Summary

For our Master thesis we decided to explore the domain of realistic dynamic network simulation. This is done with foundation in our work with Software Defined Networks(SDNs) on the 9$^{\text{th}}$ semester pre-specialization. In particular we continue to explore the realms of network simulation, realistic traffic data implementation, data plane recomputation and failure protection. Through this project we define and implement a complete packet-level simulation framework. This framework is the product of our three contributions from the project. The first being the realistic simulation and extension of the discrete event simulator OMNeT++. On the basis of our pre-specialization project we wanted to achieve a more realistic network simulation than the one we had previously used. Therefore we looked to the simulation tool OMNeT++ as a possible choice. However we found that OMNeT++ was lacking in some aspects, and we therefore saw the need to extend the tool, such that it would better suit our needs for realistic simulation. The second contribution incorporated in the framework is our genetic algorithm Spungeet. We have previously worked with genetic algorithms and therefore saw the potential that they have as a solution to generating data planes. We therefore decided to implement the algorithm Spungeet to swiftly react to changes in the dynamic network. Spungeet works by first generating a random population of individuals which in this case are represented by demand weights, it then evaluates this population before it goes on to create offspring by means of crossover and mutation of these individuals, when the update time runs out, Spungeet stores the elite individuals of the population for the next udate round and then generates and installs the data plane based on the best individual. The third contribution of our project, which is used for the network part of the framework, is the analysis and generation of real world traffic data. Since we wanted to achieve realistic simulation of a network, we got access to traffic data from a large European ISP. We then analyzed this real world traffic data and used it to generate traffic data for networks in the Internet Topology Zoo. This was based on a well studied machine learning method, Gaussian Mixture models. The model we trained allowed us to probabilistically sample realistic network patterns. Thus giving us a more accurate representation of real world networks as opposed to the static demands that the Internet Topology Zoo normally uses. The project concludes with experiments comparing our proposed solution, Spungeet, with the state of the All Shortest Paths algorithm and the genetic algorithm GAOSPF. The experiments show that Spungeet consistenly outperforms the other two solutions. Thus indicating the novelty and potential of using a genetic algorithm for swift dynamic updates of network data planes.

Modelling Dynamic Networks in OMNeT++ with a Traffic Engineering Genetic Algorithm

Andreas Leicht Madsen, Lasse B. Kær, and Rasmus-Emil N. Herum
{amad18, lkar18, rherum18}@student.aau.dk

Department of Computer Science, Aalborg University, Aalborg, Denmark

Abstract. We explore the domain of realistic simulation of dynamically changing networks. Throughout this exploration we define and implement a complete packet-level simulation framework and quickly manage changes in a dynamic network. This is accomplished by our utilization of the discrete event simulator OMNeT++ and our own non-trivial extensions of the simulation tool. Furthermore we also analyze and transform real world traffic data from a large European ISP, leveraging the realism of the traffic data to improve the otherwise static traffic demands of the Internet Topology Zoo. Finally we contribute with our own genetic algorithm Spungeet, as a means of swiftly reacting to the constant changes of a parallel running network. Spungeet leverages the fact that the order in which demands are routed makes a large difference in the ability to deliver the packets and thus it is evident that mapping the demands to individuals will enable quick generation of well performing data planes for the network. The Spungeet algorithm tries to maximize the percentage of packet delivery and performs well in both scenarios with and without failures.

1 Introduction

Modern communication networks continue to increase in size along with the demands and expectations for the quality of service of these networks. Software Defined Networks(SDNs)[30] provide network operators with a separation of the control plane and the data plane allowing for scheduling data plane updates through the software level control plane. Within SDN traffic engineering is the discipline that deals with the performance optimization and evaluation of such networks [7]. A problem in the field of traffic engineering is determining the best routing for the traffic, i.e. to optimize the performance of the network in regards to throughput, availability, stretch, path etc [4][17][19]. Within this problem there are two different challenges that we wish to overcome. Firstly is the fact that demands are constantly changing as is seen in the real world ISP data. This is a challenge since it is very uncertain how much the demand changes and whether or not it increases or decreases. Secondly is the fact that links can suddenly fail [23]. There are many reasons for why link failures might happen e.g. power outage or equipment failure [18] etc. Thus resulting in impacting traffic in the network. For this project we want to explore a possible solution to these two challenges. In this regard we first explore a way to get data for a dynamic network model. Based on traffic data from a Large European ISP containing demands over a week, we create dynamic traffic based on the real world traffic for use with the Internet Topology Zoo [22]. This is done since the Internet Topology Zoo demands are created by the simulation tool REPETITA [10] and only contains static traffic data. Therefore the data synthesis creates many varying scenarios for our testing. Secondly we want to explore the area of realistic dynamic network simulation. For this endeavor we work with the discrete event simulator OMNeT++ [25]. We however found that OMNeT++ is lacking in some features for our particular use case. Thus, we choose to extend OMNeT++ with the necessary functionality for our testing environment. With the necessary foundation established we propose the genetic algorithm Spungeet, a reactive approach to maximizing the packet delivery percentage both under heavy load and network failures. The idea behind Spungeet is based on the belief that forwarding traffic on links with a lot of excess capacity is preferred as opposed to using links that are close to congestion. Due to the evolutionary nature of genetic algorithms they are particularly well suited for finding the most fit solution from a wide range of solutions. Therefore these types of algorithms map very well to the problem of deciding which route each demand should utilize.

The paper is organized into the following sections. Section 2 details the different related approaches to the problem domain along with information on related tools and techniques. Section 3 presents a range of definitions that describe our network model. Section 4 Details the analysis and transformation of traffic data from the Large European ISP. Section 5 details how we model the dynamic network along with providing the problem statement that we wish to answer, additionally we explain how we use the dynamic network model in a system which allows for monitoring and reacting to changes in the dynamic network. Section 6 Details the simulation granularity of the project along with how we use and extend the OMNeT++ [25] simulation tool. Section 7 explains in detail how the genetic algorithm Spungeet works. Section 8 Shows our results from several different experiments comparing Spungeet, GAOSPF [8] and All Shortest Paths(From here on named ASP). Section 9 Concludes on the project and our contributions. Section 10 describes any future work that we would like to complete if we were to work more on the project.

The sections which contain our contributions are as follows:

1. The extensions made to OMNeT++ is in Section 6
2. The synthesis of real world traffic data in Section 8
3. The Spungeet algorithm compared with the state-of-the-art algorithm ASP and the genetic algorithm GAOSPF in Section 8

2 Related Work

In this section we detail some of the work related to the field of traffic engineering. We will present each paper and how it relates to this paper.

2.1 Related tools and techniques

MPLS-kit Vanerio et al. [29] implemented the simulation tool MPLS-kit. The tool provides quick generation of MPLS data planes for simulation. It allows for simulation of the data planes at a flow level. Although the quick generation of data planes is convenient, a problem which we discuss further in Subsection 6.1, is that flow level simulations are inaccurate when network links are being congested, and because they do not account for the extra size data from packet headers, which also contributes to congetion. As congestion is an important aspect in the area of traffic engineering we will not be using MPLS-kit for the experiments in this paper.
Alternative approaches to traffic modelling Recently proposed solutions for traffic prediction make use of recurrent neural networks [28][27]. The model takes as input a time series of traffic, which it then uses for predicting the future traffic. Although this is generally effective, limitations include the fact that predicted data will be similar to the training data and difficulties such as vanishing gradients which is when the gradient in gradient descent becomes too small for the network to learn efficiently. We do not try to predict future traffic in this paper, but instead just focus on generating the best possible data plane based on the current traffic.

2.2 Related solutions

DOTE[27] Is another novel solution to updating the data plane, making use of neural networks to predict future traffic and generate optimal flows based on the prediction. The solution presented makes use of the fact that the objective function of linear programs are convex/concave. This attribute is highly desirable as it makes way for using optimization methods such as stochastic gradient descent. The paper also discusses the use of different objective functions for the network traffic engineering such as maximum link utilization and maximum network throughput. Finally the proposal is realized through a five layer neural network that takes as input the last 12 traffic matrices from the network to predict the next traffic matrix. For the output it uses the predicted matrix to set the weights of flows across tunnels in the network essentially solving the multi commodity splittable flow problem. In contrast, our approach seeks to react swiftly to traffic changes and as such Spungeet has the advantage of flexibility of a reactive system as well as increased adaptability in uncertain traffic.

GAOSPF[8] Another paper which is relevant to this project is the Genetic Algorithm Open Shortest Path First (GAOSPF). This paper proposes a genetic algorithm solution to the weight setting problem for Open Shortest Path First.

The OSPF weight setting problem looks to send traffic through the network by setting weights on each link and then routing each demand through all shortest paths from source to destination. The path length is determined by the sum of edge weights in the path. Compared to GAOSPF, Spungeet extends the idea of genetic algorithms, by running in parallel with the network. Additionally it retains information between runs and rapidly adapts to the current network environment. Furthermore Spungeet outpeforms GAOSPF in all of our experiments.

All shortest paths(ASP) A state-of-the-art approach widely used by ISPs is utilizing all shortest paths to route packets. The approach works by generating a Directed Acyclic Graph(DAG) of all the shortest paths from source to target. If the route through the DAG diverges at any point, the flow is split proportional to the capacities of the edges. Advantages of this approach is minimal stretch but at the cost of being oblivious to network load. Our approach with Spungeet differs greatly by constantly adapting to the networks current status.

Essence In a previous paper we proposed the algorithm Essence[13] as a solution to creating a congestion aware data plane for a network. The solution produces a resilient but also memory and congestion aware data plane. The general idea behind the algorithm is to generate a large amount of diverse paths and then find a strong configuration in terms of stretch and link utilization. In short Essence works by selecting a single primary path for each demand. Any non-primary paths are used as backup paths, and are ordered according to their longest common prefix with the primary path. These paths are then given to the forward backward routing (FBR) algorithm[20], which generates the data plane from the paths. Comparatively, Spungeet takes the other approach of recomputing the data plane using global information. Furthermore it is also known that in the worst case, deterministic fast failover algorithms are bound to create high load, because of the limited information available[1].

3 Network Model

In this section we present our formal definitions representing the network, routing and traffic.

Definition 1 (Network). A directed graph $G = (V, E, c)$ where V represents the routers, E represents the links $E \subseteq V \times V$ and $c : E \to \mathbb{N}_{\geq 0}$ is the bandwidth of each link.

We use the typical network definition as a graph with routers modeled as vertices and links as edges. We elect to use a directed graph as traffic is directed on a link and effects the utilization of a link only in one direction. In practice, all the networks in our experiments will have links in both directions.

We now define the paths that data is forwarded on as a tunnel.

Definition 2 (Tunnel). For a network $G = (V, E, c)$ We define a tunnel as a sequence of connected edges $f = (v_0, v_1)(v_1, v_2)...(v_{n-1}, v_n)$ where every edge in f is in E and f is loop free, i.e. every edge is in f at most once. We use F to denote the set of all possible tunnels.

A tunnel $f = (v_0, v_1)(v_1, v_2)...(v_{n-1}, v_n)$ is a path in the network that can be used to forward data from v_0 to v_n. We say that f routes from v_0 to v_n. Note from the definition that a tunnel is finite in length.

We now introduce how we model network traffic using demands.

Definition 3 (Demand). Given a network $G = (V, E, c)$, a demand is a triple $(v_{in}, v_{out}, l) \in V \times V \times \mathbb{N}_{>0}$ where v_{in} is the ingress router, v_{out} is the egress router, l is a positive amount of data that needs to be delivered from v_{in} to v_{out}. We use $D \subset V \times V \times \mathbb{N}_{>0}$ to denote the set of all demands in the network.

Based on this definition we can now define how the traffic of a demand is routed across the network using the classification function.

Definition 4 (Classification Function). A classification function is any function $\gamma : D \to F$ that takes as input a demand $d = (v_{in}, v_{out}, l)$ and returns a tunnel f that routes from v_{in} to v_{out}.

The classification function returns a tunnel for each demand in the network, which is the route that the traffic of the demand is routed on. We use the term *data plane* to denote the set of all tunnels in the network, i.e. the set $\{\gamma(d) | d \in D\}$

Figure 1 depicts a small network with demands and routing. The edges have arrows in both directions, depicting that there is a directed link going in both directions. So for example there is a link (v_1, v_2) with capacity $c((v_1, v_2)) = 80$ and a link (v_2, v_1) with capacity $c((v_2, v_1)) = 80$ as well. There are two demands, $d_1 = (v_3, v_4, 40)$ and $d_2 = (v_1, v_4, 80)$ being routed on tunnels $\gamma(d_1) = (v_3, v_4)(v_4, v_2)(v_2, v_5)$ and $\gamma(d_2) = (v_1, v_2)(v_2, v_4)(v_4, v_5)$.

Given a set of demands D and classification function γ, we now define the link load $l(e)$ as:

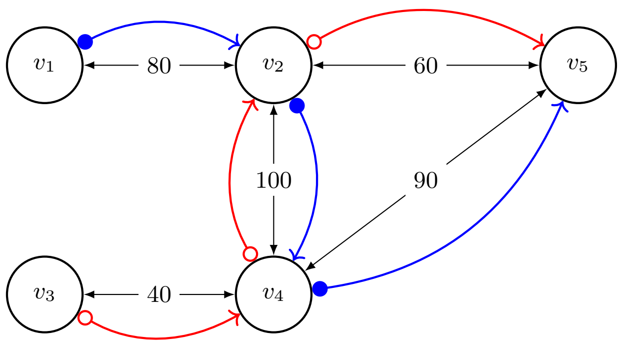
$$l(e) = \sum_{d'=(v,v',l) \in \{d | e \in \gamma(d)\}} l$$

In other words, the link load is the sum of all the loads we are trying to route on that link.

From the definition of link load we can define the link utilization as:

$$u(e) = \frac{l(e)}{c(e)}$$

The link utilization is a measure of how much load we are attempting to route on a link relative to its capacity. If $u(e) > 1$ we say that e is *congested.*



	demand	*classification*
○	$d_1 = (v_3, v_5, 40)$	$\gamma(d_1) = (v_3, v_4)(v_4, v_2)(v_2, v_5)$
●	$d_2 = (v_1, v_5, 80)$	$\gamma(d_2) = (v_1, v_2)(v_2, v_4)(v_4, v_5)$

Fig. 1: Network example with two demands additionally depicting the tunnels classified to each demand.

4 Traffic Data

In this section we present our analysis of traffic data from a large European ISP and our generation of traffic patterns based on the data which were used in our network simulations in OMNeT++.

The purpose of this section is to describe our methodology for the transformation of traffic data from a large European ISP to dynamic traffic demands.

Data Collection Large ISPs constantly monitor their network, at certain time intervals they sample the traffic of the network, here a variety of techniques can be used such as flow-based monitoring, packet sniffing and network probes [5]. The traffic dataset from the ISP consists of demands from a source to a target in hourly time intervals. The number of demands present in the ISP traffic dataset consists of a fourth of all possible demands($|V| \times |V|$) as traffic is not present between all pairs of routers.

Traffic Data Analysis Analyzing the traffic data will help us better imitate the real world behavior of network traffic. This would then allow us to extend static demands for the Internet Topology Zoo, such that the traffic will better emulate real world traffic behavior. Not only does it increase the accuracy of our network model, but it also gives a more enhanced reflection of the real world scenario that the system would be deployed in.

We will structure the European ISP data as a set of sets of demands D. Each subset is denoted with T corresponding to the hour the demands are active,

$T = \{0, 1, \ldots, 23\}$ representing each hour over 24 hours i.e. $0, 1 \ldots 23$. The time interval $t \in T$ is a unique period where D_t is exclusively active.

This implies that each demand $d = (v_{in}, v_{out}, l)$ from the set of demands D^T can be represented as a $|T|$ dimensional vector $(v_{in}, \vec{v_{out}})$ by taking the demand from v_{in} to v_{out} from each set of demands $D_t \in D^T$.

Each element in the vector represents the traffic from v_{in} to v_{out} for the corresponding time interval t, i.e. the first element $(v_{in}, v_{out})_0$ is the traffic from v_{in} to v_{out} from $00 - 01$, the second element $(v_{in}, v_{out})_1$ is the traffic in the interval $01 - 02$ and so on.

Pattern Generation Let $(v_{in}, \vec{v_{out}}) = ((v_{in}, v_{out})_0, \ldots, (v_{in}, v_{out})_{|T|-1})$ be a vector depicting a demand over time. To find the pattern we convert each demand vector into a pattern: Each element in the vector is divided by the largest element in the vector to find the pattern. This is done for each demand which results in a set of demand vectors s.t. $0 \leq (v_{in}, v_{out})'_t \leq 1, \forall t \in T$. The resulting vector is then: $(v_{in}, \vec{v_{out}})' = ((v_{in}, v_{out})'_0, (v_{in}, v_{out})'_1, \ldots, (v_{in}, v_{out})'_{|T|-1})$. We denote the set of all pattern vectors as PV.

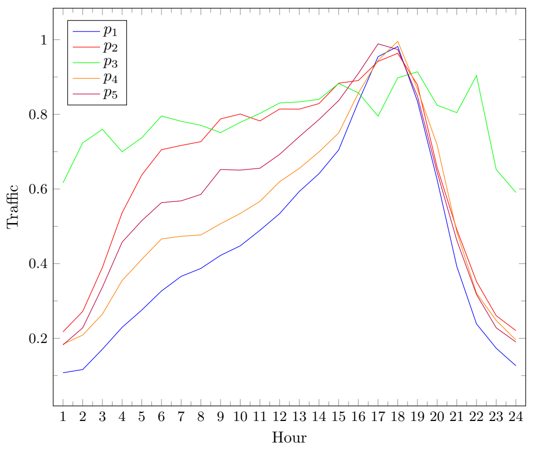


Fig. 2: Temporal patterns from large european ISP

Temporal Pattern Mapping By modelling the demands into a $|T|$-dimensional vector representation, we model the traffic data as a Gaussian mixture model[3, p. 430-438]. To do so we use the expectation maximization algorithm[3, p. 435] to fit a number of Gaussians onto all vectors in PV, in our case we use 5 distributions. To generate new demands, we sample the Gaussian distributions. Figure 2 visualizes the mean of each distribution.

Using the mixture of Gaussians we now map these patterns onto the topologies from the Internet Topology Zoo[22]. The patterns are used in tandem with the randomized gravity model created by Gay et al. [10]. To translate the static demands created by REPETITA[10], we do the following process for each network topology G and corresponding set of demands D:

1. Let $\vec{s} = \langle s_1, s_2, \ldots, s_{|T|-1} \rangle$ be a sample from PV and let $d = (v_{in}, v_{out}, l) \in D$ be a demand sample.
2. For each element s_t we generate the demand $d'_t = (v_{in}, v_{out}, s_t \cdot l)$ and add it to the set of dynamic demands D'_t.
3. We remove d from D and keep sampling PV and D sets until all demands have been exhausted.

Thus we end up with the set $\{D_0, D_1, \ldots, D_{23}\}$. After all dynamic demands have been added to D'_t, we use the set to simulate the dynamic network environment.

5 Traffic Engineering In Dynamic Networks

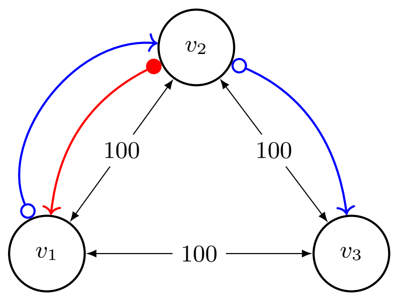
We have introduced the formal definitions for the network, traffic and routing that we will be using in the rest of the paper. In this section we present our model for dynamic data plane recomputation. Previous studies such as [24] show that both link failures and router failures happen in networks regularly. We want to create a traffic engineering algorithm that performs well during these failures. We motivate the use of global data plane recomputation through the use of a small example.

Figure 3 shows a comparison of failure recovery using data plane recomputation compared to the fast failover approach, which is when a router has a preinstalled backup tunnel that it uses if its next hop has failed. We have two demands being routed without causing congestion when all links are up on Figure 3a. Consider now Figure 3b where (v_2, v_3) has failed. In that instance, the traffic of d_2 is rerouted at v_2 on the backup tunnel $(v_2, v_1)(v_1, v_3)$. This causes congestion on link (v_2, v_1), as the capacity of the link is shared with the red demand d_1. Conversely, if the data plane was recomputed globally, the traffic of d_2 could simply be routed directly from v_1 to v_3 on link (v_1, v_3). This is a simple showcase of the limitations of fast failover in the context of traffic engineering.

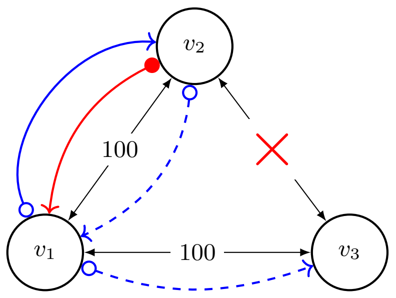
There is another motivation for dynamically updating the data plane. In the traffic data we received from the ISP, we observe that demands change dynamically over time. In order to maintain a high percentage of data reaching its destination, the data plane should be updated dynamically to account for the changing demands.

These two observations form the foundation for the problem that we seek to create a solution for:

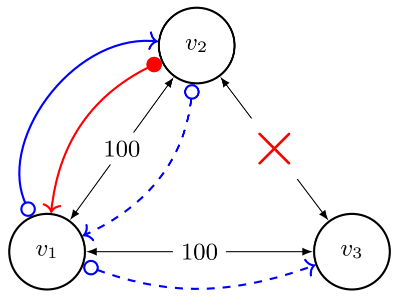
How can we maximize the data delivery percentage
in a network environment with dynamic demand changes
and possible link failures.



(a) No failed links



(b) One failed link



(c) One failed link

$d_1 = (v_2, v_1, 100)$	●
$d_2 = (v_1, v_3, 100)$	○

	Utilization	
Link	Non-failed	Single fail
(v_1, v_2)	1	1
(v_2, v_3)	1	fail
(v_1, v_3)	0	1
(v_2, v_1)	1	2
(v_3, v_2)	0	fail
(v_3, v_1)	0	1

Fig. 3: Example comparing the potential of fast failover to dynamic dataplane recomputation.

As mentioned previously, we want to ensure a high data delivery percentage in a dynamic network environment. Therefore we introduce the dynamic network model of Figure 4. This type of model will be realized through a SDN, s.t. the running network reflects the data plane and the update component reflects the control plane.

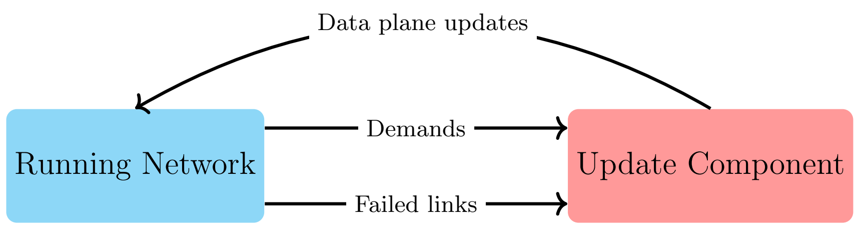


Fig. 4: Dynamic Network Model. The update component intermittently sends data planes to the network based on the demands and failed links.

In this model, the update component receives snapshots of the current traffic demands and failed links from the running network with some interval.

The computation of new data planes is delegated to the update component, which uses the demands and failed links to recompute new data planes. The model is temporal, and the flow of the model is depicted on Figure 5. The demands and failed links are emitted by the network at a fixed rate. These demands and failed links are used as input for the update component, which generates a new data plane and sends the updated data plane back to the network.

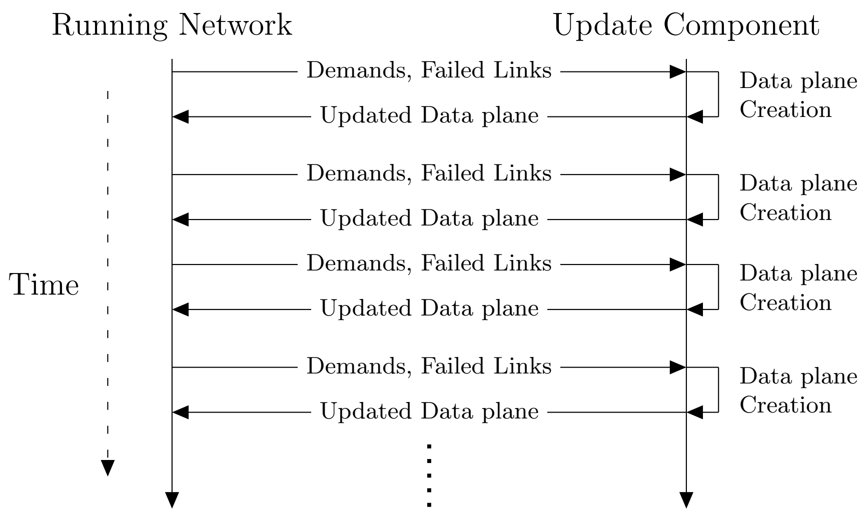


Fig. 5: Flow of the dynamic network model over time.

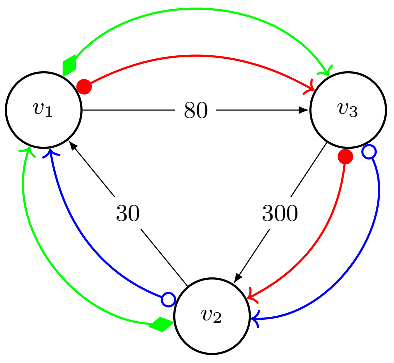
Figure 5 depicts the flow of the model over time. Over time the network outputs the traffic demands and failed links, and the update component recomputes the data plane and sends the updates back to the the network.

6 Simulation Framework

This section will detail the considerations we made when deciding how to implement and simulate our dynamic model. For this purpose we will compare two methodologies for evaluating traffic engineering algorithms; a flow level simulation, and a packet level simulation..

6.1 Issues with flow-level granularity

Many traffic engineering papers simulate their algorithm at a flow level, with tools such as MPLS-kit by Vanerio et al. [29] or REPETITA by Gay et al. [11].



D =

$d_1 = (v_1, v_2, 35\ )$	●
$d_2 = (v_3, v_1, 70\ )$	○
$d_3 = (v_2, v_3, 150\ )$	◆

	Link Utilization		
Link	64 B	264 B	Flow-based
(v_1, v_3)	1	0.78	2.31
(v_2, v_1)	1	1	7.33
(v_3, v_2)	0.56	0.40	0.35

Fig. 6: Cyclic network example

A problem with the realism of flow-level granularity is depicted on Figure 6. Observe the three demands depicted by the red circle, blue ring and green diamond. The three demands are routed using the tunnels depicted by the corresponding flows in the network figure. The problem with the flow-based simulation approach is measuring how much traffic is being dropped when there is congestion on a link. For example, the utilization on link (v_1, v_3) is 2.31, so a naive guess is that $\frac{1.31}{2.31}$ of the fraction of data will be dropped on this link. This logic is incorrect in reality, as part of the traffic of flow d_3 is being dropped on the previous link (v_2, v_1). Thus the real utilization on link (v_1, v_3) is less than 2.31. But how much less depends on how much data d_3 is dropping on (v_2, v_1), which depends on how much traffic d_2 is dropping on (v_3, v_2), which depends on how much data d_1 is dropping on (v_1, v_3) and so forth. Thus we end up with a cyclical dependency when trying to compute how much traffic is being dropped from each flow.

Another concern is with the added size of packet labels that is appended to packets. Consider some demand $d = (v, v', 100000kbps)$ in a network. In reality, the total data to be forwarded is more than 100000 kbps as the data is partitioned into packets that require packet headers. For example, a UDP packet being forwarded in an IPv4 network with MPLS, requires a header size of at least $8 + 20 + 4 = 32$ bytes. Also packet headers can vary in size depending on metadata, with IPv4 headers ranging in size from 20 to 60 bytes. We also showcase this in Figure 6 by using a packet level simulation. The table in the figure shows the link utilizations depending on the size of the packets simulated. Observe that simulating with 264 B packets results in less utilization than 64 B packets. This is because the demands are split into fewer packets, thus creating fewer labels. The reason why no link exceed 1 utilization is because the packets are dropped if the link becomes congested.

Due to the above concerns with flow-level simulations, we decided to run all the experiments in this paper using a packet level simulation tool.

For this purpose, we run our simulations in the OMNeT++ simulation environment.

6.2 OMNeT++ As A Packet Level Simulator

OMNeT++ is a discrete event simulation environment made in C++ [25]. We use the INET framework for OMNeT++, which provides tools and modules for simulating communication networks at the packet level. Our algorithms are implemented using an MPLS model in Python.

In order to simulate our experiments using OMNeT++ we made the following additions:

- A two-phase-commit module for data plane updates.
- A link weighting module that allows custom forwarding splitting ratios to be implemented (needed for GAOSPF and ASP)
- A module which intermittently (at a customizable rate) outputs the demands, link utilizations and failure events.
- Dynamic statistics collection to get exact numbers of packet drops, plots of link utilization over time, maximum link utilization, etc.
- A Dynamic Sending Interval function for UDP hosts, which dynamically updates the demands to simulate the traffic demands described in Subsection 4.

The two-phase-commit protocol takes as input a file of data plane updates. It splits the update into two phases, an *add* phase and a *remove* phase. The add phase adds all the new MPLS rules to the data plane, and is implemented immediately. The remove phase removes the MPLS rules that are no longer needed, and is implemented after a short duration when the *add* rules have been implemented and the packets using the labels of the old data plane have flushed the network. By using this two-phase approach, no intermittent routing will be created that results in packets being dropped, thus increasing the packet delivery percentage.

The link weighting module takes an XML file as input that defines the splitting ratios when using flow splitting algorithms. This functionality is used to implement the ASP algorithm.

The output module reads data from the network and intermittently outputs link utilization, traffic demands and failed links into json files. This represents the dynamic network component of our model.

The dynamic statistics collection is used in the evaluation of our algorithm. In order to get the packet delivery rate, we count the number of packets that reach their target and divide that number by the number of packets that entered the network.

The dynamic sending interval was a change we made to the already existing UDP host module, to allow the usage of the dynamic demands that were generated from the traffic data that we acquired from the large european ISP.

In total our contributions to OMNeT++/INET amount to around 1500 lines of code. Our contributions can be found at our repository[1]. Most of our code is in the inet/src/inet/p10 directory.

6.3 Simulation scale impact on precision.

When simulating a network in OMNeT++, simulation time can be very long, as every single packet has to be routed. In order to speed up the simulations, We scaled down the size of the simulation to see how this would affect the precision of the recorded measures. When scaling the simulation to size α, we generate the scaled demands:

$$D_\alpha = \{(v_{in}, v_{out}, \alpha l) | (v_{in}, v_{out}, l) \in D\}$$

And we scale the capacities such that $c_\alpha(e) = \alpha c(e)$.

Figure 7 shows the average deviation from perfect precision, in terms of the the values gathered with $\alpha = 1$. We compare with link utilization. So a y-value on the plot of e.g. 0.2 for some α' means that the recorded link utilization was 0.2% off from the the link utilization of $\alpha = 1$. Based on these results, we used an $\alpha \approx 0.1$.

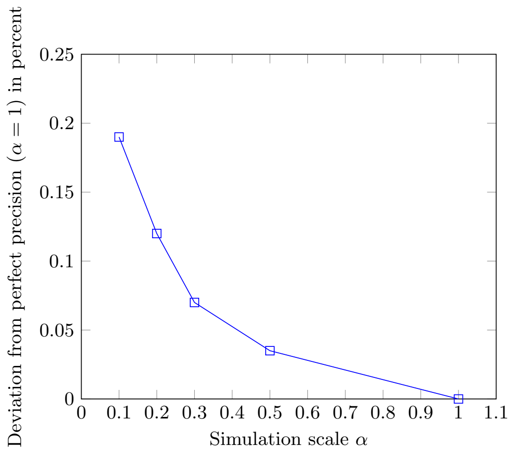


Fig. 7: Precision experiment. Shows that the deviation from perfect precision is quite small even with $\alpha = 0.1$

[1] https://github.com/Lassebk45/inet

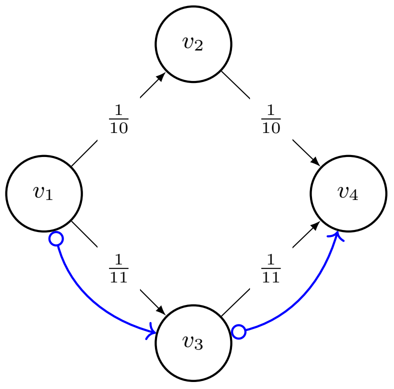
7 Spungeet

In this section we propose our solution to the problem formulation. We propose Spungeet, a genetic traffic engineering algorithm. In our dynamic network model, Spungeet corresponds to the update component.

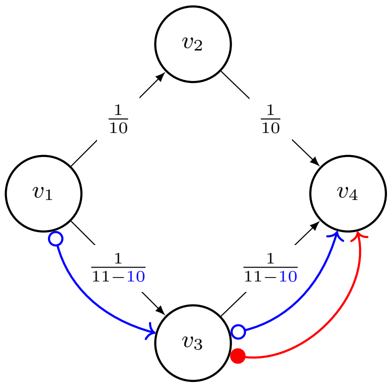
7.1 Intuition

The main intuition of the algorithm, is based on the belief that forwarding traffic on links with a lot of excess capacity is preferred as opposed to using links that are close to congestion. An idea of how to do this, is by routing on the shortest path in a graph representing the network, where link weights are set to $w(e) = \frac{1}{c'(e)}$ where $c'(e)$ denotes the excess capacity on the link (we disregard the cases where the load on the link exceeds the capacity for simplicity in the intuition). By using this weight function, links with a lot of excess capacity are weighted lower and therefore more likely to be used in the shortest path.

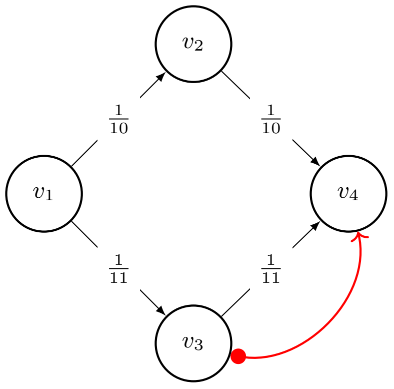
An example of the usage of this weighting function is depicted in Figure 8. The capacities are $c((v_1, v_2)) = 10$, $c((v_1, v_3)) = 11$, $c((v_2, v_4)) = 10$, $c((v_3, v_4)) = 11$ leading to the inverse capacities as observed in Subfigure 8a on the edges. We want to create tunnels for d_1 and d_2. When creating the tunnel $\gamma(d_1)$ first, the shortest path is $\gamma(d_1) = (v_1, v_3)(v_3, v_4)$ with the length $\frac{1}{11} + \frac{1}{11} = \frac{2}{11}$. We then subtract the load of d_1, 10, from the remaining capacities of edges (v_1, v_3) and (v_3, v_4), giving the updated link weights $w((v_1, v_3)) = \frac{1}{11-10} = 1$ and $w((v_3, v_4)) = \frac{1}{11-10} = 1$. These new weights are depicted on Subfigure 8b. When computing $\gamma(d_2)$ second, the shortest path is $\gamma(d_2) = (v_3, v_4)$. Computing the tunnels in this order results in congestion on link (v_3, v_4) as depicted in the table. Subfigures 8c and 8d depict the tunnels that are generated if the order of tunnel computation is reversed. Then $\gamma(d_2) = (v_3, v_4)$ and $\gamma(d_1) = (v_1, v_2)(v_2, v_4)$ are the computed tunnels. These tunnels create no congestion in the network.



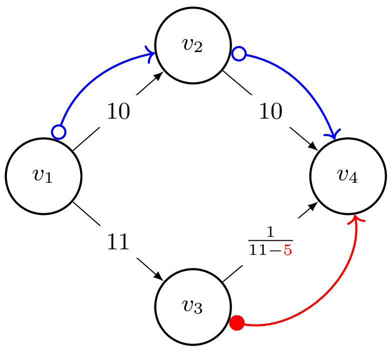
(a) d_1 first. $\gamma(d_1) = (v_1, v_3)(v_3, v_4)$



(b) d_2 second. $\gamma(d_2) = (v_3, v_4)$



(c) d_2 first. $\gamma(d_2) = (v_3, v_4)$



(d) d_1 second. $\gamma(d_1) = (v_1, v_2)(v_2, v_4)$

$d_1 = (v_1, v_4, 10)$	○
$d_2 = (v_3, v_4, 5)$	●

		Utilization	
Link	capacity	$\gamma(d_1) \rightarrow \gamma(d_2)$	$\gamma(d_2) \rightarrow \gamma(d_1)$
(v_1, v_2)	10	0	1
(v_2, v_4)	10	0	1
(v_1, v_3)	11	0.9090	0
(v_3, v_4)	11	1.5000	0.5000

Fig. 8: The tunnels created by using the shortest path with edge weights $w(e) = \frac{1}{c'(e)}$. When a tunnel $\gamma(d)$ is created for a demand $d = (v_{in}, v_{out}, l)$, then $c'(e) := c'(e) - l$ for the edges used in the tunnel. The notation $\gamma(d) \rightarrow \gamma(d')$ denotes that a tunnel was created for d and then for d'. The weights on the edges denote the weights that were used to calculate the shortest path in the network

Using the inverse capacity of links as weights for a shortest tunnel algorithm, has also been the default OSPF cost for Cisco, a major network technology firm[8]. In our case it helps us encourage the distribution of tunnels in the network.

We use the example in Figure 8 as the motivation for Spungeet, which seeks to find an ordering of the demands that reduces congestion, when computing the tunnels in that order.

As mentioned earlier, Spungeet is a genetic algorithm, which we now introduce the general concept of.

7.2 Genetic Algorithms

A genetic algorithm belongs to the class of algorithms called evolutionary algorithms[6][16][15]. These algorithms work by iteratively making new solution proposals by combining previous solutions.

Since a genetic algorithm is very much inspired by natural evolution, many technical terms are inherited from the field. An *individual* represents the solution (in this case a classification function γ) and the *population* is the set of all individuals. Based on the intuition for the algorithm, we define an individual as follows.

Definition 5 (Individual). Let D be a set of demands. An individual is a vector $\vec{ind} = \langle w_0, w_1 \dots w_{|D|} \rangle$ where $w_i \in \mathbb{N}$ represents a weight for demand d_i. A higher weight means the demand has a higher priority.

Algorithm 1: Converting an individual into a classification table γ

```
Input  : Individual $\vec{i}$
         Set of demands $D$
         Directed graph $G' = (V, E \setminus X, c)$
Output: A classification table $\gamma$, mapping a demand to a path
1 $W : E \to (0, 1]$ defined as $W(e) = \frac{1}{\max(1, c(e))}$ for all edges.
2 $\gamma(d)$ is undefined for all demands
3 $D' := D$ ordered by $\vec{i}$ in descending order
4 for $d' = (v_{in}, v_{out}, l)$ in $D'$ do
5     $f :=$ ShortestPath$(G', v_{in}, v_{out}, W)$
6     $\gamma(d) := f$
7     for $edge \in f$ do
8         $c(edge) := c(edge) - l$
          /* This updates the weight function $W$ */
9 return $\gamma$
```

It is fundamental for a genetic algorithm that each individual can represent a solution to the problem. Therefore, we define Algorithm 1, which describes how the classification function can be computed from the individual. This algorithm is based on the inverse capacity shortest path algorithm, as explained in the intuition. In the algorithm on line 1 we instantiate the inverse capacity of links as the initial link weights. We use $max(1, c(e))$ to stop division with a negative number. In line 3 we order the demands D according to the permutation defined by sorting $\vec{i}$ in descending order. Afterwards we iterate over the demands in the sorted order, creating a shortest path for each demand in the network using the inverse capacity weighting. When the path is found we add the path as a tunnel to the classification and subtract the load of the demand from the remaining capacity of each link in the path. This happens on line 7-8. This implies that the inverse capacity function returns a new value for each edge in the path.

Algorithm 2: Genetic algorithm

```
1  $n := 0$
2  Create initial population $P_0$
3  Evaluate $P_0$
4  while time limit not exceeded do
5      $n := n + 1$
6      $P_n := \emptyset$
7      while $|P_n| < |P_{n-1}|$ do
8          Select parents based on evaluation from $P_{n-1}$
9          Combine parents to create child
10         Mutate child with some probability
11         Add child to $P_n$
12     Evaluate $P_n$
13 return Best individual in $P_n$
```

Definition 6 (Population). A population is the multiset $pop = \{\overrightarrow{i_1}, \overrightarrow{i_2} \ldots \overrightarrow{i_n}\}$ of all the individuals.

With an individual and population defined, we can now describe the outline of Spungeet in Algorithm 2. $|P|$ denotes the size of the population. The algorithm proceeds by generating an initial population and evaluating each inidividual. An individual is evaluated using its fitness score. Figure 9 shows the flow of how an individual is used to compute a fitness score. We then select parents based on the evaluation, and combine them to create the individuals in the next generation. When the time limit is exceeded, the algorithm returns the individual with the best fitness score.

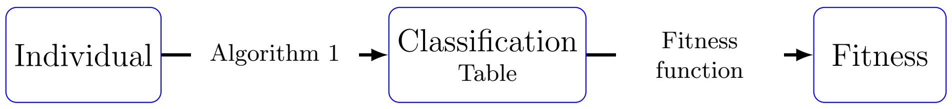


Fig. 9: Model showing how an individual is used to obtain a data plane and a fitness score

7.3 Elitism

To maintain solutions across generations we employ elitism[6, p.89]. Elitism is a method that puts the best individuals to the next generation without modification and as a result the fitness score cannot drop between iterations. Furthermore this technique compliments short update intervals in a changing network environment. This is based on the fact that losing the best individual between iterations can reduce the quality of the resulting dataplane and the primary drawback of elitism, which is stagnating at a local optima, is less severe.

7.4 Crossover

Crossover is the process of combining two individuals to generate a new individual. For the Spungeet algorithm we use the crossover function random keys[2], which inhibits the characteristics of a crossover function that enables good diversity and convergence. First, two parent individuals are selected; one from the elite class and one from the non-elite class. The crossover function works by iterating over each demand in the set of demands. For each demand the gene has 70% probability to be picked from the a parent from the elite class and 30% probability to be picked from the non-elite class. This is done until you have a weighting for each demand in the resulting child. The pseudo code for the function can be seen in Algorithm 3. The notation $c[d]$ denotes the element in c that corresponds to the demand d.

Algorithm 3: Spungeet crossover

```
Input  : Elite parent $\vec{p_1}$
         Non-elite parent $\vec{p_2}$
         A set of Demands $D$
Output: A child individual $c$
1 $\vec{c}$ := empty individual
2 for $d = (v_{in}, v_{out}, l) \in D$ do
3     if $rand(0,1) < 0.7$ then
4         $\vec{c}[d] := \vec{p_1}[d]$
5     else
6         $\vec{c}[d] := \vec{p_2}[d]$
7 return $c$
```

7.5 Mutation

A child has 10% chance of mutating. If this happens, all of the child's weights are randomized in the range $1 \leq dw_i \leq 10 \times |D|$. This seeks to negate a populations stagnation of the fitness score, which can occur in genetic algorithms[6, p. 32].

7.6 Fitness function

To assess individuals, we use a fitness function. The fitness function takes an individual and assesses how good a solution the individual brings forth. This part is very important as the solution's assessment has a direct effect on what is prioritized by the genetic algorithm[6]. In the field of traffic engineering there exists a few different optimization objectives including maximum link utilization[21][27] or maximizing throughput[17]. In our case we chose to use an optimization objective inspired by the Fortz and Thorup piece-wise linear function[9], which we also employed in the previous project Essence[13]. The Fortz and Thorup inspired function Φ is visualized in Figure 10 and defined as:

$$\Phi(u) = \begin{cases} 0.1u & \text{for } u \leq \frac{1}{20} \\ 0.3u - 0.01 & \text{for } \frac{1}{20} < u \leq \frac{1}{10} \\ u - 0.08 & \text{for } \frac{1}{10} < u \leq \frac{1}{6} \\ 2u - 0.24666 & \text{for } \frac{1}{6} < u \leq \frac{1}{3} \\ 5u - 1.24666 & \text{for } \frac{1}{3} < u \leq \frac{1}{2} \\ 10u - 3.74666 & \text{for } \frac{1}{2} < u \leq \frac{2}{3} \\ 20u - 10.41333 & \text{for } \frac{2}{3} < u \leq \frac{9}{10} \\ 70u - 55.41333 & \text{for } \frac{9}{10} < u \leq 1 \\ 500u - 485.41333 & \text{for } 1 < u \leq \frac{11}{10} \\ 5000u - 5435.41333 & \text{for } u > \frac{11}{10}\,. \end{cases}$$

Let $G = (V, E, c)$ be the network topology, $\vec{i}$ be an individual and γ be the path function returned by Algorithm 1. Then the fitness of individual $\vec{i}$ is calculated by:

$$\sum_{e \in E} \Phi(u(e)) \cdot c(e)$$

The goal of the fitness function is to assess how good the classification function corresponding with the individual is. The intuition behind this fitness function is mainly that the demands with a high weighting are given the freedom to take up the shortest routes, whereas other demands that have a lower weight are forced to choose an alternative if their shortest routes are occupied. As such, it encourages a diverse distribution which both reduces utilization on links, but also naturally reduces the length of routes. The process of extracting the fitness score from the individual is depicted in Figure 9.

7.7 Storing individuals between update cycles

Since the update cycles can be very short, we have chosen to save individuals between update cycles. The way we do it is by saving all the elite individuals and inserting them into the next cycles initial population. This is especially useful if the demands do not change much, as the strong individuals in the previous cycle will be strong again.

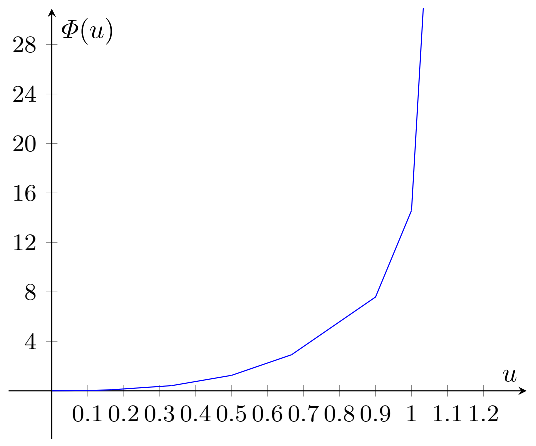


Fig. 10: Plot of the function inspired by Fortz and Thorup[9]

7.8 Parameters

Through experiments, we converged to these parameters, which brought out the best results for the genetic algorithm. They are as follows:

- Population size: The population size determines the number of individuals that are created for each generation. The trade-off is that a larger population size gives a bigger search space, but slower convergence rate and a smaller population does the opposite. To strike a balance in the dynamic network environment, we choose to use a population size of 200.
- Crossover function (random-keys) weighting: The random-keys weighting describes the probability that a demand weight will be picked from one or the other individual. In our case the probability ration of 70% chance of choosing the elite demand weight, and 30% chance of choosing the non-elite demand weight worked best. Again the trade-off is if the weighting is more skewed towards the non-elite, the search space becomes bigger and convergence slower and vice versa.
- Mutation probability: The rate of randomly mutating a child individual, randomizing all demand weights. The technique effectively seeks to introduce new orderings for demands and as a result help escape local optima. This rate is set to 10%.
- Elite population percentage: This percentage denotes the number of individuals belonging to the elite class. We set this percentage to 20% of the population, primarily to maintain the fitness score between iterations.

8 Evaluation Framework

In this section we present the performance of Spungeet. We measure the performance of the algorithm based on its *connectivity*, which is the fraction of data that reaches its target (i.e. 50% connectivity means half of the traffic reached its destination).

We compare the algorithm against GAOSPF [8] and the (ASP) algorithm used by the european ISP.

We evaluate the performance on the network topologies of the Internet Topology Zoo[22]. The Internet Topology Zoo is a well studied collection of real world ISP topologies, that contains 260 real world topologies from various ISPs. The demands are created by REPETITA[10] a tool creating traffic data based on a randomized gravity model which has shown to be a realistic synthesis method for demand matrices. The gravity model creates demands for a router such that incoming traffic is proportional to the sum of its outgoing link capacities. The traffic generated is scaled such that maximum link utilization is equal to 90% when the set of failed links is empty.

8.1 Parallelization of Spungeet split

Previous papers have shown the strength of parallelizing genetic algorithms [12]. Spungeet also takes advantage of this technique, by parallelizing a costly part of the algorithm: the fitness function. This parallelization incorporates distributing the computation of the fitness score of individuals over multiple cores. The calculation of the fitness function can be slow on large topologies and this is especially amplified when working in a dynamic network environment were swift response to failures is crucial. We saw a dimishing gain from increasing the number of cores and settled at 8 cores per simulation. On a medium sized topology the average speedup from 1 to 4 cores was 72.42%, the average speedup from 4 to 8 cores was 27.49% and the speedup from 8 to 12 cores was 9.73%. This scaling is naturally affected by Amdahl's law[14].

We now present the experiments we ran to evaluate the performance of Spungeet. We compare against the ASP algorithm that is used on the large european ISP, as well as the other genetic algorithm GAOSPF, which we explained in Section 2.

8.2 Setup

The OMNeT++ simulations and the Python update component ran on a Slurm cluster with Ubuntu 18.04.5. The jobs were parallalized using 8 cores on an AMD Opteron^{TM} Processor 6376 and 32 GB RAM. Each simulation was run for approximately 3 hours, simulating the 3 peak hours from the traffic patterns(hours 16-19). The experiments are conducted with a 10 second interval between dataplane updates.

8.3 Network selection

We ran our experiments on 40 randomly selected networks from the Topology Zoo [22], which consists of more than 250 real world topologies ranging in size from 4 nodes to 197. We apply a pruning technique where we remove all the one-degree routers (edge routers) from the network as these routers only have a single link to route traffic on, and thus their traffic demands can be trivially aggregated at the next router. This is done recursively, as a result shrinking the networks makes the simulations run faster and lets Spungeet and GAOSPF run more generations.

8.4 Percentage of delivered packets

We evaluate the performance of Spungeet by counting the fraction of packets that reach their target. Figure 11a depicts the percentage of delivered packets of Spungeet compared to GAOSPF and (ASP). In all instances, Spungeet outperforms both competing algorithms by a considerable amount, on average Spungeet is 2.17% better than GAOSPF and 3.16% better than (ASP). Note the worst three cases, where GAOSPF and ASP achieve between 82% and 84% packet delivery, whereas Spungeet delivers over 90% of packets in all simulations. Additionally Spungeet achieves 100% packet delivery for 9 topologies, which GAOSPF and ASP can replicate for only 5 topologies.

8.5 Failure scenarios

To investigate the performance of Spungeet under the threat of failure, we introduce probabilistic failure scenarios and compare the performance of Spungeet, ASP and GAOSPF. In the generated scenarios we randomly fail nodes and links based on a preset probability. Our experiments incorporates a range of failure probabilities ranging from 5% to 25%. If a node fails all ingoing and outgoing links become unavailable. If a link fails it becomes unusable. We use the ratio of failures described in [24], which are 70% link failures and 30% node failures.

To increase the impact of failures our failure model uses a weighted probability of node and link failures, where nodes connected to links with high capacity and links with high capacity are weighted higher. The weighting is done by dividing by the maximal sum for node failures and maximal link capacity for link failures. In effect this favors more threatening scenarios that most likely results in dropping packets.

For the failure scenarios we simulated 5 failure scenarios on 8 topologies for each of the algorithms.

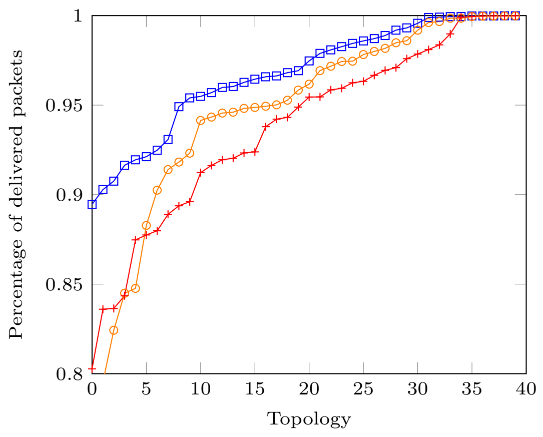
8.6 Percentage of delivered packets under failure scenarios

We also test the performance of Spungeet in failure scenarios, to evaluate how effective the recomputation of the data plane is at traffic engineering when a failure occurs.

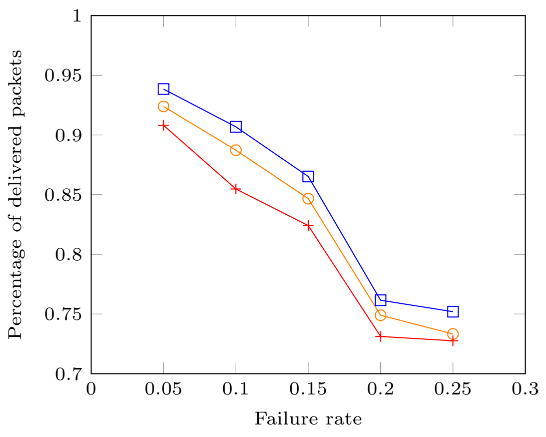
For fast failover, we deploy the widespread standard facility node protection as is detailed in [26]. This is a basic protection scheme that routes traffic around failed nodes by following the shortest path. If there is no shortest path around the node, it instead reroutes around the link. The Percentage of delivered packets under failure plot shows that Spungeet is slightly better than both GAOSPF and the (ASP)) algorithm. Spungeet is on average 2% better than GAOSPF and 4.31% better than (ASP). Each point in the plot shows the average percentage of delivered packets over the 8 topologies for each of the algorithms.

8.7 Maximum Link Utilization

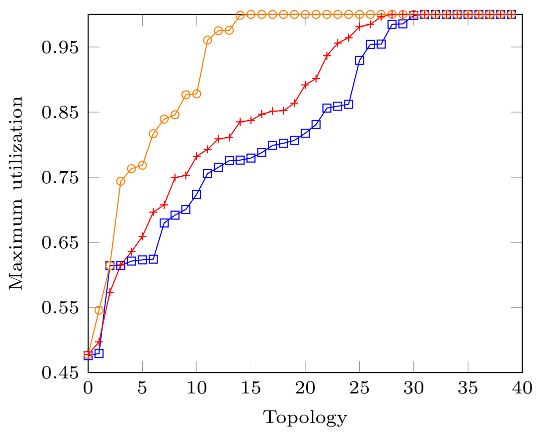
To see how well the algorithms avoid congestion on links and distribute the load, we use a classic traffic engineering measure of maximum link utilization. This measure gives use the worst case in the network, low maximum link utilization generally characterizes the dataplane as having a good distribution of load in the network. We evaluate the performance of Spungeet by gathering the largest utilization amount for each topology. Figure 11c depicts the amount of maximum utilization for each topology for Spungeet, GAOSPF and (ASP). Note that for maximum utilization the lowest score is desirable, since maximum utilization is negative measure. In all instances, Spungeet outperforms both competing algorithms by a considerable amount, on average Spungeet is 11.86% better than GAOSPF and 3.96% better than (ASP). Note how GAOSPF is only capable of achieving under 75% maximum utilization for 4 topologies while (ASP) has 9 under 75% compared with 11 topologies for Spungeet.



(a) Percentage of delivered packets



(b) Percentage of delivered packets under failure



(c) Maximum link util

Fig. 11: Comparison of Spungeet, GAOSPF, All Shortest Path.

8.8 Summary

Our experimental evaluation shows that Spungeet is consistently better than the competing genetic algorithm GAOSPF. We see Spungeet has significantly better connectivity on 5 topologies in Figure 11a. In Figure 11c we observe that Spungeet again beats the competition with a lower maximum link utilization in all but one case where the ASP algorithm is better. Although all three algorithms employ fast failover protection, we again see that Spungeet out competes both of the other algorithms. Based on these results we can conclude that Spungeet in consistently better that its competitors in all three measures.

9 Conclusion

We look at two challenges of the Traffic Engineering domain, namely the constantly changing demands of modern Software Defined Networks(SDNs) and the constant risk of links in the network suddenly failing. With these two challenges in mind we set out to design a system which would allow for monitoring and reacting to changes in a dynamic network. In our pursuit of designing this system we worked with the discrete event simulation tool OMNeT++. Our first contribution comes from the several non-trivial extensions to the simulation tool as we saw it necessary for the purpose of having more realistic simulation of the network. Our second contribution is the synthesis and transformation of real world traffic data from a Large European ISP, onto the Internet Topology Zoo traffic demands, this both increases the accuracy of our network model but also gives a better reflection of the real world scenario that our system would be deployed in. Our third and last contribution is the implementation and testing of our own genetic algorithm Spungeet. Spungeet works by first randomly assigning weights to each demand, it then goes through an interative process where it first evaluates each individual of the population, combines individuals through crossover and mutates some of the individuals. When the update time has passed, Spungeet stores elite individuals for the next update round and then creates and installs the data plane based on the best individual. Running experiments for both connectivity and packet delivery percentage shows that Spungeet is capable of beating out both All Shortest Path routing used by the Large European ISP and the GAOSPF algorithm which is a genetic algorithm for the OSPF problem. Due to the results of our experimentation there is good reason to believe that Spungeet is a worthy solution for reacting to changes in a dynamic network.

10 Future Work

In this section we will outline some of the ideas that we believe could improve this project if we were to continue working on it.

One of the ways in which we could extend the Spungeet algorithm would be to add predictive data plane generation. This could possibly help create better solutions by analysing the info received from the dynamic network, and trying to predict what state the network is going to be in later, such that it can precompute data planes and immediately install these when needed. Furthermore we could also try to add congestion aware fast rerouting to the Spungeet algorithm. This would be beneficial since rerouting often causes congestion on the links that are used for the rerouting. Therefore a solution which tries to avoid congestion when it reroutes a demand, is desirable.

Bibliography

[1] Bankhamer, G., Elsässer, R., Schmid, S.: Local fast rerouting with low congestion: A randomized approach. IEEE/ACM Transactions on Networking 30(6), 2403–2418 (2022)

[2] Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA journal on computing 6(2), 154–160 (1994)

[3] Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg (2006)

[4] Bogle, J., Bhatia, N., Ghobadi, M., Menache, I., Bjørner, N., Valadarsky, A., Schapira, M.: Teavar: striking the right utilization-availability balance in wan traffic engineering. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 29–43 (2019)

[5] Cecil, A.: A summary of network traffic monitoring and analysis techniques. Computer Systems Analysis pp. 4–7 (2006)

[6] Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer Publishing Company, Incorporated, 2nd edn. (2015)

[7] Elwalid, A., Chiu, A., Xiao, X., i_widjaja@yahoo.com, Awduche, D.O.: Overview and Principles of Internet Traffic Engineering. RFC 3272 (May 2002), https://www.rfc-editor.org/info/rfc3272

[8] Ericsson, M., Resende, M.G.C., Pardalos, P.M.: A genetic algorithm for the weight setting problem in ospf routing. Journal of combinatorial optimization 6(3), 299–333 (2002)

[9] Fortz, B., Thorup, M.: Increasing internet capacity using local search. Computational Optimization and Applications 29, 13–48 (2004)

[10] Gay, S., Schaus, P., Vissicchio, S.: Repetita: Repeatable experiments for performance evaluation of traffic-engineering algorithms. arXiv preprint arXiv:1710.08665 (2017)

[11] Gay, S., Schaus, P., Vissicchio, S.: REPETITA: repeatable experiments for performance evaluation of traffic-engineering algorithms. CoRR abs/1710.08665 (2017), http://arxiv.org/abs/1710.08665

[12] Harada, T., Alba, E.: Parallel genetic algorithms: A useful survey. ACM Comput. Surv. 53(4) (aug 2020), https://doi.org/10.1145/3400031

[13] Herum, R.E.N., Kær, L.B., Lundbergh, M.H., Madsen, A.L., Nielsen, B.N.: Essence: A congestion aware genetic algorithm for fast rerouting (2022)

[14] Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer 41(7), 33–38 (2008)

[15] Holland, J.H.: Genetic algorithms and the optimal allocation of trials. SIAM journal on computing 2(2), 88–105 (1973)

[16] Holland, J.H.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press (1992)

[17] Hong, C.Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., Wattenhofer, R.: Achieving high utilization with software-driven wan. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM. pp. 15–26 (2013)

[18] Iannaccone, G., Chuah, C.n., Mortier, R., Bhattacharyya, S., Diot, C.: Analysis of link failures in an ip backbone. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment. pp. 237–242 (2002)

[19] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al.: B4: Experience with a globally-deployed software defined wan. ACM SIGCOMM Computer Communication Review 43(4), 3–14 (2013)

[20] Johansen, N.S., Kær, L.B., Madsen, A.L., Nielsen, K.Ø., Schmid, S., Srba, J., Tollund, R.G.: Fbr: Dynamic memory-aware fast rerouting (2022)

[21] Kandula, S., Katabi, D., Davie, B., Charny, A.: Walking the tightrope: Responsive yet stable traffic engineering. ACM SIGCOMM Computer Communication Review 35(4), 253–264 (2005)

[22] Knight, S., Nguyen, H.X., Falkner, N., Bowden, R., Roughan, M.: The internet topology zoo. IEEE Journal on Selected Areas in Communications 29(9), 1765–1775 (2011)

[23] Labovitz, C., Ahuja, A., Jahanian, F.: Experimental study of internet stability and backbone failures. In: Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352). pp. 278–285. IEEE (1999)

[24] Markopoulou, A., Iannaccone, G., Bhattacharyya, S., Chuah, C.N., Diot, C.: Characterization of failures in an ip backbone. In: IEEE INFOCOM 2004. vol. 4, pp. 2307–2317 vol.4 (2004)

[25] OMNeT: Omnet++ 6.0.1 (2022), https://omnetpp.org/intro/

[26] Pan, P., Swallow, G., Atlas, A.: Fast reroute extensions to RSVP-TE for LSP tunnels. RFC 4090, 1–38 (2005), https://doi.org/10.17487/RFC4090

[27] Perry, Y., Frujeri, F.V., Hoch, C., Kandula, S., Menache, I., Schapira, M., Tamar, A.: DOTE: Rethinking (predictive) WAN traffic engineering. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). pp. 1557–1581. USENIX Association, Boston, MA (Apr 2023), https://www.usenix.org/conference/nsdi23/presentation/perry

[28] Ramakrishnan, N., Soni, T.: Network traffic prediction using recurrent neural networks. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 187–193 (2018)

[29] Vanerio, J., Schmid, S., Schou, M.K., Srba, J.: Mpls-kit: An mpls data plane toolkit. In: 2022 IEEE 11th International Conference on Cloud Networking (CloudNet). pp. 49–54. IEEE (2022)

[30] Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H.: A survey on software-defined networking. IEEE Communications Surveys & Tutorials 17(1), 27–51 (2014)