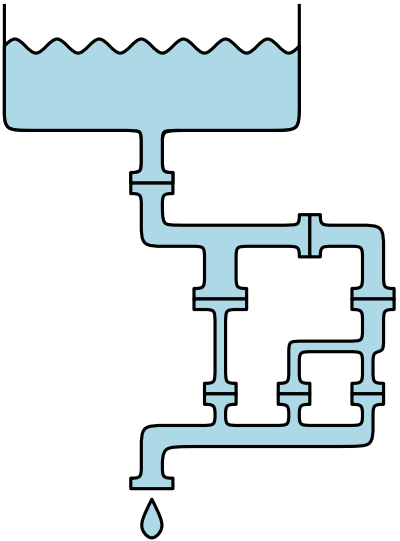
Algolab BGL Flows

Daniel Graf

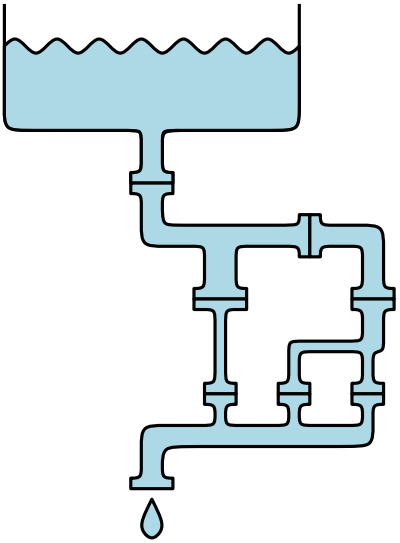
ETH Zürich

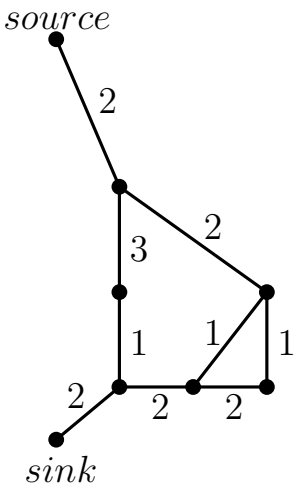
October 21, 2015

Network Flow: Example



Network Flow: Example

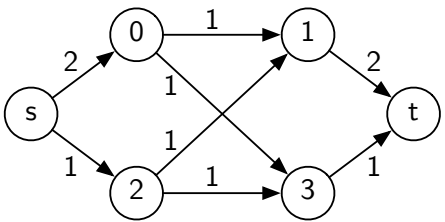




Network Flow: Problem Statement

Input: A flow network consisting of

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

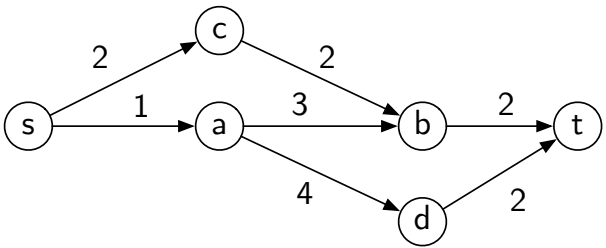


Output: A flow function $f : E \to \mathbb{N}$ such that:

- all capacity constraints are satisfied:
  $\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$
  (no pipe is overflowed)
- flow is conserved at every vertex:
  $\forall u \in V \setminus \{s, t\} :$
  $\sum_{(v,u) \in E} f(v, u) = \sum_{(u,v) \in E} f(u, v)$
  (no vertex is leaking)
- the total flow is maximal:
  $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) =$
  $\sum_{u \in V} f(u, t) - \sum_{u \in V} f(t, u)$

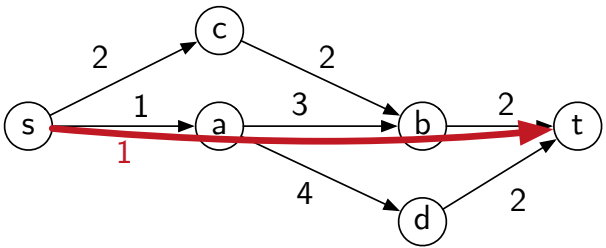
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- Take any s-t-path and increase the flow along it.
- Update capacities and repeat as long as we can.
- Problem: We can get stuck at a local optimum.



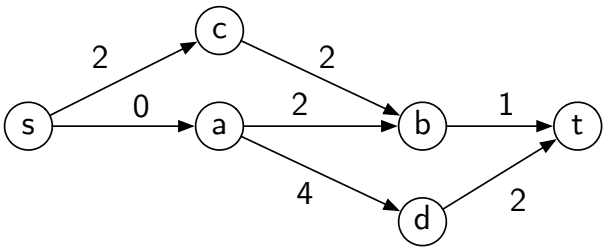
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- Take any s-t-path and increase the flow along it.
- Update capacities and repeat as long as we can.
- Problem: We can get stuck at a local optimum.



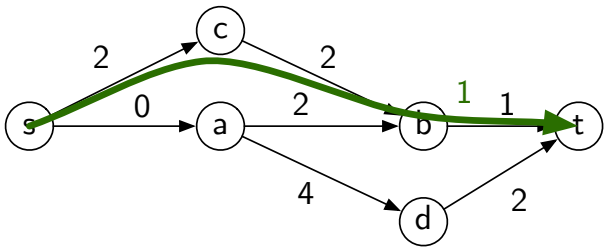
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- Take any s-t-path and increase the flow along it.
- Update capacities and repeat as long as we can.
- Problem: We can get stuck at a local optimum.



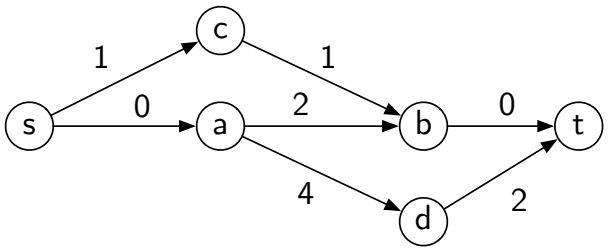
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- Take any s-t-path and increase the flow along it.
- Update capacities and repeat as long as we can.
- Problem: We can get stuck at a local optimum.



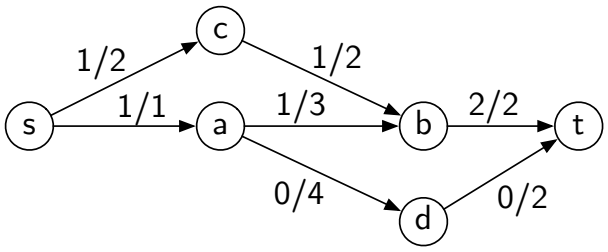
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- Take any s-t-path and increase the flow along it.
- Update capacities and repeat as long as we can.
- Problem: We can get stuck at a local optimum.



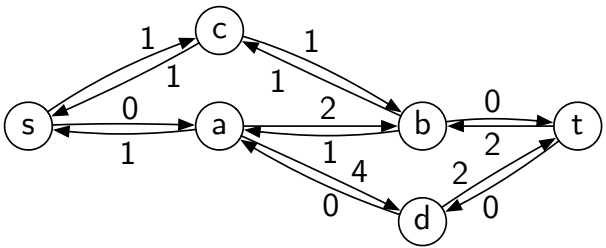
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



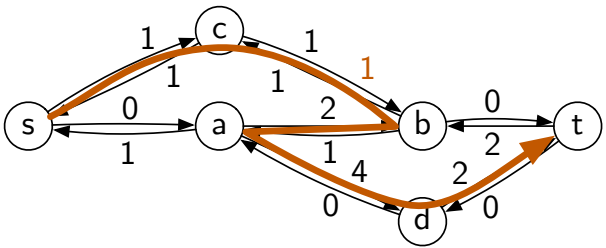
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



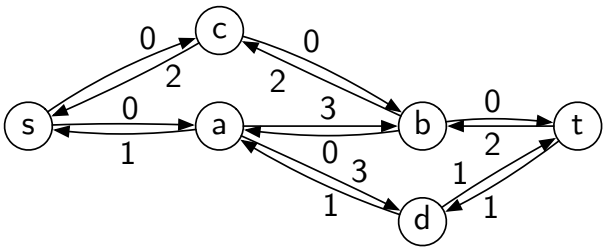
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



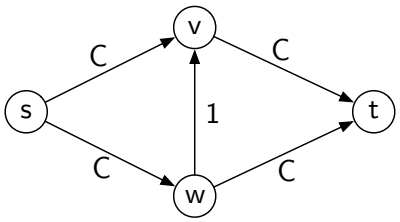
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



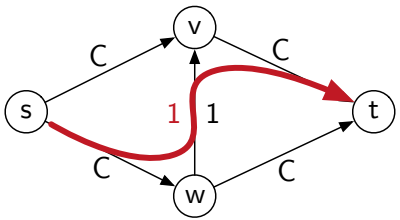
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



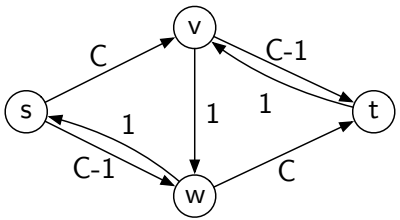
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



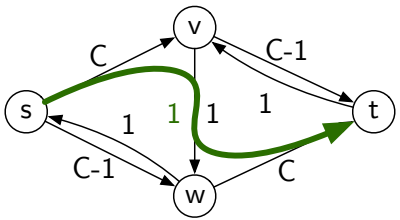
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



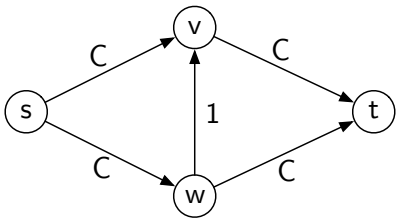
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



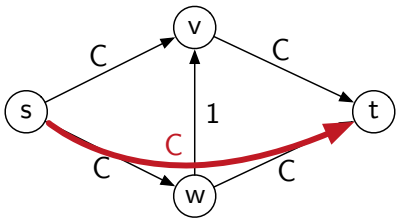
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



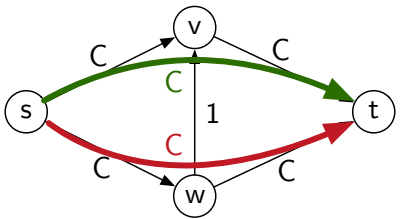
Network Flow Algorithms: Ford-Fulkerson and Edmonds-Karp

- 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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



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: Repeatedly take any augmenting path: running time $\mathcal{O}(m|f|)$.
- Edmonds-Karp: Repeatedly take the shortest augmenting path: running time: best of $\mathcal{O}(m|f|)$, $\mathcal{O}(nm \max c)$ and $\mathcal{O}(nm^2)$. [BGL-Doc].



Using BGL flows: Includes

The usual headers:

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;
```

Using BGL flows: Typedefs

The typedefs now include residual capacities and reverse edges:

⋮

```
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
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;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
```

Using BGL flows: Creating an edge

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

⋮

```
void addEdge(int from, int to, long c,
             EdgeCapacityMap &capacity, ReverseEdgeMap &rev_edge, Graph &G) {
     Edge e, reverseE;
     tie(e, tuples::ignore) = add_edge(from, to, G);
     tie(reverseE, tuples::ignore) = add_edge(to, from, G);
     capacity[e] = c;
     capacity[reverseE] = 0;
     rev_edge[e] = reverseE;
     rev_edge[reverseE] = e;
}
```

Using BGL flows: Creating the graph

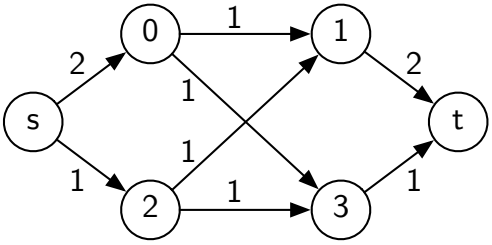
Get the properties and insert the edges:

⋮

```
Graph G(4);
EdgeCapacityMap capacity = get(edge_capacity, G);
ReverseEdgeMap rev_edge = get(edge_reverse, G);
ResidualCapacityMap res_capacity
    = get(edge_residual_capacity, G);

addEdge(0, 1, 1, capacity, rev_edge, G);
addEdge(0, 3, 1, capacity, rev_edge, G);
addEdge(2, 1, 1, capacity, rev_edge, G);
addEdge(2, 3, 1, capacity, rev_edge, G);

Vertex flow_source = add_vertex(G);
Vertex flow_sink = add_vertex(G);
addEdge(flow_source, 0, 2, capacity, rev_edge, G)
addEdge(flow_source, 2, 1, capacity, rev_edge, G)
addEdge(1, flow_sink, 2, capacity, rev_edge, G);
addEdge(3, flow_sink, 1, capacity, rev_edge, G);
```



Using BGL flows: Creating the graph (cleaned up)

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

⋮

```
Graph G(4);
EdgeCapacityMap capacity = get(edge_capacity, G);
ReverseEdgeMap rev_edge = get(edge_reverse, G);
ResidualCapacityMap res_capacity
    = get(edge_residual_capacity, G);
EdgeAdder ea(G, capacity, rev_edge);
ea.addEdge(0, 1, 1);
ea.addEdge(0, 3, 1);
ea.addEdge(2, 1, 1);
ea.addEdge(2, 3, 1);

Vertex flow_source = add_vertex(G);
Vertex flow_sink = add_vertex(G);
ea.addEdge(flow_source, 0, 2);
ea.addEdge(flow_source, 2, 1);
ea.addEdge(1, flow_sink, 2);
ea.addEdge(3, flow_sink, 1);
```

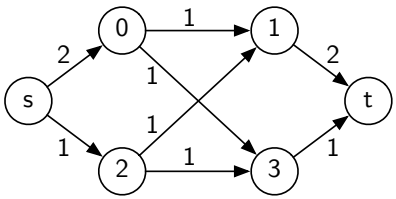
```
struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
            ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity),
      rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, rev;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(rev, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[rev] = 0;
    rev_edge[e] = rev;
    rev_edge[rev] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};
```

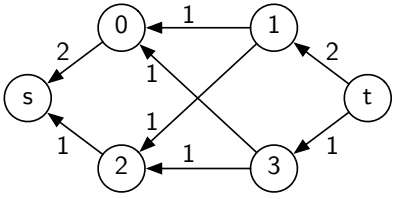
Using BGL flows: Calling the algorithm

```
long flow = edmonds_karp_max_flow(G, flow_source, flow_sink);
// Residual capacity of this flow now accessible throug res_capacity
```

Input: (with reverse edges not drawn)



Residual capacities:



Using BGL flows: Calling a different algorithm

```
#include <boost/graph/push_relabel_max_flow.hpp>
long flow = push_relabel_max_flow(G, source, sink);
// Residual capacity of this flow now accessible throug res_capacity
```

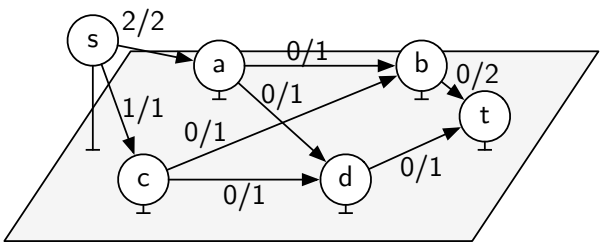
Using a different flow algorithm is very easy. Just replace the header and function call.

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

Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

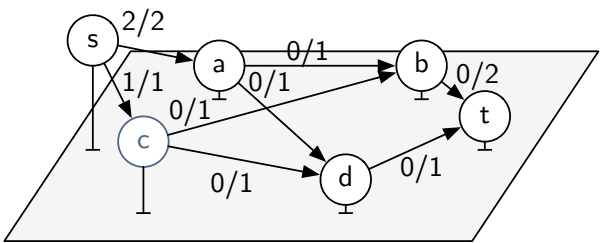
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

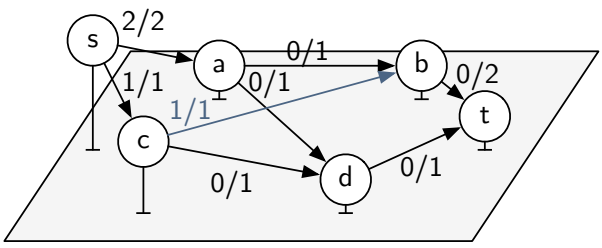
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

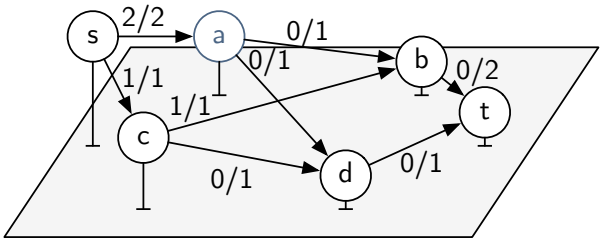
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

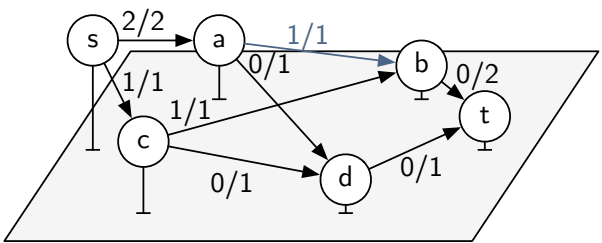
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

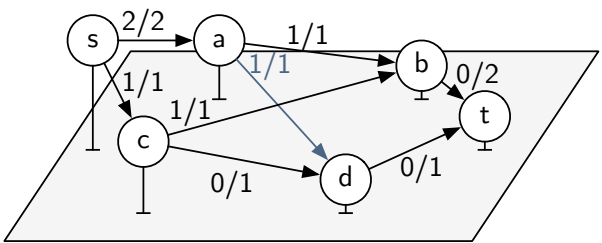
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

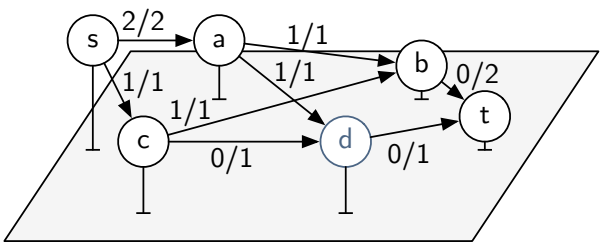
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

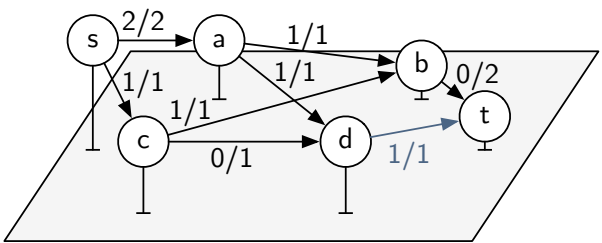
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

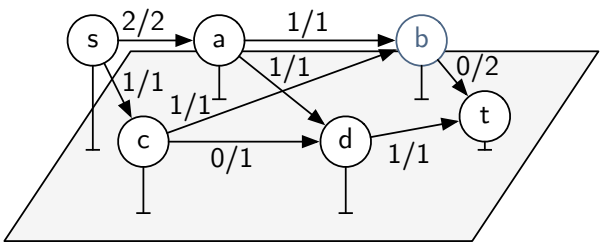
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



Network Flow Algorithms: Push-Relabel Max-Flow

Intuition: (not really needed for using it)

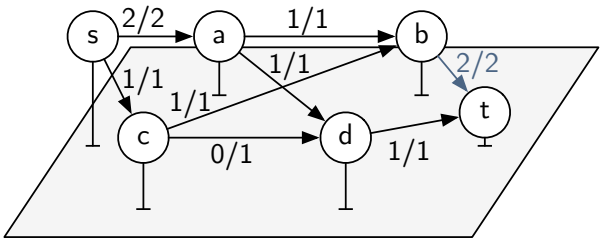
- Augment flow locally edge by edge instead of augmenting paths.
- Use height label to ensure that the flow is consistent and maximum in the end.
- Push step: increase flow along a downward out-edge of any overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible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



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 overflooded vertex.
- Relabel step: increase the height of a vertex so that a push is possible afterwards.



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.
- Goal difference used for tie breaking in the final standings.

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? 37 points still possible, so yes?
- Is this a flow problem?

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.
- Goal difference used for tie breaking in the final standings.

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? 37 points still possible, so yes?
- Is this a flow problem?

Tutorial Problem: Soccer Prediction

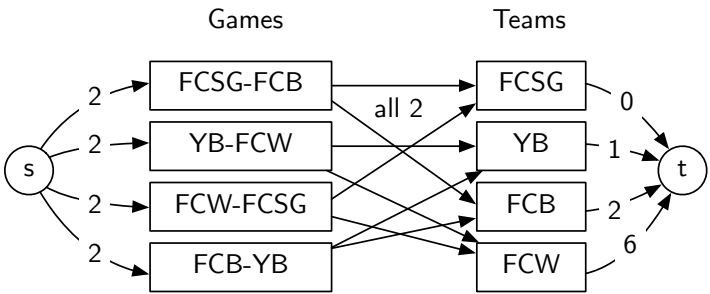
- Swiss Soccer Championship, two rounds before the end.
- 2 points awarded per game, split 1-1 if game ends in a tie.
- Goal difference used for tie breaking in the final standings.

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? 37 points still possible, so yes?
- Is this a flow problem?

Tutorial Problem: Modelling

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: Analysis

- Is this a flow problem? 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 $m = N + 3M$ edges.
  Flow is at most $2M$, edge capacity is at most $2M$.
- 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.

Tutorial Problem: Analysis

- Is this a flow problem? 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 $m = N + 3M$ edges.
  Flow is at most $2M$, edge capacity is at most $2M$.
- 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.

Tutorial Problem: Analysis

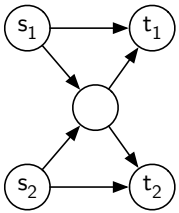
- Is this a flow problem? Yes.
- What does a unit of flow stand for? A point in the soccer ranking.
- How large is this flow graph?
  For N teams, M games, we have $n = 2 + N + M$ nodes and $m = N + 3M$ edges.
  Flow is at most $2M$, edge capacity is at most $2M$.
- What algorithm should we use?
  Push-Relabel 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.

Tutorial Problem: Analysis

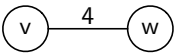
- Is this a flow problem? Yes.
- What does a unit of flow stand for? A point in the soccer ranking.
- How large is this flow graph?
  For N teams, M games, we have $n = 2 + N + M$ nodes and $m = N + 3M$ edges.
  Flow is at most $2M$, edge capacity is at most $2M$.
- What algorithm should we use?
  Push-Relabel 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.

Common tricks

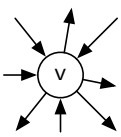
Multiple sources/sinks:



Undirected Graphs



Vertex Capacities

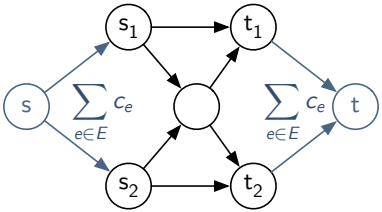


Minimum Flow per Edge

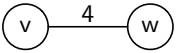
[Exercise]

Common tricks

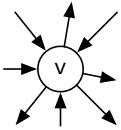
Multiple sources/sinks:



Undirected Graphs



Vertex Capacities

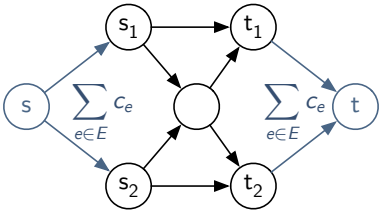


Minimum Flow per Edge

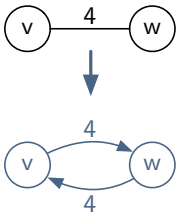
[Exercise]

Common tricks

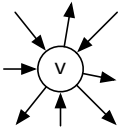
Multiple sources/sinks:



Undirected Graphs



Vertex Capacities

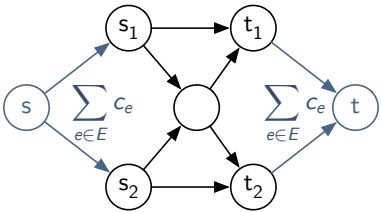


Minimum Flow per Edge

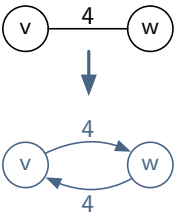
[Exercise]

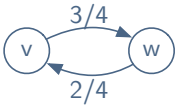
Common tricks

Multiple sources/sinks:

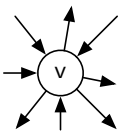


Undirected Graphs





Vertex Capacities

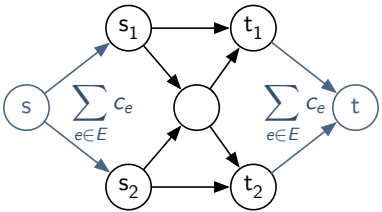


Minimum Flow per Edge

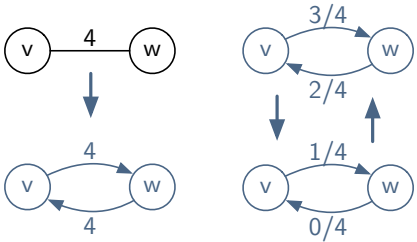
[Exercise]

Common tricks

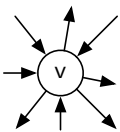
Multiple sources/sinks:



Undirected Graphs



Vertex Capacities

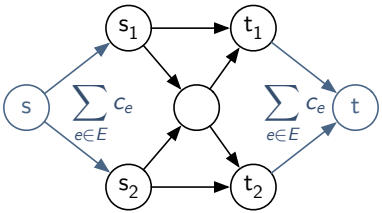


Minimum Flow per Edge

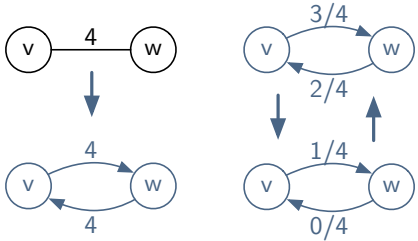
[Exercise]

Common tricks

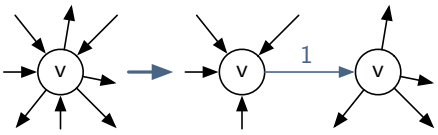
Multiple sources/sinks:



Undirected Graphs



Vertex Capacities

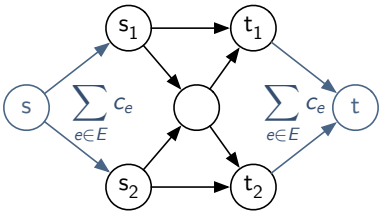


Minimum Flow per Edge

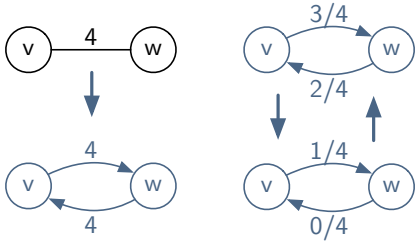
[Exercise]

Common tricks

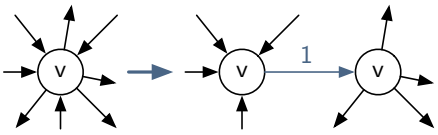
Multiple sources/sinks:



Undirected Graphs



Vertex Capacities

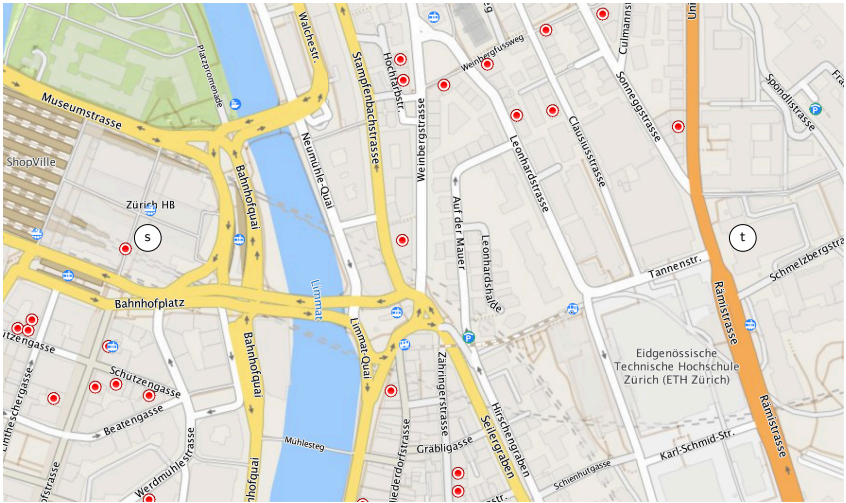


Minimum Flow per Edge

[Exercise]

Flow Application: Edge Disjoint Paths

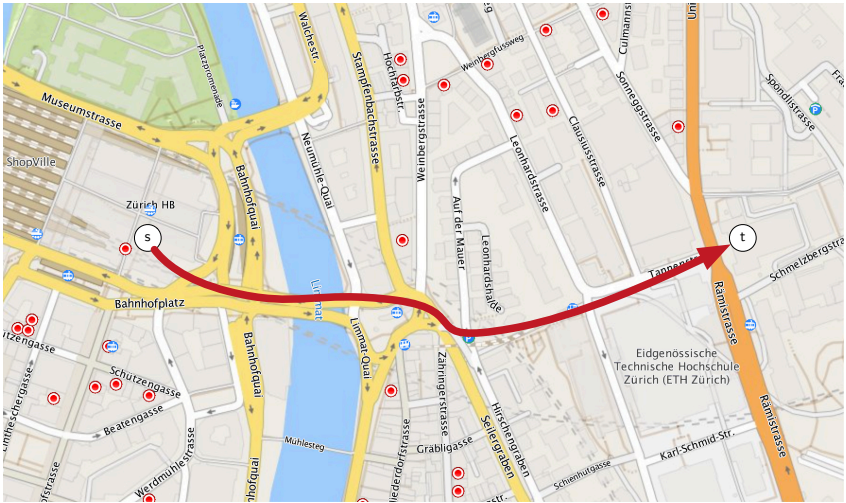
How many ways are there to get from HB to CAB without using the same street twice?



Map: search.ch, TomTom, swisstopo, OSM

Flow Application: Edge Disjoint Paths

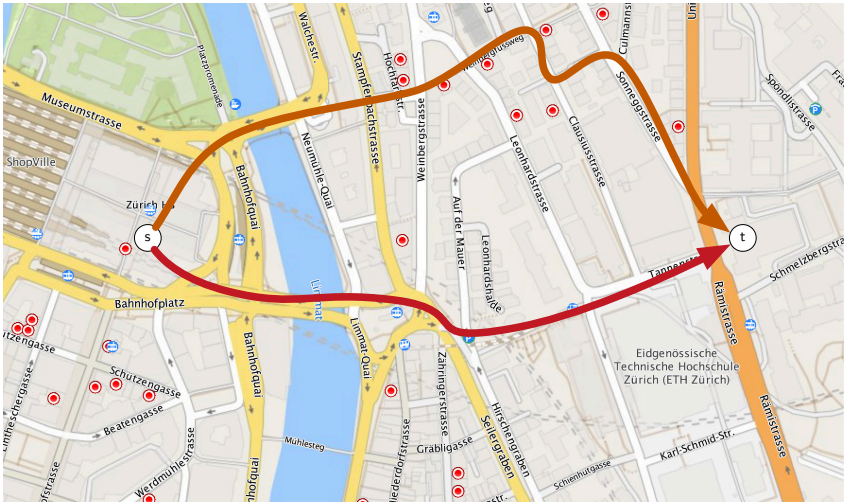
How many ways are there to get from HB to CAB without using the same street twice?



Map: search.ch, TomTom, swisstopo, OSM

Flow Application: Edge Disjoint Paths

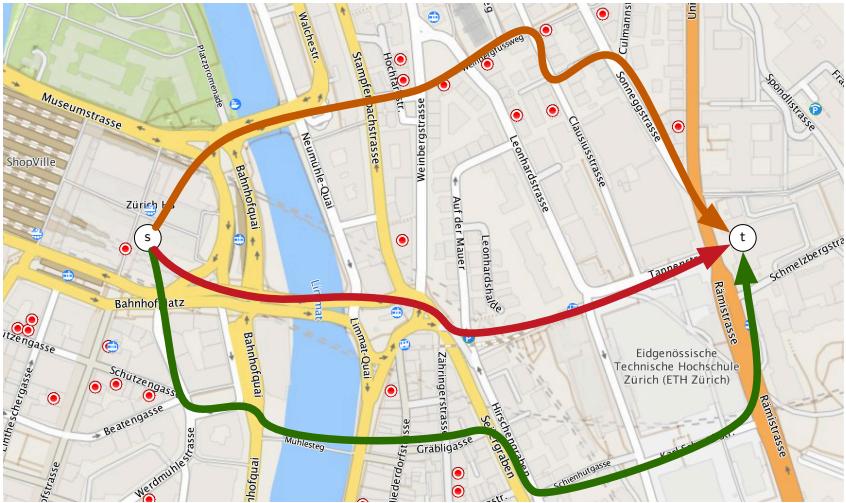
How many ways are there to get from HB to CAB without using the same street twice?



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?



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?

- Is this a flow problem? No.
- Can it be turned into a flow problem? Maybe.
- Build directed street graph by adding edges in both directions.
- Set all capacities to 1.

Lemma

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

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

- Is this a flow problem? No.
- Can it be turned into a flow problem? Maybe.
- Build directed street graph by adding edges in both directions.
- Set all capacities to 1.

Lemma

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

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

- Is this a flow problem? No.
- Can it be turned into a flow problem? Maybe.
- Build directed street graph by adding edges in both directions.
- Set all capacities to 1.

Lemma

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

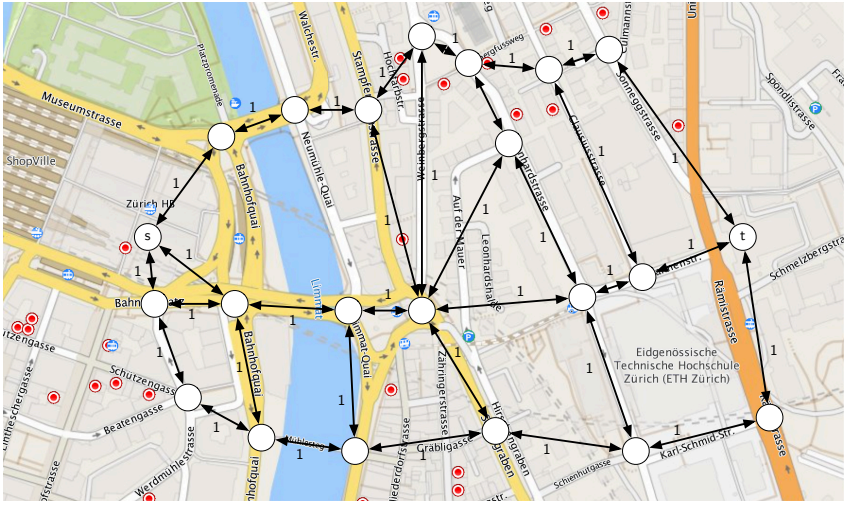
How many ways are there to get from HB to CAB without using the same street twice?

- Is this a flow problem? No.
- Can it be turned into a flow problem? Maybe.
- Build directed street graph by adding edges in both directions.
- Set all capacities to 1.

Lemma

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

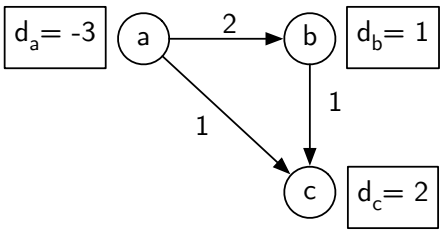
Flow Application: Circulation Problem

- 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?

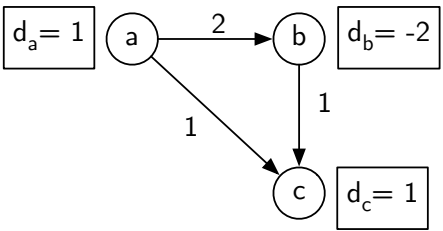
  Surely not if $\sum_{v \in V} d_v \neq 0$. Otherwise?

  Add super-source and super-sink to get a maximum flow problem.

feasible flow exists



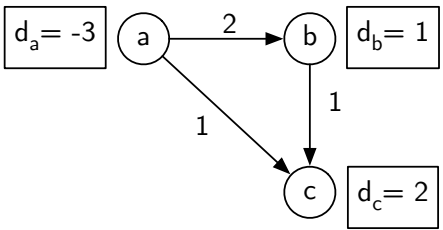
no feasible flow exists



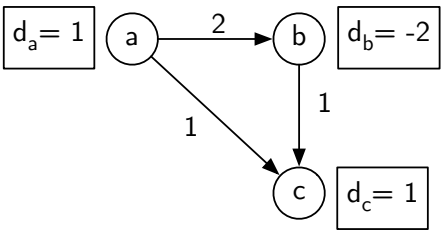
Flow Application: Circulation Problem

- 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



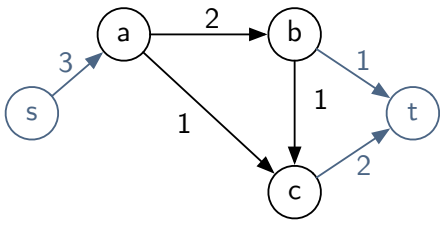
no feasible flow exists



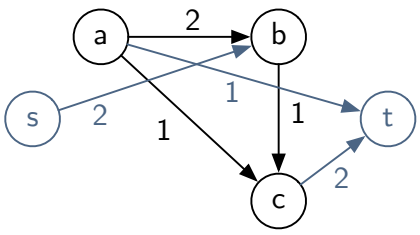
Flow Application: Circulation Problem

- 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



Problem Discussions

On the blackboard:

- Buddy Selection
- Tracking