20190129

In this class we looked at a not widely-known application of D&C: determining if a tree contains a path of length k or more. (Note: the length of a path is the number of **edges** in the path.)

First, let us consider the more general problem: given a graph G and an integer k, does G contain a path of length k? We can call this the **k-path problem**.

This is the generalized version of a well-known NP-Complete problem: the Hamiltonian Path Problem, which simply asks the question "Does graph G contain a path that includes every vertex of G?" Since this problem easily reduces to k-path (**exercise: work out this extremely simple reduction)**, we conclude that there is almost certainly no polynomial-time algorithm for k-path.

However the story changes when we restrict the question to trees.

Definition: A **tree** is a connected graph with no cycles. The practical implication of this is that for any pair of vertices in a tree, there is exactly one path of edges that joins them.

In a tree, some vertices are joined by short paths and some by longer paths. If the tree represents a message passing network, the length of the longest path is of interest as it gives some kind of measure of the maximum delay for a message to travel from its source to its destination. If there is a path of length k, some messages will take k steps to get from source to destination.

Determining the existence of a path of length k (or more) is not difficult. In fact we will develop algorithms that solve a more general (and more useful) problem: **given a tree T, what is the length of the longest path in T**?

A simple Breadth-First Search or Depth-First Search algorithm can be used **n** times, each time starting at a different vertex and computing the lengths of the paths from that vertex to all others in O(n) time (this algorithm has a higher complexity on graphs that are not trees). Since in a tree there is exactly one path

between each pair of vertices, we don't have to consider alternative paths between two vertices when looking for the longest path: the length of the longest path is simply the maximum distance between any two vertices of the tree. Finding the distances from each vertex to all the others, and keeping track of the longest distance, gives a simple O(n^2) algorithm for finding the longest path in the tree: apply an O(n) search algorithm n times.

But we can do better! Not surprisingly, given the theme of this unit, we will use a Divide and Conquer approach.

Suppose we have a tree T. Choose an arbitrary vertex **x** as the root of T - we can visualize grabbing vertex x and pulling it upwards, so that the rest of the tree "hangs down" from x. Let T_1, T_2, ... represent the subtrees that hang from x.

Now we can make an observation about the longest path in T. It may seem a bit obvious but it is very useful:

Either the longest path in T goes through vertex x ...

... or it doesn't !

Trivial as it may seem, this observation motivates our algorithm. Because if the longest path in T **doesn't** go through x, then it (the path) must be completely contained in one of the T_i subtrees ... and we can find the longest path in each subtree recursively (see how craftily Divide and Conquer creeps in).

However, it is entirely possible that the longest path in T **does** go through x (remember, we chose x arbitrarily). In this case, the longest path must "come up out of" some T_i, pass through x, then go "down into" some other T_j. And since this path is the longest possible path in T, we must be using the longest path that goes from the bottom to the top of T_i, and the longest path that goes from the bottom to the top of T_j - and in fact, we must be using the longest and second longest of the available “bottom to top” paths in the subtrees.

So for each T_i, we need the length of the longest path that is completely

contained in T_i, and we also need the length of the longest path that goes from the bottom of T_i to the top of T_i. From this information, we can compute the length of the longest path in T.

The following algorithm is fully recursive. Given a tree T and the root vertex of T, it computes and returns the two values just described. I'm using sligthly different notation than I used in class, but the logic is identical (and more detailed here):

```
Max_Tree_Path(T, x)    # x is the root of T (T may be a subtree of the
                       #                     original tree)
     # We compute two values for T:
     #         LP : the length of the longest path in T
     #         BT : the longest path in T with one end at the bottom of T
     #              and the other end at x (BT stands for "bottom to top")
     # This algorithm will return both of these values - if implemented in
     # a language that permits only single value returns, an object
     # containing both of these paths must be constructed and returned.

     # base case
     if T consists of just the vertex x:
          LP = 0     # the longest path in T contains 0 edges
          BT = 0     # the longest path from the bottom to the top of T
                     #  contains 0 edges
     else:
          Let T_1,T_2,...T_k be the subtrees that are attached to x, each
               rooted at the vertex x_i that connects directly to x
          max_subtree_LP = 0
          max_subtree_BT = -1        # Exercise: why do these start at -1
          second_max_subtree_BT = -1
          for i = 1, 2, ... k
               L, B = Max_Tree_Path(T_i, x_i)
                    # this recursive call returns the LP and BT values for
                    # subtree T_i
               if L > max_subtree_LP:
                    max_subtree_LP = L
               if B > max_subtree_BT:
                    second_max_subtree_BT = max_subtree_BT
                    max_subtree_BT = B
               elsif B > second_max_subtree_BT:
                    second_max_subtree_BT = B

          # now we have all the information we need to compute LP and BT
          BT = max_subtree_BT + 1
          LP = max( BT, max_subtree_LP, max_subtree_BT +
                                        second_max_subtree_BT + 2 )

          return LP, BT
```

This algorithm has some interesting points. In most D&C algorithms we know exactly what information we need from each subproblem. In this algorithm we don't know whether the LP or the BT values from the subtrees will be most useful to find the overall solution, so we compute both of them for each subtree.

The complexity of the algorithm may look difficult to compute. We can define MTP(T) to be the time required for Max_Tree_Path to run on a tree T, and the recurrence relation would look something like this:

MTP(T) = c + MTP(T_1) + MTP(T_2) + ... MTP(T_k) where $T_1 \ldots T_k$ are the subtrees. This is hard to analyze because we can't predict the sizes of the subtrees.

However, if we think of the "execution tree" of the recursive calls made by the algorithm, we see that every single vertex of the tree gets to be the root of a subtree exactly once during the execution of the algorithm.

So the number of calls to the function is exactly equal to the number of vertices, and the work related to each recursive call is constant (just a few comparisons and additions). This shows that the complexity is O(n).

We can also think of this as a process of passing information along the edges of the graph. Each time we finish a recursive call, two values get passed up the edge from the root of the subtree to its parent. So each edge contributes a constant amount to the total work that is done, and since there are n-1 edges, the total amount of work is O(n).

Thus the algorithm runs in O(n) time - a full order of magnitude faster than the O(n^2) algorithm that we started with. If we are dealing with a very large tree, that's a big advantage.

This style of analysis: stepping away from the line-by-line analysis of the algorithm and focusing on the the total amount of work associated with each part of the structure being processed (in this case, a tree) is a very powerful tool.

The existence of such a fast algorithm for solving on trees a problem which is known to be NP-Complete on general graphs leads to an obvious but profound question: **does it only work on trees, or can we use the same technique on other types of graphs?** For example, what if we add one edge to a tree - can we still solve the longest-path problem in polynomial time? (The answer is Yes - **try to figure out how to do it**.) It turns out that there are many families of graphs for which this and other NP-Complete problems can be solved in polynomial time. These results are beyond the scope of this course but they are not difficult to understand. For more information search for "k-terminal recursive graphs" and in particular, papers by Steve Hedetniemi and Tom Wimer. (For example, here is the link to Tom Wimer's PhD thesis: https://dl.acm.org/citation.cfm?id=913798)