Federal State Autonomous Educational Institution for
Higher Education National Research University
Higher School of Economics

Faculty of Computer Science
BSc Applied Mathematics and Information Science

BACHELOR'S THESIS

Research project

Develop a Fast Approximate Shortest-Path Algorithm for Large-Scale Network

Submitted by Mikhail Anoprenko
student of group 185, 4th year of study,

Approved by Supervisor:
Candidate of Technical Sciences, Docent, Oleg Sukhoroslov

Curator:
Candidate of Physical and Mathematical Sciences,
Lead Engineer of Key Projects, Maxim Prikhodko

Moscow, 2022

Contents

1 Introduction **4**

1.1 Motivation 4

1.2 Mathematical model and problem statement . 5

2 Known solutions **7**

3 Algorithm evaluation and testing **9**

3.1 Graph generation 9

3.2 Edge weights generator 9

3.3 Topology generator 9

3.3.1 Square Grid 10

3.3.2 Multi-Homed network 10

3.3.3 Fat-Tree network 10

3.4 Algorithm validation 11

3.4.1 Model solution 12

4 Developed techniques **13**

4.1 Limited Dijkstra's algorithm 13

4.2 BFS-based DAG 13

4.3 Clustered Dijkstra's algorithm 14

4.3.1 Clustering 14

4.3.2 Inter-cluster routing 15

4.3.3 Intra-cluster routing 15

4.3.4 Forwarding table calculation 16

4.3.5 Correctness 16

5 Evaluation results **18**

5.1 Square Grid network 18

5.2 Dual-Homed Network 19

5.3 Fat-Tree network 19

5.4 Conclusion 20

Abstract

Modern distributed systems heavily rely on efficient and fault-tolerant networks. Routing algorithms are widely used in computer networks in order to provide consistent and stable packet delivery in the presence of hardware faults.

Most common link state routing algorithms use Dijkstra's algorithm to compute shortest paths and build forwarding tables. This approach provides efficient and reliable routing, but its performance struggles when applied to large scale networks.

This research aims to resolve the performance issue of Dijkstra's algorithm by slightly relaxing the shortest paths requirement. It provides several approaches which increase routing algorithm's performance while avoiding significant loss in the quality of obtained routes.

Keywords

computer networks, approximate shortest paths, routing, link state protocols

1 Introduction

1.1 Motivation

Routing is a problem that needs to be solved in any computer network. In the most general sense, solving a routing problem in a computer network means determining how to deliver network packets from one node to another.

In IP networks the routing problem is solved by using forwarding tables. Each node of the network stores its own table containing a set of forwarding rules. Forwarding rules describe for each destination where the packet should be forwarded in order to be eventually delivered to that destination.

Except for some small ad-hoc networks, in modern computer networks topology is constantly changing. This relates to both wireless networks, where nodes may move in space forcing links between nodes to appear and disappear, and wired networks, where links and routers may break due to physical damage or high network load. This means that the routing problem should be solved automatically, especially in large-scale networks, i.e. we need a protocol and an algorithm for each node to construct its forwarding tables. This algorithm should be evaluated whenever a change in network topology occurs in order for routes to be correct and efficient.

Routing protocols are generally divided into two classes: link state and distance vector protocols. In distance vector protocols routers exchange some common information about network topology and build routes using a dynamic programming approach. This type of protocol is usually used in communication between autonomous systems. In this project we are going to focus on the other protocol type.

Link state protocols are usually used within an autonomous system. As opposed to distance vector protocols, in link state protocols each node knows the whole topology of the network and independently calculates its forwarding table. This means that the only type of information that nodes have to exchange with each other is the information about current link states in the network.

Thus, when speaking about link state routing algorithms, the problem comes down to building forwarding tables in each node using the whole information about network topology.

The goal of this project is to improve known link-state routing algorithms in order to achieve better performance while we allow ourselves to build sub-optimal solutions. We will give a formal problem statement in section 1.2 and review known approaches in section 2. The rest of the work will contain the description of developed approaches and results obtained during the testing of those approaches.

The code of the testing framework and implementations of algorithms covered in this work are available at `https://github.com/manoprenko/ba-thesis-routing`

1.2 Mathematical model and problem statement

We represent network topology as a graph. In general case this graph may be directed because links may be asymmetric.

Edges in the graph are weighted. Edge weights are positive real numbers representing the cost of a packet forwarding through the corresponding link. In practice edge weights are usually calculated based on bandwidths and latencies of links. Edge weights computation is a separate problem that we are not going to address. It is being actively researched and has some known approaches.

We will denote this graph as $G = (V, E)$, where V is the set of nodes of the network and E is the set of directed weighted edges (i.e. links) between these nodes. Also denote $n = |V|$ as the number of nodes and $m = |E|$ as the number of edges. While in networks nodes are usually identified by their IP addresses, we will numerate all nodes from 1 to n and use these numbers to identify nodes.

Each node u knows the whole topology of the network (i.e. all edge weights) and evaluates the algorithm in order to compute a forwarding table T_u. For each potential destination node $t \neq u$ of the network forwarding table contains a next hop node $v = T_u(t)$, where incoming packet should be forwarded in order to reach node t as fast as possible. There must exist a direct link from u to v.

Algorithm must be distributed meaning that no communication between nodes is allowed, except link state messages. Each node should compute its forwarding table independently using only the graph topology.

The most important requirement to the algorithm is the loop-freedom in a distributed model. This means that for any pair of source and destination nodes a packet originated in the source node and consequently forwarded through the network according to forwarding tables in each node should reach the destination at some point. Speaking more formally, for any pair of nodes s and t where $s \neq t$ let's consider the following sequence of nodes:

- $p_0 = s$,
- if $p_i = t$, then p_i is the last element of the sequence,
- otherwise, $p_{i+1} = T_{p_i}(t)$.

Loop-freedom means that for any pair of distinct nodes this sequence must be finite. This property guarantees that any packet will be eventually delivered to its destination. Given that this property is satisfied we will call the sequence $p_0, p_1, \ldots, p_k$ a *forwarding path* from s to t.

Target function of the algorithm is related to lengths of paths. Length of the path is the sum of weights of all edges belonging to this path. Let $F(u, v)$ be the length of the forwarding path from s to t and $L(u, v)$ be the length of the shortest path from s to t. We will define *relative stretch* of the path from s to t $S(s, t)$ as follows:

$$S(u, v) = \frac{F(u, v) - L(u, v)}{L(u, v)}$$

In order to efficiently utilize the bandwidth of the network and to deliver packets rapidly the lengths of forwarding paths should be as small as possible. We will compute relative stretches of paths between all pairs of distinct nodes and aim to minimize various statistics of these stretches (e.g. average, median or maximum). We will refer to the *quality* of the algorithm as a measure of its relative stretches

The second parameter we are going to optimize is algorithm's performance. We will measure the run time of developed algorithms on various graphs and aim to minimize it in average and worst case. In this work we focus on practical application of developed algorithms, so the run time is our key metric, not the asymptotic complexity of those algorithms.

Detailed description of graphs used for testing purposes will be given in section 3

2 Known solutions

Modern link state routing protocols use Dijkstra's algorithm to find shortest paths and compute forwarding tables. Examples of widely used link state protocols are Open Shortest Path First (OSPF) [1] and Intermediate System to Intermediate System (IS-IS) [2]. These protocols differ in some implementation details regarding exchange of link state information, but their approaches to the calculation of forwarding tables are essentially the same.

The algorithm for forwarding tables calculation is the following: each node u runs Dijkstra's algorithm and computes all shortest paths from u to all other nodes. Then the forwarding table is computed as follows: $T_u(t)$ is next node after u on the shortest path from u to t.

Given that all nodes use the same algorithm described above, all packets are forwarded along the shortest possible path, i.e. for any pair of nodes s and t forwarding path from s to t coincides with one of the shortest paths from s to t. This implicates that given algorithm is loop-free and has zero relative stretch for any pair of nodes.

The main disadvantage of this algorithm is its performance. When graph scale grows up to tens of thousands of nodes, Dijkstra's algorithm tends to perform slowly and consume much CPU resources. This may affect the convergence and the speed of path rebuilding when network topology changes. Dijkstra's algorithm asymptotic complexity is $O(m \log n)$ where n is the number of nodes and m is the number of edges in the graph. Despite there being an implementation of the same algorithm with asymptotic complexity $O(n \log n + m)$, it is in fact unusable in practice. Practically efficient implementation of Dijkstra's algorithm requires using a data structure with logarithmic time complexity and doesn't scale very well on large networks.

Our goal is to speed up Dijkstra's algorithm in order to improve its running time. At the same time we are allowed to build sub-optimal forwarding paths in case they remain loop-free. We aim to minimize average relative stretch of paths and have the ability to exchange algorithm's performance for relative stretch.

An approximate shortest path search is a problem being widely researched. A few protocols have been developed in order to adopt approximate paths search [3-4]. There are known optimizations of Dijkstra's algorithm used to quickly find shortest paths between pair of vertices in a graph [5-6]. All these works are not applicable in the case of solving routing problem without the ability to change the existing protocol.

There is also a modification of OSPF protocol, called iSPF, which allows to incrementally recalculate shortest paths when link states in the network change [7].

In this work we are dealing with an isolated problem: no modifications in the topology could be made and no modifications of a protocol are allowed.

3 Algorithm evaluation and testing

A framework for testing purposes was developed during the research. This framework is implemented in C++ and supports graph generation, algorithm evaluation on various graph topologies, validating loop-freedom and calculating the stretch of obtained paths. Also it provides a set of usable abstractions to implement algorithms and manipulate graphs.

3.1 Graph generation

Testing framework provides a functionality to generate various graphs for testing purposes, trying to artificially recreate a real-life network topology. Graph generation consists of two parts: edge weights generation and topology generation. For both of these parts there are several pre-written generators, and any two of such generators may be combined in order to generate a final network.

3.2 Edge weights generator

Edge weights generator is basically a random number generator which is capable of generating real numbers corresponding to edge weights according to some distribution. In order for results to be reproducible, all random number generators are seeded with some default value.

The following edge weights distributions were used to evaluate algorithms:

- normal distribution with mean value 7.5 and variance 1.25,
- discrete distribution with the following probabilities: weight 100 with probability 0.8 and one of weights 200, 300, 500, 1000, 2000 with same probabilities equal to 0.04.

3.3 Topology generator

Topology generator is capable of producing graphs of some given topology at different configurable scales without edge weights. In our implementation topology generators produce unoriented graphs and then the framework converts them to oriented by replacing each edge with two opposite oriented edges. When weights are assigned to edges, it may be done both in a symmetric manner, when opposite edges are assigned same weights, and asymmetric, when weights are assigned independently.

The following set of graph topologies was used to evaluate algorithms.

3.3.1 Square Grid

Square Grid network with side S is a network with S^2 nodes where nodes are arranged into a square grid. Adjacent nodes in the grid are connected by edges. This topology is very dense: shortest paths between some pairs of vertices contain $O(S)$ edges, while in other topologies the number of edges on the shortest path between vertices is constant and doesn't asymptotically grow with the scale of the network. Grid topology is sometimes used in computational systems and due its density it might be similar to topologies occurring in some wireless networks, where nodes are located far away from each other and the number of links is relatively small.

Grid topologies with the value of S within the interval from 30 to 200 were used to measure performance of implemented algorithms.

3.3.2 Multi-Homed network

Multi-Homed network is a network topology where nodes are divided into core routers and host nodes, where the number of core routers is relatively small. Each core router is connected to each host node. Effectively a Multi-Homed network is a complete bipartite graph with core routers in one part and host nodes in the other. This network topology is the simplest example of fault-tolerant network. It may be used in some small or wireless ad-hoc networks which don't need a huge bandwidth, but need to be resistant to router's failures.

A special case of this topology, Dual-Homed network, was used for testing purposes. Dual-Homed network is a Multi-Homed network with two core routers. The number of host nodes in our evaluations varies from 1000 to 50 000.

3.3.3 Fat-Tree network

Fat-Tree network topology [8] is parameterized by a value of k — the number of ports in switches being used to construct the network. We will consider three-level Fat-Tree networks as they are widely used in data center networks.

The network consists of $\left(\frac{k}{2}\right)^2$ core routers, connected to k pods. Each core router is connected to one of aggregation routers of each pod. Each pod consists of $\frac{k}{2}$ aggregation routers and $\frac{k}{2}$ edge routers. Within a single pod each aggregation router in connected to each edge router. Each edge router is connected to $\frac{k}{2}$ servers.

A variations of this network topology are being used in modern data centers because this topology is highly fault-tolerant and link bandwidth is distributed correspondingly to the average load in each part of the network. Also this topology supports up to $\frac{k^3}{4}$ servers and scales very well.

Fat-Tree topologies with three levels and values of k within the interval from 10 to 40 were used during the evaluation.

3.4 Algorithm validation

Testing framework ensures loop-freedom for all implemented algorithms. In order to do so, for each node u the algorithm is evaluated and the forwarding table T_u is computed. After this is done, we have to check that for each pair of distinct nodes s and t a packet originated at s eventually reaches t.

We will do this separately for all possible values of t, and for each particular t we will simultaneously check that the required property is satisfied for all possible source nodes s. This is done as follows: let's build graph G', where nodes are the same as in original network and for each node $s \neq t$ there is exactly one edge starting at this node, and its end is in the next hop node towards the destination t. In other words, the edge set of G' is $\{s \mapsto T_s(t) \mid s \in V\}$.

Notice that computed forwarding tables are loop-free if and only if all such graphs G' do not have loops in them. Thus for each graph G' we should just check that it's acyclic using simple depth-first search algorithm. Each such graph contains n vertices and $n - 1$ edges, so the asymptotic complexity of the validation is $O(n)$ for each particular node t or $O(n^2)$ for the whole network. Despite the validation itself works relatively fast, it needs the algorithm to be launched n times and this part tends to be the slowest in the whole process.

The same DFS traverse may also compute the distance from each node to t in G'. These distances are lengths of forwarding paths from corresponding nodes to t. This information may be stored in a matrix in order to compute path stretches afterwards.

Also the running time of the algorithm is measured using a system clock. As the running time for one node is usually relatively small and scattered, we measure the total running time of the algorithm launched for all graph's nodes.

3.4.1 Model solution

In order to compare implementations of algorithms, the testing framework needs a model solution, which is basically an implementation of Dijkstra's algorithm. Our implementation of the model solution is written in C++ and uses `std::priority_queue` as a core data structure of the algorithm as it tends to be the best option among the C++ standard library by performance.

For each graph both the algorithm and the model solution are evaluated and then relative stretches are calculated using matrices of lengths of forwarding paths. A few metrics of relative stretches are being observed:

- average relative stretch,
- mean relative stretch,
- 90-th percentile of relative stretch.

For both algorithms being compared their total running time is calculated in order to be observed.

To sum up, algorithm evaluation run consists of the following parts:

- The algorithm itself,
- "Model" algorithm based on Dijkstra's algorithm used to compute path stretches,
- Graph topology generator
- Edge weights generator.

4 Developed techniques

A few techniques have been developed in order to obtain desired optimizations.

4.1 Limited Dijkstra's algorithm

This is a general optimization which may be applied to an arbitrary routing algorithm. Let's choose some length bound L. It is important that the value of L is equal on all the nodes of the network, so it should be either a predefined constant or a value that may be computed based only on the graph topology and edge weights.

In each node u, after the forwarding table is computed by another algorithm, let's launch Dijkstra's algorithm with the following condition: when the algorithm reaches a node with distance more than L from the source, it immediately stops. As a result we will visit all the nodes v where the distance from u to v is less than or equal to L, because Dijkstra's algorithm visits nodes in order of the ascendance of the distance from the source.

Then for each visited node v let's change the value of $T_u(v)$ to the next hop node on the computed shortest path from u to v.

This optimization basically means the following: when the distance from the current node to the destination becomes less than or equal to L, the packet starts being forwarded along the shortest possible path. This means that if initial forwarding tables do not have forwarding loops, then this optimization doesn't create one, so it is correct and may be applied on top of any algorithm with any value of L given that it is the same for all nodes.

By varying the value of L we may exchange algorithm's performance for the stretch of constructed paths: increase of L leads to slower algorithm performance due to more nodes traversed by Dijkstra's algorithm, but decreases stretches of paths as they generally become shorter and never increase. Choosing the optimal value of L may be considered as a separate task based on required constraints. We will address this issue later.

4.2 BFS-based DAG

This approach is based on reducing the number of edges taken into consideration while searching for paths.

Namely, in the node u we do the following: launch the breadth-first search algorithm rooted at node u. In linear time it allows us to calculate for each node v the following value: $d_1(u, v)$ — minimal number of edges belonging to some path from u to v, i.e. the shortest path from u to v if we replace all edge weights with ones.

After doing this, let's consider the following subset of edges: for each edge starting in node x and ending in node y we will take this edge if and only if $d_1(u, y) = d_1(u, x) + 1$. This means that if we order all nodes by the distance d_1 from the source node u in ascending order, we will consider only edges going "from left to right". Notice that the obtained spanning graph is acyclic, i.e. it is a DAG. We will find shortest paths from u to all other nodes in this DAG. This can be done in linear time using a dynamic programming approach.

This algorithm is loop-free due to the following reason: when the packet is being forwarded towards destination t from node u to node v, the following inequality holds: $d_1(u, t) > d_1(v, t)$. This means that with each hop the distance d_1 to the destination vertex decreases. Given that it cannot be negative, this implicates that each packet will eventually reach its destination.

This algorithm is much faster than Dijkstra's algorithm and scales better due to better asymptotic complexity: $O(n+m)$. It performs very well when the variance of edge weights is relatively small. Specifically when all edge weights are the same, it finds shortest paths. When the variance of edge weights grows and their distribution becomes less uniform, this approach might experience huge growth of path stretch.

4.3 Clustered Dijkstra's algorithm

The main idea of this approach is the following: let's divide all nodes of the graph into some clusters. Then we split the routing problem into two parts: intra-cluster routing (i.e. routing witin a single cluster) and inter-cluster routing (i.e. routing between clusters). Each of this parts is solved by using the Dijkstra's algorithm or the BFS-based DAG approach mentioned earlier.

4.3.1 Clustering

The clustering must be consistent and the same on each node of the network in order to achieve loop-freedom. Distances between nodes within one cluster should be relatively small. In order to achieve this property, the following clustering algorithm is being used.

Choose an integer value of C — the limit of cluster size. Sort all edges in ascending order by their weights and consider them in this order. Initially each node belongs to its own cluster. When we consider an edge from node

u to node v, we will look at sizes of clusters these nodes belong to. In case they belong to different clusters and the sum of sizes of these clusters is less than or equal to C, then we unite these clusters into one. In order to efficiently perform this operation we are using Disjoint Set Union data structure to store clusters and unite them. This clustering algorithm is similar to Kruskal's algorithm for finding a minimum spanning forest and may be performed in time complexity $O(m \log m)$ caused by the need to sort all edges.

In general case the clustering may be arbitrary. In case it is consistent among all nodes, i.e. independent of the source node and calculated in a deterministic way, the routing algorithm will produce loop-free forwarding tables. A clustering algorithm may affect the performance and the quality of the algorithm, and thus it is a subject to further research and optimizations.

For convenience we will number all the clusters sequentially and denote the number of the cluster which the node i belongs to as c_i.

4.3.2 Inter-cluster routing

After the clustering is calculated the algorithm builds an inter-cluster forwarding table. First of all we reduce the network graph G by uniting all nodes of each cluster into one aggregated node. Thus we obtain a reduced graph G' where nodes are clusters of the initial network and edges are edges of the initial graph correspondingly translated onto new nodes.

After reducing the graph the algorithm calculates a forwarding table for the graph G' and root node c_u where u is the initial root node. This may be done by either a Dijkstra's algorithm or BFS-based DAG approach. In fact, any routing algorithm is suitable here, even this algorithm's recursive version. The inter-cluster forwarding table is a little bit different from a regular one. For each destination cluster it stores not only the number of the next hop cluster on the way towards destination, but also the edge of the initial graph corresponding to the inter-cluster edge we should traverse in order to reach the destination. Later this information will help us to construct the resultant forwarding table. We denote the obtained inter-cluster forwarding table for the cluster of node u as TG_{c_u}.

4.3.3 Intra-cluster routing

When launched in the source node u, the algorithm should calculate a regular forwarding table within the cluster c_u instead of the whole graph. As well as for the inter-cluster routing, this forwarding table may be calculated by any of approaches mentioned in this work. We denote the obtained intra-cluster forwarding table as TL_u. Note that $TL_u(t)$ is undefined if $c_u \neq c_t$.

4.3.4 Forwarding table calculation

In order to calculate the resultant forwarding table for the node u we have to combine the information from inter-cluster forwarding table TG_u and intra-cluster forwarding table TL_u.

Let's consider all possible destination nodes t and for each of them calculate the next hop node towards t. There might be two cases.

1. If $c_u = c_t$, then we don't have to perform an inter-cluster routing anymore and should simply deliver the packet to the destination within the current cluster. In order to do this we have to forward the packet to the node $TL_u(t)$. Thus, in this case $T_u(t) = TL_u(t)$.

2. If $c_u \neq c_t$, then we should first deliver the packet to the correct cluster c_t. In order to do this, let's consider the edge $TG_{c_u}(c_t)$. Denote by x the beginning node of this edge and by y the ending node of this edge. Note that x and y are nodes in the initial graph. In order to deliver the packet to the destination, we should deliver it to the node x and forward it along the considered edge.

   If $x = u$, then we may immediately forward the packet along this edge. In this case $T_u(t) = y$.

   Otherwise, we should forward the packet toward node x. As node x belongs to the same cluster as node u, i.e. $c_x = c_u$, we may refer to the value $TL_u(x)$ in the intra-cluster forwarding table in order to do this. Thus we obtain $T_u(t) = TL_u(x)$.

4.3.5 Correctness

Let's prove that this algorithm is loop-free.

First of all let's note that inter-cluster forwarding tables TG are consistent and loop-free. This is true due to the correctness of the routing algorithm used to calculate these tables. This means that for any source node s and destination node t the sequence of clusters on the forwarding path from s to t is consistent along all this path and doesn't contain loops. This implies that the sequence of inter-cluster edges on the forwarding path from s to t is also uniquely defined and consistent.

Intra-cluster forwarding tables are also consistent due to the correctness of used routing algorithm. This implies that forwarding paths within a single cluster are also uniquely defined and consistent. Thus, all parts of packet forwarding path are uniquely defined by the destination node and do not change while the packet is being forwarded along the path. This means that the algorithm is loop-free.

This algorithm consists of three parts: clustering, inter-cluster routing and intra-cluster routing. Each of these parts may be implemented and optimized independently of others. If the clustering is efficient enough than the number of nodes and edges in the reduces graph might be significantly less than in the initial graph. Given that clusters are small enough at the same time, the total running time of a routing algorithm for intra-cluster and inter-cluster routing may be less than the running time of the same algorithm on the whole network.

At the same time, clustering reduces quality of constructed routes as some potential forwarding paths are excluded from consideration.

5 Evaluation results

This section contains results of implemented algorithms' evaluation on a set of graphs generated by the framework described in section 3.

Here we denote by T the total running time of the algorithm for all nodes, measured in seconds. By S_{avg} we denote the average relative path stretch and by S_{mean} we denote the mean relative path stretch.

For convenience, algorithms in the tables will be referred to as follows:

- Model — model solution, Dijkstra's algorithm with `std::priority_queue`
- BFS-DAG — BFS-based DAG approach from section 4.2
- BFS-DAG + LD — Limited Dijkstra's algorithm heuristic from section 4.1 on top of BFS-DAG algorithm. The distance limit will be stated for each table separately as it depends on edge weights and we have no way of calculating it automatically

5.1 Square Grid network

We denote by S the side size of the grid.

Normal edge weights distribution. Distance limit for Dijkstra's algorithm is set to 100.

	Model			BFS-DAG			BFS-DAG + LD		
S	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}
30	0.05	0.00	0.00	0.04	0.00	0.00	0.06	0.00	0.00
80	3.23	0	0	1.73	0.01	0.00	2.07	0.01	0.00
120	17.50	0	0	9.16	0.02	0.00	9.77	0.02	0.00

Discrete edge weights distribution. Distance limit for Dijkstra's algorithm is set to 500.

	Model			BFS-DAG			BFS-DAG + LD		
S	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}
30	0.07	0.00	0.00	0.06	2.87	0.00	0.07	2.75	0.00
80	4.18	0.00	0.00	1.73	2.76	0.00	2.16	1.94	0.00
120	21.74	0.00	0.00	8.79	3.03	0.00	10.12	2.48	0.00

5.2 Dual-Homed Network

We denote by n the number of host nodes in the network.

Normal edge weights distribution. Distance limit for Dijkstra's algorithm is set to 10.

	Model			BFS-DAG			BFS-DAG + LD		
n	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}
3000	0.93	0.00	0.00	0.15	0.00	0.00	0.34	0.00	0.00
8000	7.53	0.00	0.00	1.13	0.00	0.00	2.63	0.00	0.00
15000	28.02	0.00	0.00	4.13	0.00	0.00	9.76	0.00	0.00

Discrete edge weights distribution. Distance limit for Dijkstra's algorithm is set to 200.

	Model			BFS-DAG			BFS-DAG + LD		
n	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}
3000	0.84	0.00	0.00	0.15	1.33	0.00	0.83	0.16	0.00
8000	5.69	0.00	0.00	1.03	1.37	0.00	1.69	1.37	0.00
15000	20.62	0.00	0.00	3.75	1.46	0.00	6.46	1.46	0.00

5.3 Fat-Tree network

We denote by k the scale parameter of a Fat-Tree network — the number of ports in a switch.

Normal edge weights distribution. Distance limit for Dijkstra's algorithm is set to 50.

	Model			BFS-DAG			BFS-DAG + LD		
k	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}
20	0.54	0.00	0.00	0.15	0.00	0.00	0.21	0.00	0.00
28	3.93	0.00	0.00	0.85	0.00	0.00	1.22	0.00	0.00
36	17.88	0.00	0.00	3.92	0.00	0.00	5.11	0.00	0.00

Discrete edge weights distribution. Distance limit for Dijkstra's algorithm is set to 500.

	Model			BFS-DAG			BFS-DAG + LD		
k	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}	T	S_{avg}	S_{mean}
20	0.54	0.00	0.00	0.14	3.05	0.00	0.42	0.39	0.00
28	3.94	0.00	0.00	0.80	2.22	0.00	2.51	0.29	0.00
36	17.74	0.00	0.00	3.53	1.87	0.00	10.02	0.23	0.00

5.4 Conclusion

Evaluation results demonstrate that BFS-based DAG approach tends to be very effective on considered network topologies. It consistently outperforms Dijkstra's algorithm up to 7 times and its average relative path stretch among all performed tests doesn't exceed 4%. Thus, this algorithm might be a good option for replacement of Dijkstra's algorithm in case of limited computing resources or large scale of the network.

Limited Dijkstra's algorithm optimization allows us to perform a more precise trade-off between performance and quality. In some cases it significantly decreases average path stretch while still outperforming Dijkstra's algorithm by almost 50%.

Clustered Dijkstra's algorithm is intentionally missing in evaluation results. Unfortunately, in our tests it showed poor performance and quality, losing to a simple Dijkstra's algorithm by running time and calculating forwarding tables with huge path stretches. Despite this fact, the algorithm seems to be very promising and it is worth mentioning in this work. All its parts, including clustering algorithm, inter-cluster and intra-cluster routing may be a subject for further research and improvement.

References

1. OSPF Version 2 RFC 2328, John Moy, 1998.
https://datatracker.ietf.org/doc/html/rfc2328

2. OSI IS-IS Intra-domain Routing Protocol, RFC 1142, 1990.
https://datatracker.ietf.org/doc/html/rfc1142

3. Jadoon, R.N.; Zhou, W.; Jadoon, W.; Ahmed Khan, I. RARZ: Ring-Zone Based Routing Protocol for Wireless Sensor Networks. Appl. Sci. 2018, 8, 1023.
https://doi.org/10.3390/app8071023

4. Slivkins, A.: Distance Estimation and Object Location via Rings of Neighbors, 2005.
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/podc05_full.pdf

5. Agarwal R.; Caesar M.; Godfrey P.B.; Zhao B.Y.: Shortest Paths in Microseconds, 2013.
https://arxiv.org/pdf/1309.0874.pdf

6. Mensah, D.N.A.; Gao, H.; Yang, L.W. Approximation Algorithm for Shortest Path in Large Social Networks. Algorithms 2020, 13, 36.
https://doi.org/10.3390/a13020036

7. El-Sayed, Hesham & Ahmed, Maaiz & Jaseemuddin, Muhammad & Petriu, Dorina. (2005). A Framework for Performance Characterization and Enhancement of the OSPF Routing Protocol.. 137-142.

8. Wang, Ting & Su, Zhiyang & Xia, Yu & Hamdi, Mounir. (2014). Rethinking the Data Center Networking: Architecture, Network Protocols, and Resource Sharing. IEEE Access. 2. 1481-1496. 10.1109/ACCESS.2014.2383439.