Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Exploring Centralized Payment Network Topology

Semester Thesis

Remo Glauser

glauserr@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Georgia Avarikioti, Tejaswi Nadahalli
Prof. Dr. Roger Wattenhofer

November 29, 2018

Acknowledgements

At this point I would like to thank all the people who supported this project and shared their expertise. The project could be realized due to the Distributed Computing Group (DISCO) headed by Prof. Dr. Roger Wattenhofer. My Supervisors Georgia Avarikioti and Tejaswi Nadahalli provided with valuable advice in off-chain payment channels, graph theory and algorithm design.

Abstract

Off-chain payment channels provide a promising solution to solve the blockchain scalability problem. But, the concept suffers from the inherent weakness of capital-inefficiency and user unfriendliness since capital is temporally locked. As a solution a Payment Service Provider (PSP) has been proposed, which takes over the capital deposits. In addition, the PSP defines the network structure and pays the costs of opening and closing channels. Whereas a fee is paid by the users in return. A PSP has a strong incentive to create the optimal network structure such that its profit is maximized, which is a NP-hard problem as shown by Avarikioti et al. [1]. This work focus on designing an approximation scheme to maximize PSP's profit. As a preliminary step, trees structures is only considered and an approximation algorithm to minimize the required capital is presented. We show in an analysis that this algorithm provides a tree structure which requires in average 12 % less capital than a star structure. In a further step, the problem is extended to connected graphs by allowing cycles and by introducing a cost function. We show an algorithm to minimize the costs. Finally, the problem of profit maximization is discussed. We show approaches of profit maximization algorithms for the offline and online case.

Contents

Chapter 1

Introduction

The high potential of distributed ledger technologies (DLT's), like blockchain, could not yet be fully exploited. Existing throughput problem remains as a fundamental impediment [2]. For instance, the two popular permission-less blockchains Bitcoin [3] and Ethereum [4] are able to process only 3-7 and 7-15 transactions per second, respectively. Whereas, the networks of credit card companies (e.g. Visa, MasterCard, etc.) are able to process some thousands of transactions per second. Two approaches to overcome the issues have been presented. In Sharding [5] [6], the blockchain is split up in multiple chains, which run in parallel in order to increase the throughput. In a Payment Network [7], transactions are processed through off-chain channels (without adding the transaction to the blockchain). An underlying blockchain is used to open and to close channels and as security guarantee. Payment Networks seem to be the more promising solution, since data volume can be taken away completely from the blockchain.
Lightning [8] and Raiden [9] are implementations of a payment network on top of the Bitcoin network and Ethereum network, respectively. Such networks allow the users to create peer-to-peer payment channels between each other, which enables to transfer money without any fees. Costs only occurs when opening and closing a channel, since in each case, a single transaction to the underlying blockchain (e.g. Bitcoin, Ethereum, etc.) is required, which comes with a transaction fee. By creating a channel the user needs to assign capital on the channel, which is not available for him during channel's existence. The locked capital is used as a security deposit. In case multiple users connect to each other, a network is created which enables to route payments from user to user through the network instead of creating new costly channels.
An user created payment network is capital inefficient, since not the optimal network structure is created such that minimal capital is required to process all payments in the network. In addition, the number of open payment channels of an user is limited to its total capital, since capital has to be assigned on its channels. These issues are addressed by introducing a Payment Service Provider (PSP) [7], which defines the network structure while taking over the capital deposit. Instead of creating a channel between two users, a three-party channel between the two users and the PSP is created, whereby the channel is funded

only by the PSP. The PSP deposits capital to open channels and pays the fees on the blockchain. In return, it gets a tiny monetary compensation for its service.
The PSP has strong incentive to minimize the required capital (capital to create the channels) and to maximize the profit. Avarikioti el al. [1] discussed the problems a PSP is faced from an algorithmic perspective. It could be shown that maximizing the profit is NP-hard given the capital assignments. Further, it has been proven that a star network structure yields a 2-approximate of the optimal capital for minimizing the capital. Moreover, it has been proven that not even for a single channel, a deterministic competitive algorithm for adaptive adversaries exists in the online case such that the profit is maximized given a capital constraint.
The contribution of these works is strongly related to the findings in [Avarikioti el al.] [1]. The focus of this project lies on designing an approximation algorithm to maximize the profit. Firstly, considering trees structures only, algorithmic solutions to minimize the required capital are discussed and a probabilistic algorithm, which provides a closer average approximation than a star structure, is presented. Further, the problem is extended to connected graphs by allowing cycles. A cost function is introduced to model the number of channels together with the required capital. It is focused on minimizing the costs. Finally, approximation schemes to maximize the profit are discussed for the offline and online case.

Model - Payment Network: It is modelled as a graph $G(V, E, C_{\mathrm{L}}, C_{\mathrm{R}})$, where vertex v can be a sender or a receiver of a transaction, where edge e is a channel between two vertices, and where $c_{e_{\mathrm{L}}}$ and $c_{e_{\mathrm{R}}}$ are the capital assignment of edge e ($v_{\mathrm{i}} - v_{\mathrm{j}}$ with $1 < i < j < n$, for n vertices) on the left side and on the right side, respectively.
A transaction $t_{\mathrm{i}} = (s_{\mathrm{i}}, r_{\mathrm{i}}, a_{\mathrm{i}})$ with $0 < i < k$ consist of a sender s_{i}, a receiver r_{i} and a payment amount a_{i}, can be routed arbitrary through the network. A strategy $S_{\mathrm{e}} = \{-1, 0, 1\}^n$ defines if a transaction is routed on edge e from left to right (1), right to left (-1) or not routed through e (0).
The model contains the characteristic that capital $c_{e_{\mathrm{i}}}$ is temporally locked on the edge e_{i} as long as e_{i} exists. Moreover, the capital $c_{e_{\mathrm{i}}}$ can only be moved along e_{i} from left to right, or right to left, since the model does not allow capital to be moved from e_{i} to any other edge e_{j} through a vertex. The only way to move capital from e_{i} to e_{j} is to unlock $c_{e_{\mathrm{i}}}$ by removing edge e_{i} (closing the channel) and assign capital on e_{j} (opening an additional channel), but to close as well as to open a channel cause a fee φ. The fee φ represents the variable fee for executing a transaction on the Bitcoin network.

Definition 1.1 (Cost)**.** Interest are obtained for capital in the economy. Since, the capital C is locked in the network, the interest on the locked capital are a part of the costs besides the number of edges m:

$$Co = \varphi m + \alpha C = m + \alpha C$$

, with a interest rate $\alpha \in [0, 1]$ and with φ assumed equal to 1.

Definition 1.2 (Profit)**.** the profit P obtained by processing k transactions is defined as:

$$P = \epsilon k - Co = k - (m + \alpha C)$$

, with a constant earning per transaction $\epsilon = 1$

Problem 1.3 (Tree Design for All Transactions)**.** Given, a set of transactions Tx, determine a tree T which requires the minimum required capital (optimal capital) to process all transactions.

Problem 1.4 (Graph Design for All Transactions)**.** Given, a set of transactions Tx, determine a graph G which minimize the costs to process all transactions.

Problem 1.5 (Transaction Selection to Maximize Profit)**.** Given, a set of transactions Tx and a capital limit $\mathrm{C}_{\text{limit}}$, determine a tree T, a set of chosen transactions T_{chosen} and a capital assignment C, such that the profit is maximized.

CHAPTER 2

Optimal Tree Approximation

In this chapter, Problem 1.3 - Tree Design for All Transactions is studied. The problem constrains the network topology to a tree structure, thus, the profit becomes a function of the number of nodes n and the required capital C, since the number of edges becomes well-defined with $m = n - 1$ and all transactions are processed. The profit P is equal to:

$$P = k - (m + \alpha C) = k - (n - 1 + C)$$

, without loss of generality α is assumed to be equal to 1, since it has no effect on the optimization problem. For a network of n nodes and a transaction set of length k, the profit becomes maximal by minimizing the required capital C, i.e by having the *optimal tree* as network structure. Throughout this chapter, the problem of calculating the optimal tree is discussed.

An algorithm is required to calculate the capital demand of a given network structure. At first such an algorithm is briefly introduced. Subsequently, challenges of designing an algorithmic solution of the problem are discussed in the first Section. Further in Section 2.2 an algorithm based on the largest payment amounts is presented and analyzed on its performance. And finally, in Section 2.3 a probabilistic algorithm, which provides a closer average approximation than a star structure is presented.

Calculate required capital for a given graph (Alg. 1): It is calculated by forwarding the transaction from the sender to the receiver through the network. While forwarding a transaction from one node to the other, the capital on the edge between them is updated according to the payment amount. In the worst case, when all nodes lie on one line, it is going to take $(n - 1)$ times until a single transaction is forwarded through the graph. It follows that to calculate the required capital for a graph, it takes

$$\mathcal{O}_{\text{cap}} = k(n - 1) \tag{2.1}$$

$$\Omega_{\text{cap}} = k$$

iterations, with k the number of transactions and n the number of nodes.

Algorithm 1: Calculate required capital for a given graph

```
input  : Graph $G = (V, E)$
input  : A set of transactions $t_{\mathrm{i}} = (s_{\mathrm{i}}, r_{\mathrm{i}}, a_{\mathrm{i}})$
output: required capital $C$ to process all transactions on $G$
$C = 0$
for each edge $e$ in $G$ do
    for direction $d$ in {left (l), right (r)} do
        On edge $e$ in direction $d$, capital $c = 0$
        On edge $e$ in direction $d$, maximum capital $c_{\mathrm{max}} = 0$
    end
end
for each transaction $t_i$ do
    /* sender, receiver, payment amount */
    $s_{\mathrm{i}}, r_{\mathrm{i}}, a_{\mathrm{i}} = t_{\mathrm{i}}$
    $c_{\mathrm{i}} = s_{\mathrm{i}}$
    while $c_i$ != $r_i$ do
        $h_{\mathrm{i}}$ = next hop on the shortest path $c_{\mathrm{i}}$ to $r_{\mathrm{i}}$
        capital on directional edge $e_{\mathrm{c_i\text{-}to\text{-}h_i}} += a_{\mathrm{i}}$
        capital on directional edge $e_{\mathrm{h_i\text{-}to\text{-}c_i}} -= a_{\mathrm{i}}$
        if capital on $e_{c_i\text{-}to\text{-}h_i} > c_{max}$ on $e_{c_i\text{-}to\text{-}h_i}$ then
            $c_{\mathrm{max}}$ on $e_{\mathrm{c_i\text{-}to\text{-}h_i}}$ = capital on $e_{\mathrm{c_i\text{-}to\text{-}h_i}}$
        end
    end
end
for each edge $e$ in $G$ do
    for direction $d$ in {left (l), right (r)} do
        $C += c_{\mathrm{max}}$ on edge $e$ in direction $d$
    end
end
return $C$
```

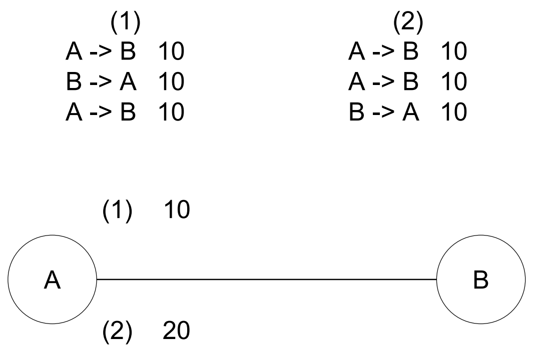


Figure 2.1: Order matters. Transactions on a single edge between node A and B

2.1 Algorithmic Tree Design

The aim is to design an algorithm, which takes a set of transactions as input and returns the optimal tree as output. Unfortunately, solutions of well-known graph problems cannot be applied, since a few characteristics of the payment network lead to significant differences, and thus, to a new problem. In a first step, these characteristics are discussed. And, it is pointed out why well-known optimization schemes can not be applied. In a second step, parameters, which can be used to design an algorithm, are briefly analyzed.

Characteristics - Moving Capacity: The capacity of one side of an edge is moving with the transaction to the other side. Assuming, node A and B have allocated an edge between each other with a value of 10 on both sides. When A sends 5 units to B, A's capacity of 10 will be reduced to 5 and B's capacity of 10 will be increased to 15. The amount of money a node is able to send along its edges (capacities) is variable and depends on the previous transactions routed through the edges.

Characteristics - Order Matters: The order of the transactions in the transaction set matters. The required capital (the optimal tree) becomes different by changing the order of the transactions. A brief example is given for two nodes, see Figure 2.1. There are 2 transactions from A to B, and 1 transactions from B to A. In the first set (1), A and B alternate with sending a value of 10. It yields to a required capital of 10. In the second set (2), the last two transactions are switched. It yields to a capital demand of 20. The dependence on the order seems not to be significant for a single edge, but it becomes challenging when it comes to multi-hops.

Parameters: Only a few parameters can be used to design an approximation algorithm based on transactions.

- using the payment amounts, e.g. creating an edge for transactions with the highest amounts.
- using the number of transactions between two nodes as a parameter, e.g. connecting the nodes with many transactions between each other.
- using the capital assignments on the edges, e.g. on a random topology, reducing the assignments one edge by changing the topology. But, it is more complex, since changing the topology might force an increase of capital on other edges.

Optimization Scheme - Dynamic Programming: It can be applied when the problem has an optimal substructure and overlapping sub-problems [10]. Unfortunately, it is not the case for Problem 1.3. Optimal substructure means, that the problem can be divided into sub-problems and the solution can be obtained by the combination of the optimal solution to its sub-problems. Going back to the problem of creating the optimal tree out of a set of transactions, there are two ways of dividing. One can split the transactions into several sets (e.g. of size 1) and calculate the optimal sub-tree for each set. But, no solution could be found to combine the sub-trees to one tree because of the moving capacity. Or, one can split the transactions along the nodes by creating node sub-sets S. All transactions between the nodes in S_i form a transaction set. But, the remaining transactions which take place between nodes of different node sub-sets (between nodes of S_i and S_j) do not enable optimal substructure. The remaining transactions have an impact on the solution of S_i and they can not been modelled as transactions from outside because of the moving capacity.

2.2 Deterministic Algorithm

In this chapter only deterministic approaches are considered. At first, an algorithm which returns the best star structure (star structure requiring minimal capital) and its capital demand for a transaction set is presented. Secondly, an algorithm which creates a tree structure based on the payment amount of the transaction is shown. Finally, the two algorithms are analyzed on their capital demand by testing them on various transaction sets.

Best star (Alg. 2): It calculates the star (or also called hub) with the minimum capital demand to process all transactions (`best star`). It takes a set of transactions TXs as input and simply brute-force every star of n nodes on TXs. Routing a transaction through the graph requires only two hops at maximum. Thus, calculating the capital demand of a star takes only $2k$ iterations

Algorithm 2: Best star

```
input  : A set $Txs$ of transactions $ti = (s_i, r_i, a_i)$
output: A star $S = (V, E)$
1  $C$ = max interger
2  $V$ = get all nodes in $S$
3  for each $v$ in $V$ do
4      $center = v$
5      $V, E$ = get star($center$, $V$)
6      $c$ = required capital of star $(V, E)$ and transaction set $Txs$
7      if $c < C$ then
8          $C = c$
9          $S = (V, E)$
10     end
11 end
12 return $S, C$
```

at maximum and k iterations at minimum for k transactions. It results in a time complexity of:

$$\Theta_{\text{star}} = \Theta(kn) \tag{2.2}$$

Create a tree based on largest payments (Alg. 3): It creates a tree according to the value of the payment amount of the transactions. A tree T is created by selecting the transactions in decreasing order of the payment amount and making sure that a path between sender and receiver exists. In case no path does exit between sender and receiver, an edged is added between them.

The algorithm has a run-time, independent on the number of nodes n. It yields a time complexity of

$$\Theta_{\text{lpa}} = \Theta(k) \tag{2.3}$$

for k transactions.

Required capital analysis (Fig. 2.2) The approximation factors of the two algorithms are analyzed experimentally. 100 transaction sets are created uniformly at random for 3 to 8 nodes (for each number of nodes 100 sets). The number of transaction (nTx) in each set is chosen to be dependant on the number of nodes n, with $nTx = 10 * n$. Finally, the algorithms are executed for each transactions set and they are compared to each other by creating a ratio of their required capital. In addition a random tree is picked and its capital deman is calculated as a further reference. The outcome of this analysis is presented in Figure 2.2. It shows the average approximation of optimal capital for `random tree` and `largest payment` algorithm related to the `best star`. It can be seen, that `largest payment` alg. provides a better approximation than just picking

Algorithm 3: Create a tree based on largest payments

```
input  : A set $Txs$ of transactions $ti = (s_{\text{i}}, r_{\text{i}}, a_{\text{i}})$
output: A tree $T = (V, E)$
1  $E \leftarrow \{\text{empty}\}$
2  $V \leftarrow \{\text{empty}\}$
3  while $Txs$ is not {empty} do
4      $t_{\text{max}}$ = get $t_{\text{i}}$ with the highest payment amount in $Txs$
5      remove $t_{\text{max}}$ from $Txs$
6      $s_{\text{i}}, r_{\text{i}}, a_{\text{i}} = t_{\text{max}}$ /* sender, receiver, payment amount */
7      if $s_i$ or/and $r_i$ does not exist in $V$ then
8          add $s_{\text{i}}$ or/and $r_{\text{i}}$ to $V$
9      end
10     if path $s_i$ to $r_i$ does not exist in $E$ then
11         add edge $s_{\text{i}} - to - r_{\text{i}}$ to $E$
12     end
13 end
14 return $V, E$
```

a tree at random. But, it can also be seen, that `best star` provides a better approximation of optimal capital.

2.3 Probabilistic Algorithm

The purpose of this chapter is to present a probabilistic algorithm, which calculates the optimal tree with probability p for a transaction set. Further, the algorithm is experimentally analyzed regarding its termination behavior, time complexity and probability of finding the optimal tree. The analysis is based on 60 transaction sets containing 200 transactions each. The transactions are generated uniformly at random for a different number of nodes. There are always 10 sets for 3, 6, 9, 12, 15 and 18 nodes. For each set 1000 iterations of the algorithm are done. These 60 transaction sets are referred as data set and the 1000 iterations of the algorithm on each set are referred as experiment for the rest of this chapter. At the end, ideas for improvements of the algorithm are discussed and analyzed.

Random Edge Optimization (REO) (Alg. 4): It takes a random tree and a set of transactions as an input. The algorithm generates the optimal tree with probability p as output. The tree which requires the minimum required capital to handle all transactions is referred to as the *optimal tree.* The idea is to optimize locally each single edge of the random tree T. At first, an edge of T'

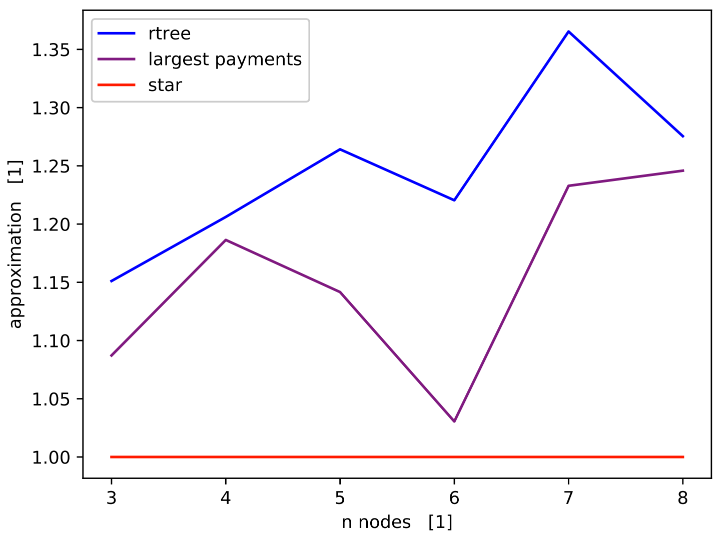


Figure 2.2: Average approximation of optimal capital for `random tree` (rtree) and `largest payment` algorithm (largest payment) related to the `best star` (star)

is randomly selected and temporarily removed. What remains are two sub-trees T_1 and T_2 of T'. In a next step, the best edge (the edge which results in the least required capital) is selected out of all possible edges, which connects the two sub-trees T_1 and T_2. That means the required capital for all trees results by connecting T_1 and T_2 with one single edge is calculated and the one with the least demand on required capital is selected. In case the best edge e_{best} is the original edge e_{r} this edge is marked. Otherwise if a new edge is selected all marks of T' are removed. The marking of the edges is introduced because when an edge of the tree T' is changed, all previous calculated best edges might not be optimal anymore. Finally, the process is repeated until all edges of T' are marked.

Termination Two questions arise:

1. Does the algorithm always terminate?

2. When does it terminate?

(1) The algorithm terminates in the worst case after all possible trees n^{n-2} have been processed, which is achieved by having a list of already processed trees (PT).

(2) For answering this question only the outer loop has to be considered. Since an analytic solution could not be found, this question is answered by counting the number of rounds of the outer while-loop in the experiment. In Figure 2.3 the outcome of the measurements is presented. It shows the rounds normalized

Algorithm 4: Random Edge Optimization (`REO`)

```
input  : $T = (V, E) \leftarrow$ a random tree
input  : $Txs \leftarrow$ a set of transactions
output: $T' = (V, E') \leftarrow$ optimal tree with probability $p$
output: $C \leftarrow$ required capital to process all transactions
1  $T' = T$
2  $PT = \{\text{empty}\}$ /* processed trees */
3  $e_{\text{best}} = \{\text{empty}\}$
4  add $T'$ to $PT$
5  $C =$ required capital of the tree $T'$ and transaction set $Txs$
6  while an unmarked edge exists in E' do
7      $e_{\text{r}} =$ randomly select an edge from $E' \setminus \{e_{\text{best}}\}$
8      $T_1, T_2 =$ subtrees of $T' \setminus \{e_{\text{r}}\}$
9      $e_{\text{best}} = e_{\text{r}}$
10     for each edge e connecting $T_1$ and $T_2$ not equal $e_r$ do
11         $T'' = (V, E' \setminus \{e_{\text{r}}\} + e)$
12         if T” not in $PT$ then
13             add $T''$ to $PT$
14             $capital =$ required capital of the tree $T''$ and transaction set
                $Txs$
15             if $capital < C$ then
16                 $C = capital$
17                 $e_{\text{best}} = e$
18             end
19         end
20     end
21     if $e_{best} == e_r$ then
22         set a mark at edge $e_{\text{best}}$
23     end
24     else
25         $E' = E' \setminus \{e_{\text{r}}\} + e_{\text{best}}$
26         remove the mark for all edges of $E'$
27     end
28 end
29 return $T', C$
```

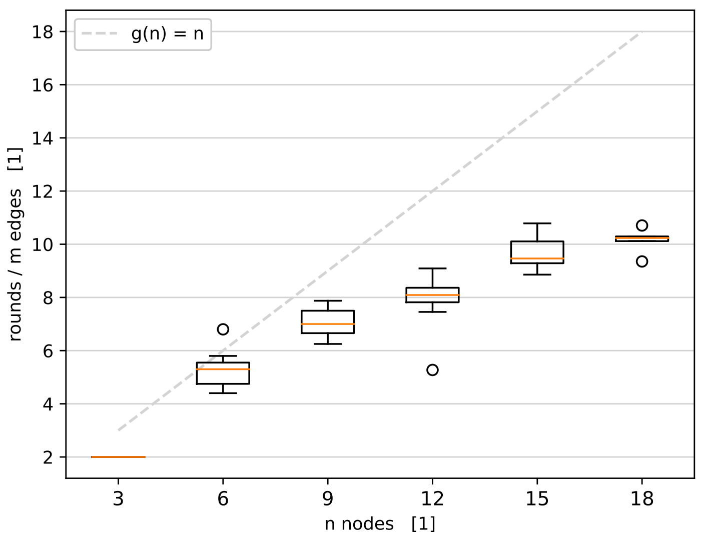


Figure 2.3: Number of rounds of the outer while-loop normalized on the number of edges m

on the number of edges over the number of nodes in the transaction set. It can be observed that the boxes are very dense, or in other words the deviation is low besides a few outliers. Further, a slight logarithmic increase of the number of rounds strike out. Nevertheless, it is close to linear, therefore, a linear complexity is assumed:

$$\mathcal{O}_{\text{while-loop}} = \mathcal{O}(nm) = \mathcal{O}(n^2) \tag{2.4}$$

$$\Omega_{\text{while-loop}} = m = n - 1$$

for $m = n - 1$.

Time complexity At first, the inner for-loop is considered. Two subtrees T_1 and T_2 remain after removing one edge from T'. The number of edges connecting the two subtrees is at least $n-1$ and at most $(\frac{n}{2})^2$, and thus

$$\mathcal{O}_{\mathrm{T_1\text{-}T_2}} = (\frac{n}{2})^2 = \frac{n^2}{4} \tag{2.5}$$

$$\Omega_{\mathrm{T_1\text{-}T_2}} = n - 1$$

Secondly, the outer while-loop, or in other words how many iterations it takes until all edges are marked, is analyzed. An edge of T' is marked when all other possible edges which connecting the subgraphs T_1 and T_2 requires more capital. It follows that it takes at least $m = n - 1$ iterations until all edges of T' are marked.

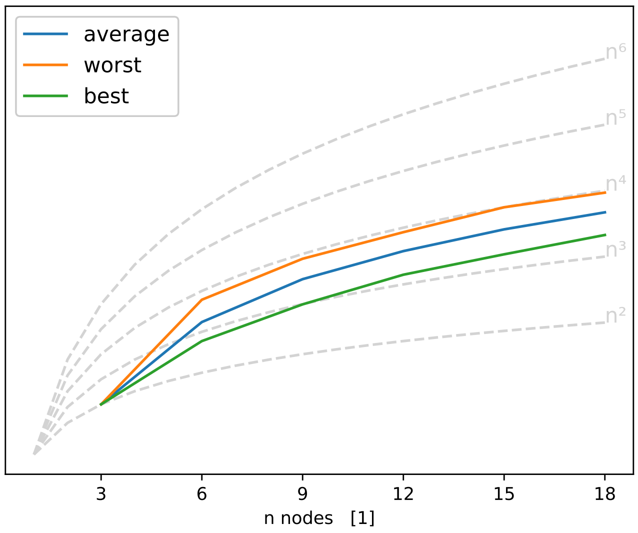


Figure 2.4: Best, average and worst measured run-time normalized on k

Finally, combining all together and also taking into account the complexity of calculating the required capital of a tree $\mathcal{O}_{\text{cap}}$ of Equation 2.1, we derive the following time complexity

$$\mathcal{O}_{\text{REO}} = \mathcal{O}_{\text{while-loop}} * \mathcal{O}_{\text{T}_1\text{-T}_2} * \mathcal{O}_{\text{cap}} = \mathcal{O}(kn^5) \tag{2.6}$$
$$\Omega_{\text{REO}} = \Omega_{\text{while-loop}} * \Omega_{\text{T}_1\text{-T}_2} * \Omega_{\text{cap}} = \Omega(kn^2)$$

for k transactions and n nodes.

In Figure 2.4 the outcome of the run-time analysis is presented. It shows the best, worst and average measured run-time over n nodes. It can be observed, that the worst scales slightly below n^4 and the best slightly above n^3. The measured run-time is within the expected bounds of $[n^2, n^5]$. Nevertheless, a brief explanation of the average complexity should be given by taking a deeper look at $\mathcal{O}_{\text{cap}}$. In the worst case, each transaction require $n-1$ hops to reach its destination, though only occurring when all nodes form a chain and each transaction is from head to tail, or vice versa, which is rarely the case. Shen et al. [11] has shown that the average diameter of a general tree structure is $\Theta(logn)$. Thus, the average complexity of calculating the required capital for general trees becomes $\mathcal{O}_{\text{avgcap}} = \mathcal{O}(logn)$. This yields an average complexity of the algorithm of $\mathcal{O}(n^4 logn)$.

Probability: The algorithm does not always return the optimal tree, since it might find a local minimum instead of the global minimum. Therefore, in the experiment the probability of finding the optimal tree is analyzed. In a first step,

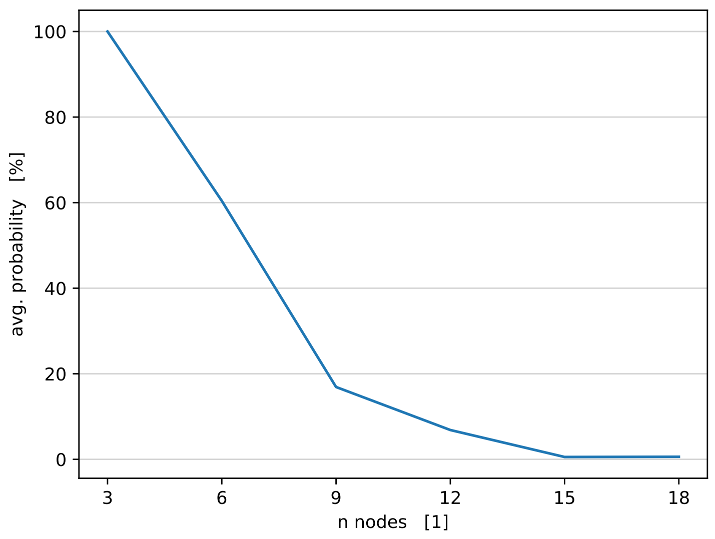


Figure 2.5: Average probability to find the optimal tree

the optimal tree was found by a brute-force algorithm. The required capital, which is required to handle all transactions, is evaluated for each possible tree with n nodes. Unfortunately, the brute-force scales with $n^{(n-2)}$, and therefore, the time frame was only acceptable for brute-forcing up to 9 nodes. For 12, 15 and 18 nodes, the minimum out of 1000 iterations was assumed to be the optimum. In Figure 2.5 the average probability is illustrated. It points out that the probability is decreasing rapidly by increasing the number of nodes. For only 3 nodes, the output is at 100 %. But, already for 9 nodes, the optimal is found with only 16 % and below 1 % for only 15 nodes. Unfortunately, the result is far from desired outcome of having a high probability for a large n.

Deviation: The deviation from the optimal tree, or in other words the approximation to the optimum, is analyzed. In Figure 2.6 the outcome of the experiment is illustrated. It shows the distribution of the deviation from the optimal tree. It can be observed, that the average deviation is monotonically increasing and that distribution gets more spread with an increasing number of nodes.

2.3.1 Ideas of Improvement

In this Section, we discuss some ideas for improving the algorithm both in terms of run-time and probability of optimality p. It was shown above that the average run-time scales polynomial with $\mathcal{O}(n^4)$ which is not efficient for a large number of nodes. Moreover, the probability p becomes very low for a large n, and thus,

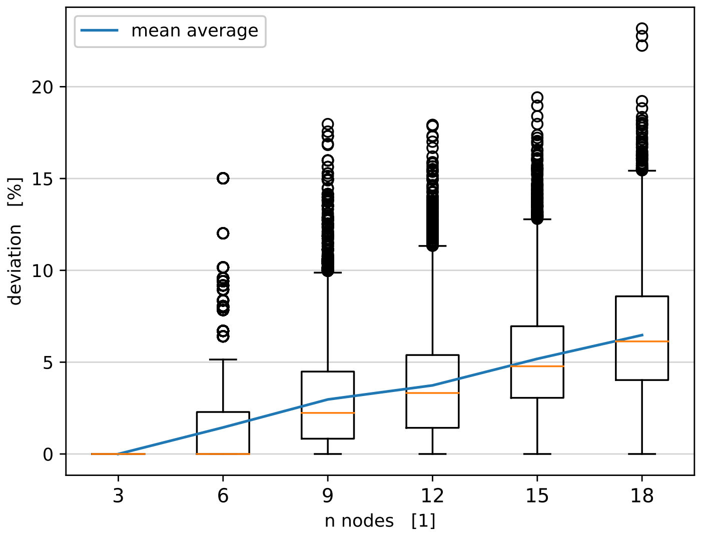


Figure 2.6: Deviation from the optimal tree

it is unlikely to find the optimum.

Tracking processed trees: This idea is already included in the presented algorithm 4. Each tree, for which the required capital was calculated, is stored by adding it to a list of processed trees. This avoids that a certain tree is processed more than once, which might be possible otherwise. This improvement has been added and implemented from the beginning, and therefore, the gain out of it was not further analyzed.

Star input: The algorithm starts with a random tree, which might be far from an optimum starting point. Thus, we consider alternative starting points. In [1] it was shown that any star yields a 2-approximate solution, which seems to be a better starting point than a random tree. Nevertheless, the experiments showed that the algorithm with a random star as input does not perform better than the original one that starts with a random tree. Figure 2.7 depicts the average probability p for a random star and a random tree as starting point. It can easily be seen that there is no improvement on p.

Threshold: Is there any threshold of how often an edge must be marked until it is guaranteed that the edge is a part of the output tree T'? If this is the case, unnecessary computation could be avoided, and thus, the run-time could be improved. Figure 2.8 shows how often edges were marked. At first, the left figure, resulting from the measurements with 15 nodes, shall be discussed. It can be seen, that some edges have been marked up to 12 times, whereas the most edges have been marked only once. Further, it sticks out that there is a

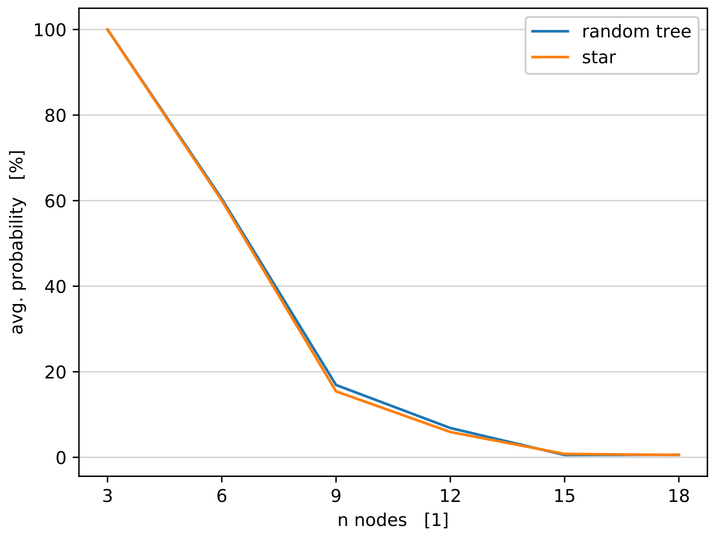
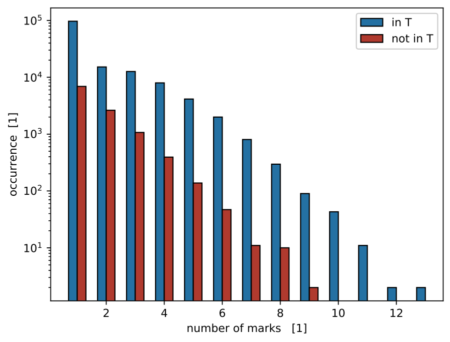


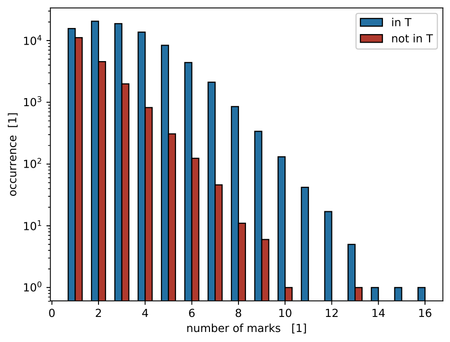
Figure 2.7: Average probability to find the optimal tree. Comparison between random tree and star as input

threshold, since the edges which are not going to be in the resulting tree T are marked only 9 times at maximum. Therefore, after an edge has been marked 10 times, it is going to be in T. Unfortunately, the gain is negligible small, since the threshold would be reached only 12 times out of 1.5 * 10^5. It is not really an option to set the threshold below 10 and to accept an false positive error, since the approximation factor would become worse. The same holds for a tree with 18 nodes, see figure on the right side.

Parallelization: `REO` comes with a polynomial complexity of $\mathcal{O}(kn^5)$, and an average complexity of $\mathcal{O}(kn^4 logn)$. Although, it scales polynomial an immense calculation effort is required to apply it on a network of 1000 nodes, since, the constant hidden in the $\mathcal{O}$-notation is rather high. The average run-time has been around 8.4 s for 18 nodes and 200 transactions on the testing machine (Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz) in the analysis. If these numbers are extrapolated to 100 nodes and 200 transactions (with n^4, worst measurement) it results an execution time of 2.5 hours, which might be acceptable. But, with 1000 nodes and 200 transactions it will take already 2.5 years. Nevertheless, there is a potential for further improvements through parallelization, since, the inner For-loop can be parallelized to a certain degree. The required capital for each tree, which emerges by connecting the two subtrees T_1 and T_2 with a single edge, is calculated at this state of the algorithm. Since, the calculations are completely independent from each other, they can be parallelized. By using a GPU the run-time can be reduce to a fraction of $\frac{n^2}{4}$ ($\mathcal{O}_{\mathrm{T_1\text{-}T_2}}$). For instance for a network of



For 15 nodes



For 18 nodes

Figure 2.8: How often a certain number of marking occurred over 1000 iterations on all sets. The blue bars stand for markings of the edges of the resulting tree T, and the red bars represent the marking of the edges are not part of T

100 nodes 2500 cores could be used (e.g 3 * GeForce GTX 960 containing 1024 cores [12]). This would reduced the run-time of $\mathcal{O}_{\mathrm{T_1\text{-}T_2}}$ from $\mathcal{O}(n^2)$ to $\mathcal{O}(1)$, and therefore, the total complexity of the algorithm would reduce from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$. Furthermore, having 1000 nodes, using 250'000 cores might rather not be an option. Nevertheless, the run-time can still be reduced linearly by applying $\Omega_{\mathrm{T_1\text{-}T_2}} = \Omega(n)$ cores. It yields a run-time of $\mathcal{O}(n^4)$. Going back to the previous hypothetical calculations with the result of 2.5 years. Parallelization might allow to reduce the run-time by a factor of 1000 by using 1000 cores. Instead of 2.5 years, it would take 22.2 hours.

2.4 Final Evaluation

In this chapter, two algorithms have been discussed. Alg. 3 creates a tree in dependence of the largest payment amount and Alg. 4 optimizes the tree edge for edge. In this section, the performance of the algorithms is briefly discussed and compared to the optimal tree.

For the analysis, again 100 transaction sets are created uniformly at random for 3 to 8 nodes (for each number of nodes 100 sets), and with number of transactions $(nTx) = 10 * n$. Because of the high computational cost of brute-forcing each tree ($\mathcal{O}(n^{n-2})$ the evaluation is only done for up to 8 nodes. On each set the following algorithms are executed:

- `random tree`: select a tree uniformly at random
- `largest payment`: Alg. 3

- `best star`: find the best star by brute-forcing
- `REO`: Alg. 4 (1 round)
- `optimal tree`: brute-forcing all trees to find the tree with the least capital demand
- `optimal graph`: brute-forcing all connected graphs to find the graph with the least capital demand

Each of them outputs a graph and its corresponding demand on capital to executed all transactions. The approximation of the optimal capital of trees is calculated by relating the resulting capital demand to the optimal capital. Finally, the average approximation is presented in Figure 2.9. Selecting a tree at random provides the worst approximation (blue cure). Further, it can be seen that using the `largest payment` algorithm (purple curve) results in a slightly better approximation, but it requires still 40 % more capital than in the optimal case at the worst point. The `best star` (red curve) gives an average approximation of up to 1.15. Further, The `REO` alg. (green curve) provides a near optimal approximation with an average factor of 1.04 at 8 nodes. Finally, the optimal connected graph (including trees) is calculated (orange curve) for up to 6 nodes and linearly extrapolated until 8 nodes (dashed line). The optimal connected graph comes with a slightly lower demand on capital as the optimal tree. But, on the other hand, it requires more edges, and thus, the costs $(m + \alpha C)$ might be higher.

In a second analysis the `REO` alg. is compared to the `best star` for a higher number of nodes. 10 data sets are created uniform at random with 500, 550 and 600 transactions for a network size of 50, 55 and 60 nodes, respectively. `REO` is executed on each data sets once and its resulting demand on required capital C_{REO} is compared to the required capital $C_{\text{best star}}$ of the `best star` structure by dividing C_{REO} through $C_{\text{best star}}$. The average of this ratio is presented in Table 2.1. All 3 network sizes result in roughly the same ratio of about 0.88, which means `REO` provides a tree structure with in average a 12 % smaller demand on capital than the `best star`.

nodes	$C_{\text{REO}}/C_{\text{best star}}$
50	0.885
55	0.887
60	0.878

Table 2.1: Avg. required capital of `REO` related to `best star`

In conclusion, the `REO` alg. provides a near optimal solution of Problem 1.3 and it offers in average a 12 % better approximation than the `best star`. But unfortunately, `REO` comes with a polynomial worst-case complexity of $\mathcal{O}(kn^5)$, and

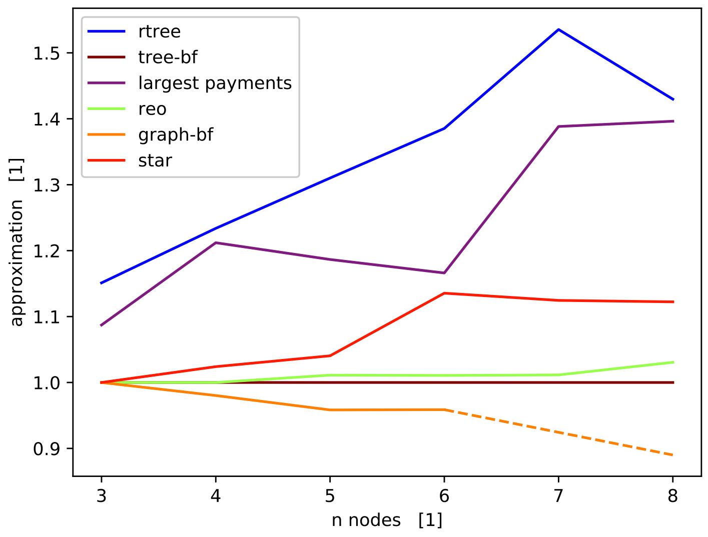


Figure 2.9: Average approximation of optimal capital for random tree (`rtree`), largest payment amount algorithm (`largest payment`), best star (`star`), `REO` algorithms and optimal graph (`graph-bf`) related to the optimal tree (tree-bf)

an average-case complexity of $\mathcal{O}((logn)kn^4)$. Although, there is the potential to further reduce the run-time by using parallelization as discussed in Section 2.3.1, the complexity is not fully satisfying, since, it does not allow to calculate the required capital of a network of thousands of nodes and thousands of transactions, that would represent a real payment network.

Chapter 3

Optimal Graph Approximation

In this chapter, Problem 1.4 - Graph Design for All Transactions is studied. It is the more general case of Problem 1.3. Since, the topology is not restricted to trees anymore, cycles are also considered. But still, all transactions shall be processed by the network, and thus, the graph has to be connected. It is focused on minimizing the costs

$$Co = (m + \alpha C)$$

, with m edges, total required capital C and interest rate α.

Extend tree to a connected graph (Alg. 5): Allowing cycles increases the complexity of the problem drastically. Hence, the optimal tree is extended with cycles such that the cost is minimized. The following algorithm is presented, Alg. 5. It takes a transactions set and its corresponding optimal tree as input as well as the current interest rate α. The return values are the best graph G (graph with minimal costs) and the corresponding cost Co. For each transaction which does not take place between direct neighbors the cost of the graph, which contains the edge between the two nodes is calculated. In case the cost can be lowered by adding an additional edge, the edge is added to the graph G. But since, the previous calculations have been based on the Graph without the new edge, the process is repeated for the previous transactions with the new G.

The algorithm (5) takes at least k iterations, in case each transaction take place between two neighboring nodes and no required capital calculation is executed. Thus, the optimal tree is also the best found graph. The worst-case complexity of $k!kn$ is far from optimal. $k!$ is caused by the fact that the calculation process is repeated in case an edge is added. Theoretically, it is possible that the last transaction in the set forces to add an edge to G, and thus, the calculations for the previous transactions is repeated. In this second round, the second last transaction is adding an edge to G. And so on until to the first transaction. It ends up in $k!$ iterations. It could be avoided by limiting the number of iterations, e.g. to $2k$, with the drawback of not finding the best solution.

$$\mathcal{O}_{\text{cycle\_extend}} = \mathcal{O}_{\text{cap}} * k! = \mathcal{O}(k!kn) \tag{3.1}$$
$$\Omega_{\text{cycle\_extend}} = \Omega(k)$$

Algorithm 5: Extend tree to connected graph (allowing cycles)

```
input  : $T = (V, E) \leftarrow$ an optimal tree
input  : $Txs \leftarrow$ the transactions of the optimal tree
input  : $\alpha \leftarrow$ interest rate on capital
output: $G = (V, E') \leftarrow$ the optimal graph
output: $Co \leftarrow$ The cost to handle all $Txs$ in $G$

1  graph $G = T$
2  capital $C$ = required capital of $G$ for all transactions $Txs$
3  number of edges $m$ = number of edges of $G$
4  cost $Co0 = m + \alpha * C$
5  transactions $tx = Txs$
6  while $tx$ is not {empty} do
7      /* sender, receiver, payment amount */
8      $s_\mathrm{i}, r_\mathrm{i}, a_\mathrm{i} = t_\mathrm{i}$ = next transaction in $tx$
9      remove $t_\mathrm{i}$ from $tx$
10     if $s_i$ and $r_i$ neighbors in $G$ then
11         continue the loop
12     end
13     $C$ = required capital of $G + e_{\mathrm{s_i\ to\ r_i}}$ for all transactions $Txs$
14     cost $Co1 = m + 1 + \alpha * C$
15     if $Co1 < Co0$ then
16         $Co0 = Co1$
17         $m+ = 1$
18         $G+ = e_{\mathrm{s_i\ to\ r_i}}$
19         $tx = Txs$
20     end
21 end
22 return $G, Co0$
```

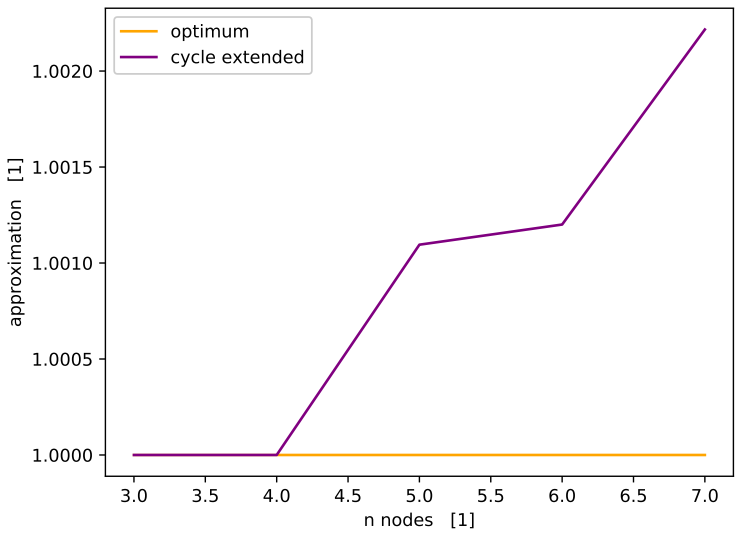


Figure 3.1: Average cost ratio $\frac{Co_{\text{cycle\_extend}}}{Co_{\text{optimal}}}$ where $Co_{\text{cycle\_extend}}$ is the cost of the graph resulting by applying `cycle extend` algorithm (Alg. 5) and Co_{optimal} is the cost of the optimal graph.

Optimal costs analysis (Fig. 3.1): The costs $Co_{\text{cycle\_extend}}$ of the graph returned by the algorithm is evaluated. The algorithm is applied on 20 transaction sets of 200 transactions and $Co_{\text{cycle\_extend}}$ is compared to the cost of the optimal solution Co_{optimal} by creating a cost ratio ($Co_{\text{cycle\_extend}}/Co_{\text{optimal}}$). It was done for 3,4,5,6 and 7 nodes, since the optimal solution could only be found by brute-forcing. In Figure 3.1 the average of the cost ratio is presented. It can be seen, that $Co_{\text{cycle\_extend}}$ is only slightly higher than the cost of the optimal graph. The algorithm provides the optimal solution (ratio $= 1$) for 3 and 4 nodes whereas its outcome deviates with about 1 per mile for 5 and 6 nodes and with about 2 per mile for a network size of 7 nodes.

Chapter 4

Maximize Profit

In this chapter, Problem 1.5 - Transaction Selection to Maximize Profit is studied. A payment provider will directly face this problem. Since a provider might not have enough capital available to handle all transactions, he is interested in processing only the most profitable transactions (maximizing profit). By restricting the problem to tree structures and with a capital limitation C_{limit} the profit is equal to:

$$P = k - (n - 1 + \alpha C_{limit})$$

, with k chosen transactions, n nodes and an interest rate α. For a given transactions set, the number of nodes n is well-known. Additionally, the interest rate α is constant. Therefore, the only variable in the equation is the number of chosen transactions k. Throughout this chapter, approximation algorithms for maximizing k, and thus, maximizing the profit P, are presented and discussed.

The challenge for maximizing k is given by the fact, that it depends on the tree structure combined with the optimal strategy of selecting transactions. In the first Section, the offline case is discussed. On the contrary, in the second Section, we focus on the online case, where the transactions are not known in advance.

4.1 Max Profit - Offline

The term offline is used, when all transactions are known in advance.

Greedy selecting (Alg. 6): The algorithm takes the capital limit C_{limit} and the transactions set Tx as input, and it returns the tree structure T as well as the list of the chosen transactions Tx_{chosen}. The algorithm greedily selects the transactions with the lowest payment amounts as long as C_{limit} is fulfilled. In the first while-loop, the smallest payment amounts are chosen, such that, their sum does not exceed C_{limit}. The second while-loop is used to calculated the effective required capital C of the Tx_{chosen} base on the `REO` algorithm presented in Section 2.3. In addition, further transactions are added as long as the required capital does not exceed the limit.

Algorithm 6: Greedy selecting (`greedy`)

```
input  : Capital limit $C_{\text{limit}}$
input  : $Tx$ a set of transactions $ti = (s_{\text{i}}, r_{\text{i}}, a_{\text{i}})$
output: Tree $T = (V, E)$ and chosen transactions $Tx_{\text{chosen}}$ complying
        $C_{\text{limit}}$
1  Total transaction amount $Tx_{\text{amount}} = 0$
2  $Tx_{\text{chosen}} = \{\text{empty}\}$
3  while $True$ do
4      /* sender, receiver, payment amount */
5      $s_{\text{i}}, r_{\text{i}}, a_{\text{i}} = t_{\text{i}} = min(Tx)$
6      $Tx_{\text{amount}} += a_{\text{i}}$
7      if $Tx_{amount} < C_{limit}$ then
8          remove $t_{\text{i}}$ from $Tx$
9          add $t_{\text{i}}$ to $Tx_{\text{chosen}}$
10     end
11     else
12         break
13     end
14 end
15 while $True$ do
16     $t_{\text{i}} = min(Tx)$
17     $T', C$ = Alg. 4 REO($Tx_{\text{chosen}} + t_{\text{i}}$)
18     if $C < C_{limit}$ then
19         remove $t_{\text{i}}$ from $Tx$
20         add $t_{\text{i}}$ to $Tx_{\text{chosen}}$
21         $T = T'$
22     end
23     else
24         break
25     end
26 end
27 return $T, Tx_{\text{chosen}}$
```

Considering the first while-loop isolated, the worst case occurs when all transaction are selected. This yields a complexity of $\mathcal{O}(k)$. Whereas, the lower limit is given with $\Omega(1)$ when no transaction is chosen. The worst case for the second while-loop occurs when in the first loop only one transaction is chosen and the subsequent transactions do not cause a higher capital demand, e.g the amount of the first transaction is just send back and forth between two participants. In this case the worst-case complexity is $\mathcal{O}(k) * \mathcal{O}_{\text{REO}} = \mathcal{O}(k^2 n^5)$. In the optimal case, the loop is executed only once, and therefore, Ω_{REO} (2.6) iterations are required. Concatenating both together, the complexity of the algorithm is given by

$$\mathcal{O}_{\text{greedy}} = \mathcal{O}(k^2 n^5) \tag{4.1}$$

$$\Omega_{\text{greedy}} = k * \Omega_{\text{REO}} = \Omega(k^2 n^2)$$

Selecting with probability (Alg. 7): A further approach is to select the transactions according to a certain probability. Alg. 7 chooses the transactions with a probability p, which depends on the payment amount a_{i}, such that small amounts are favoured. A payment with the largest payment amount of the set of transactions Txs (a_{max}) is selected with probability $a_{\text{min}}/(a_{\text{max}}+a_{\text{min}})$, a payment with the smallest payment amount of Txs (a_{min}) is selected with probability $a_{\text{max}}/(a_{\text{max}} + a_{\text{min}})$. After a selection round (for-loop) the required capital C is calculated with the `REO` algorithm. In case $C > C_{\text{limit}}$ the selection is not accepted and it is repeated with $p = p/2$. In case $C < C_{\text{limit}}$ the selection is accepted and it is added to Tx_{chosen}. The process is repeated until no transactions are selected.

Analysis of number of selected transactions (Fig. 4.1): Since it is not possible to find the optimal solution (there is no algorithm and brute-forcing all possibilities of selecting transactions on all trees is not a realistic option), the two algorithms are only compared. Running them on 20 transaction sets of 200 transactions with $C_{\text{limit}} = C_{\text{opt}}/2$ (optimal capital to process all transactions) yields to the result depicted in Figure 4.1. It can be seen, that both algorithms have similar performance. In average they selected 115 transactions out of 200. `pselect` shows a better result of 135 selected transactions only for 3 nodes.

4.2 Max Profit - Online

In the online case, the transactions are unknown in advance. There is a transactions stream, which shall be processed by the PSP. The questions arise: how to design the optimal topology without knowledge of the incoming transactions? How to assign the available capital to the edges? Which transactions shall be executed through the payment network? Answering the questions is highly challenging under the condition of profit maximization. Nevertheless, 3 strategies shall be defined to answer the questions. Strategy S_{topology} defines the topology,

Algorithm 7: Selecting with probability (`pselect`)

```
input  : $C_{\text{limit}} \leftarrow$ Capital
input  : $Txs \leftarrow$ a set of transactions
output: Chosen transactions $Tx_{\text{chosen}}$ complying $C_{\text{limit}}$
1  $a_{\text{max}}$ = maximal transaction amount of $Txs$
2  $a_{\text{min}}$ = minimal transaction amount of $Txs$
3  $Tx_{\text{chosen}} = \{\text{empty}\}$
4  $x = 0$
5  while True do
6      $Tx_{\text{selected}} = \{\text{empty}\}$
7      for each $t_i$ in $Tx$ do
8          /* sender, receiver, payment amount */
9          $s_{\text{i}}, r_{\text{i}}, a_{\text{i}} = t_{\text{i}}$
10         $p = (1 - \frac{a_{\text{i}}}{a_{\text{max}}+a_{\text{min}}})/2^x$
11         add $t_{\text{i}}$ to $Tx_{\text{selected}}$ with probability $p$
12     end
13     if $Tx_{selected}$ is {empty} then
14         break
15     end
16     $Tx_{\text{current}} = Tx_{\text{selected}} + Tx_{\text{chosen}}$
17     $T, C$ = Alg. 4 REO($Tx_{\text{current}}$)
18     if $C > Climit$ then
19         $x+ = 1$
20     end
21     else
22         $Tx_{\text{chosen}} = Tx_{\text{current}}$
23         remove all transactions of $Tx_{\text{tmp}}$ from $Tx$
24     end
25 end
26 return $Tx_{\text{chosen}}$
```

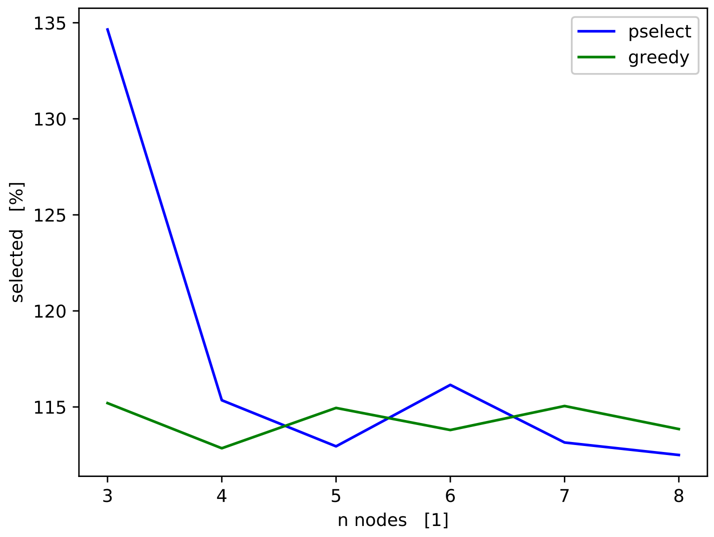


Figure 4.1: Average number of selected transactions of Alg. 6 (`greedy`) and Alg. 7 (`pselect`) with $C_{\text{limit}} = C_{\text{opt}}/2$ (optimal capital to process all transactions)

strategy S_{capital} specifies the capital assignment and strategy $S_{\text{transactions}}$ chooses the transactions.

A PSP is going to face permanently new transactions assuming a network of some hundreds of nodes. The users are going to have regular payments (e.g. paying a coffee in its favourite cafe, buying from the supermarket, etc.) to other users. It is rather unlikely that a user is going to send money to all the other users. Therefore, the transaction stream will have a certain distribution. There will might be more transactions between user A and B than between A and C. A certain patter can also be expected in terms of payment amount (e.g a coffee cost always about 4 CHF whereas its weekly shopping at the supermarket might be about 80 CHF). Thus we concluded that following a learning strategy might be beneficial in this setting. The following strategies are defined:

- S_{topology}: apply Alg. 4 - `REO` on a learning set and use the resulting tree for the future transactions
- S_{capital}: learn the payment distribution of a learning set and assign capital on the tree according to the distribution
- $S_{\text{transactions}}$: accept each transaction which can be routed through the network without creating a new edge or without reassigning capital on an existing one.

A payment distribution can be created in various ways as well as assigning capital according to a distribution. The following method is used:

- Distribution: all payments between two nodes are summed up for both direction separately and divided by the total payment amount of all transactions.

- Capital assignment: for each edge in the tree T the corresponding distribution is multiplied with the maximum available capital (C_{limit}). Each distribution value of an edge not in T the number of hops between sender and receiver is calculated. Further, the distribution value is multiplied with C_{limit} and divided by the number of hops. The resulting value is assigned for each edge in the corresponding direction along the path between sender and receiver.

Maximize profit - online (Alg. 8) Given the 3 strategies the following algorithm is defined. The input is the capital limitation C_{limit} (available capital of the PSP), a learning transaction set and a validation transaction set, which is handled as a transaction stream. The output is the chosen transactions Tx_{chosen} which could be processed. At first, a tree T is created from the learning set with the `REO` algorithm. Further, the payment distribution D is calculated and the capital is assigned on T accordingly as described above. Finally, it is iterated through the validation transaction set. Each of them is accepted and processed in case the required capital is at each edge along the path between sender and receiver.

The time complexity of the `REO` algorithm is given by Equation 2.6. Further, getting the distribution out of k_1 transactions requires to iterate through all of them, $\mathcal{O}(k_1)$. Assigning capital on T takes $n-1$ iterations for distribution values with an edge in T. It takes $n-1$ hops in the worst case to assign capital from a distribution value without an edge in T. Thus, the worst case complexity is given by $(n-1)+(n-1)*k_1) \rightarrow \mathcal{O}(n(k_1+1))$. Finally, additional $k_2*(n-1)$ iterations are required to make the decision if the transactions can be processed. At the end, effectively processing a transaction does not cause significant computation, since the states of the involved edges (given through the accepting decision) have simply to be updated by changing their value. It yields a final worst case complexity of the algorithm of:

$$\mathcal{O}_{\text{online}} = \mathcal{O}(k_1) + \mathcal{O}(n(k_1+1)) + \mathcal{O}(k_2 n) = \mathcal{O}(k_1 + n(k_2 + k_1 + 1)) \qquad (4.2)$$

Analysis of number of processed transactions (Fig.4.2) The algorithm is analyzed the following way: 4 random sets with 200 transactions for 12 nodes are created with a distribution. The sets are split into two subsets according to a learn-validation ratio β, such that $k_1 = \beta k$ and $k_2 = (1-\beta)k$. Further, the capital limit C_{limit} is set to be equal to the optimal capital of the offline

Algorithm 8: Maximize profit - online

```
input  : Capital limit $C_{\text{limit}}$
input  : $Tx_{\text{learn}} \leftarrow$ a set of transactions to learn distribution $D$ and tree $T$
input  : $\text{Tx}_{\text{validate}} \leftarrow$ a set of transactions to apply $D$ and $T$
output: Tree $T$ and chosen transactions $Tx_{\text{chosen}}$ complying with $C_{\text{limit}}$

1  $T$ = Alg. 4 REO($Tx_{\text{learn}}$)
2  $D$ = get distribution of $Tx_{\text{learn}}$
3  assign $C_{\text{limit}}$ according $D$ on $T$
4  for each $t_i$ in $Tx_{validate}$ do
5      /* sender, receiver, payment amount */
6      $s_{\text{i}}, r_{\text{i}}, a_{\text{i}} = t_{\text{i}}$
7      accept if $\text{t}_{\text{i}}$ can be processed without new capital on the edges
8      if accepted then
9          process $t_{\text{i}}$ on $T$
10         add $t_{\text{i}}$ to $Tx_{\text{chosen}}$
11     end
12 end
13 return $Tx_{\text{chosen}}$
```

case $C_{\text{opt-offline}}$. Finally, the number of transactions that can be processes out of the validation set (`score`) and out of all transactions (learning set & validation set) (`k`) is measured for different $\beta's$, as illustrated in Figure 4.2. The `score` is an indicator of how many transaction of the validation set could be processed, whereas `k` represents the profitability curve. With an increasing β the learning set becomes larger, whereas the validation set shrinks. Starting with a low β, an increase yields a higher `k` and a higher score until $\beta = 0.2$. At this point, the critical ratio is reached. A further expanding of the learning set leads still to a higher `score`. Nevertheless, it is not beneficial anymore, since `k` is decreasing. At the maximum ($\beta = 0.2$), 40 % of the transactions haven been processed. Since, C_{limit} was set equal to $C_{\text{opt-offline}}$, it can be compared with the offline case, where all transactions can be processed with the given capital. Therefore, the algorithm is 0.4-competitive in average.

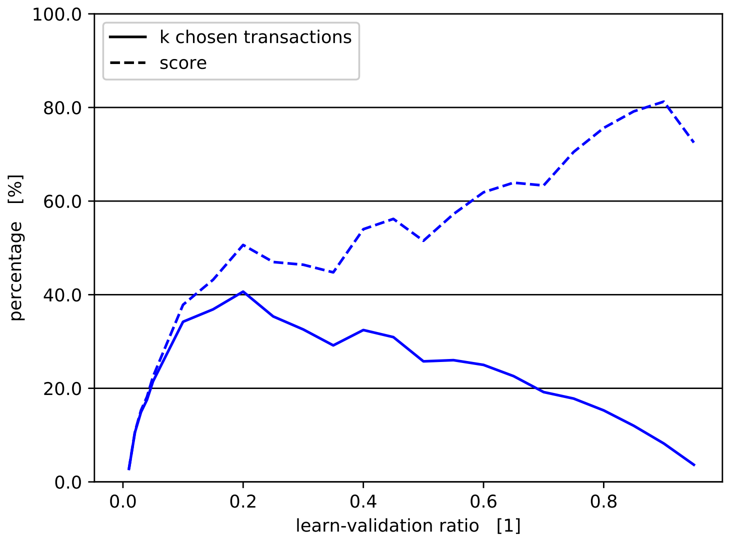


Figure 4.2: Number of processed transactions for different $\beta's$ in average for multiple sets. The `score` (dashed line) how many transactions out of the validation set could be processed. `k` (solid line) represents the number of chosen transactions divided by the total number of transactions (learning set + validation set)

CHAPTER 5

Conclusion

This project has examined various approaches to use PSP optimally. The main challenge is to resolve the problem of designing the optimal network structure. The question which needs to be answered is:

- How to design a network structure such that the profit is maximized with limited capital?

On the way of designing an algorithmic solution, the challenge was reduced to the problem of finding the optimal tree structure, which minimizes the capital demand when all transactions are processed. In a first step, the challenges of designing an approximation were discussed. While as shown common optimization schemes can not be applied, a probabilistic algorithm (`REO`) with an average complexity of $\mathcal{O}(n^4 logn)$ was presented. In the context of PSP, `REO` provides a closer approximation to the optimal capital than the best star structure. Various approaches of improving the run-time showed that the run-time can be reduced by one polynomial degree through parallelization. In the next step, the problem was extended from tree structures to connected graph structures. Since the number of edges are not fixed anymore, a cost function is introduced, which takes the number of edges and the capital demand into account. We focused on designing the optimal connected graph structure, which minimizes the cost when all transactions are processed. An algorithm was presented, which extends the optimal tree with cycles in case the cost can be reduced. Finally, the initialized question was discussed for tree structures. In the offline case it was focused on having a optimal strategy for transaction selection. Two different approaches were compared with each other, selecting greedily and selecting with probability. It was shown that they provide similar performance. Last but not least, the online case was considered. A 0.4-competitive algorithm was presented (evaluated through testing). It uses the idea of learning the tree structure and the payment distribution from previous transactions.

Significant progress could be made in the problem of minimizing the required capital for trees. The `REO` algorithm provides a significant closer approximation with 12 % less required capital than the star structure in average. Nevertheless, REO has imperfect scalability properties, with $\mathcal{O}(n^4 logn)$ in the average-case. Although, the algorithm is parallelizable to a certain degree, effort in infrastructure rises drastically with the number of nodes in the range of some thousands, and thus, it might become uneconomical. In the problem of maximizing the profit, using learning was proposed. Since, a certain pattern can be excepted in the transactions of a payment network (e.g Lightning), learning seems most promising. It was used in the sense of design the network structure and calculate the payment distribution out of the past transactions. The average competitive ratio of 0.4 from the analysis gives a clue of the potential of using a learning approach, since several parameters have space for optimization. For instance, when to accept transaction? In the presented algorithm, transactions are always accepted when they can be routed through the network which might not be optimal. Further, how to calculate the payment distribution (e.g. weighting factor, etc.) and how to assign a payment distribution on the network structure optimally? It requires further analysis. Another approach worth considering may include prediction of the future transactions and creation of the optimal network structure offline from the predicted transactions.

Bibliography

[1] G. Avarikioti, Y. Wang, and R. Wattenhofer, “Algorithmic Channel Design,” in *29th International Symposium on Algorithms and Computation (ISAAC), Jiaoxi, Yilan County, Taiwan*, December 2018.

[2] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer, “On scaling decentralized blockchains,” in *Financial Cryptography and Data Security*, J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125.

[3] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[4] “Ethereum white paper,” https://github.com/ethereum/wiki/wiki/White-Paper.

[5] “On sharding blockchains,” https://github.com/ethereum/wiki/wiki/Sharding-FAQs.

[6] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding,” in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 583–598.

[7] C. Decker and R. Wattenhofer, “A fast and scalable payment network with bitcoin duplex micropayment channels,” in *Stabilization, Safety, and Security of Distributed Systems*, A. Pelc and A. A. Schwarzmann, Eds. Cham: Springer International Publishing, 2015, pp. 3–18.

[8] “Lightning network,” https://lightning.network/.

[9] “Raiden network,” https://raiden.network/.

[10] Wikipedia contributors, “Dynamic programming — Wikipedia, the free encyclopedia,” 2018, [Online; accessed 21-November-2018]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Dynamic_programming&oldid=868874316

[11] Z. Shen, “The Average Diameter of General Tree Structures,” in *Computers & Mathematics with Applications, Elsevier Ltd.*, October 1998.

[12] N. Corporation, “Geforce gtx 960,” https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060/, November 2018.