Solution — Kingdom Defense

1 The problem in a nutshell

Circulation problem on a directed graph with edges having maximum *and minimum* capacities.

2 Modeling

We are given a directed graph G, with vertices modeling the cities of the kingdom and edges modeling the (directed) swamp paths. Every vertex has a given *supply* of soldiers (the soldiers originally stationed at the city) and a given *demand* of soldiers (the number of soldiers needed to defend the city). Each edge has a minimum capacity (or requirement) c and a maximum capacity C, denoting limits on the number of times a soldier should move along this edge.

The right toolbox Which tools that we know might be applicable to the problem setting?

STL The description seems to leave a lot of room for corner cases as well as a large variety of potential input graphs. In other words, there is not much structure given that we might be able to exploit with an ad-hoc (STL-only) solution.

CGAL The given lower and upper bounds (demand/supply, min/max capacities) would make linear programming an obvious choice to try. However, since we can not send "fractional soldiers" along a swamp path (e.g. half of a soldier along edge $e_1 = (u, v)$, his other half along an edge $e_2 = (u, w)$), we need an integral solution. In general, an LP is only guaranteed to return an integral solution in special situations, such as all vertices of its polytope/feasibility region having integer coordinates. This is known to be the case for flow problems by the Flow Integrality Theorem, but then for flow problems it will be faster to use a dedicated flow algorithm. Otherwise you should have a convincing argument why you expect the LP to have an integral solution.

Therefore we will focus on using **BGL** for the task at hand.

Graph modeling The input graph is given as an edge list with p directed edges. Other than that there are no other specifications (although there is in fact a warning that we might have loops). Since you never ever (really, we mean it!) should make any assumptions/interpretations that are not clearly supported by what is written, the right graph representation immediately follows: We have to use an `adjacency_list` with `vecS` as OutEdgeList type and `directedS` as directivity type.
(Why? `adjacency_matrix` doesn't support parallel edges, and neither does `adjacency_list` with `setS` as OutEdgeList type. `bidirectionalS` results in a considerable overhead – you should be wary of the consequences of the choice, and you don't need access to `in_edges(v,G)` anyways.)

3 Algorithm Design

Once the story of the problem is reduced to the graph with vertex supplies and demands as well as minimum and maximum edge capacities, the setting already sounds pretty familiar: While we don't have a flow problem, the task looks very similar to one. So let us quickly check that we're on the right track. From the problem description we learn that our graph has at most $l \leqslant 500$ vertices and $l^2 \leqslant 25 \cdot 10^4$ edges of capacity at most 10^6, hence `push_relabel_max_flow` will be just below the timelimit.

Suppose for a minute that we have the same problem *without the minimum requirements*. We are left with
(i) multiple "sources" (all l vertices) with a certain amount of flow "to give" (supply),
(ii) multiple "sinks" (all l vertices) that "want" a certain amount of flow (the demand),
(iii) edges e_j with maximum capacity C_j.
Hence we are faced with a circulation problem, i.e., one of the flow applications presented in the BGL Flow Tutorial. To gain more insight we now focus on the subtask specifications.

Subtask specifications

1. For the first group of test sets, there are no minimum requirements for the edges.
2. In the second group, each vertex has either only incoming edges or only outgoing edges.
3. For all other test sets, there are no additional assumptions.

First subtask: No minimum requirements As already mentioned, we have a circulation problem. By adding a super-source s and a super-sink t with edges of maximum capacity corresponding to the demands and supplies, respectively, we can check whether all demands can be satisfied by comparing the maximum flow amount to the sum $x = \sum d_i$ of all demands. We have a satisfying soldier movement in the graph if and only if `maxflow` $\geqslant x$.

Second subtask: Bipartite directed graph For the second subtask we have a vertex bipartition $V(G) = A \dot\cup B$ such that for all edges $e_j = (f_j, t_j)$ we have $f_j \in A$ and $t_j \in B$. In particular there are no loops and the input specification

> "Note that a single soldier may use the same path multiple times and each time counts as one traversal."

becomes irrelevant – this can simply not occur. While there is no particular method tailored for this subtask (i.e., a method which solves the subtask but not the complete problem), this setting is probably more accessible to analyze in order to come up with a full solution.

Full solution: Modeling minimum capacities Can we extend the circulation problem to enforce minimum edge capacities? If so, we are done. Is there any hope to extend a flow problem to account for minimum flow requirements for each edge? Yes! Recall the BGL Flow Tutorial: In the "Common Tricks" Section we've looked at several generalizations over a standard s-t-flow network and how to approach them. So far we've used the super-source/super-sink idea, while vertex capacities and undirected edges do not appear in our problem. *How to deal with minimum flow requirements was left as an exercise!*

One could think of the following heuristic: Compute a standard maximum flow from the super-source s to the super-sink t for the circulation problem with non-modified maximum edge capacities C_j and *check* whether the minimum flow requirement of each edge is satisfied. The problem is that this does not *enforce* the satisfiability of all minimum flow requirements.

What does it mean exactly that there is a minimum flow requirement of c_j from f_j to t_j? We can reformulate it, equivalently stating that

(i) a flow amount of c_j *must* leave f_j,
(ii) a flow amount of c_j *can* arrive at t_j,[1]
(iii) an additional, variable flow amount of $C_j - c_j$ *might* flow from f_j to t_j.

We can implement (iii) by modifying the edge (f_j, t_j) to have maximum capacity $C_j - c_j$ and no minimum capacity. The flow amount of c_j arriving at t_j by property (ii) now has to come from somewhere else: we add an edge (s, t_j) of capacity c_j. Most importantly, by (i) we have to accomodate that a flow amount of c_j *must* leave f_j. We do this by adding an edge (f_j, t) of capacity c_j, however *this does not yet enforce* that there is actually a flow amount of c_j through this edge. Recall that our original question was

> "Is there a maxflow of size $\geqslant x$ that satisfies all minimum edge flow requirements (where x is the sum of all demands, $x = \sum_{0 \leqslant i < l} d_i$)?"

In our modified setting, all the demands must still be satisfied, i.e., all the edges (i, t), $0 \leqslant i < l$ of capacity d_i must be fully saturated. On the other hand, property (i) also has to be satisfied, i.e., all edges (f_j, t) also have to be fully saturated.[2] Hence we have to compare `maxflow` not only to the sum of demands, but additionally to c_j:

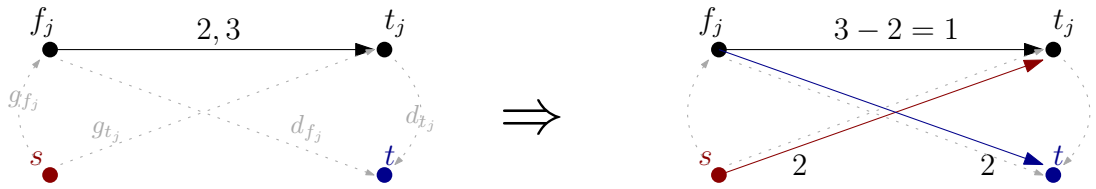

Figure 1: (left) edge with min capacity 2, max capacity 3. Question: Is `maxflow` $\geqslant x$?
(right) modified edge with maximum capacity $3 - 2 = 1$.
Additional edges of maximum capacity 2. Q: Is `maxflow` $\geqslant x + 2$?

If we do this for all edges, we will have to check:

> "Is there a flow of size `maxflow` $\geqslant \sum_{0 \leqslant i < l} d_i + \sum_{0 \leqslant j < p} c_j$?"

Finally, let us mention that we did not increase the number of vertices (only the number of edges), hence we can expect `push_relabel_max_flow` to have a similar running time as before.

Brain Teaser: Min Cost Max Flow Try to come up with a solution using a minimum cost maximum flow using the `cycle_canceling` algorithm instead of using the modifications presented above. Do you expect this solution to pass the timelimits? If so, for which subtasks?

[1] Strictly speaking, the flow amount of c_j *must* arrive at t_j – however, in the following construction we cannot check this, since the flow amount might come either from s or via the excess edge (f_j, t_j) of capacity $C_j - c_j$.
[2] This is in contrast to the edges (f_j, t) which, according to Footnote 1, are not necessarily fully saturated!

4 Implementation

The implementation itself is pretty standard – you can follow the provided sample code from the BGL Flow Tutorial. Let us just mention a few details:

1. We need to create flow edges in several places of our code, hence it is convenient to have a custom addEdge function, or even better an EdgeAdder class.
2. To add flow edges we need access to the two internal properties edge_reverse_t and edge_capacity_t, hence we use typedefs for the corresponding internal property maps.

5 A Complete Solution

```
// STL iostream
#include <iostream>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

// BGL typedefs
typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> Traits;
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
    boost::property<boost::edge_capacity_t, long,
        boost::property<boost::edge_residual_capacity_t, long,
            boost::property<boost::edge_reverse_t, Traits::edge_descriptor> > > >       Graph;
typedef boost::property_map<Graph, boost::edge_capacity_t>::type               EdgeCapacityMap;
typedef boost::property_map<Graph, boost::edge_reverse_t>::type         ReverseEdgeMap;
typedef boost::graph_traits<Graph>::vertex_descriptor                   Vertex;
typedef boost::graph_traits<Graph>::edge_descriptor                     Edge;

// Custom EdgeAdder class
class EdgeAdder {
        Graph &G;
        EdgeCapacityMap &capacitymap;
        ReverseEdgeMap  &revedgemap;

public:
        // to initialize the Object
        EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap,
                        ReverseEdgeMap &revedgemap):
                G(G), capacitymap(capacitymap), revedgemap(revedgemap){}
        // to use the Function (add an edge)
        void addEdge(int from, int to, long capacity) {
                Edge e, reverseE;
                bool success;
                tie(e, success) = boost::add_edge(from, to, G);
                tie(reverseE, success) = boost::add_edge(to, from, G);
                capacitymap[e] = capacity;
                capacitymap[reverseE] = 0;
                revedgemap[e] = reverseE;
                revedgemap[reverseE] = e;
        }
};
// Function for a single testcase
void testcases() {
        // Graph
        int l, p;       std::cin >> l >> p;
```

```
        Graph G(l);
        Vertex src = boost::add_vertex(G);
        Vertex sink = boost::add_vertex(G);

        // Maps & custom EdgeAdder
        EdgeCapacityMap capacitymap = boost::get(boost::edge_capacity, G);
        ReverseEdgeMap  revedgemap = boost::get(boost::edge_reverse, G);
        EdgeAdder eaG(G, capacitymap, revedgemap);

        // supplies & demands
        long totaldemand = 0;
        for (int i = 0; i < l; ++i) {
                long gi, di;
                std::cin >> gi >> di;
                eaG.addEdge(src, i, gi);
                eaG.addEdge(i, sink, di);
                totaldemand += di;
        }

        // paths and capacities
        long totalmincap = 0;
        for (int j = 0; j < p; ++j) {
                int fj, tj;
                long mincj, maxCj;
                std::cin >> fj >> tj >> mincj >> maxCj;
                eaG.addEdge(src, tj, mincj);
                eaG.addEdge(fj, sink, mincj);
                eaG.addEdge(fj, tj, maxCj-mincj);
                totalmincap += mincj;
        }

        // check whether demands and minimum capacities are satisfied
        long flow = push_relabel_max_flow(G, src, sink);
        if (flow >= totaldemand + totalmincap)  std::cout << "yes\n";
        else                                    std::cout << "no\n";
}

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