Finding the Eulerian path in $O(M)$

Table of Contents

A Eulerian path is a path in a graph that passes through all of its edges exactly once. A Eulerian cycle is a Eulerian path that is a cycle.

The problem is to find the Eulerian path in an **undirected multigraph with loops**.

Algorithm

First we can check if there is an Eulerian path. We can use the following theorem. An Eulerian cycle exists if and only if the degrees of all vertices are even. And an Eulerian path exists if and only if the number of vertices with odd degrees is two (or zero, in the case of the existence of a Eulerian cycle). In addition, of course, the graph must be sufficiently connected (i.e., if you remove all isolated vertices from it, you should get a connected graph).

The find the Eulerian path / Eulerian cycle we can use the following strategy: We find all simple cycles and combine them into one - this will be the Eulerian cycle. If the graph is such that the Eulerian path is not a cycle, then add the missing edge, find the Eulerian cycle, then remove the extra edge.

Looking for all cycles and combining them can be done with a simple recursive procedure:

```
procedure FindEulerPath(V)
  1. iterate through all the edges outgoing from vertex
       remove this edge from the graph,
       and call FindEulerPath from the second end of th
  2. add vertex V to the answer.
```

The complexity of this algorithm is obviously linear with respect to the number of edges.

But we can write the same algorithm in the non-recursive version:

```
stack St;
put start vertex in St;
until St is empty
  let V be the value at the top of St;
  if degree(V) = 0, then
    add V to the answer;
    remove V from the top of St;
  otherwise
    find any edge coming out of V;
    remove it from the graph;
    put the second end of this edge in St;
```

It is easy to check the equivalence of these two forms of the algorithm. However, the second form is obviously faster, and the code will be much more.

The Domino problem

We give here a classical Eulerian cycle problem - the Domino problem.

There are N dominoes, as it is known, on both ends of the Domino one number is written(usually from 1 to 6, but in our case it is not important). You want to put all the

dominoes in a row so that the numbers on any two adjacent dominoes, written on their common side, coincide. Dominoes are allowed to turn.

Reformulate the problem. Let the numbers written on the bottoms be the vertices of the graph, and the dominoes be the edges of this graph (each Domino with numbers (a, b) are the edges (a, b) and (b, a)). Then our problem is reduced to the problem of finding the Eulerian path in this graph.

Implementation

The program below searches for and outputs a Eulerian loop or path in a graph, or outputs -1 if it does not exist.

First, the program checks the degree of vertices: if there are no vertices with an odd degree, then the graph has an Euler cycle, if there are 2 vertices with an odd degree, then in the graph there is only an Euler path (but no Euler cycle), if there are more than 2 such vertices, then in the graph there is no Euler cycle or Euler path. To find the Euler path (not a cycle), let's do this: if $V1$ and $V2$ are two vertices of odd degree,then just add an edge $(V1, V2)$, in the resulting graph we find the Euler cycle (it will obviously exist), and then remove the "fictitious" edge $(V1, V2)$ from the answer. We will look for the Euler cycle exactly as described above (non-recursive version), and at the same time at the end of this algorithm we will check whether the graph was connected or not (if the graph was not connected, then at the end of the algorithm some edges will remain in the graph, and in this case we need to print -1). Finally, the program takes into account that there can be isolated vertices in the graph.

Notice that we use an adjacency matrix in this problem. Also this implementation handles finding the next with

brute-force, which requires to iterate over the complete row in the matrix over and over. A better way would be to store the graph as an adjacency list, and remove edges in $O(1)$ and mark the reversed edges in separate list. This way we can archive a $O(N)$ algorithm.

```
int main() {
    int n;
    vector<vector<int>> g(n, vector<int>(n));
    // reading the graph in the adjacency matr

    vector<int> deg(n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            deg[i] += g[i][j];
    }

    int first = 0;
    while (!deg[first])
        ++first;

    int v1 = -1, v2 = -1;
    bool bad = false;
    for (int i = 0; i < n; ++i) {
        if (deg[i] & 1) {
            if (v1 == -1)
                v1 = i;
            else if (v2 == -1)
                v2 = i;
            else
                bad = true;
        }
    }

    if (v1 != -1)
        ++g[v1][v2], ++g[v2][v1];

    stack<int> st;
    st.push(first);
```

```
vector<int> res;
while (!st.empty()) {
    int v = st.top();
    int i;
    for (i = 0; i < n; ++i)
        if (g[v][i])
            break;
    if (i == n) {
        res.push_back(v);
        st.pop();
    } else {
        --g[v][i];
        --g[i][v];
        st.push(i);
    }
}

if (v1 != -1) {
    for (size_t i = 0; i + 1 < res.size();
        if ((res[i] == v1 && res[i + 1] ==
            (res[i] == v2 && res[i + 1] ==
            vector<int> res2;
            for (size_t j = i + 1; j < res
                res2.push_back(res[j]);
            for (size_t j = 1; j <= i; ++j
                res2.push_back(res[j]);
            res = res2;
            break;
        }
    }
}

for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (g[i][j])
            bad = true;
    }
}

if (bad) {
```

```
        cout << -1;
    } else {
        for (int x : res)
            cout << x << " ";
    }
}
```