Ministry of Science and Higher Education of the Russian Federation
ITMO UNIVERSITY

GRADUATION THESIS

DYNAMIC AND STATIC NETWORKS BASED ON STATIC OPTIMALITY

Author: Feder Evgenii Aleksandrovich_____

Subject area: 01.04.02 Applied mathematics and informatics

Degree level: Master

Thesis supervisor: Aksenov V.E., PhD _____

Saint Petersburg, 2022

Student Feder Evgenii Aleksandrovich
Group M42381c Faculty of IT&P

Subject area, program/major
Technologies of software engineering

Consultant(s):

a) Stefan Schmid, PhD, TU Berlin _____

Thesis received “_____” _____ 20___

Originality of thesis _____%

Thesis completed with grade _____

Date of defense “7” June 2022

Secretary of State Exam Commission Khlopotov M.V. _____

Number of pages _____

Number of supplementary materials/Blueprints _____

CONTENTS

INTRODUCTION

The amount of traffic in data center increases each day. Thus, researchers in networks algorithms want to reduce the communication cost of requests between the pairs of nodes. Nowadays, networks in data centers are static and optimized for the worst case load. One of the ways to adapt a static network structure for the traffic is to reduce the distance between frequently communicating nodes by making the network dynamic.

Despite the fact that most of the state-of-the-art algorithms target to build optimized static networks [2, 4–7] the recent research made it possible to dynamically change parts of the physical network [12]. Thus, it is now possible to have a dynamic network topology by paying additional cost for the changes. One of the arising questions is to come up with low-cost algorithms that dynamically change a network depending on the communication requests.

One of the most popular types of networks is *Binary Tree Network*. The state-of-the-art dynamic algorithms on networks [10, 14] are based on the self-adjusting trees [13, 15]. Unfortunately, in comparison to self-adjusting trees, these self-adjusting networks do not provide any theoretical guarantees and are heuristic. However, we want to care about these guarantees, for example, the static-optimality, i.e., the dynamic network has time complexity close to the static network built on the requests given in advance. In this work, we try to find some theoretical properties that make the data structures static-optimal. And then, we want to apply the properties to existing self-adjusting network algorithms.

In Chapter 1, we provide the list of definitions that we use in the work, for example, Self-Adjusting Networks and static-optimality. Also, we talk about the motivation and set up our goals. In Chapter 2, we present a linear algorithm that builds an optimal static structure for the uniform workload and provide a proof that the structure is optimal. Then, we apply the extracted properties of static-optimal algorithms to state-of-the-art algorithms from Chapter 1. In Chapter 3, we check how the proposed algorithms work on several synthetic and real workloads. In conclusion, we briefly overview the results presented and describe directions for further studies.

The problem description and all the algorithms and proofs were obtained by the student under the supervision. There is a related work by the student published on two conferences: *SPAA* [16] and *INFOCOM* [8] (rank A and A* respectively).

CHAPTER 1. DEFINITIONS AND GOALS

1.1. The topology of networks

Let computational nodes and the connections between them form an undirected graph $G = (V, E)$, where G belongs to $\mathcal{G}$ and $\mathcal{G}$ is a topology, or family of undirected graphs. In this work, we consider that the topology of the network is a binary tree.

Definition 1. *Connected topology* is a family of connected graphs, i.e., the graphs in which a route between any pair nodes exist.

Definition 2. *Tree topology* is a connected topology, where each but one (*root*) node has *a parent*. If each node has at most two children, we name that topology as *Binary tree topology*.

Also, in this work, we use the notion of the center of the network for the tree topology. Thus, we need to find it. We can do it with the help of a known algorithm named *centroid decomposition* [1].

Definition 3. Let T be an undirected tree. *Centroid* is a node v such that if we delete it from the tree, the resulting connected components (that are also the trees) have fewer than the half of the number of vertices from the original tree.

1.2. Optimal static network

Definition 4. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m) = ((u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m))$, where $u_i, v_i \in V$, be a sequence of *routing requests* to forward a packet in between the pairs of nodes u_i and v_i.

Our main goal is to build a graph that optimizes the total cost to process all requests. We try to solve this task in two ways: 1) static, i.e., the graph cannot change during the execution, or 2) dynamic, i.e., the graph can be changed.

Definition 5. *Static optimization task* is a task when given requests apriori we build a graph $G \in \mathcal{G}$ that does not change until the end of all requests. This graph G needs to optimize the total cost function $\text{sumCost}(\text{static}, T, \sigma) = \sum_{i=1}^{m} l_i$, where l_i is a length in edges of the path between the nodes of request σ_i and "static" means that we do not change T during requests.

One example of the static optimization task is "Building *Demand-Oblivious Network* task" [3]. There, the authors try to construct a static network that optimizes the cost of the "worst-case" request. If we know any additional information about requests, we can build a more optimal network. For example, if we know the dis-

tribution of requests, we can build *Demand-Aware Network* [3] that consider some pair of nodes to communicate more frequently than another.

Another approach to find the optimal static tree is to use dynamic programming [9].

Definition 6. The network configuration $OPT(\sigma) \in \mathcal{G}$ is *optimal* on the sequence of requests σ if sumCost(static, $OPT(\sigma), \sigma$) is less than sumCost(static, H, σ) for any configuration $H \in \mathcal{G}$, where "static" means that a network does not change.

Now, we explain on how to build an optimal tree given the list of requests in advance. We divide the problem to the intervals I. We name the nodes outside I as I^*. Consider $W_I(v) = \sum_{u \in I} count(u, v) + count(v, u)$ and $W_I = \sum_{v \in I} W_I(v)$. Dynamic programming stores n^2 states: $dp[i][j]$ represents the optimal tree that is build on the nodes from i-th to j-th (later, this interval is denoted as I). The transition from one state to another is the following: we iterate over the left and right ends of the interval I and try to make k-th node in the middle of I as a new root of the subtree (this node divides I onto I' and I''). For each choice of k we take the subtree with minimum cost and move further. The recalculation of this dynamic programming solution looks like this:

$$dp[i][j] = \min_{k=i}^{j-1} dp[i][k] + dp[k+1][j] + \sum_{v \in I'} W_{I'}(v) + \sum_{v \in I''} W_{I''(v)}$$

1.3. Self-Adjusting Networks

Sometimes we allow the network to change in between the requests. Thus, we get a dynamic optimization task.

1.3.1. General definition

Definition 7. In *a dynamic optimization task*, we assume that we can change our tree after each request. Before requests, we are provided with an arbitrary graph $T_0 \in \mathcal{T}$. Our task is to build an algorithm $\mathcal{A}$ that adjusts our network and minimizes the total cost, which is calculated as sumCost$(\mathcal{A}, T_0, \sigma) = \sum_{i=1}^{m} (l_i + cost(T_i, T_{i+1}))$, where l_i is a length of the path between the nodes in the request σ_i and $cost(T_i, T_{i+1})$ is an adjustment cost to reconstruct the network from step i, $T_i \in \mathcal{T}$, to step $i+1$, $T_{i+1} \in \mathcal{T}$.

There are many models that describe how to measure the adjustment cost. In this work, we consider the *Standard Model*.

Definition 8. In *Standard Model (SM)*, the cost of changing one edge to another one is equal to 1. Thus, $cost(G_i, G_{i+1}) = r_t$, where r_t is a number of edge changes between G_i and G_{i+1}.

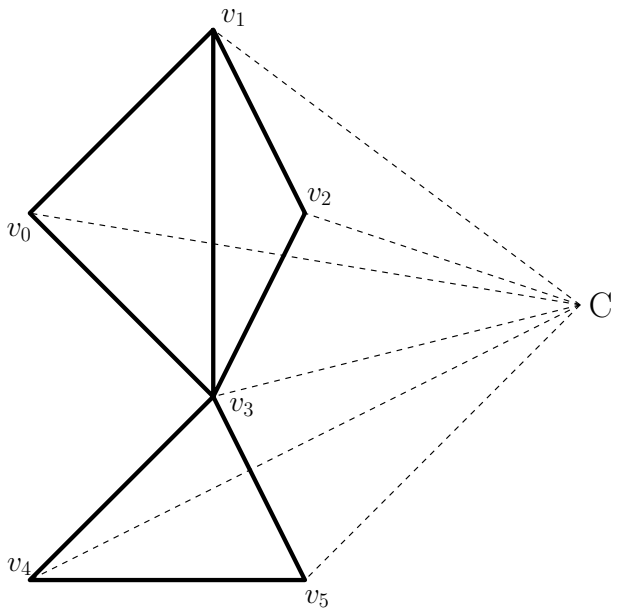

Figure 1 – All nodes are (logically and possibly physically) connected to a node C, which can request arbitrary topology changes and has complete information about the network and its statistics.

We assume that all changes to the network graph are controlled by a coordinator C. The coordinator is connected to all nodes from V and controls a structure of the network. We neglect the communication cost with the coordinator. An example of such a graph is presented on Figure 1.

Definition 9. The algorithm to perform the requests and adjust the network at each step from $T_i \in \mathcal{T}$ to $T_{i+1} \in \mathcal{T}$ is called *Self-Adjusting (Network) algorithm*.

Self-Adjusting algorithm $\mathcal{A}$ is called *c-statically optimal* if for each sequence of requests σ and for each start configuration G_0, $\text{sumCost}(\mathcal{A}, G_0, \sigma) \leqslant c \cdot \text{sumCost}(\text{static}, OPT(\sigma), \sigma)$

1.3.2. SplayNet

We start the discussion of the dynamic networks with the consideration of the most known state-of-the-art algorithm. In 2015, Schmid et al. [14] presented the study of self-adjusting data structures and introduced a new self-adjusting Binary Search Tree under routing requests called *SplayNet*.

Hereby, we shortly describe how it works. To start with, we hang the tree by the specially chosen node *root*. Consider a request (u, v). Node u asks the coordinator C for the common ancestor of u and v. Let it be node p. The route by which we pass a packet is $u - p - v$. After we find the route, we "splay" u to the place of p and then "splay" v to the position of the son of u. By that, SplayNet is a natural generalization of the classic Splay Tree algorithm, which "splays" communication partners to their common ancestor. On Figure 2 you can see an example on how to process a request from node 1 to node 5.

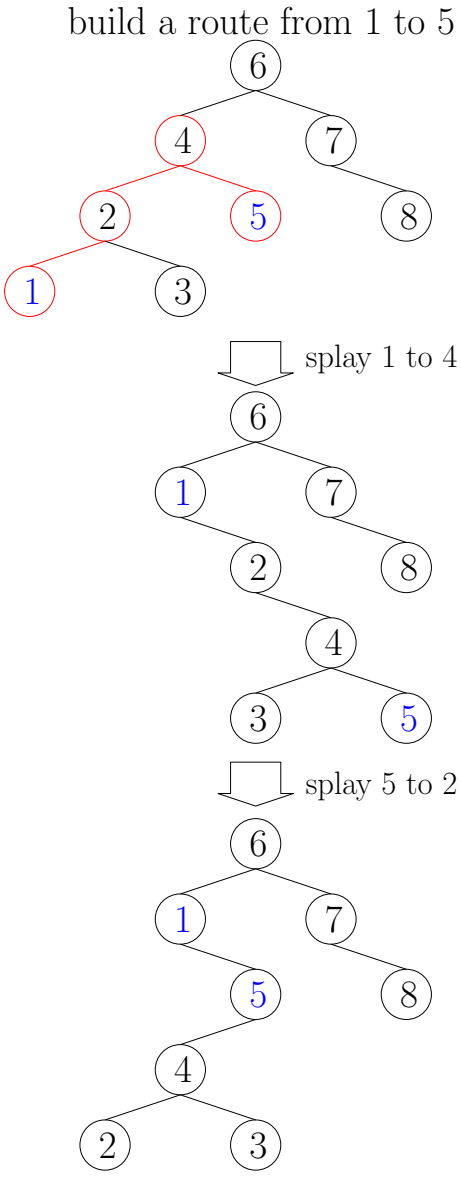

Figure 2 – Example on how to build a route and adjust in the SplayNet algorithm

1.3.3. Goals and Objectives

Nowadays, there are no dynamic communication networks that have theoretical proof of the static-optimality properties. Most of the state-of-the-art dynamic al-

gorithms are based on the self-adjusting trees. Unfortunately, in comparison to self-adjusting trees, these self-adjusting networks do not provide any theoretical guarantees and are heuristic. However, we need to care about the static-optimality. The goal of the work is to try to find some theoretical properties that make the data structures to be static-optimal. And then, apply the properties to existing self-adjusting networks.

Conclusions on Chapter 1

In this chapter, we introduced the main definitions used in our work. At first, we described two notions from the title: dynamic (self-adjusting) and static network algorithm. After that, we glanced over one of the state-of-the-art self-adjusting algorithm for the tree network topology, SplayNet. Finally, we show that the state-of-the-art algorithms do not provide static-optimal properties that could potentially increase the performance of the algorithms.

CHAPTER 2. NETWORKS BASED ON STATIC OPTIMALITY

In the previous chapter, we defined state-of-the-art self-adjusting network algorithms. However, as mentioned, the algorithms are heuristic, and they do not rely on the properties of the optimal static algorithms. In this chapter, we show what information can be extracted from optimal static structures and how it could be used in self-adjusting algorithms.

2.1. Periodic-optimal network

Our first approach is the first approach that comes to mind. We recalculate the optimal static structure using dynamic programming algorithm shown above after each request for the prefix. This solution has a problem that each reconfiguration of the network could costs a lot — up to $O(n^2)$ in the Standard Model, in the worst case. However, we still can use this idea but we have to amortize the reconstruction. We divide requests into epochs. Epoch continues until the total cost of the requests inside one epoch exceeds $O(n^2)$. At that moment, we can allow us to recalculate the static optimal structure and apply it to the network.

Unfortunately, this algorithm is no better than its predecessors, it is still based on the heuristic. The idea was based on the previous work on lazy algorithm [16]. In practice, it obviously adapts to the input workload. The experimental results on this dynamic network are shown in the Section 3.

2.2. Optimal static networks under the uniform workload

In the first chapter, we introduced an algorithm that can build an optimal static tree in cubic time. It is hard to analyze the resulting tree since, in general, it does not have any obvious patterns. Thus, we decide to look onto a more specific workload: the uniform workload.

Definition 10. The uniform workload is a sequence of requests in which each pair occurs only once.

Typically, one consider uniform workload is the one that takes each request uniformly at random. However, to simplify the proofs we use that unconventional definition. We can argue that if the number of requests is huge that definition perfectly approximates the random uniform distribution.

2.2.1. Experimental structure

The first idea was to implement a dynamic programming algorithm with the cubic complexity, find a pattern and proof that this pattern can build an optimal static

tree. Our preliminary expectation was that the optimal static tree should be the full binary tree hang by the root. Surprisingly, this is not the structure that we got. We got the binary tree that looks like the tree on the Figure 4.

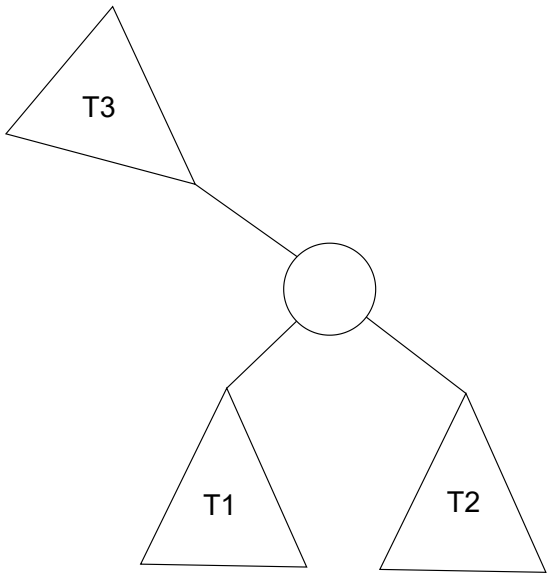

Figure 3 – Structure of the tree for uniform routing requests. Instead of invariant that all nodes have 2 sons, we can use property that *root* element have 3 sons (2 sons + a parent).

Now, let's look thoroughly into that structure.

— From the first glance at Figure 4, we can understand that $T_1, T_2, T_3, \ldots$ present a row of full binary trees. T_1 and T_2 have the same size, but sizes of $T_3, T_4 \ldots$ are different, and they decrease till the size 0 or 1.

— The next step of the exploration is to understand the structure of the resulting tree. Instead of two parts (like in binary tree) the structure has three approximately equal parts shown on Figure 3.

— Finally, we want to understand how many nodes there are in T_1, T_2, T_3 on Figure 3. After that, we could provide an algorithm with the linear time complexity instead of the previously presented cubic approach.

We start with counting the sizes of the subtrees. Intuitively, we want to make three subtrees as full binary trees rooted at the center node. Unfortunately, we have to deal with the remainder. Assume that we can find such k that $k = 3 \cdot (2^h - 1) + 1 < n$. Thus, we can build a symmetric network on k nodes, and we are left with $n - k$ nodes that should be distributed between subtrees. By considering the results of the cubic algorithm, we should add the nodes one by one to the T_3 until the size of T_3

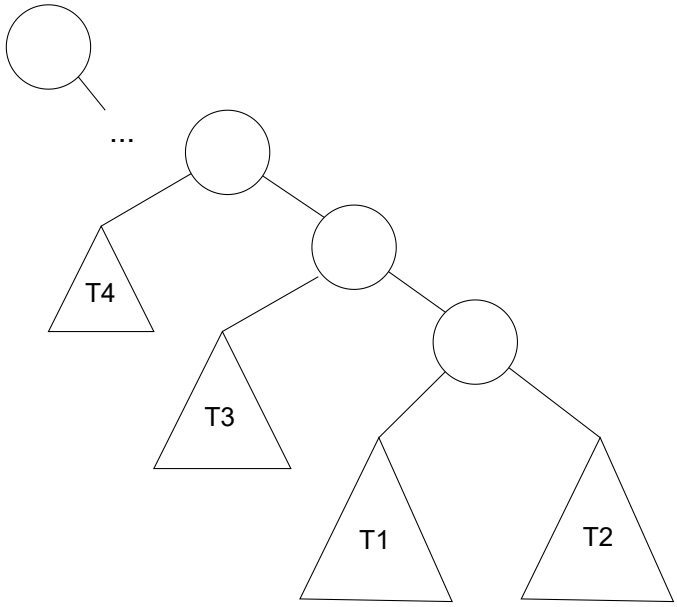

Figure 4 – Result of the experiments: tree for uniform requests. Each subtree is a full binary tree.

becomes $2^{h+1} - 1$. After that, we swap T_3 and T_1, and continue to move nodes to the T_1 subtree. When it becomes $2^{h+1} - 1$ again, we swap T_3 and T_2 and repeat.

This algorithm has a linear complexity, but we want to find sizes of subtrees in constant time. At first, we find the start depth h of our trees, like before. We need to consider some cases here, but we can omit them for the simplicity of the presentation. Then, if reminder have more than 2^h or $2 \cdot 2^h$, then we need to push it to T_1 or T_1 and T_2 respectively. All other nodes we store in T_3. The code is presented in the Appendix A.

After we understand the sizes, we need to construct our subtrees. T_1 and T_2 from Figure 3 are the easiest parts — they are just a full binary tree.

It becomes interesting With the subtree T_3. Assume that the size of T_3 is n. The algorithm iterates over bits: if the i-th bit is one, we create a tree with size $2^i - 1$ as the left child.

Listing 1 – Sketch of the construction algorithm. Its implementation is based on the recursion

```
cur_node = nullptr
for i = 1 ... maxbit(n):
    if i-th bit of n == 1:
        cur_node.right = Node(2 ^ i)
        cur_node = cur_node.right
        cur_node.left = full_binary_tree(2 ^ i - 1)
```

Unfortunately, we were not able to proof that this structure is optimal. Here we have a moment, that we don't know exact structure of the tree: this "many bits" representation hard to access in the proof. So, we start to work from the other side: from the theoretical properties that we want, i.e., that we have a special central node.

2.2.2. Theoretical structure

We start with the ideas that we come up from the experiments: it has started to be obvious that our structure needs to be like in Figure 3 and that the task is symmetric.

At first, we start with a simpler problem: we have only search requests in our binary tree network.

Definition 11. Consider S to be the root of the tree, then instead of routing requests we perform *search requests* from node S (i.e., $u_i = S$ for all i).

It is easy to show that the best static tree is a full balanced binary tree (Figure 5).

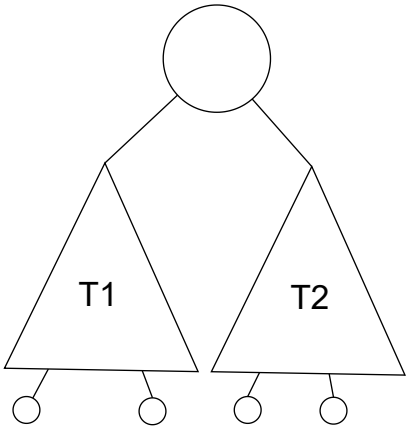

Figure 5 – Optimal static solution for the searching requests.

Lemma 12. A full balanced binary search tree is the optimal structure for the uniform search requests.

Proof. Consider some topology $\mathcal{T}$. We make small actions that reduce the total cost of the requests. And if we cannot improve anything, then, the structure is optimal. In the case of the search problem, the action is the following: if we have a node at level h_1 in T_1 and we have a space in T_2 (Figure 6), i.e., $h_2 < h_1$, then we can move the node from one place to another.

So, we do that action until we can. The final result will be a full binary tree with height h and some nodes at level $h + 1$.

□

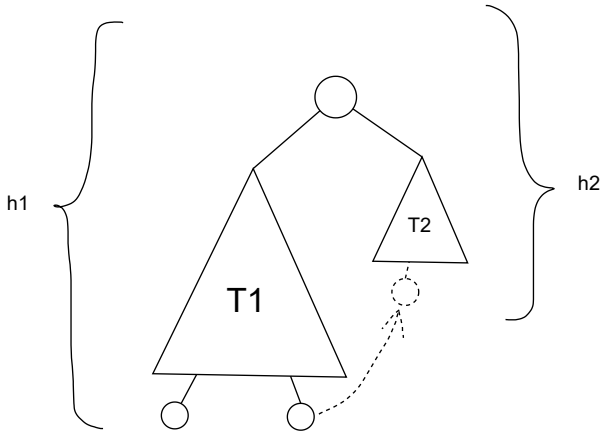

Figure 6 – Illustration for the proof of searching request.

We try to apply the same idea to the problem with routing requests. We find the centroid of the tree and instead of the action "move a node closer to the root" we perform the action: "move a node closer to the center of the network". In the end, we have a structure of three full binary trees with depth h. We store the remaining nodes from the left side on depth $h + 1$. Why from the left size? Because the total cost is lower when the nodes from the remainder are located closer to each other.

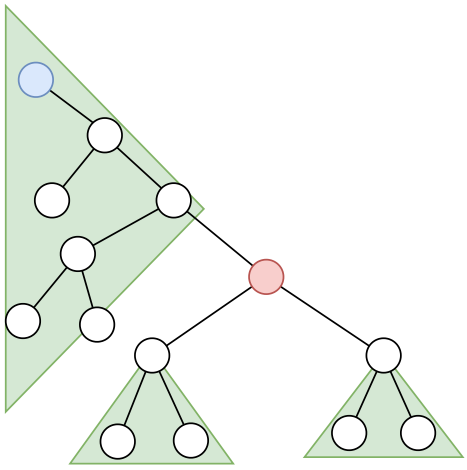

Figure 7 – Example of the static optimal structure for the uniform routing requests. In that case, red node is the center of the network and blue node is the root of the tree

Surprisingly, this structure has the same cost as the tree from Figure 3! You can see an illustration of the result in Figure 8. Also in Figure 7 you could see an example of that tree mapped into a binary searching tree. To make one from another, you need to rotate *the upper tree*.

Proof of the optimality of these structures is divided into several lemmas. Before that, let us introduce definitions that are going to be used in them.

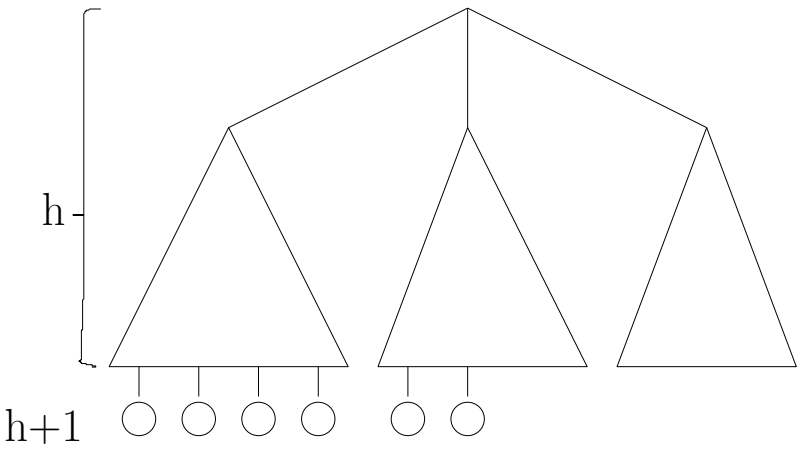

Figure 8 – Optimal structure of the network for the uniform requests. All subtrees are full and last $h + 1$ layer fills from the left. The highest node will be the center of the network and the left node will be the root of the tree.

Definition 13. *The center of the network* — a centroid that we have found before making an improvement operation. $d(v)$ in that case is denoted as *distance* from v to the center of the network. Also, C is used as the total cost of uniform requests on the tree.

Definition 14. *The potential of the edge* in network $\phi(e)$ — the number of paths going through this edge. If the tree is split onto parts with x and y nodes then $\phi(e) = x \cdot y$. *The potential of the whole tree* is $\Phi(T) = \sum_e \phi(e)$. Please note that in the case of the uniform workload the potential of the whole tree is equal to the total cost.

Lemma 15. Consider some edge e of the tree. This edge divides the tree into two parts: a tree with x nodes and a tree with y nodes. If we move a node from the first tree to the second tree, the difference in the potential of e will be $y - x + 1$.

Proof. Let cost before and after transformation be N_{before} and N_{after}, respectively. The cost before was $N_{before} = x \cdot y$. The cost after is $N_{after} = (x - 1)(y + 1) = x \cdot y + x - y + 1$. Thus, the difference is

$$N_{before} - N_{after} = y - x + 1$$

□

Now, we introduce an uplift operation of a vertex. To simplify, by that operation we can move a vertex to the higher level in another subtree. Consequently using this operation to different vertices the tree will be rolled up.

Definition 16. *Uplift operation* of a node u — a moving process of the node to an empty place v, such that $v = max_{d(v)}d(u) < d(v)$ and subtrees that lie on a path between u and v need to be full subtrees.

The algorithm for uplift operation is the following: we take the most distant leaf. Go to the parent. If another child is not a full binary tree of the corresponding size and has the space for the node, we try to move that leaf to the lower level of the child.

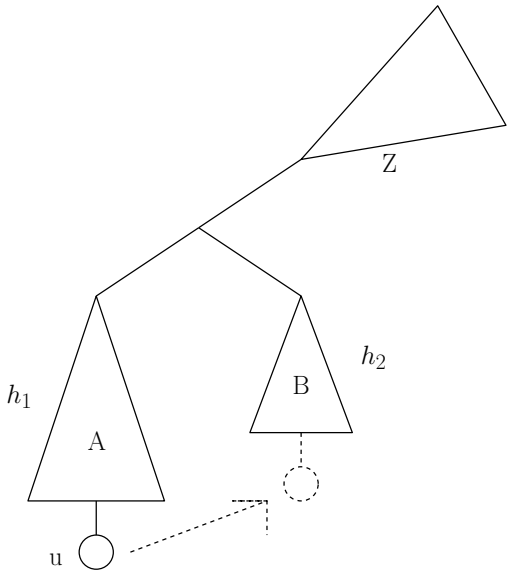

Figure 9 – Illustration of an uplift operation.

Lemma 17. Uplift operation does not increase the potential of the tree until $2^{h_1} + 2^{h_2} < \frac{n}{4}$, where h_1 is the height of the tree from which the leaf is and h_2 is the height of the tree where we move the leaf.

Proof. Suppose we move u to the position v. We start with an understanding what changes after the movement

— edge potentials change only on the path from u to v. Edges that are not on the route do not have their subtrees to change.

— A and B are full binary trees by the construction.

To prove that statement, we need to look to the difference in the potential function. So, we should consider all the edges on the path between u and v. Let $up(u, v)$

We want to prove that the potential decreases with this move, i.e., $\Delta\Phi > 0$. Lets count the left ($up(u, v)$) and right ($down(u, v)$) subpaths separately, i.e., the one that goes “up” from u and the one that goes down to v. Note that we do not

count the edge $(u, p(u))$ because it just moves to the place $(v, p(v))$ and its potential remains $1 \cdot (n-1)$.

$$\sum_{e \in u \to v} \Delta\Phi(e) = \sum_{up(u,v)} \Delta\Phi(e) + \sum_{down(u,v)} \Delta\Phi(e)$$

Consider $c_1, c_2, \ldots$ be the sizes of the other subtrees on the path $u - v$ that are lower than edge i in the tree. For the left part before the move: it was c_i nodes, and then it becomes $c_i - 1$. Also, other part will be $n - c_i$ and, $n - c_i + 1$ respectively. Here we use Lemma 15.

$$\Delta\Phi_{left} = (n - 2 \cdot c_1 + 1) + (n - 2 \cdot c_2 + 1) + \ldots \geqslant (1)$$

Now, we want to find c_1, $c_2, \ldots$ that will get the minimum difference in potential. Because constants are with minus, so we need to maximize constant. From the construction we know, that our node is the deepest node, so we don't have nodes that deeper that our in A tree. So the answer will be if all nodes that are siblings of our node for each lower subtree of edge. For the first edge it will be 3, for the second 7 etc.

$$(1) \geqslant (n - 2 \cdot 3 + 1) + (n - 2 \cdot 7 + 1) + \ldots + (n - 2 \cdot (2^{h_1+1} - 1) + 1)$$

$$= h_1 + \sum_{i=2}^{h_1+1} n - 2 \cdot (2^i - 1)$$

The same for the right part of the path. There the potential goes with plus sign, so we need to minimize them. This value is reached when we don't siblings. For the first edge it will be 1, for the second 3 etc.

$$\Delta\Phi_{right} = (2 \cdot c_1 - n + 1) + (2 \cdot c_2 - n + 1) + (2 \cdot c_3 - n + 1) + \ldots \geqslant$$

$$\geqslant (2 \cdot 1 - n + 1) + (2 \cdot 3 - n + 1) + (2 \cdot 7 - n + 1) + \ldots + (2 \cdot (2^{h_2} - 1) - n + 1)$$

$$= h_2 + \sum_{i=1}^{h_2} 2 \cdot (2^i - 1) - n$$

Let's insert the result for the left and right part together into the difference in the potential.

$$\Delta\Phi = \Delta\Phi_{left} + \Delta\Phi_{right} \geqslant h_1 + h_2 + 2 \cdot 1 - n + \sum_{i=h_2+1}^{h_1+1} n - 2 \cdot (2^i - 1)$$

$$= h_1 + h_2 + 2 \cdot 1 - n + (h_1 - h_2 + 1) \cdot (n + 2) - 2^{h_1+3} + 2^{h_2+2}$$

$$= h_1 + h_2 + 2 \cdot 1 - n + n + 2 + 2h_1 - 2h_2 + (h_1 - h_2) \cdot n - 2^{h_1+3} + 2^{h_2+2}$$

$$= 3h_1 - h_2 + 4 + (h_1 - h_2) \cdot n - 2^{h_1+3} + 2^{h_2+2}$$

Now, we use the statement that $2^{h_1} + 2^{h_2} < \frac{n}{4}$.

$$\Delta\Phi > h_1 + h_2 + 2 + 2 \cdot 1 + 2 + (4(h_1 - h_2) - 8) \cdot 2^{h_1} + (2(h_1 - h_2) + 2) \cdot 2^{h_2+1}$$

The worst case is when $h_1 = h_2 + 1$, so let's substitute it to the right part.

$$(4 - 8) \cdot 2^{h_1} + (2 + 2) \cdot 2^{h_1} = 0$$

□

The next operation that we need is to push a full binary tree to the center.

Definition 18. *Small shift of subtree* — the process of moving a constructed balanced tree (on the illustration it is A subtree) to the center. An illustration of that process you can see on the Figure 10. Also, shift could be with other orientation, but the proof for them remains the same.

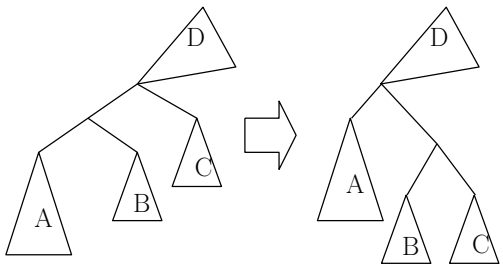

Figure 10 – Small shift.

Lemma 19. Consider A, B, C, D be the number of nodes in subtrees. Potential after small shift will decrease if $D > B$ and $A > C$ (Figure 10).

Proof. The first point is that the potential of the edges which come out of A, B and C subtrees do not change. Only one edge remains. Let's count changing of the potential of that edge.

$$(A + B + 1) \cdot (C + D + 1) - (A + D + 1) \cdot (B + C + 1) =$$

$$= AD + BC - AB - CD = (D - B) \cdot (A - C) > 0$$

□

Definition 20. *Full binary subtree of height h with the bottom level with x elements* is a full binary subtree of height h with something higher than the root of the subtree and at height $h + 1$ from left one goes nodes. Illustration of that structure you can see on Figure 11.

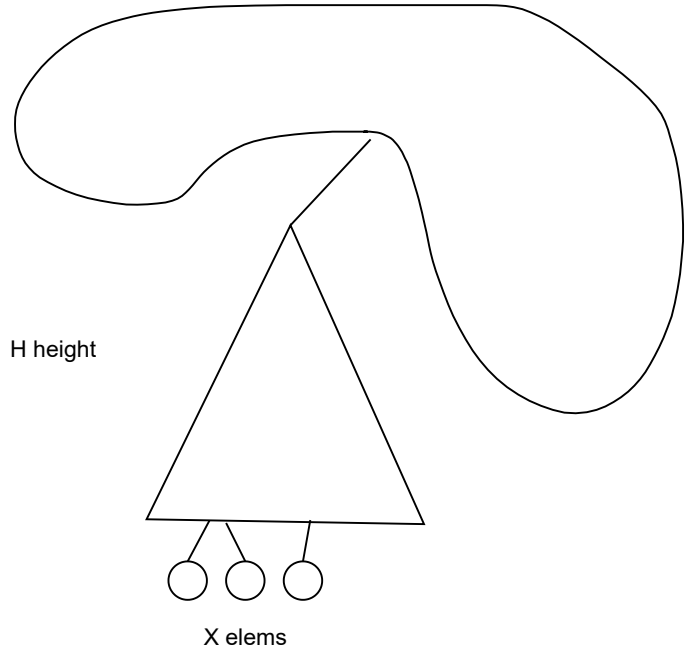

Figure 11 – Full binary subtree of height h with bottom part of x elements.

Lemma 21. If we move a node from the bigger full binary subtree of height h_1 with x_1 elements to smaller full binary subtree of height h_2 ($h_1 > h_2$) with x_2 elements, potential will not increase.

Proof. At the start of the proof we will count potential of arbitrary full binary subtree with bottom part, so I will omit indexes. Let's count the potential of each edge in the subtree. We have three types of edges:

a) Edge that connects the subtree with all other part. Its potential is $(2^h - 1 + x)(n - 2^h + 1 - x)$.

b) Edges that connect bottom nodes with tree. Their potential is $1 \cdot (n-1)$. Also we can notice that potential of edges from all bottom nodes the same. That's why the potential of that nodes is $x \cdot 1 \cdot (n-1) = x \cdot (n-1)$.

c) Edges that are inside the full binary subtree. It is hard to come up with the generic formula, but we can present a formula for nodes at the same level. Next, we look only on one level.
Consider i-th level and suppose that it connects nodes at height h and $h+1$.

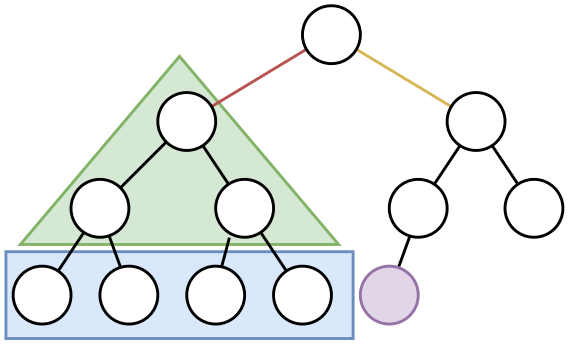

Figure 12 – Example of the full binary subtree with height 3 and 5 nodes at the bottom. Look onto 1 level. We have left and right edges. Left edge is full complected. Right edge is partially complected (only purple node at bottom level).

If we have no bottom nodes (our subtree just full binary) then it's simpler to count the potential of inner edges at level i. Let's look on one edge: the number of nodes in subtree is $2^{h-i} - 1$. Then, then potential function of that edge is $(2^{h-i} - 1)(n - 2^{h-i} + 1)$ and all edges are the same, so we can simply multiply by 2^i. The resulting summary for these edges is:

$$\sum_{i=0}^{h-1} 2^i (2^{h-i} - 1)(n - 2^{h-i} + 1) = 2^h h(n+2) - (2^h - 1)(2^{h+1} + n + 1)$$

If we have bottom edges, i.e., the subtree is not full binary. Edges of one level can divided into three parts:

— Fully equipped. The lower level is fully filled. Thus, the number of nodes in the subtree is

$$2^{h-i} - 1 + 2^{h-i} = 2^{h-i+1} - 1$$

— Partially equipped. Number of nodes in the subtree is:

$$2^{h-i} - 1 + x \,\%\, 2^{h-i} = 2^{h-i} - 1 + (x - \lfloor \frac{x}{2^{h-i}} \rfloor 2^{h-i})$$

— No equipped. The lower level is empty. Thus, the number of nodes in the subtree is

$$2^{h-i} - 1$$

Also, we need to calculate the number of edges of each type:

— The number of fully equipped edges is $\lfloor \frac{x}{2^{h-i}} \rfloor$;

— There is exactly one partially equipped edge per level;

— The number of non-equipped edges is $\lfloor \frac{2^{h+1}-x}{2^{h-i}} \rfloor$.

To get the total results we should multiply the number of edges times their potentials for each type of edges:

$$\lfloor \frac{x}{2^{h-i}} \rfloor \cdot (2^{h-i+1} - 1)(n - 2^{h-i+1} + 1) + 2^{h-i} - 1 + (x - \lfloor \frac{x}{2^{h-i}} \rfloor 2^{h-i}) +$$

$$+ \lfloor \frac{2^{h+1} - x}{2^{h-i}} \rfloor \cdot (2^{h-i} - 1)(n - 2^{h-i} + 1) = X$$

To bound the total cost, we iterate the sum over all the levels from 1 to $h - 1$ and expand floor operation as following: $x - 1 \leqslant \lfloor x \rfloor \leqslant x$. At first, we bound from below.

$$\sum_{i=1}^{h-1} X \geqslant \sum_{i=1}^{h-1} (\frac{x}{2^{h-i}} - 1 + 1) \cdot (2^{h-i+1} - 1)(n - 2^{h-i+1} + 1) +$$

$$+ (\frac{2^{h+1} - x}{2^{h-i}} - 1)(2^{h-i} - 1)(n - 2^{h-i} + 1)$$

$$= -n(5 \cdot 2^h + x - 5) + h(n(2^{h+1} + x + 1) +$$

$$+ 2^{h+2} + 2x + 1) - 3 \cdot 2^h x - \frac{5}{3} 4^h - 2^{h+2} + 4x + \frac{17}{3}$$

Then, we bound from above:

$$\sum_{i=1}^{h-1} X \leqslant \sum_{i=1}^{h-1} (\frac{x}{2^{h-i}} + 1) \cdot (2^{h-i} - 1 + 2^{h-i}) + (\frac{2^{h+1} - x}{2^{h-i}})(2^{h-i} - 1)$$

$$= -n(2^{h+1} + x - 1) + h(n(2^{h+1} + x - 1) +$$

$$+2^{h+2} + 2x - 1) - 3 \cdot 2^h x + 2^{h+1} - \frac{10}{3}4^h + 4x + \frac{7}{3}$$

As we can see, all parts that depend on x are equal in both parts.

$$f(x) = -nx + hnx + 2hx - 3 \cdot 2^h x + 4x$$

Let us count how the result changes if we increase x.

$$f(x+1) - f(x) = n(h-1) + 2h - 3 \cdot 2^h + 4$$

The biggest term in that difference is $n(h-1)$ that depends on h. So if $h_1 \leqslant h_2$ then we move node from first subtree to the second subtree and the difference in potential will be non-negative.

□

Theorem 22. The structure from Figure 8 is the optimal structure for the uniform routing requests.

Proof. Our proof is iterative. Consider the starting tree topology T. Step by step, we decrease the potential function that was described before. Let's describe that steps:

a) Make a centroid decomposition of T onto T_1, T_2 and T_3.
b) On each T_i we make centroid decomposition onto T_{i1}, T_{i2} and T_{i3}. Size of T_{ij} is less than $\frac{n}{4}$.
c) For each, T_i we take the deepest T_{ij} and make an uplift operation in it using Lemma 17. The result is the full binary tree. We can do that because the total count of nodes is less than $\frac{n}{4}$. After that, take small rotations to the center of the network using Lemma 19, but in T_i. D part will be bigger than B because we work inside T_i (Figure 10). If C is bigger that A that means that we are on our place and C is another T_{ij}.

d) Now, the structure is the following: not more than 9 subtrees that are connected. We balance them using the move of the node from the bigger subtree to the smaller one using Lemma 21.

Result structure is like we proposed before. □

2.3. Application of the property of optimal static network to the SplayNet

From the previous subsection, we understand that our intuitive understanding of the routing task on uniform workload was not correct. The center of the network is not the same as the root of the tree like all algorithms proposed before. Because of that, we patch the most popular dynamic algorithm, SplayNet — and get a new dynamic network structure named *Central SplayNet*.

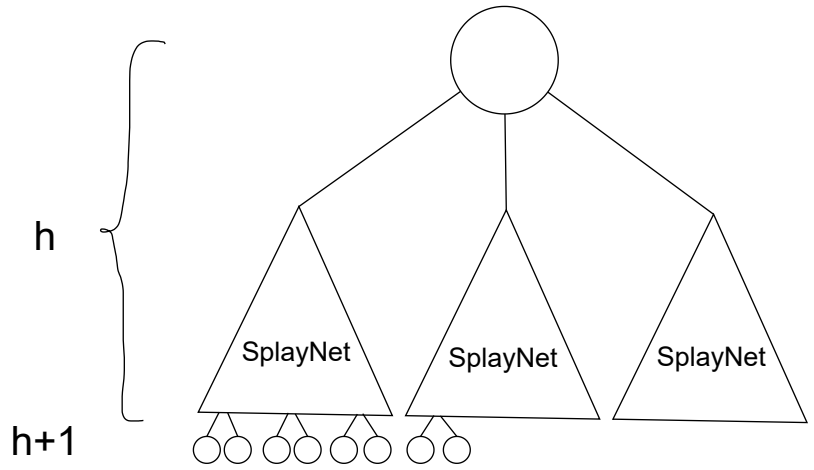

Figure 13 – CentralSplayNet algorithm construction.

Now, we have the structure similar to what is presented in the previous section (Figure 13). We maintain three SplayNet for each subtree of the center node. The main difference here from the original SplayNet is that we have three edges from the center: the left child, the right child, and the parent. Since this algorithm is again heuristic, we show its relevance by the experimental results, presented in the next chapter.

Conclusions on Chapter 2

In this section, we introduced two self-adjusting network algorithms and showed that optimal structures has important properties that can be used in dynamic algorithms on binary trees. Also, we proved that our structure for the uniform workload is optimal and that the idea: “the root of the tree is not the same as the center of the network”.

CHAPTER 3. SIMULATION BASED ON REAL DATA

In this chapter, we compare our two algorithms: periodic-optimal and CentralSplayNet, with optimal dynamic programming algorithm (that is adjusted after each request) and SplyNet presented in Chapter 1. Our main metric is the average cost of the request for each algorithm. For the request sequences we used open datasets (Facebook, HPC, pFabric, Microsoft ProjecTor from [11]) from different data centers to compare our algorithms.

3.1. Datasets

To measure the results we use real datacenter's traffic in which distribution of requests are far from random or arbitrary. One of the popular metrics of datasets is their temporal and non-temporal complexity [11]. Consider σ is the requests that we want to serve, $\Gamma(\sigma)$ is a random permutation of σ and $C(\sigma)$ is a compressed size of the σ. In paper they use gzip compression. Then, the temporal complexity is

$$TC = \frac{C(\sigma)}{C(\Gamma(\sigma))}$$

Intuitively, the value of the temporal complexity shows the following. If it is closer to 0 then similar requests comes in group. If it is closer to 1 then similar requests are distributed through the whole sequence. For non-temporal complexity consider the following definition: $universe(\sigma)$ is a set of nodes that are used as a source or a destination in some request from σ. Obviously, the universe should not always be equal to the size of the network. Then, $U(\sigma)$ is a sequence of requests of the size $|\sigma|$ and each source and destination are taken randomly from the $universe(\sigma)$. For example, $\sigma = \{(1,2);(3,10);(1,2);(3,10);\ldots;(1,2);(3,10)\}$, then $universe(\sigma) = \{1,2,3,10\}$, the size of the network is 10, and $U(\sigma)$ could be $\{(1,3);(3,10);(1,10);(10,3)...\}$. Using these definitions we define the non-temporal complexity:

$$NTC = \frac{C(\Gamma(\sigma))}{C(U(\sigma))}$$

In our work, we use three real traces from three different datacenters: pFabric, HPC and Microsoft ProjecTor (ML). You can see their properties on the Figure 14.

3.2. Metric

Due to the current limitations of network switch technologies in datacenters, we cannot measure the real time of how much our algorithms work. Instead, we

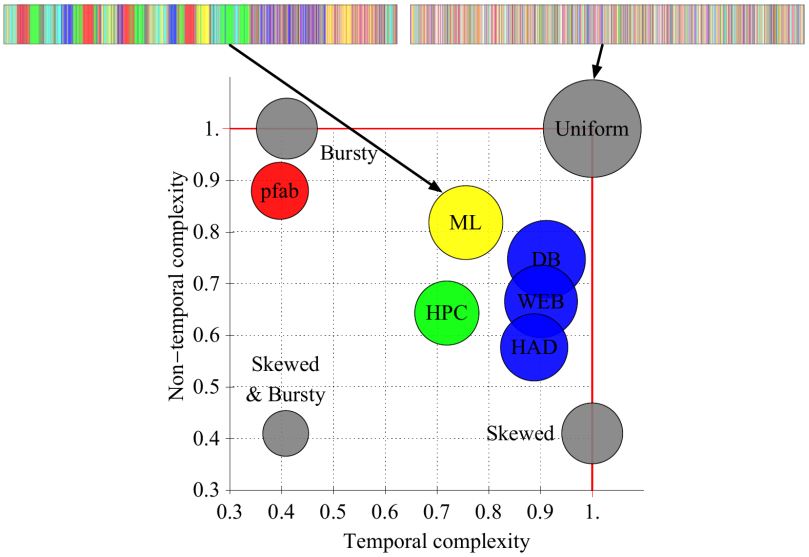

Figure 14 – The complexity map of six real traces and four reference points from Chen et al. paper [11]. pFab is a pFabric dataset, ML is Microsoft dataset, HPC is HPC.

present a simulation in a Standard Model that is described in the Chapter 1. We remind that the routing cost counts in edges. The final metric for i-th request is the average cost of all requests before i-th request:

$$avgCost(T_i, \sigma) = \frac{totalCost(T_i, \sigma)}{i}$$

3.3. Results of the simulations

For each of our experiments, we compare two known structures, i.e., the optimal static structure at each request (explained later) and SplayNet, and two others introduced in this paper structures, i.e., the periodic-optimal network and Central SplayNet. Also, for the uniform distribution, we look onto our theoretical periodic-optimal structure and show that static optimal structure tends to it.

Before going forward, the optimal structure need to be described. Consider we have sequence of request σ and prefix of size i was executed. Then static optimal structure for the prefix is a result of dynamic programming for that prefix. It means that after each request, the network is regenerated: for the first i requests it is one structure, for the first $i + 1$ it could be totally different structure. Despite that, it is a perfect baseline with which we can compare our results.

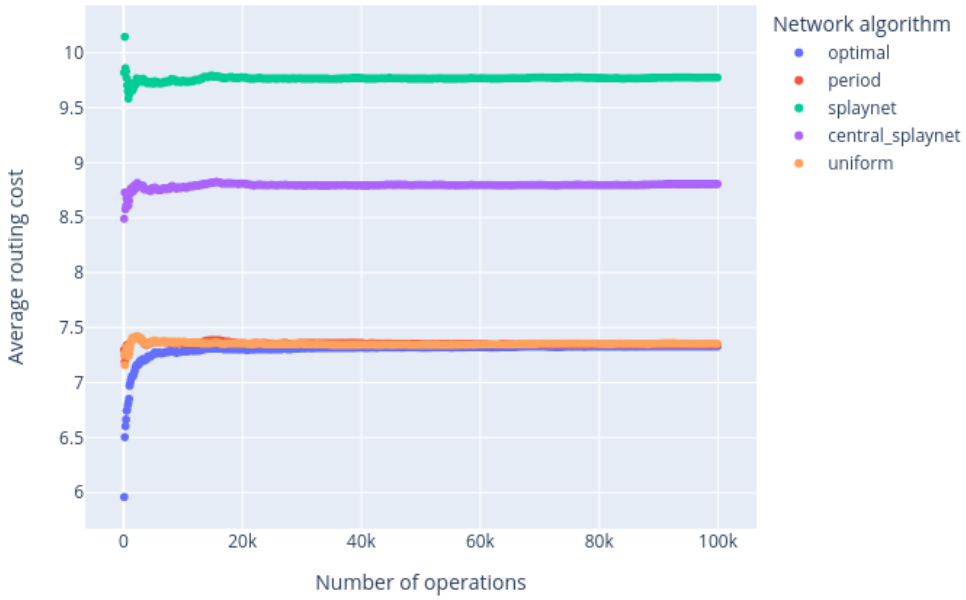

Figure 15 – Result plots for the uniform distribution workload.

3.3.1. Uniform distribution

We start with the uniformly distributed input sequence, i.e., we choose a request as a pair of nodes with uniform probability. The results are presented on Figure 15. Let us describe the result.

— At first, we can see that dynamic algorithms work worse than the static ones. It is understandable, since we do not have the locality, i.e, the probability of taking the same request to which we adjust is low.

— Secondly, we can see that our definition of uniform workload is a good approximation for the honest uniform distribution. Static optimal structure average cost goes to our structure cost.

Now let's look deeper into the plot and look at each structure separately.

At first, the optimal data structure increases at the start, because for the small amount of requests it could build more optimal structure (for example, if we don't use some nodes, they lie at deeper levels). After all nodes start to take part in requests, then the plot starts to become the constant. If we look at the structure, it doesn't change a lot, too.

Secondly, the period-optimal structure at the start don't reconfigure so the cost of the request increases. After some epochs, it starts to use properties of the distributions, so it starts to tend to the optimal network.

Then, we need to describe the uniform structure, which is our structure from the Figure 8. At the start, the distribution is not near “all pair communicates once”, so it loses to the optimal structure. But in the future it starts to move to the static optimal one. It means that our approximation of the requests is working.

Finally, let’s describe the behavior of the SplayNet and CentralSplayNet algorithms. At the start, they are trying to reconfigure starting configuration to the uniform. After some time, they make an almost optimal configuration.

3.3.2. Real workloads

In this subsection, we consider the workloads that were described in the previous subsection. You can see the simulation results on the Figures 16, 17 and 18. We can state the following results:

— The difference in the average cost between the periodic-optimal network and the static optimal network decreases with the number of requests.

— Central SplayNet works better on requests with high non-temporal complexity. Intuitively it is true since the main idea of Central SplayNet is in its lower depth, but if not all the nodes communicate, then this effect is as good as it can be.

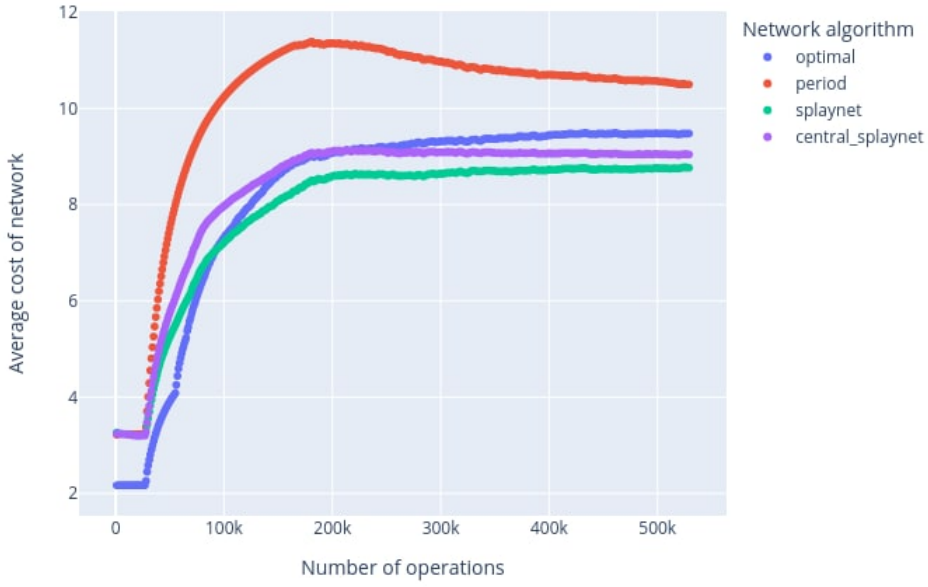

Figure 16 – Result plots of the simulation process for the HPC dataset

Now let us look deeper in each plot. At first, consider the HPC dataset from the Figure 16. Here we have 500 nodes and the workload has three phases of the distributions:

a) At the start (first 30k requests) of the dataset it uses only first 30 nodes, so all the plots are constant, and we can see that optimal is the better one, because on the small number of nodes optimal is better than the self-adjusting networks.
b) the second one (30k - 100k) is when we start to communicate nodes with big difference in their keys (for example, 5 and 450).
c) the third one is about communicating all nodes, but pairs are near each other. Then distribution of requests changes, and we see a fast cost increase cost in all plots.

Let's look onto each structure separately.

The highest one is the period optimal network. Why it has higher average cost? Because epoch size here is 25k requests, so we can see that after 25-50k requests it starts to increase slower and slower and, then, finally, decreases. It happens due to the fact that optimal static network starts to change slower and lazy structure slowly catches up to the optimal static structure.

Then, we look at the CentralSplayNet. Why it has higher average cost than SplayNet? It happens because that in the second phase we request the pairs of keys with high difference. It means that they will be from different subSplayNet, and they can't be as near as they can be in default SplayNet. But after this phase is ended, CentralSplayNet cost decreases and all the requests start to go to the same subSplayNet. We suppose that if requests continue with the similar patter, CentralSplayNet should beat SplayNet.

The next trace is the Microsoft dataset (Figure 17). The size of the network in that case is 121. This dataset is interesting because it has popular nodes that are asked more frequently. Because of that, static optimal network wins over others: we can pull popular nodes near the center. In SplayNet and CentralSplayNet these poplar nodes can be not at the top, so the average cost is higher. Also, CentralSplayNet is better than SplayNet because a lot of requests are near each other (half of the sequence).

The interesting thing here is period-optimal network. The start configuration for that network is the optimal static network for the uniform workload. Because the real workload is not uniform we start from some high cost and after each epoch,

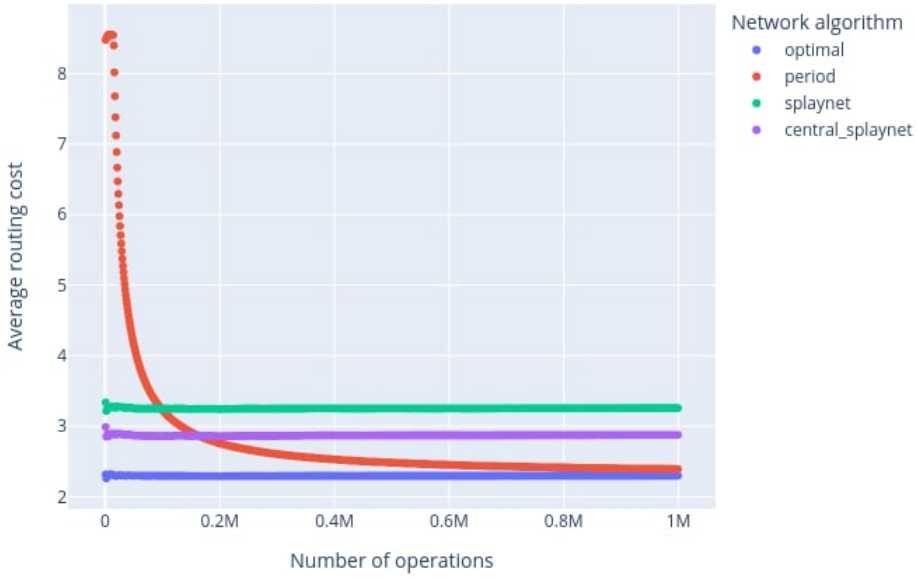

Figure 17 – Result plots of the simulation process for the Microsoft dataset

the network adapts to the input distribution. At the end, we can see that lazy adapted for the input distribution and cost is similar to the static optimal structure.

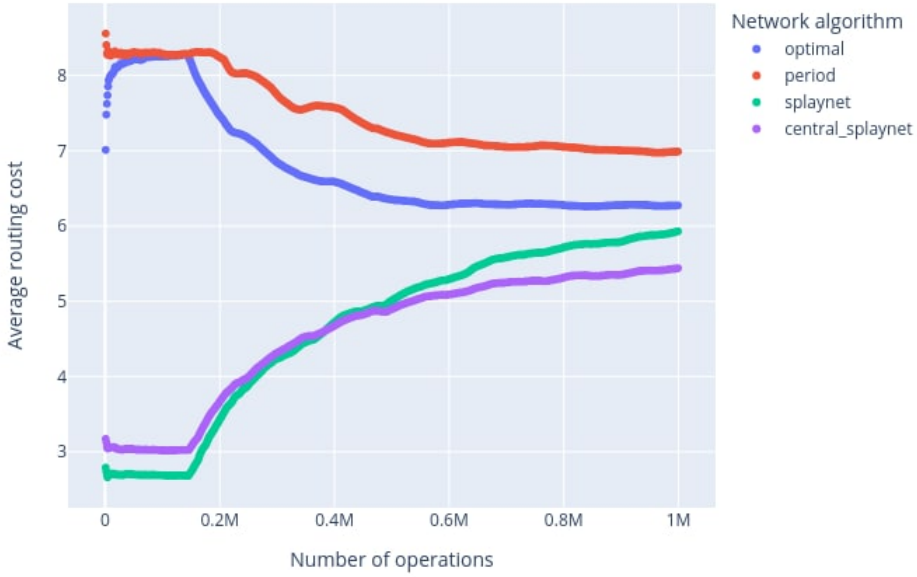

Figure 18 – Result plots of the simulation process for the pFabric dataset

The last trace is the pFabric dataset. The network has 144 nodes. The requests can be split into two parts: requests of the first part (140k requests) are repeatable, i.e., the same request can repeat 5-10 times, then the second part of requests is taken

from the normal distribution, when some nodes are requested more frequently than another.

The first part is not interesting, SplayNet and CentralSplayNet beat optimal and periodic-optimal structures, because on each pattern $(\{(x, y); (x, y); (x, y)...\})$ after first request structures splay nodes to each other. After that, the requests cost one. Central version works worse, because most of the pairs are from other subtrees.

Then we can see that the big difference in plots starts to decrease, because repeated patterns ended. And we can see that CentralSplayNet is better that SplayNet because a lot of pair keys are near each other.

Conclusions on Chapter 3

In this chapter, we show that self-adjusting network algorithms based on static optimal properties have competitive results with the current solutions for the task.

CONCLUSION

In this work, we looked onto optimal static networks. We proposed a new lazy version of Self-Adjusting algorithm that uses the static optimal structure. Also, we presented an algorithm that constructs an optimal static network under a uniform workload in linear time (also we prove that it is optimal). Finally, we found a property that leads to optimality and apply it to the known implementation of a self-adjusting network algorithm. Our version has better cost on synthetic and real workload. Please see the comparison in Chapter 3.

As for the future work, it would be interesting to consider the following research questions. At first, we would like to design new version of algorithm that use our splays and center property. Secondly, we want to apply the same ideas to the k-ary trees. Finally, we hope that ideas, found in this work, will help us to understand and to prove the optimality of self-adjusting algorithms.

REFERENCES

1 *Chiu J.* Centroid Decomposition. — 2015. — https://www.students.cs.ubc.ca/~cs-490/2014W2/pdf/jason.pdf.

2 *Avin C.*, *Mondal K.*, *Schmid S.* Demand-aware network designs of bounded degree // Distributed Computing. — 2019. — P. 1–15.

3 *Avin C.*, *Schmid S.* Toward demand-aware networking: A theory for self-adjusting networks // ACM SIGCOMM Computer Communication Review. — 2019. — Vol. 48, no. 5. — P. 31–40.

4 BCube: a high performance, server-centric network architecture for modular data centers / C. Guo [et al.] // Proceedings of the ACM SIGCOMM 2009 conference on Data communication. — 2009. — P. 63–74.

5 Beyond fat-trees without antennae, mirrors, and disco-balls / S. Kassing [et al.] // Proceedings of the Conference of the ACM Special Interest Group on Data Communication. — 2017. — P. 281–294.

6 *Al-Fares M.*, *Loukissas A.*, *Vahdat A.* A scalable, commodity data center network architecture // ACM SIGCOMM computer communication review. — 2008. — Vol. 38, no. 4. — P. 63–74.

7 *Knuth D. E.* Optimum binary search trees // Acta informatica. — 1971. — Vol. 1, no. 1. — P. 14–25.

8 Lazy Self-Adjusting Bounded-Degree Networks for the Matching Model / E. Feder [et al.] // 41th IEEE Conference on Computer Communications, INFOCOM 2020, Virtual Conference, May 2-5, 2022. — IEEE, 2022.

9 Locally self-adjusting tree networks / C. Avin [et al.] // 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. — IEEE. 2013. — P. 395–406.

10 *Oliveira Souza O. A. de*, *Goussevskaia O.*, *Schmid S.* CBNet: Minimizing Adjustments in Concurrent Demand-Aware Tree Networks // 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). — IEEE. 2021. — P. 382–391.

11 On the complexity of traffic traces and implications / C. Avin [et al.] // Proceedings of the ACM on Measurement and Analysis of Computing Systems. — 2020. — Vol. 4, no. 1. — P. 1–29.

12 Projector: Agile reconfigurable data center interconnect / M. Ghobadi [et al.] // Proceedings of the 2016 ACM SIGCOMM Conference. — 2016. — P. 216–229.

13 *Sleator D. D.*, *Tarjan R. E.* Self-adjusting binary search trees // Journal of the ACM (JACM). — 1985. — Vol. 32, no. 3. — P. 652–686.

14 Splaynet: Towards locally self-adjusting networks / S. Schmid [et al.] // IEEE/ACM Transactions on Networking. — 2015. — Vol. 24, no. 3. — P. 1421–1433.

15 The CB tree: a practical concurrent self-adjusting search tree / Y. Afek [et al.] // Distributed computing. — 2014. — Vol. 27, no. 6. — P. 393–417.

16 Toward Self-Adjusting Networks for the Matching Model / E. Feder [et al.] // Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures. — 2021. — P. 429–431.

APPENDIX A. COUNTING PHASE CODE OF THE STATIC OPTIMAL STRUCTURE CONSTRUCTION ALGORITHM

```
// we calculate start h and take it to the i
std::size_t mb = maxbit(n), i;
if ((n & (1 << (mb - 1))) != 0)
{
  i = mb - 1;
}
else
{
  i = mb - 2;
}

std::size_t start = (1 << i) - 1;
// all trees have start nodes
std::array<std::size_t, 3> sizes = {start, start, start};

std::size_t cn = n - 2 - 3 * start, ind = 0;

// until we have a lot of nodes we push them to bottom trees
while (cn > 1 + (1 << i))
{
  sizes[ind] += (1 << i);
  cn -= (1 << i);
  ind = (ind + 1) % 3;
}

// up == T_3, left = T_1, right = T_2
std::size_t up_size = sizes[2] + cn - 1, left_size = sizes[1],
   right_size = sizes[0];
```