

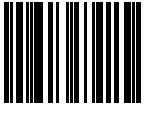
**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS**

**Исследование Алгоритмов Построения 3-Ограниченных Компьютерных Сетей
Учитывающих Нагрузку / Investigating Algorithms For Constructing 3-Bounded
Demand-Aware Computer Networks**

Обучающийся / Student Мартынов Павел Михайлович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group М34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан	
Мартынов Павел Михайлович	
25.05.2023	

(эл. подпись/ signature)

Мартынов
Павел
Михайлович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
21.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Мартынов Павел Михайлович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group М34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Исследование Алгоритмов Построения 3-Ограниченных Компьютерных Сетей Учитывающих Нагрузку / Investigating Algorithms For Constructing 3-Bounded Demand-Aware Computer Networks
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Основные вопросы, подлежащие разработке / Key issues to be analyzed

The primary goals of this article are to investigate existing data center load optimization algorithms and propose more efficient alternatives for a similar set of constraints, namely for the binary tree network topologies. Our focus lies on creating new algorithms which can alleviate restrictions of current methods and better accommodate real-world network demands. We also aim to enhance these algorithms with various heuristical approaches. Ultimately, this research seeks to introduce new efficient algorithms for constructing demand-aware networks, offering potential improvements in latency and efficiency.

Дата выдачи задания / Assignment issued on: 01.04.2023

Срок представления готовой ВКР / Deadline for final edition of the thesis 17.05.2023

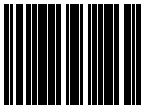
Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области фундаментальных исследований / Subject of fundamental research: да / yes

Тема в области прикладных исследований / Subject of applied research: нет / not

СОГЛАСОВАНО / AGREED:


Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
17.05.2023	

Аксенов
Виталий
Евгеньевич

(эл. подпись)

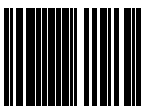
Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Мартынов Павел Михайлович	
17.05.2023	

Мартынов
Павел
Михайлович

(эл. подпись)

Руководитель ОП/ Head
of educational program

Документ подписан	
Станкевич Андрей Сергеевич	
22.05.2023	

Станкевич
Андрей
Сергеевич

(эл. подпись)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Мартынов Павел Михайлович

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования

Группа/Group М34391

Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Исследование Алгоритмов Построения 3-Ограниченных Компьютерных Сетей Учитывающих Нагрузку / Investigating Algorithms For Constructing 3-Bounded Demand-Aware Computer Networks

Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Research and development of approximate algorithms for constructing statically optimal computer networks with binary tree topology.

Задачи, решаемые в ВКР / Research tasks

Research of the existing approaches to the problem in question and the adjacent areas of research; Development of the algorithms that surpass existing ones in performance and quality of results; Development of the enhancing algorithms that can improve existing approximate solutions to provide better results; Implementation of the aforementioned algorithms; Thorough comparison of all the algorithms in question with respect to their performance and quality of the approximation.

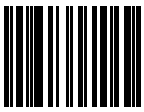
Краткая характеристика полученных результатов / Short summary of results/findings

Developed several algorithms for the construction of statically optimal computer networks for binary tree network topology, including the Maximum Spanning Tree algorithm specific to the problem in question, as well as multiple enhancement algorithms and heuristical approaches, leading to the creation of the genetic, or evolutionary, algorithm. All of the described algorithms were thoroughly tested and compared against each other as well as against pre-existing solutions, demonstrating their predominance in both performance and quality.

Наличие выступлений на конференциях по теме выпускной работы / Conference reports on the topic of the thesis

1. XII КОНГРЕСС МОЛОДЫХ УЧЕНЫХ (ОНЛАЙН ФОРМАТ), 03.04.2023 - 06.04.2023
(Конгресс, статус - всероссийский)

Обучающийся/Student

Документ подписан	
Мартынов Павел Михайлович	
25.05.2023	

(эл. подпись/ signature)

Мартынов
Павел
Михайлович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
21.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

CONTENTS

INTRODUCTION	6
1. Introduction To The Subject Area.....	8
1.1. Computer Networks	8
1.2. Demand Profile	8
1.3. Statically Optimal Network	9
1.4. Topological Limitations.....	9
1.5. Demand Profile Limitations.....	10
1.6. Examples of Certain Limited Problems	10
1.6.1. Chain Demand Graph.....	10
1.6.2. Unary Cycle Demand Graph	10
1.6.3. Complete Demand Graph	11
1.6.4. Complete Bipartite Demand Graph	12
Conclusions on Chapter 1	12
2. Approaches to the Problem.....	13
2.1. Base Approaches	14
2.1.1. Bruteforce Algorithm	14
2.1.2. Best For Binary Search Tree.....	14
2.1.3. Best BST Over Multiple Permutations	17
2.1.4. Greedy Algorithms	18
2.1.5. Maximum Spanning Tree	20
2.2. Enhancement Algorithms	20
2.2.1. Switching	20
2.2.2. Smashing	23
2.2.3. Compaction	24
2.2.4. Branch Switching.....	26
2.3. Evolutionary Approach	28
Conclusions on Chapter 2	30
3. Experiments	31
3.1. Real-World Workload Description.....	31
3.1.1. Social Network Workload.....	31
3.1.2. Microsoft Workload	31
3.1.3. pFabric Workload.....	32
3.1.4. ProjecToR Workload.....	32

3.2. Synthetic Workload Description	32
3.2.1. Dense Workload	33
3.2.2. Sparse Workload	33
3.3. Results	34
3.3.1. Synthetic Dense Workload Results	35
3.3.2. Synthetic Sparse Workload Results	35
3.3.3. Real-World Results	35
Conclusions on Chapter 3	37
CONCLUSION	45
REFERENCES	46

INTRODUCTION

Modern data centers host extremely intensive data flows that provide a heavy load onto the underlying systems, thus raising the challenging problem of optimizing the overall load onto the links and reducing latencies between hosts. Most of the data centers nowadays are designed with the worst-case scenario loads in mind, while completely ignoring the actual profile of the load that they are subjected to.

The suggested theoretical representation of this problem consists of the construction of the network, represented by a graph, from the provided demand profile. For the sake of simplicity and also to incorporate inherent limitations of modern networks, we would limit ourselves to the binary network topologies, however, some of the algorithms that are developed throughout the course of this work can be easily extended to work on a 3-bounded network topologies, n -bounded network topologies or even unbounded ones.

State-of-the-art algorithms that exist at the current moment usually provide either strive to provide the result that is optimal for some limited subset of allowed topologies of the resulting network [2, 13], thus limiting possible solution space tremendously, or tend to provide constant estimates for the optimality of the resulting network, which are most of the times too large to be applied in real-life cases. In this work, we provide a set of algorithms that both alleviate most of the constraints on the output topologies of the networks and provide better results on real-life network demand profiles than existing algorithms for this problem.

For example, one of the most well-known algorithms for this problem finds an optimal solution in the solution space that is limited to binary search trees, which, in most cases, is far from the optimal solution in an unconstrained solution space. By alleviating this constraint we allow ourselves to access more optimal solutions.

The algorithms proposed in this work mostly consist of the base algorithm with some heuristics applied to its output. We use the algorithm mentioned above as the base as well as the brand-new maximum spanning tree algorithm. Heuristics that are applied on top of the base algorithms include the algorithm that switches vertices in the resulting network to provide more optimal results, the algorithm that finds optimal solutions for small branches of the resulting network essentially finding the best possible rearrangements of vertices and edges within them and the condensing algorithm that changes the main structure of the resulting network graph by

condensing it into several large components and finding the optimal rearrangement for them.

We later derive an evolutionary algorithm based on the aforementioned heuristics as well as subtree rearrangements, which in turn allows us to employ the power of the simulated annealing algorithm.

CHAPTER 1. INTRODUCTION TO THE SUBJECT AREA

In this chapter, we define the basic definitions of the entities we are operating on in this work as well as describe the existing approaches to the problem in question.

1.1. Computer Networks

Computer networks are an integral part of modern information technology infrastructure. They allow computers and other digital devices to communicate with each other, share resources, and access information from remote locations. A computer network in a broad sense is a collection of interconnected devices such as computers, servers, routers, switches, and other network-enabled devices that communicate with each other using various networking protocols.

For the simplicity of the theoretical approach to this problem, we would represent any *computer network* as an undirected graph, where the set of vertices V represents hosts and the set of edges E represents all kinds of connections between the hosts. We consider an unweighted graph even though in real life links between different hosts can have drastically different characteristics.

1.2. Demand Profile

The *demand profile* of a network refers to the pattern of usage and traffic on the network over a certain period of time. It is a characterization of the amount and type of data that is transmitted across the network at different times of the day, week, or month.

The demand profile can be affected by various factors such as the number of users on the network, the type of applications being used, the time of day, and the location of the users. The nature of the demand profile for a network can vary widely depending on the specific network and its usage patterns. Some networks may experience relatively constant traffic levels throughout the day, while others may experience peaks and troughs of traffic based on specific events or user behavior.

In a theoretical model, *demand*, or *load*, can be defined as a matrix M of size $n \times n$ where n is the size of the network that it corresponds to. Value M_{ij} numerically denotes the amount of traffic between nodes i and j of the network — it can be expressed in the total size of the packets that are passed from one of the nodes to another, in the number of packets, in the percentage from the total number of packets passed through the system, or in any other way. In the theoretical model,

we abstract from the actual meaning of these numbers and would just operate on them.

An alternative way to represent the network load would be via a *demand graph*, which is a weighted graph corresponding to the demand represented as a matrix. In a general case, representation of the demand as a graph does not yield any benefits as it would just be a fully connected graph, but in case the load is not very sparse but rather follows some specific very heavy paths while leaving others unsaturated, the graph is a much more optimal representation of the network's demand.

1.3. Statically Optimal Network

A statically optimal network is a network design that is optimized for a specific set of traffic patterns and usage requirements. In other words, the network is designed to provide the best possible performance and efficiency for a particular set of conditions, without considering changes in traffic patterns or usage over time.

In the theoretical model in which we are working, the statically optimal network can be defined as such a network that if W is the demand graph represented by an adjacency matrix and D is a distance matrix of the constructed network, then:

$$\left(\sum_{0 \leq i, j \leq n} W_{ij} \cdot D_{ij} \right) \rightarrow \min \quad (1)$$

We would refer to the value of the sum above as the *cost* of the network, denoted as follows:

$$C(D, W) = \left(\sum_{0 \leq i, j \leq n} W_{ij} \cdot D_{ij} \right) \quad (2)$$

1.4. Topological Limitations

It is obvious that the solution to the problem of finding the statically optimal network for an arbitrary demand profile is always constant and equal to the full graph. However, this network topology does not make a lot of sense in real-world applications of the described problem, as we would not be able to connect all hosts with each other as their number grows, because most of the modern computers and servers have a very limited number of ports available on them.

Thus it seems natural to narrow down the set of network topologies that we are allowed to construct in the problem of finding statically optimal networks. We proceed to describe a few of them: line topology; tree topology; binary tree topology; and Δ -bounded topology, more specifically, 3-bounded topology.

1.5. Demand Profile Limitations

The set of constraints that we require to uphold the demand profile can also vary. We will not extensively focus on different kinds of such limitations, however, certain particularly interesting cases will be described in Chapter 2 of this work.

1.6. Examples of Certain Limited Problems

Let's explore the solutions for some of the limited versions of the statically optimal network search problem.

1.6.1. Chain Demand Graph

First, we consider a pretty simple case, in which the demand graph is a *chain*: $W_{ij} \neq 0 \Leftrightarrow i + 1 = j$. As an introduction to the more general problem that we would try to solve later, we impose the same topological limitation for the all problems described below: the resulting network would have to be a binary tree or 3-bounded tree topology.

The optimal tree for the chain is quite obvious — it would be the same chain, but now as a network graph. Its optimality can be easily proven — for each pair of adjacent vertices (i.e., such that $i + 1 = j$) we would have to add at least W_{ij} to the sum we minimize. Obviously, this is a lower bound. We can construct a graph that achieves that sum — it would be a tree, in which only vertices with adjacent numbers are connected. Such a graph matches the lower bound.

The case of the chain can be generalized on a so-called *chain forest*: a graph that contains one or more separate chains in it. You can also perceive it as a chain without some edges. Again, an example that achieves the lower bound can be constructed in a similar fashion.

1.6.2. Unary Cycle Demand Graph

This is a slightly more complicated case, in which we consider the demand graph to be a unary *cycle*:

$$W_{ij} = \begin{cases} 1 & , \text{ if } (i + 1) \bmod n = j \\ 0 & , \text{ otherwise.} \end{cases} \quad (3)$$

Again, as an introduction to the more general problem that we would try to solve later, we impose the same topological limitation for the all problems described below: the resulting network would have to be a binary tree or 3-bounded tree topology.

This case is a little bit more complicated. We need to “traverse” the network, which is the tree, in the following manner — from the 1st vertex to the 2nd, then from the 2nd to the 3rd, and so on, until we go from the n -th vertex to the 1st again. It can easily be seen that the problem of minimizing the length of this traversal is equivalent to the original problem.

Then, for each edge in the resulting graph, we can calculate the number of times we traverse it by doing the following: let sets A and B be the components of connectivity which emerges if we delete this edge, then a number of pairs of vertices adjacent in a cycle and that are in different sets would be equal to the number of traversals of this edge. Considering that both A and B are non-empty, we can see that the lower bound for the number of traversals of any given edge is two. Thus we have proven that for the cycle of n vertices, the minimal value of the sum would be $2 \cdot (n - 1)$.

Again, we can construct such a graph, that it would achieve that lower bound. If we order vertices from 1 to n (or in any shifted order) and build an arbitrary binary search tree on them, we achieve the required result.

1.6.3. Complete Demand Graph

In this scenario, we would examine the demand graph represented by a complete graph with all of the weights on the edges equal (without loss of generality, let's say that they are equal to 1).

Theorem 1. Optimal Network For Uniform Load The optimal network with a tree topology for a demand graph that is a complete graph on n vertices with equal weights is a *star graph*.

Definition 2. The star graph S_n of order n , sometimes simply known as an « n -star» [16], is a tree on n nodes with one node having vertex degree $n - 1$ and the other $n - 1$ vertices having a degree of 1.

Proof. It can be seen that there are $\frac{(n-1) \cdot (n-2)}{2}$ pairs of vertices that have a distance of 2 between them and $n - 1$ pairs that have a distance of 1. But it is obvious that every tree has exactly $n - 1$ pairs that have a distance of 1 — exactly

pairs of vertices connected by edges. This means that the star graph has a minimum sum of distances between all of the vertex pairs.

One can notice that we have not imposed any topological limitations on the network in this case. However, the case of the uniform load for the network limited to a binary tree is considered in great detail in previous research [15].

1.6.4. Complete Bipartite Demand Graph

This scenario is similar to the previous one, but we will look at the complete bipartite graph instead of a complete graph.

Theorem 3. Optimal Network For Uniform Bipartite Load The optimal network with a tree topology for a demand graph that is a complete bipartite graph $K_{n,m}$ with equal weights is constructed in the following way: we take one arbitrary vertex from each part and connect it to all of the vertices in the other part.

Proof. As in the previous theorem, we can easily see that there could only be $n + m - 1$ pairs of vertices such that distance between them is exactly 1. However, it is a well-known fact that in a bipartite graph distance between vertices in the same part is always odd, and between the vertices in different parts, it is always even. Thus, we can find a lower bound for the sum of distances between all of the pairs of vertices in this graph:

$$L = (n + m - 1) + 2 \cdot \left(\frac{n \cdot (n - 1)}{2} + \frac{m \cdot (m - 1)}{2} \right) + 3 \cdot (n \cdot m - n - m + 1) \quad (4)$$

It can easily be seen that the aforementioned graph exactly reaches this bound.

Conclusions on Chapter 1

With the introduction and sample cases out of the way, we can approach the problem that would be the main case we are looking at in this work: one where the network topology is limited to be a binary tree and there are no limitations on the demand profile.

CHAPTER 2. APPROACHES TO THE PROBLEM

In this chapter, we describe the approaches we employ to tackle the problem, starting with base approaches that are algorithms tailored to give a good initial approximation of the solution that can later be enhanced with subsequent applications of heuristical algorithms which would be described later in this chapter.

The use of multiple algorithms to solve optimization problems is a common approach that can provide several benefits. In many cases, a single algorithm may not be sufficient to provide the optimal solution or may be prone to getting stuck in local optima. By using a combination of algorithms, it is possible to overcome these limitations and achieve a better overall solution.

The base algorithm that is used to solve the problem of finding the statically optimal network may be chosen based on its effectiveness or its speed of convergence. Once the initial solution has been obtained using the base algorithm, additional enhancement algorithms can be applied to improve the solution. These enhancement algorithms may include techniques such as local search or heuristic algorithms, which refine the solution by making small adjustments to the initial solution.

One may think of base and enhancement algorithms as functions that have signatures which can be seen in the listing 1.

Listing 1 – Base and Enhancement Algorithm Signatures

```
Demand = List[List[int]]

AlgorithmResult = Tuple[Graph, int]

BaseAlgorithm = Callable[[Demand], AlgorithmResult]

EnhancementAlgorithm = Callable[[Demand, AlgorithmResult],
    AlgorithmResult]
```

As one can see, the idea behind the base algorithm is to create some initial solution for the given demand, and the idea behind an enhancement algorithm is to take some other solution (provided by either the base algorithm or another enhancement algorithm) as well as the demand profile and create a better solution on that basis.

Overall, the use of multiple algorithms to solve optimization problems can provide a more robust and reliable solution, as it takes advantage of the strengths of

different algorithms and reduces the risk of getting stuck in local optima. However, it is important to carefully select the algorithms and use them in a coordinated manner to ensure that they are complementary and not redundant. Different combinations of those approaches provide different results depending on the data set they are used for, however, the exact details of the experiments would be described in the next chapter.

2.1. Base Approaches

In this section, we discuss base algorithms, which are the fundamental building blocks for all the algorithms that we build in this work.

2.1.1. Bruteforce Algorithm

It is worth noticing that the algorithm of finding the statically optimal network for an arbitrary topology is NP-hard by reduction to MinLA [1].

We can try to solve the problem of finding the statically optimal network with a binary tree topology by iterating over all possible binary trees. There are exactly a $\frac{2n!}{(n+1)!}$ binary trees on n vertices, which is the n -th Catalan number [9].

Another approach, which is also asymptotically NP-hard is to use a SMT solver for the problem of minimization. Several SMT solvers have this functionality, most notably Z3 [10], however, this approach does not seem to gain any performance over the simple brute force.

There are some arguments for trying out different SMT solvers or even different kinds of solvers, but that falls out of the scope of this work.

2.1.2. Best For Binary Search Tree

This approach incorporates the power of dynamic programming by subsegments [17] to find a truly statically optimal network whose topology represents a binary search tree. Another way to phrase it would be: given the fixed permutation of the nodes, find the truly statically optimal network whose topology is a binary search network on the nodes in the order of the permutation. This approach was briefly mentioned in another article [13], however here we explain it in detail and give a deeper introduction to the implementation itself.

Let's strictly formulate our problem as follows: we are given n vertices, enumerated from 1 to n . We construct a binary search tree on those vertices and then make a number of pair-wise requests between them. Each request takes the

shortest path between the vertices. Those requests are represented with a matrix W of shape $n \times n$, where W_{uv} is equal to the number of requests between u and v . Given a matrix W , in a $\mathcal{O}(n^3)$ time construct a binary search tree, such that $\left(\sum_{u,v} W_{uv} \cdot d(u,v)\right) \rightarrow \min$.

We will solve this problem using dynamic programming on subsegments. For each segment $[l; r]$ we will construct such a BST that (t is the root of this tree):

$$\left(\sum_{l \leq i, j \leq r} W_{ij} \cdot d(i, j)\right) + \left(\sum_{l \leq i \leq r, j \notin [l; r]} W_{ij} \cdot d(i, t)\right) \rightarrow \min \quad (5)$$

It is easy to notice, that for a segment $[1; n]$ minimization of this value is equivalent to minimization of $\left(\sum_{u,v} W_{uv} \cdot d(u, v)\right)$ because the second term is equal to zero.

The base of dynamic programming would be values $D_{ll} = 0$ because for such subtrees *the inner* sum (first term of the value we're minimizing) would be equal to zero and sum *to the root* (second term) would also be equal to zero because distance from the root to the root is equal to zero.

Now let's describe **transition** of dynamic programming. For a segment $[l; r]$ we would iterate over all of the vertices in it, constructing a BST for each of them such that it has a root in that vertex (let's call it t) and is minimizing the aforementioned value on that segment. Let's prove, that the optimal tree with fixed root t is exactly constructed of an optimal tree on a $[l; t - 1]$ segment and an optimal tree on a $[t + 1; r]$ segment, both of which are connected to t :

$$\left(\sum_{l \leq i, j \leq r} W_{ij} \cdot d(i, j)\right) + \left(\sum_{l \leq i \leq r, j \notin [l; r]} W_{ij} \cdot d(i, t)\right) \quad (6)$$

Unfolding the terms we get:

$$\begin{aligned}
& \left(\sum_{l \leq i, j < t} W_{ij} \cdot d(i, j) \right) + \left(\sum_{t < i, j \leq r} W_{ij} \cdot d(i, j) \right) \\
& + \left(\sum_{l \leq i < t, t < j \leq r} W_{ij} \cdot (d(i, t) + d(t, j)) \right) \\
& + \left(\sum_{l \leq i < t} W_{it} \cdot d(i, t) \right) + \left(\sum_{t < j \leq r} W_{tj} \cdot d(t, j) \right) \\
& + \left(\sum_{l \leq i < t, j \notin [l; t] \cup (t; r]} W_{ij} \cdot d(i, t) \right) + \left(\sum_{t < j \leq r, i \notin [l; t] \cup (t; r]} W_{ij} \cdot d(t, j) \right)
\end{aligned} \tag{7}$$

Regrouping the terms we get:

$$\begin{aligned}
& \left(\sum_{l \leq i, j < t} W_{ij} \cdot d(i, j) \right) + \left(\sum_{t < i, j \leq r} W_{ij} \cdot d(i, j) \right) \\
& + \left(\sum_{l \leq i < t, j \notin [l; t]} W_{ij} \cdot d(i, t) \right) + \left(\sum_{t < j \leq t, j \notin (t; r]} W_{ij} \cdot d(t, j) \right) \\
& = \left(\sum_{l \leq i, j < t} W_{ij} \cdot d(i, j) \right) + \left(\sum_{t < i, j \leq r} W_{ij} \cdot d(i, j) \right) \\
& + \left(\sum_{l \leq i \leq t-1, j \notin [l; t-1]} W_{ij} \cdot d(i, t) \right) + \left(\sum_{t+1 \leq j \leq t, j \notin [t+1; r]} W_{ij} \cdot d(t, j) \right) \\
& + \left(\sum_{l \leq i \leq r, j \notin [l; t] \cup (t; r]} W_{ij} \right)
\end{aligned} \tag{8}$$

N.B.: cases of $l = t$ and $t = r$ are solved in a same manner.

It can be seen that this is equivalent to two independent terms that we already optimized via dynamic programming and another term, which is constant for a chosen vertex t . It is proven then, that we find an optimal tree on this segment if we iterate over all of the vertices on the segment and build a tree that is described above for each of them.

Now, let us describe the implementation of the aforementioned algorithm: we would iterate over lengths of segments from 1 to n , therefore considering $\mathcal{O}(n)$ segments on each iteration, each of which is processed in a $\mathcal{O}(n) \cdot T(n)$ time, where $T(n)$ is asymptotic of building optimal BST with a fixed root. If we would store a sum of all «outgoing» weights, i.e. $\sum_{i \in [l;r], j \notin [l;r]} W_{ij}$, then calculating a value that we're optimizing would take $\mathcal{O}(1)$ time and calculating this new sum for a chosen optimal value would take $\mathcal{O}(n)$ time, but it happens once during iteration over subsegments, so overall asymptotic of a dynamic step would be $\mathcal{O}(n) \cdot \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$. Total asymptotic would be $\mathcal{O}(n^3)$.

2.1.3. Best BST Over Multiple Permutations

It can be noticed that a very easy way to get better results for the problem of finding a statically optimal binary tree for the given demand profile may be to launch the aforementioned algorithm multiple times. To find a truly optimal result we would have to launch this algorithm for $n!$ times, which is, sadly, too much. However, launching it a polynomial number of times may yield some results while not sacrificing performance. This would be discussed in detail in the section with experiments.

A better and deeper description of the algorithm itself can be found in listing 2. The idea here is to achieve permuting the demand matrix without actually copying data over, thus just creating a so-called permutation lense, which overrides the access operator by providing the element by an index of a permutation element. By doing this we can easily leverage the power of the already described best BST algorithm in a section 2.1.2 in order to do this for multiple random permutations.

It is quite obvious that the time complexity of this algorithm is exactly $\mathcal{O}(N \cdot n^3)$, where N is the number of iterations that this algorithm will be launched for. If we imagine that N has some dependency from n , then we can, for example, launch n^2 iterations of the algorithm described in a section 2.1.2, thus keeping it polynomial. The results of this technique would be explored in detail in the third chapter of this work.

This base algorithm can be seen as the best effort to approach the really optimal tree, as after going through all of the $n!$ possible permutations for the sequence of nodes we would eventually find the optimal solution for the problem. Obviously, the more possible permutations we would go through, the closer we become to the optima.

Listing 2 – Best BST Over Multiple Permutations

```

class PermutationLense(Generic[E]):
    def __init__(
        self
        , permutation: List[int]
        , inner: List[E]
    ) -> 'PermutationLense':
        self.permutation = permutation
        self.inner = inner

    def __getitem__(
        self
        , index: int
    ) -> E:
        return self.inner[self.permutation[index]]

def best_bst_over_multiple_permutations(
    demand: List[List[int]]
    , how_many_permutations: int
) -> Tuple[Graph, int]:

    best_tree, best_cost = None, float('inf')

    for _ in range(how_many_permutations):
        permutation = gen_random_permutation(len(demand))

        demand_lensed = PermutationLense(permutation,
            PermutationLense(permutation, demand))

        tree, cost = find_best_bst(demand_lensed)

        if cost < best_cost:
            best_tree, best_cost = tree.reverse_permutation(
                permutation), cost

    return best_tree, best_cost

```

2.1.4. Greedy Algorithms

As we cannot get a good scaling effect for the best binary search tree algorithm described above because it only finds an answer for some fixed ordering of the vertices, we want to create some algorithm that would be scalable and produce *some* results for any demand profile provided.

In this section, we would describe the generic family of greedy algorithms that can be further investigated in greater detail in the following sections.

Let's say that we are given a demand graph D and we start with an empty set of edges E for the network we are constructing. Let's choose some function $f_0 : V \times V \rightarrow \mathbb{N}$. Among all of the vertices of the graph D we select such vertices a and b that the value of $f_0(a, b)$ is maximal among all of such pairs (or an arbitrary pair if there are several tied to be maximal) and add an edge (a, b) to the set E . Then we repeat this step, now constructing a new function $f_i : A \rightarrow \mathbb{N}$, where $A \subset V \times V$ such that $(x, y) \in A$ if and only if x and y are not connected by edges existing on this step, $p_i(x) < b$, and $p_i(y) < b$, where $p_i(v)$ is a power of vertex v before i -th step and b is a bound, 3 in our case (we can see that f_0 actually also obliges these rules). The listing 3 showcases this algorithm for the case of the single f which has limited inputs on every step. The notion of DSU in that listing stands for a Disjoint Set data structure [14].

Listing 3 – General Greedy Algorithm

```
def general_greedy(
    f: Callable[[List[List[int]], int, int], int]
    , bound: Optional[int] = None
    , demand: List[List[int]]
) -> Set[Tuple[int, int]]:
    n = len(demand)
    dsu = DSU(n)
    edges : Set[Tuple[int, int]] = set()

    for _ in range(n - 1):
        bx, by, bf = 0, 1, 0
        for x, y in product(range(n), range(n)):
            bound_requirement = (bound is None) \
                or (len(tree.adj[x]) < bound and len(tree.adj[y])
                    < bound)
            if x != y and not dsu.are_united(x, y) and
                bound_requirement:
                fv = f(demand, x, y)
                if fv >= bf:
                    bx, by, bf = x, y, fv
            edges.insert((bx, by))
            dsu.unite(bx, by)
    return edges
```

The above description is overgeneralized, but this broad definition of an algorithm allows us to experiment with f . Some obvious examples of f can include:

- $f(x, y) = W_{xy}$;
- $f(x, y) = \sum_i W_{xi} + \sum_i W_{iy}$.

One can easily see that the algorithm above works in a $\mathcal{O}(n \cdot (n^2 + \log n + \alpha(n))) = \mathcal{O}(n^3)$ time. Obviously, this can be optimized in the case of certain functions.

2.1.5. Maximum Spanning Tree

This algorithm spans from the aforementioned family of greedy algorithms and is the specification with a constant function $f_i(x, y) = W_{xy}$ with its space narrowed down to an available set of vertices. An observant reader can notice that this is equivalent to finding a *maximum spanning tree* in a demand graph, which can be done in a greatly reduced time than the asymptotic described in the previous section.

There are a plethora of well-known algorithms for finding minimum spanning trees [3, 5, 8], which can easily be adapted to find a maximum spanning tree.

This yields a very fast algorithm, especially for sparse demand profiles. However, it should be noticed that this algorithm does not account for any of the second-order demand effects in the profile, thus failing to capture a lot of patterns that occur in the demand profile. However, this is a good and efficient base algorithm for future enhancement algorithms and heuristics.

2.2. Enhancement Algorithms

In this section, we discuss the enhancement algorithms, which are meant to be applied on top of some existing solutions produced by either base algorithms or other enhancement algorithms.

2.2.1. Switching

The switching algorithm is the first of the enhancement algorithms that we cover in this work. The main idea is that we can switch two vertices in a network that are connected by an edge if it makes the cost of the network better. The computations required to do that are not hard and costly, which means that we can perform this algorithm without having to sacrifice a lot of performance.

Let's describe a procedure of doing one switch operation on a network N for an edge (a, b) . Let's say that the components into which the network splits if this edge is removed are called L and R . For clarification, these are the sets that the

below holds:

$$\begin{cases} a \in L \\ b \in R \\ L \cup R = N \\ L \cap R = \emptyset \end{cases} \quad (9)$$

Let's calculate a cost for the current state of the network D :

$$\begin{aligned} C(D, W) &= \sum_{i \in L} \sum_{j \in R} D_{ij} W_{ij} = \left(\sum_{i \in (L \setminus a)} \sum_{j \in (R \setminus b)} D_{ij} W_{ij} \right) + \\ &+ \left(\sum_{i_1 \in (L \setminus a)} \sum_{i_2 \in (L \setminus a)} D_{i_1 i_2} W_{i_1 i_2} \right) + \left(\sum_{j_1 \in (R \setminus b)} \sum_{j_2 \in (R \setminus b)} D_{j_1 j_2} W_{j_1 j_2} \right) + \\ &+ \left(\sum_{i \in (L \setminus a)} W_{ia} D_{ia} \right) + \left(\sum_{i \in (L \setminus a)} W_{ib} D_{ib} \right) + \\ &+ \left(\sum_{j \in (R \setminus b)} W_{aj} D_{aj} \right) + \left(\sum_{j \in (R \setminus b)} W_{bj} D_{bj} \right) \end{aligned} \quad (10)$$

And now for the network where a and b are switched in their places (D'):

$$\begin{aligned} C(D', W) &= \sum_{i \in L} \sum_{j \in R} D'_{ij} W_{ij} = \left(\sum_{i \in (L \setminus a)} \sum_{j \in (R \setminus b)} D'_{ij} W_{ij} \right) + \\ &+ \left(\sum_{i_1 \in (L \setminus a)} \sum_{i_2 \in (L \setminus a)} D'_{i_1 i_2} W_{i_1 i_2} \right) + \left(\sum_{j_1 \in (R \setminus b)} \sum_{j_2 \in (R \setminus b)} D'_{j_1 j_2} W_{j_1 j_2} \right) + \\ &+ \left(\sum_{i \in (L \setminus a)} W_{ia} D'_{ia} \right) + \left(\sum_{i \in (L \setminus a)} W_{ib} D'_{ib} \right) + \\ &+ \left(\sum_{j \in (R \setminus b)} W_{aj} D'_{aj} \right) + \left(\sum_{j \in (R \setminus b)} W_{bj} D'_{bj} \right) \end{aligned} \quad (11)$$

It can be noted that:

$$\begin{cases} \forall i \in (L \setminus a) : D_{ia} = D'_{ia} - 1 \\ \forall i \in (L \setminus a) : D_{ib} = D'_{ib} + 1 \\ \forall j \in (R \setminus b) : D_{aj} = D'_{aj} + 1 \\ \forall j \in (R \setminus b) : D_{bj} = D'_{bj} - 1 \\ \forall i, j \neq a, b : D_{ij} = D'_{ij} \end{cases} \quad (12)$$

Given this, we can easily compute $C(D, W) - C(D', W)$:

$$\begin{aligned} C(D, W) - C(D', W) &= \left(\sum_{i \in (L \setminus a)} \sum_{j \in (R \setminus b)} (D_{ij} - D'_{ij}) W_{ij} \right) + \\ &+ \left(\sum_{i_1 \in (L \setminus a)} \sum_{i_2 \in (L \setminus a)} (D_{i_1 i_2} - D'_{i_1 i_2}) W_{i_1 i_2} \right) + \\ &+ \left(\sum_{j_1 \in (R \setminus b)} \sum_{j_2 \in (R \setminus b)} (D_{j_1 j_2} - D'_{j_1 j_2}) W_{j_1 j_2} \right) + \\ &+ \left(\sum_{i \in (L \setminus a)} (D_{ia} - D'_{ia}) W_{ia} \right) + \left(\sum_{i \in (L \setminus a)} (D_{ib} - D'_{ib}) W_{ib} \right) + \\ &+ \left(\sum_{j \in (R \setminus b)} (D_{aj} - D'_{aj}) W_{aj} \right) + \left(\sum_{j \in (R \setminus b)} (D_{bj} - D'_{bj}) W_{bj} \right) = \\ &= \left(\sum_{j \in (R \setminus b)} W_{bj} - W_{aj} \right) + \left(\sum_{i \in (L \setminus a)} W_{ia} - W_{ib} \right) \end{aligned} \quad (13)$$

This value determines how much the cost changes if a and b are switched. Thus, if we compute this value, we can make an informed decision on whether we want to switch the vertices or not.

This is the very essence of the switching algorithm — we would call an *iteration* of the switching algorithm one passage along all of the edges in the graph. Based on the above value formula we can compute it in $\mathcal{O}(n)$ time, thus the iteration can be completed in $\mathcal{O}(n^2)$ time.

We would later experiment with a different number of iterations for the switching algorithm, but for now, it is worth noticing that only a handful of those iterations are sufficient to greatly improve the cost of the network.

2.2.2. Smashing

The switching is a good algorithm but it does not change the structure of the network. The structure of the network is a graph that represents it without respect to the ordering of the vertices, and it can easily be seen that no number of switching iterations could change that. However, if we want to get closer to the optimal network, we need to find a way to change the structure of the binary tree representing it.

The first attempt at doing that is called *branch smashing*. Recalling that we can find a truly statically optimal binary tree for some small number of vertices in a reasonable amount of time, we can construct an algorithm that would go over each subtree of the whole network which is of size N or less, where N is some constant chosen ahead of time and find a truly statically optimal binary tree for each of them.

This would take a time proportional to $\mathcal{O}(N^3 \cdot N!)$, which is a constant depending on the parameter N , which we select ahead of time. Obviously, we would not select any big N as we care for the algorithm's performance.

The high-level overview of this algorithm is as follows: we would find all of the branches with size less or equal to N that are maximal by inclusion, i.e., if there's a branch of size $N_1 \leq N$ and it has a sub-branch of size $N_2 < N_1$, then we would only consider the first branch and not the sub-branch, and after that would find the optimal network structure for each and any of them.

One may wonder how we can find such branches and what would be the asymptotics of that algorithm. Conveniently, we can find all such branches by first finding two nodes such that the path between them is the diameter of the network. As a reminder, the diameter of a graph is the length of the shortest path between the most distanced nodes. We can find such a pair of vertices using two depth-first searches of a graph launched sequentially, thus leading to complexity $\mathcal{O}(n)$. After that, for each of those vertices, we would root the tree on that vertex and calculate the sizes of subtrees, remembering each one that matches the criteria. This is also done via a series of depth-first searches and amounts for a total asymptotic of $\mathcal{O}(n)$.

Observant readers may also question this approach for it may not always find all the branches that match the criteria. That is true; however, this statement holds for graphs with a diameter of at least N , which is shown by a theorem below.

Theorem 4. Finding Branches Of Size N Or Less If the graph has a diameter of at least N , then a depth-first search from two vertices distances between which is equal to the diameter will find all of the branches of size N or less.

Proof. It can be seen that the only way in which we would not find some branch in a depth-first search is when both of the starting vertices lie within that subtree. However, that contradicts the condition that the diameter of a graph is at least N because the maximum distance between two vertices within a tree of size N is equal to $N - 1$.

Overall, it can be seen that this algorithm runs in $\mathcal{O}(n + N^3 \cdot N!)$ time. Different values of N are explored in detail in the chapter with experiments.

This algorithm, unlike the switching algorithm, can change the structure of the network it is applied to, however, for large values of n the *core* part of the network structure is unaffected. To tackle this issue we would try to enhance this approach further.

2.2.3. Compaction

Compaction is a technique derived from ideas of the divide-and-conquer approach [4]. The basic idea is taken from the branch smashing algorithm but now is applied on a global level.

To do compaction, we first split the network into a number of chunks that are chosen ahead of time, let's call it N , with roughly the same amount of elements while remembering how they were connected. Then, we do compactions recursively on the chunks, possibly rearranging the vertices inside them. Afterward, we connect the chunks again, trying all of the possible combinations of connections between them, possibly redrawing some edges.

The time complexity of this algorithm depends on the method we employ in order to split the graph into chunks. The particular technique that we would tend to use for the implementation of this algorithm would be centroid decomposition [6]. We be repetitively choosing a centroid on the subtrees to create the subgraphs with approximately equal sizes. On each iteration we would add at least one new component, thus bringing the complexity of splitting the tree into components to $\mathcal{O}(N \cdot n)$. It can be seen that the algorithm for just splitting has a $\mathcal{O}(N \cdot n \log n)$ time com-

plexity because we would have to do this component splitting on each level of the recursion. This is a usual time complexity for a lot of the divide-and-conquer algorithms.

Now, let's discuss how joining components would work. We define a notion of *connectors*, which are simply the vertices in a component that previously had an edge to a different component. Thus, for a tree that is split into N components, we would have $2N$ connectors. We would now iterate over all of the possible ways to arrange those connectors into pairs joined by an edge such that the whole tree is connected. This is done by ensuring the connectivity of the tree with the help of Disjoint Set [14] of size N . It is a well-known fact that there are exactly $\prod_{i=1}^N (2N - 2i + 1)$ ways to split $2N$ elements into pairs.

Thus, if we deem N as a constant, it can be seen that the algorithm described has a time complexity of $\mathcal{O}(n \log n)$ with an obviously enormous constant that depends on N .

As this algorithm is quite hard to understand, listing 4 showcases the rough outline of this algorithm's implementation. This algorithm can be roughly divided into three main stages: splitting, recursion, and connecting. Let's go over the steps of this algorithm in detail.

First, as we split the tree into components, we create a queue of components to split. We would iteratively extract one element from the queue, getting an index of the next component that we would split. Next, we find a centroid for just that component. As a reminder, a centroid is such a vertex, that if there are n vertices in a tree, each of the components into which the tree would be divided if we delete the centroid would have the size of no more than $\frac{n}{2}$. After finding a centroid, we split the component we were looking at into two parts. We remove the edge between the components and record it. One of the components would contain one of the subtrees spurring from the centroid, and the other one would contain the centroid itself and all of its other subtrees. Such a division ensures that by the Dirichlet principle sizes of components would not differ from each other more than two times. After that, we add newly created components to the queue and continue iterations until we have enough components.

Secondly, we launch the same algorithm recursively from all of the components. It will become evident after discussing the third part of the algorithm that

all of the invariants, such as a degree bound and absence of cycles, would hold for those components.

The third and final stage of the algorithm is reconnecting. As we recorded all of the previous connections between components in the first stage, now let's call the multiset of nodes they were connecting a set of *connectors*. Each instance of the connector in the multiset tells us that the corresponding node for that connector can be connected to some other node via an edge without exceeding its bounded degree. Incidentally, if the node appears in a list of connectors multiple times, it shows us that it is at least that much above the degree bound right now. Now we would look at all of the possible splits of the connector nodes into pairs which would be connected by edges in the resulting tree, select only valid splits and choose the best one. It can be seen that the validity of a split can be checked with time complexity of $\mathcal{O}(\alpha(N) \cdot N)$ as it follows from the Disjoint Set operations complexity [14]. The split is valid if and only if all of the components are connected by edges in that split.

Overall, as we have seen, the time complexity of this algorithm with regard to n only is $\mathcal{O}(n \log n)$. However, if we account for the selection of N it will become $\mathcal{O}(2^N \cdot N! \cdot n \log n)$. This obviously prompts us to use very small values of N in order to avoid imminent performance losses.

2.2.4. Branch Switching

This algorithm is mostly needed for the evolutionary approach that is described later. This is a very effective and cheap way to change the structure of the tree. We would examine two versions of this algorithm here: one that does not care for the costs of the resulting tree and one that does. Both of them would be quite simple and the differences would mostly be nominal.

The idea is very similar in shape and form to the one that inspires the switching algorithm — what if we just switch two branches of the tree and see what comes out of it? The first idea is just to switch them without actually checking the cost improvement, which can be very beneficial for the evolutionary algorithm, as it aggressively reshapes the network while being extremely fast, even though it may make the cost worse. The second idea is to switch the branches with much more care and precaution, by first checking whether this would improve the cost of the network and only then switching the branches.

When discussing the algorithm itself, we may naively think that it would be fine to just switch second vertices for two random edges in the network, but that

Listing 4 – Compactions Algorithm

```

def apply_compaction(
    cut_into: int
    , weights: List[List[int]]
    , tree: Graph
) -> Tuple[Graph, int]:
    # We save a queue of components that we want to split
    to_cut = deque([0])
    # And maintain a number of components
    components = 0
    # As well as the assignment of components to nodes
    colors = [0 for _ in range(tree.n)]

    # Repeat until we have the required number of components
    while components < cut_into:
        # Choose the component which we will split;
        component_to_cut = to_cut.popleft()
        # Find its centroid;
        centroid = find_centroid(tree, component_to_cut)
        # Color new components and add them to the queue.
        color_new_components(centroid, tree, to_cut)

    # Now we need to pair up all the connectors and figure out
    # which is best
    # Generate all possible splits;
    possible_splits = split_into_pairs(connectors)
    # And check whether they are ok.
    valid_splits = list(filter(check_split_correctness,
                               possible_splits))

    best_cost, best_split = float('inf'), None
    for split in valid_splits:
        cost = cost_of_split(split, tree)
        if cost < best_cost:
            best_cost, best_split = cost, split

    apply_split(best_split, tree)

    return tree, best_cost

```

may lead to the creation of a cycle. Therefore, we first need to check the correctness of the proposed switch.

The high-level description of the algorithm that does a branch swap without checking for cost improvement can be found in listing 5.

Listing 5 – Subtree Swap Algorithm

```
def swap_random_subtrees(
    max_sz: int
    , tree: Graph) -> Graph:
    # First we find a random root such that it is not a leaf
    root = random.randint(0, tree.n - 1)
    while len(tree.adj[root]) == 1:
        root = random.randint(0, tree.n - 1)

    # Then we find all the possible branch candidates,
    # by using the depth-first search and checking the sizes of
    # branches.
    candidates = find_all_candidates(tree)

    # Then, if there are at least two candidates, we switch them.
    if len(candidates) >= 2:
        [(s1, p1), (s2, p2)] = random.sample(candidates, 2)
        switch_subtrees(tree, s1, p1, s2, p2)

    return tree
```

2.3. Evolutionary Approach

The evolutionary approach is the pinnacle of what we have designed so far, as it is the combination of all the approaches above. The idea of the algorithm is the following: we would continuously evolve the population of several solutions using the enhancement algorithms eliminating some of them on each iteration.

First, we would create an initial population consisting of the applications of all the base algorithms to the demand profile that we are provided. Afterward, we would continuously evolve this population by applying certain enhancement algorithms to all members of the population and evaluating the costs of the solution, then keeping only some amount of the best-performing solutions and discarding the rest.

Let us define signatures for initial population producers and the mutators in order to later describe how the evolutionary algorithm itself would look like. They are provided in a listing 6.

Now we can explore the evolutionary algorithm itself in the listing 7.

This algorithm does a very simple thing — initially, it produces some initial results using the initial producers provided, and then for every iteration, or generation, it applies all of the available mutations to all of the solutions in the current population and then selects a number of best ones which do not exceed the prede-

Listing 6 – Evolutionary Algorithm Entity Signatures

```
Demand = List[List[int]]

AlgorithmResult = Tuple[Graph, int]

InitialProducer = Callable[[Demand], AlgorithmResult]

Mutator = Callable[[Demand, Graph, int], AlgorithmResult]
```

Listing 7 – Evolutionary Algorithm

```
def run_mutations(
    initial_producers: List[InitialProducer]
    , mutators: List[Mutator]
    , demand: List[List[int]]
    , max_iterations: Optional[int] = None
    , pop_size: Optional[int] = None
) -> Tuple[Graph, int]:

    population = [producer(demand) for producer in
        initial_producers]

    if pop_size is None:
        pop_size = len(mutators) * len(initial_producers)

    if max_iterations is None:
        max_iterations = -1

    iteration = 0
    while iteration != max_iterations:
        new_pop = [mutator(demand, tree, cost) for (tree, cost) in
            pop for mutator in mutators]
        asc_new_pop = sorted(new_pop, key=lambda x: x[1])

        if asc_new_pop[0][1] == pop[0][1]:
            break

        pop = asc_new_pop[:min(len(asc_new_pop), pop_size)]

        iteration += 1

    return sorted(pop, key=lambda x: x[1])[0]
```

terminated population size. After a set number of generations, or when the solution stops improving, the evolutionary algorithm stops.

There's no easy way to determine the time complexity of this algorithm, as it heavily depends on the set of provided initial providers and mutators, which may

drastically affect the performance. Specific sets of mutators would be investigated deeply in the next chapter along with other experiments.

Conclusions on Chapter 2

We have tediously described the benefits of the approach to the problem with the usage of base algorithms with enhancement algorithms later applied to the outputs of the latter. This design decision allowed us to make our solutions modular as well as made possible the later transition to the evolutionary algorithms.

We introduced a plethora of both base and enhancement algorithms with various strong and weak sides. Among the base algorithms, we described:

- Brute-force approach;
- Algorithm for finding the statically optimal network with a binary search tree topology;
- Algorithm for finding the statically optimal network with a binary search tree topology on multiple permutations;
- Generalized greedy algorithm;
- Algorithm for finding the maximum spanning tree on a demand graph.

Later we described a general idea behind the enhancement algorithms, and specifically investigated some of them:

- Switching algorithm;
- Branch smashing algorithm;
- Compacting algorithm;
- Branch switching algorithm;
- Evolutionary approach.

We would investigate all of the experiments and benchmarks and compare these algorithms' effectiveness for different simulated demand profiles as well as the real-world data in the next chapter.

CHAPTER 3. EXPERIMENTS

In this chapter, we would discuss the experiments proving the efficiency of the aforementioned approaches as well as compare them against each other. These experiments are necessary, as we are only investigating approximate solutions to the problem in question, and therefore need to prove that the proposed algorithms and heuristics work properly.

In order to showcase the results, we would mostly refer to several data sets that were used as test workloads and benchmarks in the previous works [7, 11, 15], as well as several synthetically simulated workloads. All of the workloads would be described in detail in section ??.

3.1. Real-World Workload Description

In this section we would some of the real-world provided workloads in detail, describing their defining characteristics and important patterns within the load.

3.1.1. Social Network Workload

The Social Network data center workload has $n = 100$ nodes and is very dense, which means that there are a lot of non-zero edges in the demand profile. A network with a dense demand profile is characterized by a consistently high level of network traffic and usage throughout the designated time period. Unlike networks with sporadic or fluctuating demand profiles, a dense demand profile reflects a continuous and sustained flow of data and communication activities across all of its nodes.

In a dense network, most nodes are directly connected to a large number of other vertices. This means that the density of edges is relatively high, resulting in a dense graph of connections within the network structure. Each vertex in a dense network tends to have a significant number of neighbors, forming a tightly interconnected network.

3.1.2. Microsoft Workload

As if the direct opposite of the previously described workload, the Microsoft workload has $n = 10000$ nodes and is very sparse, thus displaying a very interesting contrast of patterns with the latter.

In a sparse network, most vertices have a relatively low number of connections or neighbors, resulting in a sparser graph of connections within the graph structure. There are fewer edges present, and the network exhibits a more scattered or

dispersed connectivity pattern. However, the edges, or the connections, that are present in this network have a very high weight or cost in terms of our problem, thus creating an interesting and close to a real scenario where most of the traffic in the data center flows between some very loaded hosts.

3.1.3. pFabric Workload

This workload is one of the most interesting ones as it displays patterns of both of the network types explored above. A network that is partially sparse and partially dense can be seen as a combination of regions or subgraphs with varying degrees of edge density. In such a network, certain subsets of vertices or regions exhibit a sparse connectivity pattern, while other subsets or regions display a dense connectivity pattern.

This type of network can arise in various real-world scenarios where different entities or communities within the graph exhibit distinct levels of interaction or connectivity. In a partially sparse and partially dense graph, the density of edges and connectivity can vary significantly depending on the specific subset of vertices being considered. The sparse regions have a lower number of edges and exhibit a more dispersed or decentralized connectivity pattern, while the dense regions have a higher density of edges and show a more tightly interconnected network structure.

3.1.4. ProjecToR Workload

ProjecToR Workload is derived from a project ProjecToR [12] which explores reconfigurable computer networks, which is an area adjacent to the one we are investigating in this work. This demand profile is also highly varying as is described in the subsection 3.1.3.

3.2. Synthetic Workload Description

In this section, we would examine the workloads which were synthetically generated for this particular problem. Using synthetic workloads to test our solutions and algorithms for this approximation problem results provides several benefits:

- **Controlled Environment:** Synthetic workloads allow you to create controlled and reproducible scenarios for testing. You have full control over the characteristics of the workload, including its size, complexity, and specific properties. This enables us to systematically explore different aspects of the problem space, fine-tune parameters, and analyze the behavior of the approximation algorithms under various conditions.

- **Scalability:** Synthetic workloads can be generated to mimic real-world scenarios with varying data sizes, problem complexities, or network structures. This allows us to test the scalability of the approximation algorithms by increasing the workload size and evaluating their performance as the problem scales. It helps identify potential bottlenecks, efficiency issues, or limitations of the algorithms as the workload grows.
- **Comparative Analysis:** Synthetic workloads facilitate fair and unbiased comparisons between different approximation algorithms. By using the same synthetic workload to evaluate multiple algorithms, you can directly compare their performance, efficiency, and solution quality. This allows you to make informed decisions about which algorithm or approach is better suited for a particular problem.

Overall, synthetic workloads offer a controlled and flexible testing environment, providing a reliable basis for evaluating and comparing the performance of approximation algorithms. They enable researchers to gain insights into algorithm behavior, assess scalability, and refine strategies for improving the accuracy and efficiency of approximation solutions.

3.2.1. Dense Workload

The synthetically generated dense workload follows the same demand patterns as explored in a section 3.1.1. It is generated by uniformly generating each weight in the network, as can be seen on the listing 8.

Listing 8 – Generating Dense Workload

```
def gen_dense_workload(
    random_object: random.Random
    , n: int
    , max_weight: int
) -> List[List[int]]:
    return [[random_object.randint(0, max_weight) if i < j else 0
             for j in range(n)] for i in range(n)]
```

As it can be seen, each weight for the pair of vertices i and j is generated randomly and uniformly between 0 and the specified ahead-of-time max weight.

3.2.2. Sparse Workload

On the other hand, the synthetically generated sparse workload follows the same demand patterns as explored in section 3.1.2. It is generated by first determin-

ing whether there would be any demand between two hosts via a Bernoulli process, also known as an unfair coin toss, and then uniformly generating each weight in the network, as can be seen on the listing 9.

Listing 9 – Generating Sparse Workload

```
def gen_dense_workload(
    random_object: random.Random
    , n: int
    , max_weight: int
    , prob_of_demand: float
) -> List[List[int]]:
    dense = gen_dense_workload(random_object, n, max_weight)

    return [[element if random_object.random() < prob_of_demand
              else 0 for element in row] for row in dense]
```

The algorithm leverages the power of a previously created algorithm for generating dense workloads 8 and then removes some elements based on the probability of pair of nodes having a demand between them which is specified ahead of time.

3.3. Results

In this section, we will provide the results of the experiments in different formats, including, but not limited to, tables, graphs, and charts. It is important to dig deeper into the meaning of those tests by discussing the patterns that can be observed in the data that we investigate as it provides us with an understanding of the underlying demand profiles, as well as the algorithm details and specifications.

For the synthetic workloads, we would present data aggregated for 1000 runs on randomly generated data. Columns in the table represent the following values:

- Algorithm — the name of the algorithm used;
- Mean Result — mean cost of the results of this algorithm across all runs;
- Mean Deviation from Best — mean deviation from the best result across all runs;
- Mean Deviation from Best (%) — mean deviation from the best result across all runs expressed in percent;
- Minimum Deviation from Best (%) — minimum deviation from the best result across all runs expressed in percent;
- Maximum Deviation from Best (%) — maximum deviation from the best result across all runs expressed in percent.

3.3.1. Synthetic Dense Workload Results

One can find the results of the described algorithms for the synthetically generated dense graph with a maximal weight of 100, and a size of 1000 in a table 1, and results for the synthetically generated dense graph with a maximal weight of 10 and a size of 100 in a table 2.

3.3.2. Synthetic Sparse Workload Results

One can find the results of the described algorithms for the synthetically generated sparse graph with a probability of demand equal to 0.05, the maximal weight of 10, and size of 1000 in a table 3, and results for the synthetically generated sparse graph with a probability of demand equal to 0.1, the maximal weight of 10 and a size of 100 in a table 4.

It can be seen, that Maximum Spanning Tree based approaches perform substantially better on sparse workloads than Best Binary Search Tree approaches. It also showcases that sometimes the Smasher enhancement algorithm can be better than the Switcher enhancement algorithm.

3.3.3. Real-World Results

For the real-world data sets, we would be running each of the algorithms once and then displaying the results on a grid with time elapsed on the X axis and the cost of the solution generated by an algorithm on the Y axis. The shorthand names of the algorithms are displayed on top of the points on the grid, however, we would give an explanation of what they mean here:

- **BST** — Best Binary Search Tree algorithm;
- **MST** — Maximum Spanning Tree algorithm;
- **Switching over BST** — Best Binary Search Tree base algorithm enhanced with Switching algorithm;
- **Switching over MST** — Maximum Spanning Tree base algorithm enhanced with Switching algorithm;
- **Smashing over BST** — Best Binary Search Tree base algorithm enhanced with Smashing algorithm;
- **Smashing over MST** — Maximum Spanning Tree base algorithm enhanced with Smashing algorithm;
- **Switching & Smashing over BST** — Best Binary Search Tree base algorithm enhanced with both Switching and Smashing algorithms;

- **Switching & Smashing over MST** — Maximum Spanning Tree base algorithm enhanced with both Switching and Smashing algorithms;
- **Evolution** — evolutionary, or genetic, approach.

The result for the workload 3.1.1 can be seen on the graph 1, the result for the Microsoft workload 3.1.2 can be seen on the graph 2, the result for the pFabric workload can be seen on the graph 3.1.3, and the result for the ProjecToR workload 3.1.4 can be seen on the graph 4.

As an observant reader may notice that the best benchmark we can use for this case is multiple launches of the Best Binary Search Tree algorithm on different permutations, as it gives us a good perspective on where the truly optimal solution should be. However, running that algorithm for a reasonable amount of permutations takes too much time and even with a logarithmic scale, it does not fit on the graphs. As such, we have placed those results separately in a table 5. All of those experiments have been carried out for the same amount of random permutations — 10000. One can see that the result is always worse than the evolutionary approach.

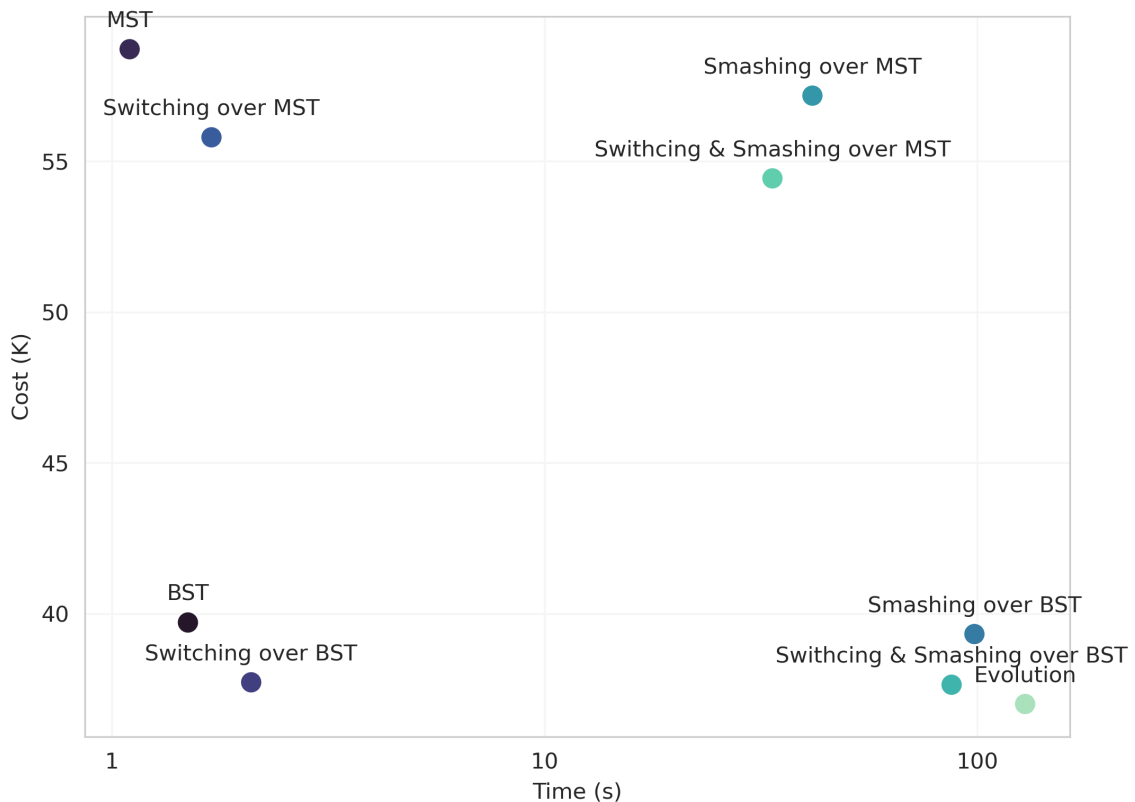


Figure 1 – Results for Social Network Workload

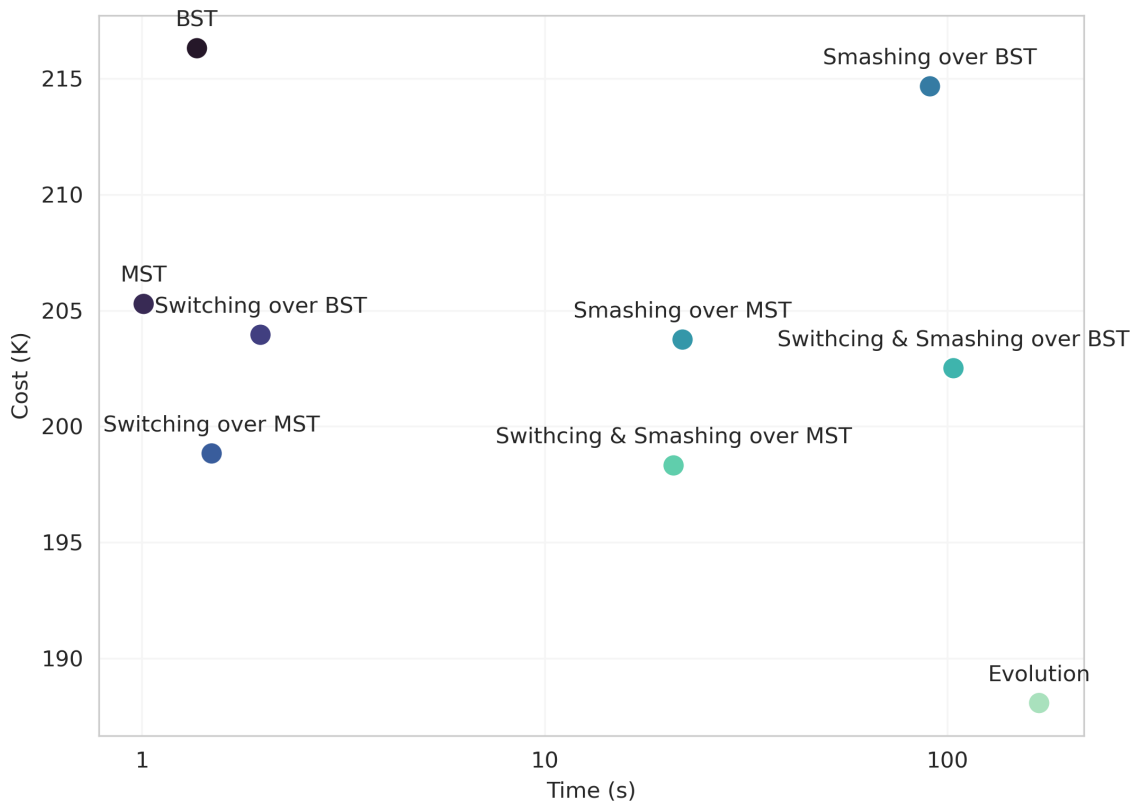


Figure 2 – Results for Microsoft Workload

Conclusions on Chapter 3

We have showcased both using the synthetically generated workloads and real-world benchmarks, that, first of all, the Maximum Spanning Tree base algorithm which we invented in this work works wondrously as a base algorithm on the sparse workloads, moreover, it works much better than the already discovered best binary search tree algorithm.

Also, we provided enough evidence to the fact that the proposed enhancement algorithms work much faster than the brute-force algorithms and showcased the fact that they are in fact superior in reliability, performance, and overall final score to those which existed before this work in the subject area.

Graphs provided by testing on the real-world data vividly display a variety of different algorithms that we developed throughout this work — there are ones, such as the Maximum Spanning Tree algorithm, that work very fast and provide a rough initial estimation, and there are ones, such as evolutionary approach, that work much

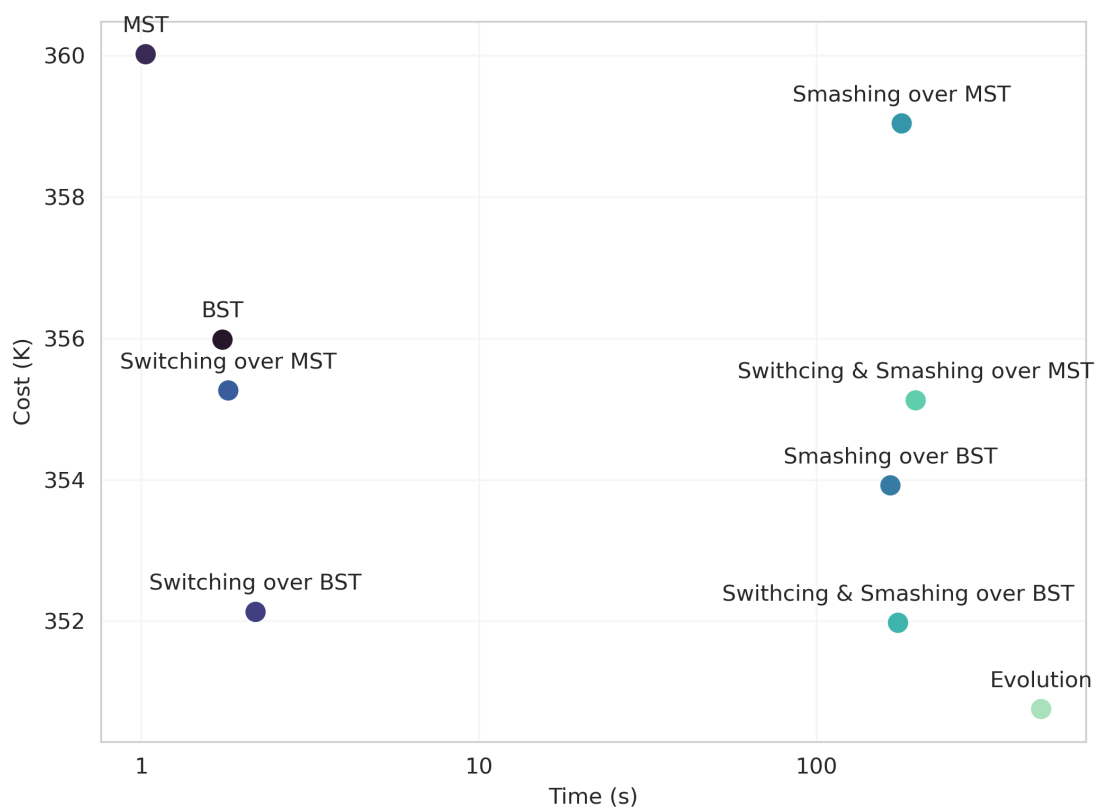


Figure 3 – Results for pFabric Workload

longer, but provide far better estimates than even pre-existing algorithms that work for marginally larger time frames.

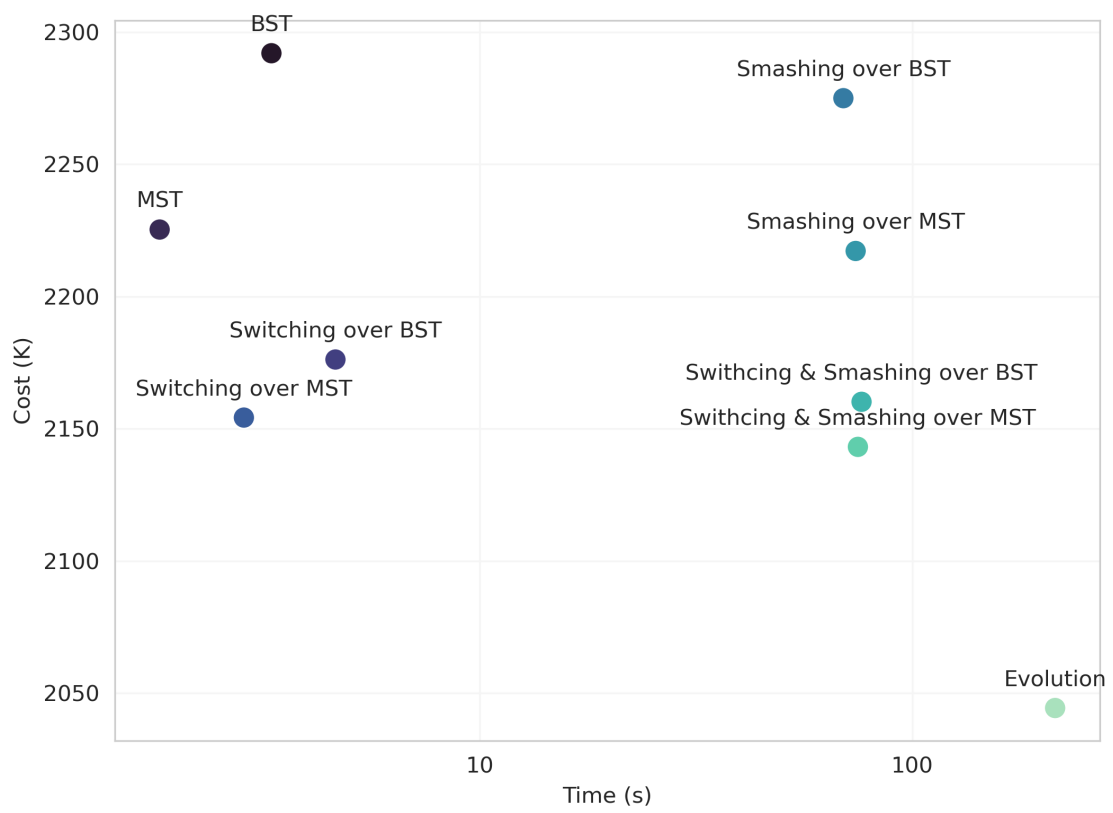


Figure 4 – Results for ProjecToR Workload

Table 1 – Synthetic Dense Workload Results For Big Graphs

Algorithm	Mean Result	Mean Deviation from Best	Mean Deviation from Best (%)	Minimum Deviation from Best (%)	Maximum Deviation from Best (%)
Best Binary Search Tree	1834538.40	29720.80	0.01%	0.02%	0.02%
Maximum Spanning Tree	2960948.90	1156131.30	0.50%	0.64%	0.82%
Best Binary Search Tree with Switcher	1811693.20	6875.60	0.00%	0.00%	0.01%
Maximum Spanning Tree with Switcher	2917672.40	1112854.80	0.48%	0.62%	0.79%
Best Binary Search Tree with Smasher	1834108.80	29291.20	0.01%	0.02%	0.02%
Maximum Spanning Tree with Smasher	2887767.70	1082950.10	0.45%	0.60%	0.79%
Multiple Launches of Best Binary Search Tree	1828223.90	23406.30	0.01%	0.01%	0.02%
Genetic Algorithm Approach Over Both Base Algorithms	1804817.60	0.00	0.00%	0.00%	0.00%

Table 2 – Synthetic Dense Workload Results For Small Graphs

Algorithm	Mean Result	Mean Deviation from Best	Mean Deviation from Best (%)	Minimum Deviation from Best (%)	Maximum Deviation from Best (%)
Best Binary Search Tree	183309.00	3466.90	0.02%	0.02%	0.02%
Maximum Spanning Tree	213318.50	33476.40	0.09%	0.19%	0.28%
Best Binary Search Tree with Switcher	180834.70	992.60	0.00%	0.01%	0.01%
Maximum Spanning Tree with Switcher	209770.90	29928.80	0.08%	0.17%	0.26%
Best Binary Search Tree with Smasher	183260.20	3418.10	0.01%	0.02%	0.02%
Maximum Spanning Tree with Smasher	211310.60	31468.50	0.08%	0.17%	0.26%
Multiple Launches of Best Binary Search Tree	182674.70	2832.60	0.01%	0.02%	0.02%
Genetic Algorithm Approach Over Both Base Algorithms	179842.10	0.00	0.00%	0.00%	0.00%

Table 3 – Synthetic Sparse Workload Results For Big Graphs

Algorithm	Mean Result	Mean Deviation from Best	Mean Deviation from Best (%)	Minimum Deviation from Best (%)	Maximum Deviation from Best (%)
Best Binary Search Tree	77675.10	28397.90	0.40%	0.58%	0.74%
Maximum Spanning Tree	63325.40	14048.20	0.17%	0.28%	0.48%
Best Binary Search Tree with Switcher	70422.70	21145.50	0.27%	0.44%	0.60%
Maximum Spanning Tree with Switcher	62952.70	13675.50	0.16%	0.27%	0.47%
Best Binary Search Tree with Smasher	77082.00	27804.80	0.38%	0.57%	0.73%
Maximum Spanning Tree with Smasher	62409.40	13132.20	0.16%	0.26%	0.45%
Multiple Launches of Best Binary Search Tree	75187.30	25910.10	0.37%	0.53%	0.71%
Genetic Algorithm Approach Over Both Base Algorithms	49277.20	0.00	0.00%	0.00%	0.00%

Table 4 – Synthetic Sparse Workload Results For Small Graphs

Algorithm	Mean Result	Mean Deviation from Best	Mean Deviation from Best (%)	Minimum Deviation from Best (%)	Maximum Deviation from Best (%)
Best Binary Search Tree	16869.20	2567.40	0.11%	0.18%	0.23%
Maximum Spanning Tree	20276.30	5974.50	0.25%	0.42%	0.56%
Best Binary Search Tree with Switcher	15655.10	1353.30	0.03%	0.10%	0.16%
Maximum Spanning Tree with Switcher	19915.50	5613.70	0.24%	0.39%	0.54%
Best Binary Search Tree with Smasher	16779.30	2477.50	0.11%	0.17%	0.23%
Maximum Spanning Tree with Smasher	19863.50	5561.70	0.22%	0.39%	0.51%
Multiple Launches of Best Binary Search Tree	16384.40	2082.60	0.09%	0.15%	0.22%
Genetic Algorithm Approach Over Both Base Algorithms	14417.00	0.00	0.00%	0.00%	0.00%

Table 5 – Multiple Permutations Over Best Binary Search Tree

Workload Name	Time Elapsed	Solution Cost	Evolutionary Algorithm Solution Cost on the Same Workload
Social Network	3h 34m	37891	37011
Microsoft	4h 11m	195388	188066
pFabric	4h 35m	354595	350758
ProjecToR	8h 43m	2097061	2044430

CONCLUSION

The research concluded with significant findings regarding the problem of constructing statically optimal computer networks with a binary tree topology. The use of base algorithms followed by enhancement algorithms presented a successful approach to the problem. The study introduced a variety of these algorithms, each having its unique strengths and weaknesses.

We have introduced the Maximum Spanning Tree algorithm, which demonstrated excellent performance, particularly with sparse workloads, outperforming even the best binary search tree algorithm that was previously known. Additionally, we've introduced a plethora of enhancement algorithms that were proven to be superior to brute-force algorithms in terms of reliability, performance, and overall quality of the approximation.

This layered approach of base and enhancement algorithms was not only efficient but also modular, facilitating the transition to evolutionary algorithms. Furthermore, an evolutionary approach, though time-consuming, provided superior estimates compared to existing algorithms operating within similar time frames. It vastly outperformed the brute-force solution previously known in both the performance and quality of the result.

The real-world data testing illustrated the versatility of the developed algorithms, contributing to a comprehensive solution for the problem of constructing statically optimal computer networks with a binary tree topology. Thus, the research effectively pushed the boundaries of current practices, introducing new, effective approaches to this problem.

REFERENCES

- 1 *Avin C., Mondal K., Schmid S.* Demand-Aware Network Designs of Bounded Degree. — 2017. — arXiv: 1705.06024 [cs.DC].
- 2 *Avin C., Schmid S.* Toward Demand-Aware Networking: A Theory for Self-Adjusting Networks. — 2018. — arXiv: 1807.02935 [cs.NI].
- 3 *Bader D. A., Cong G.* Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs // 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. — 2004. — P. 39–.
- 4 *Bentley J. L., Haken D., Saxe J. B.* A general method for solving divide-and-conquer recurrences // ACM SIGACT News. — 1980. — Vol. 12, no. 3. — P. 36–44.
- 5 *Chazelle B.* A minimum spanning tree algorithm with inverse-Ackermann type complexity // J. ACM. — 2000. — Vol. 47. — P. 1028–1047.
- 6 *Frederickson G. N., Johnson D. B.* Generating and searching sets induced by networks: Preliminary version // Automata, Languages and Programming: Seventh Colloquium Noordwijkerhout, the Netherlands July 14–18, 1980 7. — Springer. 1980. — P. 221–233.
- 7 Jupiter Evolving: Transforming Google’s Datacenter Network via Optical Circuit Switches and Software-Defined Networking / L. Poutievski [et al.] // Proceedings of ACM SIGCOMM 2022. — 2022.
- 8 *Karger D. R., Klein P. N., Tarjan R. E.* A randomized linear-time algorithm to find minimum spanning trees // J. ACM. — 1995. — Vol. 42. — P. 321–328.
- 9 *Koshy T., Salmassi M.* Parity and Primality of Catalan Numbers // The College Mathematics Journal. — 2006. — Vol. 37, no. 1. — P. 52–53. — ISSN 07468342, 19311346. — URL: <http://www.jstor.org/stable/27646275> (visited on 05/07/2023).
- 10 *Moura L. de, Bjørner N.* Z3: An Efficient SMT Solver // Tools and Algorithms for the Construction and Analysis of Systems / ed. by C. R. Ramakrishnan, J. Rehof. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. — P. 337–340. — ISBN 978-3-540-78800-3.

- 11 PFabric: Minimal near-Optimal Datacenter Transport / M. Alizadeh [et al.] // SIGCOMM Comput. Commun. Rev. — New York, NY, USA, 2013. — Aug. — Vol. 43, no. 4. — P. 435–446. — ISSN 0146-4833. — DOI: 10 . 1145/2534169 . 2486031. — URL: <https://doi.org/10.1145/2534169.2486031>.
- 12 ProjecToR: Agile Reconfigurable Data Center Interconnect / M. Ghobadi [et al.] // Proceedings of the 2016 ACM SIGCOMM Conference. — Florianopolis, Brazil : Association for Computing Machinery, 2016. — P. 216–229. — (SIGCOMM '16). — ISBN 9781450341936. — DOI: 10 . 1145 / 2934872 . 2934911. — URL: <https://doi.org/10.1145/2934872.2934911>.
- 13 SplayNet: Towards Locally Self-Adjusting Networks / S. Schmid [et al.] // IEEE/ACM Transactions on Networking. — 2016. — Vol. 24, no. 3. — P. 1421–1433. — DOI: 10 . 1109/TNET . 2015 . 2410313.
- 14 *Tarjan R. E., Leeuwen J. van.* Worst-case Analysis of Set Union Algorithms // J. ACM. — 1984. — Vol. 31. — P. 245–281.
- 15 Toward Self-Adjusting k-ary Search Tree Networks / E. Feder [et al.]. — 2023. — arXiv: 2302 . 13113 [cs.NI].
- 16 *Harary F.* Graph theory. — Addison-Wesley, MA : Princeton University Press, 1994. — 284 p.
- 17 *Iskandarov I. Z., Nurmetova B. B., Sobirov B. I.* DYNAMIC PROGRAMMING BY SUBSEGMENTS. — 2022. — URL: <https://cyberleninka.ru/article/n/dynamic-programming-by-subsegments>.