Designing Networks for Selfish Users is Hard

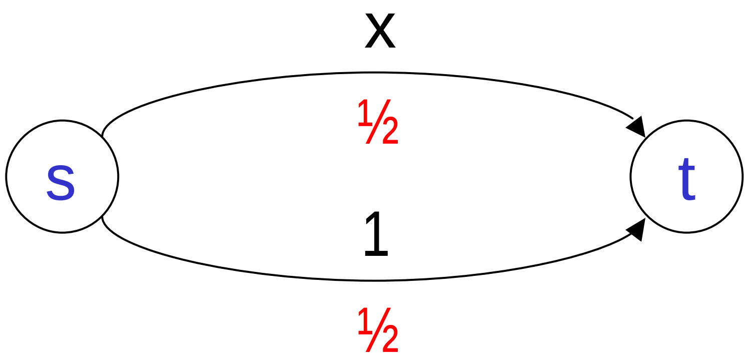
Tim Roughgarden
Cornell University

Traffic in Congested Networks

The Model:

- A directed graph $G = (V,E)$
- A source s and a sink t
- A rate r of traffic from s to t
- For each edge e, a latency function $l_e(\bullet)$

Example: $(r=1)$



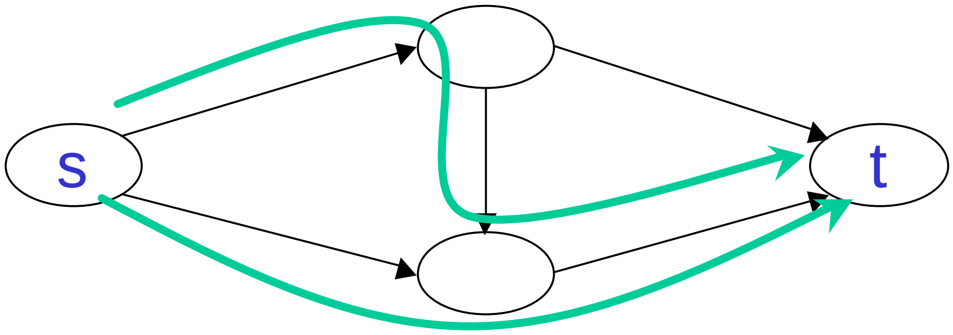
Traffic Flows

Traffic and Flows:

- f_P = amount of traffic routed on s-t path P
- flow vector f ⟺ routing of traffic

Path Latency:

- latency of path P w.r.t. flow f = sum of latencies of edges on P

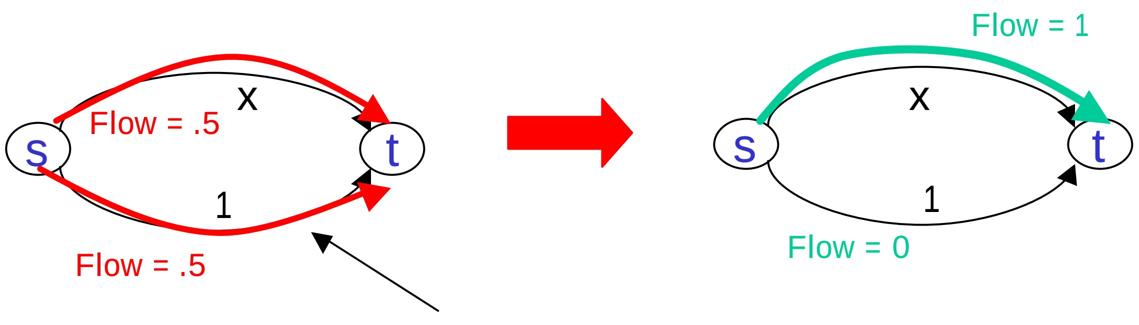


Flows as Selfish Traffic

- flow = routes of many noncooperative agents
- Examples:
  - cars in a highway system
  - packets in a network
- agents are selfish
  - want to minimize personal latency
  - will seek out path with minimum-possible latency

Flows at Nash Equilibrium

Def: A flow is at Nash equilibrium (is a Nash flow) if no agent can improve its latency by changing its path



this flow is envious!

Assumption: edge latency functions are continuous, nondecreasing

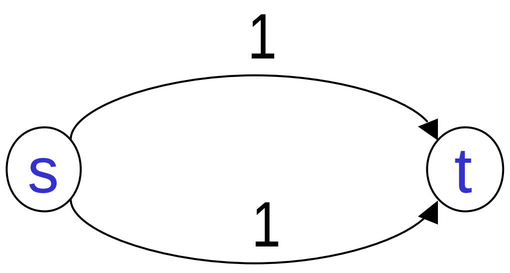
Lemma: f is a Nash flow if and only if all flow travels along minimum-latency paths (w.r.t. f)

Existence + Uniqueness

Assumption: edge latency functions are continuous, nondecreasing

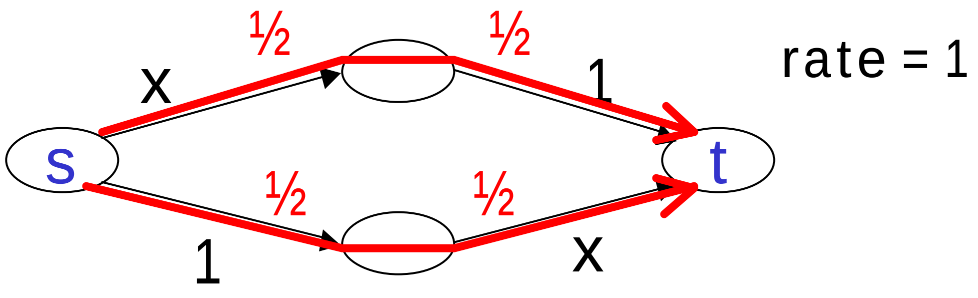
Fact: [Beckmann/McGuire/Winsten 56]

- Nash flows always exist
- Nash flows are (almost) unique
  - up to networks like:

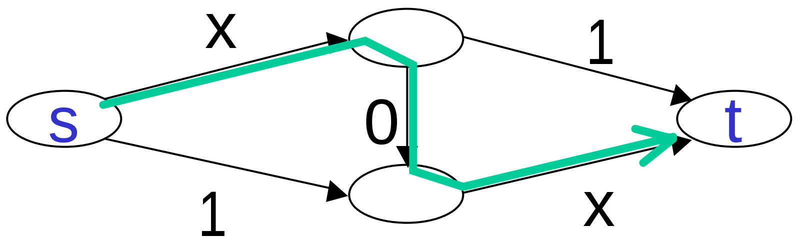


Braess's Paradox

Better network, worse Nash flow:



Cost of Nash flow = 1.5



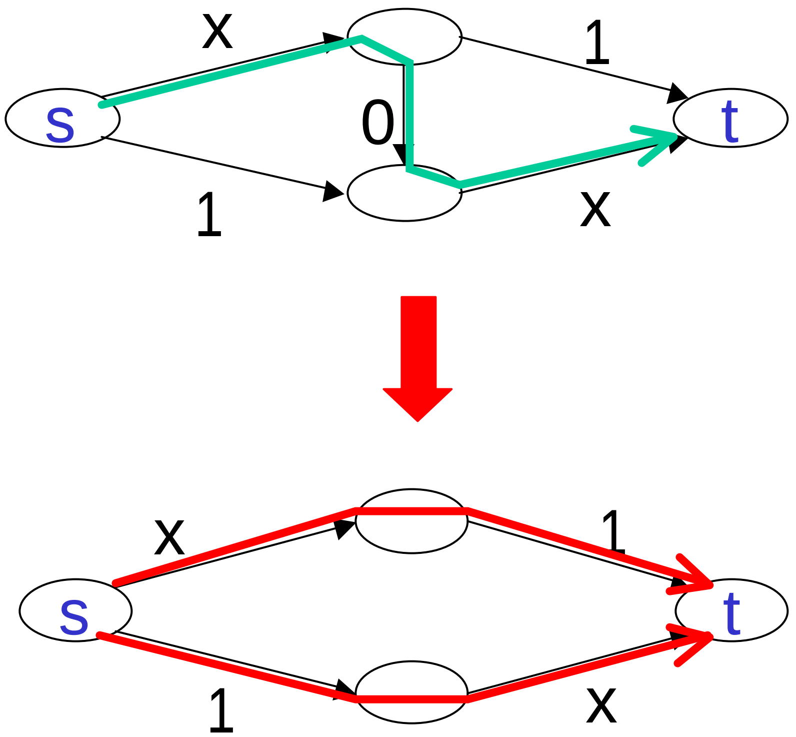
Cost of Nash flow = 2

All traffic experiences more latency!

- example from [Braess 68]

Deleting Arcs to Improve a Nash Flow

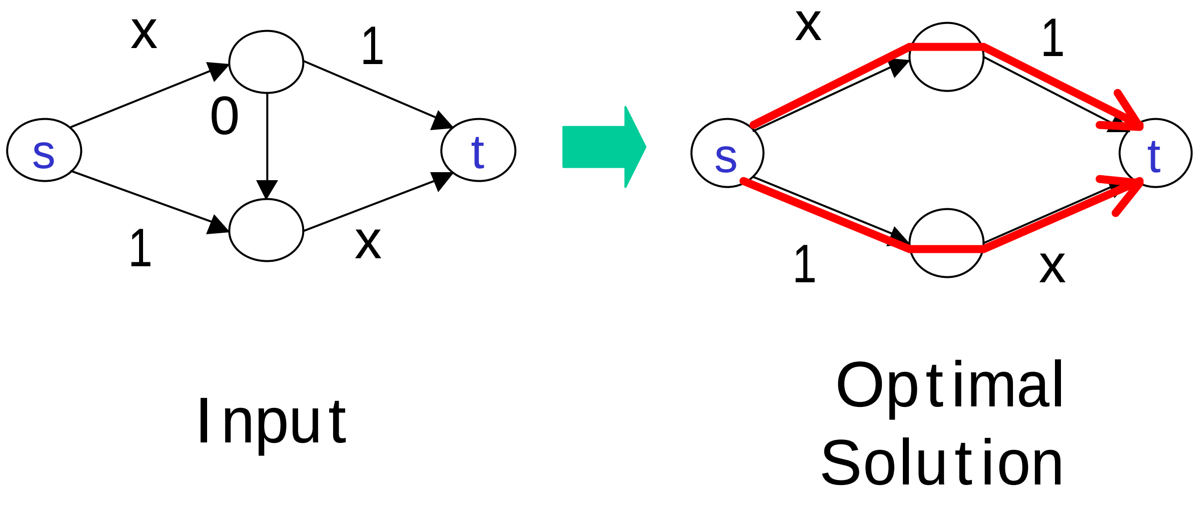
Motivating Question: how can we "fix up" networks with a bad Nash flow?



Designing Networks for Selfish Users

Formally:

- given network G = (V,E,l)
- find subnetwork minimizing latency experienced by all selfish users in a Nash flow



Previous Work

- [Braess 68], [Murchland 70]
  - network design problem defined
- [Steinberg/Zangwill 83], etc.
  - When is the trivial algorithm optimal?

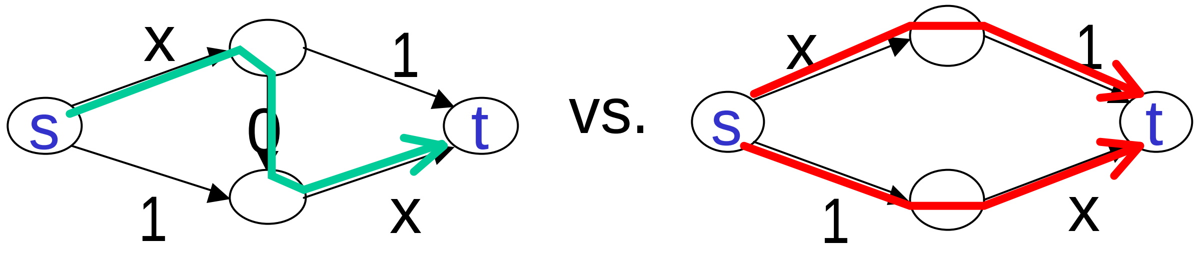
Def: The trivial algorithm is to build the entire network.

Guarantees for the Trivial Algorithm

Fact: The trivial algorithm is a $|V|/2$-approximation algorithm.

Def: a linear latency function is of the form $l_e(x)=a_ex+b_e$

Fact: For linear latency fns, the trivial algorithm is a $4/3$-approximation algorithm.



Designing Networks for Selfish Users is Hard

Thm 1: For ? > 0, no (|V|/2 - ?)-approximation algorithm exists (unless P=NP).

Thm 2: For linear latency functions, no (4/3 - ?)-approx algorithm exists (unless P=NP).

Corollary: in general, "bad edges" cannot be detected efficiently.

Linear Latency - Upper Bound

Thm: [Roughgarden/Tardos 2000] In a network with linear latency fns:

total latency of Nash flow = 4/3 × total latency of any other flow

Corollary:

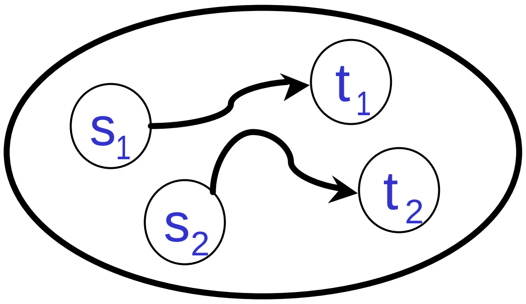
total latency of Nash flow = 4/3 × total latency of any flow at equilibrium in a subgraph

Corollary: the trivial algorithm has approximation ratio 4/3.

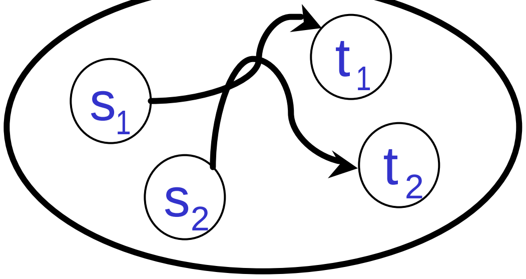
A Hard Problem

Problem 2DDP:

- Given:
  - directed graph G
  - terminals s_1, s_2, t_1, t_2
- Question:
  - are there vertex-disjoint s_1-t_1 and s_2-t_2 paths?



vs.



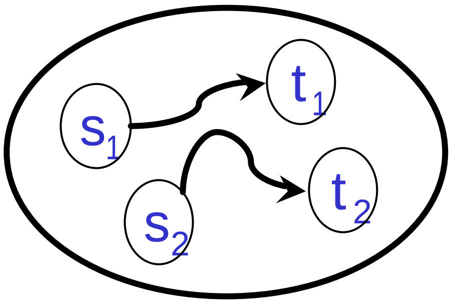
Fact: [Fortune/Hopcroft/Wyllie 80]

2DDP is NP-complete.

Linear Latency - Lower Bound

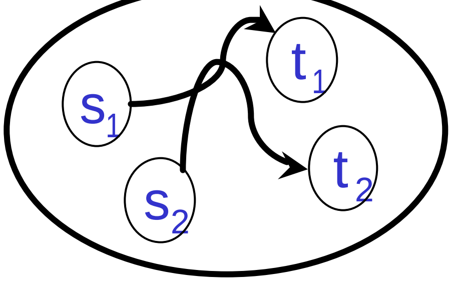
Goal: for instance G of 2DDP, produce network design instance G' so that:

G a "yes" instance



⇒ For some subgraph H of G', L(H) = 3/2

G a "no" instance

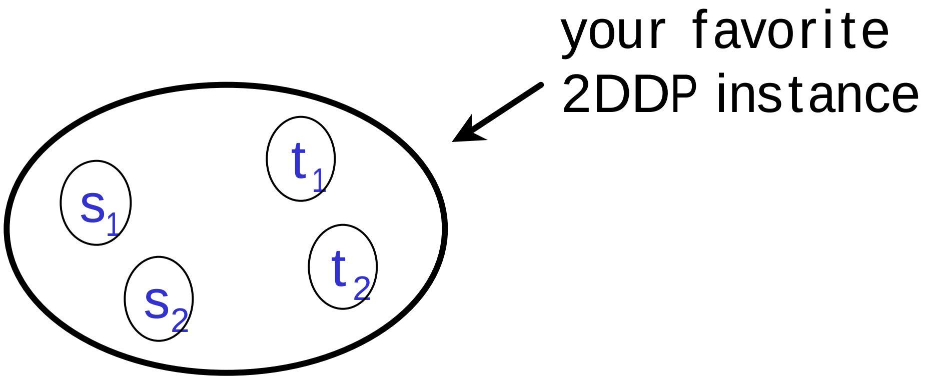


⇒ For every subgraph H of G', L(H) ≥ 2

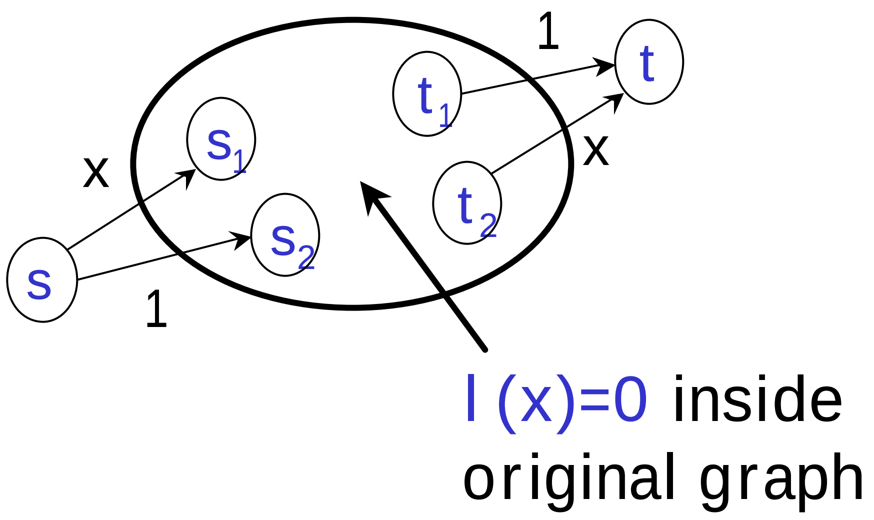
Common latency in a Nash flow in H

The Reduction

Given:



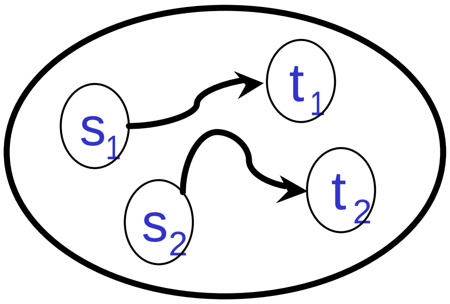
Construct:



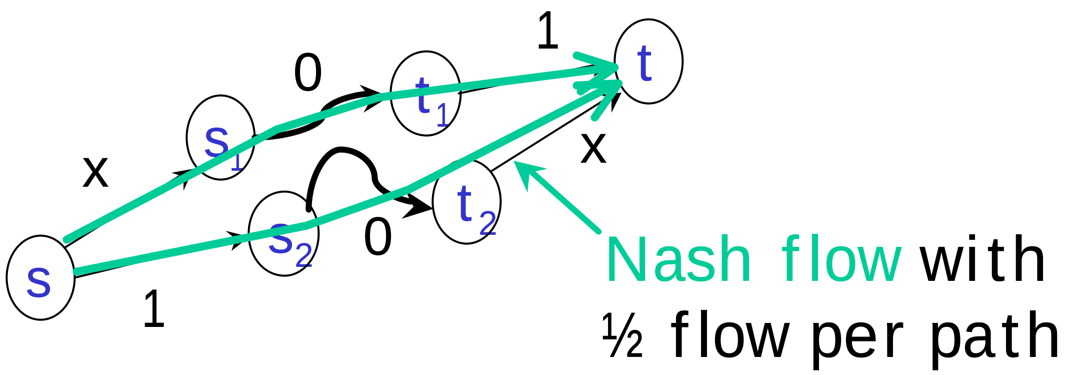
And, set traffic rate $r = 1$.

"Yes" instances of 2DDP

If 2DDP instance G has disjoint paths



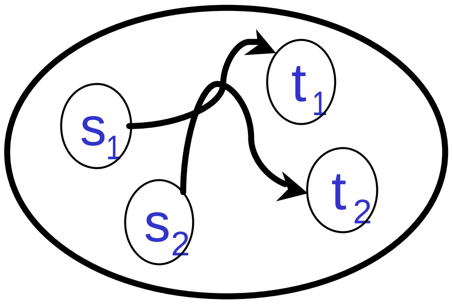
then, we can obtain H:



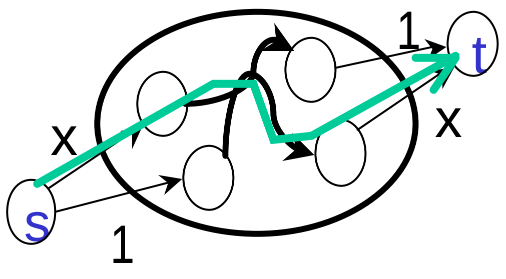
with L(H) = 3/2

"No" instances of 2DDP

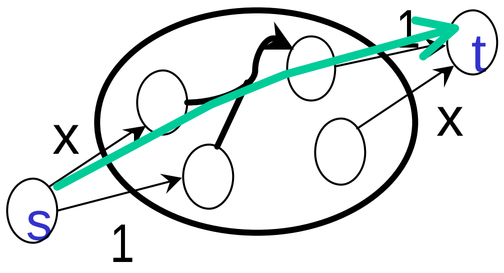
If 2DDP instance G has no disjoint paths



then, a subgraph H looks like:



or



with $L(H) = 2$

General Latency - An Easy Upper Bound?

Proof approach from linear case:

We hope: In a network with general latency fns:

total latency of Nash flow = ß × total latency of any other flow

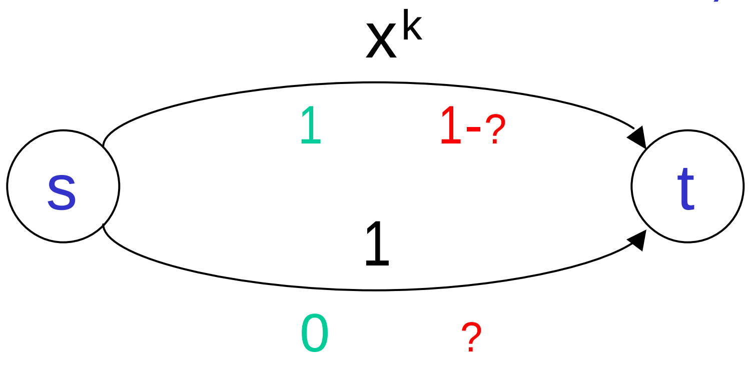
[perhaps with ß = ß(|V|,|E|)]

Then: the trivial algorithm has approximation ratio ß.

Difficulties

Problem: with general latency fns, a Nash flow can cost arbitrarily more other flows, even when $|V| = |E| = 2$:

$r = 1$, k large



Nash flow has total latency 1, but total latency ≈ 0 is possible

Conclusion: need a more refined approach for upper bound

Light Edges

Notation: (for a fixed input)

- f = Nash flow in original graph
  - L = common latency in f
- f^* = Nash flow in opt subgraph
  - L^* = common latency in f^*

Def: An edge e is light if $f^*_e \geq f_e$

- used more heavily by f^* than by f

Observation: if e is light,

$$l_e(f_e) = l_e(f^*_e) = L^*$$

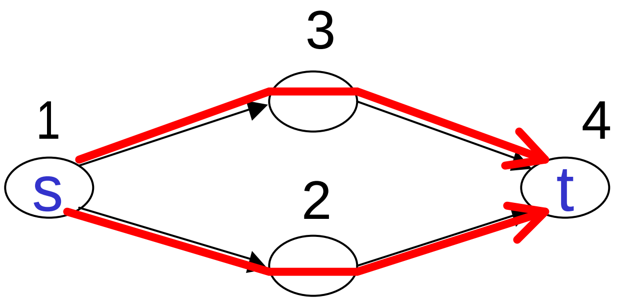
Consecutive Cuts

WLOG: our Nash flow f is acyclic

- can remove zero-latency flow cycles

Corollary: can topologically sort vertices of G w.r.t. f

- all flow arcs of f go forward



Def: ith consecutive cut = (S,V/S) where S = first i vertices in topological ordering

Light Edges Cross Consecutive Cuts

Observation: if $(S,V\setminus S)$ = some consecutive cut:

- amt of f-flow crossing cut = r
  - net flow across cut is r
  - no f-flow goes backwards
- amt of f^*-flow crossing cut $\geq r$
  - net flow across cut is r

Corollary: some edge crossing the cut is light

- used more heavily by f^* then by f

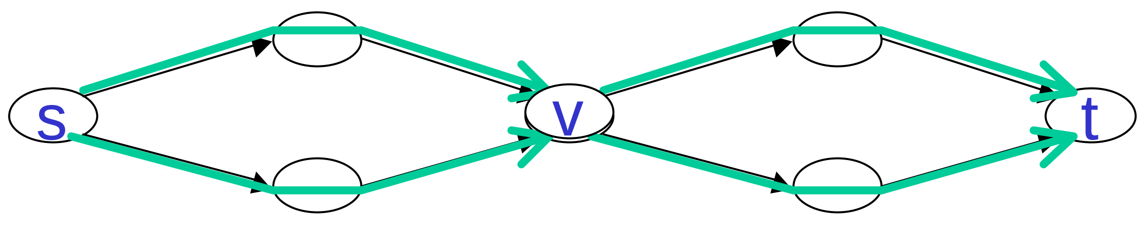
Distance Labels

Notation:

d(v) = common latency of all flow paths in f from s to v

Well-defined?

Yes:



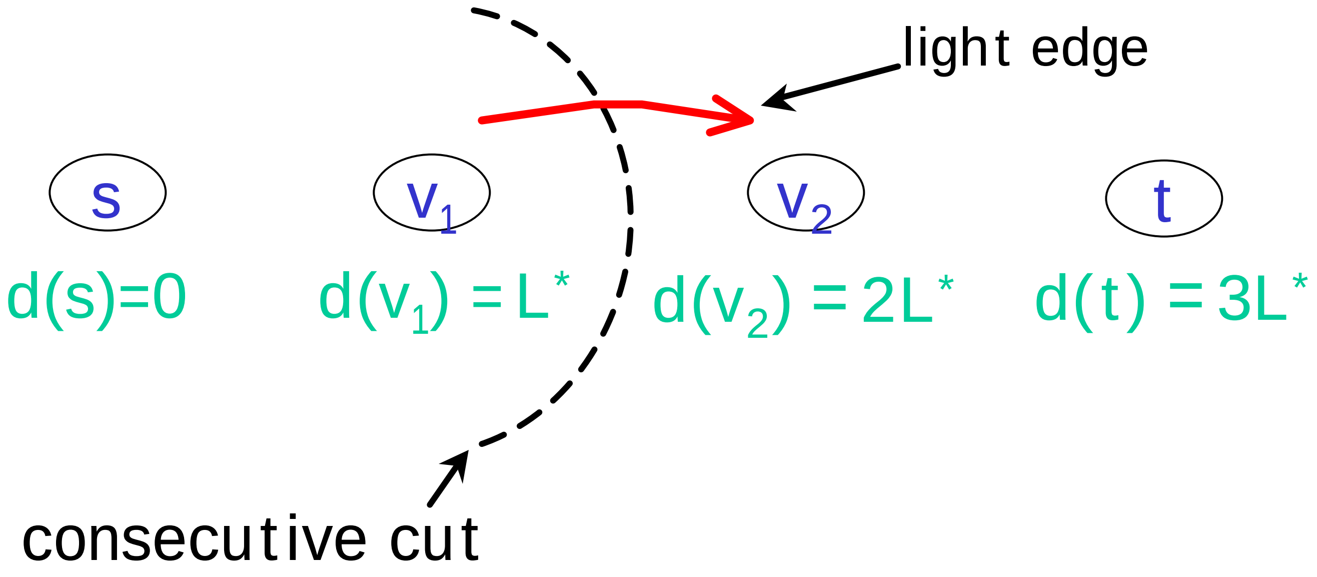
Note:

- d(s) = 0 and d(t) = L

Proof of Upper Bound

Step 1: sort vertices so that:

- all flow arcs go forward
- distance labels nondecreasing

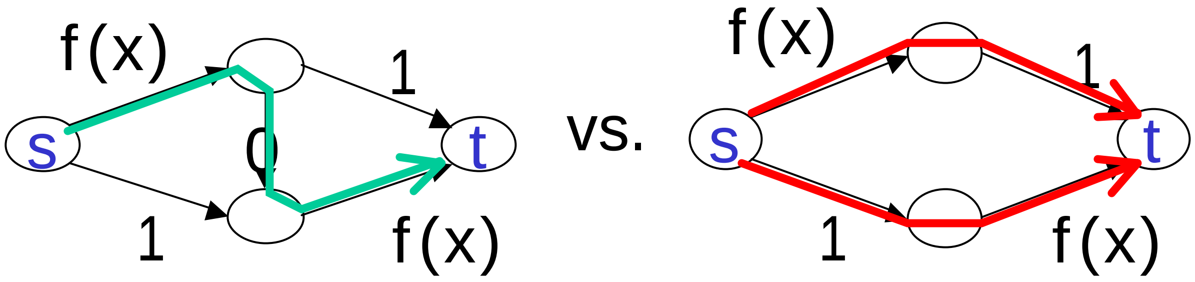


Step 2: by induction on i,

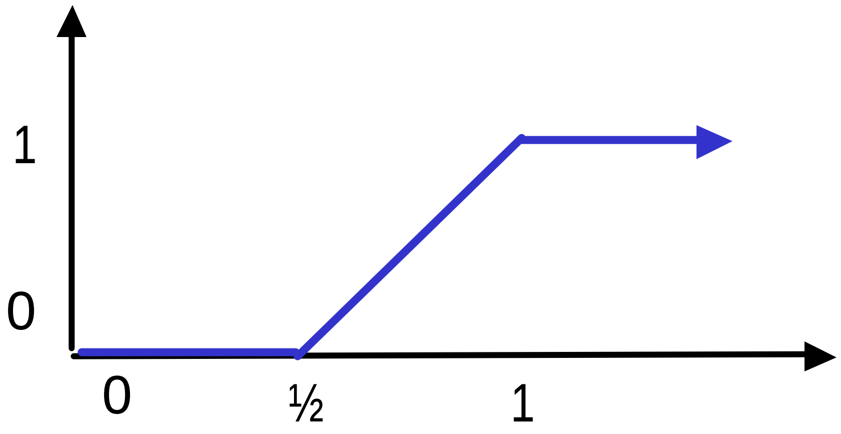
$$d(v_i) = i \times L^*$$

$$\Rightarrow L = d(t) = (|V|-1)L^*$$

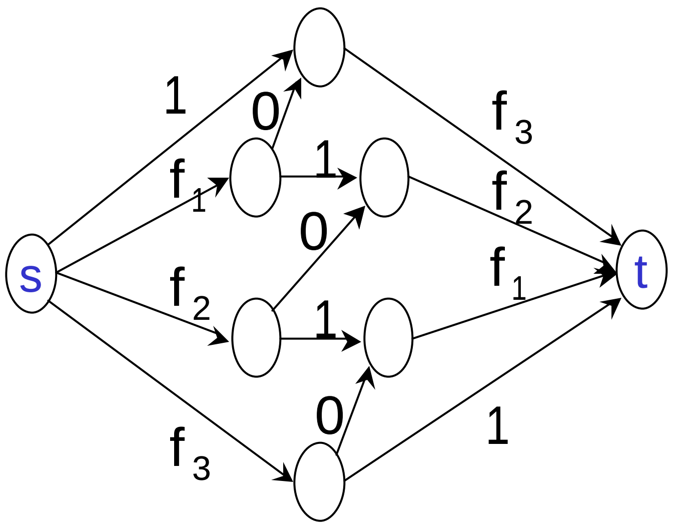
Lower Bound for the Trivial Algorithm



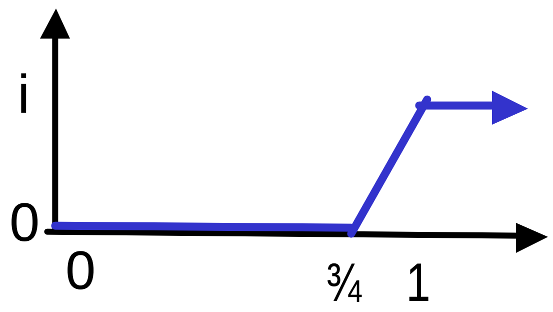
where f(x) is:



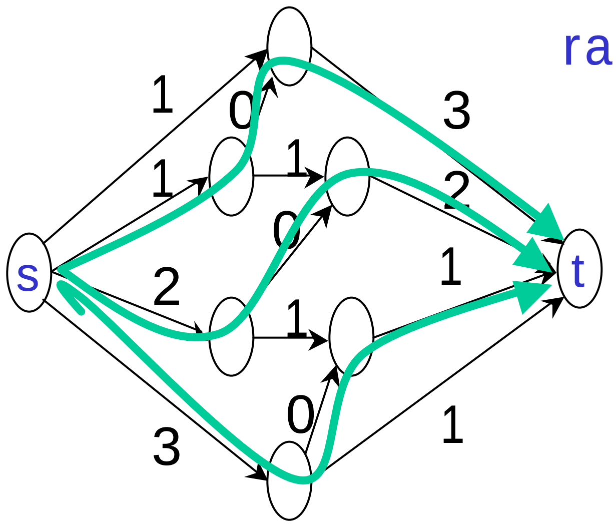
Lower Bound for the Trivial Algorithm



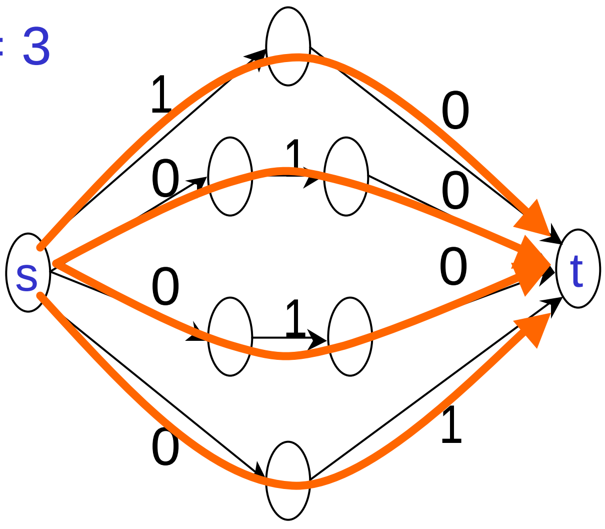
where $f_i(x)$ =



rate = 3



Nash in whole graph



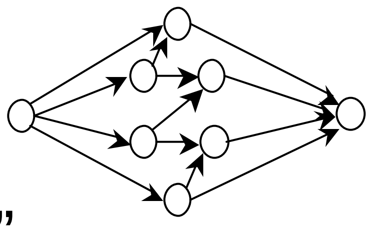
Nash in opt subgraph

Toward a Hardness Result

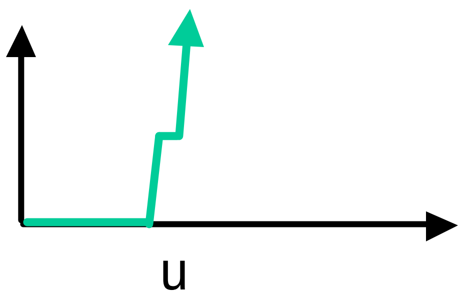
Thm: guarantee of |V|/2 is best possible, unless P=NP.

Notes on Proof:

- reduction from Partition
- construct networks like:



- replace each “cross-edge” with parallel edges representing Partition instance
- use latency functions to encode “capacities”:



Extensions

Remark: hardness of network design not particular to general, linear latency fns

E.g.: polynomials with degree = k:

- trivial algorithm achieves performance guarantee O(k/log k)
- hardness: O (k/log k)