Solution — Algocoön Group

1 The problem in a nutshell

Given a weighted, directed graph, find a (nontrivial) partition of vertices $V = V_1 \cup V_2$ that minimizes sum of weights of edges going from V_1 to V_2.

2 Modeling

The problem of the story describes a problem of cutting a sculpture into pieces. However, the problem clearly is of a graph-theoretical nature, since no spatial information is provided:

> "Every sculpture consists of several figures, each of them equipped with (possibly large) number of limbs. Each limb reaches some other figure and has a cost that the cutter will charge for separating it. [...]"

Thus we deduce that figures correspond to vertices, and limbs to edges. Moreover, in the setting described, we can deduce that both (i) edges are *weighted* (each limb has cost assigned) and (ii) edges are *directed* (figures have limbs that reach to other figures). Thus we deduce that the underlying graph G is weighted and directed. No additional information on, i.e. connectivity is provided.

The answer we are asked to provide is as follow:

> "The deal is as follows: you will decide on how to cut the sculpture (i.e., which figures you take home). Both you and your partner need to get at least one figure. To share the cost, you pay for cutting the limbs of your figures and your friend for limbs of her figures.
>
> Your objective is to write a program that will go over all sculptures and for each of them find a cutting that minimizes *your* cost."

We see that we are asked to provide a way of cutting the graph into two parts (partition the set of vertices), so that we take one part and our partner takes another part (respectively V_1 and V_2). We want the partition to be no-trivial, that is $V_1 \neq \emptyset$ and $V_2 \neq \emptyset$, and for it to minimize the total cost of edges going from V_1 to V_2.

Next we look at the task limits: We have n vertices ($2 \leq n \leq 200$) and m edges ($0 \leq m \leq 5000$), indicating that inputs are not-that-large, so we can use algorithms of quadratic or cubic complexity.

At last, let us get the remaining information out of the *input* and *output specifications:* First of all, we need to watch out for border cases (i.e. G is not strongly connected, or has no edges). Secondly, the islands are 0-based (numbered from 0 to $n-1$), which will be convenient for implementation (i.e. we can use the island numbers directly as vector indices). Finally, costs are integer positive, that is $1 \leq c \leq 1000$, thus the answer always fits into an integer.

3 Algorithm Design

To design an efficient solution to this problem, we need to relate it to terms we are familiar with and algorithms useful in those situations. In our situation, it would be an *edge cut*. $F \subset E$ is an u-v edge cut, if any path connecting u to v in G has edges from F. Equivalently, we can say that removal of F from G separates u from v. This is exactly the term we are looking for, since if we fix u and v, then:

- any partition of the graph (that puts u into V_1 and v into V_2) induces an u-v edge-cut: (F will contain every edge that goes from V_1 to V_2),
- any u-v edge cut induces a partition of the graph, that puts u into V_1 and v into V_2, (i) remove F from G, (ii) vertices still reachable from u will be in V_1 (iii) vertices unreachable from u will be in V_2.

This is a very fancy way of saying, that to cut a graph so that u and v are in separate parts, we have to remove some edges along some edge cut.

Minimum edge cut So far, we have established that we are interested in a *minimum weight* u-v *edge cut* for some pair of vertices u,v. Graph theory teaches us that:

> "weight of minimum u-v edge cut is equal to size of maximum u-v flow"

We are approaching the proper solution, but there are a few obstacles:

1. maximum flow gives us weight of edge cut, not the edges of cut or the vertices of the partition
2. we don't know which u and v we should try.

To alleviate (1), we should find a way to extract the information on the cut from the max-flow routine. To help with (2), we can try multiple different pairs of vertices, hoping that among them there is a pair that minimizes the cut size. We can also observe, that (2) is nonexistent in the subtask 1, where we are explicitly told that $u = 0$ and $v = n - 1$ is enough.

Approaching the solution through subtasks A naive approach to (2) would be to try *all* of the possible pairs u-v. There are roughly n^2 of them. Quick calculation tells us, that a single maxflow takes roughly quadratic in n time[1], thus that might work only when n is very small, like in subtask 3. We should find a way to reduce the number of pairs of vertices to test from a quadratic to ideally a linear number of pairs.

A quick glance at subtask 2 tells us that for this particular subtask, we can simply test all pairs that have $u = 0$, that is 0-1, 0-2, ..., 0-($n-1$). There is a linear number of them ($n-1$ exactly), so that should fit into the time-limit of this problem. However, that also brings us closer to solving an unrestricted problem: by testing additional $n-1$ pairs, all pairs where $v = 0$, we can test all the solutions where 0 is *not taken* by us. However, this covers all possible cases, since either we do or we do not take vertex 0. To summarize, our solution right now consists of:

1. Build G as a weighted directed graph given on the input.

[1] this is not the guarantee, but it is useful to think of the maxflow in this form

2. For $u, v \in \{(0,1),(0,2),\ldots,(0,n-1),(1,0),(2,0),\ldots,(n-1,0)\}$ compute maxflow from i to j, and find the pair that minimizes this value.

3. Extract the vertex partition *somehow* induced by the maxflow from i to j.

However, there is a fancier way to implement step 2: look into pairs $\{(0,1),(1,2),(2,3),\ldots,(n-2,n-1),(n-1,0)\}$ instead. This reduces the workload by a factor of 2, and simplifies the main loop of the program. It works exactly for the same reason previous loop worked: there must be a pair $(i,(i+1)\%n)$ such that I take i and my partner takes $(i+1)\%n$.

4 Implementation

In the following, we will show how to come up with an implementation of the full solution.

Let us start by noting that, as in most graph algorithms, we will use an adjacency list graph representation. There is absolutely no reason to deviate from this approach in this particular problem.

First thing we need to do is to read the input into an appropriate representation. We use the templates that we have available and start with following:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <climits>
// Boost includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;
// BGL typedefs
typedef adjacency_list_traits<vecS, vecS, directedS>    Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, int,
        property<edge_residual_capacity_t, int,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
typedef graph_traits<Graph>::edge_descriptor                  Edge;
typedef graph_traits<Graph>::vertex_descriptor                Vertex;
typedef graph_traits<Graph>::out_edge_iterator                OutEdgeIt;
typedef property_map<Graph, edge_capacity_t>::type            EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type     ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type             ReverseEdgeMap;

// Custom Add Edge for flow problems
void addEdge(int from, int to, int capacity, EdgeCapacityMap &capacitymap,
                                             ReverseEdgeMap &revedgemap, Graph &G) {
        Edge e, reverseE;
        bool success;
        tie(e, success)         = add_edge(from, to, G);
        tie(reverseE, success)  = add_edge(to, from, G);
        capacitymap[e]          = capacity;
        capacitymap[reverseE]   = 0;
        revedgemap[e]           = reverseE;
        revedgemap[reverseE]    = e;
}
```

Note that even though G is directed, we add edges into both directions, one with original capacity, and other with 0 capacity, linking them with `rev_edge` map (of reverse edges). This is a standard thing to do when operating with a max-flow type of a problem. Since we are making the insertions in the code in a single place, we use a custom function (instead of an object), no need for encapsulation.

We continue with standard reading of input and defining residual map, capacity map and reverse edge map.

```
void testcases() {
        int n, m;
        cin >> n >> m;  // Number of figures and limbs
        // Build Graph
        Graph G(n);
        EdgeCapacityMap capacitymap = get(edge_capacity, G);
        ReverseEdgeMap revedgemap = get(edge_reverse, G);
        ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
        // Read edges
        for (int i = 0; i < m; ++i) {
                int from, to, c;
                cin >> from >> to >> c;
                addEdge(from, to, c, capacitymap, revedgemap, G);
        }
```

max_flow It is important to note that with any maximal flow function we have to define *residual capacity* property for our use.

Let's start by looking at `push_relabel_max_flow`' BGL documentation: First of all, we can see that the algorithm returns an integer (say `maxflow`), denoting the value of the maximum flow in G. This is already enough to locate the proper source and sink for the minimizing flow:

```
// Find a min cut via maxflow
int minmaxflow = INT_MAX;
pair<int, int>  sinksource = make_pair(0,1);
// Vertex i in my group of figures
for (int i = 0; i < n; ++i) {
        int maxflow = push_relabel_max_flow(G, i, (i+1)%n);
        if (maxflow < minmaxflow) {
                sinksource = make_pair(i,(i+1)%n);
                minmaxflow = maxflow;
        }
}
```

We should mention that it is ok to call `push_relabel_max_flow` several times on the same graph – the algorithm does not modify the edges of G and starts writing to `rescapacitymap` anew. After running this loop, we will have the source and sink in the variable `sinksource`. But how do we know the structure of the flow? There is a way to extract the information about the min-cut from the residual network (network of *unused flow*), namely:[2]

> From the source vertex, do a search along edges in the residual network (i.e., non-saturated edges and back edges of edges that have flow), and mark all vertices that can be reached this way. The cut consists of all edges that go from a marked to an unmarked vertex. Clearly, those edges are saturated and thus were not traversed.

[2]Residual BFS was also provided as a code template.

Thus we will do exactly this, using STL primitives like queue and the BGL interior property map rescapacitymap which we defined earlier to be the residual network table. Since we inserted both edges and backedges into the network, and they all have appropriate capacities in the rescapacitymap, our code is very simple and uses just non-zero residual capacity edges of G.

```
// Find all figures
minmaxflow = push_relabel_max_flow(G, sinksource.first, sinksource.second);
```

(Since we need to call push_relabel_max_flow once more to get proper rescapacitymap.)

```
vector<int> vis(n, false);
vis[sinksource.first] = true;
std::queue<int> Q;
Q.push(sinksource.first);
while (!Q.empty()) {
        const int u = Q.front();
        Q.pop();
        OutEdgeIt ebeg, eend;
        for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
                const int v = target(*ebeg, G);
                if (rescapacitymap[*ebeg] == 0 || vis[v]) continue;
                vis[v] = true;
                Q.push(v);
        }
}
```

Remainder As we have the partition of the vertices ready, all that remains is to output the size of the cut (i.e., minmaxflow), and all of the vertices on our side of the partition (i.e. all of the vertices that we have visited in the DFS).

```
// Output
cout << minmaxflow << endl << count(vis.begin(), vis.end(), true);
for (int i = 0; i < n; ++i) {
        if (vis[i]) cout << " " << i;
}
cout << endl;
```

Caveats There are a few caveats you need to consider:

- The input size is small. Any form of IO is fine here.
- Always compile BGL programs with the -O2 flag to get a runtime on your system which is (roughly) comparable to the runtime on the judge.

5 A Complete Solution

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <climits>
```

```cpp
// Boost includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
// Namespaces
using namespace std;
using namespace boost;
// BGL typedefs
typedef adjacency_list_traits<vecS, vecS, directedS>    Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, int,
        property<edge_residual_capacity_t, int,
            property<edge_reverse_t, Traits::edge_descriptor> > > >     Graph;
typedef graph_traits<Graph>::edge_descriptor                        Edge;
typedef graph_traits<Graph>::vertex_descriptor                      Vertex;
typedef graph_traits<Graph>::out_edge_iterator                      OutEdgeIt;
typedef property_map<Graph, edge_capacity_t>::type                  EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type         ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type                   ReverseEdgeMap;

// Custom Add Edge for flow problems
void addEdge(int from, int to, int capacity, EdgeCapacityMap &capacitymap,
                                             ReverseEdgeMap &revedgemap, Graph &G) {
        Edge e, reverseE;
        bool success;
        tie(e, success)         = add_edge(from, to, G);
        tie(reverseE, success)  = add_edge(to, from, G);
        capacitymap[e]          = capacity;
        capacitymap[reverseE]   = 0;
        revedgemap[e]           = reverseE;
        revedgemap[reverseE]    = e;
}

// Function solving a single testcase
void testcases() {
        int n, m;
        cin >> n >> m;  // Number of figures and limbs
        // Build Graph
        Graph G(n);
        EdgeCapacityMap capacitymap = get(edge_capacity, G);
        ReverseEdgeMap revedgemap = get(edge_reverse, G);
        ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
        // Read edges
        for (int i = 0; i < m; ++i) {
                int from, to, c;
                cin >> from >> to >> c;
                addEdge(from, to, c, capacitymap, revedgemap, G);
        }

        // Find a min cut via maxflow
        int minmaxflow = INT_MAX;
        pair<int, int>  sinksource = make_pair(0,1);
        // Vertex i in my group of figures
        for (int i = 0; i < n; ++i) {
                int maxflow = push_relabel_max_flow(G, i, (i+1)%n);
                if (maxflow < minmaxflow) {
                        sinksource = make_pair(i,(i+1)%n);
                        minmaxflow = maxflow;
                }
        }
```

```
        // Find all figures
        minmaxflow = push_relabel_max_flow(G, sinksource.first, sinksource.second);
        vector<int> vis(n, false);
        vis[sinksource.first] = true;
        std::queue<int> Q;
        Q.push(sinksource.first);
        while (!Q.empty()) {
                const int u = Q.front();
                Q.pop();
                OutEdgeIt ebeg, eend;
                for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
                        const int v = target(*ebeg, G);
                        if (rescapacitymap[*ebeg] == 0 || vis[v]) continue;
                        vis[v] = true;
                        Q.push(v);
                }
        }
        // Output
        cout << minmaxflow << endl << count(vis.begin(), vis.end(), true);
        for (int i = 0; i < n; ++i) {
                if (vis[i]) cout << " " << i;
        }
        cout << endl;
}

// Function looping over the testcases
int main() {
        ios_base::sync_with_stdio(false);
        int T;  cin >> T;
        for ( ; T > 0; --T)     testcases();
        return 0;
}
```