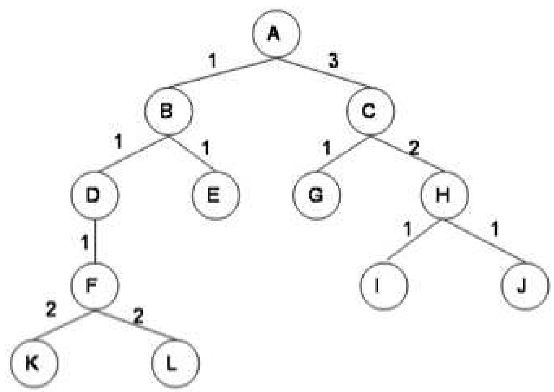
Artificial Intelligence 1
Assignment3: Search
– Given Nov. 12., Due Nov. 20. –

Problem 3.1 (Search Strategy Comparison on Tree Search) 20 pt

Consider the tree shown below. The numbers on the arcs are the arc lengths (i.e., the costs of that arc). In what order are the nodes examined by each type of search if they stop after examining the goal node I?



To enable automatic grading, when submitting your solution, upload a text file containing exactly the following:

- one line for each search algorithm in the order BFS, DFS, Iterative Deepening (with step size increasing by 1), Uniform Cost,
- in each line, the letters identifying the nodes in the correct order (upper case, no whitespace),
- if a search algorithm visits a node multiple times, every occurrence should be included,
- if there is a tie between nodes, the algorithms should examine nodes in alphabetical order.

Solution:

Search Type	List of States
Breadth First	ABCDEGHFI
Depth First	ABDFKLECGHI
Iterative Deepening	AABCABDECGHABDFECGHI
Uniform Cost	ABDECFGHKLI

Problem 3.2 (Tree Search in ProLog) 50 pt

Implement the following tree search algorithms in Prolog:

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

Remarks:

- In the lecture, 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 lecture, 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 dfsUpTp 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.
```

Problem 3.3 (Formally Modeling a Search Problem) 30 pt

Consider the Towers of Hanoi for 7 disks initially stacked on peg A.

Is this problem deterministic? Is it fully observable?

Formally model it as a Search Problem in the sense of the mathematical definition from the slides. Explain how your mathematical definition models the problem.

Note that the formal model only defines the problem — we are not looking for solutions here.

Note that modeling the problem corresponds to defining it in a programming language, except that we use mathematics instead of a programming language. Then explaining the model corresponds to documenting your implementation.

Solution: We need to give $(S, \mathcal{A}, T, I, G)$.

Because the problem is deterministic, we know $|T(a, s)| \leq 1$. Because the problem is fully observable, we know $|I| = 1$.

Let $D = \{1, 2, 3, 4, 5, 6, 7\}$ (the set of disks) and $P = \{A, B, C\}$ (the set of pegs). We put:

- $S = D \to P$, i.e., a state s is a function from disks to pegs.
  Explanation: In state s, the value $s(d)$ is the peg that disk d is on. Because disks must always be ordered by size, we do not have to explicitly store the order in which the disks sit on the pegs.

- $\mathcal{A} = \{(A, B), (B, A), (A, C), (C, A), (B, C), (C, B)\}$, i.e., an action a is a pair of different pegs.
  Explanation: (p, q) represents the action of moving the top disk of peg p to peg q.

- For $s \in S$ and $p \in P$, we abbreviate as $top(s, p)$ the smallest $d \in D$ such that $s(d) = p$.
  Explanation: $top(s, p)$ is the top (smallest) disk on peg p in state s.

  Then $T : \mathcal{A} \times S \to \mathcal{P}(S)$ is defined as follows:

  - If $top(s, q) > top(s, p)$, we put $T((p, q), s) = \{s'\}$ where $s' : D \to P$ is given by
    - $s'(d) = q$ if $d = top(s, p)$
    - $s'(d) = s(d)$ for all other values of d
  - otherwise, $T((p, q)), s) = \emptyset$

  Explanation: In state s, if the top disk of q is bigger than the top disk of p, the action (p, q) is applicable, and the successor state s' of s after applying (p, q) is the same as s except that the top (smallest) disk on peg p is now on peg q.

- $I = \{i\}$ where the state i is given by $i(d) = A$ for all $d \in D$.
  Explanation: Initially, all disks are on peg A.

- $G = \{g\}$ where the state g is given by $g(d) = B$ for all $d \in D$.
  Explanation: There is only one goal state, described by all disks being on peg B.

Note that there are many different correct solutions to this problem. In particular, you can use different definitions for S (i.e., model the state space differently), in which case everything else in the model will be different, too. Often a good model for the state space can be recognized by how straightforward it is to define the rest of the model formally.

Even if you used a different state space, a good self-study exercise is to check that the above (a) is indeed a search problem and (b) correctly models the Towers of Hanoi. Continuing the above analogy to programming languages, (a) corresponds to compiling/type-checking your implementation and (b) to checking that your implementation is correct.