Algorithm Design Assignment 1

Jianan Lin, 662024667, linj21@rpi.edu

Problem 3.9. Suppose an n-node undirected graph $G = (V, E)$ contains two nodes s and t so that the distance between them is strictly greater than $n/2$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all s-t paths. In other words, the graph obtained from G by deleting v contains no path from s to t. Give an algorithm with running time $O(m + n)$ to find such a node v.

Solution.

Part 1. We first prove the existence of such a node v. We start from s and go to t. Obviously, if we use d to denote the distance between the two nodes, then we must go through at least $d - 1$ nodes. We call a node *level* k if the distance between this node and s is k.

Using contradiction proof, we suppose there is no such node. Therefore we have for each level $k \in [1, d - 1]$, there must be at least two nodes. Otherwise if there is only one node for some level k, then we only need to cut this, then there is no path from s to t, because there is no node with distance k to s. Notice that, if one node is at level k, then it cannot be at other level k' since the distance is fixed.

In this way, we know there are at least $2(d - 1) + 2$ nodes in this graph. According to the problem, we have

$$n \geq 2(d - 1) + 2 = 2d > n,$$

where we get into a contradiction. Therefore the original proposition holds. There must exist such a node v.

Part 2. We use BFS method to find this node. The algorithm is as follows. Here we assume each node has an index from 0 to $n - 1$.

Algorithm 1 BFS

```
layer = [ ];    reached = [false × $n$];    reached[$s$] = true;
for all $v \in s.neighbors$ do
    layer.append($v$);
    reached[$v$] = true;
end for
while layer.size > 1 do
    temp = [ ];
    for all $v \in$ layer do
        for all $x \in v.neighbors$ do
            if reached[$x$] == 0 then
                temp.append($x$);
                reached[$x$] = 1;
            end if
        end for
    end for
    layer = temp;
end while
return layer[0];
```

First we prove that this algorithm is effective. In fact, this algorithm is the implementation of the previous proof. Each layer k is the set of nodes with distance k to s. Since there must be a layer with only one node and this is the cut node we need, It is easy to achieve this. When we find a layer with only one node, then we stop this algorithm and return the only node in the layer.

Next we prove the time complexity of this algorithm is $O(m + n)$, where m and n are the number of edges and nodes respectively.

Pseudocode in line 1 costs $O(n)$, which is obvious.

Pseudocode from line 2 to line 5 costs $O(\min(m, n))$ becase there are at most $\min(m, n - 1)$ neighbors for node s.

Pseudocode from line 6 to line 17 costs $O(m)$. When a node is reached, its value in the boolean array (called reached) will be changed into 1 (and never be operated again). In this way, any edge will be reached at most 2 times because each neighbor of each node is reached at most once.

Therefore we prove this algorithm is effective and $O(m + n)$.

Problem 4.6. There are some contestants. Each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. There must be one contestant in the pool at one time. We know the time each contestant finished each stage. Give an efficient algorithm that produces a schedule whose completion time is as small as possible. Here, when all the contestants finish their tasks, we say it is completed.

Solution.

Use a_i to denote the time contestant i takes to swim and b_i to denote the time contestant i takes to bike and run. Use t_i to denote the time when contestant i finishes all the parts. Then we write the greedy algorithm as follows.

Algorithm 2 Greedy

```
1: Sort the contestants by $b_i$ in descending order.
2: Send contestants into the pool in this order.
```

Then we prove this algorithm is efficient (in fact, optimal) in completion time. We know the completion time is

$$T_{ALG} = \max_{i\in[n]} t_i = \max_{i\in[n]} \left(\left(\sum_{j=1}^{i} a_j \right) + b_i \right).$$

Suppose k is this i when it reaches maximum, i.e. the time when k finishes is the completion time using the greedy algorithm. For any other case, we can transfer that case into the greedy case by repeating the swapping of two *consecutive* contestants, where the contestant in front of the other has less b (i.e. bubble sort).

Here we only need to prove: if we change the positions of two contestants $i, i + 1$ with $b_i < b_{i+1}$, then the completion time can never be worse. We consider different contestants and use i before swapping and i' after swapping.

For contestants $x > i + 1$ and $x < i$, its ending time does not change according to the definition of t_i because the formula maintains the same.

For contestants $i, i+1$, we know $t_{i+1} > t_i$ because $\sum_{j=1}^{i+1} a_j > \sum_{j=1}^{i} a_j$ and $b_{i+1} > b_i$. In this way, we have

$$\left(\sum_{j=1}^{i} a_j\right)+b_i < \min\left(\left(\sum_{j=1}^{i} a_j\right) + b_{i+1}, \left(\sum_{j=1}^{i+1} a_j\right) + b_i\right) \leq \max\left(\left(\sum_{j=1}^{i} a_j\right) + b_{i+1}, \left(\sum_{j=1}^{i+1} a_j\right) + b_i\right) < \left(\sum_{j=1}^{i+1} a_j\right)+b_{i+1}$$

Using t'_i to denote i after changing and t'_{i+1} to denote $i+1$ after changing, we have

$$t_i < \min(t'_i, t'_{i+1}) < \max(t'_i, t'_{i+1}) < t_{i+1},$$

i.e.

$$\max(t'_i, t'_{i+1}) < \max(t_i, t_{i+1}),$$

which means that after the changing (swapping), the completion time will never be longer. Therefore if we repeat this operation and finally get the greedy algorithm case, then the completion time will either be shorter or maintain. So this algorithm is efficient (optimal).