Assignment3 – Agents and Search

Problem 3.1

For each of the following *agent architectures*, develop a *PEAS* description of the task *environment*.

Additionally, characterize the *environments* of these *agents* according to the properties discussed in the *lecture*. Where not "obvious", justify your choice with a short sentence.

Finally, choose suitable designs for the agents.

1. Robot soccer player
2. Internet book-shop agent (that is: an agent for book shops that stocks up on books depending on demand)
3. Autonomous Mars rover
4. *Mathematical* theorem prover
5. First-person shooter (Counterstrike, Unreal Tournament etc.)

Problem 3.2

Explain the difference between agent function and agent program. How many agent programs can there be for a given agent function?

Solution: The function specifies the input-output relation (outside view). The program *implements* the function (inside view).

The function takes the full sequence of percepts as arguments. The program uses the internal state to avoid that.

There are either none or infinitely many programs for a function.

Problem 3.3

Explain the commonalities of and the differences between the performance measure and the utility function.

Solution: Both measure how well an agent is doing.

The performance measure is a meta-level object that defines the quality of any agent used to solve the task. It may be defined informally (but still precisely) because it only needs to be used by an outside observer, such as a human comparing multiple agents.

A utility function is a component of a particular utility-based agent. It must be defined formally (e.g., in a specification or programming language) because it must be computed as a part of applying the agent.

Problem 3.4 (Tree Search in ProLog)

Implement the following tree search algorithms in *Prolog*:

1. *BFS*
2. *DFS*
3. *IDS*Iterative Deepening (with step size 1)

Remarks:

- In the *lectures*, we talked about *expanding* nodes. That is relevant in many AI applications where the tree is not built yet (and maybe even too big to hold in memory), such as game trees in move-based games or decision trees of agents interacting with an environment. In those cases, when visiting a node, we have to expand it, i.e., compute what its children are.

  In this problem, we work with smaller trees where the search algorithm receives the fully expanded tree as input. The algorithm must still visit every node and perform some operation on it — the search algorithm determines in which order the nodes are visited.

  In our case, the operation will be to *write out the label* of the node.

- In the *lectures*, we worked with goal nodes, where the search stops when a goal node is found. Here we do something simpler: we *visit all the nodes and operate on each one* without using a goal state. (Having a goal state is then just the special case where the operation is to test the node and possibly stop.)

Concretely, your submission **must** be a single *Prolog* file that extends the following *implementation*:

```
% tree(V,TS) represents a tree.
% V must be a string - the label/value/data V of the root node
% TS must be a list of trees - the children/subtrees of the root node
% In particular, a leaf node is a tree with the empty list of children
istree(tree(V,TS)) :- string(V), istreelist(TS).

% istreelist(TS) holds if TS is a list of trees.
% Note that the children are a list not a set, i.e., they are ordered.
istreelist([]).
istreelist([T|TS]) :- istree(T), istreelist(TS).

% The following predicates define search algorithms that take a tree T
% and visit every node each time writing out the line D:L where
% * D is the depth (starting at 0 for the root)
% * L is the label

% dfs(T) visits every node in depth-first order
dfs(T) :- ???
% bfs(T) visits every node in beadth-first order
bfs(T) :- ???
%itd(T):- visits every node in iterative deepening order
itd(T) :- ???
```

Here “must” means you can define any number of additional predicates. But the predicates specified above must exist and must have that arity and must work correctly on any input `T` that satisfies `istree(T)`. “working correctly” means the predicates must write out exactly what is specified, e.g.,

```
0:A
1:B
```

for the depth-first search of the tree `tree("A",[tree("B",[])])`.

Solution:

```
% initialize with depth 0
dfs(T) :- dfsD(T,0).

% write out depth and value V of the current node, then search all children with depth D+1
dfsD(tree(V,TS), D) :- write(D), write(":"), writeln(V), Di is D+1, dfsAll(TS,Di).

% calls dfsD on all trees in a list
dfsAll([],_).
dfsAll([T|TS],D) :- dfsD(T,D), dfsAll(TS,D).

% initialize with the fringe containing T at depth 0
bfs(T) :- bfsFringe([(0,T)]).

% empty fringe - done
bfsFringe([]).
% take the first pair (D,T) in the fringe, write out D and the value V of T
% append children TS of T paired with depth D+1 to the *end* of F, and recurse
bfsFringe([(D,tree(V,TS))|F]) :- write(D), write(":"), writeln(V),
   Di is D+1, pair(Di,TS, DTS), append(F,DTS,F2), bfsFringe(F2).

% pair(D,L,DL) takes value D and list L and pairs every element in L with D, returning DL
pair(_,[],[]).
pair(D,[H|T],[(D,H)|DT]) :- pair(D,T,DT).

% initialize with cutoff 0
itd(T) :- itdUntilDone(T,0),!.

% calls dfsUpTo with cutoff C and initial depth
itdUntilDone(T,C) :- dfsUpTo(T,0,C,Done), increaseCutOffIfNotDone(T,C,Done).
% depending on the value of Done, terminate or increase the cutoff.
increaseCutOffIfNotDone(_,_,Done) :- Done=1.
increaseCutOffIfNotDone(T,C,Done) :- Done=0, Ci is C+1, itdUntilDone(T,Ci).

% dfsUpTo(T,D,U,Done) is like dfs(T,D) except that
% * we stop at cutoff depth U
% * we return Done (0 or 1) if there were no more nodes to explore

% cutoff depth reach, more nodes left
dfsUpTo(_, D, U, Done) :- D > U, Done is 0.
% write data, recurse into all children with depth D+1
dfsUpTo(tree(V,TS), D, U, Done) :- write(D), write(":"), writeln(V),
   Di is D+1, dfsUpToAll(TS,Di,U, Done).

% dfsUpToAll(TS,D,U,Done) calls dfsUpTo(T,D,U,_) on all elements of TS; it returns 1 if all
dfsUpToAll([],_,_,Done) :- Done is 1.
dfsUpToAll([T|TS],D,U,Done) :- dfsUpTo(T,D,U,DoneT),
   dfsUpToAll(TS,D,U,DoneTS), Done is DoneT*DoneTS.
```