Controller Placement in Software Defined Networks

By

Talha Ibn Aziz
Shadman Protik

Supervised by

Prof. Muhammad Mahbub Alam PhD
Department Head
Department of Computer Science and Engineering
Islamic University of Technology

بسم الله الرحمن الرحيم

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
5 November, 2018

Declaration of Authorship

This is to certify that the work presented in this thesis is the outcome of the analysis and simulations carried out by **Talha Ibn Aziz** and **Shadman Protik** under the supervision of **Prof. Muhammad Mahbub Alam PhD**, Department Head of Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Dhaka, Bangladesh. It is also declared that neither of this thesis nor any part of this thesis has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Authors:

Talha Ibn Aziz
Student ID - 144435

Shadman Protik
Student ID - 144428

Supervisor:

Prof. Muhammad Mahbub Alam PhD
Department Head
Department of Computer Science and Engineering
Islamic University of Technology (IUT)

Acknowledgement

At the outset, we express utmost gratitude to Almighty Allah for His blessings which allowed us to shape this research into reality and give it form. This thesis owes its existence to a lot of people for their support, encouragement, and guidance. We would like to express our gratitude towards them.

We are very grateful to our supervisor **Prof. Muhammad Mahbub Alam**, PhD, Department Head, Department of Computer Science and Engineering, Islamic University of Technology (IUT), for his supervision, knowledge and support, which has been invaluable for us.

We are indebted to **Dr. Salimur Choudhury**, Assistant Professor, Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada. This research work would not see the light of success without his supervision, strategy, and line of action. He has been a constant source of enthusiasm and encouragement which has allowed us to turn this endeavor into a success story.

We are thankful to **Md Sakhawat Hossen**, Assistant Professor, Department of Computer Science and Engineering, Islamic University of Technology (IUT) for his insightful discussion and valuable advice.

Finally, we seize this opportunity to express our profound gratitude to our beloved parents for their love and continuous support both spiritually and mentally.

Abstract

Controller Placement Problem (CPP) is a promising research interest in the field of Software Defined Networking (SDN). SDN decouples the network layer of the traditional network model into a control plane and data plane. The control plane consists of controllers which provide the routing decisions for the switches. The CPP deals with placing an optimal number of controllers in the network so that the data transfer throughput of the network is maximum, which is NP-Hard as it deals with multiple constraints.

For years, several impressive solutions have been proposed with a goal to create an optimal network for SDN, one of such solutions is Density Based Controller Placement (DBCP). DBCP clusters the network based on the local density of the switches. DBCP uses hop count to calculate the latencies between switches and minimizes the overall latency, so it works with unweighted graphs. However, an unweighted graph is not a good representation of a real network environment. In this paper, we propose four algorithms, where three are inspired by SPICi, a protein-clustering algorithm of Bioinformatics and they work on weighted graphs. Our algorithms cluster a network based on the maximum connectivity of the nodes and uses the local search technique to improve the clustering in terms of flow-setup latency in polynomial time complexity, and our simulation results show that our proposed algorithms outperform the existing algorithms.

Several other solutions to the CPP have been proposed which work on various constraints– some approaches work with a single parameter like the total delay of a network, reliability, load balancing, etc., while some other approaches provide exhaustive solutions which optimize multiple parameters. However, very few researches propose non-exhaustive solutions which simultaneously optimize more than one parameter. We propose another novel controller placement algorithm which clusters the SDNs in polynomial time complexity and name it **Degree-based Balanced Clustering (DBC)**.

DBC minimizes overall flow-setup latency as well as route-synchronization latency and balances the loads of the controllers at the same time. DBC divides a network into several clusters, places a controller in each cluster, and also selects an optimal number of controllers. Simulation results suggest that DBC outperforms existing state-of-the-art algorithms in terms of different latencies and also performs load balancing among the controllers.

Contents

Chapter 1

Introduction

1.1 Overview

The traditional network model– the current framework being used, consists of seven network layers (OSI model [1]). Another categorization is the TCP/IP protocol suite which categorizes the network model into five layers [1]. The layers communicate with each other and perform their specific functions to enable secure, reliable, and efficient data transfer throughout the network. Due to their layered structure, adjusting to special requirements and adapting to layer-specific changes is easier. Each layer only performs its functionalities and co-ordinates with the layer above it or the layer below it. Therefore, any change in a single layer may only cause a small change in the two layers adjacent to it, without any major adjustments to other layers. However, as the switches become more intelligent and costly, maintaining the network and accommodating changes in the network configuration becomes troublesome as different types of switches have different vendors. Re-configuring the switches require a considerable amount of time and labor, not to mention adding more switches is more costly.

The introduction of the Internet of Things (IoT) will cause an overwhelming flow of data in the network which needs to be handled by the network without any major increase in resources [2]. In order to accomplish this, the network will have to be conscious about the traffic in different regions of the network and allow for intelligent traffic routing– sending packets through less congested switches rather than the shortest path. Consequently, the switches need to have a global view of the network, which becomes troublesome as broadcasting more data to let other switches know the condition of the network will make it even more congested. As a solution to all these problems, the Software-Defined Network (SDN) came into light through a series of events which started in the 1960s [3], and is still a field of vibrant research. The introduction of SDNs [4] aims to place a device with enhanced processing and memory instead of distributing the resources to all the switches. This not only allows for cost minimization, but also allows for a central entity (like a network manager) to monitor the entire network and implement other technologies like network virtualization and traffic controlling. Therefore, the job of handling the network also becomes easier along with the increased efficiency of the network.

1.2 SDN

SDN has been a sector of intensive research for a long period of time. In recent years SDN has received a lot of attention from researchers, academicians, businessmen and also from the governments. The concept of a programmable network is slowly being shaped into reality. This can be evident from the thorough history which can be found in [5, 6, 7].

In the 1960s, Paul Baran, a researcher of Rand Corporation, US, proposed to transmit autonomous data packets [3] through the network. Later, an attempt was made to make the packet forwarding even more intelligent by introducing policy-based routing (PBR) [8]. This marked the beginning of a new type of networking that had one goal– how to make routing intelligent? where, "intelligent" means considering the condition of the network at a given time. In contrast, the de facto routing methods always select the shortest paths to the destination. In order to facilitate intelligent routing, the routing tables need to be populated while taking into account the network variables like bandwidth, traffic, link failures etc. In the traditional network model [1], the routing decisions are taken by the network layer which uses various protocols to populate the routing table, some of which are– RIP (Routing Information Protocol), OSPF (Open Shortest Path First) and BGP (Border Gateway Protocol) [9]. These protocols use hop count as metric and select the path with the lowest number of hops and no failures for data flows. The routing decisions are taken by the routers themselves. However, to route intelligently, a separate entity is required which can keep track of network changes and maintain a global perspective as the network will be flooded with broadcast messages if the routers were to keep a global view.

The SDN architecture solves this problem by dividing the network layer into control plane and data plane to enable a division of functionalities. The control plane takes the routing decisions and the data plane forwards the data. The control plane consists of a single controller [4], which leads to the formation of a single point of failure and a bottleneck due to several requests directed at the controller, especially for larger networks. Other problems like scalability and reliability also arose, leading to the **Controller Placement Problem (CPP)**, which deals with placing multiple controllers in SDN.

1.3 Contribution

In this thesis, we work with latency minimization, traffic awareness, and load balancing, while placing multiple controllers in SDNs. The contribution of our work can be summarized as follows:

- We give several algorithms which work with weighted links between switches, thus allowing us to create clusters based on network traffic, bandwidth, transmission rate, etc., which are essential in determining the condition of a network.
- Our proposed algorithms cluster SDN networks in polynomial time complexity.
- Our algorithms are both static and dynamic traffic-aware. They consider traffic before (Static) and after (Dynamic) placing controllers while clustering.
- One of our algorithms minimize flow-setup latencies and balance load simultaneously.

1.4 Organization of Thesis

The thesis work is organized as follows. In chapter 2, we propose four algorithms, which work on weighted representations of a network to minimize latency and also provide both static and dynamic traffic-awareness. In chapter 3, we propose a more independent and compact algorithm which simultaneously minimizes flow-setup latency and the maximum loads of the controllers, considering the loads of each switch to be constant. In chapter 4, we provide a summary of our work and also show potential possibilities for future work.

Chapter 2

Latency Minimization

2.1 Introduction

Software Defined Network (SDN) is the network of the new era which separates the network layer into two planes, namely control plane and data plane. For a small network, the control plane can consist of only one controller as is done in the well known standard OpenFlow [10]. The problem arises when the network is too large for one controller to handle and the solution of multiple controllers is required. Multiple controllers make the network scalable and as a result, it is easy to control a larger number of switches. Moreover, it decreases the load of each controller and the controller-to-switch latency. Typically, controller placement is done in two steps: *Clustering*– dividing the network into multiple sub-networks and *Controller Selection*– placing a controller for each sub-network. The controller might be placed separately, or it might replace a switch in the cluster.

CPP is a recent field of research [11], where many state-of-the-art algorithms have been proposed, one of which is Density Based Controller Placement (DBCP) [12]. DBCP clusters the network by selecting an optimum value of k (number of clusters) i.e., divides the network into k disjoint sets of switches and selects one controller for each set. DBCP calculates the latencies between two switches using hop count. However, hop count is not a good measure of latency, because hop count does not reflect other contributing elements like processing delay, queuing delay, bandwidth, transmission rate etc. when representing source to destination path latency. Hence a composite metric that reflects the above observations is necessary for the CPP problem of SDN. To the best of our knowledge, there is no clustering algorithm which takes into consideration all of these parameters. In this paper, we consider weighted links between two switches which can be set to any of the parameters like bandwidth, queuing latency, transmission speed etc. We propose four algorithms - Random Clustering with Local Search (RCLS), Greedy-SPICi (G-SPICi), Inverse-SPICi (I-SPICi) and Modified-SPICi (M-SPICi). RCLS works with hop count while the other three which are variations of the algorithm SPICi [13], represent the network as weighted graphs. We explain in detail our algorithms in section 2.5.

The rest of this chapter is organized as follows: We represent the background and related works along with some other notable works of SDN and CPP in Section 2.2. Section 2.3 represents CPP as a graph theory problem. Section 2.4 explains the existing SDN clustering algorithm DBCP and the protein-clustering algorithm SPICi. We explain our proposed algorithms in Section 2.5. Simulation results and Performance Evaluation are presented in Section 2.6. We present a summary of this chapter in Section 2.7.

2.2 Background and Related Works

In SDN, the decisions are taken by a controller which keeps track of the changing traffic of the network and uses this knowledge to intelligently route traffic. The controller tells the switches where to send a new packet (e.g., OpenFlow [10, 2]). In this architecture, all the switches ask the controller for routing decisions, which creates a problem for single controller-based SDNs. When the network is not small, a bottle-neck is formed– the network becomes more congested in the course of time and eventually collapses. Thus multiple controllers become a necessity.

The single-controller architecture has already been implemented extensively: the controller OpenDayLight (ODL) [14] has been deployed several times in various companies like Orange, China Mobile, AT&T, TeliaSonera, T-Mobile, Comcast, KT Corporation, Telefonica, China Telecom, Globe Telecom, Deutsche Telekom [7]. However, the problems that are faced in the case of single controllers are scalability [15, 16] and reliability when the networks are large. Consequently, multiple controllers were proposed, and the foundation was laid by R. Sherwood et. al.[17] in 2009. Thereafter, multiple controllers have been used in several applications [18, 19, 20] and a lot of research have been directed towards it [21, 11]. The questions that arise due to multiple controllers are: *How many controllers?*, *Where to place them?* and *Which switch is assigned to which controller?* [22, 23]. These questions together is called the Controller Placement Problem (CPP) and is an emerging paradigm and a field of vibrant research in the domain of SDN.

CPP is primarily of two types[11]: **(1) Capacitated Controller Placement Problem (CCPP)** and **(2) Un-capacitated Controller Placement Problem (UCPP)** (Fig. 2.1). *CCPP* considers the capacity of the controllers and load of the switches when assigning switches to each controller. It may consider the load of the switches to be fixed or variable. CCPP is of four types: *Static Traffic-aware CCPP:* Considers the network traffic and capacity when placing controllers. *Dynamic Traffic-aware CCPP:* Changes clustering even after placing controllers, based on changing traffic. *Fault-aware CCPP:* Places controllers based

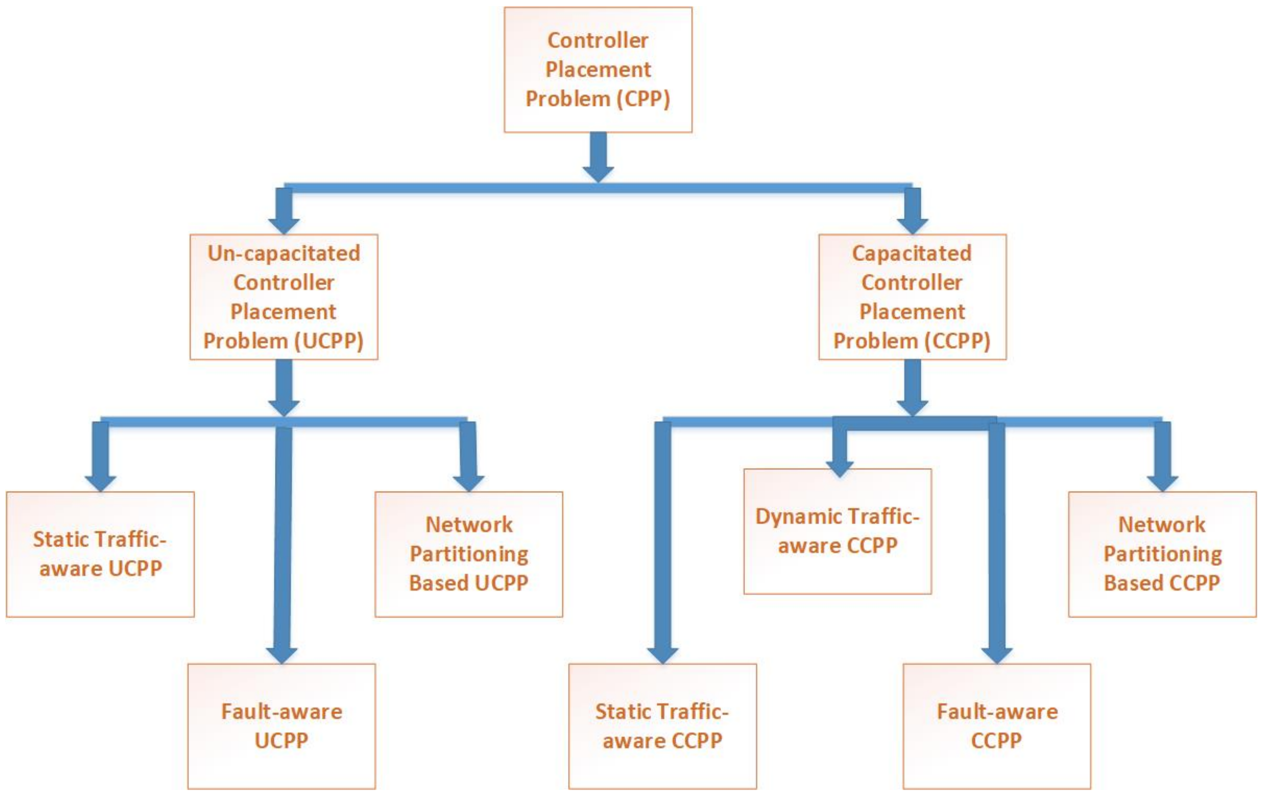
Controller Placement Problem (CPP)
Un-capacitated Controller Placement Problem (UCPP)
Capacitated Controller Placement Problem (CCPP)
Static Traffic-aware UCPP
Network Partitioning Based UCPP
Fault-aware UCPP
Dynamic Traffic-aware CCPP
Network Partitioning Based CCPP
Static Traffic-aware CCPP
Fault-aware CCPP

Figure 2.1: Classification of the Controller Placement Problem (CPP)

on reliability and resilience, i.e., to maximize fault-tolerance. *Network Partitioning Based CCPP:* Partitions the network based on capacity and may include parameters like scalability, manageability, privacy, and deployment. *UCPP* considers that the controllers have infinite capacity and only strives to maximize or minimize certain parameters of the network, separately or collectively, with the minimum number of controllers. UCPP is of three types, all of which are the uncapacitated versions of their capacitated counterparts (except Dynamic Traffic-aware CCPP).

In the current decade, many solutions have been proposed to solve this NP-Hard problem of controller placement [24, 21, 11]. Heller et. al. [24] propose a solution of CPP by selecting k controllers to minimize the average and maximum latency of the network. Sallahi et. al. [25] provide a mathematical model which simultaneously determines the controller numbers, locations and included switches while satisfying some constraints. The cost of installing a controller is an example of such a constraint. Yao et. al. [26] introduce capacitated controller placement, where they consider the load or capacity of a controller and the load of the switches, and also ensure that the capacity of a controller is not exceeded. Ozsoy et. al. [27] propose an advanced version of the k-center algorithm using links between switches which is also a work on capacitated controller placement. In [28], Yao et. al. uses flow algorithm to implement a dynamic solution which can work comfortably with changing data flows due to traffic or other reasons. Zhang et. al. [29] propose a solution which improves the resilience of a split architecture network. They improve the reliability of the network using the Min-cut algorithm [30] to find the fault tolerant and vulnerable parts of the network. Lange et. al [31] propose a solution named Pareto-based Optimal COntroller placement (POCO), which provides operators with Pareto optimal placements with respect to different performance metrics. POCO performs an exhaustive evaluation of all possible placements by default, which is feasible for small networks. For large networks POCO uses heuristics which makes it faster but less accurate. Liao et. al. [12] propose a faster algorithm named Density Based Controller Placement (DBCP). DBCP uses a threshold hop count to calculate the density of each switch in the network and places k controllers based on the density. DBCP outperforms the above-mentioned algorithms that work on UCPP, however, it uses hop counts to calculate the distances. If the goal is to create a programmable network that changes the flow path of data depending on network conditions (traffic, bandwidth etc.), then these parameters need to be considered and handled by the algorithm. Our proposed algorithms are based on SPICi (spicy, Speed and Performance In Clustering) [13], a fast clustering algorithm for biological networks, which divides a collection of proteins based on how closely they are related in terms of similarities [32]. The clusters that are formed contain closely connected proteins. SPICi clusters the network based on confidence values between two proteins and creates clusters that have maximum confidence values between them.

2.3 Problem Formulation

Networks distributed throughout the world are of different types and topologies. These networks can be represented as a bi-directional graph $G = (S, L)$ consisting of nodes and edges. Here the set of nodes S represent the switches and the set of edges L represent the links between the switches. The edges can be either weighted or unweighted based on the requirements. Our objective is to cluster the graph G into multiple sub-networks S_i such that each sub-network is a disjoint set of switches $\{s_1, s_2, ...\}$.

Let us assume that the network will be clustered into k partitions. The sub-networks can be presented as,

$$G \leftarrow \{S_1, S_2, ..., S_k\} \tag{2.1}$$

where, G is the entire network and $S_1, S_2, ..., S_k$ are k sub-networks.

2.4 Existing Algorithms

In this paper, we focus on two existing clustering algorithms. One of them is Density Based Controller Placement (DBCP) and the other is SPICi. DBCP [12] is a recently proposed algorithm for clustering Software Defined Networks (SDN). SPICi [13] is a well known protein-clustering algorithm for biological networks. These algorithms are described in the following sections.

2.4.1 Density Based Controller Placement (DBCP)

This algorithm is named Density Based Controller Placement (DBCP) because it uses local density to calculate all other parameters of the algorithm and then clusters the algorithm using those parameters. The pseudo-code for clustering using DBCP is given in algorithm 1.

DBCP uses the following equation to calculate the local density of each of the nodes in the network,

$$\rho_i = \sum_j \chi(d_{ij} - d_c) \tag{2.2}$$

where local density ρ_i of a node i is the count of all the nodes which are at most d_c distance away from i. The threshold d_c is a distance used to set a limit to the cluster diameter and consequently to find an approximate to the optimal value of k where k is the number of controllers. Here d_{ij} gives the minimum distance between nodes i and j. The value of $\chi(x)$ is 1 only for $d_{ij} < d_c$, i.e., when $x < 0$ and is 0 otherwise. Thus ρ_i is the number of nodes that can be reached from node i by traversing at most distance d_c.

The minimum distance of a node i to a higher density node is represented by,

$$\delta_i = \begin{cases} \max_{j:j\in S}(d_{ji}), & \text{if switch i has the highest } \rho_i \\ \min_{j:j\in S,\rho_j>\rho_i}(d_{ji}), & \text{otherwise} \end{cases} \tag{2.3}$$

where, δ_i is the minimum distance to a higher density node and S is the set of all switches in the network. If i is the node with highest density, δ_i is the distance of the farthest node from i. Consequently, an average of the minimum distances to higher density nodes, δ_i, is calculated for all nodes i, and denoted as δ. The value of k, the number of controllers is initialized at 0, and incremented whenever the value of δ_i of any node i is greater than δ. The switches with higher values of δ_i are selected as cluster heads of new clusters and the other switches are assigned to the nearest node with higher local density (ρ_i).

A cluster head of a network is a node from where the cluster formation initiates and a controller is a node which acts as the control plane. Each sub-network has its own controller which sends routing information to their respective data planes, and the data plane of every sub-network consists of its switches. DBCP uses the summation of three metrics to determine the controller for a cluster. These three metrics are $\pi^{avglatency}$, $\pi^{maxlatency}$, and $\pi^{inter\_controller}$. For a sub-network S_i, the average latency for a switch v is calculated as follows:

$$\pi^{avglatency}(v)_{v:v\in S_i} = \frac{1}{|S_i|}\sum_{s\in S_i} d(v,s) \tag{2.4}$$

Algorithm 1 Density Based Controller Placement (DBCP)

```
1: procedure DBCP(S,L)
2: Initialize $k := 0$
3: for $s$ in S:
4:     $\rho_s := \sum_{j \in S} \chi(d_{sj} - d_c)$
5: end for
6: for $s$ in S:
7:     $\delta_s := \min_{i: i \in S, p_i > p_s}(d_{is})$
8: end for
9: $\delta := \frac{1}{|S|} \sum_{s \in S} \delta_s$
10: for s in S:
11:     if $\delta_s > \delta$ then
12:         $k := k + 1$
13:         $s \leftarrow$ new cluster
14:     else
15:         $s \leftarrow$ cluster of nearest higher density
16: end for
```

This is the average of the distances of the node v from all other nodes in the cluster S_i, where, s is any other node of S_i. For the worst case scenario, the second metric is defined. This metric is denoted by $\pi^{maxlatency}$.

$$\pi^{maxlatency}(v)_{v:v \in S_i} = \max_{s \in S_i} d(v, s) \tag{2.5}$$

Here, $\pi^{maxlatency}(v)$ is the maximum of the distances of the node v from all other nodes s in the cluster S_i.

The inter-controller latency must be reduced as much as possible when selecting controllers. However, the controller-to-controller distance cannot be determined when the controllers are yet to be selected. Thus the third metric is used which calculates the distances from all other nodes that are not in the same cluster. It is an approximate calculation of the inter-controller distances and is denoted by $\pi^{inter\_controller}$.

$$\pi^{inter\_controller}(v)_{v:v \in S_i} = \frac{1}{|S - S_i|} \sum_{s \in (S - S_i)} d(v, s) \tag{2.6}$$

$\pi^{inter\_controller}$ for a node v is the average of the distances between v and all the nodes of other clusters.

The final metric $\pi^{latency}(v)$ can be calculated using the previously mentioned three metrics (equations 2.4, 2.5 and 2.6) as follows,

$$\begin{aligned}\pi^{latency}(v) &= \pi^{avglatency}(v) + \pi^{maxlatency}(v) + \pi^{inter\_controller}(v) \\ &= \frac{1}{|S_i|} \sum_{s \in S_i} d(v, s) + \max_{s \in S_i} d(v, s) + \frac{1}{|S - S_i|} \sum_{s \in \{S - S_i\}} d(v, s)\end{aligned} \tag{2.7}$$

Here $\pi^{latency(v)}$ is the sum of all the three values of $\pi^{avglatency}(v)$, $\pi^{maxlatency}(v)$ and $\pi^{inter\_controller}(v)$ for a switch v. Then in each cluster, the switch with the minimum value of $\pi^{latency}$ is taken as the controller of that cluster.

There are two steps of DBCP. The first step, which is, finding the value of k, requires calculating the distances between all possible pair of nodes. Although in [12] this has not been mentioned, to the best of our knowledge this can be done with complexity $O(V(V+E))$ using Dijkstra's algorithm to calculate all possible pair distances between nodes, where V is the number of nodes and E is the number of edges. As this distance is only calculated once so it can be considered as pre-calculated and does not need to be included in complexity analysis. Then calculating the value of ρ_i or local density for all nodes i has complexity $O(V^2)$ in the worst case when all other nodes are reachable by d_c number of hops. Calculating the value of δ_i for all nodes has the same complexity. Increasing and assigning controllers has a complexity of $O(V)$. If complexity is denoted by η then the complexities can be written as follows:

$$\begin{aligned}\eta_{dbcp} &= \eta_\rho + \eta_\delta + \eta_k \\ &= O(V^2) + O(V^2) + O(V) \\ &= O(V^2)\end{aligned} \tag{2.8}$$

2.4.2 Speed and Performance In Clustering (SPICi)

SPICi ('spicy', Speed and Performance In Clustering) clusters a connected undirected network $G = (V, E)$ with edges that have values of the continuous range $(0, 1)$. It is named so because it is an extremely fast algorithm for clustering biological networks. The values of the edges are called confidence values, denoted by $w_{u,v}$ for adjacent nodes u and v, and they represent similarities between two proteins [32]. The proteins are represented as nodes of the graph.

SPICi clusters a network using three variables. They are the weighted degree of a node, the density for a set of nodes and the support for a node with respect to a set of nodes. The weighted degree of a node u denoted by $d_w(u)_{u\in V}$ can be presented as:

$$d_w(u)_{u\in V} = \sum_{v:v\in V,(u,v)\in E} w_{u,v} \tag{2.9}$$

Here $d_w(u)_{u\in V}$ is the sum of all the confidence values of the edges that connect u with any other adjacent node v of the graph $G = (V, E)$. It is to be noted that only those nodes are considered which are still unclustered. The density of a set of nodes denoted by density(S), can be presented as:

$$density(S) = \frac{\sum_{(u,v)\in E} w_{u,v}}{|S| * (|S| - 1)/2} \tag{2.10}$$

In other words, *density*(S) is the sum of the confidence values of the edges that connect every node u with every other node v of the set of nodes S, divided by the number of total possible nodes that is $|S| * (|S| - 1)/2$ where $|S|$ is the number of nodes present in the set of nodes S. The support of a node u with respect to a set of nodes S can be presented as:

$$support(u, S) = \sum_{v \in S} w_{u,v} \tag{2.11}$$

For a network, *support*(u, S) is the sum of the confidence values of the edges that connect a node u with the nodes that are adjacent to it and are present in the set of nodes S. Using

Algorithm 2 : Speed and Performance In Clustering (SPICi)

```
 1: procedure SEARCH
 2: Initialize $DegreeQ := V$
 3: While $DegreeQ$ is not empty
 4:     Extract $u$ from $DegreeQ$ with the largest weighted degree
 5:     if $u$ has adjacent vertices in $DegreeQ$ then
 6:         1. Find from $u$s adjacent vertices the second seed protein v (see text)
 7:         2. $S := Expand(u, v)$
 8:     else
 9:         $S := \{u\}$
10:     $V := V - S$
11:     Delete all vertices in S from $DegreeQ$
12:     For each vertex t in $DegreeQ$ that is adjacent to a vertex in $S$, decrement its weighted degree by
    $support(t, S)$
13: procedure EXPAND(u,v)
14: Initialize the cluster $S := \{u, v\}$
15: Initialize $CandidateQ$ to contain vertices neighboring $u$ or $v$
16: While $CandidateQ$ is not empty
17:     Extract from $CandidateQ$ with the highest $support(t, S)$
18:     if $support(t, S) \geq T_s * |S| * density(S)$ and $density(S + t) > T_d$ then
19:         $S.add(t)$
20:         Increase the support for vertices connected to t in $CandidateQ$
21:         For all unclustered vertices adjacent to t, insert them into $CandidateQ$ if not already present
22:     else
23:         break from loop
24: return S
```

these parameters, SPICi clusters a network as given in the algorithm 2. The first seed is the node with the maximum weighted degree. After the selection of the first seed, the adjacent nodes are divided into five bins depending on the confidence value of the connecting edge. The bins are of ranges $(0 : 0.2 : 0.4 : 0.6 : 0.8 : 1)$, that is, they are at regular intervals of 0.2 in the already given range $(0, 1)$. Then starting from the maximum bin $(0.8 : 1)$, the node with the maximum weighted degree, d_w is taken as the second seed. SPICi uses two thresholds. These thresholds are: T_s which determines whether a node is to be included in the cluster based on the cluster size and the connectivity of the node to the cluster and: T_d which includes a node to the cluster, based on the density increased when the node is added. The algorithm gives better results when these thresholds have a value of 0.5 [13]. As SPICi is a clustering algorithm that works on biological networks it does not select any controller for any cluster. It only selects first seed and second seed and includes the nodes in each cluster. Therefore there is no Controller Selection step for SPICi.

SPICi can be divided into three phases or functions. The first function selects the first seed from a sorted queue which has a time complexity of $O(Vlog_2(V+E))$ in the worst case when all nodes are connected to all other nodes. The second function or second seed selection process requires $O(V)$ time complexity as all the weighted degrees are calculated in the first seed selection phase. The expand function calculates the support value for each node in the CandidateQ which is a sorted queue or priority queue. Therefore this phase requires a complexity of $O(Vlog_2(V+E))$. Let the complexity of the algorithm be denoted by η^{SPICi}. Then the complexities can be calculated in the following manner.

$$\begin{aligned}\eta^{SPICi} &= \eta^{f\_seed} + \eta^{s\_seed} + \eta^{expand} \\ &= Vlog_2(V+E) + O(V) + Vlog_2(V+E) \\ &= Vlog_2(V+E)\end{aligned} \tag{2.12}$$

2.5 Proposed Algorithms

We propose four algorithms to address CPP for both unweighted and weighted network. One of our proposed algorithms work on unweighted graphs and we name it Random Clustering with Local Search (RCLS). The remaining three are for weighted graphs and we name them Greedy-SPICi (G-SPICi), Inverse SPICi (I-SPICi) and Modified-SPICi (M-SPICi). We describe our proposed algorithm in the following subsections.

2.5.1 Random Clustering with Local Search (RCLS)

This algorithm is only for networks that have hop count as the distance metric, where all the edge weights of the graph are set to one. Therefore it has the same working conditions as DBCP. We choose k random cluster heads where the value of k is fixed. We assign every other node to the cluster of the nearest cluster head. We optimize the network further using local search. We perform the local search technique by including one randomly selected node in a randomly selected cluster in each iteration until the latency of the network cannot be decreased anymore (algorithm 3). The latency of the network $m_{latency}$, in this case, is calculated using the metric defined for evaluating DBCP (equation 2.17).

Algorithm 3 : Random Clustering with Local Search (RCLS)

```
 1: procedure RCLS(k, iteration)
 2: Randomly select $k$ cluster heads from the graph
 3: Include all switches to nearest cluster heads
 4: Calculate $m_{latency}$ (equation 2.17)
 5:     while $iterations > 0$ do
 6:         Local_Search(latency)
 7:         $iterations := iterations - 1$
 8: procedure Local_Search(latency)
 9:     while improvement do
10:         $a$ := any node from the graph
11:         $A$ := cluster of a
12:         $B$ = any cluster from the cluster set
13:         if already checked for the pair($a$,$B$) then
14:             continue.
15:         $A.remove(a)$
16:         $B.add(a)$
17:         set new controller heads
18:         if $new\ m_{latency} < m_{latency}$ then
19:             $m_{latency} := new\ m_{latency}$
20:             break
21:         else
22:             $B.remove(a)$
23:             $A.add(a)$
```

Initially, RCLS selects k controllers randomly and evaluates the current selection. In the following iterations, a randomly selected node a is inserted into a randomly selected cluster B. After insertion, the controllers are selected using the controller selection method of DBCP. After evaluation, if there is no improvement in terms of $m_{latency}$, the new configurations is rolled back. The process carries on until the maximum number of iteration is reached or until the value of $m_{latency}$ cannot be minimized any further. All the distances used in this algorithm are hop counts, which is the same as that of DBCP. However, this algorithm can also be applied to un-weighted graphs. In that case, the hop counts will be replaced by integer values which can represent network parameters like bandwidth, traffic, delay etc. and the value of $m_{latency}$ can be updated accordingly. The worst case time complexity is calculated as follows.

RCLS randomly selects k nodes as cluster heads, provided that k is given. This step has a complexity of k. Each node needs to be included in a cluster which requires a complexity of $k \times n$ if n is the total number of nodes. For the local search process, the algorithm randomly selects a cluster and randomly selects a node and puts the node in the cluster to check $m_{latency}$. The calculation of $m_{latency}$ has the complexity of n^3. In the worst case when a better solution does not exist, the algorithm calculates $m_{latency}$ for all possible pairs. This has a complexity of $k \times n$.

$$\begin{aligned}\eta^{RCLS} &= \eta^{cluster} + \eta^{local\_search} \\ &= O(k \times n) + O(k \times n \times n^3) \\ &= O(k \times n^4)\end{aligned} \tag{2.13}$$

2.5.2 Greedy-SPICi

Greedy-SPICi (G-SPICi) is a variation of SPICi, which does not divide the nodes connected to the first seed into bins as done in SPICi. Instead, after selecting the first seed, G-SPICi starts clustering the network indifferently, starting with the nodes adjacent to the first seed. The nodes are sorted based on their support value (equation 2.11) with respect to the entire network. Then the nodes are inserted into the cluster greedily based on the insertion condition (line 17 of algorithm 4). Therefore, we name the algorithm Greedy-SPICi (Algorithm 4).

G-SPICi starts with a node having the maximum weighted degree and then forms a cluster including it's neighboring nodes using an *EXPAND* function call (line 6). In the *EXPAND* (line 12) function, the nodes adjacent to the first seed are sorted based on a support value with respect to the present cluster being formed. For example, when the *EXPAND* function is called, the present cluster consists of only the first seed u (line 13). G-SPICi initializes a priority queue called *DegreeQ* that holds the degree of each node using equation 2.9. From *DegreeQ* the node with the highest weighted degree is extracted and the cluster is expanded

Algorithm 4 : Greedy-SPICi

```
1: procedure SEARCH(V,E)
2: Initialize $DegreeQ = V$
3: While $DegreeQ \neq empty$
4:     Extract u from DegreeQ with largest $d_w(u)$
5:     if there is $v \in DegreeQ$ such that $(v, u) \in E$ then
6:         $S := Expand(v)$
7:     else
8:         $S := \{u\}$
9:     $V := V - S$
10:    $DegreeQ := DegreeQ - S$
11:    For all $t \in DegreeQ$, do $d_w(t) := d_w(t) - support(t, S)$
12: procedure EXPAND(v)
13: Initialize cluster $S := \{u\}$
14: Initialize $CandidateQ := S$ such that $s \in S$ and $(s, u) \in E$
15: While $CandidateQ \neq empty$
16:     Extract $t$ from $CandidateQ$ with highest $support(t, S)$
17:     if $support(t, S) \geq T_s * |S| * density(S)$ and $density(S + t) > T_d$ then
18:         $S.add(t)$
19:         $CanditateQ.add(s)$ for all $(s, t) \in E$
20:         $CandidateQ.remove(t)$
21:     else
22:         break
23: return $S$
```

using *EXPAND* function. The nodes present in the formed cluster is then removed from *DegreeQ* and the support values of the rest of the nodes are updated accordingly. This process continues until there are no more nodes left in *DegreeQ*. The *EXPAND* function starts with forming the cluster S from first seed u. The nodes adjacent to the present cluster are the candidates of being included in the cluster they form the candidate list. A priority queue *CandidateQ* is formed from the support values of the members of the candidate list. Each member of the candidate list is then included in the cluster based on a conditional statement (line 17). If the node is included in the cluster then the node is removed from *CandidateQ* and its adjacent nodes are inserted into *CandidateQ* except the ones that are already there, and the support values of the nodes in *CandidateQ* are updated accordingly.

The controller selection process of G-SPICi is similar to DBCP except that it uses edge weights (positive integers) to calculate the three metrics $\pi^{avglatency}$, $\pi^{maxlatency}$ and $\pi^{inter\_controller}$, mentioned in section 2.4.1, instead of hop counts. As a result, the controllers are selected in such a way that the controller-to-switch and controller-to-controller latencies are minimized.

G-SPICi has the same complexities as SPICi, only the second seed selection is omitted and local search is added. If n is the total number of nodes and m is the total number of edges. Then the complexity of G-SPICi can be denoted by $\eta^{G-SPICi}$ where,

$$\begin{aligned} \eta^{G-SPICi} &= \eta^{f\_seed} + \eta^{expand} + \eta^{local\_search} \\ &= 2nlog_2(n+m) + O(k \times n^4) \\ &= O(k \times n^4) \end{aligned} \tag{2.14}$$

2.5.3 Inverse-SPICi

Inverse-SPICi (I-SPICi) is another variation of SPICi, which converts the edge weights of the network in such a way that the highest edge weight becomes the lowest edge weight and vice versa. It also omits the second seed selection process, like G-SPICi. SPICi forms clusters such that the connection among the nodes of the clusters are maximized where the edge weights are the similarity values. However, our goal is to minimize latency where the edge weights represent the latencies. This is why we invert the edge weights and name this algorithm Inverse-SPICi. If the weight of an edge is w and the maximum edge weight of the network is w_{max}, then the weight is inverted as follows:

$$w_{inverted} = w_{max} - w + 1$$

Here $w_{inverted}$ is the final edge weight. We subtract edge weight w from maximum edge weight w_{max}, and add 1 so that the maximum edge weight is not inverted to 0. The controller selection process is that of G-SPICi.

I-SPICi has the same complexities as G-SPICi, only the costs are inverted which requires a complexity of m. The total number of nodes is n and m is the total number of edges. The complexity of I-SPICi is denoted by $\eta^{I-SPICi}$ where,

$$\begin{aligned}\eta^{I-SPICi} &= \eta^{invert} + \eta^{G-SPICi} \\ &= O(m) + O(k \times n^4) \\ &= O(k \times n^4 + m)\end{aligned} \tag{2.15}$$

2.5.4 Modified-SPICi

Modified-SPICi or M-SPICi is similar to the original SPICi. It includes some more steps, like pre-processing and post-processing and thus is named Modified-SPICi. Pre-processing of M-SPICi is calculating the degree of incidence of the nodes. The degree of incidence of a node is the number of edges adjacent to the node. The post-processing step of M-SPICi assigns the isolated nodes to nearest clusters.

Initially, the degree of incidence (both outgoing and ingoing) of all the nodes are calculated. Then all of the nodes are divided into five partitions based on their degree of incidence from highest to lowest. The nodes in the first partition have the highest degree of incidence and consequently have the highest probability of becoming first seeds. We name this partition as 'head' partition. All of the edge costs w are changed to $1/w$. This causes the edge values which are positive integers, to be inverted and mapped to the continuous range of SPICi $(0, 1)$. After selecting the first (u) and second seeds (v), M-SPICi starts expanding the current cluster which is the set of nodes $S = \{u, v\}$. The rest of the unclustered nodes are kept in a priority queue, $CandidateQ$, based on their support values with respect to S. If a node does not meet the condition (line 26 of algorithm 5) for including in the cluster it is clear that the rest of the nodes in $CandidateQ$ will follow. However, these nodes are not yet discarded– for each node remaining, another check is performed. If the degree of a node is in the *head* partition then it is discarded as it has the potential to be cluster head. Otherwise, the node is included in the cluster, and all the nodes are clustered in this way. The isolated nodes are included in the clusters of the nearest cluster heads. Then the controllers are selected as done in G-SPICi.

The number of clusters is k, and the number of nodes and edges are n and m respectively. M-SPICi has an edge weight inversion step of complexity $O(m)$. Another pre-processing which calculates the degree of all nodes and sorts them, has a complexity of $O(m+nlog_2(n))$. The post-processing step for assigning isolated nodes is $O(n \times k)$ and the local search has a complexity of $O(k \times n^4)$. Therefore the total complexity is:

$$\begin{aligned}\eta^{M-SPICi} &= \eta^{SPICi} + \eta^{processings} + \eta^{local\_search} \\ &= nlog_2(n+m) + O(m) + O(n \times k) + O(m + nlog_2(n)) + O(k \times n^4) \\ &= O(m + nlog_2(n+m) + k \times n^4)\end{aligned} \tag{2.16}$$

Algorithm 5 : Modified-SPICi

```
1: procedure SEARCH(V,E)
2: Initialize $DegreeQ := V$
3: Perform pre-processing
4:     while do$DegreeQ \neq empty$
5:         Extract $u$ from $DegreeQ$ with largest $d_w(u)$
6:         if there is $v \in DegreeQ$ such that $(v, u) \in E$ then
7:             $v := SecondSeed(u)$
8:             $S := Expand(u, v)$
9:         else
10:            $S := \{u\}$
11:        $DegreeQ := DegreeQ - S$
12:        For all $t \in DegreeQ$, do $d_w(t) = d_w(t) - support(t, S)$
13: Perform post-processing
14: procedure SECONDSEED(u)
15: $bin$ = adjacent nodes of $u$
16: Divide $bin$ into five equal parts based on connected edge weight (0 $to$ 1)
17:     for $max(bin)$ to $min(bin)$ do
18:         if $bin \neq empty$ then
19:             find $max(d_w(v))$ for $v \in bin$
20:             return $v$
21: procedure EXPAND(u,v)
22: Initialize cluster $S := \{u, v\}$
23: Initialize $CandidateQ := S$ such that $s \in S$ and $(s, u), (s, v) \in E$
24: While $CandidateQ \neq empty$
25:     Extract $t$ from $CandidateQ$ with the highest $support(t, S)$
26:     if $support(t, S) \geq T_s * |S| * density(S)$ and $density(S + t) > T_d$ then
27:         $S.add(t)$
28:         $CanditateQ.add(s)$ for all $(s, t) \in E$
29:         $CandidateQ.remove(t)$
30:     else
31:         break
32: return $S$
```

Table 2.1: Randomized Networks used as Input

Scenario	Nodes	Edges	Edge/Node
1	40	52	1.3
2	50	68	1.36
3	60	77	1.2833
4	70	87	1.2429
5	80	108	1.35
6	90	120	1.3333
7	100	131	1.31

2.5.5 Scenario Specific Explanation

In figure 2.2 we present one of the graphical representations of a network that we considered for our experimentation. It is an actual network that expands throughout different states of USA as well as Canada, and are used for research purposes. We assign random weights to the links between the switches and then apply our proposed algorithms. For example, M-SPICi clusters the network into 9 sub-networks. The clusters are marked in figure 2.3 in nine different colors, each representing a different cluster. The controllers of the sub-networks are the nodes marked as 2, 28, 21, 6, 27, 8, 12, 16 and 24. Therefore, each cluster has a single controller and no two clusters can have any common node.

2.6 Performance Evaluation

2.6.1 Simulation Environment

We perform all our experiments using C++ (High-level language). We use seven randomly generated networks with different numbers of switches starting from 40 to 100, at regular intervals of 10. The edge to node ratio is kept from 1.1 to 1.4 to keep the networks considerably sparse (similar to current worldwide networks). The seven scenarios are shown in the table 2.1. The graphs do not contain any self-loops or multi-edges but may have cycles. We have simulated a number of scenarios but for convenience, we present seven scenarios in table 2.1.

2.6.2 Performance Metric

In Software Defined Network, each time a new packet arrives, the switch asks the controller for routing decisions. The controller decides the path of the packet and sends the routing information to all the switches in the path. If the switches are of different clusters then the information is sent to all of the controllers of those clusters. In [12], J. Liao et. al. use this

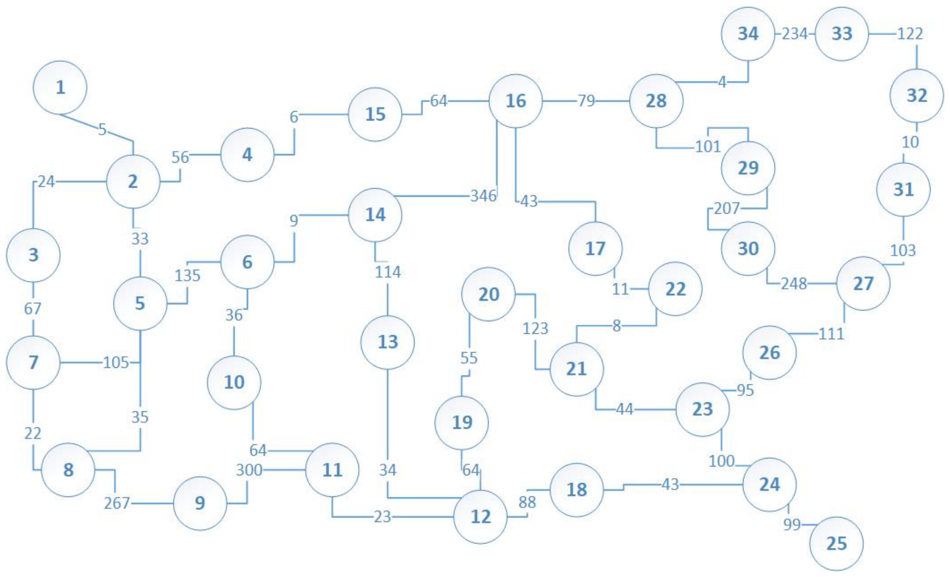

Figure 2.2: The Internet2 OS3E Network with 34 nodes and 42 edges expanding over Canada and USA used mainly for research purposes

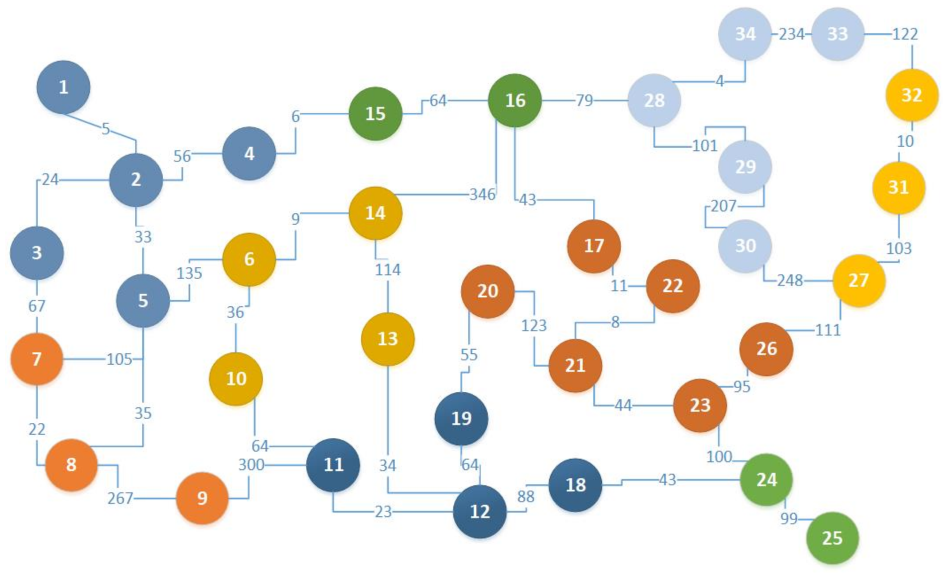

Figure 2.3: The result of clustering by M-SPICi

process to define a latency for a network denoted by $m_{latency}$.

$$m_{latency} = \frac{1}{|S|(|S|-1)} \sum_{s_i, s_j \in S, i \neq j} \{d(s_i, v_i) + \max_{s_m \in Path_{i,j}} (d(v_i, v_m) + d(v_m, s_m))\} \tag{2.17}$$

Here, $|S|$ is the number of switches in the network, s_i and s_j are any two switches in that network and v_i is the controllers of the switch s_i. The term $Path_{i,j}$ means the series of connected nodes that are in between nodes s_i and s_j. s_m is any node in the $Path_{i,j}$ and v_m is the controller of node s_m. Thus the latency for a pair of switches is the sum of the distances between s_i and the corresponding controller v_i, $d(v_i, v_m)$, and the maximum of the sum of the distances between v_i and v_m and between v_m and s_m, $max(d(v_i, v_m) + d(v_m, s_m))$. In theory, this is the maximum distance that a packet needs to traverse to set up a new route from node s_i to s_j. There are a total of $|S|(|S|-1)$ possible pairs of nodes possible. Therefore the average of the previously defined latency for all possible pairs is the latency of the network.

2.6.3 Result Comparison

We propose algorithms for both weighted and un-weighted networks. We compare our first proposed algorithm RCLS with DBCP as both work with unweighted graphs. We also compare DBCP with a local search version of DBCP to verify that there is further room for improvement. For weighted networks where the edge values are randomly assigned, we evaluate and compare the algorithms G-SPICi, I-SPICi, and M-SPICi with the well-known algorithm DBCP. As DBCP is designed for unweighted graphs, we implement a weighted version of DBCP and compare our proposed algorithm with that. In the simulation graphs, weighted implementation of DBCP is presented as W-DBCP. Therefore result comparisons are divided into un-weighted and weighted comparison. The networks are the same for both cases except for un-weighted graphs, the edge weights are set to one. The networks used for experimenting are given in Table 2.1.

Result Comparison for Un-weighted Graph

We propose RCLS for un-weighted graphs. In Fig. 2.4 we compare our proposed algorithm RCLS with existing algorithm DBCP. Simulation results suggest that RCLS outperforms DBCP in terms of $m_{latency}$ for the same value of k. Furthermore, we applied the local search technique on DBCP. In Fig. 2.4 we represented it as DBCP+LOCAL. RCLS also outperforms DBCP+LOCAL in terms of $m_{latency}$.

We observe from Fig. 2.4 that only for the first scenario (Table 2.1) where the number of nodes is 40, RCLS gives greater latency than DBCP and DBCP+LOCAL. This is because RCLS performs better for large-scale networks and for the first scenario the number of nodes

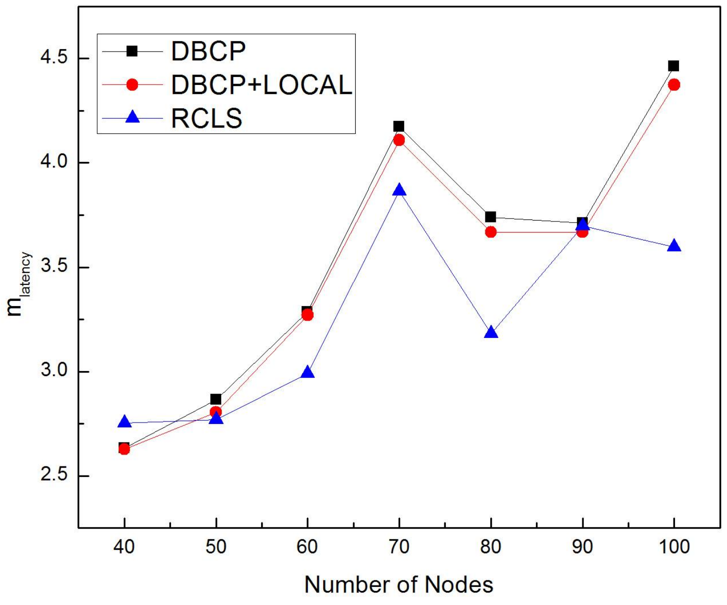

Figure 2.4: Comparison of DBCP, DBCP with local search and RCLS on un-weighted networks using $m^{latency}$ as the performance metric.

is only 40. RCLS, DBCP, and DBCP+LOCAL give almost equal latencies for scenario 2 and 6 where the network is denser (they have a higher edge/node ratio). In all other scenarios, RCLS outperforms both DBCP and DBCP+LOCAL as RCLS works better for sparser networks.

Result Comparison for Weighted Graph

We propose three algorithms for weighted graphs which are G-SPICi, I-SPICi, and M-SPICi. RCLS can also be implemented on weighted graph. We compare our proposed algorithms with weighted-DBCP. In Fig. 2.5 we represent weighted-DBCP as WDBCP, which considers edge weights instead of hop count.

In Fig. 2.5 we can see that WDBCP gives better results than our proposed algorithms in terms of $m_{latency}$, but from the Fig. 2.6, we can see that WDBCP gives a very high number of clusters than that of our proposed algorithms. This is acceptable when the cost of installing a controller is trivial, but in reality, it is not feasible to accommodate the installment cost of so many controllers. Therefore, we compare our proposed algorithms with WDBCP in terms of $k \times m_{latency}$ in Fig. 2.8 for different scenarios, to take into consideration k, the number of controllers. If we consider the cost of each controller as constant c, then the cost of controller installment for a network is $k \times c$. So the cost-latency product of a network is $c \times k \times m_{latency}$. The variables k and $m_{latency}$ vary from scenario to scenario. However, we assume the cost of

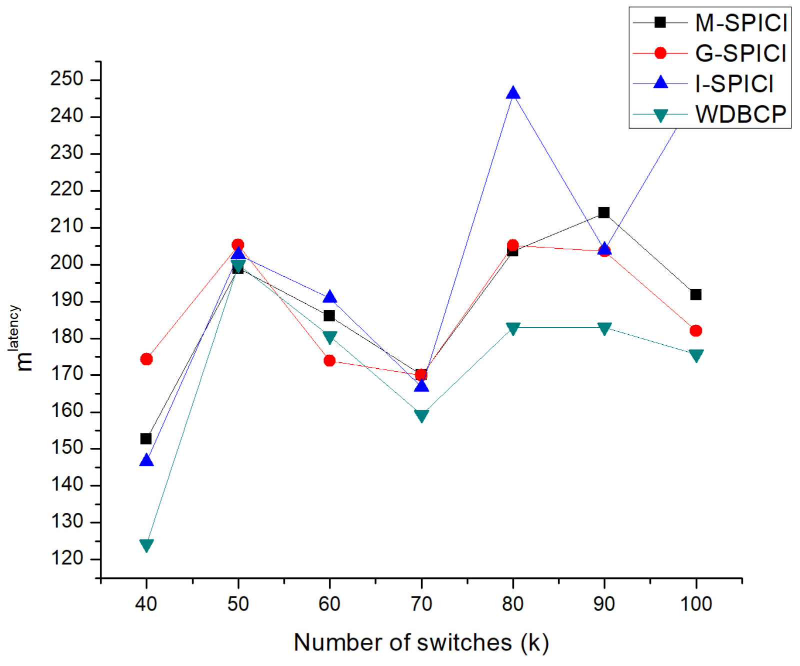

Figure 2.5: Comparison of WDBCP, G-SPICi, I-SPICi, and M-SPICi on weighted graphs using $m^{latency}$ as performance metric, where WDBCP is the implementation of DBCP using edge weights.

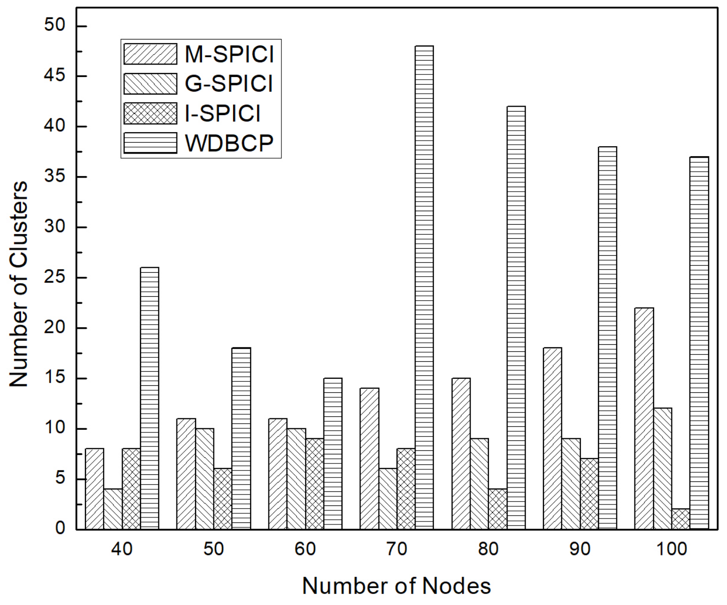

Figure 2.6: Comparison of the number of controllers k, for algorithms WDBCP, G-SPICi, I-SPICi, and M-SPICi.

installing a controller to be constant. There the cost-latency product can be simplified and represented as $k \times m_{latency}$. From Fig. 2.8 we can see that, I-SPICi and G-SPICi give better results for all of the scenarios of table 2.1, and our proposed algorithms outperform DBCP in terms of cost-latency product. For scenario 3 when the number of nodes is 60, I-SPICi and G-SPICi give the same result. For scenarios 1 and 4, where nodes are 40 and 70 respectively, G-SPICi outperforms other algorithms. For other scenarios, I-SPICi outperforms all other algorithms.

2.6.4 Analysis on the number of controllers k

The number of clusters or the value of k is different for each algorithm. One of the three problems of CPP is - *How many controllers?*. The value of $m_{latency}$ decreases with the increment of k. Therefore the best simulation result is obtained when $k = |S|$, where $|S|$ is the number of switches in the network. When all the switches are controllers, the latency of the network is minimum. However, this is not feasible as the cost of installing so many controllers is much more than required.

In Fig. 2.7 we have presented the results of RCLS on weighted graphs of different number of switches. For each network graph, the value of k is increased from 1 to $|S|$, where S is the total numbers of switches in the network. As discussed, the value of $m_{latency}$ decreases with increasing k. We have to select a k so that the latency of the network and the cost of installing the controllers is minimized. We have to select k such that increasing the value of k does negligible improvement compared to previous increments of k. We need to determine a threshold of improvement, although this might be different in different scenarios and depends on the need of the network operator. Therefore our proposed algorithm RCLS will cluster the network optimally based on the value of k given by the network operator.

2.6.5 Complexity Comparison

We calculated the complexities of all the existing algorithms and proposed algorithms for clustering Software Defined Networks (SDN) in their respective sections (section 2.4 and 3.3). We compare these algorithms in terms of complexity. The existing algorithm is DBCP and the proposed algorithms are RCLS, G-SPICi, I-SPICi and M-SPICi.

We can observe from table 2.2 that DBCP is faster than our proposed algorithms. However, this slight increase in complexity is negligible compared to the improvement of clustering in terms of latency ($m_{latency}$) and cost-latency product ($k \times m_{latency}$) and the advantages that it can offer.

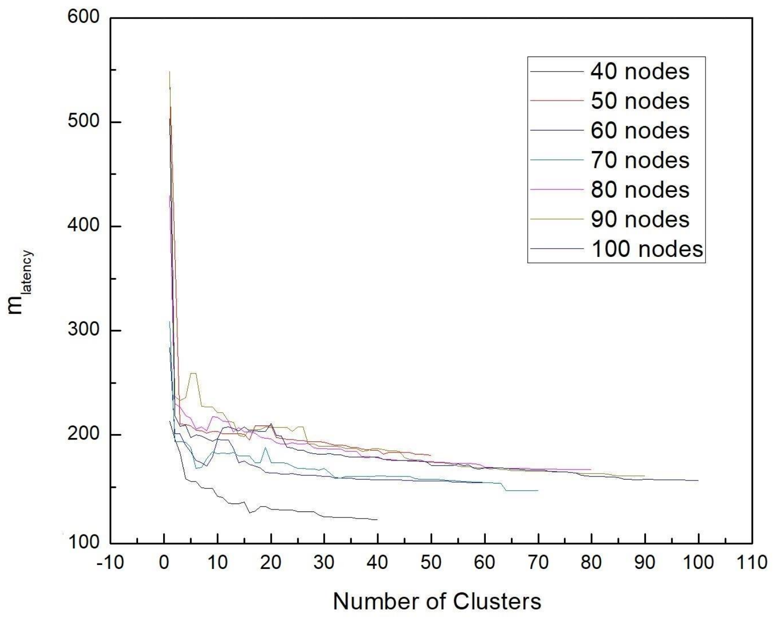

Figure 2.7: Comparison of network latency $m_{latency}$ for different numbers of clusters (k) for RCLS on weighted graphs of each scenario.

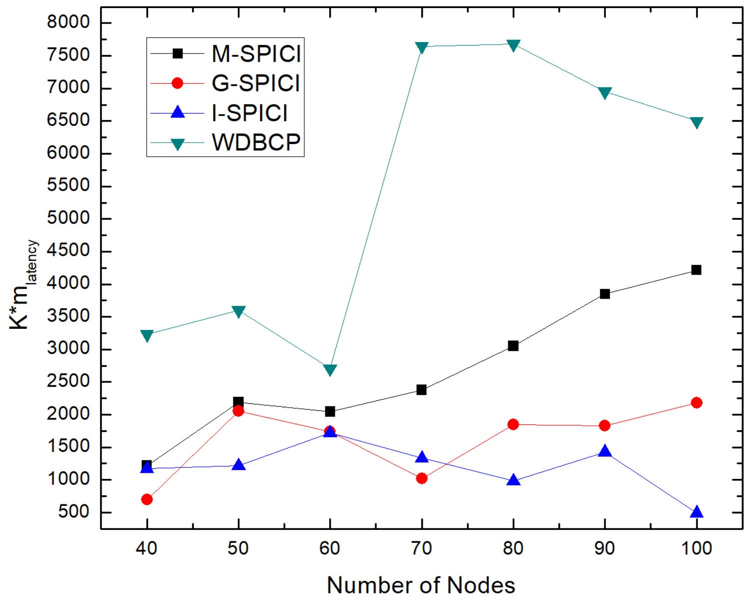

Figure 2.8: Comparison of DBCP, G-SPICi, M-SPICi, I-SPICi, and RCLS on weighted networks using cost-latency product ($k \times m^{latency}$) as the performance metric.

Table 2.2: The complexity of different algorithms

Algorithm	Complexity
DBCP	$O(V^2)$
DBCP + LOCAL	$O(k \times n^4)$
RCLS	$O(k \times n^4)$
G-SPICi	$O(k \times n^4)$
I-SPICi	$O(k \times n^4 + m)$
M-SPICi	$O(m + nlog_2(n + m) \times n^4)$

2.6.6 Traffic-Awareness

Traffic awareness is of two types– Static traffic awareness and Dynamic traffic awareness. Static traffic awareness is attained when a network considers traffic while clustering a network before placing controllers. However, dynamic traffic awareness can cluster a network even after placing controllers based on the changing traffic. Our algorithms perform a controller selection process to minimize the distance (or latency) of the overall network (using equation 3.4). More importantly at every step of the local search, the total latency of the network, $m_{latency}$ is improved by changing the controller placement and assigning the switches depending on the edge-weights which represent traffic. Thus, our algorithms are static traffic aware. After placing the controllers, the switches can be reassigned using the local search technique we used. The controller selection steps only need to be omitted from local search techniques of our proposed algorithms (line 17 of algorithm 3). In this case, controller selection is not done in each step of the local search technique, and only the switches are reassigned. Therefore, our algorithms are also Dynamic traffic-aware.

2.7 Summary

In this chapter, we presented the problem of controller placement and a solution to the weighted variant of this problem to minimize network latency and improve network throughput. We also compared our algorithms with the recent algorithm DBCP in terms of latency and improve both dynamic and traffic awareness in polynomial time complexity.

Chapter 3

Latency Minimization with Load Balancing

3.1 Introduction

3.1.1 Controller Placement Problem (CPP)

Software Defined Networks (SDNs) decouple the traditional protocol stack into control plane and data plane. The control plane consists of controllers which take the routing decisions and relay this information to the data plane. The data plane is the collection of switches which forward the data according to the routing decisions provided by the control plane. The initial design of SDN included a single controller in the control plane [4]. However, even for moderate-sized networks, a bottleneck is created due to heavy traffic concentrated at the controller. Furthermore, problems like scalability and reliability surface as network size increases [15, 16]. In order to deal with these problems, many researches propose multiple controllers [18, 19, 20], which results in the emergence of the **controller placement problem (CPP)**. The CPP deals with placing a minimal number of controllers in the SDNs, with an aim to address the above mentioned problems.

3.1.2 Related Works

A solution to the CPP is to select an optimal number of controllers, to place them in the best possible locations, and to assign them switches in an optimal way [22, 23]. In addition, there are multiple constraints that need to be satisfied– minimizing the latency among switches and controllers, maximizing reliability and resilience, and minimizing deployment cost as well as energy consumption, which results in an NP-Hard problem. Despite the complications, in recent years, several solutions have been proposed to address the CPP [21, 11, 33, 7, 2], where some of them deal with optimizing a particular constraint like latency or reliability, while some others introduce a compound metric to address two or more constraints.

Heller et. al [24] calculate the average and worst case latencies to place controllers and mention that multiple controllers is more significant in ensuring fault-tolerance than to minimize latency. Yao et. al. [28] propose dynamic scheduling strategies based on changing flows to manage controllers aiming to balance their loads. Hu et. al. [34] propose multiple algorithms for reliability-aware controller placement. Zhang et. al. [29] use a min-cut based graph partitioning algorithm to maximize resilience of SDNs and compare the algorithm with greedy approaches. In [31], Lange et. al. propose POCO (Pareto-based Optimal COntroller placement), which represents the CPP as a combinatorial optimization problem. POCO exhaustively iterates all possible combinations as viable solutions to optimize multiple metrics and uses heuristics-based PSA (Pareto Simulated Annealing) to improve the computational complexity at the cost of accuracy. Sallahi et. al. [25] propose a mathematical model for optimal controller placement considering the cost of installing controllers, load of the controllers and path set up latency. Exhaustive solutions like [31, 25] can give optimal solutions, however, their computational complexities are very high.

3.1.3 Contributions and Organization

Liao et. al. [12] propose two faster algorithms Density Based Controller Placement (DBCP) and Capacitated-DBCP (CDBCP) where, DBCP minimizes overall latency based on the local density of the nodes and CDBCP balances the load of the controllers. However, they work separately on the given network to perform controller placement and to balance the load of the controllers. To the best of our knowledge, no other methods minimize overall latency and perform load balancing simultaneously, while clustering SDNs in polynomial time complexity. In this paper, we place minimum (optimal) number of controllers in a SDN to minimize the followings:

1. **Flow setup latency:** When a switch receives a packet for which no path exists, the flow-setup procedure is initiated. The latency introduced by this procedure is the maximum delay required to set the path (and also to inform the intermediate switches) for the flow.
2. **Route synchronization latency:** When there is a change in the network, routes are changed and all the controllers have to be updated (in sync) about the changes. This latency deals with controller-to-controller latency.
3. **Load of a controller:** The volume of control traffic and processing that the switches impose on the controllers.

This requires placing the controllers in a way that minimizes both the average controller-to-switch and average controller-to-controller latencies and limits the load of the controllers.

In this paper, we propose and develop an algorithm named Degree-based Balanced Clustering (DBC), to achieve the above with a careful selection of controllers. Simulation results suggest that our proposed algorithm DBC outperforms the state-of-the-art algorithms in terms of overall latency and load balancing. We also observe that, the above criteria can be achieved if we can divide the SDN into k balanced clusters.

The rest of the paper is organized as follows– our problem formulation along with some assumptions are provided in Chapter 3.2, a detailed description of our proposed mechanism is given in Chapter 3.3, simulation results and performance evaluation are presented in Chapter 3.4 and we conclude in Chapter 4.

3.2 System Model and Assumptions

We consider the network as a bi-directional graph $G = (S, L)$, where, the set of nodes S represent the switches and the set of edges L represent the links between the switches. The edges can be either weighted or unweighted based on the requirements. We cluster the graph G into multiple sub-networks such that, each sub-network is a disjoint set of switches. There cannot be any common switch between two sub-networks and all of the switches of the network must fall into a sub-network, where each sub-network will have one and only one controller. Accordingly, if we assume that the network is partitioned into k sub-networks, then there will be k controllers and each sub-network will be a disjoint set of switches containing a single controller. For simplicity we make the following four assumptions:

1. The transmission delay of each control packet is identical and negligible.
2. The propagation delay, queuing delay and processing delay of each switch is identical.
3. The load imposed by each switch is fixed.
4. The controller can only replace a switch, and cannot be placed in any other locations.

Under these assumptions, our objective is to minimize the flow-setup latency, the inter-controller distances and the controller-to-switch distances in each sub-network, all of which are measured in hop counts. We also aim to balance the load of each controller, which is the number of switches assigned to each controller.

3.3 Proposed Mechanism

We propose a solution to the CPP referred to as degree-based balanced clustering (DBC) for SDN. The proposed mechanism divides the networks into k-clusters, assigns a controller to each cluster, and finally, finds the minimum value of k for the network which obtains the minimum latency. In the sequel, we explain the mechanism in detail.

3.3.1 Cluster Formation

The cluster formation mechanism divides the network into k clusters (sub-networks), when k is given. It finds the k cluster heads first and then, assigns each of the nodes to exactly one sub-network. The clustering mechanism deals with two goals–

- Equal load on the controllers, which requires equal number of nodes in each cluster. We assume that the switches have equal load.
- Minimum intra-cluster distances between the nodes and cluster heads, which requires direct (or shortest path) connectivity of the nodes and the cluster heads.

The first goal demands more clusters in the denser part of the network, whereas, the second goal demands the nodes with maximum direct connections with the switches to be the cluster heads. Accordingly, a cluster head should have the highest connectivity among all the switches of its cluster so that maximum number of nodes can be reached with minimum delay. Both the two goals can be achieved if the clustering is done based on degree of the node.

A degree based solution to the second goal suggests selecting k nodes with highest degrees as the cluster heads of the network. This solution ensures minimization of intra-cluster delay when the highest degree nodes are uniformly distributed (well separated). However, in typical networks, the higher degree nodes are situated in the same locality of the network. Consequently, the dense regions of the network are dominated by most of the cluster heads, and the clusters formed are not balanced in terms of load (first goal). Therefore, the cluster heads need to be a certain distance apart from each other, which we define as the minimum inter cluster head distance, T_d. The threshold, T_d, simultaneously maintains a fixed cluster head separation and balances the load of the network.

The cluster head separation dictates the number of switches in a cluster, which should be roughly $|S|/k$ for a balanced cluster. Initially, for $T_d = 1$ hop count, the nodes in the cluster are the cluster head itself and one-hop neighbors of the cluster head. Consequently, in an average case scenario, the total number of nodes is $1 + AvgDeg$, where $AvgDeg$ denotes the average node degree of the network. According to graph theory, each link simultaneously increments the degree of two switches by one, which implies that, the summation of the node degrees of a network is $2 \times |L|$ and the average node degree is $\frac{2\times|L|}{|S|}$.

We denote the total number of nodes in the periphery of the cluster as *boundary*. Initially, $boundary = AvgDeg$, as the cluster consists of a cluster head at the center and its one-hop neighbors on the border which is equivalent to $AvgDeg$ nodes on average. Each boundary node is again connected to approximately $AvgDeg$ number of nodes including the cluster head itself. Therefore, omitting the cluster head, for every increment of the value of T_d, the number of nodes in the cluster increases following the given formula,

$$boundary = boundary \times (AvgDeg - 1) \tag{3.1}$$

When the total number of nodes in the cluster exceeds $|S|/k$ nodes, we terminate the increment process. The value of T_d thus obtained is the threshold distance or cluster separation of a balanced cluster.

In DBC, we sort the nodes according to descending value of node degree (line 1 of algorithm 6) and select the node with highest degree as first cluster head. We select a cluster head from the remaining nodes, only if it is at least T_d distance away from the selected cluster heads, otherwise we omit it, and move on to the next node in the sorted list. The region of the selected node may be sparser than other parts of the network. In order to take this into account, we multiply the threshold T_d with a degree ratio (line 13 of algorithm 6), which is the maximum degree of the network divided by the degree of the selected node. The degree ratio ranges the T_d value according to the density of the node, if the degree of the node is less, the density is lower. Accordingly, the ratio has a higher value ($\frac{Maximum\ Degree}{degree\ of\ selected\ node} > 1$) and increases the cluster separation threshold to include more nodes than in a denser region. However, if k cluster heads cannot be selected in this way, we select the remaining cluster heads only based on degree and ignore their distance from other cluster heads. Consequently, all the remaining nodes are assigned to the cluster of the nearest cluster head and k clusters are formed.

3.3.2 Controller Selection

After completion of the clustering process we initiate a controller selection process. The total delay of a network depends on both controller-to-controller latency and controller-to-switch latency. The controller-to-switch latency of a network improves when the switches are in close proximity of the controller and the controller-to-controller latency depends on the latencies of the paths between controllers. Although the cluster head has high intra cluster connectivity, inter cluster distance also needs to be taken into account. Considering both intra cluster and inter cluster distances (which refers to controller-to-controller latency), the cluster may or may not be the controller. Accordingly, we define $\tau(s)$ to be the controller selection function of switch s, which comprises its intra-cluster distance $\phi(s)$ and inter-cluster distance $\sigma(s)$ values. The intra-cluster distance can be calculated as the average distance from the switch s of cluster S_i to the other nodes of S_i [12],

$$\phi(s) = \frac{1}{|S_i| - 1} \sum_{u \in S_i} dis(s, u) \tag{3.2}$$

here, u represents any other switch in the same cluster S_i, as switch s and $|S_i|$ is the total number of switches in S_i. The component $\phi(s)$ corresponds to the controller-to-switch latency and $dis(s, u)$ is the shortest path distance of s from u.

The inter-cluster distance, which is the other component of total latency τ corresponds to the controller-to-controller latency. However, we cannot calculate inter-controller distances before the controller selection process as there are no controllers to begin with. Hence, we calculate $\sigma(s)$ for a switch s of cluster S_i, as the average distance of s from all other nodes of the network that are not in S_i [12],

$$\sigma(s) = \frac{1}{|S - S_i|} \sum_{v \in (S - S_i)} dis(s, v) \tag{3.3}$$

here, v denotes the switches that belong to network S but does not belong to sub-network S_i. The set of all nodes except sub-network S_i, is denoted by $S - S_i$.

The controller selection function $\tau(s)$ can be defined as the direct summation of the component latencies $\phi(s)$ and $\sigma(s)$. Although, this assigns equal weight to the inter-cluster latency and intra-cluster latency, the inter-cluster latency dominates overall latency as controller-to-controller distance is usually greater than controller-to-switch distance. We introduce a weight variable α to control this dominance and give the user the freedom to emphasize on any component as per requirement,

$$\tau(s) = \alpha \times \phi(s) + (1 - \alpha) \times \sigma(s) \tag{3.4}$$

In our experiments, we set the values of α as 0.5. As a result, both components contribute equally in the controller selection process. The pseudocode of our proposed method is given below (algorithm 6). The algorithm takes the network switches, S, links, L, and number of clusters, k, as input, and returns k optimal network clusters $C_1, C_2, ..., C_k$ and the set of $Controllers$ for the clusters.

3.3.3 Optimum k Selection

We evaluate the performance of each clustering using its average flow setup latency and denote this evaluation function by $\Omega(S)$ for a network S. When a data packet arrives at a switch s_i, for which no entry exists in its flow table, the switch forwards the packet encapsulated in a query message to its controller c_i. The controller decides the path for the packet and notifies the other switches of the path about the new flow rule, through their controllers. The switches on the path from s_i to destination s_d, may have different controllers. Consequently, the generated control packet containing the new route, needs to be forwarded to the controllers, which in turn will instruct the concerned switches to update their flow tables. Since the process is parallel, the path setup latency is dominated by the maximum of the sum of the distances from the corresponding controller of the source c_i, to controller c_j of any intermediate node s_j, and from c_j to s_j. Therefore, the evaluation function can be calculated

Algorithm 6 : Degree-based Balanced Clustering (DBC)

```
procedure DBC
input: $k, S, L$
Sort $S$ in descending order of degree
Initialize $AvgDeg := \frac{2\times|L|}{|S|}$ and $boundary := AvgDeg$
Initialize $limit := 1 + AvgDeg$
Initialize threshold distance $T_d := 0$
Initialize $clusterHeads := \emptyset$
    while $limit < (|S|/k)$ do
        $boundary = boundary \times (AvgDeg - 1)$
        $limit := limit + boundary$; $T_d := T_d + 1$
    for each switch s in $S$ do
        $adjT_d = T_d \times \frac{max\ degree}{degree\ of\ s}$
        if $clusterHeads = \emptyset$ then
            $clusterHeads.add(s)$
        else if $clusterHeads.size = k$ then break
        else if $dis(s, clusterHeads) < adjT_d$ then
            continue
        else
            $clusterHeads.add(s)$
    if $clusterHeads.size < k$ then
        Select remaining $clusterHeads$ with max degree
Initialize clusters $C_1, C_2, ..., C_k$ as $\emptyset$
Initialize $Controllers := \emptyset$
    for each cluster $C_i$ do
        $C_i := clusterHeads[i]$
        $C_i := C_i \cup \{nearest\ nodes\ of\ C_i\}$
    for each cluster $C_i$ do
        for all $s \in C_i$ do
            Calculate $\phi(s)$ and $\sigma(s)$
            $\tau(s) := \alpha \times \phi(s) + (1 - \alpha) \times \sigma(s)$
        $Controllers.add(node\ with\ min(\tau(s)))$
output: $Controllers$
output: $C_1, C_2, ..., C_k$
```

as the average setup latency of all possible switch pairs [12],

$$\begin{aligned}\Omega(S) =& \frac{2}{|S| \times |S-1|} \sum_{s_i, s_d \in S} \{dis(s_i, c_i) + \\ & \max_{s_j \in path_{i,d}} (dis(c_i, c_j) + dis(c_j, s_j))\}\end{aligned} \tag{3.5}$$

here, $path_{i,d}$ is the shortest path from source s_i to destination s_d. We use this function as the performance evaluation metric in section 3.4.

We increment k and apply DBC repeatedly until the improvement in terms of $\Omega(S)$ is negligible. We define the improvement ratio or termination criteria ξ as follows,

$$\gamma = \frac{(\Omega_{old\ k}(S) - \Omega_{new\ k})/\Omega_{old\ k}}{new\ k - old\ k} \tag{3.6}$$

here, *old k* is the previous value of k and *new k* is the incremented value. In our simulations we notice a sharp drop in the value of γ after a certain number of increments.

3.4 Performance Evaluation

3.4.1 Simulation Environment

We developed a simulation environment using C++ (High-level language) which takes the number of nodes and connections between each node as input and clusters the network. We use randomly generated networks with different numbers of switches starting from 40 to 100, at regular intervals of 10. There are 10 different datasets for each interval, giving a total of 70 different networks. The link to switch ratio is from 1.3 to 1.45 to keep the networks considerably sparse (similar to current worldwide networks). The simulation environment and selected 70 networks are same for both our algorithm and DBCP. The networks do not contain any self-loops or multiple links between two switches but may have cycles. The average number of edges of the scenarios are 54, 66.7, 81.4, 99.3, 111.2, 125.5 and 138.9 respectively.

3.4.2 Performance Metrics

We use various performance metrics to study and evaluate different characteristics of SDNs like inter-controller latency, controller-to-switch latency, and overall network switch to switch latency. We calculate the average inter-controller latency as follows,

$$\eta^{interController} = \frac{1}{k \times (k-1)} \sum_{1 \le i,j \le k, i \ne j} dis(c_i, c_j) \tag{3.7}$$

where, c_i and c_j are controllers of the SDN. We evaluate the controller-to-switch latency using the following equation,

$$\eta^{controllerSwitch} = \frac{1}{k}\sum_{i=1}^{k}\left[\frac{1}{|S_i - 1|}\sum_{u\in S_i} dis(c_i, u)\right] \tag{3.8}$$

here, S_i is the i^{th} sub-network, c_i is the controller and u, the switch of S_i. Both $\eta^{controllerSwitch}$ and $\eta^{interController}$ consider the shortest distance from the controller to any other controller or switch. However, for setup of a new route, the switch sends a packet to the controller, which decides the new route and relays the information to other concerned controllers and switches. We calculate this latency using the equation 3.5 which has been discussed in section 3.3.3.

We also evaluate the load of each controller, considering that all switches have the same load. For an ideal clustering of an SDN, the number of nodes per cluster should be $|S|/k$. However, the actual clustering may vary from the ideal case, which can be considered as the deviation. Accordingly, the least deviation from the ideal case is the most balanced in terms of load. The deviation can be denoted by,

$$\eta^{loadDeviation} = \frac{1}{k}\max_{i=1}^{k}(||S_i| - |S|/k|) \tag{3.9}$$

where, $|S_i|$ is the number of nodes in the i^{th} sub-network and $|S|$ is the total number of nodes in the network.

3.4.3 Simulation Results

We validate our proposed method through extensive simulations. We compare our proposed algorithm with DBCP [12] in terms of flow-setup latency, average inter-controller latency, average controller-to-switch latency and load balancing. To compare flow-setup latency, we use the value of k provided by DBCP to cluster the network and place the controllers. We plot the flow setup latencies as the average of the 10 instances in Fig. 3.1. We calculate the setup latency of each instance using equation 3.5 of section 3.3.3.

Fig. 3.1 shows that DBC outperforms DBCP, however, for smaller networks the difference in performance is less (when number of switches is 40). In all other cases, the difference in performance is noticeable. We can observe that, for both the algorithms, the latency values (Ω) increase with the number of switches in the network. However, when the number of nodes is 90, the latency for both of the algorithms decrease slightly due to inconsistency of underlying topology.

In Fig. 3.2, we plot the flow-setup latency with respect to controller number k for networks containing 60, 70, and 80 switches. We observe that the latency value, Ω, decreases at a greater rate for smaller number of controllers than for higher values of k. We use this property to define an optimum k selection function (equation 3.6), using which we terminate the process when the improvement ratio, γ, is negligible (meaning negligible improvement).

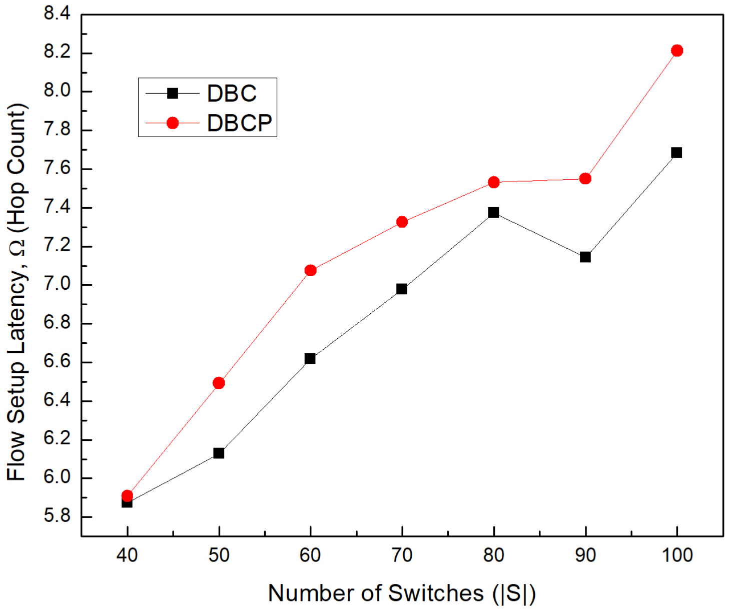

Figure 3.1: Comparison of flow-setup latency between DBC and DBCP using the value of k provided by DBCP

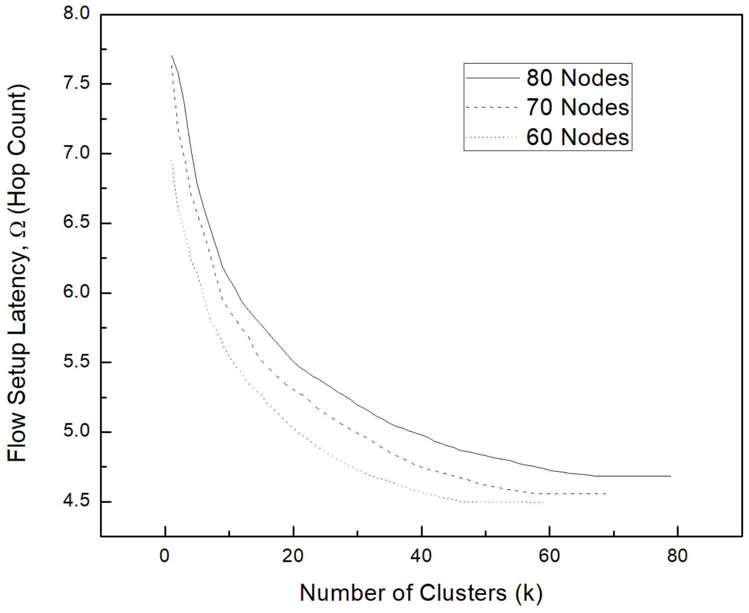

Figure 3.2: Flow-setup latency with respect to increasing cluster numbers (k) for networks containing 60, 70, and 80 nodes

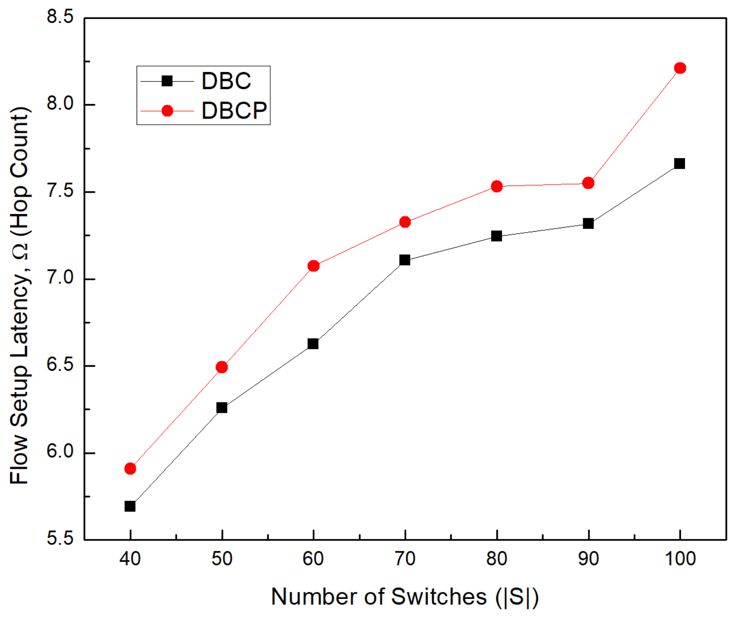

Figure 3.3: Comparison between DBC and DBCP in terms of flow-latency, Ω

We notice a sharp drop in improvement rate at higher values of k where $\gamma > 0.05$, so we set the terminating threshold at 0.05. Accordingly, we plot the flow-setup latencies for the new values of k in Fig. 3.3, where, we observe that our algorithm is more consistent than previously selected k values which were selected by DBCP. The flow-setup latencies have also improved, and the number of controllers have decreased (Fig. 3.4). In all the scenarios, DBC outperforms DBCP in terms of both flow-setup latency and minimum number of controllers.

We control the contribution of inter-controller latency and intra-cluster latency using a weight variable α (equation 3.4) which ranges from 0.0 to 1.0. When the value of α is increased, the intra-cluster latencies (equation 3.8) increase and the inter-controller latencies (equation 3.7) decrease as presented in Fig. 3.5. When the value of α is set to 0.5, inter-controller latencies are given priority by default, as the inter-controller distances are greater than intra-cluster distances. Our proposed algorithm outperforms DBCP in terms of inter-controller latency, however, DBCP gives better controller-to-switch latencies. We prioritize controller-to-controller latencies as control packets are essential in setting up new paths and conveying broken link information.

In Fig. 3.6, we compare the average, maximum and minimum loads of the controllers for each scenario. We assume that each switch imposes the same load on a controller. Accordingly, the number of switches per cluster corresponds to the load of each controller. We observe that the deviation from the ideal load per controller in a balanced SDN ($|S|/k$), which can be calculated using equation 3.9, is higher for DBCP. Our proposed algorithm

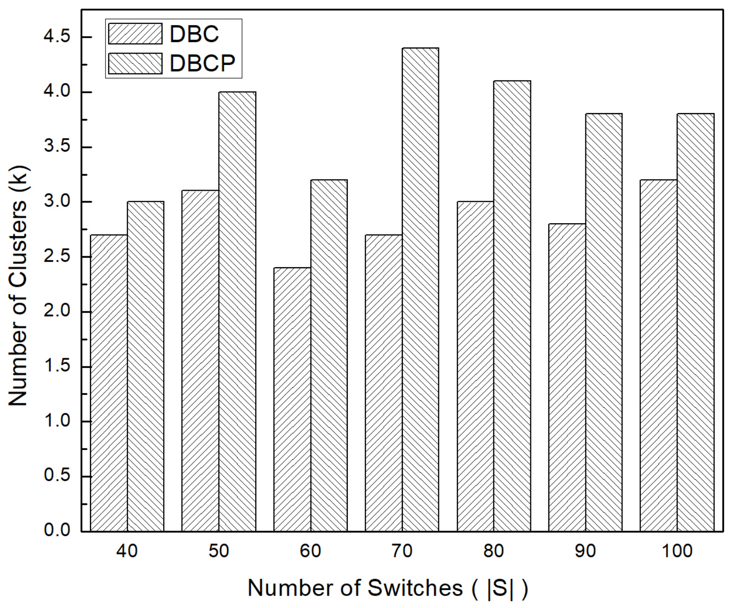

Figure 3.4: Number of controllers (k) given by DBCP and DBC for different scenarios

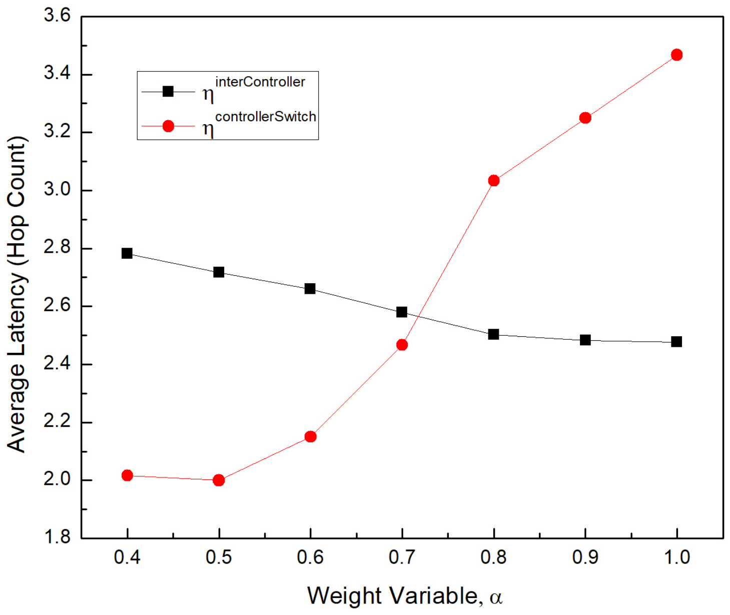

Figure 3.5: Average controller-switch and controller-controller latency with respect to increasing weight variable, α

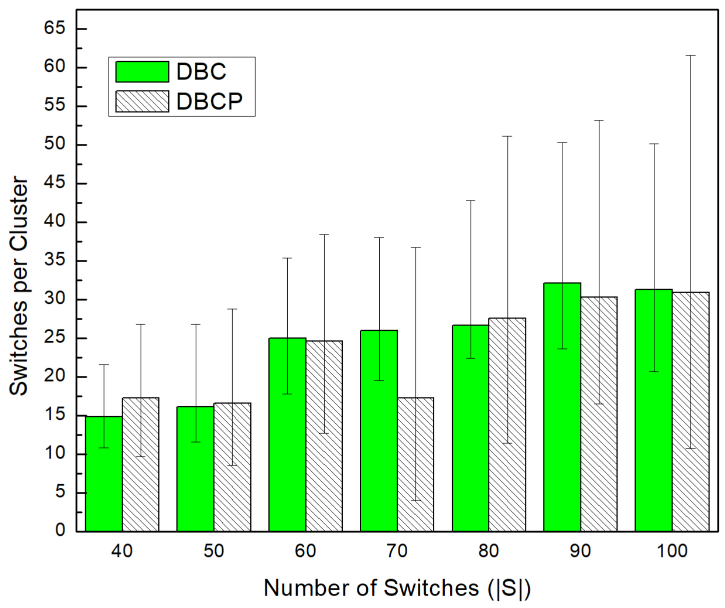

Figure 3.6: Comparison between DBCP and DBC in terms of average load per controller.

outperforms by forming clusters which are more balanced in terms of load per controller.

3.5 Summary

In this chapter, we showed that latency minimization and load balancing can be done simultaneously and proposed a new algorithm DBC which outperforms the existing algorithm DBCP in terms of flow-setup latency. DBC has polynomial time complexity and can adapt to accommodate the underlying network topology to create balanced clusters.

Chapter 4

Conclusion

4.1 Research Summary

In this paper, we address the Controller Placement Problem (CPP) of SDN. We have investigated one renowned algorithm DBCP [12] that addresses the same research problem. However, DBCP is for unweighted graphs where it uses hop count as the distance metric. Our proposed algorithm RCLS outperforms DBCP in terms of latency for unweighted graphs. However, an unweighted graph is not a good representation of a real network. Being inspired by another famous protein clustering algorithm SPICi [13] we propose 3 algorithms for weighted graphs. We validate our proposed algorithms through extensive simulations. The simulation results suggest that our proposed algorithms outperform the weighted variant of the existing DBCP algorithm in terms of cost and latency. One major contribution of our proposed algorithms is traffic awareness and also they have polynomial time complexity.

We also propose a novel clustering algorithm named Degree-based Balanced Clustering (DBC). Our proposed algorithm, DBC outperforms DBCP in terms of different latencies and balances the load of the controllers. We have shown that our algorithm has many advantages over other algorithms. DBC minimizes flow-setup latency and route synchronization latency through minimization of controller-to-switch and controller-to-controller distances. DBC creates balanced clusters with similar number of nodes and has polynomial time complexity.

4.2 Future Work

DBC can be extended to work on weighted networks which consider ongoing traffic of links and different delays like queuing delay in a congested network to intelligently select flow routes. The hop count can be replaced with average edge weights and the inter cluster separation updated accordingly. Future work can also include variable loads of switches and balance the cluster so that minimum load is imposed on each controller.

Bibliography

[1] A Behrouz Forouzan. *Data communications & networking (sie).* Tata McGraw-Hill Education, 2006.

[2] Keshav Sood, Shui Yu, and Yong Xiang. Software-defined wireless networking opportunities and challenges for internet-of-things: A review. *IEEE Internet of Things Journal*, 3(4):453–463, 2016.

[3] Paul Goransson and Chuck Black. *"1.2 Historical Background" in Software Defined Networks: A Comprehensive Approach.* Elsevier, 2014.

[4] Kate Greene. Tr10: Software-defined networking. *Technology Review (MIT)*, 2009.

[5] Sridhar KN Rao. Sdn and its use-cases-nv and nfv. *Network*, 2:H6, 2014.

[6] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[7] Jacob H Cox, Joaquin Chung, Sean Donovan, Jared Ivey, Russell J Clark, George Riley, and Henry L Owen. Advancing software-defined networks: A survey. *IEEE Access*, 5:25487–25526, 2017.

[8] Paul Goransson and Chuck Black. *"1.6 Can We Increase the Packet-Forwarding IQ" in Software Defined Networks: A Comprehensive Approach.* Elsevier, 2014.

[9] Behrouz A Forouzan and Sophia Chung Fegan. *TCP/IP protocol suite.* McGraw-Hill Higher Education, 2002.

[10] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206, 2014.

[11] Ashutosh Kumar Singh and Shashank Srivastava. A survey and classification of controller placement problem in sdn. *International Journal of Network Management*, 28:e2018, 2018.

[12] Jianxin Liao, Haifeng Sun, Jingyu Wang, Qi Qi, Kai Li, and Tonghong Li. Density cluster based approach for controller placement problem in large-scale software defined networkings. *Computer Networks*, 112:24–35, 2017.

[13] Peng Jiang and Mona Singh. Spici: a fast clustering algorithm for large biological networks. *Bioinformatics*, 26(8):1105–1111, 2010.

[14] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[15] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Towards an elastic distributed sdn controller. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 7–12. ACM, 2013.

[16] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, 2013.

[17] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.

[18] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 19–19, 2010.

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[20] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[21] Yuan Zhang, Lin Cui, Wei Wang, and Yuxiang Zhang. A survey on software defined networking with multiple controllers. *Journal of Network and Computer Applications*, 103:101–118, 2017.

[22] Reaz Ahmed and Raouf Boutaba. Design considerations for managing wide area software defined networks. *IEEE Communications Magazine*, 52(7):116–123, 2014.

[23] Stanislav Lange, Steffen Gebert, Joachim Spoerhase, Piotr Rygielski, Thomas Zinner, Samuel Kounev, and Phuoc Tran-Gia. Specialized heuristics for the controller placement problem in large scale sdn networks. In *Teletraffic Congress (ITC 27), 2015 27th International*, pages 210–218, Ghent, Belgium, 2015. IEEE.

[24] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12, NY, USA, Aug 2012. ACM.

[25] Afrim Sallahi and Marc St-Hilaire. Optimal model for the controller placement problem in software defined networks. *IEEE communications letters*, 19(1):30–33, 2015.

[26] Guang Yao, Jun Bi, Yuliang Li, and Luyi Guo. On the capacitated controller placement problem in software defined networks. *IEEE Communications Letters*, 18(8):1339–1342, 2014.

[27] F Aykut Özsoy and Mustafa Ç Pınar. An exact algorithm for the capacitated vertex p-center problem. *Computers & Operations Research*, 33(5):1420–1436, 2006.

[28] Long Yao, Peilin Hong, Wen Zhang, Jianfei Li, and Dan Ni. Controller placement and flow based dynamic management problem towards sdn. In *Communication Workshop (ICCW), 2015 IEEE International Conference on*, pages 363–368, London, UK, Jun 2015. IEEE.

[29] Ying Zhang, Neda Beheshti, and Mallik Tatipamula. On resilience of split-architecture networks. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–6, Kathmandu, Nepal, Dec 2011. IEEE.

[30] Thomas Erlebach, Alexander Hall, Linda Moonen, Alessandro Panconesi, Frits Spieksma, and Danica Vukadinović. Robustness of the internet at the topology and routing level. In *Dependable Systems: Software, Computing, Networks*, pages 260–274. Springer, 2006.

[31] Stanislav Lange, Steffen Gebert, Thomas Zinner, Phuoc Tran-Gia, David Hock, Michael Jarschel, and Marco Hoffmann. Heuristic approaches to the controller placement problem in large scale sdn networks. *IEEE Transactions on Network and Service Management*, 12(1):4–17, 2015.

[32] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814, 2005.

[33] Kushan Sudheera, Maode Ma, and P.H.J. Chong. Controller placement optimization in hierarchical distributed software defined vehicular networks. 135:225–239, Apr 2018.

[34] Yannan Hu, Wang Wendong, Xiangyang Gong, Xirong Que, and Cheng Shiduan. Reliability-aware controller placement for software-defined networks. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 672–675, Ghent, Belgium, May 2013. IEEE.