Trust Is Risk: Games, Network Health and Risk Invariance Algorithms

Orfeas Stefanos Thyfronitis Litos[1], Christos Porios[2], and Dionysis Zindros[3]

[1] University of Edinburgh
[2] Imperial College London
[3] National and Kapodistrian University of Athens
o.thyfronitis@ed.ac.uk, christos.porios@imperial.ac.uk, dionyziz@di.uoa.gr

Abstract. Trust Is Risk [1] proposes a decentralized reputation system. In an effort to deepen our understanding of the dynamics of networks created by this construction and to provide useful tools that facilitate the completion of purchases, we expand our research in three directions. We firstly describe some games, leveraging approaches from Game Theory. Secondly we introduce measures that gauge the health of such networks using the concepts of connectedness and centrality. We will finally describe several algorithms that redistribute a player's direct trust in a way that a desired indirect trust towards a specific player is achieved.

1 Games

1.1 Common Setup

We propose two different kinds of games, both finite (but possibly generalizable to the infinite setting). The first consists of one sole strategy where the players do not initially know whether they will be buyers, sellers or nothing at all, this being decided in the last moment. The second game consists of three strategies: buyers, sellers and middlemen. Before delving into the details of each game, we first describe their common elements.

The general approach taken is as follows: After a game is described in detail, each player is assigned a specific strategy and a relevant utility function. All players are considered to follow their respective strategy without deviating from it, except for one player that is allowed to follow any desired strategy; her utility function however remains unchanged. If that player is proven to have an incentive to deviate from her appointed strategy, we can deduce that the given strategies and utility functions do not constitute a Nash equilibrium. If on the other hand no player has an incentive to deviate, regardless of her appointed strategy, then

we will come to the conclusion that the given strategies and utility functions do constitute a Nash equilibrium. This approach is common in game theoretic analyses, given that allowing for all players to be rational and then searching for a Nash equilibrium constitutes a practically intractable problem [2]. Another common approach employed here is that of considering only a generic product that all buyers want and all sellers have, and not a variety of different products.

A description of the structure that is common for both games follows. The game graph has a random initial configuration where every player has a random direct trust towards every other player, as well as a random capital. These values may be uniformly distributed in an interval or may follow another distribution such as the exponential, or may have a high probability of being zero. The exact distribution however is not determined at this point, as it is not yet needed. This distribution will be common knowledge to the players. All capitals and direct trusts are publicly viewable. Further constraints may be applied to each game separately. Transaction fees are not considered.

The game consists of R rounds, the number of which is common knowledge for the players. The players play simultaneously in each round and can do any of the known actions. If two actions conflict (e.g. A reduces $DTr_{A\to B}$ and B steals from $DTr_{A\to B}$ as well), then one of the two actions is chosen with equal probability (50%). To better model a player's actions and the aforementioned conflict resolution, we demand that each fund reallocation explicitly mentions the source and the destination of the funds for each of her actions. Player A decides on the values of all the following variables. This constitutes a concrete round for A.

$$\forall B, C \in \mathcal{V}, move\left(A, (A, B), (A, C)\right) \in \mathbb{R}^{+}$$
$$\forall B, C \in \mathcal{V}, B \neq A, move\left(A, (B, A), (A, C)\right) \in \mathbb{R}^{+}$$

The first argument is the player who decides, the second argument is from which direct trust to take the funds and the third is to which direct trust to deposit the funds. The first type of moves corresponds to $Add\left(\right)$ and the second to the $Steal\left(\right)$ actions.

To clarify a detail, for any $B \in \mathcal{V}$ (including A), A is not allowed to set $move\left(A, (A, B), (A, B)\right)$ to any value different than 0. This choice is made to facilitate the analysis.

There are some constraints for player's A move:

– There is no reason to be able to deposit to and withdraw from a specific direct trust in the same round. Furthermore, such a possibility

would allow for "chain reactions" in the conflict resolution phase that would add unnecessary complications. This constraint applies only to outgoing direct trusts, because incoming direct trusts cannot be increased.

$$\forall B, C, D \in \mathcal{V}, move\left(A, (A, B), (A, C)\right) \cdot move\left(A, (A, D), (A, B)\right) = 0$$

and

$$\forall B, C, D \in \mathcal{V}, move\left(A, (A, B), (A, C)\right) \cdot move\left(A, (D, A), (A, B)\right) = 0$$

– One cannot use more funds than are available from a single direct trust.

$$\forall B \in \mathcal{V}, \sum_{C \in \mathcal{V}} move\left(A, (A, B), (A, C)\right) \leq DTr_{A \to B}$$
$$\forall B \in \mathcal{V}, \sum_{C \in \mathcal{V}} move\left(A, (B, A), (A, C)\right) \leq DTr_{B \to A}$$

If two players try to change the same direct trust, then set the relevant moves of one of the two players (chosen uniformly at random) to 0.

```
resolveConflict(A, B) :
  sum1 = Σ_{C∈V} move(A, (A, B), (A, C))
  sum2 = Σ_{C∈V} move(B, (A, B), (B, C))
  if (sum1*sum2 != 0)
    choice ←$ {A, B}
    if (choice == A)
      ∀C ∈ V, move(A, (A, B), (A, C)) = 0
    else # if (choice == B)
      ∀C ∈ V, move(B, (A, B), (B, C)) = 0

resolveAllConflicts() :
  ∀A, B ∈ V
    resolveConflict(A, B)
    resolveConflict(B, A)
```

`resolveAllConflicts()` is executed after all players choose their moves for a round. In case of an adversary, the random choice is resolved in their favor.

Fig. 1.1 depicts the evolution of a two-player game. The players' choices for the first round are represented by the solitary table at the

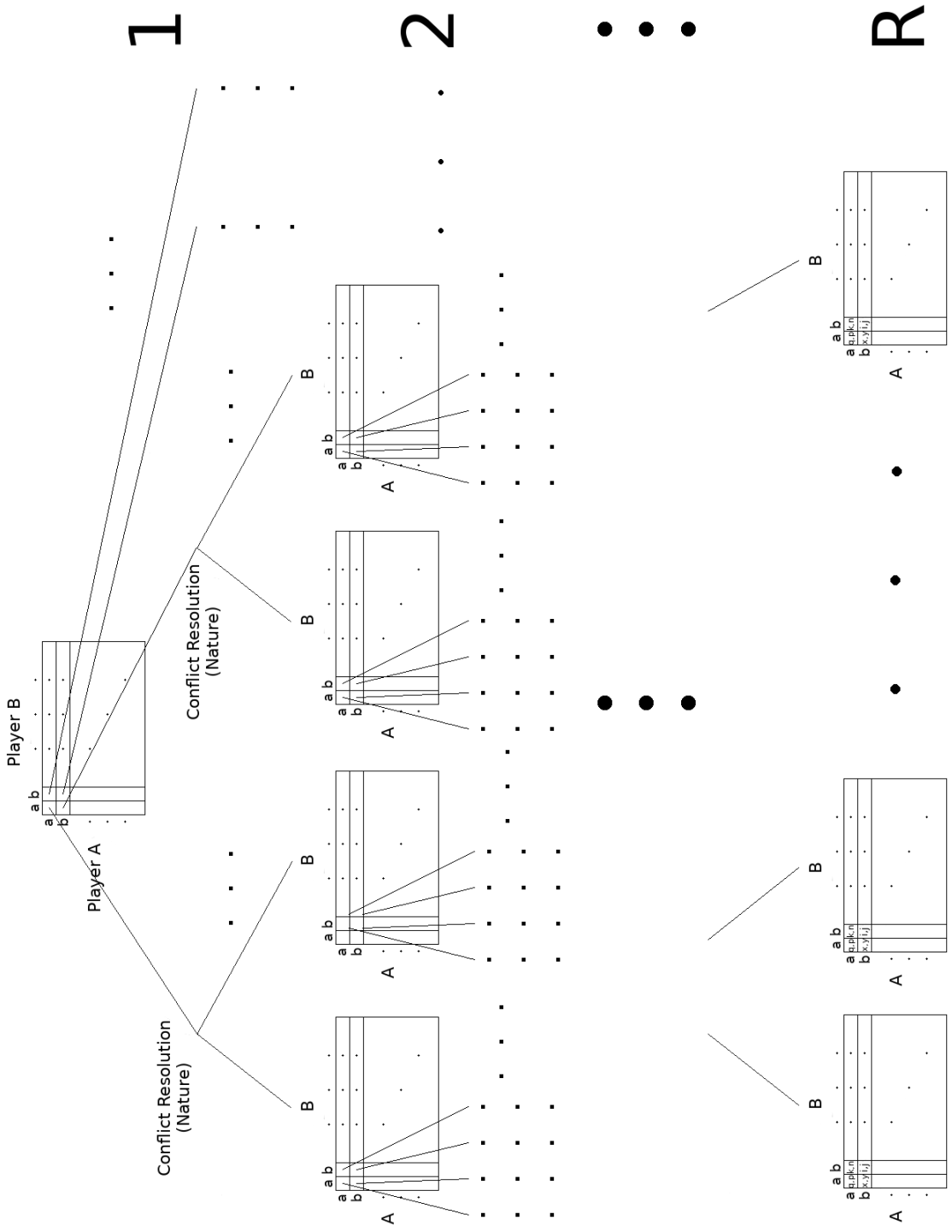

Fig. 1. The general form of the game [3]

top of the figure. More concretely, each distinct choice that player A can make is depicted as a row and likewise B's choices are represented by columns. At each intersection of choices lies a subgame. The lines departing from these intersections lead to the conflict resolution stage, which is decided by Nature. Depending on the result of the conflict resolution, a new round begins where players can choose a new course of action. All the possible states that players can be found in at the beginning of round 2 are depicted by the series of tables at the 2nd level. The game evolves in a similar fashion with alternation between players' choices and Nature's conflict resolution, until players can make their R-th choice. The keen observer can note that, instead of new departing lines, each cell of the tables of the last round (R) contains a pair of numbers which represent the players' utilities for that outcome.

Note that a concrete player's strategy consists of all the choices that she would make in every possible state that she can find herself in. Thus, even though only R choices will be made by each player, a strategy is comprised of exponentially many choices. Furthermore, if we had to consider a three-player game, we would have to replace all tables with 3-dimensional cubes and so on for more players.

We now move on to describe the individual games.

1.2 Random Roles

All players follow the same strategy, according to which each player is permitted to freely add or steal direct trust from other players. After R rounds exactly two players are selected at random (these choices follow the uniform distribution). One is dubbed seller and the other buyer. The seller offers a good that costs C, which she values at $C - l$ and the buyer values at $C + l$. The values R, C and l, as well as the uniform distribution with which the buyer and seller are chosen, are common knowledge from the beginning of the game. The exchange completes if and only if $Tr_{Buyer \to Seller} \geq C$. In this game, fig 1.1 would be augmented by an additional level at the bottom, where Nature chooses the two transacting players. There are three variants of the game, each with a different utility for the players (the first two versions have two subvariants each).

1.2.1 Hoarders

If player A is not chosen to be either buyer or seller, then her utility is equal to $Cap_{A,R}$. Intuitively players do not attach any value to having

(incoming or outgoing) direct trust at the end of the game. If the buyer and the seller do not manage to complete the exchange, the buyer's utility is $Cap_{Buyer,R}$. If on the other hand they manage to exchange the good, then the buyer's utility is $Cap_{Buyer,R}+l$. Intuitively these utilities signify that the buyer uses her preexisting capital to buy. As for the seller there exist two subvariants for her utility:

1. If the exchange is eventually not completed, the seller's utility is $Cap_{Seller,R} - l$. If on the other hand the exchange takes place, the seller's utility is $Cap_{Seller,R}$. Intuitively, the seller is first obliged to buy the good from the environment at the cost of C.
2. If the exchange is eventually not completed, the seller's utility is $Cap_{Seller,R} + C - l$. If the exchange takes place, the seller's utility is $Cap_{Seller,R} + C$. Intuitively, the seller is handed the good for free by the environment.

1.2.2 Sharers

If player A is not chosen to be either buyer or seller, then her utility is equal to

$$\sum_{\substack{B\in\mathcal{V}\\B\neq A}} \frac{DTr_{A\to B,R} + DTr_{B\to A,R}}{2} + Cap_{A,R} \ .$$

Intuitively, players attach equal value to all the funds they can directly spend, regardless of whether others can spend them as well. If the buyer and the seller do not manage to complete the exchange, the buyer's utility is

$$\sum_{\substack{B\in\mathcal{V}\\B\neq Buyer}} \frac{DTr_{Buyer\to B,R} + DTr_{B\to Buyer,R}}{2} + Cap_{Buyer,R} \ .$$

If on the other hand they manage to exchange the good, then the buyer's utility is

$$\sum_{\substack{B\in\mathcal{V}\\B\neq Buyer}} \frac{DTr_{Buyer\to B,R} + DTr_{B\to Buyer,R}}{2} + Cap_{Buyer,R} + l \ .$$

Intuitively these utilities signify that the buyer uses her preexisting accessible funds to buy. As for the seller there exist two subvariants for her utility:

1. If the exchange is not completed, the seller's utility is

$$\sum_{\substack{B\in\mathcal{V}\\B\neq Seller}} \frac{DTr_{Seller\to B,R}+DTr_{B\to Seller,R}}{2}+Cap_{Seller,R}-l \ .$$

   If the exchange takes place, the seller's utility is

$$\sum_{\substack{B\in\mathcal{V}\\B\neq Seller}} \frac{DTr_{Seller\to B,R}+DTr_{B\to Seller,R}}{2}+Cap_{Seller,R} \ .$$

   Intuitively, the seller is first obliged to buy the good from the environment at the cost of C.
2. If the exchange is not completed, the seller's utility is

$$\sum_{\substack{B\in\mathcal{V}\\B\neq Seller}} \frac{DTr_{Seller\to B,R}+DTr_{B\to Seller,R}}{2}+Cap_{Seller,R}+C-l \ .$$

   If the exchange takes place, the seller's utility is

$$\sum_{\substack{B\in\mathcal{V}\\B\neq Seller}} \frac{DTr_{Seller\to B,R}+DTr_{B\to Seller,R}}{2}+Cap_{Seller,R}+C \ .$$

   Intuitively, the seller is handed the good for free by the environment.

1.2.3 Materialists

If player A is not chosen to be either buyer or seller, then her utility is 0. If the buyer and the seller do not manage to complete the exchange, their utility is 0 as well. If on the other hand they manage to exchange the good, then the utility is l for both of them. Intuitively these utilities signify that in this game there is gain only for those who exchange goods and the gain is exactly the difference between the objective value and the subjective value that the relevant parties perceive.

1.3 Marketplace

In the second game, buyers only desire to buy as many products as possible and do not have incoming trust. Sellers only desire to sell as many products as possible in the highest possible price and do not have outgoing trust. Middlemen desire to accumulate capital and are allowed to have both incoming and outgoing trust. Their utility stems only from in-game factors. A more thorough description follows.

1.3.1 Buyers

Buyers initially are provided only with outgoing direct trust towards sellers and middlemen, no capital or incoming direct trust. They cannot reallocate their outgoing direct trust, only complete purchases of the product from sellers that are trustworthy enough.

Buyers always want to buy the cheapest products provided. To pay for a product, they use the linear program defined in subsection 3.4 to acquire the funds needed and directly entrust funds equal to the cost of the product to the seller. The rest of the funds are kept as capital. Sellers are supposed to complete their part of the exchange within a constant amount of rounds, r. If the product has arrived after r rounds, during the turn following the arrival of the product the remaining funds are reallocated to the players from whom they were taken in proportional fashion. The way they are reallocated ensures that

$$\forall B \in \mathcal{S}_1, \frac{DTr_{Buyer \to B,i}}{\sum\limits_{C \in \mathcal{S}_1} DTr_{Buyer \to C,i}} = \frac{DTr_{Buyer \to B,i+r}}{\sum\limits_{C \in \mathcal{S}_1} DTr_{Buyer \to C,i+r}} \ ,$$

where $\mathcal{S}_1$ is the set of players from whom the buyer reduced her direct trust to initiate the purchase (this set may also contain the seller).

If the product does not arrive after r rounds, the direct trust to the seller is withdrawn (if it is still available) and all the funds involved (both the price and the surplus funds removed because of the linear program) are reallocated as direct trusts towards the rest of the players in a proportional fashion. The way they are reallocated ensures that

$$\forall B \in \mathcal{S}_2, \frac{DTr_{Buyer \to B,i}}{\sum\limits_{C \in \mathcal{S}_2} DTr_{Buyer \to C,i}} = \frac{DTr_{Buyer \to B,i+r}}{\sum\limits_{C \in \mathcal{S}_2} DTr_{Buyer \to C,i+r}} \ ,$$

where $\mathcal{S}_2 = (\mathcal{V} \setminus \mathcal{S}_1) \setminus \{Buyer, Seller\}$.

The buyers' utilities are straightforward: They are equal to the amount of products they managed to buy throughout the game. This explains why buyers always prefer cheaper products: This way they are able to save funds so that they are hopefully able to buy more products.

1.3.2 Sellers

Sellers initially are provided only with incoming direct trust from buyers and middlemen, no capital or outgoing direct trust. They also

possess a limited supply of products; the quantity of the products each seller initially has is decided randomly by the environment, much like the direct trusts. These products are useless to the sellers, they only want to sell them. The amount of products each seller has is common knowledge.

During each round, each seller decides how many products she will make available for purchase during the next round. She also sets the price, which is common for all products. Obviously a seller cannot offer more products than she owns. Furthermore, if a buyer initiated a transaction during the previous round, the buyer must ship the product (reducing the amount of owned products by one). The seller may choose to convert any amount of incoming direct trust that she obtained through selling products into capital. The seller is not allowed to convert into capital any incoming direct trust that did not stem from a successful exchange.

A seller's utility is the total amount of incoming direct trust and capital at the end of the game that stems solely from successfully completed purchases.

1.3.3 Middlemen

Middlemen initially have incoming direct trust from buyers and other middlemen, outgoing direct trust to sellers and other middlemen and some capital. They do not own or desire any products, they simply want to maximize the amount of capital they own at the end of the game.

Middlemen have a very permissive strategy: They are allowed any combination of moves that do not violate the basic constraints described in the first section.

The utility of a middleman is simply the capital the middleman possesses at the end of the game.

2 Network Health

2.1 Degree of Connectedness

We would like to have a measure of how well connected is the graph. A network with a lot of disconnected cliques should score lower than a loosely connected network. This measure can be expressed as the expected

indirect trust, defined incrementally as follows:

$$Tr(A) = \sum_{B \in \mathcal{V} \setminus \{A\}} Tr_{A \to B} \qquad \text{(Total indirect trust for player } A\text{)}$$

$$ETr\,(A) = \frac{Tr(A)}{|\mathcal{V} \setminus \{A\}|} = \frac{Tr(A)}{|\mathcal{V}| - 1} \qquad \text{(Expected indirect trust for player } A\text{)}$$

$$ETr = \frac{1}{|\mathcal{V}|} \sum_{A \in \mathcal{V}} ETr\,(A) \qquad \text{(Expected indirect trust)}$$

2.2 Centrality

Another important measure is the network centrality. Intuitively, a network that resembles a mesh is preferable over a network with a central hub that acts as an intermediary between all other pairs of nodes. This is because in the latter case Trust Is Risk does not offer any realistic improvement over the existing centralized marketplaces model (e.g. ebay). Nevertheless, the expected indirect trust measure is not always able to distinguish between the two. We thus propose here some different measures to that end.

2.2.1 Degree Centrality

One possible approach is the degree centrality [4], that can be broken down as in-degree and out-degree centrality. We first define the node in-degree centrality.

$$C^d_{in}\,(A) = \sum_{B \in \mathcal{V} \setminus \{A\}} DTr_{B \to A}{}^{4} \qquad \text{(Node in-degree centrality)}$$

Let A^* be the player with the maximum centrality: $A^* = \underset{A \in \mathcal{V}}{\operatorname{argmax}}\, C^d_{in}\,(A)$. The network in-degree centrality is defined as:

$$C^d_{in} = \sum_{A \in \mathcal{V}} \left(C^d_{in}\,(A^*) - C^d_{in}\,(A) \right) \qquad \text{(Network in-degree centrality)}$$

Similarly, for the out-degree centrality we have:

$$C^d_{out}\,(A) = \sum_{B \in \mathcal{V} \setminus \{A\}} DTr_{A \to B}{}^{4} \qquad \text{(Node out-degree centrality)}$$

[4] Maybe indirect trust is more intuitive here than direct trust.

Let $A^* = \underset{A \in \mathcal{V}}{\operatorname{argmax}} C^d_{out}(A)$. The network out-degree centrality is:

$$C^d_{out} = \sum_{A \in \mathcal{V}} \left(C^d_{out}(A^*) - C^d_{out}(A) \right) \quad \text{(Network out-degree centrality)}$$

A problem of these centrality measures is that their unit is the currency used (i.e. Bitcoin) and thus may not always have an intuitive meaning. We would thus like to have a measure of *centralization* that has no units and can take values in the interval $[0, 1]$, with 0 corresponding to a network of no centralization (all nodes are equal, e.g. cycle) and 1 to a network with the maximum centralization possible (there is one central vital node for all, e.g. star). A centralization measure that achieves this target is proposed in [4]. Here we use the following modified form. For a graph $\mathcal{G}$, the in- and out-centralization are defined as:

$$Cn^d_{in} = \frac{C^d_{in}}{\max C^d_{in}} \quad \text{(in-degree centralization)}$$

$$Cn^d_{out} = \frac{C^d_{out}}{\max C^d_{out}} \quad \text{(out-degree centralization)} \ ,$$

where $\max C^d_{in}$ is defined as the maximum in-degree centrality over all networks with the same number of nodes for which the maximum direct[5] trust is equal to the maximum direct[5] trust of $\mathcal{G}$; $\max C^d_{out}$ is defined equivalently.

2.2.2 maxFlow Centrality

An alternative measure of centrality for a player $A \in \mathcal{V}$ of a network $\mathcal{G}$ can be defined as the impact that the removal of A would have on the indirect trust between the rest of the players. More specifically, let $\mathcal{G}' = \mathcal{G} \setminus \{A\}$. Then it is:

$$C^{mF}(A) = \sum_{B,C \in \mathcal{V}'} \left(Tr_{\mathcal{G},B \to C} - Tr_{\mathcal{G}',B \to C} \right) \quad \text{(Node maxFlow centrality)} \ .$$

We can now follow the same steps as previously for the relevant network definitions. Let $A^* = \underset{A \in \mathcal{V}}{\operatorname{argmax}} C^{mF}(A)$. Then the network maxFlow centrality is:

$$C^{mF} = \sum_{A \in \mathcal{V}} \left(C^{mF}(A^*) - C^{mF}(A) \right) \quad \text{(Network maxFlow centrality)}$$

[5] Maybe indirect trust is more intuitive here than direct trust.

and the centralization:

$$Cn^{mF} = \frac{C^{mF}}{\max C^{mF}} \qquad \text{(Network maxFlow centralization)} \ ,$$

where $\max C^{mF}$ is the maximum centrality over all networks with the same number of nodes for which the maximum direct[6] trust is equal to the maximum direct[6] trust of $\mathcal{G}$.

3 Risk Invariance Algorithms

We will now focus our discussion on the act of a purchase with invariant risk. Suppose the game is currently depicted by graph $\mathcal{G}$ and after the necessary preparations for paying with invariant risk, the game corresponds to graph $\mathcal{G}'$. According to the Risk Invariance theorem [1], if *Alice* wants to buy something that costs V from the vendor *Bob*, she should first find a new distribution of her outgoing direct trust such that $maxFlow_{\mathcal{G}'}(Alice, Bob) = maxFlow_{\mathcal{G}}(Alice, Bob) - V$ and then directly entrust V to *Bob*. The theorem then ensures that the initial risk (i.e indirect trust) from *Alice* to *Bob* is equal to the final. In the current section we discuss several algorithms that calculate possible new distributions for *Alice*'s outgoing direct trust. The direct trust will be referred to as capacity, because of the maximum flow context.

Let $A \in \mathcal{V}$ source, $B \in \mathcal{V}$ sink. We use the following notation:

$$C \text{ capacity configuration with } c_{Av} = DTr_{\mathcal{G},A\to v}$$
$$C' \text{ capacity configuration with } c'_{Av} = DTr_{\mathcal{G}',A\to v}$$

We will use the following notation for clarity:

$$\begin{cases} X = MaxFlow_{\mathcal{G}}(A,B) \\ X' = MaxFlow_{\mathcal{G}'}(A,B) \end{cases} \ , \text{ for some } MaxFlow \text{ execution}$$

$$\begin{cases} F = maxFlow_{\mathcal{G}}(A,B) \\ F' = maxFlow_{\mathcal{G}'}(A,B) \end{cases} \ .$$

Any subscripts or superscripts applied to X and F refer to the capacity configuration with the exact same subscripts and superscripts.

Furthermore, we suppose an arbitrary ordering of the members of $N^+(A)$. We set $n = |N^+(A)|$. Thus

$$N^+(A) = \{v_1, ..., v_n\} \ .$$

[6] Maybe indirect trust is more intuitive here than direct trust.

We use these subscripts to refer to the respective capacities and flows. Thus

$$\begin{matrix} x_i = x_{Av_i} \\ c_i = c_{Av_i} \end{matrix} \text{ , where } i \in [n] \ .$$

There is no difference in the capacities that are not outgoing from A between the graphs $\mathcal{G}$ and $\mathcal{G}'$, since A can only modify the capacities outgoing from herself.

3.1 Preliminaries

We now move on to some preliminary definitions and results.

Definition 1 (Trust Reduction).
Trust Reduction on neighbour i is defined as $\delta_i = c_i - c'_i$.
Flow Reduction on neigbour i is defined as $\Delta_i = x_i - c'_i$.
We will also use the standard notation for 1-norm and ∞-norm:

$$||\delta_i||_1 = \sum_{i=1}^{n} \delta_i$$
$$||\delta_i||_\infty = \max_{1 \leq i \leq n} \delta_i$$

Theorem 1 (Saturation Theorem).

$$(\forall i \in [n], c'_i \leq x_i) \Rightarrow (\forall i \in [n], x'_i = c'_i)$$

Proof Sketch. A new valid flow Y for turn $j-1$ can be created starting with X by reducing the flows along $(v_i, ..., B)$ paths until all (A, v_i) flows can be reduced to c'_i. The flow Y is then valid for the capacities of turn j and, since all capacities c'_i are saturated, Y is furthermore a maximum flow in $\mathcal{G}_j$. □

Corollary 1 (Trust Transfer Corollary).
Let $V \in [0, F]$. We create a C' where

$$\forall i \in [n], c'_i \leq x_i \text{ and}$$
$$\sum_{i=1}^{n} c'_i = F - V \ .$$

It then holds that $F' = F - V$.

Proof. From the Saturation theorem (1) we can see that $x'_i = c'_i$. It holds that

$$F' = \sum_{i=1}^{n} x'_i = \sum_{i=1}^{n} c'_i = F - V \ .$$

□

Theorem 2 (Invariable Trust Reduction with Naive Algorithms).

If $\forall i \in [n]$ it is $c'_i \leq x_i$, then $||\delta_i||_1$ and $||\Delta_i||_1$ are independent of x'_i, c'_i.

Proof. Since $\forall i \in [n], c'_i \leq x_i$, by applying the Saturation theorem (1) we see that $x'_i = c'_i$, thus $\delta_i = c_i - x'_i$ and $\Delta_i = x_i - x'_i$. We know that $\sum_{i=1}^{n} x'_i = F - V$, so we have

$$||\delta_i||_1 = \sum_{i=1}^{n} \delta_i = \sum_{i=1}^{n} (c_i - x'_i) = \sum_{i=1}^{n} c_i - F + V \text{ and}$$
$$||\Delta_i||_1 = \sum_{i=1}^{n} \Delta_i = \sum_{i=1}^{n} (x_i - x'_i) = \sum_{i=1}^{n} x_i - F + V \ .$$

thus $||\delta_i||_1, ||\Delta_i||_1$ are independent from x'_i and c'_i. □

Until now $MaxFlow$ has been viewed purely as an algorithm. This algorithm is not guaranteed to always return the same flow configuration when executed muliple times on the same graph. However, the corresponding flow value, $maxFlow$, is always the same. Thus $maxFlow$ can be also viewed as a function from a matrix of capacities to a non-negative real number. Under this perspective, we prove the following theorem. Let $\mathcal{C}$ be the family of all capacity matrices $C = [c_{vw}]_{\mathcal{V}\times\mathcal{V}}$.

Theorem 3 (maxFlow Continuity).
Suppose a directed weighted graph with a source A, a sink B and a capacity configuration $C \in \mathcal{C}$. Let also $p \in \mathbb{N} \cup \{\infty\}$. The function $maxFlow : \mathcal{C} \to \mathbb{R}^+$ is continuous with respect to the $||\cdot||_p$ norm.

Proof Sketch. An infinitesimal modification to any capacity leads to a no more than infinitesimal modification to the total $maxFlow$. □

3.2 Naive Algorithms

Here we show three naive algorithms for calculating the new capacities that make use of the Trust Transfer corollary (1). To prove the correctness of the algorithms, it suffices to prove that

$$\forall i \in [n], c'_i \leq x_i \text{ and} \tag{1}$$

$$\sum_{i=1}^{n} c'_i = F - V \ . \tag{2}$$

Due to the nature of these algorithms, the only inputs necessary are the initial flows x_i, the number of A's neighbours n and the desired flow reduction V. Proofs of correctness and complexity can be found in the Appendix.

```
First Come First Served Trust Transfer
Input : old flows x_i, value V, length n
Output : new capacities c'_i
1  fcfs((x_i), V) :
2    F_cur = F = sum_{i=1}^{n} x_i
3    if (F < V)
4      return(⊥)
5    for (i = 1 to n)
6      c'_i = x_i
7    i = 1
8    while (F_cur > F - V)
9      reduce = min(x_i, F_cur - (F - V))
10     F_cur = F_cur - reduce
11     c'_i = x_i - reduce
12     i += 1
13   return(⋃_{i=1}^{n} {c'_i})
```

The previous algorithm nullifies all outgoing trust to one player after the other in the order they are given in the input, until the desired indirect trust is achieved. The complexity of this algorithm is $O(n)$.

```
Absolute Equality Trust Transfer ( ||Δ_i||_∞ minimizer)
Input : old flows x_i, value V, length n
Output : new capacities c'_i
1  abs((x_i), V) :
```

```
2   F_cur = F = Σ_{i=1}^{n} x_i
3   if (F < V)
4     return(⊥)
5   X = preprocess(x_i)
6   empty = 0
7   reduction = 0
8   while (F_cur > F - V)
9     (i, X) = popMin(X)
10    F_prov = F_cur - (n - empty)*(x_i - reduction)
11    if (F_prov > F - V)
12      reduction = x_i
13      empty += 1
14      F_cur = F_prov
15    else
16      aux = reduction
17      reduction += (F_cur - (F - V)) / (n - empty)
18      F_cur -= (n - empty)*(reduction - aux)
19      #lines 16 & 18 can be replaced by break. In this
20      #case, the loop (line 8) can become while (TRUE).
21  for (i = 1 to n)
22    c'_i = max(0, x_i - reduction)
23  return(⋃_{i=1}^{n} {c'_i})
```

This algorithm finds a `reduction` value such that the configuration $\forall i \in [n], c'_i = \max(0, x_i - \texttt{reduction})$ achieves the desired flow.

The function `preprocess`(x_i) returns a data structure `X` containing the set of flows (x_i), such that the corresponding function `popMin(X)` is able to repeatedly return a tuple consisiting of the index of the minimum element and a new data structure missing exactly the minimum element. Examples of such pairs of functions are:

$$\begin{cases} \texttt{preprocess = quickSort} \\ \texttt{popMin = } (x_1, \texttt{ X} \setminus \{x_1\}) \end{cases} \quad \text{and}$$

$$\begin{cases} \texttt{preprocess = FibonacciHeap} \\ \texttt{popMin = (find-min(X),delete-min(X))} \end{cases}.$$

In the general case, the complexity of the algorithm `abs` is proven to be $O(preprocess) + O(n)\, O(popMin)$. In both specific cases, the complexity is $O(n \log n)$. This algorithm also minimizes the $||\Delta_i||_\infty$ norm for the

specific set of old flows x_i given as input. A proof of this fact can be found in the Appendix.

```
Proportional Equality Trust Transfer
Input : old flows x_i, value V, length n
Output : new capacities c'_i
prop((x_i), V) :
  F = sum_{i=1}^{n} x_i
  if (F < V)
    return(⊥)
  for (i = 1 to n)
    c'_i = x_i - V/F * x_i
  return(⋃_{i=1}^{n} {c'_i})
```

This algorithm reduces all outgoing direct trust proportionally in such a way that the desired flow is achieved. The complexity of this algorithm is proven to be $O(n)$.

3.3 $||\delta_i||_\infty$ minimizer

Naive algorithms result in $c'_i \leq x_i$, thus according to the Invariability Theorem (2), $||\delta_i||_1$ is invariable for any of the possible results C' of these algorithms and the resulting norms are not necessarily the minimum possible. The following algorithm concentrates on finding a configuration C' that achieves $F' = F - V$ while minimizing the $||\delta_i||_\infty$ norm.

```
||δ_i||_∞ minimizer
Input : old capacities c_i, source A, sink B, value V,
    length n, ε_1, ε_2
Output : new capacities c'_i
dinfmin(c_i, A, B, V, ε_1, ε_2) :
  F = maxFlow(A, B, C)
  if (F < V)
    return(⊥)
  if ((ε_1 < 0) or (ε_2 < 0) or (F - V - ε_1 < 0) or
     (F - V + ε_2 > F))
    return(⊥)
  δ_max = max(C)
  δ* = BinSearch(0, δ_max, F - V, n, A, B, C, ε_1, ε_2)
  for (i = 1 to n)
```

```
    c'_i = max(0, c_i - δ*)
  return(⋃_{i=1}^{n} {c'_i})
```

Since trust should be considered as a continuous unit and binary search bisects the interval containing the solution on each recursive call, inclusion of the ϵ-parameters in `BinSearch` is necessary for the algorithm to complete in a finite number of steps.

```
Binary Search Function for ||δ_i||_∞ minimizer
Input : bot, top, F', n, source A, sink B, capacities C,
                                                  ε_1, ε_2
Output : δ*
BinSearch(bot, top, F', n, A, B, C, ε_1, ε_2) :
  if (bot == top)
    return(bot)
  else
    mid = (top + bot)/2
    for (i = 1 to n)
      c'_i = max(0, c_i - mid)
    if (maxFlow(A, B, C') < F' - ε_1)
      return(BinSearch(bot, mid, F', n, A, B, C, ε_1, ε_2))
    else if (maxFlow(A, B, C') > F' + ε_2)
      return(BinSearch(mid, top, F', n, A, B, C, ε_1, ε_2))
    else
      return(mid)
```

Let $\delta \in \left[0, \max_{1 \leq i \leq n} \{c_i\}\right]$. Furthermore, let C' such that $\forall i \in [n], c'_i = \max(0, c_i - \delta)$. We define $maxFlow(\delta) = maxFlow_{\mathcal{G}'}(A, B) = F'$ and $MaxFlow(\delta) = X'$. Conventions similar to F, X and C hold for F', X' and δ as far as subscripts are concerned.

Lemma 1 (maxFlow Monotonicity).
It holds that $\forall \delta \in \left[0, \max_{1 \leq i \leq n} \{c_i\}\right]$ *such that* $maxFlow(\delta) < F$*, the function* $maxFlow(\delta)$ *is strictly decreasing with respect to* δ*.*

Proof Sketch. We use contradiction: If we suppose that $maxFlow(\delta)$ is not always strictly decreasing, then we can find a flow for a specific δ which is larger than the expected $maxFlow$. □

From the previous lemma we deduce that, given a $V \in (0, F)$, if we deter-

mine a δ such that $maxFlow\,(\delta) = F - V$, this δ is unique. Furthermore,

$$||\delta_i||_\infty = \max_{1\leq i\leq n} \delta_i = \max_{1\leq i\leq n} (c_i - c'_i) = \max_{1\leq i\leq n} (c_i - \max(0, c_i - \delta)) = \delta \ .$$

It is proven that the two algorithms work as expected, that is an invocation of `dinfmin` with valid inputs returns a capacity C' that yields the desired $maxFlow$ and minimizes $||\delta_i||_\infty$. The complexity of `BinSearch` is $O\left((maxFlow + n)\log_2\left(\frac{top-bot}{\epsilon_1+\epsilon_2}\right)\right)$ and the complexity of `dinfmin` is $O\left((maxFlow + n)\log_2\left(\frac{\delta_{max}}{\epsilon_1+\epsilon_2}\right)\right)$.

3.4 $||\delta_i||_1$ minimizer

We will now concentrate on finding a capacity configuration C' such that $F' = F - V$ and that minimizes $\sum_{i=1}^{n} (c_i - c'_i) = ||\delta_i||_1$ as well. We treat the flow problem as a linear programming problem. Next we see the formulation of the problem in this form, along with a breakdown of each relevant matrix and vector.

In matrix form, the maximum flow problem can be seen as:

$$[AX]_{1..(n^2+n)} \leq [B]_{1..(n^2+n)}$$
$$[AX]_{(n^2+n+1)..(n^2+2n-1)} = [B]_{(n^2+n+1)..(n^2+2n-1)}$$
$$AX \begin{cases} \leq B, \forall i \in \{1, ..., n^2+n\} \\ = B, \forall i \in \{n^2+n+1, ..., n^2+2n-1\} \end{cases}$$
$$\max CX$$

Where:

$$X = \Big\lfloor \underbrace{c'_{11} \dots c'_{1j} \dots c'_{1n}}_{n} \Big\rfloor$$
$$\left.\begin{array}{l} \underbrace{f'_{11} \dots f'_{1j} \dots f'_{1n}}_{n} \\ \vdots \\ f'_{i1} \dots f'_{ij} \dots f'_{in} \\ \vdots \\ f'_{n1} \dots f'_{nj} \dots f'_{nn} \end{array}\right\rfloor^{\mathsf{T}} \Bigg\} n$$

$$X : \left(n^2 + n\right) \times 1$$

$$\begin{aligned} B = [&\underbrace{c_{11} \dots c_{1j} \dots c_{1n}}_{n} \\ &\left.\begin{array}{l} \underbrace{0 \dots 0}_{n} \\ \underbrace{c_{21} \dots c_{2j} \dots c_{2n}}_{n} \\ \vdots \\ c_{i1} \dots c_{ij} \dots c_{in} \\ \vdots \\ c_{n1} \dots c_{nj} \dots c_{nn} \end{array}\right\} n \\ &\underbrace{0 \dots 0}_{n-2} \\ &F - V]^\intercal \end{aligned}$$

$$B : \left(n^2 + 2n - 1\right) \times 1$$

$$C = [\underbrace{0 \dots 0}_{n} \underbrace{1 \dots 1}_{n} \underbrace{0 \dots 0}_{n(n-1)}]$$

$$C : 1 \times \left(n^2 + n\right)$$

The matrix A is implied by the following constraints.

$$A : \left(n^2 + 2n - 1\right) \times \left(n^2 + n\right)$$

Let $n = |\mathcal{V}|$. Also, let the source correspond to 1 and the sink to n. The constraints are:

$$\begin{aligned} \forall j \in [n]\,, \quad & c'_{1j} \le c_{1j} && (n \text{ constraints}) \\ \forall j \in [n]\,, \quad & f'_{1j} - c'_{1j} \le 0 && (n \text{ constraints}) \\ \forall i \in [n] \setminus \{1\}\,, \forall j \in [n]\,, \quad & f'_{ij} \le c_{ij} && (n^2 - n \text{ constraints}) \\ \forall i \in [n] \setminus \{1, n\}\,, \quad & \sum_{j \in [n]} f'_{ij} = \sum_{j \in [n]} f'_{ji} && (n - 2 \text{ constraints}) \\ & \sum_{j \in [n]} f'_{1j} = F - V && (1 \text{ constraint}) \end{aligned} \tag{3}$$

Note that the matrix A is comprised only of 0's, 1's and -1's. The constraints are $n^2 + 2n - 1$ in total. The special constraints are:

$$\forall j \in [n]\,, c'_{1j} \geq 0$$
$$\forall i, j \in [n]\,, f'_{ij} \geq 0$$

The desired optimization is

$$\max \sum_{j \in [n]} f'_{1j} \ .$$

We would like to find a solution that, except for maximizing the flow, also minimizes $||\delta_i||_1$ at the same time. More precisely, we would like to optimize

$$\min \sum_{v \in \mathcal{V}} \left(c_{Av} - c'_{Av}\right)$$

as well. Since we wish to optimize with regards to two objective functions, we approach the problem as follows: Initially, we ignore the minimization and derive the dual of the previous problem with respect to the maximisation. We then substitute the two problems' optimisations with an additional constraint that equates the two objective functions. Due to the Strong Duality theorem of linear programming [5], this equality can be achieved only by the common optimal solution of the two problems. Next we treat the combination of constraints and variables of the primal and the dual problem, along with the newly introduced constraint and the previously ignored $||\delta_i||_1$ minimisation as a new linear problem. For every $j \in [n]$, the solution to this problem contains a c'_{1j}. These capacities will comprise the new configuration that player A requires.

We will now describe the dual problem in detail. In matrix form, the dual problem can be seen as:

$$A^\intercal Y \geq C^\intercal$$
$$\min B^\intercal Y$$

Where $A^\intercal, B^\intercal$ and $C^\intercal$ are known and

$$
\begin{aligned}
Y = [& \underbrace{y_{c11} \cdots y_{c1j} \cdots y_{c1n}}_{n} \\
& \left.\begin{array}{l}
\underbrace{y_{fc11} \cdots y_{fc1j} \cdots y_{fc1n}}_{n} \\
\vdots \\
y_{fci1} \cdots y_{fcij} \cdots y_{fcin} \\
\vdots \\
y_{fcn1} \cdots y_{fcnj} \cdots y_{fcnn}
\end{array}\right\} n \\
& \underbrace{y_{f2} \cdots y_{fi} \cdots y_{f(n-1)}}_{n-2} y_F]^\intercal
\end{aligned}
$$

$$
Y : \left(n^2 + 2n - 1\right) \times 1
$$

Note that Y has one element for each constraint of the primal problem. The constraints of the dual problem are:

$$
\begin{aligned}
\forall j \in [n]\,, \qquad & y_{c1j} - y_{fc1j} \geq 0 \ (n \text{ constraints}) \\
& y_{fc11} + y_F \geq 1 \ (1 \text{ constraint}) \\
\forall j \in [n] \setminus \{1, n\}\,, & y_{fc1j} - y_{fj} + y_F \geq 1 \ (n-2 \text{ constraints}) \\
& y_{fc1n} + y_F \geq 1 \ (1 \text{ constraint}) \\
\forall i \in [n] \setminus \{1, n\}\,, \qquad & y_{fci1} + y_{fi} \geq 0 \ (n-2 \text{ constraints}) \\
\forall i, j \in [n] \setminus \{1, n\}\,, & y_{fcij} + y_{fi} - y_{fj} \geq 0 \ (n^2 - 4n + 4 \text{ constraints}) \\
\forall i \in [n] \setminus \{1, n\}\,, \qquad & y_{fcin} + y_{fi} \geq 0 \ (n-2 \text{ constraints}) \\
& y_{fcn1} \geq 0 \ (1 \text{ constraint}) \\
\forall j \in [n] \setminus \{1, n\}\,, \qquad & y_{fcnj} - y_{fj} \geq 0 \ (n-2 \text{ constraints}) \\
& y_{fcnn} \geq 0 \ (1 \text{ constraint})
\end{aligned}
\tag{4}
$$

The constraints are $n^2 + n$ in total. The special constraints are:

$$
\begin{aligned}
\forall j \in [n]\,, \ & y_{c1j} \geq 0 \\
\forall i, j \in [n]\,, & y_{fcij} \geq 0 \\
\forall i \in [n] \setminus \{1, n\}\,, \ & y_{fi} \in \mathbb{R} \\
& y_F \in \mathbb{R}
\end{aligned}
$$

The desired optimization is

$$\min \left\{ \sum_{j\in[n]} c_{1j} y_{c1j} + \sum_{i\in[n]} \sum_{j\in[n]} c_{ij} y_{fcij} + (F - V)\, y_F \right\} .$$

Everything is now in place to define the linear problem whose solution C' yields $maxFlow\,(C') = F - V$ and minimizes $||\delta_i||_1$. The constraints are (3) and (4) supplemented with a constraint that equates the two problems' optimality functions:

$$\sum_{j\in[n]} f'_{1j} = \sum_{j\in[n]} c_{1j} y_{c1j} + \sum_{i\in[n]} \sum_{j\in[n]} c_{ij} y_{fcij} + (F - V)\, y_F \ .$$

The desired optimisation is

$$\min \sum_{j\in[n]} \left(c_{1j} - c'_{1j} \right) .$$

The final linear program consists of $2n^2 + 2n - 1$ variables and $2n^2 + 2n$ constraints. There exist a variety of algorithms that solve linear programs, such as simplex and ellipsoid algorithm. The ellipsoid algorithm requires polynomial time in the worst-case scenario, but for practical purposes algorithms of the simplex category seem to exhibit better behavior [6].

Appendix

Let source $A \in \mathcal{V}$, sink $B \in \mathcal{V}$, capacity configuration C and a candidate flow X. For X to be valid with respect to C the following two conditions must hold:

$$\forall (v, w) \in \mathcal{E}, x_{vw} \leq c_{vw} \tag{5}$$

$$\forall v \in \mathcal{V} \setminus \{A, B\}, \sum_{w\in N^+(v)} x_{wv} = \sum_{w\in N^-(v)} x_{vw} \tag{6}$$

Proof of Theorem 1: Saturation
We will show constructively that in turn $j - 1$, there exists some valid flow Y such that

$$\forall i \in [n], y_i = c'_i \tag{7}$$

The algorithm that calculates the desired flow can be seen here.

```
Saturated Flow Creation
Input : flow X, capacity C', source A, sink B, length n
Output : Y
1 saturate(X, C', A, B) :
2   Y = X
3   for (i = 1 to n)
4     while (y_i > c'_i)
5       (path, pathFlow) = findPath(Y, v_i, B)
6       reduction = min(pathFlow, y_i - c'_i)
7       for (e in path)
8         y_e = y_e - reduction
9       y_i = y_i - reduction
```

The function `findPath()` is a DFS that starts from v_i and ends at B, returning the edges comprising the path followed and the minimum edge flow encountered on this path. Edges with flow equal to 0 are not traversed.

We have to prove that the algorithm terminates, that the resulting flow is valid for $\mathcal{G}_{j-1}$ and that $\forall i \in [n], y_i = c'_i$. Before proving these properties, we acquire some intermediate results that we will need later.

First of all, it is straightforward to see that `reduction` is always positive. This variable is set in line 6 to be the minimum of $y_i - c'_i$ and `pathFlow`. The former is always positive as can be seen in the condition of the `while` loop (line 4) and so is the latter, as the flow value returned by `findPath()` in line 5 is always positive as explained previously. Thus

$$\texttt{reduction} > 0 \ . \tag{8}$$

Furthermore, given how `pathFlow` (line 5) and `reduction` (line 6) are set, we see that `reduction` is not greater than any of the flows on the `path` during the execution of lines 7 - 9:

$$\forall e \in \texttt{path}, \texttt{reduction} \leq y_e \ . \tag{9}$$

Also from line 6, we have that

$$\texttt{reduction} \leq y_i - c'_i \tag{10}$$

during the execution of lines 7 - 9.

Y is initialized in line 2 to be equal to X, thus initially Y is a valid flow for $\mathcal{G}_{j-1}$. Subsequently, the lines 7 - 9 are the only ones that modify Y. We will now see that the modification is such that the resulting flow

is still valid for $\mathcal{G}_{j-1}$. This holds because no flow is increased according to (8) and thus no flow can surpass its capacity, no flow is reduced to a negative value according to (9) and (10) and finally the reduction of the incoming flow for every node on the `path` is equal to the reduction of the outgoing flow and both equal to `reduction`, thus the necessary properties (5) and (6) for a flow to be valid hold always after the execution of line 9.

We can now prove that the function `findPath()` will always succeed in finding a $(v_i, ..., B)$ `path`. Suppose that this function is unable to find such a `path`. Then it is $y_i > c'_i$ (since the execution is in the `while` loop), there exists no path from v_i to B and the flow is valid. These three affirmations cannot hold simultaneously, thus the initial proposition holds.

We will subsequently prove that the `while` loop (line 4) eventually completes for every $i \in [n]$ and that at the end of the execution the desired property (7) holds. Since Y is a DAG, the function `findPath()` cannot return a `path` that contains an edge (A, v_k) for any k, thus the flow y_k is modified only when $i = k$. Consequently it suffices to show that for every $i \in [n]$ the `while` loop completes and at the end of its execution it is $y_i = c'_i$.

If `reduction` is set to $y_i - c'_i$ in line 6, then y_i will be subsequently set to c'_i in line 9 and the `while` loop will be broken with y_i having the desired value. If, on the other hand, `reduction` is set to `pathFlow` in line 6, then at least one flow other than y_i will be set to zero in line 8. Since `findPath()` cannot fail and the number of edges in the graph $|\mathcal{E}|$ is finite, we deduce that the `while` loop cannot execute indefinitely, thus the algorithm terminates.

Let $i \in [n]$ and $y_{i,old}$ the value the variable y_i was set to before the last execution of line 9 of the relevant `while` loop. We are now ready to prove that at the end of the execution of the `while` loop, it is $y_i = c'_i$. The condition of the loop (line 4) assures that after line 9, $y_i \leq c'_i$. Suppose that $y_i < c'_i$. Then we can deduce that `reduction` was set to `pathFlow` the last time line 6 was executed because, according to the previous paragraph, if it had been set to $y_{i,old} - c'_i$, then it would have held that $y_i = c'_i$ after line 9. This means that

$$\texttt{pathFlow} \leq y_{i,old} - c'_i \tag{11}$$

according to line 6 and

$$y_i = y_{i,old} - \texttt{pathFlow} \Rightarrow \texttt{pathFlow} = y_{i,old} - y_i \tag{12}$$

after line 9. (11) and (12) together give

$$y_{i,old} - y_i \leq y_{i,old} - c'_i \Rightarrow y_i \geq c'_i \ ,$$

which contradicts with the supposition. Thus at the end of the `while` loop, it is $y_i = c'_i$. We have proven that the flow Y that is output by the `saturate()` algorithm is a valid flow for $\mathcal{G}_{j-1}$ and that $\forall i \in [n], y_i = c'_i$.

Since Y is valid for $\mathcal{G}_{j-1}$ and all the flows outgoing from A are compatible with the capacities of $\mathcal{G}_j$, Y is also a valid flow for $\mathcal{G}_j$. Furthermore, since all capacities c'_i are saturated, there can be no more outgoing flow from A, thus Y is a maximum flow in $\mathcal{G}_j$, that is $Y = X'$. □

Proof of Theorem 3: maxFlow Continuity
Let $C_0 \in \mathcal{C}$. We want to prove that

$$\forall \epsilon > 0, \exists \delta > 0 : 0 < ||C - C_0||_p < \delta \Rightarrow |F - F_0| < \epsilon \ .$$

We will prove it by contradiction. Suppose that

$$\exists \epsilon > 0 : \forall \delta > 0, 0 < ||C - C_0||_p < \delta \Rightarrow |F - F_0| \geq \epsilon \ .$$

Let $v_1, u_1 \in V(\mathcal{G})$. Let C such that

$$c_{v_1 u_1} = c_{0, v_1 u_1} + \frac{\epsilon}{2}$$
$$\forall (v, u) \in E(\mathcal{G}) \setminus \{(v_1, u_1)\}, c_{vu} = c_{0,vu} \ .$$

Due to the construction, for $\delta = \epsilon$ we have

$$0 < ||C - C_0||_p < \delta \ . \tag{13}$$

Any valid flow for C_0 is also valid for C, thus

$$F_0 \leq F \ . \tag{14}$$

Also, it is obvious by the way that C was constructed that

$$F \leq F_0 + \frac{\epsilon}{2} \tag{15}$$

From (14) we have $F_0 \leq F + \frac{\epsilon}{2}$, which, in combination with (15), gives

$$|F - F_0| \leq \frac{\epsilon}{2} < \epsilon \ ,$$

which, together with (13) contradicts our supposition. Thus $maxFlow$ is continuous on C_0. Since C_0 is arbitrary, the result holds for all $C_0 \in \mathcal{C}$, thus $maxFlow$ is continuous with respect to $||\cdot||_p$ for any $p \in \mathbb{N} \cup \{\infty\}$. □

Proof of correctness for algorithm `fcfs`

We will first show that at the end of the execution, $i \leq n+1$. Suppose that $i > n+1$ on line 13. This means that $F_{cur,n}$ exists and $F_{cur,n} = F - \sum_{i=1}^{n} x_i = 0 \leq F - V$ since, according to the condition on line 3, $F - V \geq 0$. This means however that the `while` loop on line 8 will break, thus $F_{cur,n+1}$ cannot exist and $i = n + 1$ on line 13, which is a contradiction, thus the first proposition holds. We can also note that, even if $i = n + 1$ at the end of the execution, the `while` loop will break right after the last incrementation, thus the algorithm will never try to read or write the nonexistent objects x_{n+1}, c'_{n+1}.

We will now show that $\forall i \in [n], c'_i \leq x_i$, as per the requirement (1). Let $i \in [n]$. In line 6 we can see that $c'_i = x_i$ and the only other occurence of c'_i is in line 11 where it is never increased ($reduce \geq 0$), thus we see that the requirement (1) is satisfied.

We will finally show that $\sum_{i=1}^{n} c'_i = F - V$. From line 2 we see that $F_{cur,0} = F$. Let $i \in [n]$ such that $F_{cur,i}$ exists. If $F_{cur,i} \leq F - V$, then $F_{cur,i+1}$ does not exist because the `while` loop (line 8) breaks after calculating $F_{cur,i}$. Else

$$F_{cur,i+1} = F_{cur,i} - \min\left(x_{i+1}, F_{cur,i} - F + V\right) \ . \text{ (lines 9 - 10)}$$

If $\nexists i \in [n] : \min\left(x_i, F_{cur,i-1} - (F - V)\right) = F_{cur,i-1} - (F - V)$, then $\forall i \in [n], \min\left(x_i, F_{cur,i} - (F - V)\right) = x_i$, thus from line 11 it will be $\forall i \in [n], c'_i = 0$ and from line 10, $F_{cur,n} = 0$. However, we have

$$\left.\begin{array}{l} min\left(x_n, F_{cur,n-1} - (F - V)\right) \neq F_{cur,n-1} - (F - V) \\ F_{cur,n-1} = x_n \end{array}\right\} \Rightarrow$$
$$\Rightarrow x_n < x_n - (F - V) \Rightarrow F < V$$

which is a contradiction, since if this were the case the algorithm would have failed on lines 3 - 4. Thus

$$\exists i \in [n] : \min\left(x_{i+1}, F_{cur,i} - (F - V)\right) = F_{cur,i} - (F - V)$$

That is the only $i \in [n]$ such that $F_{cur,i+1} = F - V$, so

$$\forall 0 < k < i, F_{cur,k} = F_{cur,k-1} - x_k \Rightarrow F_{cur,i} = F - \sum_{k=1}^{i-1} x_k \ .$$

Furthermore, since $F_{cur,i+1} = F - V$, it is

$$c'_{i+1} = x_{i+1} - F_{cur,i} + F - V = x_i - F + \sum_{k=1}^{i-1} x_k + F - V \Rightarrow$$

$$\Rightarrow c'_{i+1} = \sum_{k=1}^{i} x_k - V \ ,$$

$$\forall k \leq i, c'_k = 0 \text{ and}$$

$$\forall k > i+1, c'_k = x_k \ .$$

In total, we have

$$\sum_{k=1}^{n} c'_k = \sum_{k=1}^{i} x_k - V + \sum_{k=i+1}^{n} x_k = \sum_{k=1}^{n} x_k - V \Rightarrow \sum_{k=1}^{n} c'_k = F - V \ .$$

Thus the requirement (2) is satisfied. □

Complexity of algorithm `fcfs`
Since i is incremented by 1 on every iteration of the `while` loop (line 12) and $i < n + 1$ at the end of the execution, the complexity of the `while` loop is $O(n)$ in the worst case. The complexity of lines 3 - 4 and 7 is $O(1)$ and the complexity of lines 2, 5 - 6 and 13 is $O(n)$, thus the total complexity of algorithm 3.2 is $O(n)$. □

Proof of correctness for algorithm `abs`
First of all, we can note that, if $F_{prov} \leq F - V$ in line 11, then the execution will enter the `else` clause of line 15. Therefore, in line 18, F_{cur} will get the value $F - V$, as we can see by executing the lines 16 - 18 by hand. This in turn means that the loop in line 8 will break right after the `else` clause is executed. Furthermore, the assignment in line 14 in combination with the truth of the statement $F_{prov} > F - V$ in line 11 shows that, if the execution enters the `if` clause of line 11, then the loop of line 8 will be executed at least once more. These two observations amount to the fact that the `else` clause will be executed exactly one time and afterwards the `while` loop will break.

We use the notation $F_{cur,0}$, $reduction_0$ and $empty_0$ to refer to the initial values of the corresponding variables, as set in lines 2, 7 and 6 respectively. Furthermore, the notation $empty_j$, $reduction_j$, $F_{cur,j}$ and i_j is used to refer to the values of the corresponding variables after the j-th iteration of the `while` loop. i_j is chosen in line 9. From lines 10, 12, 14, 11

and 16 to 18 we see that

$$F_{cur,j} = \begin{cases} F_{prov,j}, & F_{prov,j} > F - V \\ F - V, & F_{prov,j} \leq F - V \end{cases} \text{ , where}$$

$$F_{prov,j} = F_{cur,j-1} - (n - empty_{j-1}) \left(x_{i_j} - x_{i_{j-1}} \right), j \geq 1 \text{ and } x_{i_0} = 0 \ .$$

It is worth noting that the maximum number of iterations is n, or else $j \leq n$. This holds because, if we suppose that $F_{cur,n+1}$ exists, it is

$$F_{cur,n} > F - V \geq 0 \tag{16}$$

However, we can easily see that in this case

$$\begin{aligned} F_{cur,n} &= F_{cur,0} - \sum_{j=1}^{n} (n - (j-1)) \left(x_{i_j} - x_{i_{j-1}} \right) = \\ &= \sum_{j=1}^{n} x_{i_j} - \sum_{j=1}^{n} (n - (j-1)) \, x_{i_j} + \sum_{j=1}^{n-1} (n-j) \, x_{i_j} = \\ &= \sum_{j=1}^{n} x_{i_j} - \sum_{j=1}^{n-1} [(n - (j-1)) - (n-j)] x_{i_j} - (n - (n-1)) \, x_{i_n} = \\ &= \sum_{j=1}^{n} x_{i_j} - \sum_{j=1}^{n-1} x_{i_j} - x_{i_n} = 0 \ , \end{aligned}$$

which is a contradiction to (16), thus $F_{cur,n+1}$ does not exist and $j \leq n$. This means that `popMin()` will never fail.

We will now show that $\forall j \in [n], empty_j < n$. At line 6, it is $empty_0 = 0 < n$. *empty* is again modified in line 13, where it is incremented by at most 1 at each iteration of the `while` loop (line 8). As we saw above, the iterations cannot exceed n and *empty* is not incremented in the last iteration which consists of the `else` clause, thus $\forall j \in [n], empty_j < n$.

Next, we will show that $\forall i \in [n], c'_i \leq x_i$, as per the requirement (1). From line 22, we see that it suffices to prove that $reduction \geq 0$. In line 7, *reduction* is initialized to 0. In line 12, *reduction* is set to x_i, which is always a non-negative value. The last line where *reduction* is modified is 17. In this line, it is $F_{cur} > F - V$ or else the `while` loop would have broken before beginning this iteration and $n > empty$ as we previously saw. Thus the non-negative variable *reduction* is increased and the resulting value is always positive.

We will finally show that $\sum_{i=1}^{n} c'_i = F - V$, which satisfies the requirement (2). Let $k, 0 \leq k \leq n$ be such that at the end of the execution

$$\forall j \leq k, c_{i_j} = 0 \wedge \forall j > k, c_{i_j} > 0 \ .$$

The following holds:

$$\sum_{j=1}^{n} c'_{i_j} = \sum_{j=k+1}^{n} c'_{i_j} = \sum_{j=k+1}^{n} \left(x_{i_j} - reduction_{k+1} \right) =$$

$$\sum_{j=k+1}^{n} \left(x_{i_j} - \left(x_{i_k} + \frac{F_{cur,k} - (F - V)}{n-k} \right) \right)$$

$$\sum_{j=k+1}^{n} \left(x_{i_j} - \left(x_{i_k} + \frac{F_{cur,0} - \sum_{l=1}^{k} (n - (l-1)) \left(x_{i_l} - x_{i_{l-1}} \right) - F + V}{n-k} \right) \right)$$

$$\sum_{j=k+1}^{n} \left(x_{i_j} - \left(x_{i_k} + \frac{F - \sum_{l=1}^{k} (n-l+1)\, x_{i_l} + \sum_{l=1}^{k-1} (n-l)\, x_{i_l} - F + V}{n-k} \right) \right)$$

$$\sum_{j=k+1}^{n} x_{i_j} - (n-k)\, x_{i_k} - \frac{n-k}{n-k} \left(- \sum_{l=1}^{k-1} x_{i_l} - (n-k+1)\, x_{i_k} + V \right)$$

$$\sum_{j=k+1}^{n} x_{i_j} - (n-k)\, x_{i_k} + \sum_{j=1}^{k-1} x_{i_j} + (n-k+1)\, x_{i_k} - V$$

$$= \sum_{j=1}^{n} x_{i_j} - V = F - V \ ,$$

thus the desired property holds. □

Complexity of algorithm `abs`

Lines 3 - 4, 6 - 7 and 10 - 18 have a complexity of $O(1)$. Lines 2 and 21 - 23 have a complexity of $O(n)$. The `while` loop of line 8 is repeated at most n times, as we saw in the proof of correctness. Thus the total complexity is

$$O(preprocess) + O(n)\, O(popMin) \ .$$

If the flows are first sorted, it would be

$$O(preprocess) = O(quicksort) = O(nlogn) \text{ and}$$
$$O(popMin) = O(1) \ ,$$

amounting to a total complexity of $O(nlogn)$. In the case a Fibonacci heap is used, it is

$$O(preprocess) = O(FibonacciHeap) = O(n) \text{ and}$$
$$O(popMin) = O(find\text{-}min) + O(delete\text{-}min) = O(logn) \ ,$$

thus the total complexity is again $O(nlogn)$. □

Proof that algorithm `abs` minimizes $||\Delta_i||_\infty$
Let *reduction* be the final value of the corresponding variable. It holds that

$$\forall i \in [n] : c'_i > 0, \Delta_i = reduction \ ,$$
$$\forall i \in [n] : c'_i = 0, \Delta_i = x_i \text{ and}$$
$$\forall i \in [n] : c'_i = 0, reduction \geq x_i \ ,$$

thus we deduce that

$$||\Delta_i||_\infty = \max_{1 \leq i \leq n} (x_i - c'_i) = reduction \ .$$

With the capacity configuration C' resulting from `abs()`, it holds that $\sum_{i=1}^{n} c'_i = F - V$. Suppose that there exists a configuration C_1 that maintains that property:

$$\sum_{i=1}^{n} c_{1,i} = F - V \tag{17}$$

and furthermore

$$||\Delta_{1,i}||_\infty = b < reduction \ . \tag{18}$$

Then it must be

$$\forall i \in [n], \Delta_{1,i} \leq b \Rightarrow \forall i \in [n], c_{1,i} \geq x_i - b \ .$$

Without loss of generality, suppose that x_i are sorted in ascending order. Then

$$\exists k' \in [n] \cup \{0\} : \begin{cases} \forall i \leq k', x_i \leq reduction \\ \forall i > k', x_i > reduction \end{cases}$$
$$\text{and } \exists k_1 \in [n] \cup \{0\} : \begin{cases} \forall i \leq k_1, x_i \leq b \\ \forall i > k_1, x_i > b \end{cases} \ .$$

Since $b < reduction$, it is $k_1 \leq k'$. It is:

$$\forall i \in [k_1], 0 \leq c_{1,i} \leq x_i \text{ and}$$
$$\forall i \in [n] \setminus [k_1], x_i - b \leq c_{1,i} \leq x_i \ .$$

Let all $c_{1,i}$ assume the smallest possible value according to the above restriction. Then

$$\forall i \in \left[k'\right] \setminus \left[k_1\right], x_i > b \text{ and} \tag{19}$$

$$\sum_{i=1}^{n} c_{1,i} = \sum_{i=k_1+1}^{n} \left(x_i - b\right) \overset{(19)}{\geq} \sum_{i=k'+1}^{n} \left(x_i - b\right) \overset{(18)}{>}$$

$$> \sum_{i=k'+1}^{n} \left(x_i - reduction\right) = \sum_{i=1}^{n} c'_i = F - V \ .$$

We see that even with the minimum possible C_1 configuration, the hypothesis (17) is violated, thus the existence of C_1 is a contradiction. We have thus proven the proposition. □

Proof of correctness for algorithm `prop`

We will first show that $\forall i \in [n], c'_i \leq x_i$. Let $i \in [n]$. According to line 6, which is the only line where c'_i is modified, it is

$$c'_i = x_i - \frac{V}{F} x_i = x_i \left(1 - \frac{V}{F}\right) \ . \tag{20}$$

Since $0 < V \leq F$, it is

$$0 < \frac{V}{F} \leq 1 \Rightarrow 0 \leq 1 - \frac{V}{F} < 1 \ . \tag{21}$$

From (20) and (21), along with the fact that $x_i \geq 0$, it is straightforward to see that $c'_i \leq x_i$, thus the requirement (1) is satisfied.

We will now show that $\sum\limits_{i=1}^{n} c'_i = F - V$. At the end of the execution it is

$$\sum_{i=1}^{n} c'_i = \sum_{i=1}^{n} \left(x_i - \frac{V}{F} x_i\right) = \sum_{i=1}^{n} x_i - \frac{V}{F} \sum_{i=1}^{n} x_i \overset{\sum_{i=1}^{n} x_i = F}{=} F - V \ ,$$

thus the requirement (2) is satisfied. □

Complexity of algorithm `prop`

The complexity of lines 2, 5 - 6 and 7 is $O\left(n\right)$ and the complexity of lines 3 - 4 is $O\left(1\right)$, thus the total complexity of the algorithm is $O\left(n\right)$. □

Proof of Lemma 1: maxFlow Monotonicity

Supppose that

$$\exists \delta_1, \delta_2 : \delta_1 < \delta_2 \wedge maxFlow\left(\delta_1\right) \leq maxFlow\left(\delta_2\right) < F \ .$$

We choose X'_1, X'_2 such that

$$\forall i \in [n], x'_{1,i} \leq x_i \wedge x'_{2,i} \leq x_i \ .$$

This is always possible because we can derive the maximum flows X'_1 and X'_2 starting with X and reducing the flow along paths beginning from edges with capacities $c'_{1,i} < x_i$ and $c'_{2,i} < x_i$ respectively, until the flows become valid.

Define $MinCut\,(\delta)$ as the minimum cut set of the $MaxFlow\,(\delta)$ configuration. Let

$$S_j = \{i \in [n] : v_i \in N^+\,(A) \cap MinCut\,(\delta_j)\}, j \in \{1, 2\} \ .$$

It holds that $S_j \neq \emptyset$. Suppose that $S_j = \emptyset$. Since $F > F'_j$, there exists a path from A to B on G'_j with positive flow not used, thus X'_j is not the maximum flow, which is a contradiction. Thus $S_j \neq \emptyset, j \in \{1, 2\}$.

Moreover, it holds that $S_1 \subseteq S_2$, since $\forall i \in [n], c'_{2,i} \leq c'_{1,i}$. More precisely, it is

$$\forall i \in [n] : c'_{2,i} > 0, c'_{2,i} < c'_{1,i} \ .$$

Every node in the $MinCut\,(\delta_j)$ is saturated, thus

$$\forall i \in S_1, x'_{j,i} = c'_{j,i}, j \in \{1, 2\} \ .$$

Thus $\sum\limits_{i \in S_1} x'_{2,i} < \sum\limits_{i \in S_1} x'_{1,i}$ and, since $maxFlow(\delta_1) \leq maxFlow(\delta_2)$, we conclude that for X'_1, X'_2 it is $\sum\limits_{i \in [n] \setminus S_1} x'_{2,i} > \sum\limits_{i \in [n] \setminus S_1} x'_{1,i}$. However, since $x'_{i,j} \leq x_i, j \in \{1, 2\}$, the configuration X'' such that

$$\begin{aligned} \forall i \in S_1, x''_i &= x'_{1,i} \\ \forall i \in [n] \setminus S_1, x''_i &= x'_{2,i} \end{aligned}$$

is a valid flow configuration for C'_1 and then

$$F'_1 \geq \sum_{i \in S_1} x''_i + \sum_{i \in [n] \setminus S_1} x''_i = \sum_{i \in S_1} x'_{1,i} + \sum_{i \in [n] \setminus S_1} x'_{2,i} > maxFlow\,(\delta_1) \ \ ,$$

which is a contradiction because $F'_1 = maxFlow\,(\delta_1)$ by the hypothesis. Thus $maxFlow\,(\delta_1) > maxFlow\,(\delta_2)$ and, since δ_1, δ_2 were chosen arbitrarily with the restriction $\delta_1 < \delta_2$, we deduce that the proposition holds. □

We will now prove that `BinSearch` returns the desired δ^* when we provide it with an appropriate interval as input.

Proof of correctness for algorithm `BinSearch`
Suppose that

$$\left[F' - \epsilon_1, F' + \epsilon_2\right] \subset \left[maxFlow\left(top\right), maxFlow\left(bot\right)\right] \quad .$$

We will prove that

$$maxFlow\left(\delta^*\right) \in \left[F' - \epsilon_1, F' + \epsilon_2\right] \quad .$$

First of all, we should note that if an invocation of `BinSearch` returns without calling `BinSearch` again (line 3 or 13), its return value will be equal to the return value of the initial invocation of `BinSearch`, as we can see on lines 9 and 11, where the return value of the invoked `BinSearch` is returned without any modification. The case where `BinSearch` is called recursively is analyzed next:

If $maxFlow\left(\frac{top+bot}{2}\right) < F' - \epsilon_1$ (line 8) then, by the maxFlow monotonicity lemma, $\delta^* \in \left[bot, \frac{top+bot}{2}\right)$. As we see on line 9, the interval $\left(\frac{top+bot}{2}, top\right]$ is discarded when the next `BinSearch` is recursively called. Since $F' + \epsilon_2 \leq maxFlow\left(bot\right)$, we have

$$\left[F' - \epsilon_1, F' + \epsilon_2\right] \subset \left[maxFlow\left(\frac{top + bot}{2}\right), maxFlow\left(bot\right)\right]$$

and the length of the available interval is divided by 2.

Similarly, if $maxFlow\left(\frac{top+bot}{2}\right) > F' + \epsilon_2$ (line 10) then, by the maxFlow monotonicity lemma, it holds that $\delta^* \in \left(\frac{top+bot}{2}, top\right]$. According to line 11, the interval $\left[bot, \frac{top+bot}{2}\right)$ is discarded when the next `BinSearch` is recursively called. Since $F' - \epsilon_1 \geq maxFlow\left(top\right)$, we have

$$\left[F' - \epsilon_1, F' + \epsilon_2\right] \subset \left[maxFlow\left(top\right), maxFlow\left(\frac{top + bot}{2}\right)\right]$$

and the length of the available interval is divided by 2.

As we saw, it holds that

$$\left[F' - \epsilon_1, F' + \epsilon_2\right] \subset \left[maxFlow\left(top\right), maxFlow\left(bot\right)\right]$$

in every recursive call and $top - bot$ is divided by 2 in every call. It is $|[F' - \epsilon_1, F' + \epsilon_2]| = \epsilon_1 + \epsilon_2$. Let bot_0, top_0 the input values given to the

initial invocation of `BinSearch`, bot_j, top_j the input values given to the j-th recursive call of `BinSearch` and $len_j = |[bot_j, top_j]| = top_j - bot_j$. We have

$$\forall j > 0, len_j = top_j - bot_j = \frac{top_{j-1} - bot_{j-1}}{2} \Rightarrow$$
$$\forall j > 0, len_j = \frac{top_0 - bot_0}{2^j} \ .$$

We understand that in the worst case

$$len_j = \epsilon_1 + \epsilon_2 \Rightarrow 2^j = \frac{top_0 - bot_0}{\epsilon_1 + \epsilon_2} \Rightarrow j = \log_2(\frac{top_0 - bot_0}{\epsilon_1 + \epsilon_2}) \ .$$

Also, as we saw earlier, δ^* is always in the available interval, thus it holds that

$$maxFlow\left(\delta^*\right) \in \left[F' - \epsilon_1, F' + \epsilon_2\right] \ .$$

□

Complexity of algorithm `BinSearch`
Lines 2 - 3 have complexity $O\left(1\right)$, lines 6 - 7 have complexity $O\left(n\right)$, lines 8 - 13 have complexity $O\left(maxFlow\right) + O\left(BinSearch\right)$. As we saw in the proof of correctness for `BinSearch`, we need at most $\log_2\left(\frac{top-bot}{\epsilon_1+\epsilon_2}\right)$ recursive calls of `BinSearch`. Thus the function `BinSearch` has worst-case complexity

$$O\left((maxFlow + n)\log_2\left(\frac{top - bot}{\epsilon_1 + \epsilon_2}\right)\right) \ .$$

□

Proof of correctness for algorithm `dinfmin`
Let $C' =$ `dinfmin(`c_i`, V, `ϵ_1`, `ϵ_2`)`. We will show that

$$F' \in [F - V - \epsilon_1, F - V + \epsilon_2] \ .$$

We can easily see that

$$maxFlow\left(0\right) = F \text{ and}$$
$$maxFlow\left(\max_{i\in[n]}\{c_i\}\right) = 0 \ ,$$

thus $\delta^* \in \left(0, \max_{i\in[n]}\{c_i\}\right)$. From the proof of correctness for `BinSearch`, we know that $maxFlow\left(\delta^*\right) \in [F - V - \epsilon_1, F - V + \epsilon_2]$, given that ϵ_1, ϵ_2

are chosen so that $F - V - \epsilon_1 \geq 0, F - V + \epsilon_2 \leq F$, so as to satisfy the condition $[F' - \epsilon_1, F' + \epsilon_2] \subset [maxFlow\,(top)\,, maxFlow\,(bot)]$. The last condition is always met due to lines 5 - 7. □

Complexity of algorithm `dinfmin`

The complexity of lines 3 - 4 and 5 - 7 is $O\,(1)$, the complexity of lines 2, 8, 10 - 11 and 12 is $O\,(n)$ and the complexity of line 9 is $O\,(BinSearch) = O\left((maxFlow + n)\log_2\left(\frac{\delta_{max}}{\epsilon_1+\epsilon_2}\right)\right)$, thus the total complexity of dinfmin is $O\left((maxFlow + n)\log_2\left(\frac{\delta_{max}}{\epsilon_1+\epsilon_2}\right)\right)$. □

References

1. Thyfronitis Litos O. S., Zindros D.: Trust Is Risk: A Decentralized Financial Trust Platform. IACR: Cryptology ePrint Archive (2017)
2. Daskalakis C., Goldberg P. W., Papadimitriou C. H.: The complexity of computing a Nash equilibrium. SIAM Journal on Computing: vol. 1(39), pp. 195–259 (2009)
3. Marks R.: Combining Simultaneous and Sequential Games. http://www.agsm.edu.au/bobm/teaching/SGTM/lect06pr-3.pdf p. 3 (2009)
4. Freeman L. C.: Centrality in Social Networks: Conceptual Clarification. Social Networks: pp. 215–239 (1978)
5. Ahuja R. K., Magnanti T. L., Orlin J. B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall: https://ocw.mit.edu. License: Creative Commons BY-NC-SA. (1993)
6. Illés T., Terlaky T.: Pivot versus interior point methods: Pros and cons. European Journal of Operational Research: vol. 140(2), pp. 170–190 (2002)