Efficient Controller Placement Algorithms for Software Defined Networks

Talha Ibn Aziz *, Shadman Protik *, Md Sakhawat Hossen * Salimur Choudhury†,Muhammad Mahbub Alam *
* Department of Computer Science and Engineering, IUT, Dhaka, Bangladesh
†Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada
Email: *talhaibnaziz@iut-dhaka.edu, *shadmanprotik@iut-dhaka.edu, *sakhawat@iut-dhaka.edu,
†salimur.choudhury@lakeheadu.ca, *mma@iut-dhaka.edu

***Abstract*—The controller placement problem (CPP), in software defined networks (SDNs), deals with placing an optimal number of controllers. The placement of controllers aims to maximize the network throughput and minimize the latencies. In this paper, we propose two controller placement algorithms, namely random clustering with local search and greedy clustering algorithm (GCA). We also propose two extended algorithms of GCA. Simulation results show that our proposed algorithms outperform existing state-of-the-art SDN clustering algorithms in polynomial time complexity. They can also be adjusted to meet changing clustering requirements.**

***Index Terms*—CPP, SDN, latencies, clustering**

I. Introduction

Software Defined Networks (SDNs) decouples the traditional protocol stack into the control plane and the 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 only a single controller in the control plane. However, for moderate-sized networks, a bottleneck is created due to heavy traffic concentrated at the controller. Furthermore, problems like scalability [1], [2] and reliability surface as network size increases. Researchers propose multiple controllers as a viable solution [3], [4], [5], to tackle these problems. There are various constraints that need to be met– minimizing the latency among switches and controllers, maximizing reliability and resilience and minimizing deployment cost and energy consumption. A solution to the CPP is to select an optimal number of controllers, placing them in the best locations possible, and assigning the switches to them in an optimal way [6], [7]. In addition, there are more than one constraints, which is why it is NP-Hard.

In recent years, several solutions have been proposed to address the CPP [8], [9]– some solutions minimize latency [10] or control traffic [11], some maximize reliability [12], [13] and resilience [14], while some optimize a composite metric [15], [16]. One such solution, Density Based Controller Placement (DBCP) [17], clusters the network to minimize latency and maximize reliability using hop count as the distance metric. However, hop count is not adequate to represent the overall condition of a network. Therefore, our proposed methods cluster a network to optimize a variable which can be set to any parameter– traffic, bandwidth, latency etc. The contribution of our work can be summarized as follows:

- We propose three 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.

The rest of the paper is organized as follows– We represent the background and related work along with some other notable works of SDN and CPP in Section II. Section III represents CPP as a graph theory problem. Section IV explains the existing SDN clustering algorithm DBCP and the protein-clustering algorithm SPICi. We explain our proposed algorithms in Section V. Simulation results and Performance Evaluation are presented in Section VI and we conclude in Section VII.

II. Background and Related Work

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 [18], [19], [20].

Intelligent routing started from an idea in 1960s [21], [22]. In the traditional network model [23], the routing decisions are taken by network layer routers using various protocols to populate the routing table like RIP (Routing Information Protocol), OSPF (Open Shortest Path First) and BGP (Border Gateway Protocol) [24]. In SDN however, the decisions are taken by a central entity called "controller" which keeps track of the changing traffic of the network and uses this knowledge to route traffic intelligently. The controller is the control plane and the switches are the data plane of the network layer (e.g., OpenFlow [25], [26]) where the controller tells the switches where to send a new packet. In this architecture, all the switches ask a single controller for routing decisions. This is manageable for small size and medium-size networks, however, for large-scale networks, a bottle-neck is formed. The network becomes more congested in the course of time

and eventually collapses. Thus multiple controllers become a necessity.

Initially, the control plane of a SDN consisted of only one controller. This architecture has already been implemented extensively: the controller OpenDayLight (ODL) [27] 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 [20]. However, the concept of only one controller is feasible for a small or even medium-sized networks, but when the network becomes larger it becomes very difficult for one controller to handle the routing. The problems that are faced in the case of single controllers are scalability [1], [2] and reliability when the networks are large. Consequently, multiple controllers were proposed, and the foundation was laid by R. Sherwood et. al. [28] in 2009. Thereafter multiple controllers have been used in several applications [3], [4], [5] and a lot of research have been directed towards it [8], [9]. The questions that arise due to multiple controllers are: *How many controllers? Where to place them?* and *Which switch is assigned to which controller?* [6], [7]. Answering all of these questions together is called the Controller Placement Problem or CPP in short, which is an emerging paradigm and a field of vibrant research in the domain of SDN. In the current decade, many solutions have been proposed to solve this NP-Hard problem of controller placement [10], [8], [9]. Heller et. al. [10] propose a solution of CPP by selecting k controllers to minimize the average and maximum latency of the network. Sallahi et. al. [16] provide a mathematical model which simultaneously determines the controller numbers, locations and included switches while meeting some constraints. An example of this constraint could be the cost of installing a controller. Yao et. al. [29] introduce capacitated controller placement. They consider the load or capacity of a controller and the load of the switches, thus the capacity of a controller cannot be exceeded. Ozsoy et. al. [30] propose an advanced version of the k-center algorithm using links between switches which is also a work on capacitated controller placement. In [11], 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. There are also notable research works on other parameters like reliability. Zhang et. al. [14] propose a solution which improves the resilience of a split architecture network. They improve the reliability of the network using the Min-cut algorithm [13] to find the fault tolerant and vulnerable parts of the network. There are also heuristic-based approaches to solve the CPP problem. Lange et. al [15] propose a well-known algorithm named Pareto-based Optimal COntroller placement (POCO). POCO 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. [17] 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 then defines a distance for each switch which is the minimum distance to a higher density node and takes an average of these distances. If for a node or switch the minimum distance to higher density node is greater than the average distance then another controller is needed. In this way, DBCP increments the value of k from zero, which is the number of controllers, to a suitable value. Then keeping the switches which have a higher than average value of their minimum distance to higher density node as cluster heads, DBCP assigns the other switches to the clusters of their nearest higher density switch. DBCP is a state-of-the-art-algorithm which outperforms the above-mentioned algorithms that work on UCPP, but the distance metric used is hop count. 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 the well-known algorithm SPICi (spicy, Speed and Performance In Clustering) [31], 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.

III. Problem Formulation

Networks are distributed throughout the whole world. Each of them is of different types and topologies depending on the requirement. In SDN only the controllers take the routing decisions. Therefore, to demonstrate clustering, all the nodes are considered as switches (routers are replaced by switches). We consider the network 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 contains a set of switches $S_i = \{s_1, s_2, ...\}$, where s_1, s_2 etc. are 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.

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

$$S_{net} \leftarrow \{S_1 = \{s_{1,1}, s_{1,2}, ...s_{1,i}\}, S_2, ..., S_k\} \quad (1)$$

where, S_{net} is the total network and $S_1, S_2, ..., S_k$ are k disjoint sets of switches (sub-networks).

IV. Existing Algorithm

In this paper, we focus on DBCP [17], which is a recently proposed algorithm for clustering Software Defined Networks (SDN). 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 network using those parameters.

DBCP uses the following equation to calculate the local density of the i_{th} node, ρ_i, in the network,

$$\rho_i = \sum_j \chi(d_{ij} - d_c) \tag{2}$$

where ρ_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 $d_{ij} < d_c$ that is 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,

$$\delta_i = \begin{cases} \max_{j:j \in S}(d_{ji}), & \text{if switch i has the highest } \rho_i \\ \min_{j:j \in S, \rho_j > \rho_i}(d_{ji}), & \text{otherwise} \end{cases} \tag{3}$$

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 holds the distance of the farthest node from i. Then an average of the minimum distance to higher density node δ_i is calculated for all nodes i and denoted as δ. The value of k, the number of controllers is initialized as 0. If the value of δ_i of a node i is greater than δ_i, then k is incremented. 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.

DBCP uses the summation of three metrics to determine the controller for a cluster. The cluster head of a network is a node from where the cluster formation starts. A controller is a node which acts as the control plane. Each sub-network has its own controller. These controllers send routing information to their respective data planes. The data plane of every sub-network consists of its switches. 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:

$$\pi^{avglatency}(v)_{v:v \in S_i} = \frac{1}{|S_i|} \sum_{s \in S_i} d(v, s) \tag{4}$$

This is the average of the distances of the node v from all other nodes. It is the sum of all distances between s and v of a cluster S_i divided by the total number of switches in the cluster $|S_i|$. For the worst case scenario, the second metric is defined. This metric is denoted by $\pi^{maxlatency}$.

$$\pi^{maxlatency}(v)_{v:v \in S_i} = \max_{s \in S_i} d(v, s) \tag{5}$$

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 $\pi^{inter\_controller}$.

$$\pi^{inter\_controller}(v)_{v:v \in S_i} = \frac{1}{|S - S_i|} \sum_{s \in (S - S_i)} d(v, s) \tag{6}$$

$\pi^{inter\_controller}$ for a node v is the sum of the distances between v and all the nodes of other clusters. This sum is then divided by the total number of switches in the network that are in other clusters. The final metric $\pi^{latency}(v)$ can be presented using three metrics in equations 4, 5 and 6 by,

$$\begin{aligned} \pi^{latency}(v) &= \pi^{avglatency}(v) + \pi^{maxlatency}(v) \\ &\quad + \pi^{inter\_controller}(v) \\ &= \frac{1}{|S_i|} \sum_{s \in S_i} d(v, s) + \max_{s \in S_i} d(v, s) \\ &\quad + \frac{1}{|S - S_i|} \sum_{s \in \{S - S_i\}} d(v, s) \end{aligned} \tag{7}$$

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.

V. Proposed Algorithms

We propose four algorithms to address the CPP for both unweighted and weighted network. All of the algorithms are explained in their respective subsections

A. Proposed Algorithm for Unweighted Graphs

We propose one algorithm for unweighted graphs and we name it Random Clustering with Local Search (RCLS) and we explain RCLS in the following subsection.

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 chose k random cluster heads where the value of k is fixed. We assign every other node to the cluster whose cluster head has the shortest distance from that node. Then we optimize the network 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 1). The latency of the network $m_{latency}$, in this case, is calculated using the metric defined for evaluating DBCP (equation 11).

In the first iteration, RCLS selects k controllers randomly and evaluates the current selection. In the next iteration, 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. The network thus formed is the again evaluated similarly using $m_{latency}$. If the value is lower than the previous configuration then the change is made permanent. On the other hand, if the value is higher, the new configurations is rolled back to its previous configuration. Then a new a and b are chosen and this process carries on until the maximum number of iteration is reached

Algorithm 1 : Random Clustering with Local Search (RCLS)

```
1: procedure RCLS(k)
2: Select k cluster heads randomly
3: Assign every node to nearest cluster head
4: Select Controllers
5: Calculate m_latency
6:     while improvement do
7:         for each node a and cluster B do
8:             Include node a in cluster B
9:             Select new controllers
10:            Calculate new m_latency
11:            if new m_latency < m_latency then
12:                Swap node a to cluster B
13: Return Current Clustering
```

or until the value of $m_{latency}$ does not decrease anymore. 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.

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 $k \times n$. Therefore, RCLS has a complexity of $O(k \times n^4)$.

B. Proposed Algorithms for Weighted Graphs

We propose three algorithms for weighted graphs and we name them Greedy Clustering Algorithm (GCA), Inverse SPICi (IGCA) and Probabilistic Greedy Clustering Algorithm (PGCA). All of the algorithms proposed for weighted graphs cluster a network using three variables namely the weighted degree variable, the density variable, and the support variable. The weighted degree of a node u is denoted by $d_w(u)_{u \in V}$ and can be presented as:

$$d_w(u)_{u \in V} = \sum_{v:v \in V,(u,v) \in E} w_{u,v} \tag{8}$$

Here $d_w(u)_{u \in V}$ is the sum of all the edge weights 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 is denoted by density(S), and can be presented as:

$$density(S) = \frac{\sum_{(u,v) \in E} w_{u,v}}{|S| * (|S| - 1)/2} \tag{9}$$

In other words, $density(S)$ is the sum of the edge weights connecting all possible pairs of nodes of 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 denoted by:

$$support(u, S) = \sum_{v \in S} w_{u,v} \tag{10}$$

For a network, $support(u, S)$ is the sum of the edge weights that connect a node u with the nodes that are adjacent to it, and are present in the set of nodes S.

1) Greedy Clustering Algorithm: Greedy Clustering Algorithm (GCA) is a variation of SPICi. GCA does not divide the nodes connected to the first seed into bins as done in SPICi. Instead, after selecting the first seed, GCA starts clustering the network indifferently, starting with the nodes adjacent to the first seed. The nodes are sorted based on their support value (equation 10) with respect to the whole network. Then the nodes are inserted into the cluster greedily based on the insertion condition (line 17 of algorithm 2). Therefore, we name the algorithm Greedy Clustering Algorithm (Algorithm 2).

Algorithm 2 : Greedy Clustering Algorithm

```
1: procedure SEARCH(V,E)
2: Initialize DegreeQ = V
3: While DegreeQ ≠ empty
4:     Extract u from DegreeQ with largest d_w(u)
5:     if there is v_{v:v,u∈E} ∈ DegreeQ then
6:         S ← Expand(v)
7:     else
8:         S ← {u}
9:     V ← V − S
10:    DegreeQ ← DegreeQ − S
11:    d_w(t)_{t:t∈DegreeQ,(t,s)_{s∈S}∈E} = d_w(t) − support(t, S)
12: procedure EXPAND(V,E,u)
13: Initialize the cluster S ← {u}
14: Initialize CandidateQ = S_{S:s∈S,(s,u)∈E}
15: While CandidateQ ≠ empty
16:    Extract t from Candidate with the highest
       support(t, S)
17:    if support(t, S) ≥ T_s * |S| * density(S) and
       density(S + t) > T_d then
18:        S ← S + {t}
19:        CanditateQ ← CandidateQ + {s_{s:(s,t)∈E}}
20:        CandidateQ ← CandidateQ −
       {s_{s:s∉CandidateQ}}
21:    else
22:        break from loop
23: return S
```

GCA starts with a node having the maximum weighted degree and then forms a cluster including it's neighboring nodes using an *EXPAND* function (line 12 of algorithm 2). In the *EXPAND* 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 first

seed u (line 8 of algorithm 2). GCA initializes a priority queue called $DegreeQ$ that holds the degree of each node using equation 8. From $DegreeQ$ the node with the highest weighted degree is extracted and the cluster is expanded 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 of algorithm 2). 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 GCA is same as that of 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 V, instead of hop counts. As a result, the controllers are selected such that the controller-to-switch and controller-to-controller latencies are minimized where the latencies are integer values.

GCA 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 GCA is $O(k \times n^4)$.

2) Inverse Greedy Clustering Algorithm: Inverse Greedy Clustering Algorithm (IGCA) is a variation of GCA. IGCA converts the edge weights of the network such that the highest edge weight becomes the lowest edge weight and vice versa. 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 Greedy Clustering Algorithm. However, the controller selection process is unchanged.

IGCA inverts the edge weights. If the weight of an edge is w and the maximum edge weight of the whole 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.

IGCA has the same complexities as GCA, 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 IGCA is $O(k \times n^4 + m)$.

3) Probabilistic Greedy Clustering Algorithm: Probabilistic Greedy Clustering Algorithm or PGCA is similar to the original SPICi. It includes some more steps, like pre-processing and post-processing and thus is named Probabilistic Greedy Clustering Algorithm. Pre-processing of PGCA 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. In case of directed graphs, the number of outgoing edges is called out-degree and ingoing edges are called in-degree. The post-processing step of PGCA assigns the isolated nodes to nearest clusters.

Initially, the degree of incidence (both outgoing and incoming) 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. Thus the nodes in the first partition have the highest degree of incidence and have the highest probability of becoming first seeds. We name this partition as seed 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 $(0, 1)$.

Algorithm 3 : Probabilistic Greedy Clustering Algorithm

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 $v_{v:v,u\in E} \in DegreeQ$ **then**
6: $v \leftarrow secondseed(DegreeQ, E, u)$
7: **if** $v \neq null$ **then** $S \leftarrow Expand(u, v)$
8: **else**
9: $S \leftarrow \{u\}$
10: $V \leftarrow V - S$
11: $DegreeQ \leftarrow DegreeQ - S$
12: $d_w(t)_{t:t\in DegreeQ,(t,s)_{s\in S}\in E} = d_w(t) - support(t, S)$
13: **procedure** SECONDSEED(V,E,u)
14: $bin[i]_{i:i=(1,5)} \leftarrow s_{s:s\in V,(s,u)\in E}$
15: **for i = 1 to binSize**
16: **if** $bin[i] \neq empty$ **then**
17: **return** v **if** $d_w(v) = \max_{s:s\in bin[i]} d_w(s)$
18: **return** $null$
19: **procedure** EXPAND(V,E,u,v)
20: **Initialize** the cluster $S \leftarrow \{u, v\}$
21: **Initialize** $CandidateQ = S_{S:s\in S,(s,u),(s,v)\in E}$
22: **While** $CandidateQ \neq empty$
23: Extract t from $CandidateQ$ with the highest $support(t, S)$
24: **if** $support(t, S) \geq T_s * |S| * density(S)$ and $density(S + t) > T_d$ **then**
25: $S \leftarrow S + \{t\}$
26: $CanditateQ \leftarrow CandidateQ + \{s_{s:(s,t)\in E}\}$
27: $CandidateQ \leftarrow CandidateQ - \{s_{s:s\notin CandidateQ}\}$
28: **else**
29: break from loop
30: **return** S

After selecting the first (u) and second seed (v), PGCA starts expanding the current cluster which is the set of nodes $S \leftarrow u, v$. The rest of the unclustered nodes are kept in a

TABLE I: 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

priority queue $CandidateQ$ based on their support values with respect to S. If a node does not meet the condition (line 24 of algorithm 3) 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 as in SPICi. For each node remaining in the $CandidateQ$ another check is performed. If the degree of a node is in the first seed partition then this node is discarded as it has the ability to form a new cluster. Otherwise, the node is included in the cluster. When all the nodes are clustered in this way, some isolated nodes remain. These are included in the adjacent clusters of the nearest first seeds or cluster heads. Then the controllers are selected as done in GCA.

The number of clusters is k, and the number of nodes and edges are n and m respectively. PGCA 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 $O(m + nlog_2(n + m) + k \times n^4)$.

VI. Performance Evaluation

A. Simulation Environment

We perform the 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 I. 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 I.

B. 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 [17], J. Liao et. al. use this process to define a latency for a network denoted by $m_{latency}$.

$$m_{latency} = \frac{1}{|S|(|S|-1)} \sum_{s_i, s_j \in S, i \neq j} \{d(s_i, v_i) + \max_{s_m \in Path_{i,j}} (d(v_i, v_m) + d(v_m, s_m))\} \quad (11)$$

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.

C. 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 GCA, IGCA, and PGCA 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 I.

1) Result Comparison for Un-weighted Graph: We propose RCLS for un-weighted graphs. In figure 1 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 figure 1 we represented it as DBCP+LOCAL. RCLS also outperforms DBCP+LOCAL in terms of $m_{latency}$.

We observe from figure 1 that only for the first scenario (Table I) 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 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.

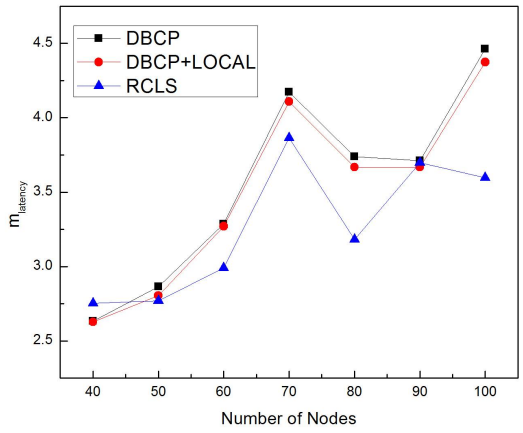

Fig. 1: Comparison of DBCP, DBCP with local search and RCLS on un-weighted networks using $m^{latency}$ as the performance metric.

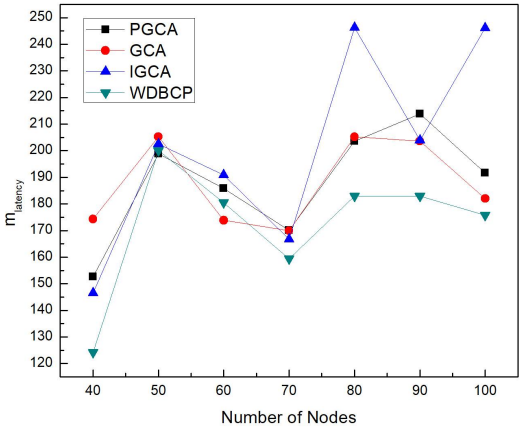

Fig. 2: Comparison of WDBCP, GCA, IGCA, and PGCA on weighted graphs using $m^{latency}$ as performance metric.

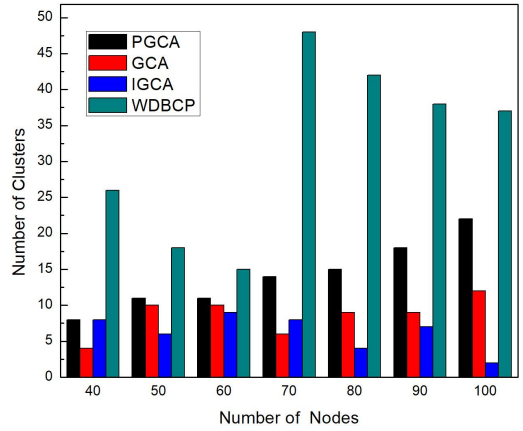

Fig. 3: Comparison of the number of controllers k, for algorithms WDBCP, GCA, IGCA, and PGCA.

2) Result Comparison for Weighted Graph: We propose three algorithms for weighted graphs which are GCA, IGCA, and PGCA. We compare our proposed algorithms with weighted-DBCP. In figure 2, we represent weighted-DBCP as WDBCP, which considers edge weights instead of hop count. The mean of the edge weights of the graph is not constant in each scenario as they were chosen randomly. As a result, in figure 2, the $m_{latency}$ curve of each algorithm is not gradual but a series of abrupt changes. The number of clusters (k) of PGCA increases proportionally with the increase of number of nodes (figure 3) due to its probabilistic approach of creating clusters. As total number of nodes in the graph increases, PGCA ensures that some nodes are discarded from each cluster while some nodes are included, based on their weighted degree, in other words their potential to create new clusters. Therefore, it exhibits a balance between the number of clusters (k) and latency ($m_{latency}$). GCA performs moderately, giving a balanced number of clusters as it chooses the nodes greedily starting with the node with maximum weighted degree. Consequently, the denser parts of the network will stick together, making it an algorithm best fitted for general cases, that is, networks with more uniformity. IGCA depends on the type of input greatly as it inverses the edge weights, and so the parts of the network which are less dense are clustered earlier. It selects nodes to include in the cluster based on minimum weighted degree, causing some distant nodes to be connected with each other. This approach gives a higher $m_{latency}$ value but reduces the number of controllers need as well (figure 3).

We can see that WDBCP gives better results than our proposed algorithms in terms of $m_{latency}$, but from the figure 3, 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 $k * m_{latency}$ in figure 5 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 $k * c$. So the cost-latency product of a network is $c * k * m_{latency}$. The variables k and $m_{latency}$ may vary from scenario to scenario. However, we assume the cost of installing a controller to be constant. There the cost-latency product can be simplified and represented as $k{*}m_{latency}$. From figure 5 we can see that, IGCA and GCA give better results for all of the scenarios of table I, and our proposed algorithms outperform DBCP in terms of cost-latency product. PGCA is consistent and gradual due to the balance it maintains between k and $m_{latency}$ and GCA performs better in sparser networks like scenarios 1, 3 and 4, due to its simple greedy nature. As IGCA reduces the number of controllers k, it performs better in terms of cost-latency product. Accordingly, for scenario 3 when the edge/node ratio is 1.28, IGCA and GCA give the same result and for scenarios 1 and 4, with the edge-node ratios 1.3 and 1.24 respectively, GCA outperforms other algorithms. For other scenarios with edge/node ratio greater than 1.3, IGCA outperforms all other algorithms in terms of

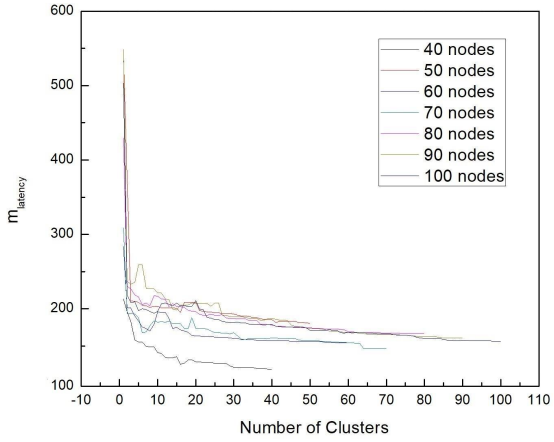

Fig. 4: Comparison of network latency $m_{latency}$ for different numbers of clusters (k) for RCLS on weighted graphs of each scenario.

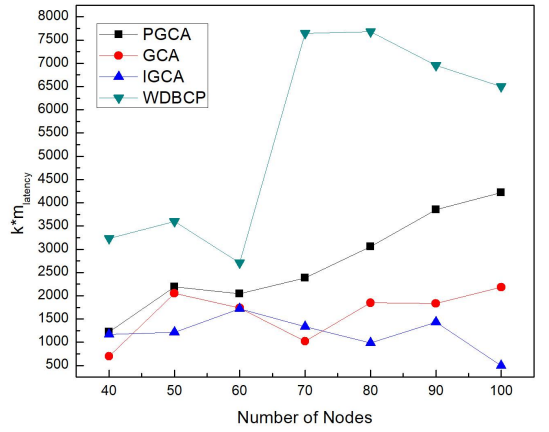

Fig. 5: Comparison of DBCP, GCA, PGCA, IGCA, and RCLS on weighted networks using cost-latency product ($k \times m^{latency}$) as the performance metric.

cost-latency product.

D. 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 figure 4 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.

VII. Conclusion

In this paper, we address the Controller Placement Problem (CPP) of SDN and propose four network clustering algorithms. DBCP is a recent clustering algorithm which addresses the same research problem and is designed for unweighted graphs where it uses hop count as the distance metric. One of our proposed algorithms, RCLS outperforms DBCP in terms of latency for unweighted graphs. However, an unweighted graph is not a good representation of a real network. We propose three algorithms for weighted graphs and validate them 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. We have shown that our algorithms have many advantages over other algorithms.

Firstly our algorithms have polynomial time complexity. Secondly they work with weighted graph which can be fitted to any variable– traffic, bandwidth, latency etc. Thirdly they can cluster the network even after the controllers have been selected, based on important factors like incoming and outgoing traffic, and latency and bandwidth in a congested network, using local search technique.

References

[1] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 7–12.

[2] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *Nsdi*, vol. 10, 2010, pp. 19–19.

[4] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[5] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "Geni: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, pp. 5–23, 2014.

[6] R. Ahmed and R. Boutaba, "Design considerations for managing wide area software defined networks," *IEEE Communications Magazine*, vol. 52, no. 7, pp. 116–123, 2014.

[7] S. Lange, S. Gebert, J. Spoerhase, P. Rygielski, T. Zinner, S. Kounev, and P. Tran-Gia, "Specialized heuristics for the controller placement problem in large scale sdn networks," in *Teletraffic Congress (ITC 27), 2015 27th International*. IEEE, 2015, pp. 210–218.

[8] Y. Zhang, L. Cui, W. Wang, and Y. Zhang, "A survey on software defined networking with multiple controllers," *Journal of Network and Computer Applications*, 2017.

[9] A. K. Singh and S. Srivastava, "A survey and classification of controller placement problem in sdn," *International Journal of Network Management*, p. e2018, 2018.

[10] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 7–12.

[11] L. Yao, P. Hong, W. Zhang, J. Li, and D. Ni, "Controller placement and flow based dynamic management problem towards sdn," in *Communication Workshop (ICCW), 2015 IEEE International Conference on*. IEEE, 2015, pp. 363–368.

[12] Y. Hu, W. Wendong, X. Gong, X. Que, and C. Shiduan, "Reliability-aware controller placement for software-defined networks," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE, 2013, pp. 672–675.

[13] T. Erlebach, A. Hall, L. Moonen, A. Panconesi, F. Spieksma, and D. Vukadinović, "Robustness of the internet at the topology and routing level," in *Dependable Systems: Software, Computing, Networks*. Springer, 2006, pp. 260–274.

[14] Y. Zhang, N. Beheshti, and M. Tatipamula, "On resilience of split-architecture networks," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*. IEEE, 2011, pp. 1–6.

[15] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic approaches to the controller placement problem in large scale sdn networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 4–17, 2015.

[16] A. Sallahi and M. St-Hilaire, "Optimal model for the controller placement problem in software defined networks," *IEEE communications letters*, vol. 19, no. 1, pp. 30–33, 2015.

[17] J. Liao, H. Sun, J. Wang, Q. Qi, K. Li, and T. Li, "Density cluster based approach for controller placement problem in large-scale software defined networkings," *Computer Networks*, vol. 112, pp. 24–35, 2017.

[18] S. K. Rao, "Sdn and its use-cases-nv and nfv," *Network*, vol. 2, p. H6, 2014.

[19] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[20] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, "Advancing software-defined networks: A survey," *IEEE Access*, vol. 5, pp. 25 487–25 526, 2017.

[21] P. Goransson and C. Black, *"1.2 Historical Background" in Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

[22] ——, *"1.6 Can We Increase the Packet-Forwarding IQ?" in Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

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

[24] B. A. Forouzan and S. C. Fegan, *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[25] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[26] K. Sood, S. Yu, and Y. Xiang, "Software-defined wireless networking opportunities and challenges for internet-of-things: A review," *IEEE Internet of Things Journal*, vol. 3, no. 4, pp. 453–463, 2016.

[27] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 1–6.

[28] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, p. 132, 2009.

[29] G. Yao, J. Bi, Y. Li, and L. Guo, "On the capacitated controller placement problem in software defined networks," *IEEE Communications Letters*, vol. 18, no. 8, pp. 1339–1342, 2014.

[30] F. A. Özsoy and M. Ç. Pınar, "An exact algorithm for the capacitated vertex p-center problem," *Computers & Operations Research*, vol. 33, no. 5, pp. 1420–1436, 2006.

[31] P. Jiang and M. Singh, "Spici: a fast clustering algorithm for large biological networks," *Bioinformatics*, vol. 26, no. 8, pp. 1105–1111, 2010. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btq078

[32] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *nature*, vol. 435, no. 7043, p. 814, 2005.