Short Course on Parametric Network Flow

S. Thomas McCormick

Sauder School of Business, UBC
Döllnsee Summer School, June 2011

Some real rowers

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).
- It also has a non-negative capacity u_{ij} on each arc $i \longrightarrow j$.

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).
- It also has a non-negative capacity u_{ij} on each arc $i \longrightarrow j$.
- A feasible flow x on the arcs satisfies

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).
- It also has a non-negative capacity u_{ij} on each arc $i \to j$.
- A feasible flow x on the arcs satisfies
  - Boundedness: $0 \leq x_{ij} \leq u_{ij}$ for all $i \to j \in A$.

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).
- It also has a non-negative capacity u_{ij} on each arc $i \to j$.
- A feasible flow x on the arcs satisfies
  - Boundedness: $0 \leq x_{ij} \leq u_{ij}$ for all $i \to j \in A$.
  - Conservation: $\sum_i x_{ij} = \sum_k x_{jk}$ for all $j \neq s, t$.

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).
- It also has a non-negative capacity u_{ij} on each arc $i \to j$.
- A feasible flow x on the arcs satisfies
  - Boundedness: $0 \leq x_{ij} \leq u_{ij}$ for all $i \to j \in A$.
  - Conservation: $\sum_i x_{ij} = \sum_k x_{jk}$ for all $j \neq s, t$.
  - Alternative way to express conservation: Define the excess of node j to be $e_j := \sum_i x_{ij} - \sum_k x_{jk}$. Then conservation is equivalent to enforcing that $e_i = 0$ for all $j \neq s, t$.

Review and notation: Max Flow

- A max flow/min cut network has nodes N, directed arcs A, distinguished nodes s (source) and t (sink).
- It also has a non-negative capacity u_{ij} on each arc $i \to j$.
- A feasible flow x on the arcs satisfies
  - Boundedness: $0 \le x_{ij} \le u_{ij}$ for all $i \to j \in A$.
  - Conservation: $\sum_i x_{ij} = \sum_k x_{jk}$ for all $j \ne s, t$.
  - Alternative way to express conservation: Define the excess of node j to be $e_j := \sum_i x_{ij} - \sum_k x_{jk}$. Then conservation is equivalent to enforcing that $e_i = 0$ for all $j \ne s, t$.
- The objective is to maximize the (net flow out of s) $= \mathrm{val}^* := \max_{x \text{ feasible}} \mathrm{val}(x) = \sum_j x_{sj} - \sum_k x_{ks} = -e_s$. By conservation, $-e_s = +e_t =$ (net flow into t).

Review and notation: Min Cut

- A cut is a partition of nodes $N = S \cup T$ with $S \cap T = \emptyset$ and $s \in S$, $t \in T$; usually we refer to the cut just by S.

Review and notation: Min Cut

- A cut is a partition of nodes $N = S \cup T$ with $S \cap T = \emptyset$ and $s \in S$, $t \in T$; usually we refer to the cut just by S.
- The set of (forward) arcs induced by cut S is $\delta^+(S) := \{i \to j \in A \mid i \in S,\, j \in T\}$. We abuse notation and often refer to both S and $\delta^+(S)$ as a cut.

Review and notation: Min Cut

- A cut is a partition of nodes $N = S \cup T$ with $S \cap T = \emptyset$ and $s \in S$, $t \in T$; usually we refer to the cut just by S.
- The set of (forward) arcs induced by cut S is $\delta^+(S) := \{i \to j \in A \mid i \in S,\, j \in T\}$. We abuse notation and often refer to both S and $\delta^+(S)$ as a cut.
- The capacity of cut S is $\mathrm{cap}(S) := u(\delta^+(S)) := \sum_{i \to j \in \delta^+(S)} u_{ij}$.

Review and notation: Min Cut

- A cut is a partition of nodes $N = S \cup T$ with $S \cap T = \emptyset$ and $s \in S$, $t \in T$; usually we refer to the cut just by S.
- The set of (forward) arcs induced by cut S is $\delta^+(S) := \{i \to j \in A \mid i \in S,\, j \in T\}$. We abuse notation and often refer to both S and $\delta^+(S)$ as a cut.
- The capacity of cut S is $\mathrm{cap}(S) := u(\delta^+(S)) := \sum_{i \to j \in \delta^+(S)} u_{ij}$.
- The objective of Min Cut is $\mathrm{cap}^* := \min_{\text{cuts } S} \mathrm{cap}(S)$.

Review and notation: Min Cut

- A cut is a partition of nodes $N = S \cup T$ with $S \cap T = \emptyset$ and $s \in S$, $t \in T$; usually we refer to the cut just by S.
- The set of (forward) arcs induced by cut S is $\delta^+(S) := \{i \to j \in A \mid i \in S,\, j \in T\}$. We abuse notation and often refer to both S and $\delta^+(S)$ as a cut.
- The capacity of cut S is $\mathrm{cap}(S) := u(\delta^+(S)) := \sum_{i \to j \in \delta^+(S)} u_{ij}$.
- The objective of Min Cut is $\mathrm{cap}^* := \min_{\text{cuts } S} \mathrm{cap}(S)$.

Theorem (F & F)

For any capacities u, $\mathrm{val}^* = \max_x \mathrm{val}(x) = \min_S \mathrm{cap}(S) = \mathrm{cap}^*$, *i.e., the value of a max flow equals the capacity of a min cut.*

Parametric max flow/min cut

- Suppose now that the capacities are functions of a scalar parameter λ instead of being constant, i.e., the capacity of $i \longrightarrow j$ is now $u_{ij}(\lambda)$ instead of being constant.

Parametric max flow/min cut

- Suppose now that the capacities are functions of a scalar parameter λ instead of being constant, i.e., the capacity of $i \to j$ is now $u_{ij}(\lambda)$ instead of being constant.
- For now we will assume that each $u_{ij}(\lambda)$ is an affine function of λ, i.e., $u_{ij}(\lambda) = a_{ij} + b_{ij}\lambda$ for some data a_{ij}, b_{ij}.

Parametric max flow/min cut

- Suppose now that the capacities are functions of a scalar parameter λ instead of being constant, i.e., the capacity of $i \to j$ is now $u_{ij}(\lambda)$ instead of being constant.
- For now we will assume that each $u_{ij}(\lambda)$ is an affine function of λ, i.e., $u_{ij}(\lambda) = a_{ij} + b_{ij}\lambda$ for some data a_{ij}, b_{ij}.
- Thus $\mathrm{val}(x)$, val^*, $\mathrm{cap}(S)$, and cap^* are functions of λ, i.e., $\mathrm{val}(x, \lambda)$, $\mathrm{val}^*(\lambda)$, $\mathrm{cap}(S, \lambda)$, and $\mathrm{cap}^*(\lambda)$.

Parametric max flow/min cut

- Suppose now that the capacities are functions of a scalar parameter λ instead of being constant, i.e., the capacity of $i \to j$ is now $u_{ij}(\lambda)$ instead of being constant.
- For now we will assume that each $u_{ij}(\lambda)$ is an affine function of λ, i.e., $u_{ij}(\lambda) = a_{ij} + b_{ij}\lambda$ for some data a_{ij}, b_{ij}.
- Thus $\mathrm{val}(x)$, val^*, $\mathrm{cap}(S)$, and cap^* are functions of λ, i.e., $\mathrm{val}(x, \lambda)$, $\mathrm{val}^*(\lambda)$, $\mathrm{cap}(S, \lambda)$, and $\mathrm{cap}^*(\lambda)$.
- Ideally we want to solve the Parametric Flow Problem:

(PFP): Find $\mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda)$ for *all* values of λ.

Parametric max flow/min cut

- Suppose now that the capacities are functions of a scalar parameter λ instead of being constant, i.e., the capacity of $i \to j$ is now $u_{ij}(\lambda)$ instead of being constant.
- For now we will assume that each $u_{ij}(\lambda)$ is an affine function of λ, i.e., $u_{ij}(\lambda) = a_{ij} + b_{ij}\lambda$ for some data a_{ij}, b_{ij}.
- Thus $\mathrm{val}(x)$, val^*, $\mathrm{cap}(S)$, and cap^* are functions of λ, i.e., $\mathrm{val}(x, \lambda)$, $\mathrm{val}^*(\lambda)$, $\mathrm{cap}(S, \lambda)$, and $\mathrm{cap}^*(\lambda)$.
- Ideally we want to solve the Parametric Flow Problem:

  (PFP): Find $\mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda)$ for *all* values of λ.

- Since Max Flow and Min Cut are dual LPs, PFP is just a parametrized LP; either Max Flow with parametrized RHS, or Min Cut with parametrized objective vector (original vector a, parametric RHS/obj b).

Parametric max flow/min cut

- Suppose now that the capacities are functions of a scalar parameter λ instead of being constant, i.e., the capacity of $i \to j$ is now $u_{ij}(\lambda)$ instead of being constant.
- For now we will assume that each $u_{ij}(\lambda)$ is an affine function of λ, i.e., $u_{ij}(\lambda) = a_{ij} + b_{ij}\lambda$ for some data a_{ij}, b_{ij}.
- Thus $\mathrm{val}(x)$, val^*, $\mathrm{cap}(S)$, and cap^* are functions of λ, i.e., $\mathrm{val}(x, \lambda)$, $\mathrm{val}^*(\lambda)$, $\mathrm{cap}(S, \lambda)$, and $\mathrm{cap}^*(\lambda)$.
- Ideally we want to solve the Parametric Flow Problem:

$$\text{(PFP): Find } \mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda) \text{ for } \textit{all} \text{ values of } \lambda.$$

- Since Max Flow and Min Cut are dual LPs, PFP is just a parametrized LP; either Max Flow with parametrized RHS, or Min Cut with parametrized objective vector (original vector a, parametric RHS/obj b).
- Thus by parametric LP theory we know that $\mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda)$ is a piecewise linear concave function of λ.

Why solve PFP?

1. It's a cool problem to consider: since Max Flow/Min Cut is easier than LP, is parametric Max Flow/Min cut easier than parametric LP?

Why solve PFP?

1. It's a cool problem to consider: since Max Flow/Min Cut is easier than LP, is parametric Max Flow/Min cut easier than parametric LP?
2. It has a surprisingly large number of applications. See the bibliography and problem set to get some idea of them: scheduling, baseball elimination, ratio objectives, etc, etc ...

Why solve PFP?

1. It's a cool problem to consider: since Max Flow/Min Cut is easier than LP, is parametric Max Flow/Min cut easier than parametric LP?
2. It has a surprisingly large number of applications. See the bibliography and problem set to get some idea of them: scheduling, baseball elimination, ratio objectives, etc, etc ...
3. If we are willing to consider some special structures, then we can find surprisingly fast algorithms that exactly solve PFP for all λ in a very clever way.

Why solve PFP?

1. It's a cool problem to consider: since Max Flow/Min Cut is easier than LP, is parametric Max Flow/Min cut easier than parametric LP?
2. It has a surprisingly large number of applications. See the bibliography and problem set to get some idea of them: scheduling, baseball elimination, ratio objectives, etc, etc ...
3. If we are willing to consider some special structures, then we can find surprisingly fast algorithms that exactly solve PFP for all λ in a very clever way.
4. Solving PFP gives us some insight into general parametric submodular optimization (see problems), and into the internals of the Push-Relabel Max Flow algorithm.

Why solve PFP?

1. It's a cool problem to consider: since Max Flow/Min Cut is easier than LP, is parametric Max Flow/Min cut easier than parametric LP?
2. It has a surprisingly large number of applications. See the bibliography and problem set to get some idea of them: scheduling, baseball elimination, ratio objectives, etc, etc ...
3. If we are willing to consider some special structures, then we can find surprisingly fast algorithms that exactly solve PFP for all λ in a very clever way.
4. Solving PFP gives us some insight into general parametric submodular optimization (see problems), and into the internals of the Push-Relabel Max Flow algorithm.
5. (Often in applications λ is fractional, so in general we'll have fractional flows here.)

Bad news and good news

- Bad news: As is the case with parametric LP, Carstensen showed that PFP in general can have an exponential number of pieces in the function $\mathrm{val}^*(\lambda)$ $\Longrightarrow$ no hope to find $\mathrm{val}^*(\lambda)$ in polynomial time.

Bad news and good news

- Bad news: As is the case with parametric LP, Carstensen showed that PFP in general can have an exponential number of pieces in the function $\mathrm{val}^*(\lambda)$ $\Longrightarrow$ no hope to find $\mathrm{val}^*(\lambda)$ in polynomial time.
- Good news: If we assume some special structure, then we get two great properties:

Bad news and good news

- Bad news: As is the case with parametric LP, Carstensen showed that PFP in general can have an exponential number of pieces in the function $\mathrm{val}^*(\lambda)$ $\implies$ no hope to find $\mathrm{val}^*(\lambda)$ in polynomial time.
- Good news: If we assume some special structure, then we get two great properties:
  1. Structural Property: Each piece of $\mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda)$ has an associated min cut optimal for that piece. If S_1 and S_2 are min cuts for consecutive pieces, then we'll show that $S_1 \subset S_2$, i.e., that these min cuts are nested. Thus there are at most $n-1$ such min cuts $\implies$ a polynomial algorithm is possible.

Bad news and good news

- Bad news: As is the case with parametric LP, Carstensen showed that PFP in general can have an exponential number of pieces in the function $\mathrm{val}^*(\lambda)$ $\implies$ no hope to find $\mathrm{val}^*(\lambda)$ in polynomial time.
- Good news: If we assume some special structure, then we get two great properties:
  1. Structural Property: Each piece of $\mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda)$ has an associated min cut optimal for that piece. If S_1 and S_2 are min cuts for consecutive pieces, then we'll show that $S_1 \subset S_2$, i.e., that these min cuts are nested. Thus there are at most $n-1$ such min cuts $\implies$ a polynomial algorithm is possible.
  2. Algorithmic Property: With some clever tricks we can adapt Push-Relabel such that it can compute all $O(n)$ pieces in the same asymptotic times as a single max flow (!!!). Thus we can solve PFP using only $O(1)$ max flows.

Bad news and good news

- Bad news: As is the case with parametric LP, Carstensen showed that PFP in general can have an exponential number of pieces in the function $\mathrm{val}^*(\lambda)$ $\implies$ no hope to find $\mathrm{val}^*(\lambda)$ in polynomial time.
- Good news: If we assume some special structure, then we get two great properties:
  1. Structural Property: Each piece of $\mathrm{val}^*(\lambda) = \mathrm{cap}^*(\lambda)$ has an associated min cut optimal for that piece. If S_1 and S_2 are min cuts for consecutive pieces, then we'll show that $S_1 \subset S_2$, i.e., that these min cuts are nested. Thus there are at most $n-1$ such min cuts $\implies$ a polynomial algorithm is possible.
  2. Algorithmic Property: With some clever tricks we can adapt Push-Relabel such that it can compute all $O(n)$ pieces in the same asymptotic times as a single max flow (!!!). Thus we can solve PFP using only $O(1)$ max flows.
- Furthermore, the "special structure" is not too special: it already includes most of the important applications of PFP.

The GGT special structure

- Gallo, Grigoriadis, and Tarjan (GGT) noticed the following special structure:

  For $s \to j$ arcs, $u_{sj}(\lambda)$ is non-decreasing in λ, i.e., $b_{sj} \geq 0$;

  For $i \to j$ arcs with $i \neq s$, $j \neq t$, $u_{ij}(\lambda)$ is constant, i.e., $b_{ij} = 0$;

  For $i \to t$ arcs, $u_{it}(\lambda)$ is non-increasing in λ, i.e, $b_{it} \leq 0$.

The GGT special structure

- Gallo, Grigoriadis, and Tarjan (GGT) noticed the following special structure:

  For $s \to j$ arcs, $u_{sj}(\lambda)$ is non-decreasing in λ, i.e., $b_{sj} \geq 0$;
  For $i \to j$ arcs with $i \neq s$, $j \neq t$, $u_{ij}(\lambda)$ is constant, i.e., $b_{ij} = 0$;
  For $i \to t$ arcs, $u_{it}(\lambda)$ is non-increasing in λ, i.e, $b_{it} \leq 0$.

- Intuitively, as λ increases, it gets harder to saturate arcs with tail s, and easier to saturate arcs with head t, and so min cuts get bigger.

The GGT special structure

- Gallo, Grigoriadis, and Tarjan (GGT) noticed the following special structure:

  For $s \to j$ arcs, $u_{sj}(\lambda)$ is non-decreasing in λ, i.e., $b_{sj} \geq 0$;
  For $i \to j$ arcs with $i \neq s$, $j \neq t$, $u_{ij}(\lambda)$ is constant, i.e., $b_{ij} = 0$;
  For $i \to t$ arcs, $u_{it}(\lambda)$ is non-increasing in λ, i.e, $b_{it} \leq 0$.

- Intuitively, as λ increases, it gets harder to saturate arcs with tail s, and easier to saturate arcs with head t, and so min cuts get bigger.
- We call parametric max flow networks with this property GGT parametric flow networks, or just GGT networks.

The GGT special structure

- Gallo, Grigoriadis, and Tarjan (GGT) noticed the following special structure:

  For $s \to j$ arcs, $u_{sj}(\lambda)$ is non-decreasing in λ, i.e., $b_{sj} \geq 0$;
  For $i \to j$ arcs with $i \neq s$, $j \neq t$, $u_{ij}(\lambda)$ is constant, i.e., $b_{ij} = 0$;
  For $i \to t$ arcs, $u_{it}(\lambda)$ is non-increasing in λ, i.e, $b_{it} \leq 0$.

- Intuitively, as λ increases, it gets harder to saturate arcs with tail s, and easier to saturate arcs with head t, and so min cuts get bigger.
- We call parametric max flow networks with this property GGT parametric flow networks, or just GGT networks.
- Our goal now is to prove the Structural and Algorithmic Properties for GGT networks.

The GGT special structure

- Gallo, Grigoriadis, and Tarjan (GGT) noticed the following special structure:

  For $s \to j$ arcs, $u_{sj}(\lambda)$ is non-decreasing in λ, i.e., $b_{sj} \geq 0$;
  For $i \to j$ arcs with $i \neq s$, $j \neq t$, $u_{ij}(\lambda)$ is constant, i.e., $b_{ij} = 0$;
  For $i \to t$ arcs, $u_{it}(\lambda)$ is non-increasing in λ, i.e, $b_{it} \leq 0$.

- Intuitively, as λ increases, it gets harder to saturate arcs with tail s, and easier to saturate arcs with head t, and so min cuts get bigger.
- We call parametric max flow networks with this property GGT parametric flow networks, or just GGT networks.
- Our goal now is to prove the Structural and Algorithmic Properties for GGT networks.
  - We'll get the Structural Property as a corollary of our algorithm so let's focus first on algorithms.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.
- Even so, it is not very clear how to compute $\mathrm{cap}^*(\lambda)$ and its min cuts.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.
- Even so, it is not very clear how to compute $\mathrm{cap}^*(\lambda)$ and its min cuts.
- Simple idea:

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.
- Even so, it is not very clear how to compute $\mathrm{cap}^*(\lambda)$ and its min cuts.
- Simple idea:
  - Choose some initial λ_0 and find a max flow/min cut as usual.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.
- Even so, it is not very clear how to compute $\mathrm{cap}^*(\lambda)$ and its min cuts.
- Simple idea:
  - Choose some initial λ_0 and find a max flow/min cut as usual.
  - Then choose some new value, say $\lambda_1 > \lambda_0$, and see if λ_1's min cut is different from λ_0's min cut.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.
- Even so, it is not very clear how to compute $\mathrm{cap}^*(\lambda)$ and its min cuts.
- Simple idea:
  - Choose some initial λ_0 and find a max flow/min cut as usual.
  - Then choose some new value, say $\lambda_1 > \lambda_0$, and see if λ_1's min cut is different from λ_0's min cut.
  - If so, then there is a breakpoint between λ_0 and λ_1, and we can try some search procedure to find it.

A building block for a PFP algorithm for GGT networks

- To solve PFP we need to:
  - Find all $O(n)$ breakpoints of $\mathrm{cap}^*(\lambda)$;
  - Find all $O(n)$ optimal min cuts for the $O(n)$ pieces of $\mathrm{cap}^*(\lambda)$;
  - Find some way to represent the optimal flows for all values of λ.
- It turns out that most of the applications only want the min cuts, not the max flows, so let's ignore the max flows for now.
- Even so, it is not very clear how to compute $\mathrm{cap}^*(\lambda)$ and its min cuts.
- Simple idea:
  - Choose some initial λ_0 and find a max flow/min cut as usual.
  - Then choose some new value, say $\lambda_1 > \lambda_0$, and see if λ_1's min cut is different from λ_0's min cut.
  - If so, then there is a breakpoint between λ_0 and λ_1, and we can try some search procedure to find it.
  - If not, then try an even larger λ_2 . . . ?!?

The basic subproblem for GGT PFP

- Consider this: given an optimal x^0 for λ_0, and given some $\lambda_1 > \lambda_0$, try to find an optimal x^1 for λ_1 by starting at x^0 instead of starting from scratch.

The basic subproblem for GGT PFP

- Consider this: given an optimal x^0 for λ_0, and given some $\lambda_1 > \lambda_0$, try to find an optimal x^1 for λ_1 by starting at x^0 instead of starting from scratch.
  - Note that λ_1 might be given online, i.e., its value will depend in some way on x^0 (or its min cut).

The basic subproblem for GGT PFP

- Consider this: given an optimal x^0 for λ_0, and given some $\lambda_1 > \lambda_0$, try to find an optimal x^1 for λ_1 by starting at x^0 instead of starting from scratch.
  - Note that λ_1 might be given online, i.e., its value will depend in some way on x^0 (or its min cut).
  - Our real aim is to be able to do this repeatedly for a whole sequence of λ's, $\lambda_0 < \lambda_1 < \cdots < \lambda_k$ with $k = \Theta(n)$ using only $O(1)$ max flow total work instead of $\Theta(n)$ max flow total work.

The basic subproblem for GGT PFP

- Consider this: given an optimal x^0 for λ_0, and given some $\lambda_1 > \lambda_0$, try to find an optimal x^1 for λ_1 by starting at x^0 instead of starting from scratch.
  - Note that λ_1 might be given online, i.e., its value will depend in some way on x^0 (or its min cut).
  - Our real aim is to be able to do this repeatedly for a whole sequence of λ's, $\lambda_0 < \lambda_1 < \cdots < \lambda_k$ with $k = \Theta(n)$ using only $O(1)$ max flow total work instead of $\Theta(n)$ max flow total work.
- We concentrate on the Goldberg-Tarjan Push-Relabel max flow algorithm.

The basic subproblem for GGT PFP

- Consider this: given an optimal x^0 for λ_0, and given some $\lambda_1 > \lambda_0$, try to find an optimal x^1 for λ_1 by starting at x^0 instead of starting from scratch.
  - Note that λ_1 might be given online, i.e., its value will depend in some way on x^0 (or its min cut).
  - Our real aim is to be able to do this repeatedly for a whole sequence of λ's, $\lambda_0 < \lambda_1 < \cdots < \lambda_k$ with $k = \Theta(n)$ using only $O(1)$ max flow total work instead of $\Theta(n)$ max flow total work.
- We concentrate on the Goldberg-Tarjan Push-Relabel max flow algorithm.
- We now review the key properties of Push-Relabel that we'll need here.

Key properties of Push-Relabel

1. It maintains a preflow x such that $e_i(x) \geq 0$ for all $i \neq s, t$.

Key properties of Push-Relabel

1. It maintains a preflow x such that $e_i(x) \geq 0$ for all $i \neq s, t$.
2. It maintains distance labels d such that $d_t = 0$, $d_s = n$, and $d_i \leq d_j + 1$ for all residual arcs $i \to j$, i.e., either $i \to j$ with $x_{ij} < u_{ij}(\lambda)$, or $j \to i$ with $x_{ji} > 0$; such a d is valid w.r.t. x.

Key properties of Push-Relabel

1. It maintains a preflow x such that $e_i(x) \geq 0$ for all $i \neq s, t$.
2. It maintains distance labels d such that $d_t = 0$, $d_s = n$, and $d_i \leq d_j + 1$ for all residual arcs $i \to j$, i.e., either $i \to j$ with $x_{ij} < u_{ij}(\lambda)$, or $j \to i$ with $x_{ji} > 0$; such a d is valid w.r.t. x.
3. The running time analysis depends only on the fact that the d_i are monotone non-decreasing throughout the algorithm, and that each $d_i \leq 2n$ during the algorithm.

Key properties of Push-Relabel

1. It maintains a preflow x such that $e_i(x) \geq 0$ for all $i \neq s, t$.
2. It maintains distance labels d such that $d_t = 0$, $d_s = n$, and $d_i \leq d_j + 1$ for all residual arcs $i \to j$, i.e., either $i \to j$ with $x_{ij} < u_{ij}(\lambda)$, or $j \to i$ with $x_{ji} > 0$; such a d is valid w.r.t. x.
3. The running time analysis depends only on the fact that the d_i are monotone non-decreasing throughout the algorithm, and that each $d_i \leq 2n$ during the algorithm.
4. When the algorithm finds an optimal preflow, $S^* = \{i \in N \mid d_i \geq n\}$ is a min cut.

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.
  - We assume throughout that we consider only values of λ such that $u_{ij}(\lambda) \geq 0$ to avoid feasibility questions. How?

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.
  - We assume throughout that we consider only values of λ such that $u_{ij}(\lambda) \geq 0$ to avoid feasibility questions. How?
- Update the λ_0-optimal preflow x^0 into a λ_1-feasible preflow x^1 w.r.t. the λ_0-optimal d_i's via the following Flow Update procedure:

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.
  - We assume throughout that we consider only values of λ such that $u_{ij}(\lambda) \geq 0$ to avoid feasibility questions. How?
- Update the λ_0-optimal preflow x^0 into a λ_1-feasible preflow x^1 w.r.t. the λ_0-optimal d_i's via the following Flow Update procedure:
  - For each arc $s \rightarrow i$ set $x^1_{si} = u_{si}(\lambda_1)$ [note: wrong in GGT paper].

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.
  - We assume throughout that we consider only values of λ such that $u_{ij}(\lambda) \geq 0$ to avoid feasibility questions. How?
- Update the λ_0-optimal preflow x^0 into a λ_1-feasible preflow x^1 w.r.t. the λ_0-optimal d_i's via the following Flow Update procedure:
  - For each arc $s \to i$ set $x^1_{si} = u_{si}(\lambda_1)$ [note: wrong in GGT paper].
  - For each arc $i \to j$ with $i \neq s$, $j \neq t$, set $x^1_{ij} = x^0_{ij}$.

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.
  - We assume throughout that we consider only values of λ such that $u_{ij}(\lambda) \geq 0$ to avoid feasibility questions. How?
- Update the λ_0-optimal preflow x^0 into a λ_1-feasible preflow x^1 w.r.t. the λ_0-optimal d_i's via the following Flow Update procedure:
  - For each arc $s \to i$ set $x^1_{si} = u_{si}(\lambda_1)$ [note: wrong in GGT paper].
  - For each arc $i \to j$ with $i \neq s$, $j \neq t$, set $x^1_{ij} = x^0_{ij}$.
  - For each arc $i \to t$ set $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\}$.

A fundamental algorithmic observation

- Somehow we choose some λ_0 and run ordinary Push-Relabel to get optimal preflow x^0.
- Now get/compute $\lambda_1 > \lambda_0$, and the new capacities $u_{ij}(\lambda_1)$.
  - We assume throughout that we consider only values of λ such that $u_{ij}(\lambda) \geq 0$ to avoid feasibility questions. How?
- Update the λ_0-optimal preflow x^0 into a λ_1-feasible preflow x^1 w.r.t. the λ_0-optimal d_i's via the following Flow Update procedure:
  - For each arc $s \to i$ set $x^1_{si} = u_{si}(\lambda_1)$ [note: wrong in GGT paper].
  - For each arc $i \to j$ with $i \neq s$, $j \neq t$, set $x^1_{ij} = x^0_{ij}$.
  - For each arc $i \to t$ set $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\}$.
- We need to prove that this x^1 is a feasible preflow w.r.t. x^0's optimal d_i's.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \rightarrow i$ and $i \rightarrow t$.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \rightarrow i$ and $i \rightarrow t$.
  - On $s \rightarrow i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \to i$ and $i \to t$.
  - On $s \to i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.
  - On $i \to t$ we have $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq u_{it}(\lambda_1)$, which is feasible and non-negative.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \to i$ and $i \to t$.
  - On $s \to i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.
  - On $i \to t$ we have $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq u_{it}(\lambda_1)$, which is feasible and non-negative.
- Second we need to show that x^1 is a preflow:

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \to i$ and $i \to t$.
  - On $s \to i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.
  - On $i \to t$ we have $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq u_{it}(\lambda_1)$, which is feasible and non-negative.
- Second we need to show that x^1 is a preflow:
  - Let $i \neq s, t$ be any node.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \to i$ and $i \to t$.
  - On $s \to i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.
  - On $i \to t$ we have $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq u_{it}(\lambda_1)$, which is feasible and non-negative.
- Second we need to show that x^1 is a preflow:
  - Let $i \neq s, t$ be any node.
  - In moving from x^0 to x^1, flows incident to i change only on $s \to i$ and $i \to t$.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \to i$ and $i \to t$.
  - On $s \to i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.
  - On $i \to t$ we have $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq u_{it}(\lambda_1)$, which is feasible and non-negative.
- Second we need to show that x^1 is a preflow:
  - Let $i \neq s, t$ be any node.
  - In moving from x^0 to x^1, flows incident to i change only on $s \to i$ and $i \to t$.
  - Since $b_{si} \geq 0$, $u_{si}(\lambda_1) \geq u_{si}(\lambda_0)$. Since x^1_{si} is chosen to saturate $s \to i$, we have that $x^1_{si} = u_{si}(\lambda_1) \geq u_{si}(\lambda_0) \geq x^0_{si}$. Thus flow change on $s \to i$ can only increase e_i.

Proof that Flow Update produces a feasible x^1

- First we need to show that x^1 satisfies boundedness:
  - In moving from x^0 to x^1, flow changes only on arcs $s \to i$ and $i \to t$.
  - On $s \to i$ we have $x^1_{si} = u_{si}(\lambda_1)$, which is feasible and non-negative.
  - On $i \to t$ we have $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq u_{it}(\lambda_1)$, which is feasible and non-negative.
- Second we need to show that x^1 is a preflow:
  - Let $i \neq s, t$ be any node.
  - In moving from x^0 to x^1, flows incident to i change only on $s \to i$ and $i \to t$.
  - Since $b_{si} \geq 0$, $u_{si}(\lambda_1) \geq u_{si}(\lambda_0)$. Since x^1_{si} is chosen to saturate $s \to i$, we have that $x^1_{si} = u_{si}(\lambda_1) \geq u_{si}(\lambda_0) \geq x^0_{si}$. Thus flow change on $s \to i$ can only increase e_i.
  - Since $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} \leq x^0_{it}$, this flow change can also only increase e_i.

Proof that old d_i's are still valid for x^1

- The only problem that can arise is if Flow Update creates a new residual arc $i \to j$ where we had $d_i > d_j + 1$.

Proof that old d_i's are still valid for x^1

- The only problem that can arise is if Flow Update creates a new residual arc $i \to j$ where we had $d_i > d_j + 1$.
- If we had $d_i > d_j + 1$ then by validity of d w.r.t. x^0 we had that $x^0_{ij} = u_{ij}(\lambda_0)$, and/or $x^0_{ji} = 0$.

Proof that old d_i's are still valid for x^1

- The only problem that can arise is if Flow Update creates a new residual arc $i \to j$ where we had $d_i > d_j + 1$.
- If we had $d_i > d_j + 1$ then by validity of d w.r.t. x^0 we had that $x^0_{ij} = u_{ij}(\lambda_0)$, and/or $x^0_{ji} = 0$.
- Since x^1 differs from x^0 only on $s \to i$ and $i \to t$ arcs, only these are possible counterexamples.

Proof that old d_i's are still valid for x^1

- The only problem that can arise is if Flow Update creates a new residual arc $i \to j$ where we had $d_i > d_j + 1$.
- If we had $d_i > d_j + 1$ then by validity of d w.r.t. x^0 we had that $x^0_{ij} = u_{ij}(\lambda_0)$, and/or $x^0_{ji} = 0$.
- Since x^1 differs from x^0 only on $s \to i$ and $i \to t$ arcs, only these are possible counterexamples.
- Suppose that $d_s > d_i + 1$ (so that $x^0_{si} = u_{si}(\lambda_0)$ and $x^0_{is} = 0$). Then Flow Update makes $x^1_{si} = u_{si}(\lambda_1)$, so that $s \to i$ is again saturated w.r.t. x^1; and leaves $x^1_{is} = x^0_{is} = 0$, so this is not a counterexample.

Proof that old d_i's are still valid for x^1

- The only problem that can arise is if Flow Update creates a new residual arc $i \to j$ where we had $d_i > d_j + 1$.
- If we had $d_i > d_j + 1$ then by validity of d w.r.t. x^0 we had that $x^0_{ij} = u_{ij}(\lambda_0)$, and/or $x^0_{ji} = 0$.
- Since x^1 differs from x^0 only on $s \to i$ and $i \to t$ arcs, only these are possible counterexamples.
- Suppose that $d_s > d_i + 1$ (so that $x^0_{si} = u_{si}(\lambda_0)$ and $x^0_{is} = 0$). Then Flow Update makes $x^1_{si} = u_{si}(\lambda_1)$, so that $s \to i$ is again saturated w.r.t. x^1; and leaves $x^1_{is} = x^0_{is} = 0$, so this is not a counterexample.
- Suppose that $d_i > d_t + 1$ (so that $x^0_{it} = u_{it}(\lambda_0)$ and $x^0_{ti} = 0$). Then Flow Update makes $x^1_{it} = \min\{x^0_{it}, u_{it}(\lambda_1)\} = u_{it}(\lambda_1)$ (since $b_{it} \leq 0$, and so $u_{it}(\lambda_1) \leq u_{it}(\lambda_0) = x^0_{it}$), so that $i \to t$ is again saturated w.r.t x^1; and leaves $x^1_{ti} = x^0_{ti} = 0$, so this is also not a counterexample.

Flow Update has nice consequences

1. We can start with this "advanced" warm-start x^1 and the same d instead of having to cold-start Push-Relabel.

Flow Update has nice consequences

1. We can start with this "advanced" warm-start x^1 and the same d instead of having to cold-start Push-Relabel.
2. Since the running time of Push-Relabel depends only on the d's, and since during the λ_0 and λ_1 solves d is non-decreasing and bounded by $2n$, the same running time applies to both.

Flow Update has nice consequences

1. We can start with this "advanced" warm-start x^1 and the same d instead of having to cold-start Push-Relabel.
2. Since the running time of Push-Relabel depends only on the d's, and since during the λ_0 and λ_1 solves d is non-decreasing and bounded by $2n$, the same running time applies to both.
   - (Flow Update does cause some overhead: $O(n)$ work for each λ_h, for a total of $O(n^2)$, which is not a bottleneck.)

Flow Update has nice consequences

1. We can start with this "advanced" warm-start x^1 and the same d instead of having to cold-start Push-Relabel.
2. Since the running time of Push-Relabel depends only on the d's, and since during the λ_0 and λ_1 solves d is non-decreasing and bounded by $2n$, the same running time applies to both.
   - (Flow Update does cause some overhead: $O(n)$ work for each λ_h, for a total of $O(n^2)$, which is not a bottleneck.)
3. Even better, once we transform the initial x^1 into an optimal x^1, if we now have a $\lambda_2 > \lambda_1$, we can do the same Flow Update trick to the optimal x^1 (and its new optimal d) to get an initial feasible x^2, then for $\lambda_3 > \lambda_2$, ...

Flow Update has nice consequences

1. We can start with this "advanced" warm-start x^1 and the same d instead of having to cold-start Push-Relabel.
2. Since the running time of Push-Relabel depends only on the d's, and since during the λ_0 and λ_1 solves d is non-decreasing and bounded by $2n$, the same running time applies to both.
   - (Flow Update does cause some overhead: $O(n)$ work for each λ_h, for a total of $O(n^2)$, which is not a bottleneck.)
3. Even better, once we transform the initial x^1 into an optimal x^1, if we now have a $\lambda_2 > \lambda_1$, we can do the same Flow Update trick to the optimal x^1 (and its new optimal d) to get an initial feasible x^2, then for $\lambda_3 > \lambda_2$, ...
4. Thus using Flow Update solves for all $k = \Theta(n)$ values of λ, $\lambda_0 < \lambda_1 < \cdots < \lambda_k$, in $O(1)$ Push-Relabels.

Flow Update has nice consequences

1. We can start with this "advanced" warm-start x^1 and the same d instead of having to cold-start Push-Relabel.
2. Since the running time of Push-Relabel depends only on the d's, and since during the λ_0 and λ_1 solves d is non-decreasing and bounded by $2n$, the same running time applies to both.
   - (Flow Update does cause some overhead: $O(n)$ work for each λ_h, for a total of $O(n^2)$, which is not a bottleneck.)
3. Even better, once we transform the initial x^1 into an optimal x^1, if we now have a $\lambda_2 > \lambda_1$, we can do the same Flow Update trick to the optimal x^1 (and its new optimal d) to get an initial feasible x^2, then for $\lambda_3 > \lambda_2$, ...
4. Thus using Flow Update solves for all $k = \Theta(n)$ values of λ, $\lambda_0 < \lambda_1 < \cdots < \lambda_k$, in $O(1)$ Push-Relabels.
5. Even better, we get the Structural Property for free: Since the min cut S_h for each λ_h is determined by the set of nodes i with $d_i \geq n$, and since d is monotone non-decreasing, we get that $S_0 \subseteq S_1 \subseteq \cdots \subseteq S_k$.

From the basic algorithm towards PFP

- Whew! We have made good progress, but the "sequence of λ's" problem is still far from PFP.

From the basic algorithm towards PFP

- Whew! We have made good progress, but the "sequence of λ's" problem is still far from PFP.
  - However, even this is good enough for an important subclass of GGT-type problems, see Question 4.

From the basic algorithm towards PFP

- Whew! We have made good progress, but the "sequence of λ's" problem is still far from PFP.
  - However, even this is good enough for an important subclass of GGT-type problems, see Question 4.
- Since we often don't care about max flows, often it's simpler to run Push-Relabel only until we get an optimal *preflow* instead of an optimal *flow*.

From the basic algorithm towards PFP

- Whew! We have made good progress, but the "sequence of λ's" problem is still far from PFP.
  - However, even this is good enough for an important subclass of GGT-type problems, see Question 4.
- Since we often don't care about max flows, often it's simpler to run Push-Relabel only until we get an optimal *preflow* instead of an optimal *flow*.
  - Another improvement: many GGT applications generate *bipartite* networks where $N = \{s\} \cup V_1 \cup V_2 \cup \{t\}$ such that all arcs are in (s, V_1), (V_1, V_2), or (V_2, t). Often $|V_1| \ll |V_2|$, and then we can replace n by $|V_1|$ in the running time.

From the basic algorithm towards PFP

- Whew! We have made good progress, but the "sequence of λ's" problem is still far from PFP.
  - However, even this is good enough for an important subclass of GGT-type problems, see Question 4.
- Since we often don't care about max flows, often it's simpler to run Push-Relabel only until we get an optimal *preflow* instead of an optimal *flow*.
  - Another improvement: many GGT applications generate *bipartite* networks where $N = \{s\} \cup V_1 \cup V_2 \cup \{t\}$ such that all arcs are in (s, V_1), (V_1, V_2), or (V_2, t). Often $|V_1| \ll |V_2|$, and then we can replace n by $|V_1|$ in the running time.
- Suppose that somehow we have optimal preflow x^1 and min cut S_1 for λ_1, and x^3, S_3 for $\lambda_3 > \lambda_1$.

From the basic algorithm towards PFP

- Whew! We have made good progress, but the "sequence of λ's" problem is still far from PFP.
  - However, even this is good enough for an important subclass of GGT-type problems, see Question 4.
- Since we often don't care about max flows, often it's simpler to run Push-Relabel only until we get an optimal *preflow* instead of an optimal *flow*.
  - Another improvement: many GGT applications generate *bipartite* networks where $N = \{s\} \cup V_1 \cup V_2 \cup \{t\}$ such that all arcs are in (s, V_1), (V_1, V_2), or (V_2, t). Often $|V_1| \ll |V_2|$, and then we can replace n by $|V_1|$ in the running time.
- Suppose that somehow we have optimal preflow x^1 and min cut S_1 for λ_1, and x^3, S_3 for $\lambda_3 > \lambda_1$.
- Let's assume that there are one or more breakpoints between λ_1 and λ_3, so that $S_1 \neq S_3$.

Geometry of cuts S_1 and S_3

- Any cut S determines a line in the 2-d space with x-axis λ and y-axis $\mathrm{cap}^*(\lambda)$: Let S' denote $S - \{s\}$. Then $\mathrm{cap}(S, \lambda) = a(\delta^+(S)) + \lambda[b(s, \bar{S}') + b(S', t)]$, so $a(\delta^+(S))$ is the intercept, and $b(s, \bar{S}') + b(S', t)$ is the slope of this line.

Geometry of cuts S_1 and S_3

- Any cut S determines a line in the 2-d space with x-axis λ and y-axis $\mathrm{cap}^*(\lambda)$: Let S' denote $S - \{s\}$. Then $\mathrm{cap}(S, \lambda) = a(\delta^+(S)) + \lambda[b(s, \bar{S}') + b(S', t)]$, so $a(\delta^+(S))$ is the intercept, and $b(s, \bar{S}') + b(S', t)$ is the slope of this line.
- If S is a min cut for some value of λ, then the line $\mathrm{cap}(S, \lambda)$ is part of $\mathrm{cap}^*(\lambda)$.

Geometry of cuts S_1 and S_3

- Any cut S determines a line in the 2-d space with x-axis λ and y-axis $\mathrm{cap}^*(\lambda)$: Let S' denote $S - \{s\}$. Then $\mathrm{cap}(S, \lambda) = a(\delta^+(S)) + \lambda[b(s, \bar{S}') + b(S', t)]$, so $a(\delta^+(S))$ is the intercept, and $b(s, \bar{S}') + b(S', t)$ is the slope of this line.
- If S is a min cut for some value of λ, then the line $\mathrm{cap}(S, \lambda)$ is part of $\mathrm{cap}^*(\lambda)$.
- Denote the line for S_1 as $a_1 + \lambda b_1$, so that, e.g., $a_1 = a(\delta^+(S_1))$.

Geometry of cuts S_1 and S_3

- Any cut S determines a line in the 2-d space with x-axis λ and y-axis $\mathrm{cap}^*(\lambda)$: Let S' denote $S - \{s\}$. Then $\mathrm{cap}(S, \lambda) = a(\delta^+(S)) + \lambda[b(s, \bar{S}') + b(S', t)]$, so $a(\delta^+(S))$ is the intercept, and $b(s, \bar{S}') + b(S', t)$ is the slope of this line.
- If S is a min cut for some value of λ, then the line $\mathrm{cap}(S, \lambda)$ is part of $\mathrm{cap}^*(\lambda)$.
- Denote the line for S_1 as $a_1 + \lambda b_1$, so that, e.g., $a_1 = a(\delta^+(S_1))$.
- Since $\mathrm{cap}^*(\lambda)$ is concave, $b_3 \leq b_1$. Since we're assuming that there are breakpoints between λ_1 and λ_3 in fact $b_3 < b_1$.

Geometry of cuts S_1 and S_3

- Any cut S determines a line in the 2-d space with x-axis λ and y-axis $\mathrm{cap}^*(\lambda)$: Let S' denote $S - \{s\}$. Then $\mathrm{cap}(S, \lambda) = a(\delta^+(S)) + \lambda[b(s, \bar{S}') + b(S', t)]$, so $a(\delta^+(S))$ is the intercept, and $b(s, \bar{S}') + b(S', t)$ is the slope of this line.
- If S is a min cut for some value of λ, then the line $\mathrm{cap}(S, \lambda)$ is part of $\mathrm{cap}^*(\lambda)$.
- Denote the line for S_1 as $a_1 + \lambda b_1$, so that, e.g., $a_1 = a(\delta^+(S_1))$.
- Since $\mathrm{cap}^*(\lambda)$ is concave, $b_3 \leq b_1$. Since we're assuming that there are breakpoints between λ_1 and λ_3 in fact $b_3 < b_1$.
- The intersection of S_1's line and S_3's line is $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$, with $\lambda_1 < \lambda_2 < \lambda_3$ (Why?).

Doing GGT for $\lambda = \lambda_2$

- Going upwards from λ_1 to λ_2, we already know how to do GGT, but we don't yet know how to go downwards from λ_3 to λ_2.

Doing GGT for $\lambda = \lambda_2$

- Going upwards from λ_1 to λ_2, we already know how to do GGT, but we don't yet know how to go downwards from λ_3 to λ_2.
- Consider the reversed network where we reverse all arcs, interchange s and t, and replace λ with $-\lambda$.

Doing GGT for $\lambda = \lambda_2$

- Going upwards from λ_1 to λ_2, we already know how to do GGT, but we don't yet know how to go downwards from λ_3 to λ_2.
- Consider the reversed network where we reverse all arcs, interchange s and t, and replace λ with $-\lambda$.
  - Note that the reversed network still has the GGT property.

Doing GGT for $\lambda = \lambda_2$

- Going upwards from λ_1 to λ_2, we already know how to do GGT, but we don't yet know how to go downwards from λ_3 to λ_2.
- Consider the reversed network where we reverse all arcs, interchange s and t, and replace λ with $-\lambda$.
  - Note that the reversed network still has the GGT property.
  - Then decreases of λ in the original network correspond to increases of $-\lambda$ in the reversed network, so we can still use GGT to move from λ_3 downwards to λ_2.

Doing GGT for $\lambda = \lambda_2$

- Going upwards from λ_1 to λ_2, we already know how to do GGT, but we don't yet know how to go downwards from λ_3 to λ_2.
- Consider the reversed network where we reverse all arcs, interchange s and t, and replace λ with $-\lambda$.
  - Note that the reversed network still has the GGT property.
  - Then decreases of λ in the original network correspond to increases of $-\lambda$ in the reversed network, so we can still use GGT to move from λ_3 downwards to λ_2.
- This ability to run GGT in both directions to do a sort of binary search is the key to solving the full PFP.

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.
- The strongly connected components of the residual graph w.r.t. a max flow is the same for any max flow (exercise).

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.
- The strongly connected components of the residual graph w.r.t. a max flow is the same for any max flow (exercise).
- There is a 1–1 correspondence between min cuts and closures (ideals; see Question 3) of the residual graph.

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.
- The strongly connected components of the residual graph w.r.t. a max flow is the same for any max flow (exercise).
- There is a 1–1 correspondence between min cuts and closures (ideals; see Question 3) of the residual graph.
- Thus unions and intersections of min cuts are again min cuts, and so there is a unique minimal min cut $S_{\min}$ and maximal min cut $S_{\max}$.

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.
- The strongly connected components of the residual graph w.r.t. a max flow is the same for any max flow (exercise).
- There is a 1–1 correspondence between min cuts and closures (ideals; see Question 3) of the residual graph.
- Thus unions and intersections of min cuts are again min cuts, and so there is a unique minimal min cut $S_{\min}$ and maximal min cut $S_{\max}$.
- For GGT networks, if $S \subseteq S'$ are two cuts, then (the slope of S) $\geq$ (the slope of S'); in particular, (the slope of $S_{\min}$ for λ') $:= \mathsf{sl}_{\min}(\lambda') \geq \mathsf{sl}_{\max}(\lambda') :=$ (the slope of $S_{\max}$ for λ').

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.
- The strongly connected components of the residual graph w.r.t. a max flow is the same for any max flow (exercise).
- There is a 1–1 correspondence between min cuts and closures (ideals; see Question 3) of the residual graph.
- Thus unions and intersections of min cuts are again min cuts, and so there is a unique minimal min cut $S_{\min}$ and maximal min cut $S_{\max}$.
- For GGT networks, if $S \subseteq S'$ are two cuts, then (the slope of S) $\geq$ (the slope of S'); in particular, (the slope of $S_{\min}$ for λ') $:= \mathsf{sl}_{\min}(\lambda') \geq \mathsf{sl}_{\max}(\lambda') :=$ (the slope of $S_{\max}$ for λ').
- λ' is a breakpoint iff $\mathsf{sl}_{\min}(\lambda') > \mathsf{sl}_{\max}(\lambda')$.

Facts about min cuts and breakpoints

- Any max flow network can have many min cuts.
- The strongly connected components of the residual graph w.r.t. a max flow is the same for any max flow (exercise).
- There is a 1–1 correspondence between min cuts and closures (ideals; see Question 3) of the residual graph.
- Thus unions and intersections of min cuts are again min cuts, and so there is a unique minimal min cut $S_{\min}$ and maximal min cut $S_{\max}$.
- For GGT networks, if $S \subseteq S'$ are two cuts, then (the slope of S) $\geq$ (the slope of S'); in particular, (the slope of $S_{\min}$ for λ') $:= \mathsf{sl}_{\min}(\lambda') \geq \mathsf{sl}_{\max}(\lambda') :=$ (the slope of $S_{\max}$ for λ').
- λ' is a breakpoint iff $\mathsf{sl}_{\min}(\lambda') > \mathsf{sl}_{\max}(\lambda')$.
- Question 8 (b) shows that if S_1 is a min cut for λ_1 and S_3 is a min cut for λ_3, then $S_1 \cap S_3$ is a min cut for λ_1 and $S_1 \cup S_3$ is a min cut for λ_3. Therefore if $\lambda_2 \in [\lambda_1, \lambda_3]$ and S_2 is a min cut for λ_2, then $S' = (S_1 \cup S_2) \cap S_3$ is also a min cut for λ_2, with $S_1 \subseteq S' \subseteq S_3$.

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.
- First we need to compute an initial value of λ_1 small enough, and of λ_3 large enough, that $\lambda_1 \leq \lambda_{\min} \leq \lambda_3$.

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.
- First we need to compute an initial value of λ_1 small enough, and of λ_3 large enough, that $\lambda_1 \leq \lambda_{\min} \leq \lambda_3$.
  - Denote $N - \{s, t\}$ by N'. Suppose that λ is so small that $u_{si}(\lambda) + u(N', i) < u_{it}(\lambda)$. Then it would always greedily pay to put i on the t-side of a cut instead of the s side.

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.
- First we need to compute an initial value of λ_1 small enough, and of λ_3 large enough, that $\lambda_1 \leq \lambda_{\min} \leq \lambda_3$.
  - Denote $N - \{s, t\}$ by N'. Suppose that λ is so small that $u_{si}(\lambda) + u(N', i) < u_{it}(\lambda)$. Then it would always greedily pay to put i on the t-side of a cut instead of the s side.
  - This condition translates into $a_{si} + \lambda b_{si} + u(N', i) < a_{it} + \lambda b_{it}$. By GGT we know that $b_{si} - b_{it} \geq 0$, and the condition only makes sense for i such that $b_{si} - b_{it} > 0$.

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.
- First we need to compute an initial value of λ_1 small enough, and of λ_3 large enough, that $\lambda_1 \leq \lambda_{\min} \leq \lambda_3$.
  - Denote $N - \{s, t\}$ by N'. Suppose that λ is so small that $u_{si}(\lambda) + u(N', i) < u_{it}(\lambda)$. Then it would always greedily pay to put i on the t-side of a cut instead of the s side.
  - This condition translates into $a_{si} + \lambda b_{si} + u(N', i) < a_{it} + \lambda b_{it}$. By GGT we know that $b_{si} - b_{it} \geq 0$, and the condition only makes sense for i such that $b_{si} - b_{it} > 0$.
  - This leads to choosing

$$\lambda_1 = \min_{\substack{i \in N' \text{ s.t.} \\ b_{si} - b_{it} > 0}} \left(\frac{a_{it} - a_{si} - u(N', i)}{b_{si} - b_{it}} \right) - 1.$$

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.
- First we need to compute an initial value of λ_1 small enough, and of λ_3 large enough, that $\lambda_1 \leq \lambda_{\min} \leq \lambda_3$.
  - Denote $N - \{s, t\}$ by N'. Suppose that λ is so small that $u_{si}(\lambda) + u(N', i) < u_{it}(\lambda)$. Then it would always greedily pay to put i on the t-side of a cut instead of the s side.
  - This condition translates into $a_{si} + \lambda b_{si} + u(N', i) < a_{it} + \lambda b_{it}$. By GGT we know that $b_{si} - b_{it} \geq 0$, and the condition only makes sense for i such that $b_{si} - b_{it} > 0$.
  - This leads to choosing

$$\lambda_1 = \min_{\substack{i \in N' \text{ s.t.} \\ b_{si} - b_{it} > 0}} \left(\frac{a_{it} - a_{si} - u(N', i)}{b_{si} - b_{it}} \right) - 1.$$

  - Similar analysis on the other side says that we should choose

$$\lambda_3 = \max_{\substack{i \in N' \text{ s.t.} \\ b_{si} - b_{it} > 0}} \left(\frac{a_{it} - a_{si} + u(i, N')}{b_{si} - b_{it}} \right) + 1.$$

Finding the smallest breakpoint

- Denote the smallest breakpoint by $\lambda_{\min}$.
- First we need to compute an initial value of λ_1 small enough, and of λ_3 large enough, that $\lambda_1 \leq \lambda_{\min} \leq \lambda_3$.
  - Denote $N - \{s, t\}$ by N'. Suppose that λ is so small that $u_{si}(\lambda) + u(N', i) < u_{it}(\lambda)$. Then it would always greedily pay to put i on the t-side of a cut instead of the s side.
  - This condition translates into $a_{si} + \lambda b_{si} + u(N', i) < a_{it} + \lambda b_{it}$. By GGT we know that $b_{si} - b_{it} \geq 0$, and the condition only makes sense for i such that $b_{si} - b_{it} > 0$.
  - This leads to choosing

$$\lambda_1 = \min_{\substack{i \in N' \text{ s.t.} \\ b_{si} - b_{it} > 0}} \left(\frac{a_{it} - a_{si} - u(N', i)}{b_{si} - b_{it}} \right) - 1.$$

  - Similar analysis on the other side says that we should choose

$$\lambda_3 = \max_{\substack{i \in N' \text{ s.t.} \\ b_{si} - b_{it} > 0}} \left(\frac{a_{it} - a_{si} + u(i, N')}{b_{si} - b_{it}} \right) + 1.$$

  - *All* breakpoints are in $[\lambda_1, \lambda_3]$.

Smallest breakpoint algorithm

1. At a generic step we have an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\min}$. We are going to iteratively reduce λ_3 until it becomes $\lambda_{\min}$.

Smallest breakpoint algorithm

1. At a generic step we have an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\min}$. We are going to iteratively reduce λ_3 until it becomes $\lambda_{\min}$.
2. Compute an initial min cut S_1 at λ_1, and S_3 at λ_3.

Smallest breakpoint algorithm

1. At a generic step we have an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\min}$. We are going to iteratively reduce λ_3 until it becomes $\lambda_{\min}$.
2. Compute an initial min cut S_1 at λ_1, and S_3 at λ_3.
3. If $S_1 = S_3$ (i.e., if the slopes of S_1's line equals the slope of S_3's line), then $\lambda_3 = \lambda_{\min}$, so we can stop.

Smallest breakpoint algorithm

1. At a generic step we have an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\min}$. We are going to iteratively reduce λ_3 until it becomes $\lambda_{\min}$.
2. Compute an initial min cut S_1 at λ_1, and S_3 at λ_3.
3. If $S_1 = S_3$ (i.e., if the slopes of S_1's line equals the slope of S_3's line), then $\lambda_3 = \lambda_{\min}$, so we can stop.
4. Otherwise, compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.

Smallest breakpoint algorithm

1. At a generic step we have an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\min}$. We are going to iteratively reduce λ_3 until it becomes $\lambda_{\min}$.
2. Compute an initial min cut S_1 at λ_1, and S_3 at λ_3.
3. If $S_1 = S_3$ (i.e., if the slopes of S_1's line equals the slope of S_3's line), then $\lambda_3 = \lambda_{\min}$, so we can stop.
4. Otherwise, compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
5. Continue the λ_3 cut computation in a reversed GGT downwards from λ_3 to λ_2, getting min cut S_2 at λ_2.

Smallest breakpoint algorithm

1. At a generic step we have an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\min}$. We are going to iteratively reduce λ_3 until it becomes $\lambda_{\min}$.
2. Compute an initial min cut S_1 at λ_1, and S_3 at λ_3.
3. If $S_1 = S_3$ (i.e., if the slopes of S_1's line equals the slope of S_3's line), then $\lambda_3 = \lambda_{\min}$, so we can stop.
4. Otherwise, compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
5. Continue the λ_3 cut computation in a reversed GGT downwards from λ_3 to λ_2, getting min cut S_2 at λ_2.
6. Replace λ_3 by λ_2, and S_3 by S_2, and go to 3.

Analysis of smallest breakpoint algorithm

- Each time that we compute a new λ_2 that turns out not to be $\lambda_{\min}$, we find a new slope of the curve $\mathrm{cap}^*(\lambda)$.

Analysis of smallest breakpoint algorithm

- Each time that we compute a new λ_2 that turns out not to be $\lambda_{\min}$, we find a new slope of the curve $\mathrm{cap}^*(\lambda)$.
- This means that we can have at most $n-1$ new values of λ_2 before terminating.

Analysis of smallest breakpoint algorithm

- Each time that we compute a new λ_2 that turns out not to be $\lambda_{\min}$, we find a new slope of the curve $\mathrm{cap}^*(\lambda)$.
- This means that we can have at most $n-1$ new values of λ_2 before terminating.
- All min cuts come from the descending sequence of λ_3's, so they all can be computed via (reversed) GGT in $O(1)$ min cut time.

Analysis of smallest breakpoint algorithm

- Each time that we compute a new λ_2 that turns out not to be $\lambda_{\min}$, we find a new slope of the curve $\mathrm{cap}^*(\lambda)$.
- This means that we can have at most $n-1$ new values of λ_2 before terminating.
- All min cuts come from the descending sequence of λ_3's, so they all can be computed via (reversed) GGT in $O(1)$ min cut time.
- Thus the whole algorithm takes $O(1)$ min cut time.

Analysis of smallest breakpoint algorithm

- Each time that we compute a new λ_2 that turns out not to be $\lambda_{\min}$, we find a new slope of the curve $\mathrm{cap}^*(\lambda)$.
- This means that we can have at most $n-1$ new values of λ_2 before terminating.
- All min cuts come from the descending sequence of λ_3's, so they all can be computed via (reversed) GGT in $O(1)$ min cut time.
- Thus the whole algorithm takes $O(1)$ min cut time.
  - There is some overhead in computing successive values of λ_2. Each one costs $O(m)$ time and there are only $O(n)$ of them, so this is not a bottleneck.

Analysis of smallest breakpoint algorithm

- Each time that we compute a new λ_2 that turns out not to be $\lambda_{\min}$, we find a new slope of the curve $\mathrm{cap}^*(\lambda)$.
- This means that we can have at most $n-1$ new values of λ_2 before terminating.
- All min cuts come from the descending sequence of λ_3's, so they all can be computed via (reversed) GGT in $O(1)$ min cut time.
- Thus the whole algorithm takes $O(1)$ min cut time.
  - There is some overhead in computing successive values of λ_2. Each one costs $O(m)$ time and there are only $O(n)$ of them, so this is not a bottleneck.
  - There is an analogous algorithm for finding the *largest* breakpoint.

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.
- Since $\mathrm{cap}^*(\lambda)$ is piecewise linear, $\lambda_{\max}$ equals a breakpoint.

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.
- Since $\mathrm{cap}^*(\lambda)$ is piecewise linear, $\lambda_{\max}$ equals a breakpoint.
  - (Unless the slope is positive at the initial value of λ_3, or negative at the initial value of λ_1, in which case the max is unbounded, and these are easy to check.)

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.
- Since $\mathrm{cap}^*(\lambda)$ is piecewise linear, $\lambda_{\max}$ equals a breakpoint.
  - (Unless the slope is positive at the initial value of λ_3, or negative at the initial value of λ_1, in which case the max is unbounded, and these are easy to check.)
- As before, we keep an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\max}$.

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.
- Since $\mathrm{cap}^*(\lambda)$ is piecewise linear, $\lambda_{\max}$ equals a breakpoint.
  - (Unless the slope is positive at the initial value of λ_3, or negative at the initial value of λ_1, in which case the max is unbounded, and these are easy to check.)
- As before, we keep an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\max}$.
  - We maintain that the slope at λ_1 is positive, and at λ_3 is negative $\implies \lambda_{\max} \in [\lambda_1, \lambda_3]$.

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.
- Since $\mathrm{cap}^*(\lambda)$ is piecewise linear, $\lambda_{\max}$ equals a breakpoint.
  - (Unless the slope is positive at the initial value of λ_3, or negative at the initial value of λ_1, in which case the max is unbounded, and these are easy to check.)
- As before, we keep an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\max}$.
  - We maintain that the slope at λ_1 is positive, and at λ_3 is negative $\implies \lambda_{\max} \in [\lambda_1, \lambda_3]$.
- But this time, when we compute λ_2, it might happen that $\lambda_{\max}$ is in $[\lambda_2, \lambda_3]$, and so we'd have to replace λ_1 by λ_2.

Solving $\max_\lambda \mathrm{cap}^*(\lambda)$

- Across all values of λ, which one has the largest min cut value (max flow value)? Call it $\lambda_{\max}$.
- Since $\mathrm{cap}^*(\lambda)$ is piecewise linear, $\lambda_{\max}$ equals a breakpoint.
  - (Unless the slope is positive at the initial value of λ_3, or negative at the initial value of λ_1, in which case the max is unbounded, and these are easy to check.)
- As before, we keep an interval $[\lambda_1, \lambda_3]$ guaranteed to contain $\lambda_{\max}$.
  - We maintain that the slope at λ_1 is positive, and at λ_3 is negative $\implies \lambda_{\max} \in [\lambda_1, \lambda_3]$.
- But this time, when we compute λ_2, it might happen that $\lambda_{\max}$ is in $[\lambda_2, \lambda_3]$, and so we'd have to replace λ_1 by λ_2.
- This means that we now need to run GGT both upwards from λ_1 *and* downwards from λ_3.

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. Simultaneously continue the λ_1 GGT upwards to λ_2, and the λ_3 GGT downwards to λ_2 until the first one completes, getting min cut S_2 at λ_2.

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. Simultaneously continue the λ_1 GGT upwards to λ_2, and the λ_3 GGT downwards to λ_2 until the first one completes, getting min cut S_2 at λ_2.
4. Further compute slopes $\mathrm{sl}_{\min}(\lambda_2)$ and $\mathrm{sl}_{\max}(\lambda_2)$ of $\mathrm{cap}^*(\lambda)$ at λ_2 (how?).

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. Simultaneously continue the λ_1 GGT upwards to λ_2, and the λ_3 GGT downwards to λ_2 until the first one completes, getting min cut S_2 at λ_2.
4. Further compute slopes $\mathsf{sl}_{\min}(\lambda_2)$ and $\mathsf{sl}_{\max}(\lambda_2)$ of $\mathrm{cap}^*(\lambda)$ at λ_2 (how?).
5. If $0 \in [\mathsf{sl}_{\min}, \mathsf{sl}_{\max}]$, then $\lambda_{\max} = \lambda_2$.

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. Simultaneously continue the λ_1 GGT upwards to λ_2, and the λ_3 GGT downwards to λ_2 until the first one completes, getting min cut S_2 at λ_2.
4. Further compute slopes $\mathsf{sl}_{\min}(\lambda_2)$ and $\mathsf{sl}_{\max}(\lambda_2)$ of $\mathrm{cap}^*(\lambda)$ at λ_2 (how?).
5. If $0 \in [\mathsf{sl}_{\min}, \mathsf{sl}_{\max}]$, then $\lambda_{\max} = \lambda_2$.
6. Otherwise, if $\mathsf{sl}_{\min} > 0$ do: Finish the upwards GGT if it was not the first to stop, replace λ_1 by λ_2 and S_1 by S_2 and abandon the downward GGT by returning it to its λ_3 state;

Algorithm for computing $\lambda_{\max}$

1. As before, compute initial $[\lambda_1, \lambda_3]$ and min cuts S_1 and S_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. Simultaneously continue the λ_1 GGT upwards to λ_2, and the λ_3 GGT downwards to λ_2 until the first one completes, getting min cut S_2 at λ_2.
4. Further compute slopes $\mathrm{sl}_{\min}(\lambda_2)$ and $\mathrm{sl}_{\max}(\lambda_2)$ of $\mathrm{cap}^*(\lambda)$ at λ_2 (how?).
5. If $0 \in [\mathrm{sl}_{\min}, \mathrm{sl}_{\max}]$, then $\lambda_{\max} = \lambda_2$.
6. Otherwise, if $\mathrm{sl}_{\min} > 0$ do: Finish the upwards GGT if it was not the first to stop, replace λ_1 by λ_2 and S_1 by S_2 and abandon the downward GGT by returning it to its λ_3 state;
7. If instead $\mathrm{sl}_{\max} < 0$ do: Finish the downwards GGT if it was not the first to stop, replace λ_3 by λ_2 and S_3 by S_2 and abandon the upward GGT by returning it to its λ_1 state, and go to 2.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.
- There is a monotone increasing sequence of λ_2's replacing λ_1, and decreasing λ_2's replacing λ_3, so both GGTs take $O(1)$ min cut time.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.
- There is a monotone increasing sequence of λ_2's replacing λ_1, and decreasing λ_2's replacing λ_3, so both GGTs take $O(1)$ min cut time.
  - The time "wasted" in running a GGT that is abandoned and backed up is dominated by the time spent on the accepted GGT.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.
- There is a monotone increasing sequence of λ_2's replacing λ_1, and decreasing λ_2's replacing λ_3, so both GGTs take $O(1)$ min cut time.
  - The time "wasted" in running a GGT that is abandoned and backed up is dominated by the time spent on the accepted GGT.
  - There is only $O(m)$ overhead in backing up an abandoned GGT to its previous state, and so this is not a bottleneck.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.
- There is a monotone increasing sequence of λ_2's replacing λ_1, and decreasing λ_2's replacing λ_3, so both GGTs take $O(1)$ min cut time.
  - The time "wasted" in running a GGT that is abandoned and backed up is dominated by the time spent on the accepted GGT.
  - There is only $O(m)$ overhead in backing up an abandoned GGT to its previous state, and so this is not a bottleneck.
- How to compute $\mathrm{sl}_{\min}$ and $\mathrm{sl}_{\max}$? Clearly $\mathrm{sl}_{\min}$ comes from $S_{\max}$, which can be found using an $O(m)$ search; similarly $\mathrm{sl}_{\max}$ comes from $S_{\min}$.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.
- There is a monotone increasing sequence of λ_2's replacing λ_1, and decreasing λ_2's replacing λ_3, so both GGTs take $O(1)$ min cut time.
  - The time "wasted" in running a GGT that is abandoned and backed up is dominated by the time spent on the accepted GGT.
  - There is only $O(m)$ overhead in backing up an abandoned GGT to its previous state, and so this is not a bottleneck.
- How to compute $\mathrm{sl}_{\min}$ and $\mathrm{sl}_{\max}$? Clearly $\mathrm{sl}_{\min}$ comes from $S_{\max}$, which can be found using an $O(m)$ search; similarly $\mathrm{sl}_{\max}$ comes from $S_{\min}$.
  - More overhead at each step, but again this is not a bottleneck.

Analysis of algorithm for computing $\lambda_{\max}$

- Again there is a change in slope every time we get a new λ_2, so there are at most $n-1$ iterations.
- There is a monotone increasing sequence of λ_2's replacing λ_1, and decreasing λ_2's replacing λ_3, so both GGTs take $O(1)$ min cut time.
  - The time "wasted" in running a GGT that is abandoned and backed up is dominated by the time spent on the accepted GGT.
  - There is only $O(m)$ overhead in backing up an abandoned GGT to its previous state, and so this is not a bottleneck.
- How to compute $\mathrm{sl}_{\min}$ and $\mathrm{sl}_{\max}$? Clearly $\mathrm{sl}_{\min}$ comes from $S_{\max}$, which can be found using an $O(m)$ search; similarly $\mathrm{sl}_{\max}$ comes from $S_{\min}$.
  - More overhead at each step, but again this is not a bottleneck.
- Thus the whole algorithm takes $O(1)$ min cut time.

Finding all breakpoints of $\mathrm{cap}^*(\lambda)$

- This algorithm would also clearly solve the $\lambda_{\min}$ and $\lambda_{\max}$ problems, but it is more complicated, so it was useful to go through those as warm-ups first.

Finding all breakpoints of $\mathrm{cap}^*(\lambda)$

- This algorithm would also clearly solve the $\lambda_{\min}$ and $\lambda_{\max}$ problems, but it is more complicated, so it was useful to go through those as warm-ups first.
- We use the same idea as the $\lambda_{\max}$ algorithm, except that instead of throwing away the endpoint of the interval $[\lambda_1, \lambda_2]$ or $[\lambda_2, \lambda_3]$ not containing $\lambda_{\max}$, we keep both subintervals and recursively apply the same ideas.

Finding all breakpoints of $\mathrm{cap}^*(\lambda)$

- This algorithm would also clearly solve the $\lambda_{\min}$ and $\lambda_{\max}$ problems, but it is more complicated, so it was useful to go through those as warm-ups first.
- We use the same idea as the $\lambda_{\max}$ algorithm, except that instead of throwing away the endpoint of the interval $[\lambda_1, \lambda_2]$ or $[\lambda_2, \lambda_3]$ not containing $\lambda_{\max}$, we keep both subintervals and recursively apply the same ideas.
- Suppose that we start on $[\lambda_1, \lambda_3]$ having optimal flow x^1 at λ_1 resulting from an upwards GGT to λ_1, and optimal flow x^3 at λ_3 resulting from a downwards GGT to λ_3, and we want to extend to $\lambda_2 \in [\lambda_1, \lambda_3]$.

Finding all breakpoints of $\mathrm{cap}^*(\lambda)$

- This algorithm would also clearly solve the $\lambda_{\min}$ and $\lambda_{\max}$ problems, but it is more complicated, so it was useful to go through those as warm-ups first.
- We use the same idea as the $\lambda_{\max}$ algorithm, except that instead of throwing away the endpoint of the interval $[\lambda_1, \lambda_2]$ or $[\lambda_2, \lambda_3]$ not containing $\lambda_{\max}$, we keep both subintervals and recursively apply the same ideas.
- Suppose that we start on $[\lambda_1, \lambda_3]$ having optimal flow x^1 at λ_1 resulting from an upwards GGT to λ_1, and optimal flow x^3 at λ_3 resulting from a downwards GGT to λ_3, and we want to extend to $\lambda_2 \in [\lambda_1, \lambda_3]$.
- We can simultaneously continue the upwards GGT from x^1 to λ_2 to get $x^{2,\mathsf{U}}$, and the downwards GGT from x^3 to λ_2 to get $x^{2,\mathsf{D}}$.

Finding all breakpoints of $\mathrm{cap}^*(\lambda)$

- This algorithm would also clearly solve the $\lambda_{\min}$ and $\lambda_{\max}$ problems, but it is more complicated, so it was useful to go through those as warm-ups first.
- We use the same idea as the $\lambda_{\max}$ algorithm, except that instead of throwing away the endpoint of the interval $[\lambda_1, \lambda_2]$ or $[\lambda_2, \lambda_3]$ not containing $\lambda_{\max}$, we keep both subintervals and recursively apply the same ideas.
- Suppose that we start on $[\lambda_1, \lambda_3]$ having optimal flow x^1 at λ_1 resulting from an upwards GGT to λ_1, and optimal flow x^3 at λ_3 resulting from a downwards GGT to λ_3, and we want to extend to $\lambda_2 \in [\lambda_1, \lambda_3]$.
- We can simultaneously continue the upwards GGT from x^1 to λ_2 to get $x^{2,\mathsf{U}}$, and the downwards GGT from x^3 to λ_2 to get $x^{2,\mathsf{D}}$.
- Then we could recurse on $[\lambda_1, \lambda_2]$ with initial flows x^1 and $x^{2,\mathsf{D}}$, and on $[\lambda_2, \lambda_3]$ with initial flows $x^{2,\mathsf{U}}$ and x^3.

A fundamental problem

- Fundamental problem: x^1 and $x^{2,\mathsf{U}}$ are part of the *same* upwards GGT (similarly, x^3 and $x^{2,\mathsf{D}}$ are part of the *same* downwards GGT).

A fundamental problem

- Fundamental problem: x^1 and $x^{2,\mathsf{U}}$ are part of the *same* upwards GGT (similarly, x^3 and $x^{2,\mathsf{D}}$ are part of the *same* downwards GGT).
- When we next do, e.g., $\lambda_{1.5} \in [\lambda_1, \lambda_2]$, we'll have to re-do the upwards GGT from x^1, and so we'd have to re-do GGT upwards work from λ_1. GGT is strong, but it can't buy us re-doing upwards work from λ_1 $O(n)$ times for free.

A fundamental problem

- Fundamental problem: x^1 and $x^{2,\mathsf{U}}$ are part of the *same* upwards GGT (similarly, x^3 and $x^{2,\mathsf{D}}$ are part of the *same* downwards GGT).
- When we next do, e.g., $\lambda_{1.5} \in [\lambda_1, \lambda_2]$, we'll have to re-do the upwards GGT from x^1, and so we'd have to re-do GGT upwards work from λ_1. GGT is strong, but it can't buy us re-doing upwards work from λ_1 $O(n)$ times for free.
- We have to either get a new x^1 from scratch, or a new $x^{2,\mathsf{D}}$ from scratch for $[\lambda_1, \lambda_2]$ (and similarly either a new x^3 from scratch, or a new $x^{2,\mathsf{U}}$ from scratch for $[\lambda_2, \lambda_3]$).

A fundamental problem

- Fundamental problem: x^1 and $x^{2,\mathsf{U}}$ are part of the *same* upwards GGT (similarly, x^3 and $x^{2,\mathsf{D}}$ are part of the *same* downwards GGT).
- When we next do, e.g., $\lambda_{1.5} \in [\lambda_1, \lambda_2]$, we'll have to re-do the upwards GGT from x^1, and so we'd have to re-do GGT upwards work from λ_1. GGT is strong, but it can't buy us re-doing upwards work from λ_1 $O(n)$ times for free.
- We have to either get a new x^1 from scratch, or a new $x^{2,\mathsf{D}}$ from scratch for $[\lambda_1, \lambda_2]$ (and similarly either a new x^3 from scratch, or a new $x^{2,\mathsf{U}}$ from scratch for $[\lambda_2, \lambda_3]$).
- But then we'd be computing $O(n)$ flows from scratch, and GGT would not really save us any time. :-(

A fundamental problem

- Fundamental problem: x^1 and $x^{2,\mathsf{U}}$ are part of the *same* upwards GGT (similarly, x^3 and $x^{2,\mathsf{D}}$ are part of the *same* downwards GGT).
- When we next do, e.g., $\lambda_{1.5} \in [\lambda_1, \lambda_2]$, we'll have to re-do the upwards GGT from x^1, and so we'd have to re-do GGT upwards work from λ_1. GGT is strong, but it can't buy us re-doing upwards work from λ_1 $O(n)$ times for free.
- We have to either get a new x^1 from scratch, or a new $x^{2,\mathsf{D}}$ from scratch for $[\lambda_1, \lambda_2]$ (and similarly either a new x^3 from scratch, or a new $x^{2,\mathsf{U}}$ from scratch for $[\lambda_2, \lambda_3]$).
- But then we'd be computing $O(n)$ flows from scratch, and GGT would not really save us any time. :-(
- Bottom line: we need four flows for $[\lambda_1, \lambda_2]$ and $[\lambda_2, \lambda_3]$, but we can legitimately carry over only two of the flows via GGT, so we need to compute two flows from scratch.

Contraction saves the day

- Recall that λ_2 has a maximal min cut $S_{\max}(\lambda_2)$, and that there is always a min cut $S_{2.5}$ for $\lambda_{2.5} \in [\lambda_2, \lambda_3]$ with $S_{\max}(\lambda_2) \subseteq S_{2.5}$; similarly, λ_2 has a minimal min cut $S_{\min}(\lambda_2)$ (s.t. $S_{\min}(\lambda_2) \subseteq S_{\max}(\lambda_2)$), and there is always a min cut $S_{1.5}$ for $\lambda_{1.5} \in [\lambda_1, \lambda_2]$ with $S_{1.5} \subseteq S_{\min}(\lambda_2)$.

Contraction saves the day

- Recall that λ_2 has a maximal min cut $S_{\max}(\lambda_2)$, and that there is always a min cut $S_{2.5}$ for $\lambda_{2.5} \in [\lambda_2, \lambda_3]$ with $S_{\max}(\lambda_2) \subseteq S_{2.5}$; similarly, λ_2 has a minimal min cut $S_{\min}(\lambda_2)$ (s.t. $S_{\min}(\lambda_2) \subseteq S_{\max}(\lambda_2)$), and there is always a min cut $S_{1.5}$ for $\lambda_{1.5} \in [\lambda_1, \lambda_2]$ with $S_{1.5} \subseteq S_{\min}(\lambda_2)$.
- Thus w.l.o.g. when processing $\lambda_{2.5}$ we can throw away $S_{\max}(\lambda_2)$.

Contraction saves the day

- Recall that λ_2 has a maximal min cut $S_{\max}(\lambda_2)$, and that there is always a min cut $S_{2.5}$ for $\lambda_{2.5} \in [\lambda_2, \lambda_3]$ with $S_{\max}(\lambda_2) \subseteq S_{2.5}$; similarly, λ_2 has a minimal min cut $S_{\min}(\lambda_2)$ (s.t. $S_{\min}(\lambda_2) \subseteq S_{\max}(\lambda_2)$), and there is always a min cut $S_{1.5}$ for $\lambda_{1.5} \in [\lambda_1, \lambda_2]$ with $S_{1.5} \subseteq S_{\min}(\lambda_2)$.
- Thus w.l.o.g. when processing $\lambda_{2.5}$ we can throw away $S_{\max}(\lambda_2)$.
- Do this via contraction: Replace $S_{\max}(\lambda_2)$ by a new node s', and for each $i \notin S_{\max}(\lambda_2)$, put $u'_{s'i} = u(S_{\max}(\lambda_2), i)$; call this $G(S_{\max}(\lambda_2))$.

Contraction saves the day

- Recall that λ_2 has a maximal min cut $S_{\max}(\lambda_2)$, and that there is always a min cut $S_{2.5}$ for $\lambda_{2.5} \in [\lambda_2, \lambda_3]$ with $S_{\max}(\lambda_2) \subseteq S_{2.5}$; similarly, λ_2 has a minimal min cut $S_{\min}(\lambda_2)$ (s.t. $S_{\min}(\lambda_2) \subseteq S_{\max}(\lambda_2)$), and there is always a min cut $S_{1.5}$ for $\lambda_{1.5} \in [\lambda_1, \lambda_2]$ with $S_{1.5} \subseteq S_{\min}(\lambda_2)$.
- Thus w.l.o.g. when processing $\lambda_{2.5}$ we can throw away $S_{\max}(\lambda_2)$.
- Do this via contraction: Replace $S_{\max}(\lambda_2)$ by a new node s', and for each $i \notin S_{\max}(\lambda_2)$, put $u'_{s'i} = u(S_{\max}(\lambda_2), i)$; call this $G(S_{\max}(\lambda_2))$.
- Similarly for processing $\lambda_{1.5}$ we contract $\bar{S}_{\min}(\lambda_2)$ into t'; call this $G(\bar{S}_{\min}(\lambda_2))$.

Contraction saves the day

- Recall that λ_2 has a maximal min cut $S_{\max}(\lambda_2)$, and that there is always a min cut $S_{2.5}$ for $\lambda_{2.5} \in [\lambda_2, \lambda_3]$ with $S_{\max}(\lambda_2) \subseteq S_{2.5}$; similarly, λ_2 has a minimal min cut $S_{\min}(\lambda_2)$ (s.t. $S_{\min}(\lambda_2) \subseteq S_{\max}(\lambda_2)$), and there is always a min cut $S_{1.5}$ for $\lambda_{1.5} \in [\lambda_1, \lambda_2]$ with $S_{1.5} \subseteq S_{\min}(\lambda_2)$.
- Thus w.l.o.g. when processing $\lambda_{2.5}$ we can throw away $S_{\max}(\lambda_2)$.
- Do this via contraction: Replace $S_{\max}(\lambda_2)$ by a new node s', and for each $i \notin S_{\max}(\lambda_2)$, put $u'_{s'i} = u(S_{\max}(\lambda_2), i)$; call this $G(S_{\max}(\lambda_2))$.
- Similarly for processing $\lambda_{1.5}$ we contract $\bar{S}_{\min}(\lambda_2)$ into t'; call this $G(\bar{S}_{\min}(\lambda_2))$.
- One contraction costs $O(m)$, and it's easy to "project" e.g. x^3 onto $G(S_{\max}(\lambda_2))$.

Contraction saves the day

- Recall that λ_2 has a maximal min cut $S_{\max}(\lambda_2)$, and that there is always a min cut $S_{2.5}$ for $\lambda_{2.5} \in [\lambda_2, \lambda_3]$ with $S_{\max}(\lambda_2) \subseteq S_{2.5}$; similarly, λ_2 has a minimal min cut $S_{\min}(\lambda_2)$ (s.t. $S_{\min}(\lambda_2) \subseteq S_{\max}(\lambda_2)$), and there is always a min cut $S_{1.5}$ for $\lambda_{1.5} \in [\lambda_1, \lambda_2]$ with $S_{1.5} \subseteq S_{\min}(\lambda_2)$.
- Thus w.l.o.g. when processing $\lambda_{2.5}$ we can throw away $S_{\max}(\lambda_2)$.
- Do this via contraction: Replace $S_{\max}(\lambda_2)$ by a new node s', and for each $i \notin S_{\max}(\lambda_2)$, put $u'_{s'i} = u(S_{\max}(\lambda_2), i)$; call this $G(S_{\max}(\lambda_2))$.
- Similarly for processing $\lambda_{1.5}$ we contract $\bar{S}_{\min}(\lambda_2)$ into t'; call this $G(\bar{S}_{\min}(\lambda_2))$.
- One contraction costs $O(m)$, and it's easy to "project" e.g. x^3 onto $G(S_{\max}(\lambda_2))$.
- Ideally, when we re-compute some flows from scratch, we can use contraction to compute them on smaller and smaller networks.

Still not so easy

- If we are unlucky, it might happen that all the $S_{\max}(\lambda_2)$'s equal $\{s\}$, and so when we "contract" $S_{\max}(\lambda_2)$ the network doesn't really shrink, and contraction doesn't help.

Still not so easy

- If we are unlucky, it might happen that all the $S_{\max}(\lambda_2)$'s equal $\{s\}$, and so when we "contract" $S_{\max}(\lambda_2)$ the network doesn't really shrink, and contraction doesn't help.
- But if $S_{\max}(\lambda_2) = \{s\}$, we also have that $S_{\min}(\lambda_2) = \{s\}$, and so the other contraction (of $\bar{S}_{\min}(\lambda_2)$ into t') will produce a *really* little network.

Still not so easy

- If we are unlucky, it might happen that all the $S_{\max}(\lambda_2)$'s equal $\{s\}$, and so when we "contract" $S_{\max}(\lambda_2)$ the network doesn't really shrink, and contraction doesn't help.
- But if $S_{\max}(\lambda_2) = \{s\}$, we also have that $S_{\min}(\lambda_2) = \{s\}$, and so the other contraction (of $\bar{S}_{\min}(\lambda_2)$ into t') will produce a *really* little network.
- We get a choice of which flow we want to re-start, so let's choose the one whose contracted network is smaller.

Still not so easy

- If we are unlucky, it might happen that all the $S_{\max}(\lambda_2)$'s equal $\{s\}$, and so when we "contract" $S_{\max}(\lambda_2)$ the network doesn't really shrink, and contraction doesn't help.
- But if $S_{\max}(\lambda_2) = \{s\}$, we also have that $S_{\min}(\lambda_2) = \{s\}$, and so the other contraction (of $\bar{S}_{\min}(\lambda_2)$ into t') will produce a *really* little network.
- We get a choice of which flow we want to re-start, so let's choose the one whose contracted network is smaller.
- This idea is the basis of GGT's recursive subroutine SLICE.

Still not so easy

- If we are unlucky, it might happen that all the $S_{\max}(\lambda_2)$'s equal $\{s\}$, and so when we "contract" $S_{\max}(\lambda_2)$ the network doesn't really shrink, and contraction doesn't help.
- But if $S_{\max}(\lambda_2) = \{s\}$, we also have that $S_{\min}(\lambda_2) = \{s\}$, and so the other contraction (of $\bar{S}_{\min}(\lambda_2)$ into t') will produce a *really* little network.
- We get a choice of which flow we want to re-start, so let's choose the one whose contracted network is smaller.
- This idea is the basis of GGT's recursive subroutine SLICE.
  - This is a divide and conquer algorithm.

Subroutine SLICE

1. Input: interval $[\lambda_1, \lambda_3]$, optimal upwards GGT flow x^1 for λ_1, optimal downwards GGT flow x^3 for λ_3, and $\lambda_2 \in [\lambda_1, \lambda_3]$.

Subroutine SLICE

1. Input: interval $[\lambda_1, \lambda_3]$, optimal upwards GGT flow x^1 for λ_1, optimal downwards GGT flow x^3 for λ_3, and $\lambda_2 \in [\lambda_1, \lambda_3]$.
2. Simultaneously continue x^1 upwards towards λ_2 and x^3 downwards towards λ_2.

Subroutine SLICE

1. Input: interval $[\lambda_1, \lambda_3]$, optimal upwards GGT flow x^1 for λ_1, optimal downwards GGT flow x^3 for λ_3, and $\lambda_2 \in [\lambda_1, \lambda_3]$.
2. Simultaneously continue x^1 upwards towards λ_2 and x^3 downwards towards λ_2.
3. Stop as soon as the first of these two λ_2 computations finds an optimal flow x^2; say the upwards computation ends first (other case is symmetric).

Subroutine SLICE

1. Input: interval $[\lambda_1, \lambda_3]$, optimal upwards GGT flow x^1 for λ_1, optimal downwards GGT flow x^3 for λ_3, and $\lambda_2 \in [\lambda_1, \lambda_3]$.
2. Simultaneously continue x^1 upwards towards λ_2 and x^3 downwards towards λ_2.
3. Stop as soon as the first of these two λ_2 computations finds an optimal flow x^2; say the upwards computation ends first (other case is symmetric).
4. Compute $S_{\min}(\lambda_2)$, $S_{\max}(\lambda_2)$.

Subroutine SLICE

1. Input: interval $[\lambda_1, \lambda_3]$, optimal upwards GGT flow x^1 for λ_1, optimal downwards GGT flow x^3 for λ_3, and $\lambda_2 \in [\lambda_1, \lambda_3]$.
2. Simultaneously continue x^1 upwards towards λ_2 and x^3 downwards towards λ_2.
3. Stop as soon as the first of these two λ_2 computations finds an optimal flow x^2; say the upwards computation ends first (other case is symmetric).
4. Compute $S_{\min}(\lambda_2)$, $S_{\max}(\lambda_2)$.
5. If $|S_{\min}(\lambda_2)| \leq n/2$ then re-compute x^1 and $x^{2,\mathsf{D}}$ on $G(\bar{S}_{\min}(\lambda_2))$ for $[\lambda_1, \lambda_2]$ (and use the already-computed $x^{2,\mathsf{U}}$ and x^3 projected onto $G(S_{\max}(\lambda_2))$ for $[\lambda_2, \lambda_3]$).

Subroutine SLICE

1. Input: interval $[\lambda_1, \lambda_3]$, optimal upwards GGT flow x^1 for λ_1, optimal downwards GGT flow x^3 for λ_3, and $\lambda_2 \in [\lambda_1, \lambda_3]$.
2. Simultaneously continue x^1 upwards towards λ_2 and x^3 downwards towards λ_2.
3. Stop as soon as the first of these two λ_2 computations finds an optimal flow x^2; say the upwards computation ends first (other case is symmetric).
4. Compute $S_{\min}(\lambda_2)$, $S_{\max}(\lambda_2)$.
5. If $|S_{\min}(\lambda_2)| \leq n/2$ then re-compute x^1 and $x^{2,\mathsf{D}}$ on $G(\bar{S}_{\min}(\lambda_2))$ for $[\lambda_1, \lambda_2]$ (and use the already-computed $x^{2,\mathsf{U}}$ and x^3 projected onto $G(S_{\max}(\lambda_2))$ for $[\lambda_2, \lambda_3]$).
6. Else ($|S_{\min}(\lambda_2)| > n/2 \implies |\bar{S}_{\max}(\lambda_2)| \leq n/2$) let the computation of $x^{2,\mathsf{D}}$ finish, then re-compute $x^{2,\mathsf{U}}$ and x^3 on $G(S_{\max}(\lambda_2))$ for $[\lambda_2, \lambda_3]$ (and use the already-computed x^1 and $x^{2,\mathsf{D}}$ projected onto $G(\bar{S}_{\min}(\lambda_2))$ for $[\lambda_1, \lambda_2]$).

Outer algorithm for computing all breakpoints

1. Compute initial λ_1, λ_3 such that all breakpoints are in $[\lambda_1, \lambda_3]$, and find optimal flows x^1, x^3 at λ_1, λ_3.

Outer algorithm for computing all breakpoints

1. Compute initial λ_1, λ_3 such that all breakpoints are in $[\lambda_1, \lambda_3]$, and find optimal flows x^1, x^3 at λ_1, λ_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.

Outer algorithm for computing all breakpoints

1. Compute initial λ_1, λ_3 such that all breakpoints are in $[\lambda_1, \lambda_3]$, and find optimal flows x^1, x^3 at λ_1, λ_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. If $\mathsf{sl}_{\min}(\lambda_2) < \mathsf{sl}_{\max}(\lambda_2)$ then report λ_2 as a breakpoint.

Outer algorithm for computing all breakpoints

1. Compute initial λ_1, λ_3 such that all breakpoints are in $[\lambda_1, \lambda_3]$, and find optimal flows x^1, x^3 at λ_1, λ_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. If $\mathsf{sl}_{\min}(\lambda_2) < \mathsf{sl}_{\max}(\lambda_2)$ then report λ_2 as a breakpoint.
4. Call SLICE with $[\lambda_1, \lambda_3]$, x^1, x^3, and λ_2.

Outer algorithm for computing all breakpoints

1. Compute initial λ_1, λ_3 such that all breakpoints are in $[\lambda_1, \lambda_3]$, and find optimal flows x^1, x^3 at λ_1, λ_3.
2. Compute $\lambda_2 = \frac{a_3 - a_1}{b_1 - b_3}$ as before.
3. If $\mathsf{sl}_{\min}(\lambda_2) < \mathsf{sl}_{\max}(\lambda_2)$ then report λ_2 as a breakpoint.
4. Call SLICE with $[\lambda_1, \lambda_3]$, x^1, x^3, and λ_2.
5. As long as networks have at least 3 nodes, and $\mathsf{sl}_{\min}$ at the left endpoint is greater than $\mathsf{sl}_{\max}$ at the right endpoint (i.e., the interval still contains another breakpoint), recursively call the main algorithm from Step 2 for $[\lambda_1, \lambda_2]$ and $[\lambda_2, \lambda_3]$ and the flows from SLICE.

Analysis of the all-breakpoints algorithm

- It should be clear by now that the algorithm finds all breakpoints, and finishes in finite time; what's hard is to get its running time.

Analysis of the all-breakpoints algorithm

- It should be clear by now that the algorithm finds all breakpoints, and finishes in finite time; what's hard is to get its running time.
- At any give time we have two active GGTs being computed. Each GGT costs $O(mn \log(n^2/m))$ via the Goldberg-Tarjan FIFO dynamic trees version of Push-Relabel.

Analysis of the all-breakpoints algorithm

- It should be clear by now that the algorithm finds all breakpoints, and finishes in finite time; what's hard is to get its running time.
- At any give time we have two active GGTs being computed. Each GGT costs $O(mn \log(n^2/m))$ via the Goldberg-Tarjan FIFO dynamic trees version of Push-Relabel.
- There is also work in computing extra flows from scratch. Let $T(n, m)$ denote the run time of this work when the current network has n nodes and m arcs, which has this recurrence:

$$T(m,n) = \max\{T(n_1,m_1) + T(n_2,m_2) + O(n_1 m_1 \log(n_1^2/m_1))$$
$$\text{s.t. } n_1, n_2 \geq 3;\ n_1 + n_2 \leq n + 2;\ n_1 \leq n_2;$$
$$m_1, m_2 \geq 1;\ m_1 + m_2 \leq m + 1\}$$

Analysis of the all-breakpoints algorithm

- It should be clear by now that the algorithm finds all breakpoints, and finishes in finite time; what's hard is to get its running time.
- At any give time we have two active GGTs being computed. Each GGT costs $O(mn \log(n^2/m))$ via the Goldberg-Tarjan FIFO dynamic trees version of Push-Relabel.
- There is also work in computing extra flows from scratch. Let $T(n, m)$ denote the run time of this work when the current network has n nodes and m arcs, which has this recurrence:

$$\begin{aligned} T(m,n) = \max\{&T(n_1,m_1) + T(n_2,m_2) + O(n_1 m_1 \log(n_1^2/m_1)) \\ &\text{s.t. } n_1, n_2 \geq 3;\ n_1 + n_2 \leq n + 2;\ n_1 \leq n_2; \\ &m_1, m_2 \geq 1;\ m_1 + m_2 \leq m + 1\} \end{aligned}$$

- The solution is $T(m, n) = O(mn \log(n^2/m)$; yay! This is the $O(1)$ min cut time we wanted :-).

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)
  - $O(mn \log_{m/(n \log n)} n)$ King, Rao, Tarjan Push-Relabel (Babenko et al.)

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)
  - $O(mn \log_{m/(n \log n)} n)$ King, Rao, Tarjan Push-Relabel (Babenko et al.)
  - $O(n^3)$ Karzanov, Wave versions of Dinic (Martel)

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)
  - $O(mn \log_{m/(n \log n)} n)$ King, Rao, Tarjan Push-Relabel (Babenko et al.)
  - $O(n^3)$ Karzanov, Wave versions of Dinic (Martel)
  - $O(mn \log n)$ Pseudoflow Algorithm (Hochbaum)

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)
  - $O(mn \log_{m/(n \log n)} n)$ King, Rao, Tarjan Push-Relabel (Babenko et al.)
  - $O(n^3)$ Karzanov, Wave versions of Dinic (Martel)
  - $O(mn \log n)$ Pseudoflow Algorithm (Hochbaum)
- But the $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ Goldberg-Rao algorithm does not work (yet); scaling algorithms don't seem to adapt to GGT well.

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)
  - $O(mn \log_{m/(n \log n)} n)$ King, Rao, Tarjan Push-Relabel (Babenko et al.)
  - $O(n^3)$ Karzanov, Wave versions of Dinic (Martel)
  - $O(mn \log n)$ Pseudoflow Algorithm (Hochbaum)
- But the $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ Goldberg-Rao algorithm does not work (yet); scaling algorithms don't seem to adapt to GGT well.
  - But Tarjan et al. can use *any* max flow algorithm as a black box with $O(\min\{n, \log(nU)\})$ longer run time to solve PFP when all $u_{si} = \lambda$, u_{it} constant.

Notes on all-breakpoints algorithm

- This fully solves PFP in $O(1)$ min cut time, as promised.
- But pretty complicated: contraction, recursion, dynamic trees.
- Other max flow algorithms with the Algorithmic Property:
  - $O(mn \log(n^2/m))$ Max Distance Push-Relabel (Gusfield & Tardos)
  - $O(mn \log_{m/(n \log n)} n)$ King, Rao, Tarjan Push-Relabel (Babenko et al.)
  - $O(n^3)$ Karzanov, Wave versions of Dinic (Martel)
  - $O(mn \log n)$ Pseudoflow Algorithm (Hochbaum)
- But the $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ Goldberg-Rao algorithm does not work (yet); scaling algorithms don't seem to adapt to GGT well.
  - But Tarjan et al. can use *any* max flow algorithm as a black box with $O(\min\{n, \log(nU)\})$ longer run time to solve PFP when all $u_{si} = \lambda$, u_{it} constant.
- Gusfield & Martel extend to non-linear $u(\lambda)$'s with an $O(\log D)$ penalty, λ_h not given in order ...

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).
- If we have flow $\hat{x}$ and $s \to i \to k$ and $s \to j \to k$ with $\hat{x}_{si} < \hat{x}_{sj}$, if $\hat{x}_{ik} < u_{ik}$ and $\hat{x}_{jk} > 0$ we could push flow around the cycle $s \to i \to k \leftarrow j \leftarrow s$ and better "balance" $\hat{x}$.

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).
- If we have flow $\hat{x}$ and $s \to i \to k$ and $s \to j \to k$ with $\hat{x}_{si} < \hat{x}_{sj}$, if $\hat{x}_{ik} < u_{ik}$ and $\hat{x}_{jk} > 0$ we could push flow around the cycle $s \to i \to k \leftarrow j \leftarrow s$ and better "balance" $\hat{x}$.
- Tarjan et al. give an $O(mn^2 \log(nU))$ (slow) "Star Balancing" algorithm that produces a balanced flow. From this you can read off all breakpoints and min cuts (Question 7 (c)).

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).
- If we have flow $\hat{x}$ and $s \to i \to k$ and $s \to j \to k$ with $\hat{x}_{si} < \hat{x}_{sj}$, if $\hat{x}_{ik} < u_{ik}$ and $\hat{x}_{jk} > 0$ we could push flow around the cycle $s \to i \to k \leftarrow j \leftarrow s$ and better "balance" $\hat{x}$.
- Tarjan et al. give an $O(mn^2 \log(nU))$ (slow) "Star Balancing" algorithm that produces a balanced flow. From this you can read off all breakpoints and min cuts (Question 7 (c)).
- Babenko et al. tried Star Balancing on such networks, and it is significantly faster than GGT Push-Relabel.

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).
- If we have flow $\hat{x}$ and $s \to i \to k$ and $s \to j \to k$ with $\hat{x}_{si} < \hat{x}_{sj}$, if $\hat{x}_{ik} < u_{ik}$ and $\hat{x}_{jk} > 0$ we could push flow around the cycle $s \to i \to k \leftarrow j \leftarrow s$ and better "balance" $\hat{x}$.
- Tarjan et al. give an $O(mn^2 \log(nU))$ (slow) "Star Balancing" algorithm that produces a balanced flow. From this you can read off all breakpoints and min cuts (Question 7 (c)).
- Babenko et al. tried Star Balancing on such networks, and it is significantly faster than GGT Push-Relabel.
  - However, GGT Push-Relabel was faster on more general networks.

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).
- If we have flow $\hat{x}$ and $s \to i \to k$ and $s \to j \to k$ with $\hat{x}_{si} < \hat{x}_{sj}$, if $\hat{x}_{ik} < u_{ik}$ and $\hat{x}_{jk} > 0$ we could push flow around the cycle $s \to i \to k \leftarrow j \leftarrow s$ and better "balance" $\hat{x}$.
- Tarjan et al. give an $O(mn^2 \log(nU))$ (slow) "Star Balancing" algorithm that produces a balanced flow. From this you can read off all breakpoints and min cuts (Question 7 (c)).
- Babenko et al. tried Star Balancing on such networks, and it is significantly faster than GGT Push-Relabel.
  - However, GGT Push-Relabel was faster on more general networks.
  - But "simple GGT" (no concurrent GGTs) was a bit faster than full GGT, despite a theoretically worse running time.

A slower (but better?) PFP algorithm

- Consider special case of bipartite network (s, L, R, t) where $u_{sj} = \lambda \; \forall j \in L$ (Question 7).
- If we have flow $\hat{x}$ and $s \to i \to k$ and $s \to j \to k$ with $\hat{x}_{si} < \hat{x}_{sj}$, if $\hat{x}_{ik} < u_{ik}$ and $\hat{x}_{jk} > 0$ we could push flow around the cycle $s \to i \to k \leftarrow j \leftarrow s$ and better "balance" $\hat{x}$.
- Tarjan et al. give an $O(mn^2 \log(nU))$ (slow) "Star Balancing" algorithm that produces a balanced flow. From this you can read off all breakpoints and min cuts (Question 7 (c)).
- Babenko et al. tried Star Balancing on such networks, and it is significantly faster than GGT Push-Relabel.
  - However, GGT Push-Relabel was faster on more general networks.
  - But "simple GGT" (no concurrent GGTs) was a bit faster than full GGT, despite a theoretically worse running time.
- Moral: worst-case running time is not a good practical guide, and practical performance can depend on the class of instances.

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?
  - Partially answered by several; general theory?

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?
  - Partially answered by several; general theory?
  - Key idea: finding a good Flow Update.

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?
  - Partially answered by several; general theory?
  - Key idea: finding a good Flow Update.
- Extend to parametric optimization problems beyond max flow/min cut?

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?
  - Partially answered by several; general theory?
  - Key idea: finding a good Flow Update.
- Extend to parametric optimization problems beyond max flow/min cut?
  - Partially answered by Topkis, Fleischer & Iwata, Milgrom & Shannon.

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?
  - Partially answered by several; general theory?
  - Key idea: finding a good Flow Update.
- Extend to parametric optimization problems beyond max flow/min cut?
  - Partially answered by Topkis, Fleischer & Iwata, Milgrom & Shannon.
- Extend to problems with multiple parameters?

Research questions

- More general max flow models than GGT with nested min cuts (Structural Property)? E.g., parameters on arcs not at s, t.
  - Partially answered by Mc '00; Granot, Mc, Queyranne, Tardella.
- Which max flow algorithms have the Algorithmic Property?
  - Partially answered by several; general theory?
  - Key idea: finding a good Flow Update.
- Extend to parametric optimization problems beyond max flow/min cut?
  - Partially answered by Topkis, Fleischer & Iwata, Milgrom & Shannon.
- Extend to problems with multiple parameters?
  - Partially answered by Mc '00, GMcQT, but only very special cases.