

Algolab BGL Flows

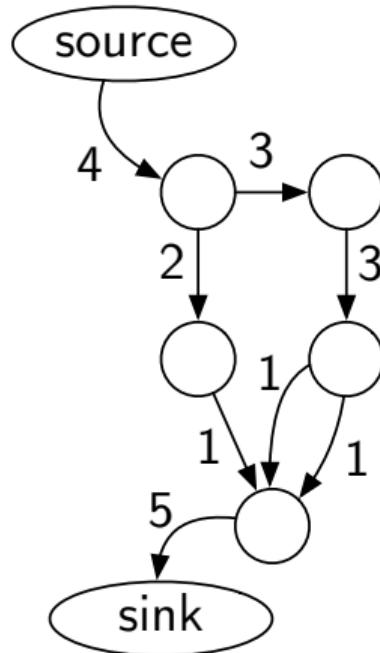
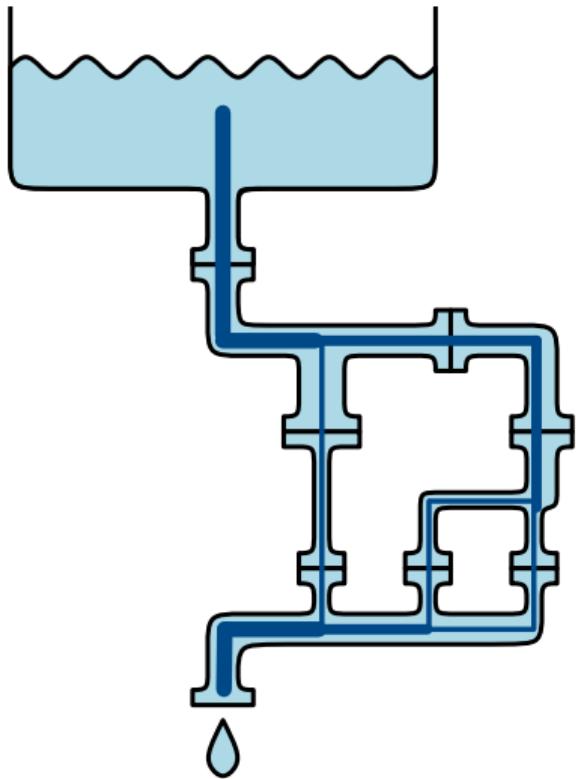
Stefano Leucci

Slides by Daniel Graf

ETH Zürich

October 24, 2018

Network Flow: Example

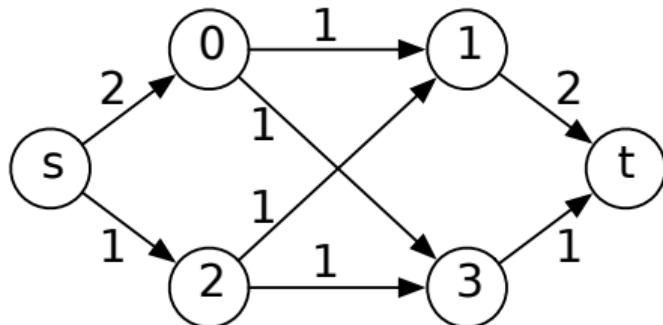


Network Flow: Problem Statement

Input: A flow network consisting of

- ▶ directed graph $G = (V, E)$
- ▶ source and sink $s, t \in V$
- ▶ edge capacity $c : E \rightarrow \mathbb{N}$.

$$n = |V|, m = |E|$$



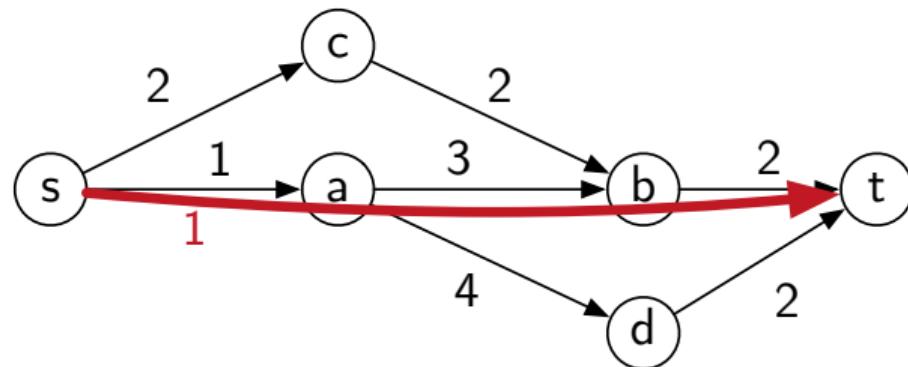
Output: A flow function $f : E \rightarrow \mathbb{N}$ such that

- ▶ all capacity constraints are satisfied:
 $\forall uv \in E: 0 \leq f(uv) \leq c(uv)$
(no pipe is overflowed)
- ▶ flow is conserved at every vertex:
 $\forall u \in V \setminus \{s, t\}:$
 $\sum_{vu \in E} f(vu) = \sum_{uv \in E} f(uv)$
(no vertex is leaking)
- ▶ the *total flow* $|f|$ is maximized*:
 $|f| := \sum_{sv \in E} f(sv) = \sum_{ut \in E} f(ut)$
(s -out-flow equals t -in-flow)

* Assuming that there are no edges into s and out of t .

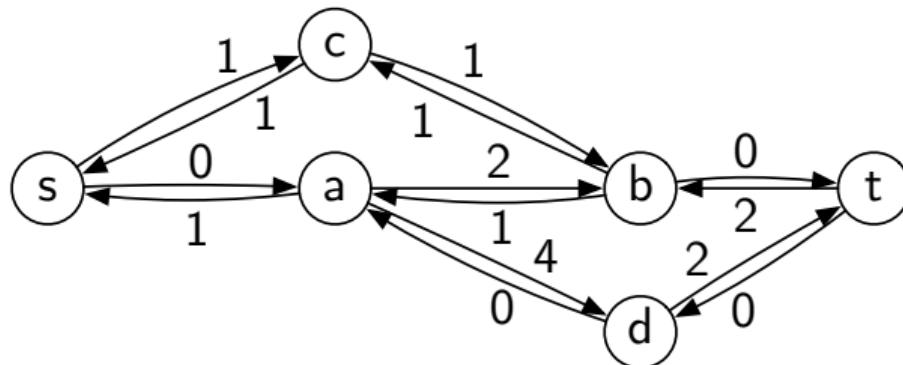
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- ▶ Take any $s-t$ -path and increase the flow along it.
- ▶ Update capacities and repeat as long as we can.
- ▶ Are we done? Problem: We can get stuck at a local optimum.
Maximal \neq maximum, remember?
"Greedy never works!" (unless you prove it)



Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- ▶ Solution: Keep track of the flow and allow paths that *reroute* units of flow. These are called *augmenting paths* in the *residual network*.
- ▶ Ford-Fulkerson (1955): Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- ▶ Edmonds-Karp (1972) / Dinitz (1970): Repeatedly take any *shortest* (as in: fewest edges) augmenting path: running time: best of $\mathcal{O}(m|f|)$ and $\mathcal{O}(m(nm))$ [BGL-Doc].
Simple bound for total flow: $|f| \leq n \max c$



Brief Excursion: Pseudo-Polynomial Running Times

Why is $\mathcal{O}(nm \max c)$ in general considered to be worse than $\mathcal{O}(nm^2)$?

- ▶ $\mathcal{O}(nm \max c)$ is what we call a *pseudo-polynomial* running time.
It depends on the largest integer present in the input.
- ▶ With i bits in the input, we can represent capacities up to 2^i .
Thus such an algorithm might take exponential time.
- ▶ This makes us depend on the input being *nice*, i.e. all capacities being small.

This does not make the bound meaningless:

- ▶ If e.g. $\max c = 1$, then it is good to know that both algorithms run in $\mathcal{O}(nm)$.

Using BGL flows: Includes

The usual headers:

```
1 // STL includes
2 #include <iostream>
3 #include <vector>
4 #include <algorithm>
5
6 // BGL includes
7 #include <boost/graph/adjacency_list.hpp>
8 #include <boost/graph/push_relabel_max_flow.hpp>
9 #include <boost/graph/edmonds_karp_max_flow.hpp>
10
11 // Namespaces
12 using namespace std;
13 using namespace boost;
```

Using BGL flows: Includes

The usual headers:

```
1 // STL includes
2 #include <iostream>
3 #include <vector>
4 #include <algorithm>
5
6 // BGL includes
7 #include <boost/graph/adjacency_list.hpp>
8 #include <boost/graph/push_relabel_max_flow.hpp>
9 #include <boost/graph/edmonds_karp_max_flow.hpp>
10
11 // Namespaces
12 // using namespace std; // try to avoid if you don't mind writing
13 // using namespace boost; // some extra letters
```

Another Brief Excursion: Why to be careful with the using directive?

Why should you be careful with the `using namespace std/boost/CGAL` directive?

- ▶ code readability: is this an entity that is defined locally or in one (of many) libraries?
- ▶ name lookup: simple names quickly get ambiguous
 - ▶ which queue? `std::queue` or `boost::queue`?
 - ▶ which source? a local variable called `source` or the `boost::source` function?

Using BGL flows: Typedefs

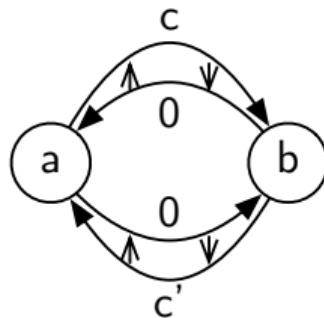
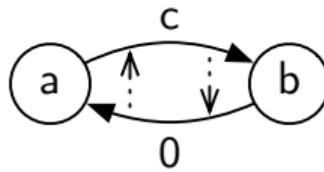
The typedefs now include residual capacities and reverse edges:

```
14 // Graph Type with nested interior edge properties for Flow Algorithms
15 typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> Traits;
16 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
17     boost::property<boost::edge_capacity_t, long,
18     boost::property<boost::edge_residual_capacity_t, long,
19     boost::property<boost::edge_reverse_t, Traits::edge_descriptor>
20     > > > Graph;
21
22 // Interior Property Maps
23 typedef boost::property_map<Graph, boost::edge_capacity_t>::type EdgeCapacityMap;
24 typedef boost::property_map<Graph, boost::edge_residual_capacity_t>::type ResidualCapacityMap;
25 typedef boost::property_map<Graph, boost::edge_reverse_t>::type ReverseEdgeMap;
26 typedef boost::graph_traits<Graph>::vertex_descriptor Vertex;
27 typedef boost::graph_traits<Graph>::edge_descriptor Edge;
```

Using BGL flows: Creating an edge

Helper function to add a directed edge and its reverse in the residual graph:

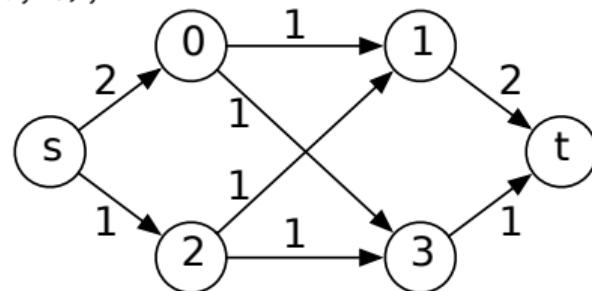
```
26 void addEdge(int from, int to, long capacity,
27   EdgeCapacityMap &capacitymap,
28   ReverseEdgeMap &revedgemap,
29   Graph &G)
30 {
31   Edge e, rev_e;
32   bool success;
33   boost::tie(e, success) = boost::add_edge(from, to, G);
34   boost::tie(rev_e, success) = boost::add_edge(to, from, G);
35   capacitymap[e] = capacity;
36   capacitymap[rev_e] = 0; // reverse edge has no capacity!
37   revedgemap[e] = rev_e;
38   revedgemap[rev_e] = e;
39 }
```



Using BGL flows: Creating the graph

Get the properties and insert the edges:

```
39 // Create Graph and Maps
40 Graph G(4);
41 EdgeCapacityMap capacitymap = boost::get(boost::edge_capacity, G);
42 ReverseEdgeMap revedgemap = boost::get(boost::edge_reverse, G);
43 ResidualCapacityMap rescapacitymap
    = boost::get(boost::edge_residual_capacity, G);
44
45
46 addEdge(0, 1, 1, capacitymap, revedgemap, G);
47 addEdge(0, 3, 1, capacitymap, revedgemap, G);
48 addEdge(2, 1, 1, capacitymap, revedgemap, G);
49 addEdge(2, 3, 1, capacitymap, revedgemap, G);
50
51 Vertex source = boost::add_vertex(G);
52 Vertex target = boost::add_vertex(G);
53 addEdge(source, 0, 2, capacitymap, revedgemap, G);
54 addEdge(source, 2, 1, capacitymap, revedgemap, G);
55 addEdge(1, target, 2, capacitymap, revedgemap, G);
56 addEdge(3, target, 1, capacitymap, revedgemap, G);
```



Using BGL flows: adding edges

Simplify addEdge function by capturing the property maps in an EdgeAdder object:

```
26 void addEdge(int from, int to, long capacity,
27   EdgeCapacityMap &capacitymap,
28   ReverseEdgeMap &revedgemap,
29   Graph &G)
30 {
31   Edge e, rev_e;
32   bool success;
33   boost::tie(e, success) = boost::add_edge(from, to, G);
34   boost::tie(rev_e, success) = boost::add_edge(to, from, G);
35   capacitymap[e] = capacity;
36   capacitymap[rev_e] = 0; // reverse edge has no capacity!
37   revedgemap[e] = rev_e;
38   revedgemap[rev_e] = e;
39 }

26 class EdgeAdder {
27   Graph &G;
28   EdgeCapacityMap &capacitymap;
29   ReverseEdgeMap &revedgemap;
30
31 public:
32   // to initialize the Object
33   EdgeAdder(Graph & G,
34             EdgeCapacityMap &capacitymap,
35             ReverseEdgeMap &revedgemap):
36     G(G),
37     capacitymap(capacitymap),
38     revedgemap(revedgemap){}
39
40   // to use the Function (add an edge)
41   void addEdge(int from, int to, long capacity) {
42     Edge e, rev_e;
43     bool success;
44     boost::tie(e, success) = boost::add_edge(from, to, G);
45     boost::tie(rev_e, success) = boost::add_edge(to, from, G);
46     capacitymap[e] = capacity;
47     capacitymap[rev_e] = 0; // reverse edge has no capacity!
48     revedgemap[e] = rev_e;
49     revedgemap[rev_e] = e;
50   }
51 };
```

Using BGL flows: Creating the graph (cleaned up)

Simplified graph creation with EdgeAdder object:

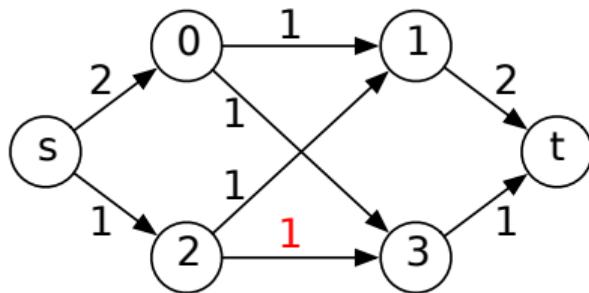
```
39 // Create Graph and Maps
40 Graph G(4);
41 EdgeCapacityMap capacitymap = get(edge_capacity, G);
42 ReverseEdgeMap revedgemap = get(edge_reverse, G);
43 ResidualCapacityMap rescapacitymap
44     = get(edge_residual_capacity, G);
45
46
47 addEdge(0, 1, 1, capacitymap, revedgemap, G);
48 addEdge(0, 3, 1, capacitymap, revedgemap, G);
49 addEdge(2, 1, 1, capacitymap, revedgemap, G);
50 addEdge(2, 3, 1, capacitymap, revedgemap, G);
51
52 Vertex source = add_vertex(G);
53 Vertex target = add_vertex(G);
54 addEdge(source, 0, 2, capacitymap, revedgemap, G);
55 addEdge(source, 2, 1, capacitymap, revedgemap, G);
56 addEdge(1, target, 2, capacitymap, revedgemap, G);
57 addEdge(3, target, 1, capacitymap, revedgemap, G);

50 // Create Graph and Maps
51 Graph G(4);
52 EdgeCapacityMap capacitymap = get(edge_capacity, G);
53 ReverseEdgeMap revedgemap = get(edge_reverse, G);
54 ResidualCapacityMap rescapacitymap
55     = get(edge_residual_capacity, G);
56 EdgeAdder eaG(G, capacitymap, revedgemap);
57
58 eaG.addEdge(0, 1, 1);
59 eaG.addEdge(0, 3, 1);
60 eaG.addEdge(2, 1, 1);
61 eaG.addEdge(2, 3, 1);
62
63 Vertex source = add_vertex(G);
64 Vertex target = add_vertex(G);
65 eaG.addEdge(source, 0, 2);
66 eaG.addEdge(source, 2, 1);
67 eaG.addEdge(1, target, 2);
68 eaG.addEdge(3, target, 1);
```

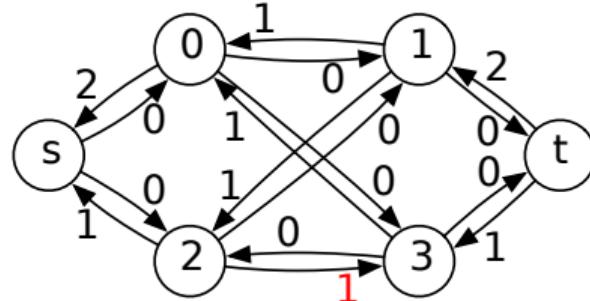
Using BGL flows: Calling the algorithm

```
1 long flow = boost::edmonds_karp_max_flow(G, source, target);
2 // The flow algorithm uses the interior properties
3 // - edge_capacity, edge_reverse (read access),
4 // - edge_residual_capacity (read and write access).
5 // The residual capacities of this flow are now accessible through rescapacitymap.
```

Input: (with reverse edges not drawn)



Residual capacities:



If you want the flow value $f(e)$, compute $\text{capacitymap}[*e] - \text{rescapacitymap}[*e]$

Using BGL flows: Calling a different algorithm

```
1 #include <boost/graph/push_relabel_max_flow.hpp>
2 long flow = boost::push_relabel_max_flow(G, source, target);
```

Using a different flow algorithm is very easy. Just replace the header and function call.

The Push-Relabel Max-Flow algorithm by Goldberg and Tarjan (1986) [\[BGL-Doc\]](#) is almost always the best option with running time $\mathcal{O}(n^3)$.

Recall: Edmonds-Karp Max-Flow algorithm requires time $\mathcal{O}(\min(m|f|, nm^2))$.

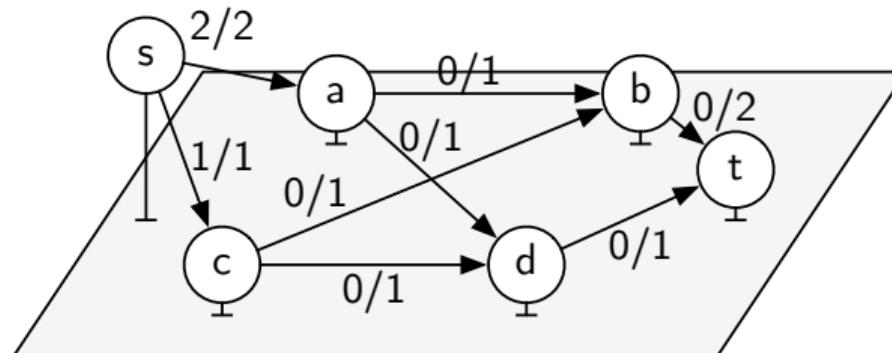
But even if $m, |f| \in \mathcal{O}(n)$, we usually observe that Push-Relabel is at least as fast.

Never say never: Tailoring graphs s.t. Edmonds-Karp beats Push-Relabel is artificial, but possible.

Network Flow Algorithms: Push-Relabel Max-Flow

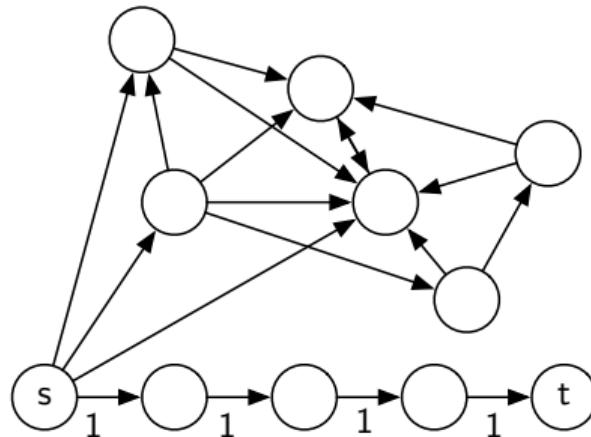
Intuition: (not really needed for using it)

- ▶ Augment flow locally edge by edge instead of augmenting paths.
- ▶ Use height label to ensure that the flow is consistent and maximum in the end.
- ▶ Push step: increase flow along a downward out-edge of any overflowed vertex.
- ▶ Relabel step: increase the height of a vertex so that a push is possible afterwards.



Network Flow Algorithms: Push-Relabel Max-Flow "Worst-Case"

Intuition: how does a graph look like where Edmonds-Karp outperforms Push-Relabel?



"Push-Relabel Killer"

s - t -path + disjoint s -reachable random graph
on a graph with $n \approx 200'000$, $m \approx 600'000$
observed speed difference: $\approx 3x$ (0.6s vs. 0.2s)

But 3x is not as much as you might expect.
Push-Relabel scales much better here than $\Theta(n^3)$.

One last coding slide: Debugging, Execution and Testing

- ▶ compile with optimizations: -O2 can cause big speedups for BGL algorithms
- ▶ minimal compiler errors, -Wall, and -Wextra are your friends
- ▶ cgal_create_cmake_script also works for BGL (see [\[FAQ\]](#) for C++11 issues)
- ▶ use assert(...) to verify your assumptions (add #undef NDEBUG on the judge)
- ▶ use std::cerr for your debug output (does not cause WA on the judge)

- ▶ Pipe Input/Output from/to files:
(no need for copy/pasting)
- ▶ Use diff to check:
(no output = no mistake)

```
1 $ time ./program < t.in > t.myout 2> t.err  
2     real 0m0.349s  
3     user 0m0.335s  
4     sys 0m0.009s
```

```
1 $ diff t.out t.myout  
2      1c1  
3      < 21  
4      ---  
5      > 42
```

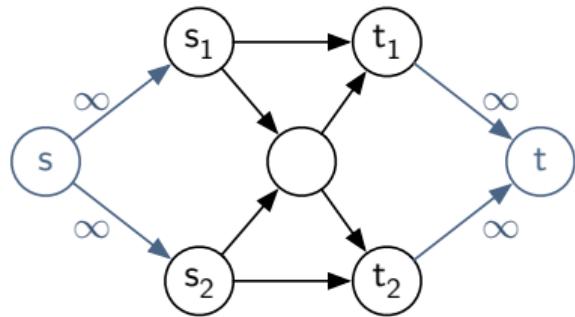
- ▶ One line to do it all: (quick way of running it on all the samples before submitting)

```
1 $ for f in *.in; do echo "$f"; time ./program < $f > ${f%.*}.myout; diff ${f%.*}.out ${f%.*}.myout;  
→ done
```

Common tricks

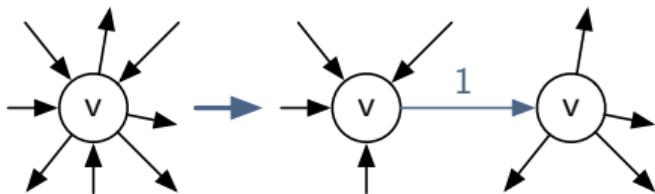
Multiple sources/sinks

with e.g. $\infty \approx \sum_{e \in E} c(e)$



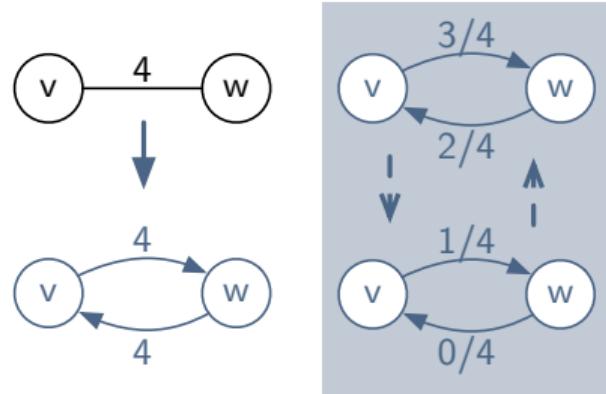
Vertex capacities

split into in-vertex and out-vertex



Undirected graphs

antiparallel edges with flow reducible to one direction



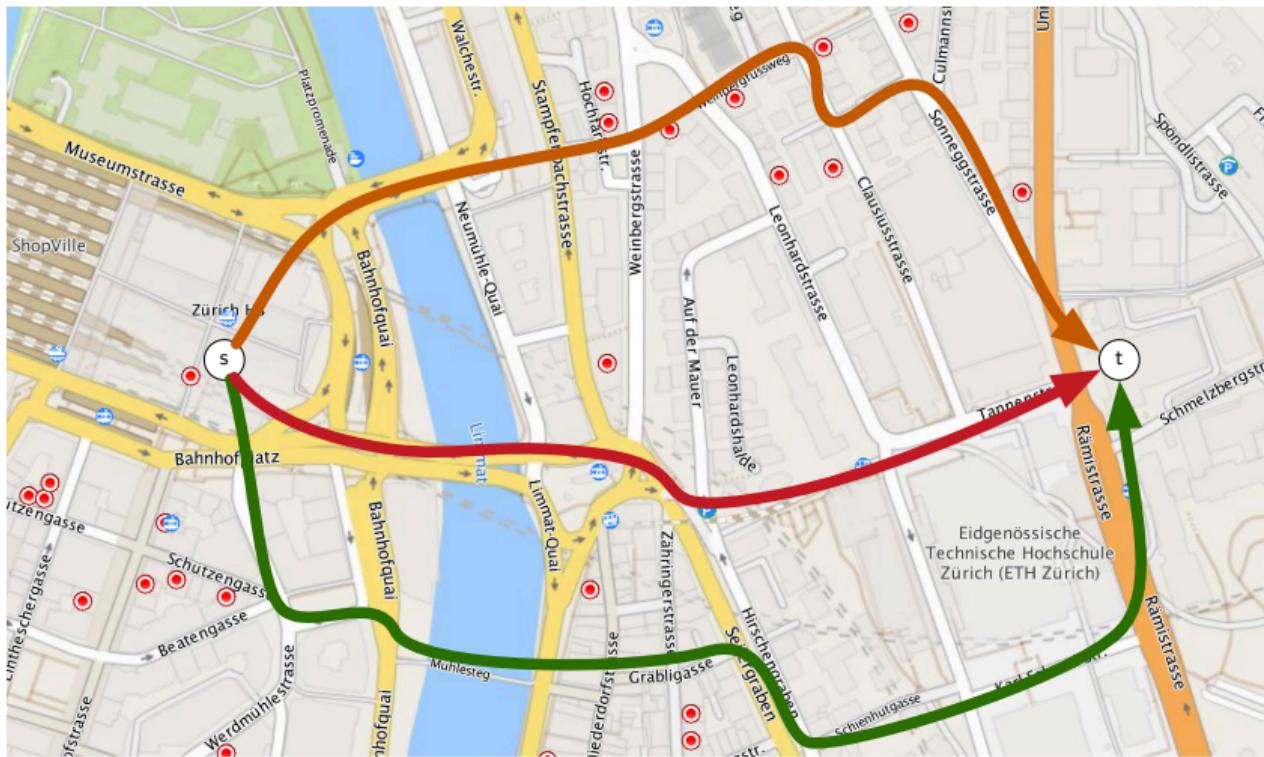
Minimum flow per edge

how to enforce $c_{\min}(e) \leq f(e) \leq c_{\max}(e)$?

[Exercise]

Flow Application: Edge Disjoint Paths

How many ways are there to get from HB to CAB without using the same street twice?



Map:
search.ch,
TomTom,
swisstopo,
OSM

Flow Application: Edge Disjoint Paths

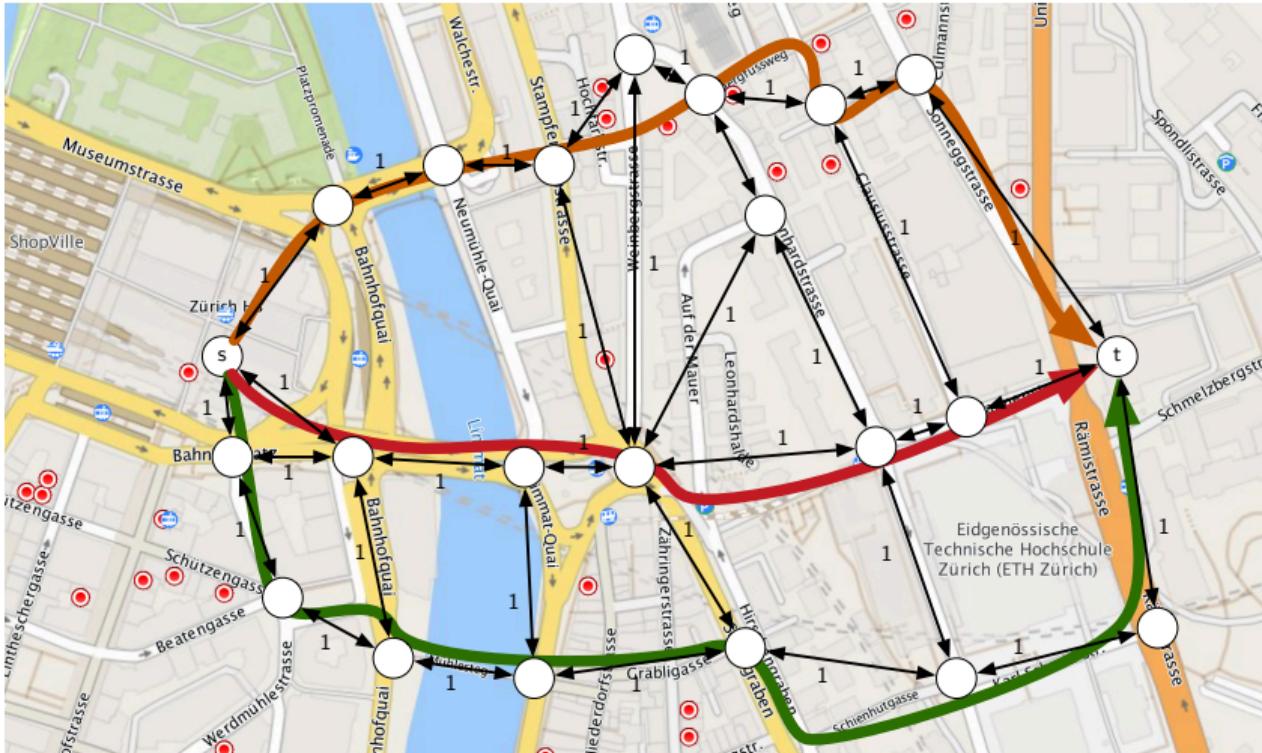
How many ways are there to get from HB to CAB without using the same street twice?

- ▶ Does this look like a flow problem? Not immediately.
- ▶ Can it be turned into a flow problem? Maybe.
- ▶ What we need according to the problem definition
 - ▶ $G = (V, E)$, *directed* street graph by adding edges in both directions for each street.
 - ▶ $s, t \in V$, intersections of HB and CAB.
 - ▶ $c : E \rightarrow \mathbb{N}$ with all capacities set to 1.

Lemma (Flow Decomposition)

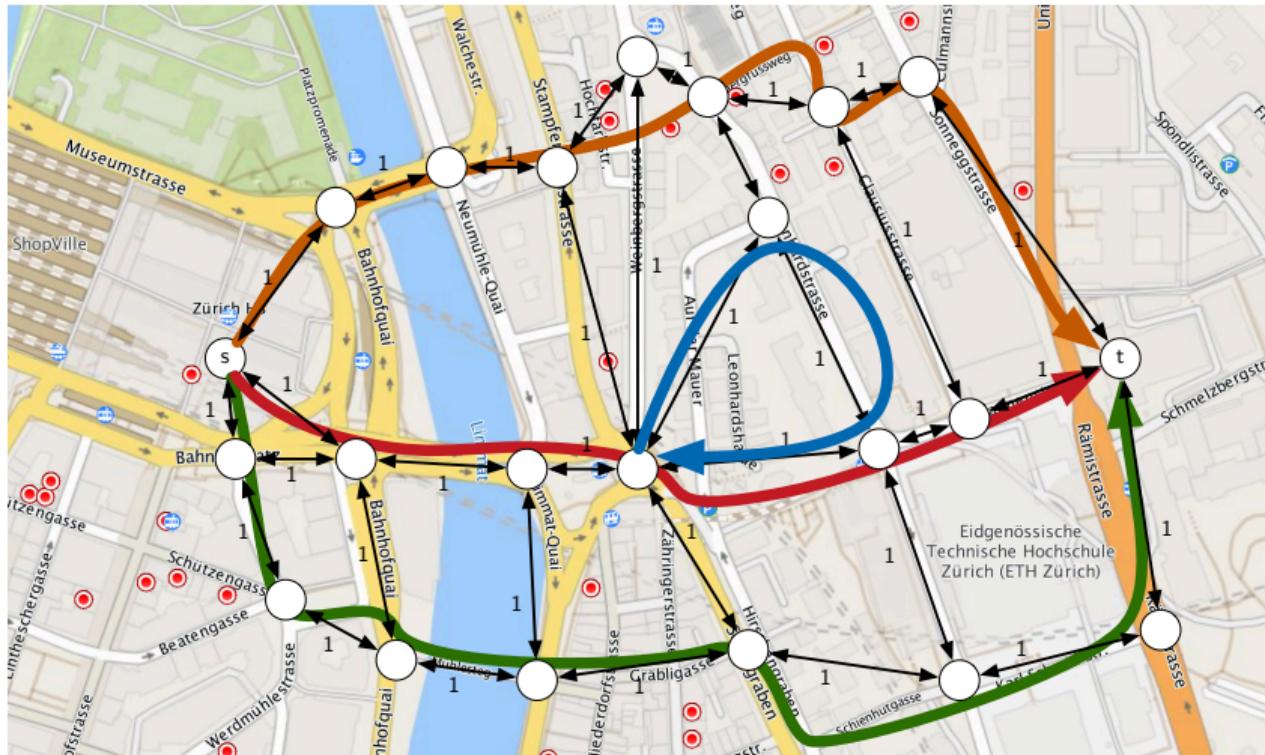
In a directed graph with unit capacities, the maximum number of edge-disjoint s - t -paths is equal to the maximum flow from s to t .

Flow Application: Edge Disjoint Paths



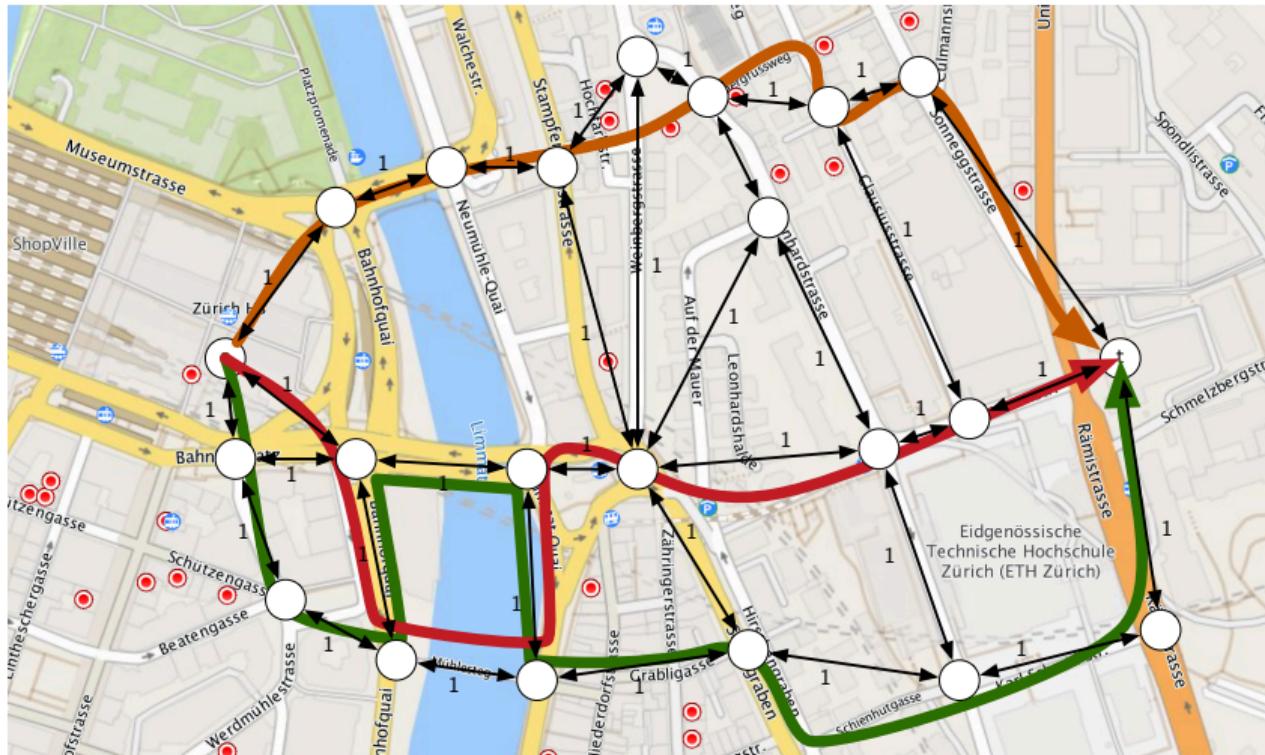
Map:
search.ch,
TomTom,
swisstopo,
OSM

Flow Application: Edge Disjoint Paths: Handle Additional Cycles



Map:
search.ch,
TomTom,
swisstopo,
OSM

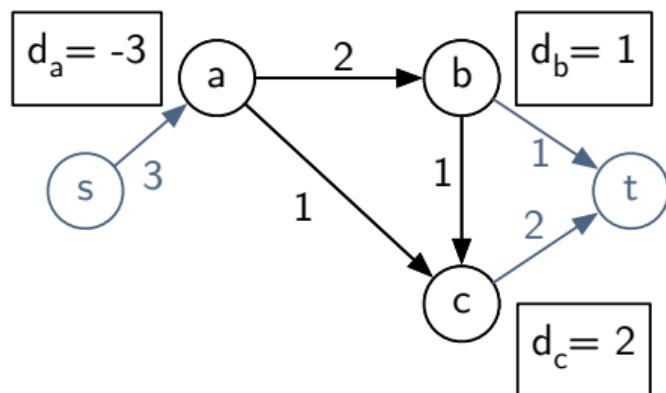
Flow Application: Edge Disjoint Paths: Handle Crossing Paths



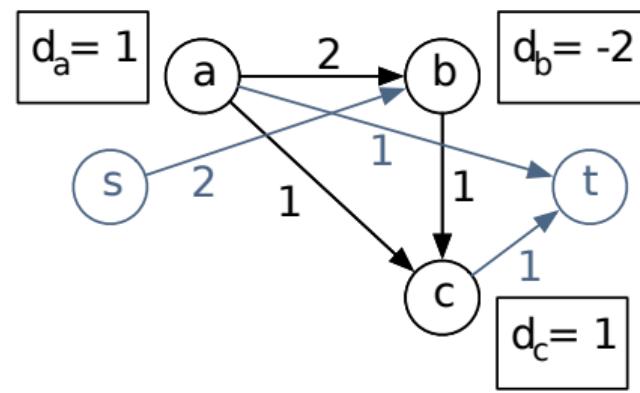
Map:
search.ch,
TomTom,
swisstopo,
OSM

Flow Application: Circulation Problem

- ▶ Multiple sources with a certain amount of flow to give (**supply**).
- ▶ Multiple sinks that want a certain amount of flow (**demand**).
- ▶ Model these as negative or positive demand per vertex d_v .
- ▶ Question: Is there a feasible flow? Surely not if $\sum_{v \in V} d_v \neq 0$. Otherwise?
Add super-source and super-sink to get a maximum flow problem.



feasible flow exists



no feasible flow exists

Tutorial Problem: Soccer Prediction

- ▶ Swiss Soccer Championship, two rounds before the end.
- ▶ 2 points awarded per game, split 1-1 if game ends in a tie.

Team	Points	Remaining Games
FC St. Gallen (FCSG)	37	FCB, FCW
BSC Young Boys (YB)	36	FCW, FCB
FC Basel (FCB)	35	FCSG, YB
FC Luzern (FCL)	33	FCZ, GCZ
FC Winterthur (FCW)	31	YB, FCSG

- ▶ Can FC Luzern still win the Championship? (Assume favourable tie-breaking).
37 points still possible, so yes?
But at least one other team will have to score more, so no?
- ▶ Does this look like a flow problem? Not immediately.

Tutorial Problem: Modelling

Simplifying assumptions: FC Luzern will win all remaining games.

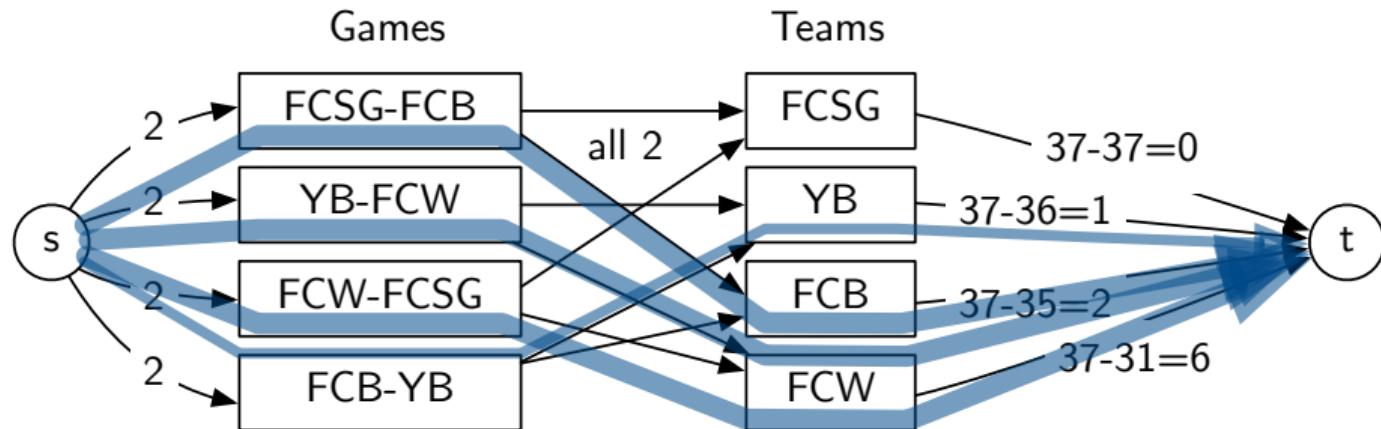
Flow problem modelling:

- ▶ graph with vertices denoting teams and games
- ▶ source, target candidates?
 - ▶ games *distribute* points
 - ▶ teams *receive* points
- ▶ edges between games and teams
- ▶ capacities:
 - ▶ 2 points to distribute per game
 - ▶ $37 - x$ limit on the points received by a team with score x

Final question: Can we let the points *flow* from the games to the teams so that all the teams end up with at most 37 points?

Tutorial Problem: Modelling

Final question: Can we let the points *flow* from the games to the teams so that all the teams end up with at most 37 points?



The answer is yes if and only if the maximum flow is $\#points = 2 \cdot \#\text{games} = 8$.
Note: This approach fails for the current system with 3:0, 1:1, 0:3 splits.

Tutorial Problem: Analysis

- ▶ Does it look like a flow problem now? Yes.
- ▶ What does a unit of flow stand for? A point in the soccer ranking.
- ▶ How large is this flow graph?

For N teams, M games, we have $n = 2 + N + M$ nodes and $m = N + 3M = \mathcal{O}(n)$ edges.
Flow is at most $2M = \mathcal{O}(n)$.

- ▶ What algorithm should we use?
Push-Relabel Max-Flow runs in $\mathcal{O}(n^3)$.
Edmonds-Karp Max-Flow runs in $\mathcal{O}(m|f|) = \mathcal{O}(n^2)$.
Push-Relabel Max-Flow is still faster in practice.

Flow Problems: Summary and Outlook

Summary

What we have seen today:

- ▶ "classical" network flow problem
- ▶ graph modifications for variations
- ▶ solves edge-disjoint paths
- ▶ solves circulation

What you will see in the problems:

- ▶ more tricks and combinations with other techniques like binary search

CGAL-Outlook

Flow-related in the next CGAL lecture:

- ▶ **linear programming** formulation
(equivalent, but slower
as we use a more powerful tool)

BGL-Outlook

What we will see in the next BGL lecture:

- ▶ relation to **minimum cuts**
- ▶ extras for **bipartite graphs**: max matchings, independent sets, **min vertex cover**
- ▶ flows with **cost**: Min Cost Max Flow problem, applications and algorithms