```python
#!/usr/bin/env python

import os
import sys

from graph_pb2 import Graph
from graph_pb2 import FeatureNode
from method_tokens import tokenize_methods_for_file, get_source_dict_graph, get_id_to_node_graph,
tokenize_methods_for_graph

import numpy as np
import string
import re

def filter_tokens(arr, lambda_function):
    return [n for n in filter(lambda_function, arr)]

def condition(node, id_mapping, source_mapping, path):
    if node == None or len(path) < 2:
        return False
    first = path[-2]
    second = path[-1]
    if ((first.contents == "VARIABLE" or first.contents == "METHOD") and second.contents == "NAME"):
        return True
    if (first.contents == "CLASS" and second.contents == "SIMPLE_NAME"):
        return True
    return False

def generate_new_name(length,b=string.ascii_uppercase):
   d, m = divmod(length,len(b))
   return generate_new_name(d-1,b)+b[m] if d else b[m]

def precompute_new_names(length, path):
    with open(path, 'w') as f:
        for i in range(length):
            f.write(generate_new_name(i) + '\n')

def get_new_names(length, path):
    if not os.path.isfile(path):
        precompute_new_names(max(length, 100000), path)
    with open(path) as myfile:
        head = [next(myfile) for x in range(length)]
        if len(head) < length:
            with open(path, 'a') as f:
                for i in range(length - len(head) + 1):
                    f.write(generate_new_name(len(head) + i) + '\n')
        return [x.rstrip() for x in head]

def combine(firstId, secondId):
    return (firstId << 16) | secondId

def get_obfuscation_names(nodeId, id_mapping, source_mapping, visited, path):
    needs_obfuscation = set()
    if combine(nodeId, path[-1].id) in visited:
        return needs_obfuscation
    node = id_mapping[nodeId]
    if condition(node, id_mapping, source_mapping, path):
        needs_obfuscation.add(node.contents)
    visited.add(combine(nodeId, path[-1].id))
    path.append(node)
    edgeTo = source_mapping.get(nodeId)
    if (edgeTo != None and len(edgeTo) > 0):
        for edge in edgeTo:
            needs_obfuscation |= get_obfuscation_names(edge.destinationId, id_mapping, source_mapping, visited, path)
    path.pop()
    return needs_obfuscation

def create_names_mapping(old_names_set, new_names_arr):
    new_dict = dict()
    index = 0
    for old_name in old_names_set:
        new_dict[old_name] = new_names_arr[index]
        index += 1
    return new_dict

def isValidSymbolMth(contents, new_names_mapping):
    splitUp = contents.split(".")
    length = len(splitUp)
    last = splitUp[length - 1]
    if (last[-2] == '(' and last[-1] == ')' and last[:-2] in new_names_mapping):
        return True
    for sub_path in splitUp:
        for splt in re.split(r'\$1*', sub_path):
            if splt in new_names_mapping:
                return True
    return False
```

```python
def middle_substitute(splitUp, new_names_mapping):
    to_return = []
    for sub_path in splitUp:
        new_subpath = ''
        index = 0
        newSplit = re.split(r'\$1*', sub_path)
        for splt in newSplit:
            if splt in new_names_mapping:
                splt = new_names_mapping[splt]
            new_subpath += splt + ('$' if index != 0 else '$1')
            index += 1
        to_return.append(new_subpath[:-1] if len(newSplit) > 1 else new_subpath[:-2])
    return to_return

def substituteSymbolMth(contents, new_names_mapping):
    splitUp = contents.split(".")
    length = len(splitUp)
    to_return = middle_substitute(splitUp, new_names_mapping)
    last = to_return[length - 1]
    if (last[-2] == '(' and last[-1] == ')' and last[:-2] in new_names_mapping):
        to_return[length - 1] = new_names_mapping[last[:-2]] + '()'

    return '.'.join(to_return)

def isValidSymbolVar(contents, new_names_mapping):
    splitUp = contents.split(".")
    for sub_path in splitUp:
        for splt in re.split(r'\$1*', sub_path):
            if splt in new_names_mapping:
                return True
    return False

def substituteSymbolVar(contents, new_names_mapping):
    splitUp = contents.split(".")
    length = len(splitUp)
    to_return = middle_substitute(splitUp, new_names_mapping)
    return '.'.join(to_return)

def substitute_all(nodes, new_names_mapping):
    for node in nodes:
        if (node.type == FeatureNode.IDENTIFIER_TOKEN and node.contents in new_names_mapping):
            node.contents = new_names_mapping[node.contents]
        elif (node.type == FeatureNode.METHOD_SIGNATURE and node.contents[:-2] in new_names_mapping):
            node.contents = new_names_mapping[node.contents[:-2]] + '()'
        elif (node.type == FeatureNode.SYMBOL_MTH and isValidSymbolMth(node.contents, new_names_mapping)):
            node.contents = substituteSymbolMth(node.contents, new_names_mapping)
        elif (node.type == FeatureNode.SYMBOL_VAR and isValidSymbolVar(node.contents, new_names_mapping)):
            node.contents = substituteSymbolVar(node.contents, new_names_mapping)


def obfuscate_path(path, precomputed_name_files):
    with open(path, "rb") as f:
        g = Graph()
        g.ParseFromString(f.read())
        return obfuscate_graph(g, precomputed_name_files)


def obfuscate_graph(g, precomputed_name_files):
    id_mapping = get_id_to_node_graph(g)
    source_mapping = get_source_dict_graph(g)
    start_node = g.ast_root
    initialPath = []
    initialPath.append(start_node)
    to_obfuscate = get_obfuscation_names(start_node.id, id_mapping, source_mapping, set(), initialPath)

    new_names = get_new_names(len(to_obfuscate), precomputed_name_files)
    new_names_mapping = create_names_mapping(to_obfuscate, new_names)
    substitute_all(g.node, new_names_mapping)
    return g

if __name__ == "__main__":
    filePath = sys.argv[1]
    precomputed_name_files = "precomputed_names.txt"
    with open(filePath, "rb") as f:
        untouched = Graph()
        untouched.ParseFromString(f.read())
        obfuscated_graph = obfuscate_path(filePath, precomputed_name_files)
        before = tokenize_methods_for_graph(untouched)
        after = tokenize_methods_for_graph(obfuscated_graph)

        print("BEFORE:")
        print(before)

        print("AFTER:")
        print(after)
```