**The laboratory work 13**

**Creating facts for the Wumpus world**

We need to build an agent that is able to move safely through its world, collect gold, kill the Wumpus and then exit. Given that you are starting with no knowledge of the world, you are going to have to have your agent figure things out as it goes along. You will need to build up a set of inference rules that will fire as new facts are known. For example, if you are on a square with no breeze then you can add this fact to the data, add facts about what square are adjacent to it and then infer that those squares have solid floors.

That is, ASSERT: Calm(Cell 11), Adjacent(Cell 11, Cell 12), Adjacent(Cell 11, Cell 21)

Then use a rule like: Calm(?x) V Adjacent(?x ?y) => Solid(?y)

You need to use your knowledge base to figure out which squares have solid floors and no Wumpus, are safe (solid floor and no Wumpus), and where the Wumpus is so you can shoot it. As you wander, you should probably also be thinking about how to save information about what constitutes as a good path back to the start. The approach we are taking is to use a knowledge base as the repository of what is true (i.e. known by the system) and another system (a planner) to make decisions based on what it knows and what it wants to accomplish.

The knowledge side of things requires a repository of facts and a repository of rules. As new rules are added, we check to see if they allow you to infer anything given the fact that you know. As you add new facts, you need to check to see if any rules can be fired.

In order to access this information, you will need to be able to:

**ASSERT** facts and rules. (This room has no breeze).

**ASK** if things you care about are true. (In this room I want to go into Safe?)

The agent should have specific goals it is trying to accomplish (pick up the gold) and know that it will have to take actions in order to set up the conditions that makes this possible (stand in the same room as the gold).

The challenge is that you will need to be able to not just add facts but also draw inferences from them. Your inference rules will need to have variables so that they are general. You will need to be able to MATCH facts against patterns in your inference rules. When you draw an inference, you will need to INSTANTIATE the result and thus create a new fact with all of the variables replaced with the elements from the fact that was matched against it.

The code for matching and instantiating is represented below.

This is the file FOPC.py

|  |
| --- |
| # arguments that the predicate is applied to.  #  # e.g. ('clean', 'Cell 11')  #  # A pattern is the same thing, except in that a pattern might include variables  #  # In this instance, variables are just strings with a "?" as the first character.  #  # When a statement matches a pattern, it is within the context of a set of bindings.  #  # e.g. when ('clean', 'Cell 11') matches ('clean', '?x") the result is  # ('?x': 'Cell 11')  #  # when they don't match, the result is just False  #  # The bindings list can then be used to set the context of later matches  # Just test to see if something is a variable  def is\_variable(x):  return x[0] == "?"  # Match a statement by matching its elements where each match produces a potentially increasing  # set of bindings that are passed down to constrain the rest of the matches in the statement  def match(statement, pattern, bindings):  for e,p in zip(statement,pattern):  bindings = element\_match(e, p, bindings)  if bindings is False:  return False  return bindings      # Elements are matched in the contest of a set of existing bindings.  # if the two elements match, then just return the current bindings  # if the pattern element is a variable, test to see if it is bound.  # if it is bound, use the binding to see if there is a match  # if it is not bound, bind it to the current statement element, add the binding to bindings list and return the  # expanded bindings list    def element\_match(e, p, bindings):  if e == p:  return bindings  elif is\_variable(p):  if bindings.get(p):  p = bindings.get(p)  return element\_match(e,p,bindings)  else:  bindings[p] = e  return bindings  else:  return False  # bind takes an element and binding list. If the element is a variable that is bound,  # it returns the element it is bound to. Otherwise, it just hands back the element    def bind(x, bindings):  return bindings.get(x,x)  # instantiate takes a full statement and a bindings list. it binds all variables in the  # statement and returns it.    def instantiate(statement, bindings):  return map(lambda(x): bind(x, bindings), statement) |