. In the comments at the beginning of `ops.lisp`, save the plans that GPS found four your set of operators for Problems 2 and 3. Submit your paper analysis in `gps_paper.pdf`.

Happy Reading, Writing, and Knowledge Engineering!