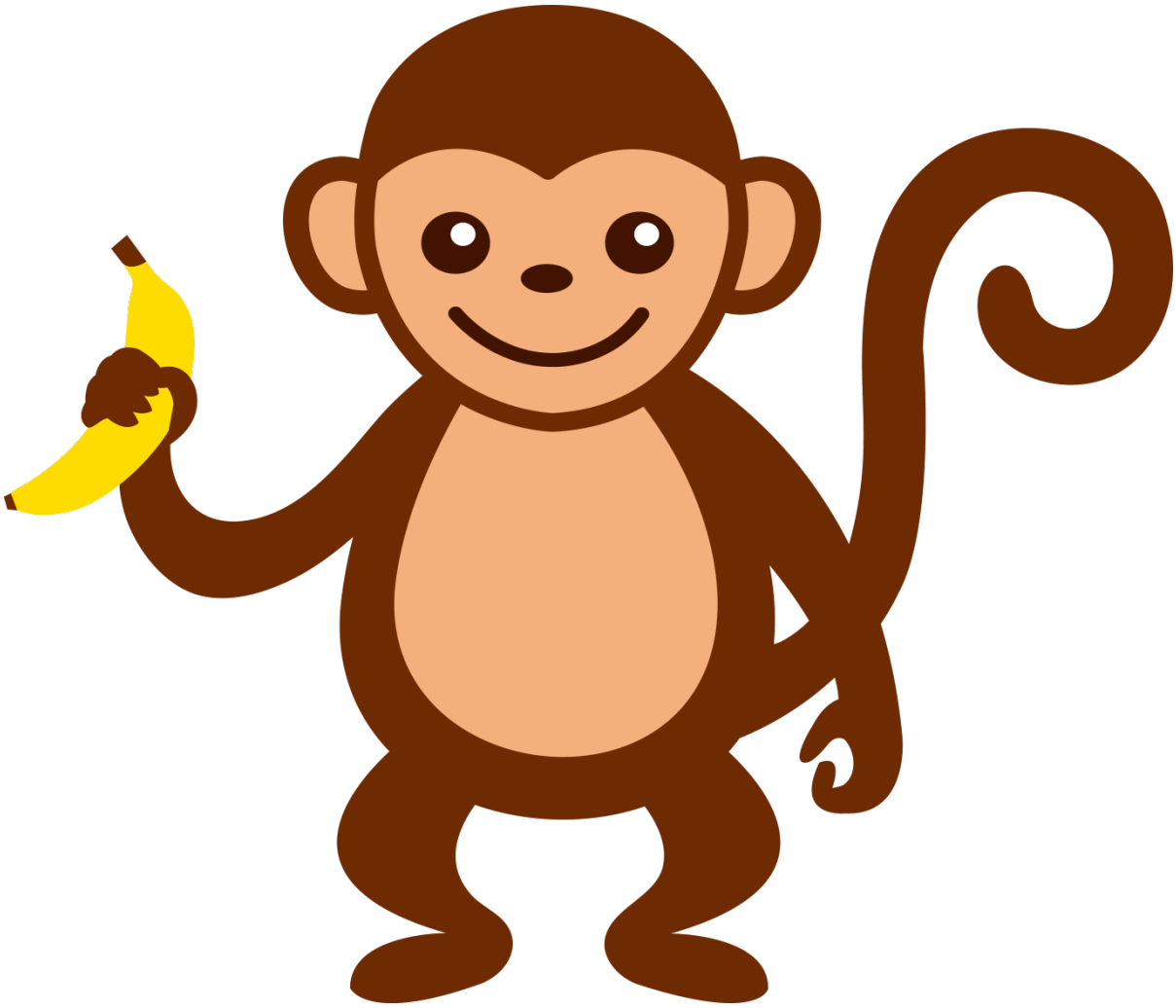
Monkey and Banana

Problem Solving with Search



This tutorial has been developed with reference to the following sources:
"Prolog Programming for Artificial Intelligence" (3rd Edition) by I. Bratko, Addison-Wesley, 2001.

REPRESENTING A PROBLEM IN PROLOG

The most important task one needs to undertake when trying to get the computer to solve a problem is to represent the problem in an appropriate way. In conventional systems, it is usual to work out the solution to the problem yourself, then describe the steps involved to the computer using a standard programming language such as C. This activity is called **procedural imperative programming**.

When developing "intelligent" computer systems, the aim is to provide the computer with the knowledge to solve the problem and allow its own inferencing capability to find the solution. This means that the real task now becomes to represent the knowledge appropriately using a language such as Prolog. This activity is called **declarative programming**.

THE "MONKEY AND BANANA" PROBLEM

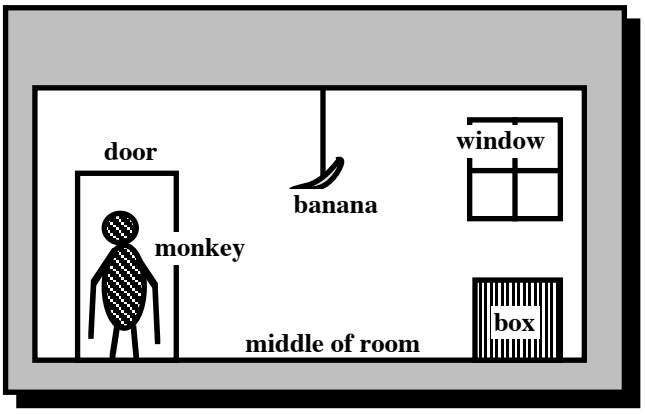
To illustrate how this can be done, we will use a classical AI problem (the Monkey and Banana problem) which is an extremely simple version of the sort of common-sense reasoning problems that will regularly be faced by "intelligent" robots in the future.

There is a monkey at the door into a room. In the middle of the room is a banana hanging from the ceiling. The monkey is hungry and wants the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey can use. The monkey can perform the following actions:

- walk on the floor;
- climb the box;
- push the box around (if at the box); and
- grasp the banana (if on the box directly under the banana).

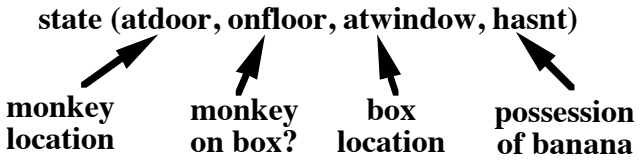
To get a handle on the problem, think of the monkey world as being in some state, that can change with time. The current state is determined by the positions of the objects. For example the initial state is determined by:

1. Monkey is **at door**.
2. Monkey is **on floor**.
3. Box is **at window**.
4. Monkey **hasn't** the banana.



THE PROBLEM STATES

Let us represent each possible state the monkey world can be in by using the functor **state** to combine the four pieces of information that determine the state. In essence we have a frame to represent the problem state. So the initial state can be represented by:



The desired state (or goal state) would be any state which looks like this:-

state(_, _, _, has).

Remember that the underline means "don't care" so it does not matter what appears here.

The other possible components of the problem state would be:

- the monkey at the window;
- the monkey in the middle of the room;
- the box in the middle of the room;
- the box at the door; and
- the monkey on the box.

1. Use these suggestions and your own commonsense understanding of the problem to write down some of the possible problem states (say about five) that would occur in the problem space. **Don't type these states into the computer**.

THE MOVES

Now we have to determine what are the allowed moves which can change the monkey world from one state to another. There are four types of moves:

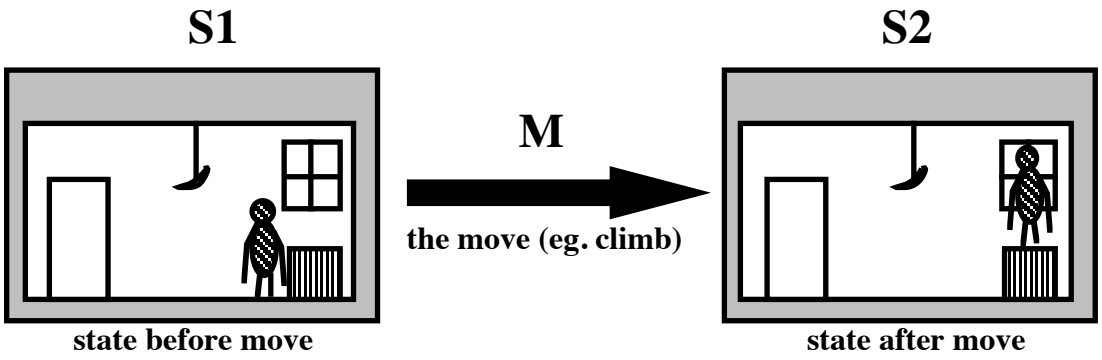
1. **grasp** banana.
2. **climb** box.
3. **push** box.
4. **walk** around.

Not all moves are possible in every possible state of the world. For example, the move grasp is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet. Such rules can be represented in Prolog as a predicate called **move** with three arguments:

move(S1, M, S2)

This predicate summarises all the information that is relevant about this move.

- **S1** is the state before the move;
- **M** is the name of the move being executed; and
- **S2** is the state after the move.



For example, the move **grasp** can be defined by the following Prolog clause:

```
move(   state(middle, onbox, middle, hasnt),      /*State before move*/
        grasp,                                    /*Name of move*/
        state(middle, onbox, middle, has)).       /*State after move*/
```

This fact states that there exists a move called **grasp**, which can only be made if the monkey is **on the box** in the **middle** of the room and **doesn't have** the banana and which, when completed, will result in the monkey still being on the box in the middle of the room, but now it **has** the banana.

In a similar way we can express the fact that the monkey on the floor can **walk** from any horizontal position P1 to any other position P2. The monkey can do this regardless of the position of the box and whether it has the banana or not. All this can be defined by the Prolog fact:

```
move(   state(P1, onfloor, B, H),
        walk(P1, P2),                             /*Walk from P1 to P2*/
```

state(P2, onfloor, B, H)).

This clause says many things, including:

- the move involves a walk from position **P1** to position **P2**;
- the monkey is **on the floor** before and after the move;
- the box is at a point **B** which is the same after the move; and
- the has-banana status **H** remains the same after the move.

This clause actually specifies a whole set of moves, because it is applicable to any situation that matches the specified state before the move.

The two other moves, **push** and **climb** can be similarly specified.

The monkey, provided it is **on the floor** and is in a position **P1** which is the same as the box position (**P1**), can **push** the box from **P1** to another position **P2**. After it has done this both monkey and box will be at position **P2** and the **has-banana** status of the monkey will not have altered.

2. Represent the move **push from P1 to P2** as a Prolog fact, as was done for the walk move.

Finally, the monkey, provided that it is **on the floor** and is at a position **P** which is the same as the box, can **climb** onto the box. The monkey will then be **on the box**, but the positions of the box and the monkey and the has-banana status of the monkey will not have altered.

3. Represent the move **climb** as a Prolog fact as was done for the grasp move.

CAN THE MONKEY GET THE BANANA?

Remember that Prolog is a declarative language. It enables you to describe the knowledge relevant in a particular situation; then it is able to answer questions posed to it.

In this case the question is:

Given an initial starting state (**S**), can the monkey reach the banana?

This question can be represented by the predicate **canget (S)** which has the value **true** if the monkey **can get** the banana from state **S**. The task would then be to specify a particular starting state and see if this predicate is true or false.

We still need some more knowledge before such questions can be answered. For any state in which the monkey already has the banana, the predicate must be true. This can be represented by the fact:

```
canget(state( _, _, _, has)).
```

In other states one or more moves may be necessary. The monkey can get the banana from any state **S1** if there is some move **M** from state **S1** to some other state **S2**, such that the monkey can then get the banana from state **S2**. This can be represented by the following recursive rule:

```
canget(S1) :- move(S1, M, S2), canget(S2).
```

HOW PROLOG SOLVES THE PROBLEM

We have now provided the computer with all of the relevant knowledge about the monkey "world". Let us see how it can solve a particular problem posed to it.

The recursive **canget(S1)** rule drives the problem-solving process in the following way. Say we ask if the monkey can get the banana from the initial state (ie. at door, on floor, with box at window and without banana) by typing in the query:

?- canget(state(atdoor, onfloor, atwindow, hasnt)).

Prolog tries to answer this question by matching the query to a fact or rule in the knowledgebase. It first attempts to match it to the:

canget(state(_, _, _, has)).

fact but that won't match because the monkey hasn't yet obtained the banana.

It can however match the head (or conclusion) of the **canget (S1)** rule, provided **S1** is given the value **state (atdoor, onfloor, atwindow, hasnt).**

For this conclusion to be true however, both premises of the rule must be true. Prolog now tries to match the first premise, namely:

move (S1, M, S2)

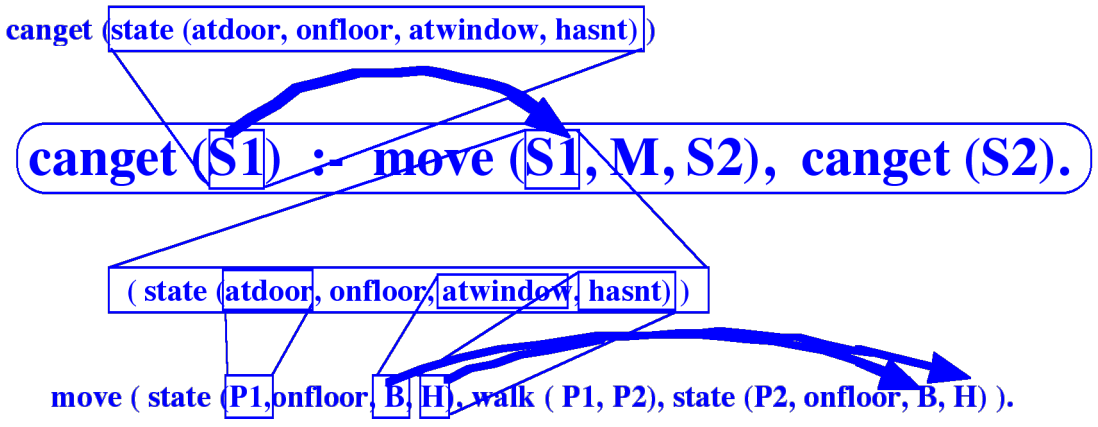
which will match the following fact, representing the **walk** move:

move (state(P1, onfloor, B, H), walk (P1, P2), state(P2, onfloor, B, H)).

provided:

S1 = state(P1, onfloor, B, H) with **P1 = atdoor, B = atwindow** and **H = hasnt**
M = walk(P1, P2) where we already know that **P1 = atdoor**
S2 = state(P2, onfloor, B, H) where we already know that **B = atwindow** and **H = hasnt**

This multi-level matching process can be illustrated diagrammatically as follows:



Note that, in the process of matching the move (S1, M, S2) premise to one of the move facts in the knowledgebase, a number of variables have to be set to specific values (eg. **P1** was set to **atdoor**). Whenever some variable has been set to a particular value, that is called a **variable binding** and remains in force for the rest of the rule. In some cases (eg. with **P2**) no variable binding was needed to achieve the match, so the variable P2 remains unbound (for the present).

Prolog then tries to match the second premise, namely:

canget(S2)

keeping all of the variable bindings that it has already made, which means that:

S2 = state(P2, onfloor, atwindow, hasnt)

In effect, the robot has now "moved" or more correctly "planned its move" from the door out into the room. Note that **P2** has not yet been determined, meaning that the monkey has not yet decided exactly where in the room it wants to go.

In attempting to match this second premise, Prolog is effectively making the following query:

canget(state (P2, onfloor, atwindow, hasnt)).

which will match to the **canget(S1)** rule again, just as the original query did, but this time with:

S1 = state(P2, onfloor, atwindow, hasnt)

Prolog continues to match the **canget(S1)** rule recursively until it reaches a state in which it has the banana, at which time it will be able to match the fact:

state(_, _, _, has).

and so terminate the recursion.

RUNNING THE PROGRAM

4. Create a prolog file called **monkey.pl**. Then type in:
- the Prolog facts specifying the different moves the monkey can make;
- the canget clause specifying the states in which the monkey has the banana; and
- the recursive canget rule which allows the monkey to reach such a state.

The moves must be typed in in the following order:
1. grasp move
2. climb move
3. push move
4. walk move

This ordering is important. You will see why shortly.

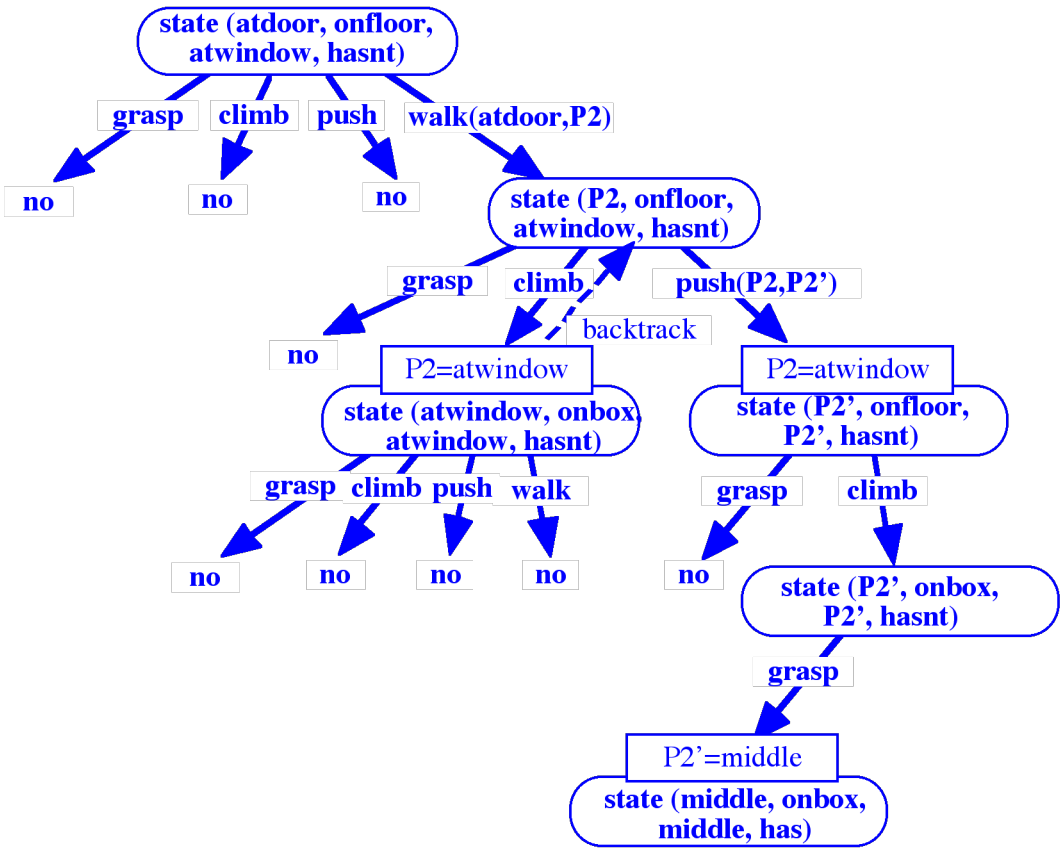
5. Consult the **monkey.pl** file in the usual way. Then pose the following question:

?- canget(state(atdoor, onfloor, atwindow, hasnt))

which basically asks if it is possible for the monkey to get the banana from the state where the monkey is at the door and standing on the floor, the box is at the window and the monkey hasn't got the banana.

Prolog's answer should be **yes**. (If Prolog seems to go into an infinite loop, you can escape from the run by pressing **Ctrl-C** and pressing **a** for **Abort**)

The complete reasoning process carried out to reach this answer is illustrated thus:



The monkey's search for the banana starts at the top node (initial state) and proceeds down the search tree, as indicated. Alternative moves are tried in **left to right** order (as defined by the top to bottom order of the Prolog clauses). At one point this search makes a wrong move, reaches a **dead-end** from which it is unable to make any move, and so needs to **backtrack** to recover.

6. To confirm that the problem is being solved in the manner indicated, insert print statements into the recursive canget rule thus:

```
canget(S1) :- move(S1, M, S2),
              write(S1), nl, write(M), nl, write(S2), nl, nl,
              canget(S2).
```

Now ask the following question again and watch the moves as they are printed out, to see if they match the diagram above:

```
?- canget(state (atdoor, onfloor, atwindow, hasnt)).
```

The solution to this problem was found very quickly, with backtracking occuring only once. However, this was lucky. If the clauses had been ordered differently the problem could have been much more difficult to solve.

7. Change the order of clauses by typing the **walk** clause at the top and deleting it from further down. Then ask the same question again. Looking at the printout will show that the monkey now walks around without getting anywhere.

Such problems occur when blind search techniques are used. The search technique used here is a **blind depth-first search**. Because the **walk** clause is now checked before the others and because it is a very general clause, which will match in most situations, there is a danger that the program will enter into an infinite recursion without ever trying the other move clauses.

In order to solve this problem, it is necessary to use heuristic knowledge about the problem-solving domain itself to assist in the reasoning process. In this case the **order** of the operators serves as an **heuristic**. Basically we are saying that, for the monkey and banana domain, we realise that the monkey should always try to grasp the banana, if it possibly can, as a first option, because that would solve the problem immediately. This means that the **grasp** clause should appear above the rest in the Prolog listing. If it can't do that, it should try to climb the box if it can and so on. The **walk** clause should be at the bottom of the listing, so that it is only tried when all other options have failed.

The recognition that general-purpose reasoning systems which took no account of the specific characteristics of a problem domain were unlikely to be practicable for realistically-sized problems was a major milestone in the history of AI research. It led to the subsequent development of domain-specific AI programs which we now call **expert systems**.

MONKEY AND BANANA QUESTION SHEET

STUDENT NAME .....

Question 1.
The monkey is getting bored with the "Banana Game" and decides to confuse the scientists by getting back off the box and putting the box back under the window after getting the banana. Define a new **climbdown** operator which will allow it to get back off the box, once it has the banana. Write down the Prolog representation of the **climbdown** move operator.

.....

.....

.....

.....

To distinguish it from the climbdown operator, and to indicate that it should only be used when the monkey does not have the banana, the climb procedure will also need to be modified. Change the name of the modified operator to **climbup** and write down the Prolog representation of the **climbup** move operator.

.....

.....

.....

.....

Question 2.
Now modify the goal state so that the **box** would be left back **at the window** after the monkey's plan has been carried out. Write down the Prolog clause representing the new goal state.

.....

Now type in the same Prolog query as before (this time asking if the monkey can get the banana, and confuse the scientists, when it is starting at the door and is on the floor without the banana, and the box is under the window). What is this **query**, as written in Prolog, and what is the **answer** to the query ?

.....

Write down the **steps** (as printed out by Prolog).that the monkey now plans to follow in order to achieve its goal of getting the banana but also confusing the scientists.

.....

.....

.....

.....

.....

.....

.....

The **monkeypro** system is a very simple **planning** system, based on the standard **problem space approach** , and is typical of the earliest work done in AI planning. Since then, **planning** has developed into a distinctive sub-field within AI with specific techniques being developed to deal with the complexities of **real-world planning environments**.

Recent work in **planning**, particularly in **robotics**, takes into account the **time it takes** to carry out a planning operation and the possibility that the **world** itself **may change** while the plan is being created or executed. Such planning algorithms need to **monitor** the environment continually, using **sensors,** and must be prepared to **re-plan** if the sensors indicate that changes have occurred in the world before the plan has been completely executed.