CS/ECE 374 FALL 2018
Homework 6 Problem 3

Zhe Zhang (zzhan157@illinois.edu)
Ray Ying (xinruiy2@illinois.edu)
Anqi Yao (anqiyao2@illinois.edu)

3. Given a graph $G = (V, E)$ a matching is a subset of edges in G that do not *intersect*. More formally $M \subseteq E$ is a matching if every vertex $v \in V$ is incident to at most one edge in M. Matchings are of fundamental importance in combinatorial optimization and have many applications. Given G and non-negative weights $w(e), e \in E$ on the edges one can find the maximum weight matching in a graph in polynomial time but the algorithm requires advanced machinery and is beyond the scope of this course. However, finding the maximum weight matching in a tree is easier via dynamic programming.

   - **Not to submit:** Given a tree $T = (V, E)$ and non-negative weights $w(e), e \in E$, describe an efficient algorithm to find the weight of a maximum weight matching in T.
   - Solve the previous part even though it is not required to be submitted for grading. It will help you think about this part.
     (a) Given a tree $T = (V, E)$ describe an efficient algorithm to *count* the number of distinct matchings in T. Two matchings M_1 and M_2 are distinct if they are not identical as sets of edges. Unlike the maximum weight matching problem, the problem of couting matchings is known to be hard in general graphs, but trees are easier.
     (b) Write a recurrence for the exact number of matchings in a path on n nodes. For the base case of a single node tree assume that the answer is 1 since an empty set is also a valid matching. Would the answer for a path with $n = 500$ fit in a 64-bit integer word? Briefly justify your answer.
     (c) How would you implement your counting algorithm from part (a), more carefully, to run on a 64 bit machine? Accounting for this more careful implementation, what is the running time of your algorithm?

Solution:

1. Since there is only one path between any two vertices in a tree, we can randomly assign a vertex $v_i \in V(G)$ to be the root of the tree. The vertices a node is connected to except his parent, be the children of that node. Applying this, we can simply build a tree in $O(n)$ (n being the number of vertices in G).
   After we build the tree, we count the number of matching in a tree starting from the root. We first make a global variable called "count" that keep the count of the number of matching. We will set count = 1 initially since according to question b below, empty set is also a valid matching. We define a function:

   $$countN(node, cond)$$

   The parameter "cond" specifying if the node is already in the matching or not. We assume that the node is not in matching if "cond" is 0, and node is in the matching then "cond" is 1. When the "cond" is 0, we can either choose one of the edges the node connects to the child and perform "countN" on all other child node with "cond" = 0. or we can choose no edge,

then we have to perform function "countN" on all its child nodes with "cond" = 0. If the "cond" is 1, we need to preform this function "countN" to the all the child node. At anystep, if we choose an edge we have to add 1 to the global variable "count"(one edge is also able to called a matching). We also define a function getchildren(node) that returns the child of the node.
The function will described below:

```
count = 1
cond = 0
countN(node, cond):
  if (node == NULL) :
      return
  childrenList = getchildren(node)
  if (childrenList == NULL) :
      return
  if (cond == 0) :
      count+ = sum_{i=0}^{len(childrenList)} countN(childrenList[i], 0)
      for matchNode in childrenList:
          for childnode in childrenList:
              if (childnode! = matchNode):
                  count+ = countN(childnode, 0)
              if (childnode == matchNode):
                  count+ = 1 + countN(matchNode, 1)
  if (cond == 1) :
      for childnode in childrenList:
          count+ = countN(childnode, 0)
```

We simply apply the function "$countN(root, 0)$" where root is the root of the tree, global variable "count" will return the result of number of matching in the tree. The big-O complexity of the function is $O(n^3)$.

2. When we are counting the number of matching in a path, at each edge, we are making a choice whether we want the edge to be in the matching or not. Starting from one endpoint of the path, if the edge is in the matching, the next edge it adjacent to cannot be in the matching since matching does not share vertices. If the edge is not in the matching, the next edge can be in the matching. In the recursion, when there are only 2 edges, 3 nodes, base case for the number of matching is 3 since there are 3 combination in total. When there is only 1 edge, 2 nodes, the base case is 2 since for 1 edge, there are only 2 choice, in the matching or not in the matching. When there is only 1 node left, we want to set it to 1 since the case that the set is empty. The recursive function will be as below:

```
COUNTP(num):
  if (num == 0||num == 1) :
      return 0
  if (num == 2) :
      return 2
  if (num == 3) :
      return 3
  return countP(n−1)+countP(n−2)
```

When n is 500, the result won't fit in a 64-bit integer word. We are breaking $countP(n)$ into two sub-function $countP(n-1)$ and $countP(n-2)$ when the base case isn't reached. At the very start, we will have $countP(500)$. In the first split, it becomes $countP(499)$ and $countP(498)$. We want to find out the number of times that one can split. We can take the smaller one since it's always going to be the first one to reach the base case. The number is decreasing at 2 per split. So the least split time would be $(500-3)/2 > 248$. If every split in the recursion reach the base case at 248 times, we can have at least $2^2 48$ number of base cases(base case of 2 or 3, not 1 or 0 in this case). The number of base case is greater than 1, the total size would be definitely greater than a 64—bit integer word.

3. Just like what we have in part b, if there are too many nodes in a path, we cannot fit it into 64—bit machine. In a tree, any two vertices in the tree have exactly one path between between them. The best implementation for part a to fit is just finding the root so that maximum root-leaf distance will be fixed within a 64-bit length. That's, not changing what we are doing, but finding the center of a tree where the eccentricity is smallest. Find the center of a tree will take $O(kn)$ time where k is the degree of the vertex and n is the number of vertex because we have to go through every vertex to find the center. Once we find the center, we build the tree the same way as in part a. Since in the part a, the base case will always be the leaf nodes. We can actually stores the value of each children since we are using that value a lot. The time complexity won't change since the choosing root is $big-O(n)$ where doing the whole recursion will take $O(n^3)$. So the whole time complexity will be $O(n^3)$

■