A Contemporary Variation on a Classic Robotics and Declarative Programming Problem

Revision 0.6

Sam Burkhart
Electrical and Computer Engineering
Portland State University
Portland, OR
skb@pdx.edu

Armaan Roshani
Electrical and Computer Engineering
Portland State University
Portland, OR
armaan@pdx.edu

David Hernandez
Electrical and Computer Engineering
Portland State University
Portland, OR
dhern2@pdx.edu

Matt Fleetwood
Electrical and Computer Engineering
Portland State University
Portland, OR
fleet@pdx.edu

Dr. Marek Perkowski
Electrical and Computer Engineering
Portland State University
Portland, OR
h8mp@pdx.edu

***Abstract*—The purpose of this project is to re-create a classic experiment in robotics and Artificial Intelligence (A.I.) known informally as the monkey and banana experiment. Here, we detail how Prolog can be integrated into Python in order to conduct this experiment. Prolog is used as a representation for the robot's world knowledge and as a means to determine what the robot needs to do next. Python is used to query Prolog. A vision system using OpenCV and Aruco markers determines where relevant objects (e.g. the soda container, the ramp, the box, etc.) are located. Python uses this information to query a Prolog program, which provides the direction (such as left, right, backwards, forwards, and so on) the robot should take in order to acquire the goal object. In this case, the goal is a tall soda container. We used Python sockets for sending commands to the robot.**

***Index Terms*—OpenCV, Aruco, computer vision, Prolog, Python, sockets, client, server, declarative programming, Object-Oriented Programming, robotics**

I. Introduction

Traditionally there are at least two significant distinctions between programming languages: declarative and imperative. Prolog, or Programming Logic, is used as a declarative language. In contrast, Python for example can be used in an imperative style. This means instead of describing what the program should do to solve a problem, a description of how the program should solve the problem is provided instead. For instance, using a declarative approach to describing an autonomous robot, one could say the robot avoids obstacles. On the other hand using an imperative description, one could say the robot uses a sonic sensor to detect if an object is within 15 centimeters or less, otherwise it continues along its current path. Declarative descriptions describe what a phenomena does while imperative descriptions describe how the phenomena occurs. Prolog can also be described procedurally, which ultimately specifies how Prolog answers questions the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program [Braitko]. A more detailed discussion on the procedural and declarative aspects of our Prolog program is described below in the Design and Experiment section.

Prolog can be treated as a database that is queried from Python. Pyswip is an existing Python 2 standard for handling this interface. From the Python side of things, the Prolog database is instantiated through an object or Prolog class. That is, to instantiate the Prolog class in Python as the variable prolog, the instruction is:

$$prolog = Prolog() \tag{1}$$

This can be seen on line 89 in the sudoko.py program, which is posted on our GitHub. The major differences across different applications in this regard are two-fold:

- The filename will typically be different for different applications, and will need to be modified in Python.
- The handling of query results will also vary for distinct applications, since for one query it might make sense for Python to simply check the result as a bool (true or false), to check if the result is a string ('turn left 5 degrees'), or to check if there are multiple solutions (e.g. "John is the father of Mary; John is the father of Jane", etc.).

Aside from the two differences listed above, the use for Pyswip should be exactly the same across different applications that interface Prolog from Python. For instance, consider the sudoku.py and sudoku.pl programs in the Pyswip GitHub repository. This example shows how one can provide a legal sudoku puzzle in Python and then query the Prolog program in order to solve the puzzle. Other than creating and querying the

Prolog object and database, Python simply prints a nice picture (using pretty print) of the solution to the puzzle. The result of the example puzzle in the provided source code is shown below in Figure 1. This example demonstrates a simple test to ensure Prolog is working correctly when it is called from Python. Python 3 was used in the Anaconda environment and the Spyder IDE (Integrated Development Environment).

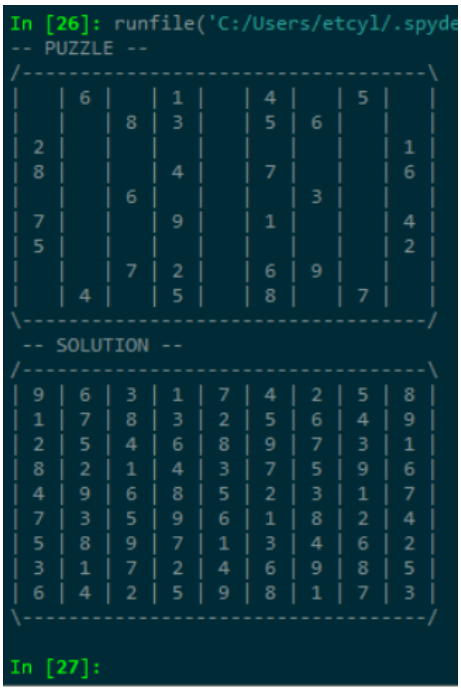


Fig. 1: Input and output sudoku puzzle using Prolog from Python.

II. Description of Prolog Backtracking

A. *Family Tree Program in Prolog*

Prolog is a powerful declarative language because of its inherent ability for backtracking. As an example, a simple trace output for the family tree program is shown in Figure below. (The complete Prolog program can be found on our GitHub.) For brevity, only one rule is investigated to demonstrate backtracking in this example - the related rule.

```
related(X, Y) :-
    parent(Y, X);
    grandparent(X, Y);
    grandparent(Y, X);
    greatgrandparent(X, Y);
    greatgrandparent(Y, X);
    sibling(X, Y);
    aunt(X, Y);
    aunt(Y, X);
    uncle(X, Y);
    uncle(Y, X);
    sibling(Y, X).
```

Fig. 2: Family tree program rule for defining if two people are related.

This rule says X and Y are related if any of the following are true: Y is a parent of X; Y is a grandparent of X, Y is a sibling of X; and so on (the opposite is also true: if X is a parent of Y, X and Y are also related). In Fig. 3 , Prolog repeatedly tries to prove X and Y are related by using the definition of the rules. For instance, Prolog tries to see if matthew is related to tim by first checking if tim is a parent of matthew. Some X is a parent of some Y if X is a mother or a father of Y. The parent rule is checked first because it is defined as the first goal with a semicolon. The semicolon indicates to Prolog that if this rule succeeds, then the rest of the rules (grandparent, greatgrandparent, sibling, etc.) for the related predicate are not attempted. Prolog backtracks from a rule if it fails, trying the parent rule, then the grandparent rule, and finally the uncle rule, which succeeds. While backtracking, Prolog references other possible terms by using the underscore NUM notation, where NUM is some constant value like 2708. This means Prolog is looking for some term that makes a predicate true. For instance, Call (11) sibling(matthew, 2708) ? creep means Prolog is attempting to unify some term with the sibling predicate and the term matthew. In other words, its checking to see if there is a person who is defined as being the sibling of matthew. Fig. 4 below shows a graphical representation of Prolog processing a query, which uses unification and resolution.

```
?- consult(family_tree).
true.

?- trace.
true.

[trace]  ?- related(matthew, tim).
   Call: (8) related(matthew, tim) ? creep
   Call: (9) parent(tim, matthew) ? creep
   Call: (10) mother(tim, matthew) ? creep
   Fail: (10) mother(tim, matthew) ? creep
   Redo: (9) parent(tim, matthew) ? creep
   Call: (10) father(tim, matthew) ? creep
   Fail: (10) father(tim, matthew) ? creep
   Fail: (9) parent(tim, matthew) ? creep
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) grandparent(matthew, tim) ? creep
   Call: (10) parent(matthew, _2708) ? creep
   Call: (11) mother(matthew, _2708) ? creep
   Fail: (11) mother(matthew, _2708) ? creep
   Redo: (10) parent(matthew, _2708) ? creep
   Call: (11) father(matthew, _2708) ? creep
   Fail: (11) father(matthew, _2708) ? creep
   Fail: (10) parent(matthew, _2708) ? creep
   Fail: (9) grandparent(matthew, tim) ? creep
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) grandparent(tim, matthew) ? creep
   Call: (10) parent(tim, _2708) ? creep
   Call: (11) mother(tim, _2708) ? creep
   Fail: (11) mother(tim, _2708) ? creep
   Redo: (10) parent(tim, _2708) ? creep
   Call: (11) father(tim, _2708) ? creep
   Fail: (11) father(tim, _2708) ? creep
   Fail: (10) parent(tim, _2708) ? creep
   Fail: (9) grandparent(tim, matthew) ? creep
   Redo: (8) related(matthew, tim) ? creep
   Call: (9) greatgrandparent(matthew, tim) ? creep
   Call: (10) grandparent(matthew, _2708) ? creep
   Call: (11) parent(matthew, _2708) ? creep
   Call: (12) mother(matthew, _2708) ? creep
   Fail: (12) mother(matthew, _2708) ? creep
   Redo: (11) parent(matthew, _2708) ? creep
   Call: (12) father(matthew, _2708) ? creep
   Fail: (12) father(matthew, _2708) ? creep
   Fail: (11) parent(matthew, _2708) ? creep
   Fail: (10) grandparent(matthew, _2708) ? creep
   Fail: (9) greatgrandparent(matthew, tim) ? creep
   Redo: (8) related(matthew, tim) ? creep
```

Fig. 3: Trace output for the family tree program.

```
Redo: (8) related(matthew, tim) ? creep
Call: (9) uncle(tim, matthew) ? creep
Call: (10) brother(tim, _2708) ? creep
Call: (11) male(tim) ? creep
Exit: (11) male(tim) ? creep
Call: (11) sibling(tim, _2708) ? creep
Exit: (11) sibling(tim, victoria) ? creep
Exit: (10) brother(tim, victoria) ? creep
Call: (10) parent(victoria, matthew) ? creep
Call: (11) mother(victoria, matthew) ? creep
Exit: (11) mother(victoria, matthew) ? creep
Exit: (10) parent(victoria, matthew) ? creep
Exit: (9) uncle(tim, matthew) ? creep
Exit: (8) related(matthew, tim) ? creep
true .

[trace]  ?-
```

Fig. 4: Trace output showing successful unification of the related rule.

B. The Original Banana and Monkey Experiment by Braitko

The banana and monkey experiment is discussed in the Braitko text, Chapter 2 page 49. As the author notes, this is a simple example of problem solving to show how the mechanisms of matching and backtracking can be used in Prolog [Braitko]. In the variation described by Braitko, a monkey standing in a room needs to grab a hanging banana, which is suspended somehow from the ceiling. Since the monkey cannot reach the banana, it needs to locate a box in the room that can be pushed below the banana such that the monkey can climb on the box. Once standing on the box, the monkey can grab the banana, which represents the goal of this scenario.

Braitko describes how the world state of this example can be viewed. For instance, "the initial state of the world is determined by:

- Monkey is at door.
- Monkey is on floor.
- Box is at window.
- Monkey does not have banana."

Similarly, only "four types of moves are allowed:

- Grasp banana.
- Climb box.
- Push box.
- Walk around.

The complete program for this example can be found in the appendix of this report, or online in the PDF here. Fig. 5 procedurally (or imperatively) describes how Prolog solves a query. A tree is shown to graphically represent this description. For example, at the root query, the monkey is at the door on the floor, the box is at the window, and the monkey does not have the banana.

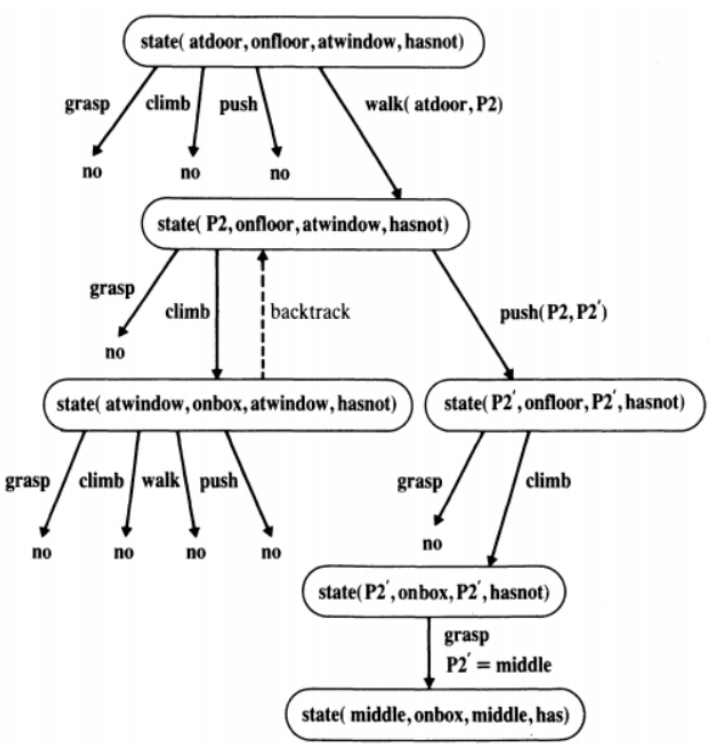
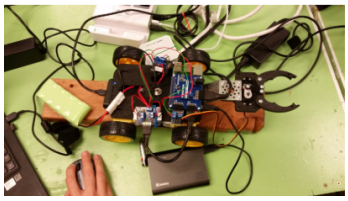


Fig. 5: Original Prolog search chart from Chapter 2 in Braitkos book, *Prolog Programming for A.I.*

III. Donkey Car and Claw Robot: Object-Oriented Programming

There are two robots used for this project. One, called the Donkey Car, is shown below in Fig. . The other is called the Claw Robot. It is shown below in Fig 7. Both robots are mobile and utilize a RaspberryPi 2. They where written in Python to facilitate a relatively simple (OOP) class interface, which ultimately means that commands are easily performed in functions such as turn left, turn right, etc.



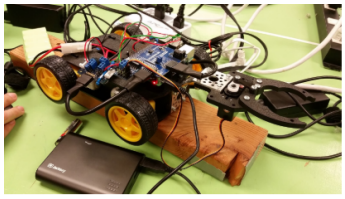


Fig. 6: Images of the claw robot.

IV. Aruco Markers in OpenCV using Python

Aruco markers provided a way to determine presence and orientation of ID (identification) markers. (More details on calibration, pros and cons, examples of using Aruco, etc.). An image of Aruco markers used for the can and robot are shown in Fig. 7.

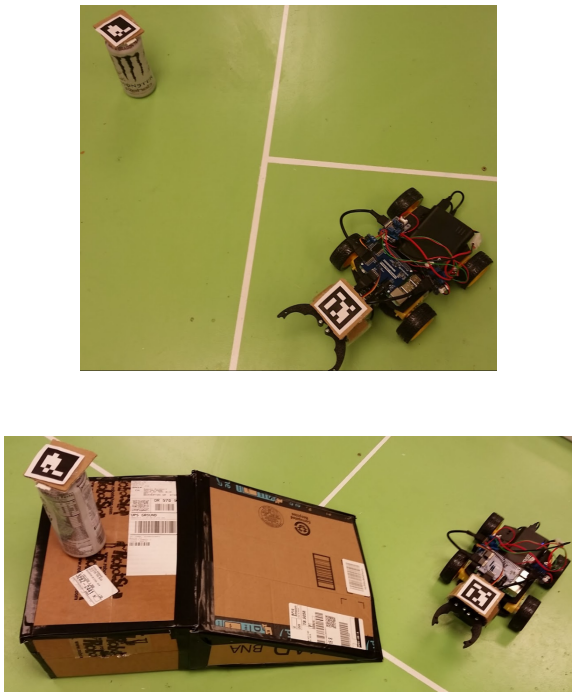


Fig. 7: Images of Aruco markers for the can and robot.

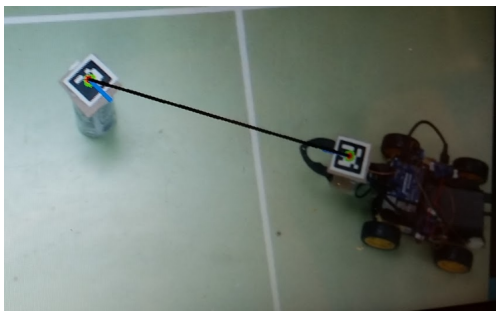


Fig. 8: OpenCV video output showing the vector from the robot's Aruco marker to the can's Aruco marker.

V. Design and Experiment

We attempt to re-create a variation of the original banana and monkey experiment by using the Donkey Car and Claw robots, a box, a ramp, Prolog for the hierarchy of rules, and Python for I/O (Input/Output). A camera vision system uses OpenCV in Python and is described further below.

A. Project Flowchart and Description

Our project flowchart can be seen below in Figure 1. The system begins execution with reading from a sensor, in this case an USB camera. The image stream from the camera is then fed into a classification module within Python. This module uses Arcuo markers in order to determine whether there are any object matches in the frame. That is, the module checks if any known objects, such as the tall soda can, the robot, the ramp, or the box in our case, are present in the image frame. The orientation in degrees of any detected objects is also provided. Essentially, this provides the knowledge of where objects are (if any are detected) and what direction they are facing.

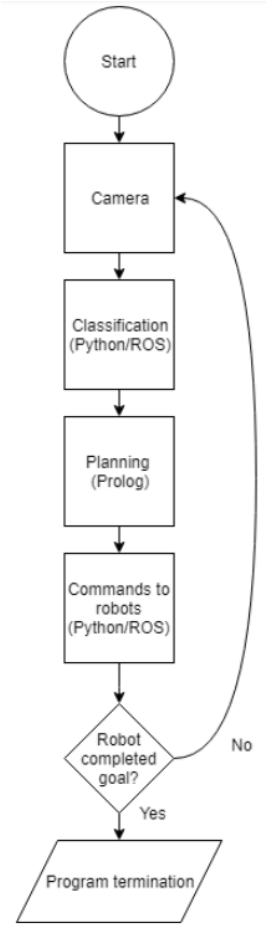


Fig. 9: High-level flowchart for the Python and Prolog system.

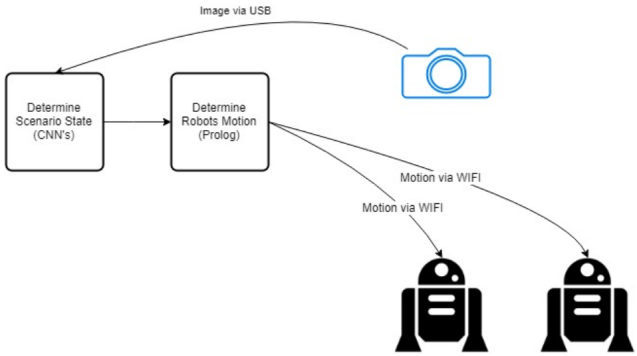


Fig. 10: Black-box for the Python and Prolog system.

This information is then used by the planning module within Prolog. Python queries Prolog using the Aruco information by asking where the robot should go next given the current positions and angles of detected objects. The planning module first checks if the robot has met the goal of the query. This means if the robot is at the goal, i.e. the soda can, then the planning module returns false because there is nothing further to do. Otherwise, the robot has not yet achieved the goal.

Once the planning is conducted in Prolog, Python receives the result of the query and checks what to do next. If the module returns false, then the program terminates because the robot has achieved the goal. If not, then the planning module tells Python where the robot should move next. For instance, a result query could be, 500, 60, which means the robot needs to move 500 centimeters after turning 60 degrees counterclockwise from its current position. The commands are sent from one computer, in our case a laptop computer (include system specifications?). This is achieved using Python sockets. The computer acts as the client by sending commands to the robots, which act as the server. In terms of ROS topics, the robots are listening to the topic, and the computer is sending commands or publishing to this topic. The robots then simply use a Python program to check and decode commands. These commands include simple movements such as left, right, forwards, and backwards.

Our design ideas can be split into the following phases below:

- Phase 1: getting robots to specific locations on map.
  - Robot Motion controlled using client-server setup for one robot.
  - Identify different objects using Aruco Markers.
  - Mapping Directions from robot to object using Prolog.
  - Writing master program using Python.
- Phase 2: making robots adjust objects.
  - Robot motion - Setup second robot for client-server.
  - Identify orientation of objects.
  - Mapping directions to adjust objects to correct orientation.
  - Update master program to use new features.
- Phase 3: robots will be able to adjust and move objects to reach locations on map.
  - Grabbing and moving objects.
  - Identify orientation of objects.
  - Identifying objects and their orientation.
  - Update master program algorithms for multi-tasking.

VI. Results

The results of this experiment show at least one contemporary approach to solving the classic robotics problem of the banana and monkey. All designs, documents, and source code and be found on out GitHub here. All video footage can be found here

Acknowledgment

Thank you Marek Perkowski for lab use and oversight of this project.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, “Title of paper if known,” unpublished.

[5] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.