CS 5600/6600: F24: Intelligent Systems
Assignment 11
Knowledge Engineering for the General Problem Solver (GPS)

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 10, 2023

Learning Objectives

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

Problem 1 (2 points)

The read part of this assignment is the article "GPS, A Program that Simulates Human Thought" by A. Newell and P. Simon. This article influenced many AI researchers and created an intellectual framework of AI planning that is still with us today.

GPS was a major breakthrough in AI planning. Its knowledge engineering methodology of designing 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. We will investigate STRIPS in our upcoming lectures as an AI planner that expanded the GPS framework by integrating automated theorem proving into operator selection.

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 Lisp implementation of the GPS planner, and `ops.lisp` is where you will 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.

Here's how the Lisp operator data structres print out in the Lisp window. Recall that `\#S` means that it's a Lisp data structure similar to the C struct or a C++ class w/o inheritence.

```
> *school-ops*
(#S(OP :ACTION DRIVE-SON-TO-SCHOOL
       :PRECONDS (SON-AT-HOME CAR-WORKS)
       :ADD-LIST ((EXECUTE DRIVE-SON-TO-SCHOOL) S
                  ON-AT-SCHOOL)
    :DEL-LIST (SON-AT-HOME))
 #S(OP :ACTION SHOP-INSTALLS-BATTERY
       :PRECONDS (CAR-NEEDS-BATTERY SHOP-KNOWS-PROBLEM
                  SHOP-HAS-MONEY)
       :ADD-LIST ((EXECUTE SHOP-INSTALLS-BATTERY)
                  CAR-WORKS)
       :DEL-LIST NIL)
 #S(OP :ACTION TELL-SHOP-PROBLEM
       :PRECONDS (IN-COMMUNICATION-WITH-SHOP)
       :ADD-LIST ((EXECUTE TELL-SHOP-PROBLEM)
                  SHOP-KNOWS-PROBLEM)
       :DEL-LIST NIL)
 #S(OP :ACTION TELEPHONE-SHOP
       :PRECONDS (KNOW-PHONE-NUMBER)
```

```
       :ADD-LIST ((EXECUTE TELEPHONE-SHOP)
                  IN-COMMUNICATION-WITH-SHOP)
       :DEL-LIST NIL)
 #S(OP :ACTION LOOK-UP-NUMBER
       :PRECONDS (HAVE-PHONE-BOOK)
       :ADD-LIST ((EXECUTE LOOK-UP-NUMBER)
                  KNOW-PHONE-NUMBER)
       :DEL-LIST NIL)
 #S(OP :ACTION GIVE-SHOP-MONEY
       :PRECONDS (HAVE-MONEY)
       :ADD-LIST ((EXECUTE GIVE-SHOP-MONEY)
                  SHOP-HAS-MONEY)
       :DEL-LIST (HAVE-MONEY)))
```

Let's define our initial state of the world as follows.

```
(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).

Now everything is in place for us to 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. The function `use` is defined in `gps.lisp`. You can verify that the list `*school-ops*` contains exactly 6 operators. Calling the function `use` below tells the GPS planner which library of operators it will be working with.

```
> (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 plant (i.e., a human, a robot, a software agent, etc.) 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))
```

Once we have a plan, we can feed it to the plant's hardware one instruction at a time by iterating over its steps. Since there's no actual hardware for this example, we'll print each step instead.

```
> (dolist (plan-step son-at-school-plan)
     (print plan-step))

(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)
NIL
```

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 tree unfolding layer by layer.

As shown below (you may want to spend a few minutes analyzing the goal-operator tree), 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))
```

To turn the planning tracer off, do this.

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

Problem 2 (1 point)

The Sussman Anomaly is a very famous AI planning problem named after its discoverer Dr. Gerald Sussman. Sometimes I use this well-known AI problem in my CS5000: Computability Theory to show my students the differences between deterministic and non-deterministic finite state automata.

For those of you who are not familiar with it, imagine a simple plant whose hardware has a camera and a gripper (i.e., it's a camera-arm unit). The plant is supposed to build block towers in a simple blocks world. Obvisously, the more blocks in the world, the harder to build towers. 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`. Furthermore, `B` and `C` are clear (i.e., there is no block on either of them).

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

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

As you must have guessed, the symbols `clear-c` and `clear-b` indicate that the tops of `C` and `B` are clear and other blocks can be placed on them or they can be grabbed by the plant's gripper.

Write the operators that allow the world to build the `ABC` tower from the initial state of the block world in `*blocks-world*` and avoid the Sussman Anomaly (i.e., the `PREREQUISITE-CLOBBERS-SIBLING-GOAL` problem). I solved this problem with 3 GPS operators. Below is my run with the tracer on.

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

> (gps *blocks-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 *blocks-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))
```

Three things to note about the above run. First, the operators are defined in such a way that the order of sub-goals in the top goal is irrelevant. In other words, the planner solves the problem regardless of whether the top goal is defined as `(a-on-b b-on-c)` or `(b-on-c a-on-b)`. Second, those of you who have had the experience of defining a deterministic finite state automaton to solve this problem, should appreciate what the means-ends analysis has done for us: we define operators and then let the planner figure out a solution path for the plant, instead of hard coding all possible paths in an automaton with labeled transitions. Third, GPS can work in multiple domains and can switch from domain to domain dynamically at run time so long as it has access to appropirate operator libraries.

Problem 2 (2 points)

Here's another famous AI planning classic. It's known as the Monkey and Bananas Problem. As 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. 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 is a trace of my 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 for your operators for Problems 2 and 3. Submit your paper analysis in `gps_paper.pdf`.

Happy Reading, Writing, and Knowledge Engineering!

And, remember a fundamental takeaway of CS5600/6600: F24: Intelligent Systems:

$$\text{GPS} \neq \text{GPS}.$$

I will let you choose which side of this inequality stands for the General Problem Solver.