Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Balanced Routing in Micropayment Channel Networks

Master's thesis

Sascha Schmid

saschmi@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Conrad Burchert
Prof. Dr. Roger Wattenhofer

December 20, 2017

Abstract

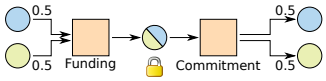
Cryptocurrencies like Bitcoin have scalability problems. To overcome this, micropayment channels have been proposed, which can be used to generate a payment network where off-chain transactions are forwarded over multiple nodes. This thesis presents a routing algorithm that has a small memory footprint and balances the network edges. We use forwarding fees as incentives to participate and to steer the traffic. The core idea of the algorithm builds on multiple combined minimum spanning trees providing approximate routing information. Our simulations show that the algorithm scales well to big networks of at least a hundred thousand nodes and provides balance to the edges to guarantee a long lasting good performance.

Contents

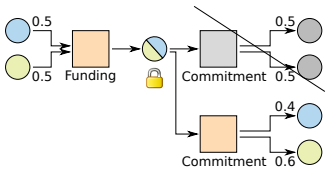
Chapter 1

Introduction

For years now Bitcoin had a problem with its scalability. While other payment options like Visa are able to handle up to tens of thousands of transactions per second [1] and over a few hundred transactions per second on average, Bitcoin supports less than 10 transactions per second and will struggle to scale above 100 transactions per second with known protocol improvements and parameter changes [2, 3]. Although many different approaches were tried to scale up Bitcoin, no solution pushed through so far. One approach that sparked quite some interest in the Bitcoin community were micropayment networks [4, 5, 6] such as the Lightning Network (LN) approach by Poon and Dryja. Before, all transactions had to be verified by the blockchain, which proved to be a bottleneck. What they did in their paper was to set up channels between two Bitcoin nodes that do not need the blockchain to trade funds in between them. The problem is that building such a channel (and eventually, tearing it down) is an on-chain transaction, which needs to be verified by the blockchain, and the funds that are initially committed to this new channel are frozen on this channel, i.e., they cannot be used for anything else (see Figure 1.1). So, building a LN channel for only one transaction is inefficient, since it takes at least two on-chain transactions (actually more when counting the additional transactions used for fraud detection and response) to establish and tear down the channel as opposed to only one transaction if the transaction itself was done on the blockchain directly. Also,



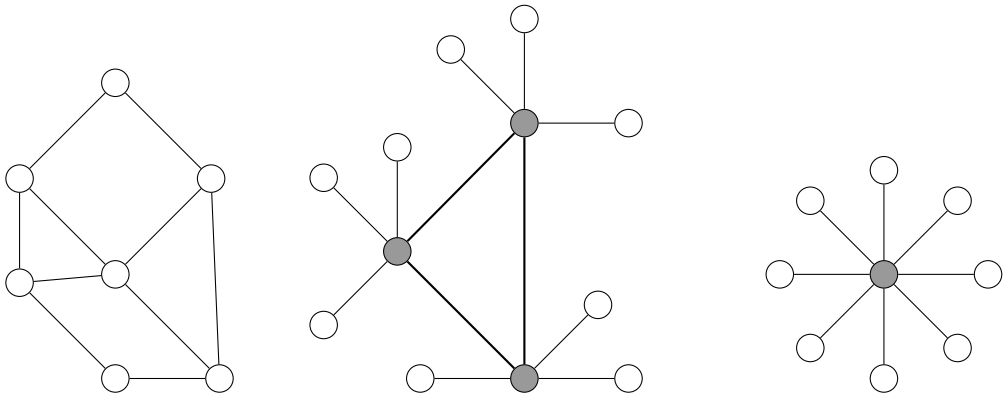
(a) Nodes A and B (left) open a micropayment channel by investing funds in a transaction with a shared output (middle), which then can be distributed, but not yet committed, to both parties in a second transaction (right).



(b) When updating a channel, the second transaction is invalidated and a new transaction is done with adjusted outputs.

Figure 1.1: Creating and updating micropayment channels. Images from Burchert et al. [7].

when one node establishes x different channels, it has to split its funds on these channels, e.g., $\frac{1}{x}$ on each channel, which reduces the size of transactions that can be sent. Lastly, although some good suggestions have been made, no standard way of routing transactions in such a payment network has been decided on. The problem is that routing in such networks has to consider the special nature of these channels to prevent them from being off balance. We would like to present an approach to solve routing in such payment networks.
Since the actual network does not yet exist, it is hard to assume any attributes the network will have. Most Bitcoin supporters would prefer a completely decentralized network, i.e., all nodes have similar degree (compare Figure 1.2a), as opposed to a more centralized hub-and-spoke network (Figure 1.2b) or even worse, a completely centralized star layout (Figure 1.2c). This is not only due to the increased robustness of a decentralized network, as there is no single point of failure (or small set of important nodes in the hub-and-spoke network), but also increased confidentiality, as spying on the network is easier when most traffic passes through a small set of nodes.



(a) Decentralized (b) Hub-and-Spoke (c) Star

Figure 1.2: Different kinds of networks

1.1 Lightning Network Channels

As can be seen in Figure 1.1, to set up a Lightning Network channel, two nodes generate a funding transaction with their respective committed funds as input and a shared output. The shared output is distributed among both parties with a (not yet broadcast) commitment transaction. Each commitment transaction consists of two transactions with identical output, but one can only be broadcast by A (C1a) and one only by B (C1b), as they are only signed by the other party. Each commitment transaction also includes a delivery transaction (D1a and D1b

respectively), which allows the non broadcasting node to collect its funds immediately, as well as a breach remedy transaction (BR1a and BR1b) for all outdated commitment transactions, which is used to grant the non broadcasting node all funds on the channel if an outdated commitment transaction is broadcast. The breach remedy transactions are only generated when a new commitment transaction is generated, i.e., BR1a is generated when C2a is generated, BR2a when C3a is generated and so on.
Assume both parties initially invested 0.5 BTC, then the shared transaction output is worth 1.0 BTC and the initial, not yet broadcast, output distribution is 0.5 BTC each (C1a and C1b). Now assume node A is paying node B 0.1 BTC. A creates a new commitment transaction which sets the distribution to 0.4 BTC for A and 0.6 BTC to B (C2b). A signs this transaction and sends it to B. B then creates the same transaction (C2a), signs it and sends it to A. Then A creates the signed breach remedy transaction (BR1a) for the previous commitment transaction and sends it to B, while B does the same thing. Now A and B can delete the previous commitment transaction (C1a and C1b), but keep the breach remedies (BR1a and BR1b).

Chapter 2

Algorithm

As our main contribution we present a routing algorithm which can be used in general graphs as well as in special graphs such as micropayment channel networks. First we describe the algorithm for general graphs and later adapt it to micropayment channel networks. The core idea behind our algorithm is using multiple Minimum Spanning Trees (MSTs), e.g. 20 at a time, to route the transaction at every node to the appropriate neighbor, e.g., the neighbor with minimum known distance to the goal node. The node then considers the distance from all its neighbors on all MSTs to the goal node. Figure 2.1 shows how two trees may find an optimal path.

2.1 General Graphs

For general graphs $G = (V, E)$ and two nodes $S, D \in V, S \neq D$, the goal is to find the shortest path from S to D on G.

Definition 2.1. Given a Graph $G = (V, E)$ with a MST T, with maximum depth d_T and root R, a start node S and a goal node D. The *distance* $dist_T(S, D)$ of S and D on T is the minimum number of hops required on T to get from S to D. The *depth* $d_T(node)$ of any node is defined as $dist_T(R, node)$. A node $K = K_T(S, D)$ with depth $d_T(K)$ is a *lowest common ancestor* of S and D if K is an ancestor of S and of D and no node K' exists that is an ancestor of S and D with depth $d_T(K') > d_T(K)$.

The MSTs are calculated with an adapted version of Dijkstra's algorithm. Since we want the cheapest (here: shortest) paths from the root to all other nodes and the tree does not have to be available instantly, but as soon as possible, Dijkstra's algorithm can be adapted to find such trees by gradually increasing the distance in a flood and echo setup for the case where all edge costs are one, or by e.g. adding timeslots in the case where edges have different costs. This allows for generating such trees without global view. While it is possible that,

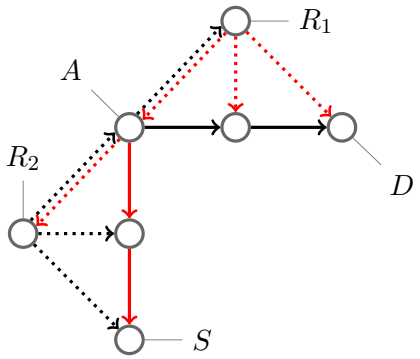


Figure 2.1: Example of two trees in a network with roots R_1 and R_2. Dotted lines imply multiple nodes on the path. Assuming the distance from S via R_1 to D is smaller than the distance from S via R_2 to D, then the algorithm finds the shortest path S-A-D

with a wait time of, e.g., 5 seconds, a non optimal edge (i.e., an edge from node A to node B, which results in a cost from the root to B which is not minimal) might be chosen because the edge on the cheapest path had a delay of more than the wait time, the optimal edge should be neglected as it is unreliable with such delays (at least at the moment).
When the next hop needs to be found from some node S to reach node D, for each neighbor N_j of S and each MST T_i, $dist_{T_i}(N_j, D)$ can be computed and the next hop is the neighbor with the minimum distance. If the minimum distance is achieved by multiple neighbor-tree-pairs, the next hop can be chosen randomly from these neighbors. Assuming, for a tree T_i, that the lowest common ancestor of S and D is K_i, then $dist_{T_i}(S, D) = dist_{T_i}(S, K_i) + dist_{T_i}(K_i, D)$. If at any node A on the path a neighbor N of A is chosen as next hop, with $dist_T(N, D) < dist_T(A, D) - 1$, then the originally estimated path length is reduced by $dist_T(A, R) + dist_T(R, N) - 1$.

Lemma 2.2. *If K_i is equal to S or D, the direct path from S to D on T_i will have optimal length.*

Proof. Because all T_i are MST, the path from the R_i to any other node on T_i is optimal. For any subtree of T_i, the path from the root of the subtree to any other node in the subtree will be optimal. □

Note: If the number of trees is equal to the number of nodes, every path will be optimal.

Lemma 2.3. *Given two nodes S and D and a set of MSTs $T_1 ... T_n$ with lowest common ancestors K_i. A lower bound for the optimal path length from S to D*

is given by $\max_i(|d_{T_i}(S) - d_{T_i}(D)|)$. *An upper bound is given by* $\min_i(d_{T_i}(S) + d_{T_i}(D) - d_{T_i}(K_i))$.

Proof. Due to how the MSTs are generated, for any two nodes A and B that are connected, the depth can only differ by at most one. Using this, the difference in depth between nodes S and D implies a minimum of edges needed between S and D. Since this holds for all trees, the maximum difference over all trees defines the lower bound. The upper bound follows from routing only over the tree with minimal initial distance. The upper bound is tight, since the network could actually be a tree. □

Note: If $S \neq D$, the minimum distance is 1, even if $d_{T_i}(S) = d_{T_i}(D)\ \forall T_i$. Also, if the network is actually a tree (or close to a tree), having more trees will not increase performance while increasing computational cost (i.e., the number of trees should be higher if the network has many edges).

Lemma 2.4. *The achieved cost to go from some node S to some node D is bounded by the optimal cost +* $\min_i(\min(d_{T_i}(S), d_{T_i}(D)) - d_{T_i}(K_{T_i}(S, D))) \times 2$

Proof. Assuming only one MST (as additional MSTs only reduce the upper bound and increase the lower bound). Given nodes S and D, with depths $d_{T_i}(S)$ and $d_{T_i}(D)$, and lowest common ancestor $K_i = K_{T_i}(S, D)$. The minimum distance is $|d_{T_i}(S) - d_{T_i}(D)|$ as given by Lemma 2.3. The worst case cost when routing from S over K_i to D are $(d_{T_i}(S) - d_{T_i}(K_i)) + (d_{T_i}(D) - d_{T_i}(K_i)) = d_{T_i}(S) + d_{T_i}(D) - 2 \times d_{T_i}(K_i)$. Assume $d_{T_i}(S) < d_{T_i}(D)$, then the additional cost is equal to $(d_{T_i}(S) + d_{T_i}(D) - 2 \times d_{T_i}(K_i)) - (d_{T_i}(S) - d_{T_i}(D))$ which is equal to $(d_{T_i}(S) - d_{T_i}(K_i)) \times 2$. Likewise for $d_{T_i}(S) > d_{T_i}(D)$. □

Definition 2.5. When generating a MST, each node is assigned an *Address*. Each node sends address propositions to all its prospective children. When receiving multiple propositions at the same time, a node can choose freely which proposal to use (in general, being closer to the root is better for the node, since the cost will probably be smaller to send transactions).

A simple approach would be to use the depth of the node in this tree, an id (or hash) of the root of this tree and then the path from the root to the node, e.g. a node with address $3|R_i|2|1|3$ would be a node with depth 3 and the path from the root R_i would be the second child and then the first child and then the third child.
There are possibilities to reduce the memory usage of these addresses by using a compressed form of address. Compressing the address would save memory, but would increase the computation for the next hop, since the address would have to be decompressed to get important information about the node.
An actual implementation of this addressing scheme could be using 10 bits for

Figure 2.2: New edge type.
Each edge now consists of two edges, one in each direction, with separate capacities (amount of tokens) and fees.

the depth (which allows trees with a depth of up to 1023), 32 bits (or even less) for the root id (or a hash thereof) and depth $\times$ 16 bits for the path encoding. While this in theory allows for big addresses, when a node has depth 1023, our simulations showed that with random graphs of size 1 million, the maximum depth was rarely over 60, so, with high probability, even huge graphs will not use the whole 1023 depth layers. Also, having 16 bits per level for the path encoding allows for trees with over 65000 children per node, which is probably too much and should be adapted to the size and connectivity of the actual implemented network.

2.2 Micropayment Channel Networks

Moving on to micropayment channel networks. While the nodes in the graph are still the same, new edges are needed to fit this model. A payment network edge consists of funds or *Tokens* (Definition 2.6) on either side of the edge (compare Figure 1.1). Each transaction passing this edge pushes some tokens from one side to the other. If the amount of tokens on an edge in one direction is smaller than the transaction size, the transaction cannot be routed over this edge. This increases the amount of edges from E to $2 \times E$.Westill use the same notation for start node S, goal node D, and for an edge, the endpoints A and B. The goals of the algorithm are now not only to find the path, but also keep the tokens on the edges in balance and minimize the cost for S (see Definition 2.7).

Definition 2.6. On every payment network edge, funds have to be used to create the edge. These funds are then distributed between the two endpoints. We call these funds *Tokens* and the relative distribution of these tokens to both nodes the *Balance*.

Definition 2.7. Whenever a transaction is sent over an edge from A to B, a *Fee* F_{AB} has to be paid. The receiving end of the transaction, B, decides, how big the fee is. The fee is paid half and half by S and A to B.

The fee is used as a incentive to get nodes to join the routing scheme. Alternatively, the fee could be paid by S alone (which would result in higher cost for

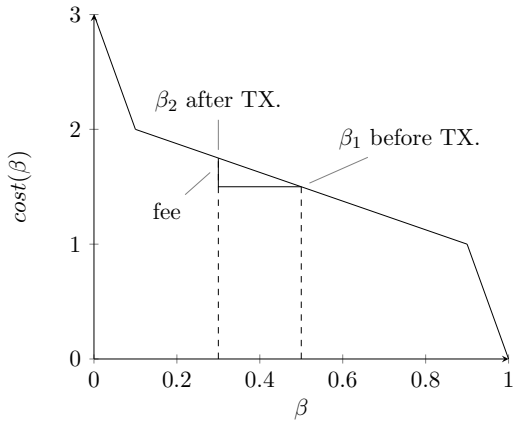


Figure 2.3: A possible cost function. Cost functions should be decreasing and if possible reward multiple small transactions as opposed to one big transaction.

any sender of transactions) or by A alone (which would make it hard for nodes to actually gain from routing transactions). To decide, how big a fee is, a cost function should be used. Every edge could have its own cost function. Although having a globally defined cost function would make the routing algorithm perform better (as edges would be balanced around 0.5, thus allowing for bigger transactions), having possibly different cost functions on all edges just shifts the resulting balance, since then the balance on one edge potentially also depends on the cost functions of all neighboring edges, i.e., edges that share one endpoint, because they could be a lot cheaper even if the path afterwards would be more expensive.

Definition 2.8. Given a directed graph $G = (V, 2 \times E)$, MSTs are generated using Dijkstra's algorithm. For each tree, an α_T, generated when generating the tree, can be used as a fee estimator when going down in the tree, which is generally cheaper, and α'_T for going up in the tree, which is generally expensive. The α's should be adjusted to the age of the tree, i.e., the longer a tree exists, the more we expect to pay using this tree.

Note: In this case, only paths from any node to its descendants (not vice versa) are optimal wrt. cost.
The MSTs have a time to live and have to be redone after some time. In our simulation, we used a global cost function similar to the one in Figure 2.3, but instead of taking the difference between the cost of β_1 and β_2,weset the cost as the function value of β_1, and then implemented the age by estimating a base $\alpha = 0.5$ and increasing this with the age a of the tree, i.e., $\alpha = 0.5 + \frac{a}{100}$ until, at age 100, $\alpha = \alpha' = 1.5$, which is the cost of an edge with half of all tokens on either side (cost of $\beta = 0.5$). Based on available memory on devices, some costs or capacities of edges in the neighborhood of node A can be stored at A to

increase performance. Also, depending on the rate with which fees are adjusted, an exact α can be stored.

Definition 2.9. When looking for a path, a *probe* is sent from S to D to find a path. The probe contains the destination address, as well as the path so far with the cost of each edge.

The size of the probes should be as small as possible. Assuming an address size per node A of number of trees $\times$ the size of one tree address, where one tree address consists of $10 + 32 + d_T(A) \times 16$ bits (Definition 2.5f.), and assuming an average depth of ca. 47 (this would be for huge networks), this would yield an average tree address size of just under 800 bits or 100 bytes. Multiplying this with the number of trees (e.g., 50) yields 5000 bytes for the destination address. With a maximum payload of about 65500 bytes per ip packet, we would still have more than 60000 bytes for the path, where one hop in the path can be stored with two tree addresses plus one byte for the cost, which amounts to around 200 bytes. So, even long paths can be stored in a probe while it still fits in one ip packet.

Definition 2.10. To decide on the next hop for a probe, a *scoring function* is used. For any pair of neighbor N and tree T_i, a score is computed based on the fee from A to its neighbor plus the estimated cost from the neighbor to D on T_i. $score = F_{AN} + \alpha'_{T_i} \times dist_{T_i}(N, K_{T_i}(N, D)) + \alpha_{T_i} \times dist_{T_i}(K_{T_i}(N, D), D)$.

At any point on the path, every tree could provide the optimal path. One could stop checking every neighbor tree pair once a pair is found where the neighbor is a ancestor of D, if the tree is reasonably new.

Definition 2.11. Given two nodes A and B, connected by an edge with fees F_{AB} from A to B and F_{BA} from B to A. The rate at which the two fees are updated is decided by the two endpoints A and B, but should lie between updating after every transaction and updating with every new tree generation.

Note: If no transaction passed the edge between two tree generation events, the fees do not have to be updated (but could be non the less). Setting the fees too low will send to many transactions over the edge, thus potentially gathering all tokens on one side, whereas setting the fee too high will prevent any transaction from taking that edge. The rate should be slow enough to prevent overreaction to small bulks of transactions in one direction, but fast enough to prevent the edge from being one sided.
Our proposition is to either update the fees every x transactions and every time a new tree is generated or every time some threshold is passed in the balance (e.g., when node A only holds 40%, 20%, 10%, etc. of the tokens). In the balancing simulations, the fees were updated after every transaction.

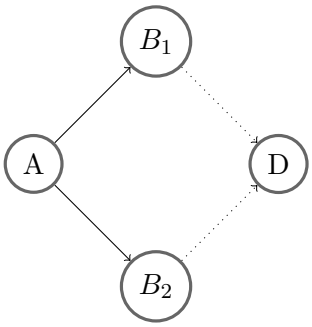


Figure 2.4: two possible paths from A to D
The fee F_{AB_1} influences the amount of traffic B_2 gets from A, since A chooses based on F_{AB_1} and F_{AB_2} when the paths from B_1 or B_2 respectively to D are similarly expensive.

Definition 2.12. The global minimum and maximum fees are called F_- and F_+ respectively. These are not explicitly given but depend on the network.

Note: it is perfectly legal to demand a negative fee to attract transactions or to have a really high fee to divert transactions.

Definition 2.13. An edge between nodes A and B is called *balanced*, if the ratio between tokens on either side of the edge are almost constant ($\beta \in [c - \epsilon, c + \epsilon]$ during some time frame Δt, for some small ϵ).

The balance of the edges depends on two factors. The first is, during some time frame Δt, all nodes might pay or receive some transactions. The resulting net gain/loss of some nodes can influence the balances of all edges. An example would be a network with only two nodes A and B, where in some time frame, A pays B some amount of tokens while B does not pay A. But, this is expected to happen gradually and can be countered by renegotiating the edges every once in a while. The second factor is the cost function(s) that is (are) used. If a global cost function is used (and we ignore net gain/loss), then the edges should balance around even distributions. With no globally used cost function, the edges will balance toward some other value, based on the cost functions used.

Lemma 2.14. *With adjusted fees after every x transactions, this algorithm provides load balancing, i.e., if there is a node A with neighbors B_1 and B_2, and multiple transactions going to some destination D, where both B_1 and B_2 achieve similar scores as next hop, A will distribute the transactions among both neighbors.*

Proof. For a pair S and D, assume that there is a node A on the path from S to D, with two neighbors B_1 and B_2 and potential paths from B_1 to D and from

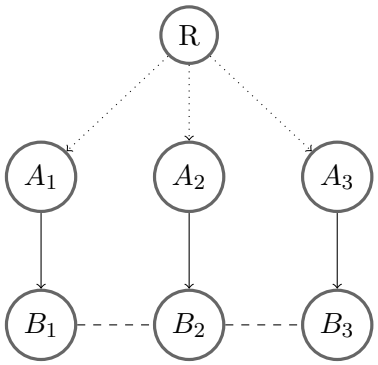


Figure 2.5: Upper bound on Cost

B_2 to D (see Figure 2.4). Depending on the cost from A to B_1 and from A to B_2, as well as the expected cost from B_1 to D and B_2 to D, a path is chosen and the fees are adapted. WLOG, assume B_1 was the cheaper next hop. B_1 will be chosen as next hop in future transactions as long as it is expected to be cheaper. As soon as the score of B_1 is worse than the score of B_2 because of the adapted fees, the transactions will be routed over B_2. As long as the two costs are similar, transactions will be equally distributed over both possible paths. If other transactions change the fees on these edges (e.g. by going from B_1 to A), A will route its transactions to rebalance the two edges. □

The order in which transactions arrive at a node A can influence the cost, if after every probe the cost is adapted. Assume, for some edge between nodes A and B, 10 transactions arrive almost at the same time, 5 going from A to B, the other 5 in the opposite direction. If the 5 transactions from A to B arrive at A before any of the other transactions arrived at B, then the last transaction going from A to B will have a much higher fee than the first one. Whereas, if the transactions arrive alternately, all fees will be relatively low. This implies that waiting on other transactions might reduce cost and improve the performance, but the wait time should be really small, as even a small wait time could lead to big delay if the path is long.

Lemma 2.15. *Given a start node S and goal node D on a payment network with newly generated MSTs T_i and lowest common ancestor $K = K_{T_i}(S, D)$. Assume the optimal path wrt. cost has cost optcost. Then the upper bound for the cost of the found path is* $\min_i(optcost + (d_{T_i}(S) - d_{T_i}(K)) \times 2 \times F_+)$.

Proof. Let $cost(A)$ be the cost to go from root R_i to node A on a single tree T_i. If the optimal path is entirely on a given tree T_i, then the optimal path is found, as any path lying entirely on a tree is known from the start. Assume the optimal path is not entirely on a tree. Let there be nodes B_1, B_2, B_3 and

A_1, A_2, A_3 as shown in Figure 2.5, A_i being the parent of B_i in this tree. Based on Definition 2.8 we know that $cost(B_1) = cost(A_1) + F_{A_1B_1}$ and $cost(B_1) \leq cost(B_2) + F_{B_2B_1}$, i.e., going from R to any B_i via the corresponding A_i is cheaper than (or equally expensive as) via any other B_j. From this it follows that $F_{B_iB_j} \geq cost(B_j) - cost(B_i)$. Let $S = B_1$ and $D = B_3$ and the optimal path from B_1 to B_3 be the direct path via B_2. The obvious found path is from B_1 to B_3 via K. Let us call the optimal path p_1 and the path via K p_2. Then $cost(p_1) = F_{B_1B_2} + F_{B_2B_3} \geq cost(B_3) - cost(B_2) + cost(B_2) - cost(B_1) = cost(B_3) - cost(B_1)$. The cost of the second path can be approximated by $cost(p_2) \leq (d(B_1) - d(K)) \times F_+ + (cost(B_3) - cost(K))$ (going from B_1 to K plus going from K to B_3). Since $cost(p_1) = optcost$, the worst case additional cost of p_2 is $p_2 - p_1$ which is $(d(B_1) - d(K)) \times F_+ + cost(B_1) - cost(K)$. Because no matter the cost function, all fees are less or equal to F_+, this is less or equal to $2 \times (d(B_1) - d(K)) \times F_+$ □

Note: In our simulations, the factor $2 \times F_+$ was actually equal to 3 (F_+ in our cost function, because $cost(\beta) + (cost(1 - \beta)) = 3 \forall \beta$ because of our cost function. Also, our simulations showed that the average distance of all nodes to the root nodes was around 4.5 with 100'000 nodes and 10 trees, so even for big graphs the average distance to any root will be < 10 with high probability. Obviously, for MSTs that are not newly generated, additional costs may arise.

2.3 Optimization

In this section we would like to present some improvements to the algorithm to increase the performance and resilience against malicious or non responsive nodes in the network.

2.3.1 Fake Tokens

The first addition are fake tokens to increase the capacity of edges if both endpoints of an edge trust each other sufficiently.

Definition 2.16. Given an edge in the network between nodes A and B. The amount of real tokens on this edge is limited by the funds both parties were willing to invest in this edge. The *fake tokens* can be used just like real tokens, without the need to actually invest them (or even owning any funds). If one endpoint wants to add fake tokens to the edge, the other endpoint has to first accept this, i.e. A gives B the possibility to spend more than he invested. This signals also the willingness to lose funds if the counterpart acts malicious.

Note: With fake tokens, edges have more capacity, thus being able to route more transactions, gain more fees and being more resilient to bulk transactions in one direction. Also, nodes can create edges between them with only fake tokens

and no real tokens, which they would not even need to confirm with on-chain transactions.

Lemma 2.17. *Given some edge between nodes A and B using fake tokens. Tearing down the edge with fake tokens out of balance only affects the funds of A and B.*

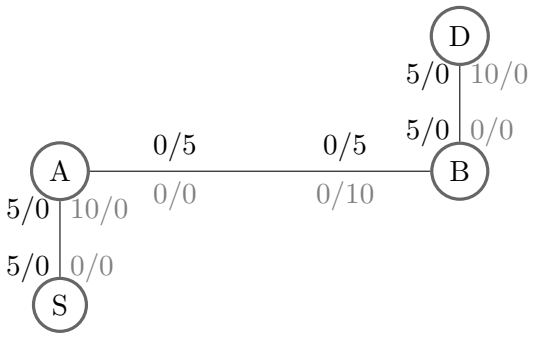


Figure 2.6: Before (black) and after (gray) a transaction from S to D of size 5. The tokens are written as (tokens/fake tokens) as stored on each side of the edge.

Proof. As shown in Figure 2.6, given a transaction of 5 tokens from node S to node D over the edge e between nodes A and B, any fake token used on e is first translated from real to fake token at A and then from fake to real token at B. S and D have their desired balance (S spent its tokens, D received the tokens), but A has 5 real tokens actually belonging to B. □

Fake tokens can be viewed as loans given between endpoints of an edge to increase the edge's performance. Having more tokens on the edge allows the two nodes to route more transactions over the edge which increases their income. Also, edges with more tokens (fake or not) can generally be cheaper to attract more transactions since transactions do not affect the balance that much.

2.3.2 Probe Splits

The next improvement concerns probes. Assume a probe is sent along the optimal path based on cost, but one of the nodes on the path is offline, or worse, malicious. This is why we need multiple probes that probe different paths.

Definition 2.18. Each probe that is sent out is sent with a *split potential* $sp \in \mathbb{N}$. At any point, if the split potential $sp > 1, sp \in \mathbb{N}$, a probe can be split randomly in two probes with split potentials $sp_1 + sp_2 = sp$, where $sp_1, sp_2 > 0$.

In our simulation, we allowed up to 4 probes. Results might be improved if more probes are allowed, but computational cost rises too. More probes imply more found paths but also a reduction in actual throughput, since only one transaction is sent no matter how many probes are sent to find the best path. The original sender can choose the best one from the resulting paths. This decision can be based on total cost, path length or any other factor. Also, when finding two paths p_1 and p_2, if they share at least one node, a combination of the paths will also yield a valid path.

Definition 2.19. The *split chance* depends on the estimated distance to the goal and the remaining split potential. If the distance is short, the split chance is reduced, while a big remaining split potential increases the chance of a split.

Reducing the chance of a split based on the estimated distance helps to prevent cases where the distance between S and D in hops is small, e.g. 2, but still 8 splits are done. In such cases, one or two splits at most usually find the best path.

2.3.3 Reverse Probes

Definition 2.20. Similar to the normal probe, sent from S to D to find the best path from S to D, a *Reverse Probe* is sent from D to S to find the best path from S to D. Since S decides on the actual path used for the transaction, one message from D to S containing the found paths can be omitted.

This would also mean that the scoring function should be adapted. First of all, the considered fee is not F_{AB} but F_{BA}, since the transaction would traverse the edge from B to A. Secondly, the remaining part of the scoring function would have to swap α for α', since going down and going up in the tree would be the other way around when probing reversely.

2.3.4 Resilience

Another big optimization is making probes (and the routing scheme all together) tamper proof. For this, at least two parts of the scheme should use signatures:

- The address generation, where each node signs the address propositions for its children and sends the signed addresses for all ancestors along with an address proposition.
- The fee on each edge in the found path, as stored in the probe.

If the address propositions are not signed and passed along with new address propositions, it is possible for malicious nodes to choose their address, which

could lead to transactions being routed in an inefficient way or not at all.
Also, if the fees that are written in the probes are not signed or protected in some other way, malicious nodes could change these fees to whatever value they want.

2.3.5 Score Computation

Finally, concerning the computation of the scoring function, a node does not have to compute the score for every possible tree-neighbor pair.

Lemma 2.21. *Given a graph $G = (V, E)$ with one MST T_i and a node A with neighbors $N = N_{T_i} + N_{NT_i}$, where N_{T_i} are A's neighbors on T_i and N_{NT_i} A's neighbors in G that are not in N_{T_i}. For any node $D \neq A$, with $dist(A, D)$ known, there is exactly one node $n \in N_T$ where $dist(n, D) = dist(A, D) - 1$, for all others, the distance is higher than A's distance to D.*

Proof. The MST T_i with root R_i can be rewritten as a tree with root D. All the edges and distances stay the same. Now the depth of a node gives the distance of this node to D (as opposed to the distance to R_i before). Since any node in a tree can have at most one parent (exactly one parent if the node is not root), A can have only one parent. So, the shortest path from A to D on this tree is via A's parent. □

Given $t = (T_1...T_t)$ trees and neighbors $N = N_{T_i} + N_{NT_i}$, the number of computed scores is $\sum_{i=1}^{t} (|N_{NT_i}| + 1)$

Chapter 3

Simulation

We simulated three different versions of our algorithm. In the first simulation, we looked at general graphs, where edges have unlimited tokens and all fees are equal to 1, thus the goal was to find the shortest path. In the second simulation, we calculated the resulting cost on a snapshot of a payment channel network with a global cost function similar to the one shown in Figure 2.3, but with fees based on $cost(\beta_1) \in [0, 3]$ and not the difference of the two costs of betas (pre- and post- transaction), because the transactions did not yet have sizes. We did not adjust the tokens or fees after a transaction. The goal was to see if, with all newly generated trees, the algorithm could find paths with cost as close to the optimal as possible. In the third simulation, we adapted the fees after every transaction and initialized the trees with ages, i.e., potentially non optimal trees. The goal there was to demonstrate that the algorithm balances the edges even if some trees are not optimal.
The simulation had two stages. Firstly, we did the simulation with networks of 1000 nodes where one node was flagged as offline (it did take part in MST generation, but not in transaction routing). Secondly, we did the same simulations again with networks of 100'000 nodes, where 31 nodes were offline in the distance simulation, 48 in the cost simulation and 82 in the balancing simulation. Then we repeated the three simulations for both sizes of networks ten times with increasing numbers of trees $T = 2, 4, ..., 20$. The trees were generated with the network and not changed when increasing the number of trees (i.e., the two trees used when $T = 2$ were the same trees as the first two when $T = 20$). For each pair of simulation and network size, a new network was generated. This resulted in some networks of the same size in terms of nodes having different amounts of edges. For each simulation, 100'000 random transactions were done.

3.1 Metrics

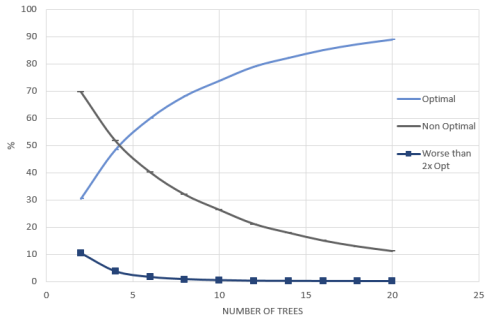
The collected metrics were the average cost of found paths (or distance in the first simulation), the average optimal cost (distance) and the percentage of transactions that cost more than the optimal cost, more than twice the optimal cost,

more than three times and more than five times the optimal (although the percentage of transactions that cost over five times the optimum was rarely more than 1%, usually much less. Additionally, we tracked the standard deviation of the balances of all edges in the network for the balancing simulation. Since every edge actually consists of two edges, from A to B and vice versa, the average balance is always 0.5, since any balance from A to B is mirrored by the edge in reverse direction. We also tracked various other metrics such as number of paths found or number of splits done (but these stayed almost constant), the amount of edges in the taken path that had a balance $\beta > 0.5$, i.e., that did not help balancing the network (which was less than 0.002% in the worst case), a tree rating, where the rating was higher if the tree influenced more decisions in the scoring process (if all trees had the same age, all had almost the same rating, else the rating was proportional to the age). In the distance simulation, we observed a correlation between a node being close to a root and the same node observing more traffic, but, in the later two simulations, no such correlation was observable (mainly because in the second and third simulation, going down in a tree was preferable, so transactions tried to avoid going towards a root).

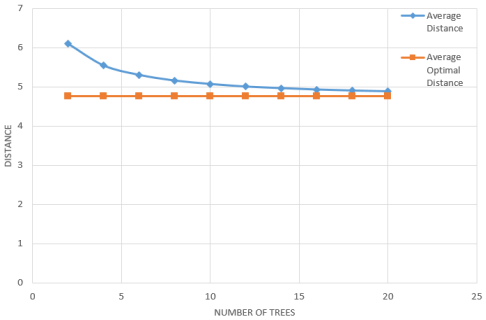
3.2 Distance Simulation

Figure 3.1 shows the results of the distance simulation with 1000 nodes. In Figure 3.1a it is shown that, even with only a few trees, the amount of non optimal transaction might be big, but almost all of them are less than two times the optimal. Using 20 trees we even achieve a rate of non optimal paths of slightly more than 10%. As can be seen in Figure 3.1b, the average Distance of the found paths is almost the same as the average optimal distance. This shows that even if the found path is not optimal, the additional cost is, on average, low. It also shows that increasing the number of trees only improves performance up to some point.
For the distance simulation with 100'000 nodes, the results in Figure 3.1 show that even if the network is one hundred times the size of the previous simulation, the results are not much worse. In Figure 3.1d, the average extra distance is almost equal to 1, implying that, on average, the found path is only one hop longer than the optimal. In Figure 3.1c, the amount of non optimal transactions is worse than before, but we expect the results to improve when even more trees are used. Also, the amount of transactions that are worse than twice the optimum is less than 1% when using only 20 trees.

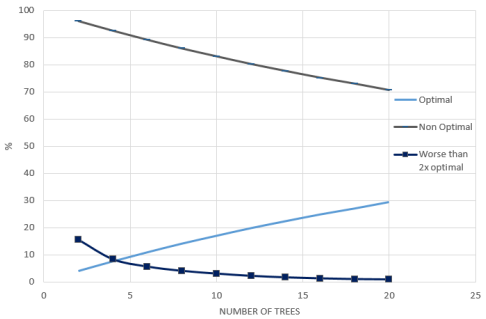
Figure 3.1: 1^{st} Simulation: Distance



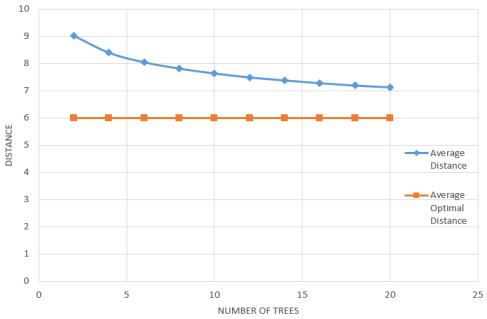
(a) Percentage of optimal and non optimal transactions depending on the number of trees with 1000 nodes.



(b) Distances of found paths vs. optimal distances with 1000 nodes



(c) Percentage of optimal and non optimal transactions depending on the number of trees with 100'000 nodes.

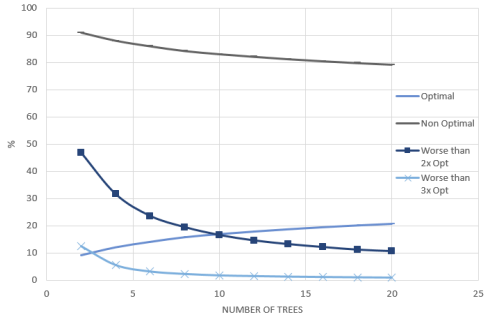


(d) Distances of found paths vs. optimal distances with 100'000 nodes

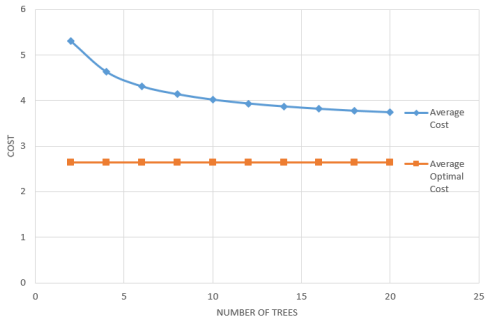
3.3 Cost Simulation

In the second simulation, edges were generated with a random capacity $c \in [1, 10]$ and a random balance $\beta \in [0, 1]$, which split the capacity (or total amount of tokens) on both endpoints of the edge. In Figure 3.2 we can see the results of the second simulation. As was expected, the percentage of non optimal paths (Figure 3.2a) was higher than in the distance simulation. This is mainly because in the cost simulation, usually only one optimal path exists, whereas in the distance simulation, because all edges have cost one, potentially multiple paths of optimal length exist, so the probability to find at least one of those paths is higher.
The cost simulation with 100'000 nodes (Figure 3.2) achieved comparable results to the 1000 node simulation. Again, most transactions were not optimal, but most cost less than twice the optimum, and only 1% cost more than three times the optimum (at 20 trees). Also, the average cost was at 169% of the average optimal cost when using 20 trees (as opposed to 141% of the optimal cost in

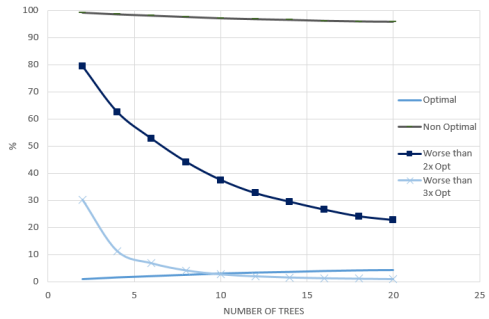
Figure 3.2: 2^{nd} Simulation: Cost



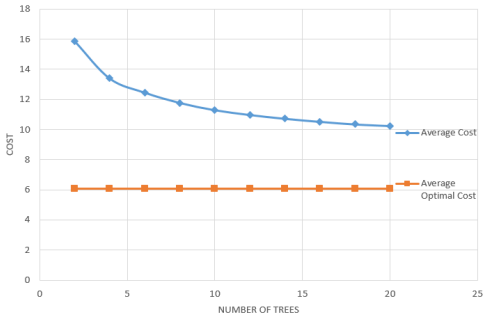
(a) Percentage of optimal and non optimal transactions depending on the number of trees with 1000 nodes.



(b) Cost of found paths vs. optimal cost with 1000 nodes.



(c) Percentage of optimal and non optimal transactions depending on the number of trees with 100'000 nodes.



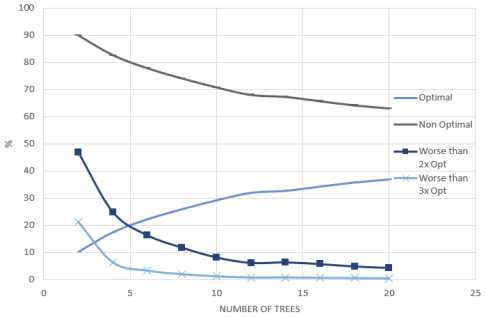
(d) Cost of found paths vs. optimal cost with 100'000 nodes.

the smaller simulation). Similar to the distance simulation, the performance can be expected to improve if more than 20 trees are used, although the increase in performance per added tree will soon be small.

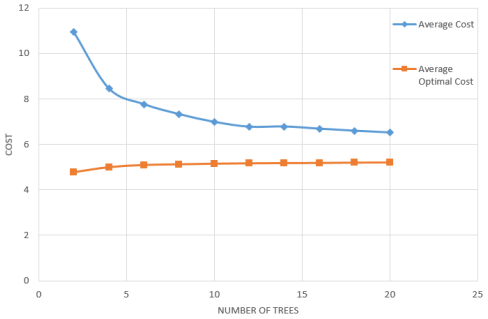
3.4 Balancing Simulation

The third simulation is closest to an actual working network. When generating the trees, a random age between 1 and 100 was assigned to each tree (mean age: 49.3, sd: 28.3). Then, for each tree generation, a new network was generated where each edge's balance β was changed randomly to a value $\beta' \in [\beta - x, \beta + x]$, where x depends on the age of the tree and $\beta' \in [0, 1]$. On this new, distorted network a MST was generated which then was copied into the actual network, thus generating potentially non optimal MSTs in the original network. During the routing process, the age of the MST was accounted for when computing the score of a neighbor-tree-pair (as described in Definition 2.8f.). While in previous

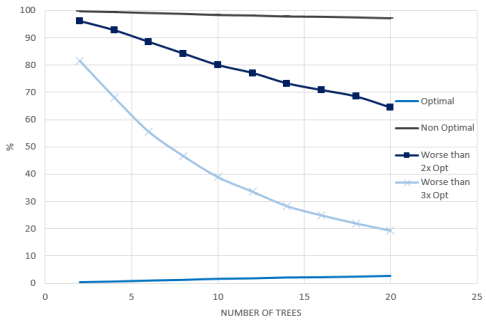
Figure 3.3: 3^{rd} Simulation: Balancing



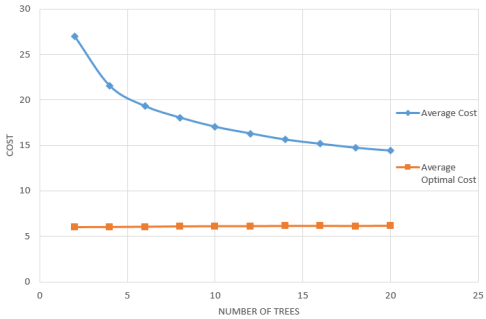
(a) Percentage of optimal and non optimal transactions depending on the number of trees with 1000 nodes.



(b) Cost of found paths vs. optimal cost with 1000 nodes.



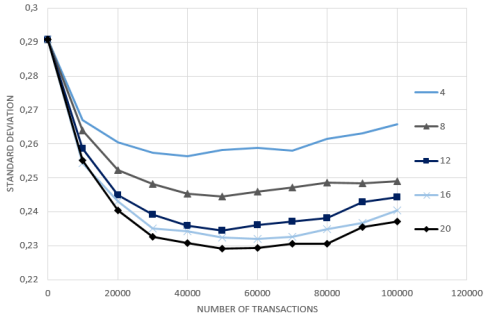
(c) Percentage of optimal and non optimal transactions depending on the number of trees with 100'000 nodes.



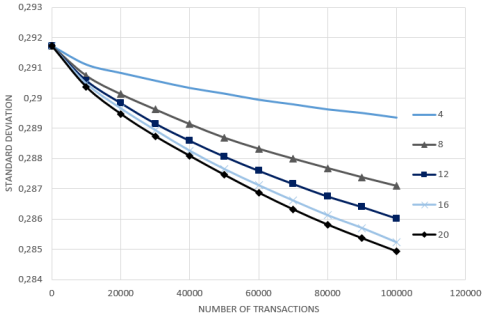
(d) Cost of found paths vs. optimal cost with 100'000 nodes.

simulations all trees equally influenced the routing decisions, in this simulation the newest tree influenced most decisions, while the oldest tree influenced hardly any decision. Figure 3.3 shows the resulting average costs and optimal costs and the percentage of non optimal transactions. The results of the balancing simulation on a 100'000 node network are shown in Figure 3.3. This simulation was by far the worst performing simulation, achieving only an average cost of 234% of the optimal average cost using 20 trees, thus being the only simulation that had an average of more than two times the average optimal cost.
In Figure 3.4, the change in standard deviation over all edges is shown for five of the ten tests ($T = 4, 8, ..., 20$) in the small network. It should be noted that, even with only four trees, the standard deviation after 100000 transactions is still lower than it was in the beginning. Looking at the changes in the bigger network (Figure 3.4b), the standard deviation changes a lot slower than before, due to more edges existing in the network and therefore each transaction having a smaller influence on the total standard deviation. It should be noted that, even for the case with only 4 trees, the standard deviation was still sinking after 100'000 transactions.

Figure 3.4: Standard Deviation in 3^{rd} Simulation



(a) Change of Standard Deviation of edge balances per 100'000 transactions when using $t = 4, 8, ..., 20$ trees.



(b) Change of Standard Deviation of edge balances per 100'000 transactions when using $t = 4, 8, ..., 20$ trees.

Chapter 4

Related Work

There have already been some suggestions on how to tackle this routing problem, but many of those did not emphasize enough the special nature of edges in a payment channel network as opposed to more common graphs.
For example, the global beacon-/ or landmark-approach[8] was suggested as an option to handle the routing problem in the Lightning Network. The problem with this approach is that for a transaction from some node A to some node B (that do not know the path to each other), each possible path passes a beacon node, which might result in a sizable detour. Also, since all traffic will be routed over at least one beacon node, edges close to the beacon node suffer from a potentially high traffic load, which could throw them off balance.
A second beacon approach [9] was made using local beacons, i.e., each node chooses randomly, e.g., $\sqrt{n}$ personal beacon nodes. When nodes A and B need to route a transaction, they just exchange their beacon sets and hope that they have a beacon in common. While this approach might remove the high traffic load for the global beacon nodes, it suffers from questionable scalability and does not address the case when two nodes do not share a beacon.
Another approach that, at least partly, uses beacons, is Flare[10]. This algorithm uses a hybrid of proactive and reactive protocols. First, it stores some amount of edges in the network (generally the neighborhood of that node), as well as paths to some beacon nodes. Then, as soon as a transaction needs to be routed, it dynamically scores the different possible paths, after probing them, based on, e.g., the fees on these paths. It then routes along the best path. If, for some reason, the path does not work, the second best path is tried and so on. If no path works, beacons are used to find new paths. Our improvement on this method is that we do not rely on storing parts of the network on every node, mainly because this information might be volatile, but also to save on memory consumption. Additionally, for nodes that are relatively far away from each other, this algorithm still has the problems that the previous attempts had, that potentially big detours arise based on the beacon approach for long distance transactions.

Chapter 5

Discussion

When trying to tackle the routing problem in micropayment channel networks, being able to balance the edges in the network is an important factor that should not be neglected. The three stated goals for a routing algorithm in these kind of networks, finding a path, balancing the network and minimizing the cost, summarize the results of the simulations.We have shown that our algorithm can find a path of nearly minimal length in the first simulation, that the algorithm can minimize the cost in the second and third simulation and that it balances the network edges if new trees are generated fast enough.
Although the algorithm did not perform as good in the second and third simulation as it did in the first one, reaching an average cost of less than 200% of the average optimal (except the last simulation, which was under 300%) is still acceptable.We did an additional balancing simulation on a different network with 100'000 nodes and 50 trees to see if better results are possible. While the average cost was 188% of the average optimal cost, the percentage of transactions that cost more than twice the optimum sank to 39%, while only 4.6% cost more than three times the optimum.
While this algorithm was designed with payment channel networks in mind, it does work well with other routing problems. If balancing of edges is not a concern, then the algorithm can be used after generating a set of trees with no need to frequently generate new trees (unless the network itself changes). Then performance could be optimized by using an ideal set of roots. Ideas on how to find these roots can be adapted from Thorup [11]. One should focus on distributing roots such that all nodes have a small distance to at least one root.
Interestingly, in the 1000 node networks, the second simulation did worse in terms of average cost than the third, even though the third simulation was expected to do worse because of the change of fees. This is most likely due to the different amount of edges in the networks, as the networks were randomly generated, the network in the second simulation had an average degree of 18 while the average degree in the third simulation was around 7. So, because of more edges in the network, it was harder to find the optimal path with only a few trees.
Looking at the results of the balancing simulation in figure 3.4, it is interesting to see that the standard deviation (SD) of all edges in the network is still lower

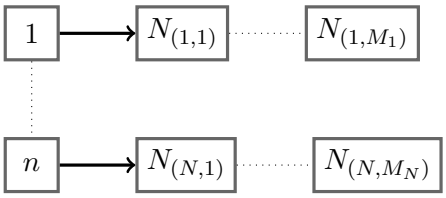


Figure 5.1: Lists of neighbors

than the original SD, even after 100'000 transactions. However, the original balances were generated randomly, so a fast decrease in the first few transactions is to be expected. Assuming an average usage of 100 transactions per second, this means that even with only 2 MSTs, the network will be balanced after 16.6 minutes or 100'000 transactions. Than means that it suffices to generate a new tree every 10 minutes or every new block on the blockchain. In the large network simulation, no matter the amount of used trees, after 100'000 transaction, the standard deviation was still sinking. Since all transactions were done randomly, i.e., start and goal node were decided randomly, this was to be expected. However, if used in a real environment, the transactions will probably not be randomly distributed, but rather mostly localized, i.e., transactions will tend to repeat themselves or at least transactions involving one specific node will include a limited set of other nodes in most cases (e.g., the same people shopping at the same store).
Also, because the results on average cost in the third simulation were just as good as the results of the second simulation, we can conclude that older trees do not limit the functionality of the algorithm. This, together with the change in standard deviation, implies that having more trees (up to some point) will increase the performance of the algorithm, even if the trees are old and the information stored in them is deprecated.

When comparing our approach to storing the whole graph at every node, it is apparent that storing the whole graph will produce better paths, as the whole information is available at every node. However, our algorithm uses less memory. Let us look at three different ways to store the whole network. The simplest one is using a bitmap to store, which nodes are connected to which (one bit is enough in the distance approach, for the cost approach it would need more than one bit to not only store an existing edge, but also the cost). This leads to a usage of at least n^2 bits. The second approach would be to store all edges as a list of neighbors (Figure 5.1), where for every node all neighbors are stored in a list. This approach needs for the neighbors alone at least $2 \times E \times log_2(n)$ bits. The third approach would be storing all edges as pairs (A,B), which uses $E \times 2 \times log_2(n)$ bits.
Say we have a network with 100'000 nodes and 50 trees. Also, assume that the average depth of any node in any tree is 60 (which is an overestimation), then the memory consumption of one node address is the number of trees times the

tree address size, or $50 \times (10 + 32 + 60 \times 16) = 50'100 \approx \frac{n}{2}$ bits. The average memory consumption is therefore the addresses of the node plus the addresses of all neighbors, $\frac{2E}{n} \times \frac{n}{2} = E$. This beats the other solutions by a factor of $2log_2(n)$. This is obviously an overestimation of memory consumption of our algorithm, since an average depth of 60 is really high and also 16 bits per level of depth is a lot. Still it can be seen that the memory consumption is lower than storing the whole graph.

However, there are limitations to our algorithm. Firstly, if the average degree of the nodes is high, that is, the network contains many edges, the algorithm is less effective. The performance could be increased by adding more trees, but this would also increase the computational cost, message size and memory usage and therefore is only useful up to some point. A second problem is that, with malicious intent and enough nodes under control, an attacker might damage the network by, e.g., having a set of nodes that try to separate two sets of nodes such that any transaction from a node in one set to any node in another set will pass over at least one malicious node and is then routed for a high cost to a different malicious node before being routed to the destination.

There are still some problems left which could be tackled in future work. Some examples would be increasing the security, that is, adding more signatures and other control mechanisms to prevent attacks like the one described above, finding an addressing scheme that uses less memory than our approach while still providing the information necessary without decoding, using different probes such as slower probes, which could allow nodes to store or hold probes for a very short amount of time to increase the balancing and decrease cost if another transaction going in opposite direction is found or building completely trust based networks (i.e., only edges with fake tokens) and the resulting possibilities and problems.

Chapter 6

Conclusion

In this thesis we looked at newly emerging routing problems in micropayment channel networks. Because of the special nature of these networks, mainly the volatility of the edges, the usual routing algorithms do not work as intended. If the balancing factor is not accounted for, algorithms run the risk of rendering edges one sided and thus increasing the difficulty to find appropriate paths. Our algorithm showed that it is capable of solving the routing problem in general graphs as well as in these special networks, while the memory usage and computational cost scales well with increasing sizes of networks. It really shines when working with big networks with a relatively small number of connections. While for small networks, storing a routing table still outperforms our approach (by using a lot more memory), this solution is not possible for large networks, because the table would be too big and because of the volatile character of the edges, the number of status update messages would congest the channels.

Bibliography

[1] Trillo, M.: Stress test prepares visanet for the most wonderful time of the year (2013)

[2] Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün, E.: On scaling decentralized blockchains. In: 3rd Workshop on Bitcoin Research. (2016)

[3] Decker, C., Wattenhofer, R.: Information propagation in the bitcoin network. In: 13th IEEE International Conference on Peer-to-Peer Computing. (September 2013)

[4] Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)

[5] Russell, R.: Reaching the ground with lightning (2015) https://github.com/ElementsProject/lightning/blob/master/doc/deployable-lightning.pdf.

[6] Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Symposium on Stabilization, Safety, and Security of Distributed Systems. (2016)

[7] Burchert, C., Decker, C., Wattenhofer, R.: Scalable Funding of Bitcoin Micropayment Channel Networks. In: 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Boston, Massachusetts, USA. (November 2017)

[8] Russel, R.: Ionization protocol: flood routing. in: Lightning network development discussion. (2015)

[9] Bairn, A.: Ionization protocol: flood routing. in: Lightning network development discussion. (2015)

[10] Prihodko, P., Zhigulin, S., Sahno, M., Ostrovskiy, A., Osuntokun, O.: Flare: An approach to routing in lightning network. (2016)

[11] Thorup, M., Zwick, U.: Compact routing schemes. In: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures. (2001) 1–10