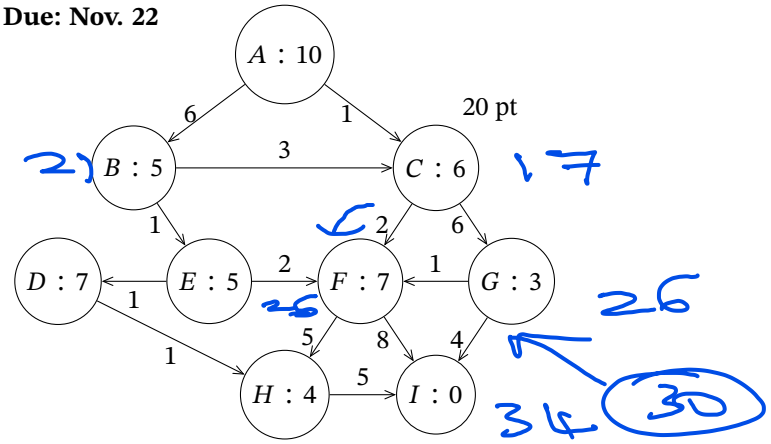
Assignment4 – Search

Given: Nov. 16. Due: Nov. 22

Problem 4.1 (Search Algorithms) 20 pt

Consider the following directed graph:



Every node is labeled with $n : h(n)$ where n is the identifier of the node and $h(n)$ is the heuristic for estimating the cost from n to a goal node. Every edge is labeled with its actual cost.

1. Assume that I is the goal node. Argue whether or not the heuristic is admissible.

Now assume you have already expanded the node A. List the **next** 4 **nodes (i.e., excluding** A**)** that will be expanded using the respective algorithm. If there is a tie, break it using alphabetical order.

2. depth-first search
3. breadth-first search
4. uniform-cost search
5. greedy-search
6. A^*-search

Problem 4.2 (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",[])])`.

Problem 4.3 (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.