Homework 2

Jiahao Liu   January 24, 2024

1. What is the tight bound on worst-case runtime performance of the procedure below? Give an explanation for your answer.

```
int c = 0;
for(int k = 0; k <= log(n); k++)
    for(int j = 1; j <= 2^k; j++)
            c++;
return c;
```

For each outer loop iteration, the inner loop will run c++ for 2^k times. So the total running time is:

$$\sum_{k=0}^{\lfloor log_2 n \rfloor} 2^k = \frac{1 * (1 - 2^{\lfloor log_2 n \rfloor + 1})}{1 - 2} = 2 * 2^{\lfloor log_2 n \rfloor} - 1$$

For worst-case, $\lfloor log_2 n \rfloor$ is just $log_2 n$, so the running time $2n - 1$, the tight bound is $\Theta(n)$.

2. Given an undirected graph G with n nodes and m edges, design an O(m+n) algorithm to detect whether G contains a cycle. Your algorithm should output a cycle if G contains one.

We can use DFS with state label and parent array to detect whether there is a cycle in a graph. The algorithm with DFS is as follows:

```
DFS with state label and parent array:

Run DFS start from any random node V:
    label V as visited and visit all the neighbors of V.
        If the neighbor is not visited, then set the parent of this neighbor as V and run DFS on this neighbor node;
        If the neighbor is visited and it's not the parent node of V, we find a cycle and can print the cycle by traceback the parent array start from V. Then return true, because we find a cycle.
If all nodes are labeled, return false.
Run this DFS start from all the unvisited nodes in graph G: If any of them returns true, then there is a cycle in the graph.
Else, all the nodes and edges are visited and there is no cycle in the graph.
Because each node and edge is labeled at most once, the time complexity is $O(m + n)$.
```

Starting from an arbitrary vertex s, run BFS to obtain a BFS tree T, which takes O(m+n) time.
If G = T, then G is a tree and has no cycles.
Otherwise, G has a cycle and there exists an edge $e = (u, v) \in G - T$, Let w be the least common ancestor of u and v. There exist a unique path P_1 in T from u to w and a unique path P_2 in T from w to v. Both P_1 and P_2 can be found in O(m) time. Finally, output the cycle $P_1 \cup P_2 \cup \{e\}$.

```
int n, m; // n is the number of nodes, m is the number of edges
int h[N], e[M], ne[M], idx; // Adjacency List for the graph
bool st[N]; // if the node is visited
int pre[N]; // the previous node on the path before the current node
bool dfs(int u) {
    st[u] = true; // label the node as visited
    for (int i = h[u]; i != -1; i = ne[i]) { // visit all the neighbors of the current node
        int j = e[i];
        if (!st[j]) { // if the node is not visited, then dfs the node
            pre[j] = u;
            if(dfs(j)) return true;
        } else if (j != pre[u]) { // the node is visited and is not the previous node
            for(int k = u; k != j; k = pre[k]) cout << k << " "; // print the cycle
            return true; // then there is a cycle
        }
    }
    return false;
}
bool hasCycle() {
    for(int i = 1; i <= n; i++) { // visit all the nodes from 1 to n
        if(!st[i]) { // only dfs the node that is not visited
            if(dfs(i)) return true;
        }
    }
    return false;
}
void add(int a, int b){ // add an edge from a to b
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}
int main(){
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b; // input edge a->b with nodes in the range [1, n]
        add(a, b), add(b, a); // add the undirected edge
    }
    if(hasCycle()) puts("Yes");
    else puts("No");
    return 0;
}
```

3. For each of the following indicate if $f = O(g)$ or $f = \Omega(g)$ or $f = \Theta(g)$ or none of these.

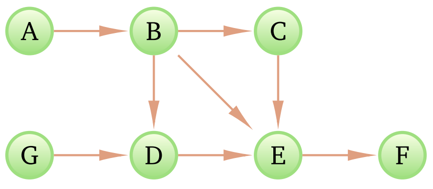
f(n)	g(n)	Answer
$nlog(n)$	$n^2log^2(n)$	$f(n) = O(g(n))$
$log(n)$	$log(log(5^n))$	$f(n) = \Theta(g(n))$
$n^{1/3}$	$log^3(n)$	$f(n) = \Omega(g(n))$
2^n	2^{3n}	$f(n) = O(g(n))$
$n^4/log(n)$	$nlog^4(n)$	$f(n) = \Omega(g(n))$

Since n grows exponentially faster than $logn$, meanwhile $(logn)^9$ grows polynomially faster than $logn$, n is therefore expected to grow faster than $(logn)^9$.

4. Indicate for each pair of expressions (A,B) in the table below, whether $A = O(B), A = \Omega(B), A = \Theta(B)$. Assume that k and C are positive constants.

A	B	O	Ω	Θ
$n^3 + log(n) + n^2$	Cn^3	Yes	Yes	Yes
n^2	$Cn * 2^{log(n)}$	Yes	Yes	Yes
$2^n * 2^k$	n^{2k}	No	Yes	No

5. Find the total number of possible topological orderings in the following graph and list all of them



There are 7 possible topological orderings:

A B G D C E F   A G B D C E F   G A B D C E F
A B C G D E F   A B G C D E F   A G B C D E F   G A B C D E F

6. Given a directed graph with m edges and n nodes where every edge has weight as either 1 or 2, find the shortest path from a given source vertex s to a given destination vertex t. Time complexity is O(m+n)

Solution 1: For every edge weight 2, we can split it into two edges with weight 1. Then we can use BFS to find the shortest path from s to t. In worst-case the time complexity is $O(2m + 2n)$, but still $O(m + n)$.

```
BFS for shortest path:

Build the graph into Adjacency List with adding an additional node into weight 2 edge to split
it into two edges with weight 1;
Create a distance array and initialize all the distance to infinity
Initialize the distance of source node to 0.
Create a queue and push the source node into the queue.
While the queue is not empty:
    Pop the first t node from the queue and go through all its neighbors:
        If the neighbor is not visited:
            update the distance of the neighbor by adding the distance of t and 1
            push this neighbor into the queue.
After the queue is empty, the distance array is the shortest path from source node to all the
nodes.
```

```
int h[N], e[M], ne[M], idx; // directed graph without weight
int q[N], hh, tt=-1; // queue for BFS
int dist[N]; // dist[N] is the distance from source node to node N
int n, m, s, t;
void add(int a, int b){
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}
void dfs(int u){
    dist[u] = 0; // initialize the distance of source node to 0
    q[++tt] = u; // push the source node into the queue
    while(hh <= tt){
        int t = q[hh++]; // pop the first node from the queue
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if(dist[j] == -1){ // if the neighbor is not visited
                dist[j] = dist[t] + 1;
                q[++tt] = j; // push the neighbor into the queue
            }
        }
    }
}
int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    memset(dist, -1, sizeof dist);
    int addNodes = 0;
    while (m -- ){
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        if(c == 2){
            addNodes ++; // count the number of nodes to be added
            add(a, n+addNodes); // add an edge from a to n+addNodes
            add(n+addNodes, b); // add an edge from n+addNodes to b
        }else{
            add(a, b); // add an edge from a to b with weight 1
        }
    }
    scanf("%d%d", &s, &t);
    dfs(s);
    printf("%d\n", d[t]);
    return 0;
}
```

Solution 2: suppose it's a directed acyclic graph, then we can use topological sort to find the shortest path from s to t. The algorithm includes topological sort and BFS for shortest path. The total time complexity is still $O(m + n)$.

Topological sort for shortest path of DAG:

```
Add all the nodes with indegree 0 into the queue;
While the queue is not empty:
    Pop the first node from the queue and go through all its neighbors:
        Decrease the indegree of the neighbor by 1;
        If the indegree of the neighbor is 0, then add it into the queue;
If there is still node with indegree not 0, then there is a cycle in the graph, this algorithm fails;
Else, the queue is the topological order of the graph.
Create a distance array and initialize all the distance to infinity
Initialize the distance of source node to 0.
While the queue is not empty:
    Pop the first node from the queue and go through all its neighbors:
        If the distance of the neighbor is larger than the distance of the current node plus the
weight of the edge, then update the distance of the neighbor.
```

```cpp
int h[N], e[M], w[M], ne[M], idx; // directed graph
int q[N], d[N], hh, tt=-1; // d[N] is the indegree of node, q[N] is for topological sort
int dist[N]; // dist[N] is the distance from source node to node N
int n, m;
void add(int a, int b, int c){
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}
bool toposort(){
    for (int i = 1; i <= n; i ++ ){ // indegree of the node form 1 to n
        if(!d[i]) q[++tt] = i;
    }
    while(hh <= tt){
        int t = q[hh++];
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            d[j] --; // delete the edge t->j
            if(!d[j]) q[++tt] = j;
        }
    }
    if(tt == n-1) return true;
    else return false;
}
int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    memset(dist, 0x3f, sizeof dist);
    while (m -- ){
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
        d[b] ++; // remember to add the indegree of node b
    }
    int s, t;
    scanf("%d%d", &s, &t);
    if(toposort()){
        dist[s] = 0; // initialize the distance of source node to 0
        hh = 0, tt = n;
        while(hh <= tt){
            int t = q[hh++];  // pop the node in topological order
            for(int i = h[t]; i != -1; i = ne[i]){
                int j = e[i];
                dist[j] = min(dist[j], dist[t] + w[i]); // update the distance of neighbor
            }
        }
    }
    if(dist[t] == 0x3f3f3f3f) puts("-1");
    else printf("%d\n", dist[t]);
    return 0;
}
```

7. Given functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Check the following statements, give a proof or counterexample.

a. $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$ True

Proof. Since $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, there exist n_1, n_2, C_1, C_2 such that $0 \le f_1(n) \le C_1 g_1(n)$ for all $n \ge n_1$ and $0 \le f_2(n) \le C_2 g_2(n)$ for all $n \ge n_2$. Then for all $n \ge max(n_1, n_2)$, we have $0 \le f_1(n) \cdot f_2(n) \le C_1 g_1(n) \cdot C_2 g_2(n) = C_1 C_2 g_1(n) g_2(n)$, which means $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$. □

b. $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ True

Proof. Since $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, there exist n_1, n_2, C_1, C_2 such that $0 \le f_1(n) \le C_1 g_1(n)$ for all $n \ge n_1$ and $0 \le f_2(n) \le C_2 g_2(n)$ for all $n \ge n_2$. Then for all $n \ge max(n_1, n_2)$, we have $f_1(n) + f_2(n) \le C_1 g_1(n) + C_2 g_2(n) \le Max(C_1, C_2)(g_1(n) + g_2(n))$, which means $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$. □

c. $f_1(n)^2 = O(g_1(n)^2)$ True

Proof. Since $f_1(n) = O(g_1(n))$, there exist n_1, C_1 such that $0 \le f_1(n) \le C_1 g_1(n)$ for all $n \ge n_1$. Then for all $n \ge n_1$, we have $f_1(n)^2 \le C_1^2 g_1(n)^2$, which means $f_1(n)^2 = O(g_1(n)^2)$. □

d. $lg(f_1(n)) = O(lg(g_1(n)))$ False

Proof. Consider the case that $f_1(n) = 2$ and $g_1(n) = 1$, then $lg(f_1(n)) = lg(2)$ and $lg(g_1(n)) = lg(1) = 0$, which means there is no such constant C that $lg(2) <= C * 0$. □

8. Design an algorithm which, given a directed graph $G = (V, E)$ and a particular edge $e \in E$, going from node u to node v determines whether G has a cycle containing e. The running time should be bounded by $O(|V| + |E|)$. Explain why your algorithm runs in $O(|V| + |E|)$ time

We can just run BFS/DFS start from node v, and check if it can reach node u. With label array and no traceback, all the nodes and edges are visited at most once, so the time complexity is $O(|V| + |E|)$.

```
DFS for connectivity from v to u:

Run DFS from node v:
    In DFS, label the node as visited and visit all the neighbors of the current node.
        If the neighbor is u, then return true.
        If the neighbor is not visited, then dfs the neighbor node.
If the function is not returned, then return false.
```

```cpp
int h[N], e[M], ne[M], idx; // Adjacency List for the graph
bool st[N]; // if the node is visited
int n, m, u, v; // n: nodes, m: edges, (u, v): the particular edge
bool dfs(int x) {
    st[x] = true; // label the node as visited
    for (int i = h[x]; i != -1; i = ne[i]) { // visit all the neighbors of the current node
        int j = e[i];
        if(j == u) return true;
        else if(!st[j]) { // if the node is not visited, then dfs the node
            if(dfs(j)) return true;
        }
    }
    return false;
}
void add(int a, int b){ // add an edge from a to b
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}
int main(){
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b; // input edge a->b with nodes in the range [1, n]
        add(a, b); // add the directed edge
    }
    scanf("%d%d", &u, &v);
    if(dfs(v)) puts("Yes, there is a cycle containing the edge (u, v)");
    else puts("No, there is no cycle containing the edge (u, v)");
    return 0;
}
```

9. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Proof. Assume there is an edge e = (x, y) in G that does not belong to T. Since T is a DFS tree, one of x or y is the ancestor of the other. On the other hand, since T is a BFS tree, x and y differ by at most 1 layer. Now since one of x and y is the ancestor of the other, x and y should differ by exactly 1 layer. Therefore, the edge e = (x, y) should be in the BFS tree T. This contradicts the assumption. Therefore, G cannot contain any edges that do not belong to T □

10. Suppose that an n-node undirected graph G=(V,E) contains two nodes s and t such that the distance between s and t 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.

Proof. Let's prove by contradiction. Suppose there is no such node v, then there must exist at least two paths from s to t, which are of length greater than n/2 and not connected except the two end nodes. In this case, there would be at least n+1 nodes in the graph, which is a contradiction. So there must exist some node v, not equal to either s or t, such that deleting v from G destroys all s-t paths. □

We can just run BFS/DFS from s and t seperately to find the first node that has more than one child nodes. The time complexity is $O(m + n)$.

```
BFS for path intersection:
Build the undirected graph into Adjacency List;
Run BFS from node s and node t seperately:
    push the node into the queue and label it as visited;
    While the queue is not empty:
        pop the first node from the queue and check the number of its child nodes.
        If the number of child nodes is larger than 1, then return the current node.
        If the number of child nodes is 1, label it as visited and push into the queue.
    After the while loop is finished, return -1.
If both BFS from s and t return -1, then there is no such node v. Else, return any one from BFS.
```

```cpp
int h[N], e[M], ne[M], idx; // Adjacency List for the graph
bool st[N]; // if the node is visited
int q[N], hh, tt = -1; // queue for BFS
int n, m, s, t; // n: nodes, m: edges, (s, t): the two nodes
int bfs(int x) {
    q[++tt] = x; // push the node into the queue
    st[x] = true; // label the node as visited
    while(hh <= tt) {
        int t = q[hh++]; // pop the first node from the queue
        int cnt = 0; // count the number of child nodes
        for(int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            cnt ++;
            if(!st[j]){
                st[j] = true; // label the node as visited
                q[++tt] = j; // push the neighbor into the queue
            }
        }
        if(cnt > 1) return t;
    }
    return -1;
}
void add(int a, int b){ // add an edge from a to b
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}
int main(){
    memset(h, -1, sizeof h);
    cin >> n >> m;
    while(m --) {
        int a, b;
        cin >> a >> b; // input edge a->b with nodes in the range [1, n]
```

```
            add(a, b); add(b, a); // add the undirected edge
        }
        scanf("%d%d", &s, &t);
        int v1 = bfs(s); // run BFS from s
        memset(st, false, sizeof st);
        int v2 = bfs(t); // run BFS from t
        if(v1 == -1 && v2 == -1) puts("No such node v");
        else if(v1 != -1) printf("The node v is %d\n", v1);
        else printf("The node v is %d\n", v2);
        return 0;
    }
```