Solutions

1. 10 pts)

Algorithm 1 Basic Euler Circuit Algorithm.

procedure Euler(G = (V,E) connected multigraph with all vertices of even degree).

P: = Do a depth-first search from an arbitrary chosen vertex until be back at that vertex.

H := H while removing the edges of this circuit (P).

while H has edges do

Pick a vertex v on P with an unused edge and use DFS from this vertex to find a circuit p, update P by inserting p at v.

H := H while removing the edges of the circuit p.

end while

return P

2. 20 pts

For the breadth-first search tree, when \( n = 0 \) the tree is a vertex. Given the tree \( T_n \) for \( Q_n \), the tree for \( Q_{n+1} \) consists of \( T_n \) with one extra child of the root, coming first in left-to-right order, and that child is the root of a copy of \( T_n \). And all vertices in \( T_n \) for \( Q_n \) will have incident edge with the child. For the depth-first search tree, the tree will depend on the order in which the vertices are picked. Because \( Q_n \) has a Hamilton path, it is possible that the tree will be a path. However, if bad choices are made, then the path
might run into a dead end before visiting all the vertices, in which case the tree will have to branch.

3. 20 pts

Proved by contradiction. Assume e is the edge with smallest weight and is not included in some minimum spanning tree; That is, suppose that the minimum spanning tree T contains only edges with weights larger than e’s weight. If we add e to T , then we will obtain a graph with exactly one simple circuit, which contains e. We can then delete some other edge in this circuit, resulting in a spanning tree with weight strictly less than that of T (since all the other edges have larger weight than e has). This is a contradiction to the fact that T is a minimum spanning tree. Thus, an edge with smallest weight must be included in T .

4. 20 pts

For a connected graph $G(V, E)$ has no vertex of degree 1, which means that all vertices have degree greater than 2 ($\geq 2$). Since we have $2|E| = \sum_{v \in V} \text{deg}(v)$, assume $|V| = n$, since $\text{deg}(v) \geq 2$, $2|E| = \sum_{v \in V} \text{deg}(v) \geq \sum_{v \in V} 2 \geq 2n$, we have $2|E| \geq 2n \rightarrow |E| \geq n$. If there is no cycle we have a path $v_1, v_2, \cdots , v_n$, whose number of edges is $n - 1$, one more edge must be within $v_1, v_2, \cdots , v_n$, which causes a cycle since $|E| \geq n$.

5. 10+20 pts

1) If there is an Euler circuit, P, in G. Every time a vertex is listed, that accounts for two edges adjacent to that vertex, the one before it in the list and the one after it in the list. This circuit uses every edge exactly once. So every degree must be even.

Suppose every degree is even. We will show that there is an Euler circuit by induction on the number of edges in the graph. The base case is for a graph G with two vertices with two edges between them. This graph is obviously Eulerian. Suppose we have a graph G on $k \geq 2$ edges. We start at an arbitrary vertex v and follow edges, arbitrarily selecting one after another until we return to v. Call this trail C. We know that we will return to v eventually because every time we encounter a vertex other than v we
are listing one edge adjacent to it. There are an even number of edges adjacent to every vertex, so there will always be a suitable unused edge to list next. So this process will always lead us back to v. Let E be the edges of C. The graph G^E has components G_1, G_2, \ldots, G_k. These each satisfy the induction hypothesis: connected, less than k edges, and every degree is even. We know that every degree is even in G^E, because when we removed C, we removed an even number of edges from those vertices listed in the circuit. By induction, each circuit has an Eulerian circuit, call them C_1, C_2, \ldots, C_k. Since G is connected, there is a vertex a_i in each component C_i on both C and C_i. Assume the vertices a_1, a_2, \ldots, a_k are visited in that order. We describe an Euler circuit in G by starting at v follow C until reaching a_1, follow the entire G_1 ending back at a_1, follow W until reaching a_2, follow the entire G_2, ending back at a_2 and so on. End by following W until reaching a_k, follow the entire G_k, ending back at a_k, then finish off C, ending at v

2) The idea of algorithm is same as Q1. It involves following steps:

1. Make sure all vertices in the graph have even degree, which takes \(O(|V|)\) time to figure it out.

2. If all the vertices have even degree (a) we first pick a vertex to the the start vertex. (b) Use DFS to find a cycle that begins and ends at the start vertex. This is now your current circuit. (c) If there is not a vertex on the current circuit that is incident to an unmarked edge, you are done. If there is such a vertex, use DFS to find a random cycle using unmarked edges that begin and ends at this vertex. Mark the edges in this cycle as you find it. Splice this cycle into the current circuit to make a new, larger current circuit that begins and ends at the start vertex. Repeat this step. Actually, we visit all edges in graph once, which should be \(O(|E|)\).

In the following, we use C++ to implement the algorithm, which takes \(O(|V| + |E|)\)

class Graph
{
    // # of vertices
    int V;
    // A dynamic array of adjacency lists
    list<int> *adj;
public:

3
Graph(int V) { this->V = V; adj = new list<int>[V]; }

~Graph() { delete [] adj; }

// functions to add and remove edge
void addEdge(int u, int v) {adj[u].push_back(v); adj[v].push_back(u);}
void rmvEdge(int u, int v);

// Methods to print Eulerian circuit
void EulerCir();
void Path(int s);
int DFS(int v, bool visited[]);
bool isValidNextEdge(int u, int v);
};
void Graph::EulerCir()
{
    // Find a vertex with odd degree
    int u = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
            u = i; break;
    Path(u);
    cout << endl;

    // Print Euler circuit starting from vertex u
    void Graph::Path(int u)
    {
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
            int v = *i;
            // If edge u-v is not removed and it's a a valid next edge
            if (v != -1 && isValidNextEdge(u, v))
            {
                cout << u << "-" << v << " ";
                rmvEdge(u, v);
                Path(v);
            }
        }
    }

    bool Graph::isValidNextEdge(int u, int v)
// The edge u–v is valid in one of the following two cases:
// 1) If v is the only adjacent vertex of u
int count = 0; // To store count of adjacent vertices
list<int>::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
    if (*i != -1)
        count++;
if (count == 1)
    return true;

bool visited[V];
int count1 = DFSCount(u, visited);
rmvEdge(u, v);
int count2 = DFSCount(u, visited);
addEdge(u, v);
return (count1 > count2)? false: true;

void Graph::rmvEdge(int u, int v)
{
    // Find v in adjacency list of u and replace it with −1
    list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);
    *iv = -1;
    // Find u in adjacency list of v and replace it with −1
    list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);
    *iu = -1;
}

// A DFS based function to count reachable vertices from v
int Graph::DFS(int v, bool visited[])
{
    // Mark the current node as visited
    visited[v] = true;
    int count = 1;
    // Recur for all vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (*i != -1 && !visited[*i])
            count += DFS(*i, visited);
    return count;
}

bool isValidNextEdge(int u, int v);

};