Flow Report

abax, bemn, diem, emba, mkkr, sebni

October 27, 2020

Results

Our implementation successfully computes a flow of 163 on the input file, confirming the analysis of the American enemy.

We have analysed the possibilities of decreasing the capacities near Minsk. Our analysis is summaried in the following table:

Case	4W–48	4W–49	Effect on flow
1	30	20	no change
2	20	30	no change
3	20	20	no change
4	10	20	-10
5	20	10	-10
6	10	10	-20
7	0	10	-30
8	10	0	-30
9	0	0	-40

In case 4, the bottleneck is

10–25, 11–25, 11–24, 12–23, 18–22, 20–23, 20–21, 27–26, 28–30

The comrade from Minsk is advised to protect his base, since attacks on its railways may result in a considerable impact for the motherland. However a better analysis on the interpretation should be delegated to military experts, we do not take responsability for any lost lifes that may happen due to our interpretation.

Implementation details

We use a straightforward implementation of Ford Fulkerson.
The implementation is split into three files: `mincut.py`, `flow.py` and `fordfulkerson.py`. The first simply prints the minimum cut and the maximum flow. The second one parses the input, and decides on IDs for nodes and edges. The last, implements the actual algorithm.
We use DFS to to find an augmenting path. Our DFS implementation returns a path in reverse order, which doesn't matter as we have to traverse the entire path when traversing. And we can do it without reversing it. We also do not rebuild the entire G_f (residual graph)

after augmentation, instead we only update the edges that are part of the path augmented. [1].
The running time is $O(Ef)$ where f is the max-flow of the graph, since capacities are integers and this implies that at each step, the flow will be increased at least one, and running DFS takes $O(E)$.
No better bound is guaranteed like in the Edmonds Karp variation, since getting the path is done through DFS. A bound that does not depend on f is as given on K&T: $O(EC)$ where C is the sum of all capacities out from the source node, since clearly $f < C$.
While we did eliminate the need to reconstruct the entire graph after augmenting a path, this does not affect the running time (worst case), since the augmented path may traverse the entire graph (Thereby making us update the entire graph).
The undirected edges are represented by constructing two directed edges (in both G and G_f). Edge `capacity`, `flow`, and `start` and `termination` node (ids) are stored in a separate datatype (`Edge`):

```
class Edge:
def __init__(self, id, s, t, c):
    self.s = s
    self.t = t
    self.f = 0
    self.c = c
    self.id = id
```

It is here important to note that edges point to node IDs, which can be looked up in both G and G_f (the same goes for nodes pointing to edge IDs). The direction is visible by what is the start and termination node. Secondly, the backwards edges are distinguishable by their negative IDs.

[1] The backwards edges in G_f are constructed by taking the negative id of the edge in G. IDs are controlled by the user of the algorithm, and the user should here be aware to only provide positive ids to edges, starting from 1.