# CMSC 723: Computational Linguistics I

Assignment 1: Searching Graphs & Climbing Trees

Jimmy Lin (Instructor) and Melissa Egan (TA)

Due: **September 16, 2009**

**NOTE:** This assignment is meant as a "warm-up" exercise with the goal of introducing you to Python. If these problems takes you more than a few hours, you might want to reconsider whether you have the programming background necessary for the course.

Submit your solutions by email to Melissa Egan with the subject "CMSC 723: Assignment 1" before 4:00pm on the due date.

## 1 Graphs

A directed acyclic graph (DAG) consists of a set of nodes and a set of directed arcs (also known as directed edges) between those nodes. For example, Figure 1 shows a DAG.

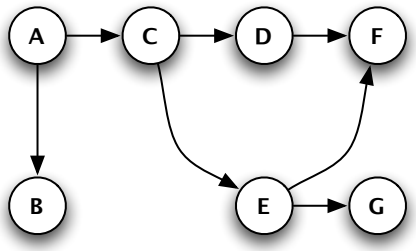


Figure 1: An example of a directed acyclic graph.

There are many ways to represent graphs. One choice is to store, for each node, a list of all the nodes it is connected to via outgoing edges. In Python, one might use a dictionary, as in Listing 1. For example, `edges['A']` is a list containing all nodes connected to node A via outgoing edges.

Listing 1: Encoding an acyclic graph with a Python dictionary

```
edges = { 'A': ['C', 'B'],
          'C': ['D', 'E'],
          'D': ['F'],
          'E': ['F', 'G'],
          'B': [ ],
          'F': [ ],
          'G': [ ] }
```

Something you do frequently with DAGs is SEARCH: start at a node, and find another node. For example, in the above graph a search starting at D for target node F would succeed, but a search starting at D for target node B would fail. Search of this kind is the heart of a great deal of work in artificial intelligence; for example, game-playing programs (such as tic-tac-toe, chess, checkers, backgammon, etc.) often view the game as a search for a winning board configuration, and parsing a sentence can be viewed as searching for a legal parse tree that fits the sentence.

There are different algorithms for searching a graph, but one of the most common ones is the depth-first search (DFS) algorithm. Algorithm 1 shows how depth-first search works on a graph.

**Algorithm 1** DFS($S, T$): An algorithm for depth-first search

**Require:** A starting node S and the target node T.

Create a list $L$, initially containing $S$ as its only member
**while** $L$ is not empty **do**
    Pop the top node $x$ off $L$ (think of $L$ as a stack)
    **if** $x$ is the target **then**
        Report **success** and quit — we found the target !
    **else**
        **for** each outgoing edge ($x$,$y$) of $x$ **do**
            Put y on the front of list $L$ (that is, push $y$ onto stack $L$)
        **end for**
    **end if**
**end while**
Report **failure** — $T$ can't be reached from $S$ !

### Problems (30 points)

1. Implement the DFS algorithm in Python. Please turn in: (a) the program listing, and (b) examples of the algorithm running on the graph in Figure 1 with three searches: the first search should start at `A` and look for `G` (success), the second search should start at `A` and look for `D` (success), and the third search should start at `C` and look for `A` (failure).

   Note: Graph information can be hard-coded in your program. However, the start node and the end node must be input as command-line parameters. For example, suppose your DFS program is called `dfs.py`, and you want to search for target node `B` starting at node `A`. Your program must work with the following command line invocation:

   ```
   dfs.py A B
   ```

2. Modify the algorithm and implementation so that on success the program returns its path through the graph (i.e., list of nodes that were examined, in order) rather than just saying it succeeded. Demonstrate using the searches from (1) above.

3. Modify the algorithm and implementation so that on success the program returns the actual path from the start node to the end node. Demonstrate using the searches from part (1) above.

   Note: Since problems (1) through (3) are closely related, you can elect to turn in a program listing that addresses all three problems if you wish. Alternatively, you can turn in a program listing for each problem.

## 2 Trees

In graph theory, a *tree* can be thought of as a simple graph in which two nodes are connected by exactly one path. For this assignment, we will confine our attentions to a *rooted tree*, a directed tree in which one node has been deemed to be the *root* of the tree and all other nodes are oriented away from the root. For such a tree, we can easily define hierarchical relationships such as parent and child. An example tree is shown in Figure 2 with node `A` as the root of the tree. In this tree, nodes `B` and `C` are the children of node `A`, etc. It also follows that a node in a rooted tree may have at most one parent node. The example graph shown in the previous section is *not* a tree since the node `F` has two parents.

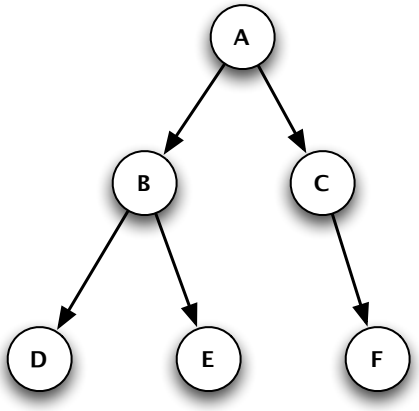


Figure 2: An example of a rooted tree.

Instead of using a simple Python dictionary for rooted trees, we have provided you with a simple library called `trees.py` that makes it very easy to create trees of the form needed for this assignment. Listing 2 shows how you would create the example tree shown here using this library.

One of the operations employed on a rooted tree is to find the *Lowest Common Ancestor (LCA)* of any two nodes in the tree. The LCA of any two nodes $u$ and $v$ is simply as the lowest (or deepest) node in the tree that has both $u$ and $v$ as descendants (where we allow a node to be a descendant of itself). For example, in our example tree, nodes `D` and `E` have `B` as their LCA. Given how we have defined a rooted tree, finding the LCA is straightforward as shown in Algorithm 2.

**Algorithm 2** LCA$(U, V)$: An algorithm to find the LCA

**Require:** A Rooted Tree T which contains the given nodes U and V.

1: Find the path $P_U$ from the node $U$ to the root of $T$. $P_U$ starts at $U$, contains all intervening nodes between $U$ and the root, and ends at the root node.
2: Find the path $P_V$ from the node $V$ to the root of $T$.
3: Find the intersection $L$ of these two paths that is furthest from the root node.
4: **return** $L$

Listing 2: Creating and Exploring a Rooted Tree

```
# import the trees library
from trees import Tree

# Create an empty tree with all the nodes but no edges.
# The first argument is a list of all nodes and the second
# says which one is the root.
t = Tree(['A','B','C','D','E','F'], 'A')

# Create a list of connections between the various nodes.
# n1->n2 means n1 is the child of n2.
connections = ['D->B','E->B','B->A','F->C','C->A']

# Make the connections
t.connect(connections)

# Print out the root node ('A')
print t.root

# Print the parent of node 'E' ('B')
print t['E'].parent
```

## Problem (20 points)

4. Extend the Trees class defined in `trees.py` with an additional method called `lca()` which takes the names of two nodes as input and returns their LCA. Please turn in: (a) the program listing, and (b) traces of your program running on a few examples.

   Note: You should try to make full use of all existing code and may implement any supporting methods that you deem necessary. However, you should *not* alter any of the existing methods in either the `Node` or the `Tree` class.