Controller Placement with Optimal Availability

Ran Xu

Delft University of Technology

Controller Placement with Optimal Availability

by

Ran Xu

to obtain the degree of Master of Science
in Electrical Engineering
Track Wireless Communication and Sensing
at the Delft University of Technology,
to be defended publicly on Friday July 14, 2023 at 14:00 PM.

Student number:	5476968	
Project duration:	November 14, 2022 – July 14, 2023	
Thesis committee:	Prof. dr. ir. R.E. Kooij,	TU Delft, chair
	Dr. J.L.A. Dubbeldam,	TU Delft
Supervisors:	Prof. dr. ir. R.E. Kooij,	TU Delft, supervisor
	Fenghua Wang,	TU Delft, daily supervisor

TU Delft

Preface

With this thesis, "Controller placement with optimal availability", I complete the Master of Science degree in Electrical Engineering at Delft University of Technology. The thesis has been carried out at the Network Architectures and Services (NAS) group. First, I would like to express my gratitude to my supervisor Rob Kooij for giving me this chance to research on this nice topic. During the graduation period, he gave me a lot of encouragement and guidance. And my presentation skill has been greatly improved with his help. I would like to thank my daily supervisor Fenghua Wang for her valuable advice for my thesis. She taught me step by step how to use the cluster to run programs and helped me solve many problems. I would like to thank Dr. Johan Dubbeldam for being my thesis committee. Also, I would like to thank the members of the NAS group for giving me a lot of advice on my mid-term review. I would like to thank my friend Xiaoxian Yan for her company. Traveling with her is my happiest memory. Last but not least, I would like to thank my parents for their love and support in these two years.

Ran Xu
Delft, July 2023

Abstract

The controller placement problem concerns the placement of controllers on Software-Defined Networks such that a pre-defined objective is optimized. In this thesis, we conduct research on the controller placement problem with network availability as the performance metric. Unlike other approximate evaluations, we compute the exact value with the path decomposition algorithm, which allows us to accurately measure the quality of different placements. After that, we investigate on the graph metrics' effect on network availability and develop a placement strategy based on degree and distance. Greedy algorithm and genetic algorithm are also introduced to address the controller placement problem. We analyze the optimal placement of OS3E network and other 100 real-world networks. We find that different placements affect availability a lot, which indicates that it is necessary to find a strategy to place controller such that a near-optimal placement is achieved. Finally, four placement strategies are tested on Erdős–Rényi random graphs, Barabási–Albert random graphs, and 155 real-world graphs from the Topology Zoo. Results show that the performance of these four strategies is almost same for most networks. However, the complexity of these four methods is very different, which suggests that the controller placement strategy based on graph metrics is efficient.

Contents

1 Introduction

In contrast to traditional networks, Software-Defined Networks (SDN) achieve a logically centralized control architecture by decoupling the network control plane and the data plane [35]. SDN networks are structured into three distinct planes: the data plane, the control plane, and the management plane. The data plane comprises the networking devices responsible for the efficient forwarding of data packets. The control plane governs the decision-making process for network traffic handling. The management plane remotely monitors and configures the control functionality. This division of planes in SDN provides a flexible and scalable architecture, enhancing network programmability and control [17].

The SDN controllers, as the principal elements within the control plane, play an important role in acquiring comprehensive network-wide information and serving as the central decision-maker. The controllers are required for communication between network devices and applications, highlighting their critical function in SDN architectures. Recognizing its significance, the Controller Placement Problem (CPP) in SDN was first introduced by Heller *et al.* [10]. This problem aims to address two fundamental questions:

- How many controllers are needed?
- Where in the topology should they be placed?

In [10], Heller *et al.* propose a solution for placing a single controller to manage an entire network. However, multiple controllers are often required for wide-area SDN deployments. Multiple controllers can back up each other, thereby mitigating the issue of single-point-of-failures. By adopting a multi-controller design, the performance of networks can be significantly enhanced [11]. However, it is crucial to carefully consider the placement of these controllers as it has a substantial impact on overall network performance. Consequently, the development of effective multi-controllers placement strategies becomes increasingly imperative. This thesis focuses on optimizing availability through the placement of multiple controllers. Additionally, we propose a novel termed “controller reachability” that serves as an indicator of availability.

1.1. Objectives

The objectives of this thesis are:

1. Try to answer two questions proposed by Heller *et al.*: How many controllers are needed? Where in the topology should they be placed?
2. Define controller reachability as a measurement of the availability and propose a method to accurately calculate it when multiple controllers are placed.
3. Find the controller placement strategies such that the availability is optimized.

1.2. Contributions

The main contributions of this thesis are:

1. Propose a novel metric "controller reachability" that serves as an indicator of availability.
2. Apply the path decomposition algorithm to the controller reachability calculation.
3. Find the optimal placement of more than 40,000 graphs and conclude graph metrics' effect on controller reachability.
4. Develop 4 controller placement strategies. One of them is based on the graph metrics, and the other three are heuristic methods.
5. Analyze the optimal placement of 101 real-world networks to conclude the effect of different placements and the number of needed controllers.
6. Compare different placement strategies on synthetic networks and 155 real-world networks. Find the best placement strategy.

1.3. Thesis outline

The structure of this thesis is as follows:

1. Chapter 2 introduces the controller placement problem and the related work. Graph metrics and graph models related to this thesis are introduced. The dataset used in this thesis and the problem that we focus on are introduced.
2. Chapter 3 briefly introduces the principle of enumeration method and Monte Carlo simulation. Then the path decomposition algorithm is introduced, which is an algorithm that can compute the exact all-terminal reliability of a network with restricted pathwidth. At the same time, we describe how to transform the problem of controller reachability into the problem of all-terminal reliability.
3. Chapter 4 introduces the findings we obtain from the optimal placement analysis of numerous graphs. Firstly, we identify two influential metrics, namely degree and distance, that significantly impact controller reachability. Subsequently, some graphs with intersecting controller reachability are presented. The reason behind this is discussed.

4. Chapter 5 introduces the placement strategies based on graph metrics, the greedy algorithm, and the genetic algorithm (classic GA and heuristic GA).
5. Chapter 6 presents the optimal placement of 101 real-world networks. We try to answer the questions "how does the placement affect controller reachability?" and "how many controllers are needed?". Subsequently, we apply different placement strategies on synthetic networks and 155 real-world networks. The performances of different strategies are compared.
6. Chapter 7 presents the conclusion and future work.

2

Background

In this chapter, some background knowledge is introduced. Section 2.1 introduces some commonly used performance metrics in controller placement problems. Section 2.2 reviews related work. Section 2.3 introduces graph metrics and graph models used in this thesis. Section 2.4 introduces the real-world network dataset. Section 2.5 introduces the graph representation of SDN network and defines the controller reachability.

2.1. Performance metrics

The performance of a network can be measured using many criteria. Some commonly used metrics in controller placement problems are introduced as follows.

Latency

Due to the frequent message exchanges between controllers and switches, the latency between controllers and switches is particularly important. The overall latency consists of packet transmission latency, propagation latency, switch queuing latency, and controller processing latency [32]. The controller placement problem for latency aims to find the placement such that the latency is minimized.

Availability

The availability of a system refers to its ability to continue operating even in the presence of failures. In the study by Kumari *et al.* [18], two frequently employed metrics, fault tolerance and reliability, are presented as indicators of availability. Fault tolerance measures the resilience of a network in recovering from different failure scenarios. The computation of this metric involves tallying the count of controller-less nodes, which are nodes that cannot establish connections with any controller, across different scenarios. Reliability is another metric that considers the probability of component failures. It measures the percentage of failed control paths for each failure scenario and combines the percentages with the corresponding probabilities. The controller placement problem for availability aims to find the placement such that the availability is maximized.

In this thesis, we present a novel measure of availability called "controller reachability", which will be explained in 2.5. By employing controller reachability as the performance metric, we explore the impact of different placements on network performance.

Deployment cost

Deployment cost is usually considered with other metrics to form a multiple objectives problem. It can be minimized by reducing the number of controllers, which will influence the performance of the network. Therefore, the cost metric can be optimized under the constraints of other performance metrics like capacity, latency and availability. The controller placement problem for deployment cost aims to determine the SDN planning, including optimal controller number, location, type of controller as well as the interconnection between controllers and controllers/switches such that the deployment cost is minimized with different constraints [25].

2.2. Related work

In [10], Heller *et al.* propose the controller placement problem and try to find the optimal placement in wide-area networks such that the placement minimizes propagation delays. Average latency (k-median) and worst-case latency (k-center) are considered as performance metrics of propagation delays to minimize and the brute force algorithm is used. They find that the random placements are far from optimal in almost all topologies and the placements affect the network performance a lot. In most topologies, average latency and worst latency cannot be both optimized. For small networks, one controller can meet the latency requirement.

In [12], Hu *et al.* formulate the reliability-aware control placement (RCP) problem. A metric is proposed to reflect the reliability of networks, called expected percentage of control path loss. In the RCP problem, they look for the number of controllers and the placement in a network with given failure probability of each component such that the reliability is optimized. They also point out that reliability and latency cannot be optimized at the same time for most networks, however, the placement with optimal reliability brings a quite sufficient average latency.

In [23], Ros *et al.* introduce the fault tolerant controller problem and develop a heuristic algorithm to determine the placement. The lower bound reliability is considered as a performance metric. In this algorithm, they assume that the node with higher connectivity ranks better than others and try to find the minimum number of controllers to reach five nines reliability (99.999%).

In [37], Zhong *et al.* define a metric that reflects the reliability of the control network as the average number of disconnected switches when a single edge fails. To minimize the number of controllers and maximize reliability, a min-cover based method is employed. In this method, the cover denotes a set of nodes whose neighborhood encompasses all switches within the network. After identifying a cover with the minimum size, the controllers are placed.

For the controller placement problem, there is a lot of work using reliability as the performance metric. However, most of them use approximation values. In this thesis, an algorithm that gives exact values is used to calculate the controller reachability, which allows us to accurately measure the quality of different placements.

2.3. Graph Theory Basis

A graph is a mathematical representation of a network and it describes the relationship between nodes and links. The set of nodes is denoted as V, with the number of nodes represented as N. The set of links is denoted as E, with the number of links represented as L. A graph is denoted by $G(N,L)$, representing a network with N nodes and L links. In this section, graph metrics and graph models related to this thesis are introduced.

2.3.1. Graph metrics

Degree

Degree d_j of a node j in a graph $G(N,L)$ is defined as the number of its neighboring nodes [30]. Degree is an essential metric for studying theoretical and real networks. It is the simplest measurement of centrality, where higher values mean that the node is more central (important) [9]. In a regular graph, every node has the same degree.

The minimum node degree of a graph is defined as,

$$d_{\min} = \min_{j \in G} d_j \tag{2.1}$$

The average degree of a graph is defined as,

$$E[D] = \frac{1}{N} \sum_{j=1}^{N} d_j = \frac{2L}{N} \tag{2.2}$$

The average degree can be used to present whether a graph is dense or sparse.

Connectivity

A graph is connected if there is a path between each pair of nodes. Connectivity is defined as the minimum number of elements that need to be removed to separate the remaining nodes into two or more isolated subgraphs. The edge-connectivity $\lambda(G)$ is the minimum number of edges to remove to disconnect graph G. The node-connectivity $\kappa(G)$ is the minimum number of nodes to remove to disconnect graph G. The node-connectivity is less or equal to edge-connectivity and both of them are less or equal to the minimum degree of the graph.

$$\kappa(G) \leq \lambda(G) \leq d_{\min}(G) \tag{2.3}$$

Shortest path

Given an undirected network G, a nonnegative weight w_l associated with each edge $l \in E$, an origin node $s \in V$, and a destination node $t \in V$, the shortest path is the path such that the total weight from s to t is minimal [34]. In unweighted graphs, the shortest path is the path between s and t with the minimum number of edges.

2.3.2. Graph models

A random graph is a graph whose topology is determined in a random way. It models the irregular behaviour of real-world networks by making connections through a random process [29]. Different random graph models produce graphs with different properties based on different stochastic ways. Two random graph models are used in this thesis.

Erdős–Rényi model

The Erdős–Rényi (ER) random graph is a well-known random graph model [7], which can be generated by letting every possible edge between any two nodes independently exists with an edge probability p. The model can be represented as $G(n,p)$ where n is the number of nodes, and p is the edge probability.

Since each edge is present independently with probability p, the degree distribution is binomial as follows,

$$P(D=k) = \binom{n-1}{k} p^k (1-p)^{n-1-k} \tag{2.4}$$

The average degree of Erdős–Rényi random graph is

$$E[D] = (n-1)p \tag{2.5}$$

The expected link number is

$$L = \frac{n(n-1)p}{2} \tag{2.6}$$

The threshold for the connectedness of $G(n,p)$ is expressed as $p_{th} = \ln(n)/n$. If p is smaller than p_{th}, the generated graph is almost certainly disconnected. If p is larger than p_{th}, the generated graph is almost certainly connected.

Barabási–Albert model

A power-law degree distribution was first observed in real-world networks by Barabási and Albert. They argued that the scale-free nature of real-world networks is based on two generic mechanisms: (1) networks expand continuously by the addition of new vertices; (2) new vertices attach preferentially to sites that are already well connected [2].

The Barabási–Albert (BA) model can be represented as $G(n,m)$ where n is the total number of nodes in the graph, m is the number of new links established whenever a new node is added.

In this thesis, the process of generating a BA graph starts with a star network of $m+1$ nodes. At each time, one new node is added into the network by being connected with m nodes that already exist. The connection probability is proportional to the degree of the existing nodes, therefore, the new node prefers to attach to a high-degree node. If an existing node i has degree d_i, the probability that the new node chooses node i to connect is

$$p_i = \frac{d_i}{\sum_{j \in G} d_j}. \tag{2.7}$$

2.4. Topology Zoo

The Internet Topology Zoo is a dataset of 232 network data created from the information that network operators make public [14]. Much research related to controller placement problem is studied based on the networks from the Topology Zoo. In this thesis, the analysis of real-world networks is also based on networks from the Topology Zoo.

From the Topology Zoo we choose 150 small size ($11 \leq n \leq 50$) connected graphs and 5 middle/large size ($50 < n$) connected graphs. The average node degree and network sizes are shown in Fig. 2.1. The networks used for analysis are all sparse networks where the average node degree varies from 1.875 to 4.48. In this dataset, Abilene is the smallest network with 11 nodes and 14 edges, Cogentco is the largest network with 197 nodes and 243 edges.

2.5. Problem

The SDN network is represented as an undirected graph $G(N,L)$, where N represents the number of nodes, L represents the number of links. With the assumption that connections between controllers and nodes are always operational, we can present that controllers are co-located with nodes in the data plane. A graph representation of a SDN network is shown in Fig. 2.2, where each node is a switch and the red node indicates that a controller is co-located with a switch.

We consider that the switches/controllers (nodes) are always operational, and the physical links between switches are operational with probability p. In this thesis, the availability is quantified through the concept of controller reachability, which is defined as the probability that each switch in the network is connected to at least one controller.

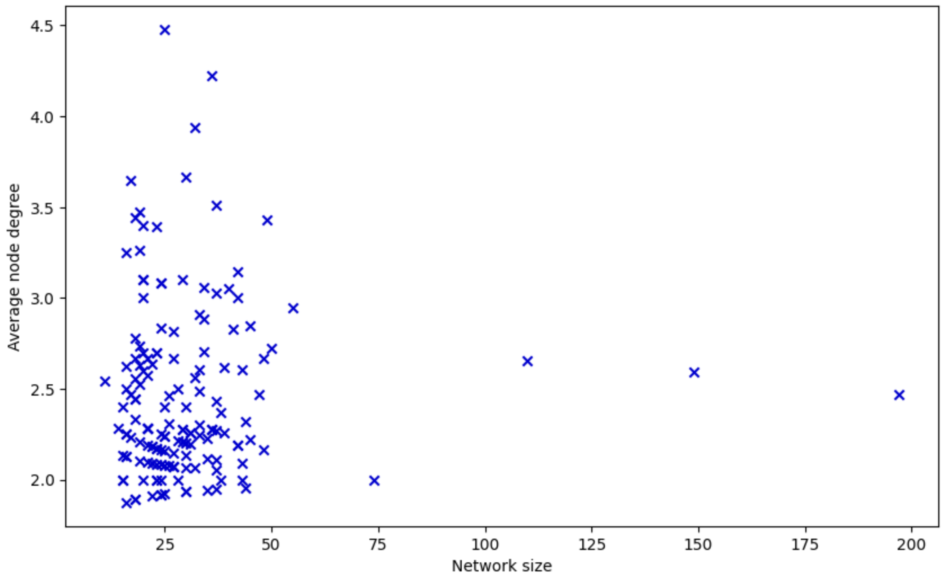


Figure 2.1: Average node degree vs. network size of networks from the Topology Zoo we analyse.

What is the reason behind the performance difference when applying different placements? Does there exist an optimal placement? If K controllers will be deployed in a network, where should they go such that the controller reachability is optimized? Can we conclude the minimum number of controllers to meet a certain requirement? Those are the questions to answer in this thesis.

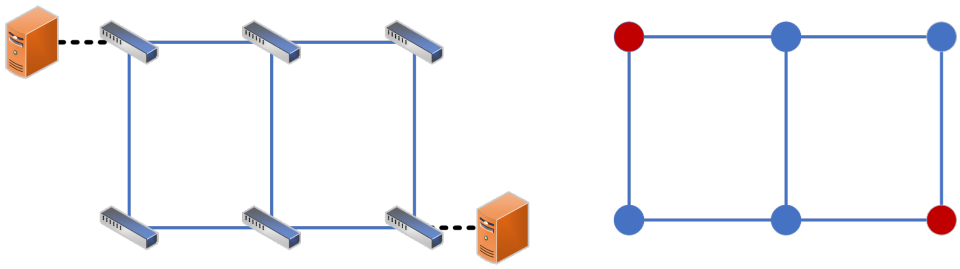


Figure 2.2: Graph representation of SDN network with switches and controllers.

3

Controller reachability evaluation

In this chapter, controller reachability evaluation methods are introduced. Section 3.1 introduces how to calculate controller reachability using enumeration. Section 3.2 introduces how to estimate controller reachability using Monte Carlo simulation. Section 3.3 introduces a method to transform controller reachability into all-terminal reliability. Meanwhile, we introduce the path decomposition algorithm, which is an algorithm that can compute the exact all-terminal reliability of a network. Section 3.4 proves that controller reachability is not a submodular function.

3.1. Enumeration

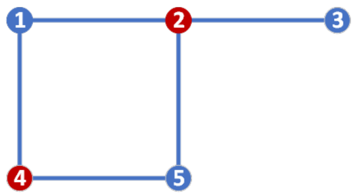


Figure 3.1: An example network. Each node is a switch and the red node means a controller is co-located with a switch.

A working state s can be represented as a set of binary numbers consisting of 0s and 1s, where each element I_{ij} indicates whether the link between node i and j is working or not. Based on the assessment of the state elements, we can know whether the state meets the requirement that each node is connected to at least one controller, which can be represented as a binary number I_s. For instance, the state of the example network is $(I_{12}, I_{23}, I_{14}, I_{25}, I_{45})$. Consider a state (1, 1, 1, 0, 0), which indicates that there are link failures between nodes 2 and 5, as well as between nodes 4 and 5. In this case, the node 5 cannot reach any controller. Therefore, this state has $I_s = 0$.

Exact computation of controller reachability can be done by complete enumeration of all states. The controller reachability can be expressed as

$$P_{cr} = \sum_{s \in A} I_s P_s, \tag{3.1}$$

where A is the complete set of states, I_s is a binary number which indicates whether state s meets the requirement that each node is connected to at least one controller, P_s is the probability of state s.

The controller reachability of the example network in Fig. 3.1 is computed by enumerating 2^5 states. The result is $4p^3 - 4p^4 + p^5$.

However, the running time grows exponentially with the number of links. For a graph with L edges, the size of the complete states set is 2^L. Therefore, we can only use brute force enumeration for small networks.

3.2. Standard Monte Carlo simulation

Monte Carlo Simulation is a sampling technique, which has been one of the most widely used methodologies for reliability estimation [8]. When sampling, each link fails with probability $1 - p$. If each node in the result network can reach at least one controller, $I_t = 1$. Otherwise, $I_t = 0$. The estimated controller reachability can be expressed as

$$\hat{P_{cr}} \approx P_{crMCS} = \frac{1}{t} \sum_{1}^{t} I_t \tag{3.2}$$

where t is the sampling times, I_t is a binary number which indicates whether this sample meets the requirement that each node is connected to at least one controller.

However, Monte Carlo Simulation is not computationally efficient in estimating rare event probability [3]. This method requires many samples to get a good approximation, which can result in a long total runtime if each sample takes a long time to process.

Monte Carlo simulation of the example network in Fig. 3.1 is shown in Fig. 3.2.

3.3. Path decomposition

Enumeration offers accurate results but is burdened with high computational complexity. On the other hand, Monte Carlo simulation provides an approximation, but requires a significant amount of computation time to achieve higher levels of accuracy. Due to constraints imposed by network size and computation time, both enumeration and Monte Carlo simulation approaches have limitations. In this thesis, the path decomposition algorithm is employed as an alternative solution. This algorithm enables the computation of the all-terminal reliability polynomial for graphs with restricted pathwidth [22]. The application of this algorithm to evaluate controller reachability will be explained first, followed by a detailed exposition of its functioning principles.

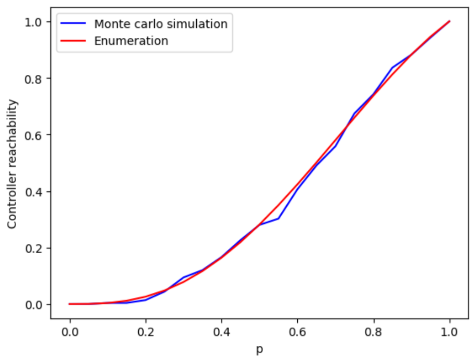


Figure 3.2: A Monte Carlo simulation is conducted on the example network with 500 samples. In this figure, the x-axis denotes the link operational probability p, the y-axis denotes the controller reachability. The blue curve corresponds to the Monte Carlo simulation's approximate value, whereas the red curve is the exact value obtained through enumeration.

3.3.1. Controller reachability vs. All-terminal reliability

As mentioned in Chapter 2, the controller reachability is defined as the probability that each node is connected to at least one controller. Therefore, the network after links failure can be a disconnected network, but each component must have at least one controller.

The example network in Fig. 3.1 is represented as a graph, wherein solid lines denote working edges and dashed lines indicate failed edges. Three distinct cases are presented in Fig. 3.3. In the first case, the network remains connected. It is evident that each node establishes a connection with at least one controller. In the second case, the network undergoes disconnection, resulting in its division into two components (145 and 23) due to the failure of two specific edges. Nevertheless, each component is equipped with a controller, thereby ensuring that every node remains connected to at least one controller. In the third case, the network undergoes disconnection, leading to its division into two components (1245 and 3) as a consequence of the failed edges. Node 3 fails to establish connectivity with any controller within this case.

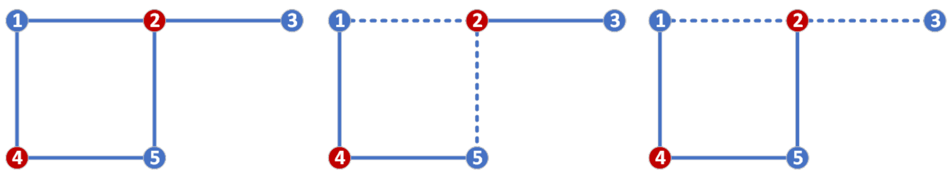
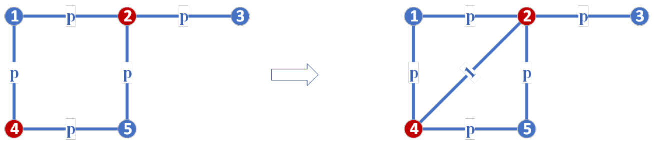
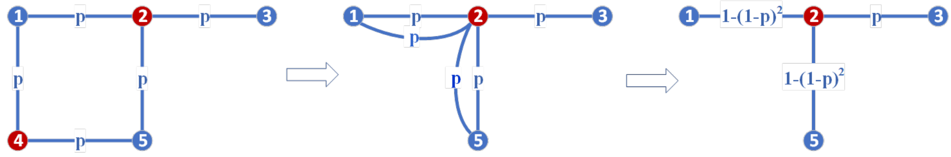


Figure 3.3: Three cases of network partitions after edges failure. The solid lines indicate working edges and the dashed lines indicate failed edges.

There is a way to relate the controller reachability with the network all-terminal reliability, which is defined as the probability that each node can communicate with every other node in a network [8]. The main idea of this approach is to reestablish connectivity among the network components. This can be achieved by introducing additional always operational links between controllers or by merging all controllers into a single node. In this thesis, the second method is used.



(a) Introduce additional always operational links between controllers.



(b) Merge all controllers into a single node.

Figure 3.4: Two ways to reestablish connectivity among the network components.

After merging all nodes with the controller into a single node, the result network may have duplicate links. The topology is further simplified by removing these duplicate links and adjusting the corresponding link probabilities. Subsequently, a graph with a single controller is obtained, and its all-terminal reliability is equivalent to the controller reachability of the original graph. The process of controllers merging serves multiple purposes. It not only reduces the number of controllers to one but also reduces the overall number of nodes and edges in the network. This reduction has the benefit of simplifying the subsequent path decomposition process, resulting in decreased complexity. From the rightmost figure in Fig. 3.4b, we can immediately determine the controller reachability because the graph is connected if and only if all links are operational. Hence, we get $(1-(1-p)^2)^2 p$ which can be simplified to the expression $4p^3 - 4p^4 + p^5$ we already obtained in Section 3.1.

Now the computation of controller reachability is transformed into the computation of all-terminal reliability, which is an NP-hard problem [33]. There are many techniques that evaluate the all-terminal reliability [8]. In this thesis, the decomposition method is used [4].

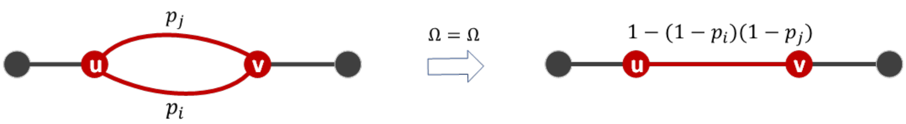
3.3.2. Graph reductions

Before the decomposition of a graph, graph reductions can be used to further simplify the graph. Consider a connected graph $G = (N, L)$ whose nodes are always operational and edges are operational with link probability. Reliability-preserving reductions are used to simplify the graph topology and adjust link probability [26]. It reduces the complexity of calculating $R(G)$. After reductions, $R(G) = \Omega R(G')$, where Ω is a multiplicative factor (initially, Ω is 1).

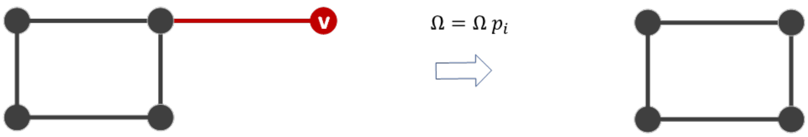
In this thesis, three simple reductions are used.

- Parallel reduction: Suppose $e_i = (u, v)$ and $e_j = (u, v)$ are two parallel edges with link probability p_i, p_j. Parallel reduction replaces e_i and e_j with a single edge $e_{new} = (u, v)$, which has link probability $p_{new} = 1 - (1 - p_i)(1 - p_j)$. Ω is unchanged.
- Degree-1 reduction: Suppose v is a node whose degree is 1. It is connected to the rest network through a single edge e_i with link probability p_i. Degree-1 reduction removes this edge and multiplies Ω with p_i.

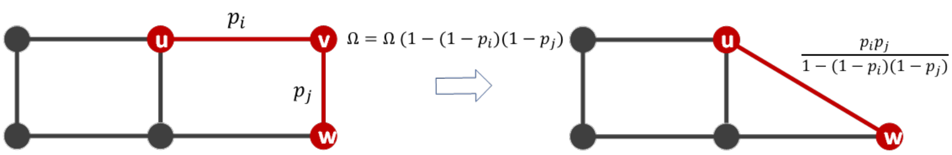
- Degree-2 reduction: Suppose v is a node whose degree is 2. It is connected to the rest network through two edges $e_i = (u,v)$ and $e_j = (v,w)$ with link probability p_i, p_j. Degree-2 reduction replaces e_i and e_j with a single edge $e_{new} = (u,w)$, which has link probability $p_{new} = p_i p_j / (1-(1-p_i)(1-p_j))$. After reduction, multiply Ω with $(1-(1-p_i)(1-p_j))$.



(a) Parallel reduction



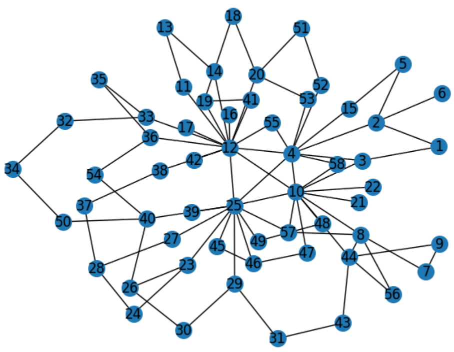
(b) Degree-1 reduction



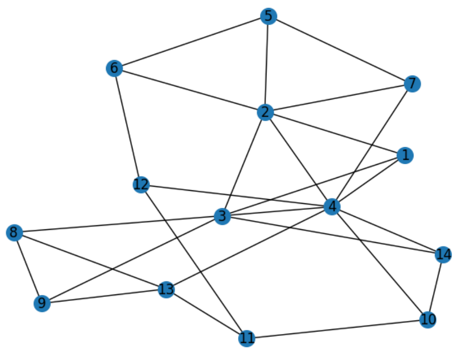
(c) Degree-2 reduction

Figure 3.5: Examples of Reliability-preserving reductions.

Graph reductions reduce the number of nodes and the number of edges, simplifying the calculation. An example is shown in the Fig. 3.6. This is the DFN network from the Topology Zoo with 58 nodes and 87 edges. We consider nodes are always operational and edges are independently operational with link operational probability p. After reductions, the network has 14 nodes and 27 edges, where each edge has a different link operational probability and the minimum node degree is 3.



(a) Original DFN network



(b) DFN network after reductions

Figure 3.6: Graph reduction of DFN network.

3.3.3. Decomposition method

The decomposition method for evaluating the reliability of a network whose elements fail independently with known probabilities was first introduced by A.Rosenthal [24]. A.Pönitz and P.Tittmann [22] presented the path decomposition algorithm which can compute the all-terminal reliability polynomial of a graph with restricted pathwidth in polynomial time by node and edge operation. The method used in this thesis is based on this algorithm. I will start with some basic concepts and then introduce the algorithm.

- Decomposition principle: A connected graph can be represented as $G(V,E)$, where V is the set of nodes, E is the set of edges. The decomposition considers a subgraph $H(V',E')$ and its complement $H_C(V'',E'')$ such that H and H_C are separated by the boundary set $F = V' \cap V''$ [4]. In the beginning, H is a null graph and H_C is graph G. After the entire graph is processed, H is graph G and H_C is a null graph. During processing, the states of the boundary set store all information needed to calculate the all-terminal reliability of subgraph H.

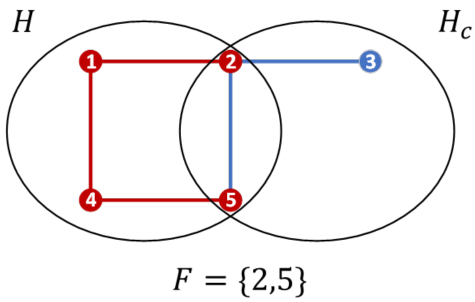


Figure 3.7: An example of the boundary set. In the subgraph H, the set of nodes $V' = \{1,2,4,5\}$. In the complement graph H_C, the set of nodes $V'' = \{2,3,5\}$. The boundary set $F = V' \cap V'' = \{2,5\}$.

- The partition: Since each edge of H fails independently, there will be several events associated with different probabilities. If each connected component C_1, C_2,..., C_r of H has at least one node from the boundary set F, this is a working event. Otherwise, it is a failure event that will lead to the disconnection of graph G. For each working event, we can get the partition π with r block: $B_1 = C_1 \cap F$, $B_2 = C_2 \cap F$,..., $B_r = C_r \cap F$. Events with the same partition are equivalent. The set can have many partitions and two of them are extreme cases. The first extreme case is that the partition has one block that contains all nodes in the set. The second extreme case is that the partition has $|F|$ blocks such that each block contains one node from the set.

  One example of different partitions is shown in Fig. 3.8. In this graph, the red part is the subgraph H that has been processed, the blue part is the complement H_C of the subgraph. They share the boundary set $F = \{2,4\}$. Three events are shown. For the first event, there is one connected component C in subgraph H. The partition π of the boundary set has one block $B = C \cap F$, $\pi = [24]$. For the second event, there are two connected components C_1, C_2 in subgraph H. The partition has two blocks $B_1 = C_1 \cap F, B_2 = C_2 \cap F$, $\pi = [2][4]$. For the third event, there are three components. However, the component with node 1 is disconnected with the boundary set. Therefore, it is a failure event.

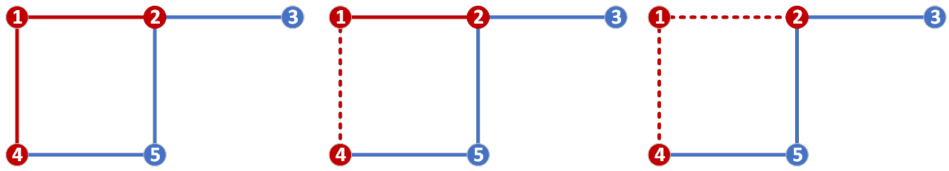


Figure 3.8: Different partitions of three events.

- Path decomposition: A path decomposition of G is represented by a sequence of active nodes set (sometimes more than boundary set). For graph G with n nodes, the path decomposition is a sequence of active nodes set $(X_1, X_2, ..., X_{2n+1})$ where the first and the last set is $\emptyset$. The inclusion of node v into the active node set is called the activation of node v, while the removal of node v from the active node set is called the deactivation of node v.
- The states: A state is a pair (π, P_π) where π is one partition of X and P_π is the probability of the presence of partition π. The states of active nodes set X is a set of pairs that includes all possible partitions of X. In the path decomposition sequence, the states of each step can be calculated from the previous states by node and edge operations.

Node and edge operation

Node and edge operations are the basis of the decomposition method, which facilitate the transformation of states with different active nodes set $X_i \to X_{i+1}$. For a graph with N nodes and L edges, we can get a $2N + L$ steps series where N steps are activations of node, N steps are deactivations of node, L steps are edge processing. The node and edge operations involve the transformation of states as follows,

- Activation of node v: The activation of node v extends all partitions by a singleton without changing the probability.

$$\{(\pi, P_\pi)\} \to \{(\pi \mid v, P_\pi)\} \tag{3.3}$$

- The deactivation of node v: If v is the last node in the boundary set, the corresponding probability is the final all-terminal reliability, that is $(v, P_v) \to (\emptyset, P_v)$. If v is not the last node, for each state, the node v is removed from partition π. And there are two cases. If v is a singleton in π, the corresponding state is removed from the state set, that is $(\pi, P_\pi) \to \emptyset$. Otherwise, the states with the same partition after removal will be merged into one state, and their probability will be added up together.

$$\bigcup_{\pi \in M(\sigma, v)} \{(\pi, P_\pi)\} \to \left\{ \left(\sigma, \sum_{\pi \in M(\sigma, v)} P_\pi \right) \right\} \tag{3.4}$$

where σ is the partition after removal, $M(\sigma, v)$ is the set of all partitions that can be obtained from the partition σ by inserting node v into one block.

- Processing edge $e = \{u, v\}$: The processing of edge e with link probability p_e brings two states, corresponding to either the failing or the working of e.

$$\{(\pi, P_\pi)\} \to \{(\pi, (1-p_e)P_\pi), (\pi \vee e, p_e P_\pi + P_{\pi \vee e})\} \tag{3.5}$$

where $\pi \vee e$ is the partition after u and v are connected by edge e.

How to determine Path Decomposition

Given that the transformation of states is required for each node operation and edge operation, the computational complexity of the algorithm primarily relies on the number of possible partitions within the boundary set. Consequently, the algorithm exhibits an exponential complexity with respect to the maximum size of the boundary set. Therefore, it is important to find a nice decomposition such that the maximum boundary set is small.

The width of a path decomposition is defined as $\max|X_i| - 1$. The pathwidth of a graph is defined as the minimum width among all possible path decompositions. However, finding the pathwidth of a graph is NP-hard [21].

A heuristic algorithm is used in [22] to obtain an upper bound pathwidth. This algorithm defines the neighborhood $N(X)$ of a node set X as the set of neighboring nodes of X (excluding the nodes already in X). During each step, the algorithm selects a node from the neighborhood of the active set, aiming to minimize the width at the current stage. By considering each node as an initial node, the algorithm identifies a path decomposition with the smallest possible width, which is also the upper bound pathwidth we find. Furthermore, to enhance the algorithm's efficiency, an additional improvement can be made by minimizing the size of each active node set in the path decomposition sequence $(X_1, X_2, ..., X_{2n+1})$. The improved algorithm operates as follows:

- Start with a node v
- Choose the next node from neighborhood $N(X)$. The prior choice is the node that can cause node deactivation of other nodes in set X. If there is no such node, choose the node that has more neighbors in set X.
- Continue this process until every node is included.
- Record the maximum width $\max|X_i| - 1$ of each set in $(X_1, X_2, ..., X_{2n+1})$.
- Record the sum of width of each set in $(X_1, X_2, ..., X_{2n+1})$
- The width of this path decomposition is $\max|X_i| - 1$.
- Repeat this process with each node as the start node.
- The solution with the minimal found width as well as the minimal sum of width is the final solution.

Decomposition algorithm

After determining the path decomposition $(X_1, X_2, ..., X_{2n+1})$, we can convert it into a series which represents the node and edge operations. The length of the series is $2N + L$, with N steps node activation, N steps node deactivation, and L steps edge processing. The decomposition algorithm is shown in Algorithm 1.

Algorithm 1 Decomposition algorithm

Input: network, link probability, series
Output: all-terminal reliability

for step in series **do**
  if step is activation of node v **then**
    use Eq.(3.3) to update the state
  else if step is deactivation of node v **then**
    if v is the last node **then**
      return all-terminal reliability
    else
      use Eq.(3.4) to update the state
      merge the states with the same partitions
    end if
  else if step is processing edge $e = (u, v)$ **then**
    use Eq.(3.5) to update the state
    merge the states with the same partitions
  end if
end for

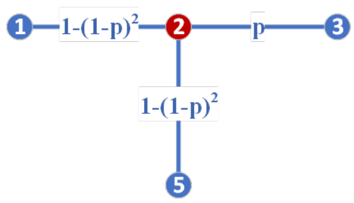


Figure 3.9: An example graph. This graph is the controllers-merged graph of the example graph used in previous sections.

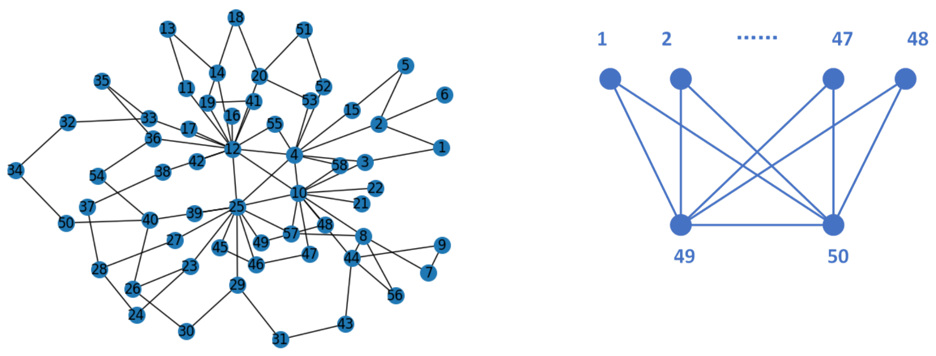
In order to show the decomposition process intuitively, the graph in Fig 3.9 is used as example to compute controller reachability. Each step of the decomposition is shown below,

- Apply the algorithm to determine the path decomposition: ($\emptyset$, $\{1\}$, $\{1,2\}$, $\{2\}$, $\{2,5\}$, $\{2\}$, $\{2,3\}$, $\{3\}$, $\emptyset$)
- Convert the path decomposition into a series: (1, 2, $\{1,2\}$, -1, 5, $\{2,5\}$, -5, 3, $\{2,3\}$, -2, -3)
- Activation of node 1: $\{(1,1)\}$
- Activation of node 2: $\{(1/2,1)\}$
- Process edge (1, 2): $\{(1/2, 1-2p+p^2), (12, 2p-p^2)\}$
- Deactivation of node 1: $\{(2, 2p-p^2)\}$
- Activation of node 5: $\{(2/5, 2p-p^2)\}$

- Process edge (2, 5): $\{(2/5, 2p-5p^2+4p^3-p^4), (25, 4p^2-4p^3+p^4)\}$
- Deactivation of node 5:$\{(2, 4p^2-4p^3+p^4)\}$
- Activation of node 3:$\{(2/3, 4p^2-4p^3+p^4)\}$
- Process edge (2, 3): $\{(2/3, 4p^2-8p^3+6p^4-p^5), (23, 4p^3-4p^4+p^5)\}$
- Deactivation of node 2: $\{(3, 4p^3-4p^4+p^5)\}$
- Deactivation of node 3: $\{(\emptyset, 4p^3-4p^4+p^5)\}$
- The all-terminal reliability polynomial of the graph is $4p^3-4p^4+p^5$

The result is the same as the polynomial we obtained by enumeration in section 3.1.

Examples



(a) DFN network **(b)** Crown graph with 50 nodes

Figure 3.10: Two medium-size networks. The first network is the DFN network from the Topology Zoo with 58 nodes and 87 edges. The second network is the crown graph with 48 nodes above and 2 nodes below as well as 97 edges. The nodes below are connected to all the nodes above as well as to each other.

We conduct verification tests on two medium-size networks as shown in Fig. 3.10. In each network, a single controller is placed, resulting in two sets of controller reachability polynomials. By comparing the obtained crown graph polynomial with the explicitly determined all-terminal reliability polynomial in [38], we have successfully validated the correctness of this algorithm.

The controller reachability polynomial coefficients of the DFN network are presented as an array $[a_0, a_1, \cdots, a_L]$, with the first term representing the coefficient of p^0 and the last term representing the coefficient of p^{87}.
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,2060018066147024,-48922083832382748,563773811808216652,-4197967847144067808,2269
0952719633304819,-94825753065586701500,318636033303977154569,-883991742306099561069,
2063187833079466097099,-4107128216259914397525,7045296855043259221780,-1049441917014
9777036700,13651868541903364365850,-15573399654668691827604,15621777150479073230122,

-1380111208281665986044 8,1074324850423645185917 2,-736424971098764920257 2,4437581566432477389260,-2343826185117056970440,1080479398811705865712,-4321917525082967354 44,14883380301122954522 4,-436688009630801506 64,10765772602951229416,-2188273057961438688,357092254403354256,-44975523105486720,4104282291518016,-241516521469440,6881218790400]

The controller reachability polynomial coefficients of the crown graph are presented as an array $[a_0, a_1, \cdots, a_L]$, with the first term representing the coefficient of p^0 and the last term representing the coefficient of p^{97}.
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7036874417766400,-2516386291793264 64,4577346071268687872,-5621470460502448537 6,521678544223623708672,-3882123602388527874048,24016952839219036815360,-126524141017734108413952,577355852607375838019584,-2311313508637496198561792,8197925891160955303231488,-25965145028967635739475968,73907794630095462786400256,-190057374465755313346707456,443477150533016808069267456,-942399216324538102745923584,1829372869132327263852822528,-3252226789269948355470950400,5306271371021675492062265344,-7959411858806378158180270080,10991571966291958839388405760,-13989275385599502060391235584,16422193962637889414639910912,-17790710710498060337525293056,17790710991831066823882702848,-16422194886542141485221937152,13989277176623500618838835200,-10991574943709823426308866048,7959416345204870410892476416,-5306277565587851929744048128,3252234637591338250751967232,-1829381983810286694853312512,942408900791055691459067904,-443486541557688076668370944,190065660669386853678710784,-73914423593979707309408256,25969932614154965709258752,-8201031351846325679431680,2313111406932045349543936,-578277851733129405693440,126939040624357259950080,-24178864880830964046336,3936094282926043949568,-536740129489919442624,59637792165546812288,-5185894970917121632,331014572611731360,-13792273858822143,281474976710656]

3.4. Is controller reachability a submodular function?

A submodular function is a set function that exhibits a diminishing returns property. For a submodular function, greedy algorithms can achieve a near-optimal solution with a performance within $1 - 1/e$ of the optimal solution [27]. It has been found that submodular functions are useful in real-world problems such as in social network [20] and sensor placement [16]. In [28], a submodularity-based method is applied in a controller placement problem, whose goal is to find a set of controllers such that this set can maximize a predefined submodular function. The maximum covered submodular problem, maximization control submodular problem, and maximization network quality factor submodular problem of controller placement problem are discussed in their work. Hence, the question arises: does controller reachability exhibit the submodularity as well?

Submodularity is a property of a set function, i.e., functions $f : 2^V \to \mathbb{R}$ that assign each subset $S \subseteq V$ a value of $f(S)$ [15].

Definition 1 (Submodularity) *A function $f : 2^V \to \mathbb{R}$ is submodular if for every $A \subseteq B \subseteq V$*

and $e \in V \setminus B$ *it holds that*

$$\Delta(e \mid A) \geq \Delta(e \mid B) \tag{3.6}$$

where $\Delta_f(e \mid S) = f(S \cup \{e\}) - f(S)$ *is the discrete derivative of* f *at* S *with respect to e.*

An equivalent definition is, a function $f : 2^V \to \mathbb{R}$ is submodular if for every $A, B \subseteq V$,

$$f(A \cap B) + f(A \cup B) \leq f(A) + f(B) \tag{3.7}$$

Unlike the submodularity exhibited by the cover-based function in controller placement problems, we prove that controller reachability is not submodular. This assertion is supported by various counterexamples. We will illustrate the non-submodularity using a path graph containing n nodes, as depicted in Fig. 3.11. In our example, V refers to the nodes of the network, S refers to the place where controllers are located, $f(S)$ is the corresponding controller reachability, $f(\emptyset) = 0$.

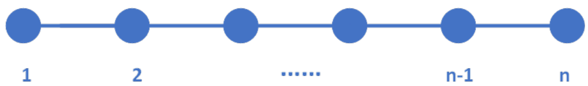


Figure 3.11: Path graph with n nodes.

Let $A = \{1\}$, $B = \{n\}$. Therefore,

- $f(A \cap B) = f(\emptyset) = 0$
- $f(A \cup B) = f(\{1, n\}) = p^{n-1} + (n-1)(1-p)p^{n-2}$
- $f(A) = f(\{1\}) = p^{n-1}$
- $f(B) = f(\{n\}) = p^{n-1}$

According to Eq.(3.7), for submodularity we need

$$\begin{aligned} p^{n-1} + (n-1)(1-p)p^{n-2} &\leq 2p^{n-1} \\ \frac{n-1}{n} &\leq p \end{aligned} \tag{3.8}$$

which obviously does not hold for p sufficiently small, i.e. $0 < p < \frac{n-1}{n}$

4

Findings from optimal controller placement

In this chapter, several findings regarding the optimal placement of controllers are presented. Section 4.1 introduces the relationship between degree and controller reachability. Section 4.2 introduces the relationship between distance and controller reachability. Section 4.3 use some graph examples to show that a network might have a different optimal placement with different link operational probability p.

4.1. Degree vs. Controller reachability

Degree is an important metric in graph theory. Edge connectivity $\lambda(G)$, the minimum number of links whose removal disconnects G, is inherently bounded by the minimum node degree. Consider two graphs with the same parameters N, L, and connectivity. The graph that has more nodes/edges responsible for the low connectivity is less reliable than the other one [30]. All of these show that the degree has a great impact on the reliability of the network. This prompts the question: does controller reachability relate to node degree?

In the case study conducted on a specific graph from the Topology Zoo (Fig. 4.1), several interesting observations were made regarding the optimal placement of controllers and its relationship with node degree. When considering different numbers of controllers ($K = 2,3,4,5$) and a high link operational probability ($p = 0.99$), it is surprising to find that the optimal controller reachability occurred when the controllers were placed on nodes with a degree of 1. This outcome is counterintuitive, as the optimal placement concentrates controllers on a specific side of the network. This finding suggests that nodes with a degree of 1 have a significant influence on controller reachability.

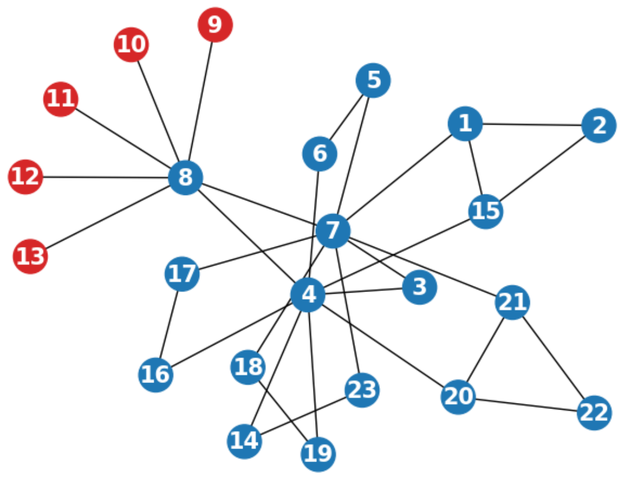


Figure 4.1: Optimal placement of Aconet when $K = 5, p = 0.99$.

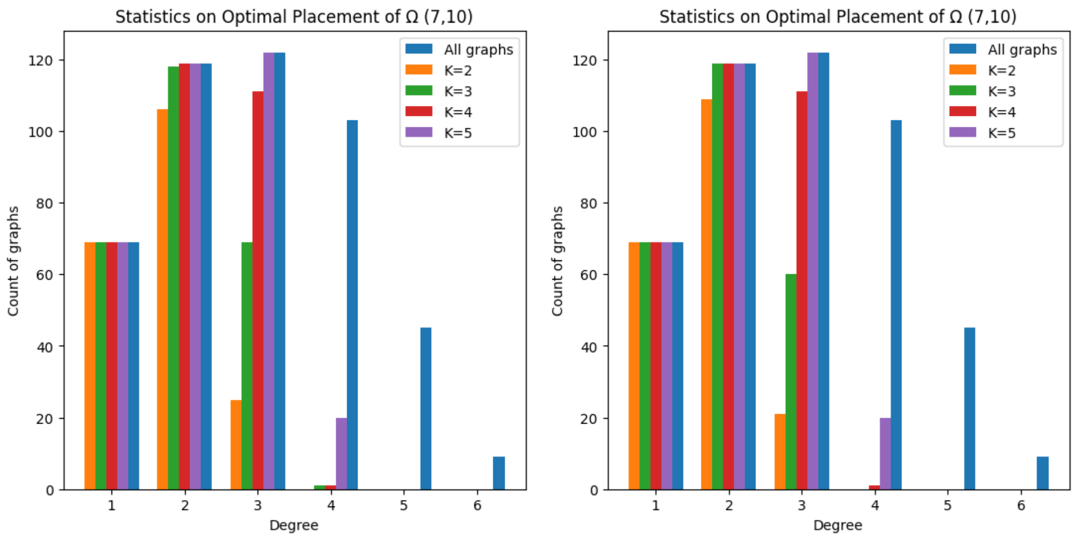
In order to verify whether this is a coincidence, more graphs are tested. For small graphs, all non-isomorphic connected graphs with the same number of nodes and links can be generated with tools in Nauty and Traces [19]. The class of graphs is defined as $\Omega(N,L)$, where N is the number of nodes, L is the number of edges. Three graph classes with different average degree are chosen, the number of graphs in each class is shown in the Table 4.1.

	Average degree	Number of graphs
$\Omega(7,10)$	2.86	132
$\Omega(10,12)$	2.4	8548
$\Omega(9,18)$	4	33366

Table 4.1: The average degree and the number of graphs in three graph classes.

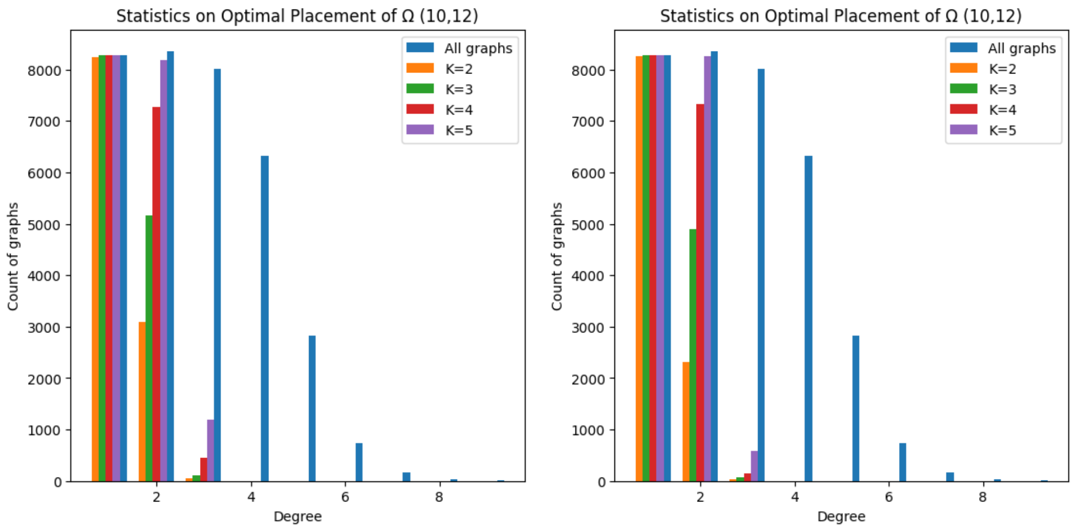
In order to investigate the optimal controller placement in these three graph classes, the enumeration approach is employed. For each graph within the selected classes, all possible placements are examined to identify the optimal placement when considering 2, 3, 4, and 5 controllers. To assess the controller reachability of each placement, the path decomposition algorithm presented in Chapter 3 is utilized to compute the controller reachability polynomial. Additionally, the exact controller reachability is determined by calculating the probabilities over a range of link probabilities, from 0.1 to 0.99. Throughout this process, the degree of the optimal controller placement is recorded. The gathered data yield statistical results, which are presented in Fig. 4.2.

Fig. 4.2 provides valuable insights into the relationship between node degree and the optimal placement of controllers. The blue bars represent the number of graphs containing nodes with a specific degree, while the remaining bars indicate the number of graphs where the optimal placement of K controllers includes at least one node with that degree. For comparative analysis, two different link probabilities, $p = 0.1$ and $p = 0.99$, are chosen to assess their influence on the placement. Among the 132 graphs examined, it is observed that 69 of them have nodes with a degree of 1. Remarkably, the optimal placement for each of these graphs, regardless of the number of controllers ($K = 2,3,4,5$), consistently included a node with degree 1. The analysis of the data reveals a significant trend suggesting a preference for nodes with lower degrees in the optimal placement of controllers. This trend remains consistent across all three graph classes for both high link operational probability and low link operational probability. Notably, the trend is particularly prominent within the graph class $\Omega(10,12)$, which exhibits a lower average degree when compared to the other two classes.



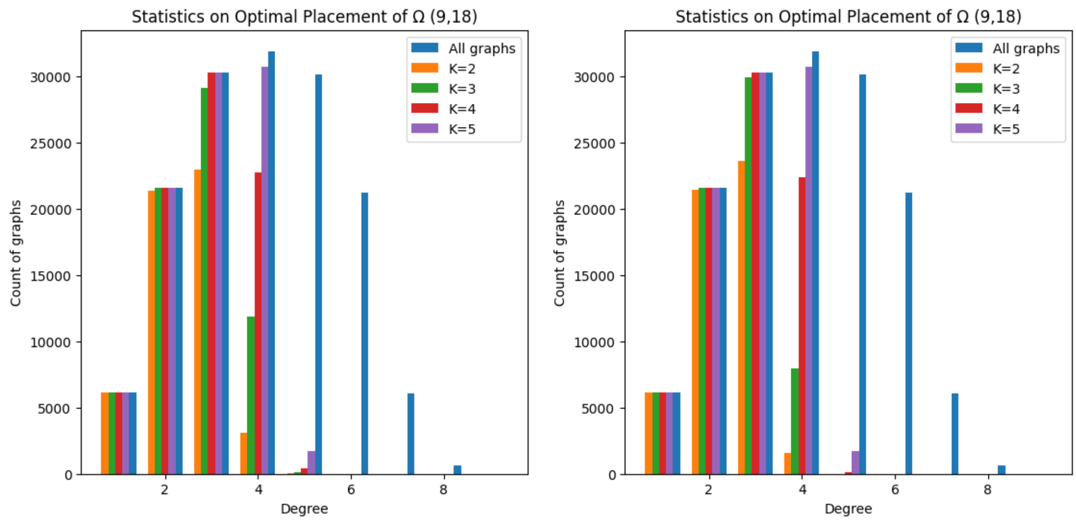
(a) $p = 0.1$ **(b)** $p = 0.99$

Figure 4.2: Statistics on optimal placement of graph class $\Omega(7,10)$. The blue bars represent the number of graphs containing nodes with a specific degree, while the remaining bars indicate the number of graphs where the optimal placement of K controllers includes at least one node with that degree.



(a) $p = 0.1$ **(b)** $p = 0.99$

Figure 4.3: Statistics on optimal placement of graph class $\Omega(10,12)$. The blue bars represent the number of graphs containing nodes with a specific degree, while the remaining bars indicate the number of graphs where the optimal placement of K controllers includes at least one node with that degree.



(a) $p = 0.1$ **(b)** $p = 0.99$

Figure 4.4: Statistics on optimal placement of graph class $\Omega(9,18)$. The blue bars represent the number of graphs containing nodes with a specific degree, while the remaining bars indicate the number of graphs where the optimal placement of K controllers includes at least one node with that degree.

Additionally, a degree-based random placement strategy is employed to assess the varying contributions of nodes with different degrees. The network under consideration is GtsCe network from the Topology Zoo, comprising 149 nodes. Specifically, this network consists of 12 nodes with a degree of 1, 80 nodes with a degree of 2, 35 nodes with a degree of 3, 10 nodes with a degree of 4, 8 nodes with a degree of 5, and 4 nodes with a degree higher than 5.

During the sampling process, a subset of 5 nodes is randomly selected from the node set, where each node has the same degree. The chosen nodes are then utilized to construct different placements based on the number of controllers K. For instance, the first node in the subset represents the placement when $K = 1$, the first two nodes in the subset represent the placement when $K = 2$, and so on. In this way, we simulate the sequential process of randomly placing 5 controllers on the nodes with a specific degree. The controller reachability of each placement is subsequently computed with the link operational probability of $p = 0.99$. To ensure reliable outcomes, the sampling procedure is repeated 100 times for each degree, and the resulting values are averaged to provide a representative assessment. The results are depicted in Fig. 4.5.

It is evident that placing controllers on nodes with a degree of 1 significantly enhances controller reachability. Conversely, placing controllers on nodes with a degree of 2 yields only marginal improvement in reachability. Placing controllers on nodes with higher degrees offers negligible improvement in controller reachability.

When placing controllers with optimized controller reachability, the nodes with a higher likelihood of being disconnected have a more pronounced impact on controller reachability compared to the nodes with a lower likelihood of disconnection. When edge failures happen, the probability of a node becoming disconnected exhibits an inverse relationship with its degree. Therefore, placing controllers at the nodes with low degrees can effectively improve the

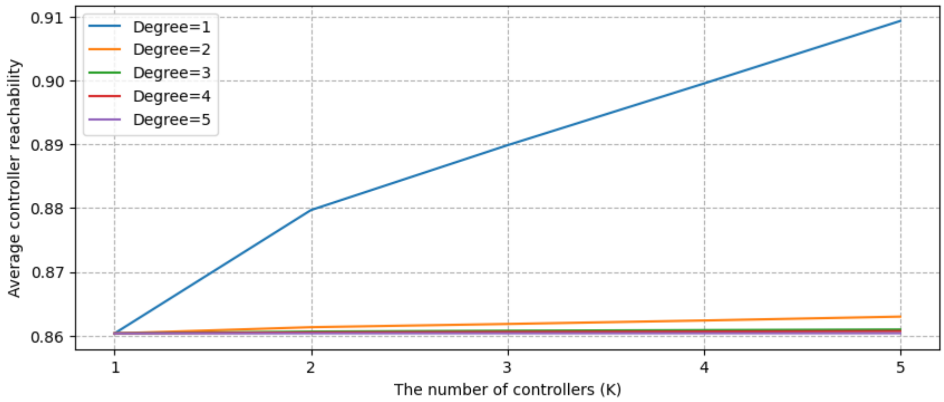
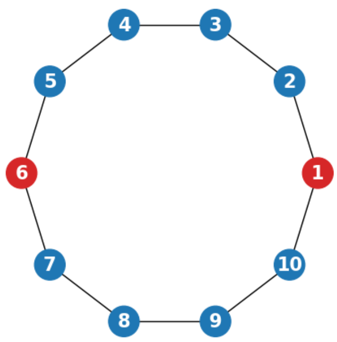


Figure 4.5: The average controller reachability when sequentially placing 5 controllers on nodes with a specific degree. The link operational probability $p = 0.99$. The x-axis denotes the number of controllers, the y-axis denotes the average controller reachability. Different curves represent different specific degrees.

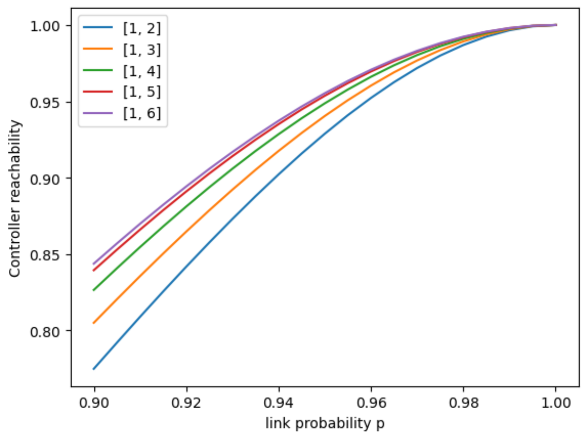
controller reachability.

Based on the above findings, it can be concluded that the degree of nodes is a significant graph metric that strongly influences controller reachability. The networks with a lot of degree 1 nodes exhibit a lower controller reachability. The nodes with degree of 1 are more responsible for the low controller reachability since they are likely to be disconnected. Therefore, the optimal placement tends to occur at low degree nodes.

4.2. Distance vs. Controller reachability



(a) Ring graph with $N = 10$

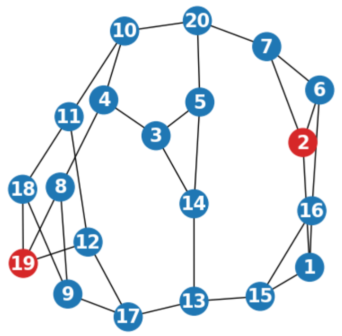


(b) Controller reachability of different placements ($K = 2$)

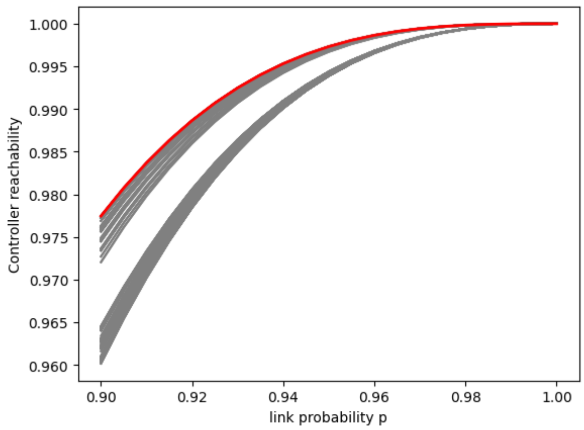
Figure 4.6: The placement of 2 controllers on a 10 nodes ring graph. Placing 2 controllers at the maximum distance, such as at (1,6), is the optimal placement.

The distance between two nodes is defined as the length of the shortest path between two nodes. In a study conducted by M.Chujyo and Y.Hayashi [5], it was discovered that adding links based on distance can enhance the robustness of a network. Additionally, our intuition suggests that the placement of controllers should be distributed throughout the network to prevent concentration in any specific region. This prompts the question: does controller reachability relate to distance?

To exclude the influence of degree, a ring graph is used to discover the relationship between distance and controller reachability. If we set $N = 10$, there are 5 different placements due to symmetry. The controller reachability with different placements is shown in Fig. 4.6. If two controllers are placed in this 10 nodes ring graph, the optimal placement is to place nodes such that the ring is evenly divided. Furthermore, we enumerate all possible placements of a random regular graph (N=20, every node has a degree of 3). The controller reachability with different placements is shown in Fig. 4.7. In this graph, the node pairs (2,19) and (6,19) both have the maximum distance with a shortest path length of 6. From the controller reachability curves for all placements, it is observed that these two placements are the optimal placements.



(a) Random regular graph with $N = 20$ and the node degree of 3



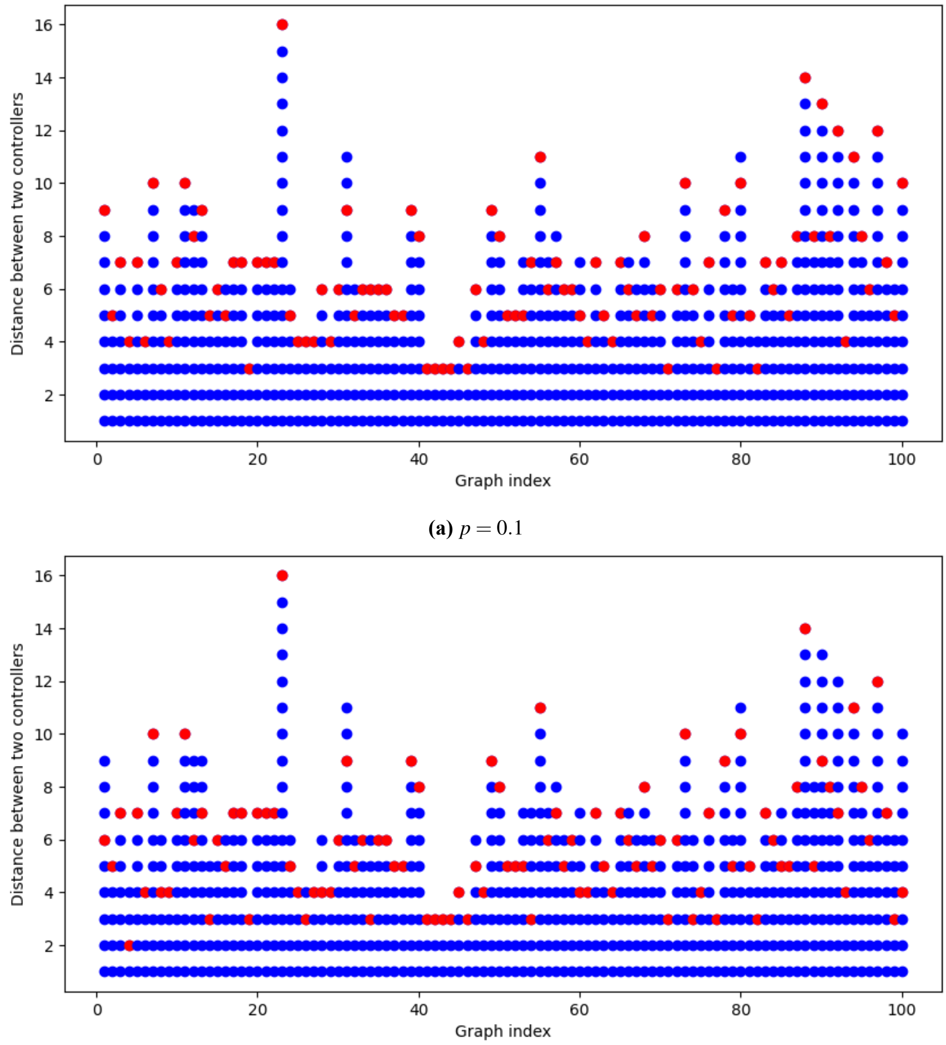
(b) Controller reachability of different placements ($K = 2$)

Figure 4.7: The placement of 2 controllers on a 20 nodes random regular graph. Placing 2 controllers at the maximum distance, such as at (2,19) or (6,19), is the optimal placement. The red curve represents the controller reachability of the optimal placement.

More graphs are tested to discover the relationship between distance and controller reachability.

Fig. 4.8 shows all possible distances between two controllers for 100 real-world graphs. Each column represents the data of a graph. Each data point represents a possible distance between two controllers. The red points represent the distance between two controllers when the optimal placements are employed. The position of the red data points within their respective columns allows us to assess the relationship between the optimal placement of controllers and the distance. If a red point is on the top of its column, it indicates that the optimal placement of this graph is the node pair with the maximum distance.

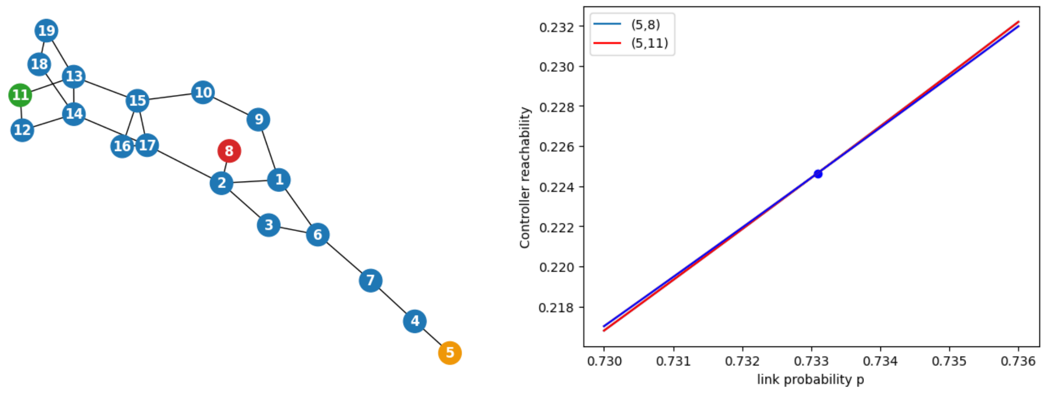
When the link operational probability $p = 0.1$, the optimal placements of 84 graphs are observed to be the node pairs with the maximum distance, while the optimal placements of 13 graphs correspond to the node pairs with the second maximum distance. When the link operational probability $p = 0.99$, the optimal placements of 67 graphs are observed to be the node pairs with the maximum distance, while the optimal placements of 14 graphs correspond



(a) $p = 0.1$

(b) $p = 0.99$

Figure 4.8: Statistics on optimal placement of 100 real-world graphs ($K = 2$). The x-axis denotes the index of graphs, the y-axis denotes the distance between two controllers. In this figure, each point represents a possible distance between two controllers. The red points represent the distance between two controllers when the optimal placements are employed.



(a) Aarnet

(b) Intersection of controller reachability

Figure 4.9: Intersection of controller reachability due to different optimal placement.

to the node pairs with the second maximum distance. The optimal placement never will consist of two neighboring nodes, i.e. with a distance of 1.

Based on the above findings, it can be concluded that the distance between nodes is a significant graph metric that influences the controller reachability. When placing two controllers, the optimal placement tends to occur at the node pair with maximum distance. This preference for maximizing distance helps ensure that no nodes within the network are too far away from both controllers. If we add more controllers based on this idea, the controllers can be placed at the nodes which are far away from existing controllers. The nodes that are far away from the controllers are likely to be disconnected due to link failures, which means they are more responsible for the low controller reachability than other nodes that are not likely to be disconnected. Therefore, placing controllers at the nodes far away from the existing controllers can efficiently improve the controller reachability.

4.3. Different optimal placement with different p

4.3.1. Optimal placement changes

Why the optimal placement changes?

In the case of network Aarnet with two controllers, it is observed that the optimal placement of controllers varies as the link operational probability p increases. To investigate this phenomenon, we enumerate all possible placements of two controllers and compare their controller reachability across a range of p values from 0 to 1. Remarkably, we find that the optimal placement changes at a specific threshold value of $p = 0.7331$, which is shown in Fig. 4.9. Below this threshold, the optimal placement is identified as (5, 11), while above this threshold, the optimal placement shifts to (5, 8). The underlying reasons for this change in the optimal placement will be further investigated and analysed.

Based on the topology of Aarnet, an interesting observation can be made regarding the choices of (5,8) and (5,11), which correspond to the metrics discussed in Section 4.1 and Sec-

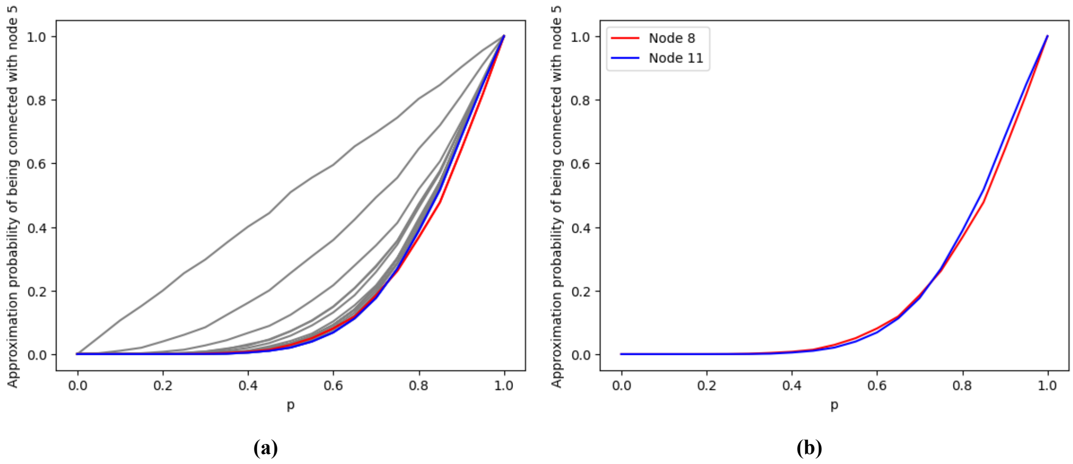


Figure 4.10: The approximate 2-terminal reliability between node 5 and other node. The x-axis denotes the link operational probability p, the y-axis denotes the approximate 2-terminal reliability. The red curve is the approximate 2-terminal reliability between node 5 and node 8. The blue curve is the approximate 2-terminal reliability between node 5 and node 11.

tion 4.2 respectively. It appears that the selection of the nodes for controller placement is influenced by the value of the link operational probability p. When p is relatively low, we select a pair of nodes with maximum distance between them. Conversely, when p is high, we select the nodes with a degree of one. It is worth noting that the controller located at node 5 remains fixed in both scenarios.

With 10000 runs of Monte Carlo simulations, Fig. 4.10 is obtained, which is the approximate 2-terminal reliability between node 5 and other nodes. We can see that node 11 and node 8 are the two nodes that have the lowest probability of being connected with node 5. Besides, the 2-terminal reliability curves of these two nodes also have an intersection at 0.72, which is very close to the intersection point of controller reachability in Fig. 4.9.

Based on the above findings, we can conclude that, as p changes, the most likely disconnected nodes changes due to the network topology. Therefore the optimal placement changes.

Does the change of optimal placement have a lot of impact?

The change of optimal placement brings a new problem. Can the optimal placement we obtain at p_1 also perform well at p_2?

Fig. 4.11 shows the controller reachability of all possible placements of Aarnet, where the red curve is the optimal placement at high p, and the blue curve is the optimal placement at low p. It is obvious that these two curves are either optimal or close-to-optimal in the range from 0 to 1. Especially, the red curve almost overlaps with the blue curve at the low p region, while there is a small gap between the two curves at the high p region.

Out of the 100 real-world networks sourced from the Topology Zoo, a total of 22 networks exhibit variations in their optimal placement as the value of p increases. Our research focuses specifically on these 22 networks, which aims to evaluate the performance of the optimal

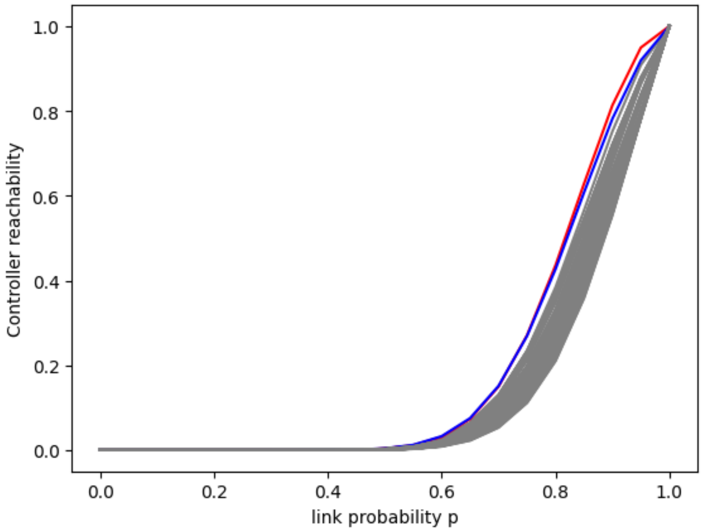


Figure 4.11: The controller reachability of all possible placements of Aarnet (K=2). The x-axis denotes the link operational probability p, the y-axis denotes the controller reachability. The red curve is the controller reachability when controllers are at nodes (5,8). The blue curve is the controller reachability when controllers are at nodes (5,11).

placement obtained at $p = 0.99$ when confronted with a different link operational probability, specifically $p = 0.1$. The results are shown in Fig. 4.12. After applying the max-min scaling ($x' = \frac{x - \min(x)}{\max(x) - \min(x)}$), the optimal placements obtained at $p = 0.1$ have a scaled controller reachability of 1, and the worst placements obtained at $p = 0.1$ have a scaled controller reachability of 0. The orange squares represent the average performance of all possible placements at $p = 0.1$, while the blue points represent the performance if we place controllers on the optimal placement obtained at $p = 0.99$ but calculate the controller reachability with $p = 0.1$. The position of the blue points within their respective columns allows us to assess the impact of the change in the optimal placement. If blue points are close to 1, the optimal placements we obtain at $p = 0.99$ also perform well at $p = 0.1$. The result shows that the optimal placements of 16 graphs obtained at $p = 0.99$ can achieve 80% performance of the optimal placement obtained at $p = 0.1$ when confronted with the link probability $p = 0.1$. When we employ the optimal placement obtained at a high link probability p, it is possible to serve as a close-to-optimal solution at a low link probability p. The change of optimal placement does not have a lot of impact on most networks. Besides, the optimal placement we acquire when the operational probability p is high is more valuable, as real-world networks in general exhibit high link operational probabilities.

4.3.2. More networks with changing optimal placement

Although it is also possible to observe a change in the optimal placement when K is larger than 2, here we only consider the simplest case where $K = 2$. From the graph class $\Omega(7, 10)$, 5 graphs with such properties are found. All of them have one intersection of controller reachability. The yellow node is the node that is always selected when placing two controllers, the red node is the other selected node when p is high, and the green node is the other selected node when p is low. The values of the intersection points are shown in Table 4.2.

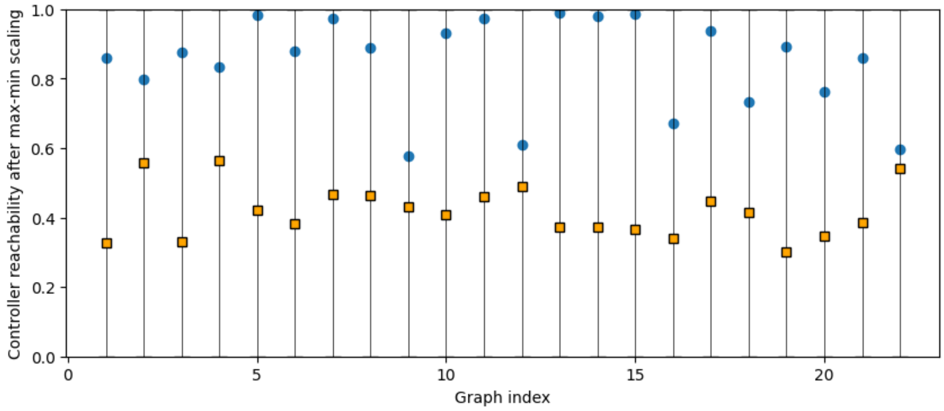


Figure 4.12: The assessment of the performance of the optimal placement obtained at $p = 0.99$ when confronted with a different link operational probability $p = 0.1$. In this figure, the x-axis denotes the index of the graph, and the y-axis denotes the controller reachability at $p = 0.1$ after max-min scaling. The orange squares represent the average scaled controller reachability of all possible placements, and the blue points represent the scaled controller reachability if we employ the optimal placement obtained at $p = 0.99$.

Graph *a*	*p*=0.38609673
Graph *b*	*p*=0.26215674
Graph *c*	*p*=0.26215674
Graph *d*	*p*=0.56222482
Graph *e*	*p*=0.29289322

Table 4.2: Intersection point of controller reachability.

From these 5 graphs, we can observe that the network topology determines whether there is an intersection. For the graph a, b, and d, if the link operational probability p is high, the second controller is at the node which is closer to the first controller but has a lower degree, if link operational probability p is low, the second controller is at the node which is far away from the first controller but has a higher degree. For the graph c, the distance between the two controllers is the same as the placement changes, but the distance between node 5/7 and the controllers varies. For the graph e, the distance between node 4/5/7 and the controllers varies as the placement changes. Additionally, this is the only graph whose node degree of different optimal placements remains unchanged.

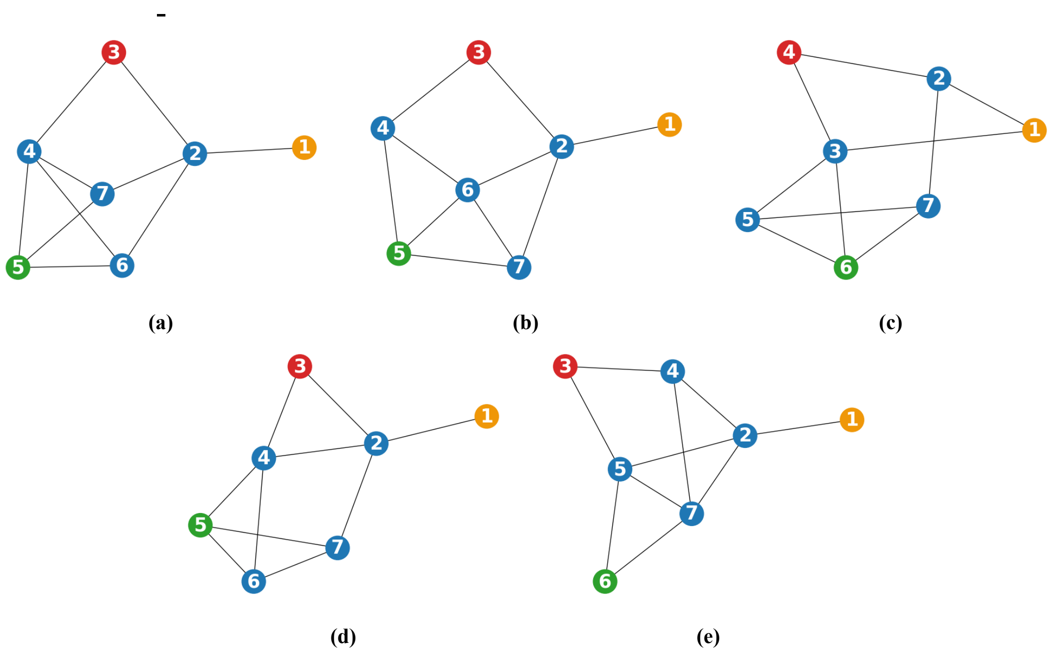


Figure 4.13: 5 graphs with different optimal placements with two controllers at different p. The yellow node is the node that is always selected when placing two controllers, the red node is the other selected node when the link operational probability p is high, and the green node is the other selected node when the link operational probability p is low.

5

Controller placement strategies

In this chapter, strategies to place K controllers in a graph are introduced. The first strategy is based on two graph metrics, distance and degree, which we have found to have a great impact on the controller reachability in Chapter 4. The second strategy is the greedy algorithm. The third kind of strategies are about genetic algorithms, which are classical GA and heuristic GA.

5.1. Controller placement strategy based on degree and distance

Degree and distance are found to have a great impact on the controller reachability when placing multiple controllers in Chapter 4. We propose an effective way to place K controllers based on degree and distance. The main idea of this strategy is to group the nodes in different sets based on degree. Nodes with lower degrees will be selected with higher priority to place controllers. If choosing among nodes with the same degree, the algorithm will attempt to maximize the distance between the controllers (minimize the distance between controllers and nodes).

In the graph G, we denote the lowest degree as d_1, the second lowest degree as d_2, the i-th lowest degree as d_i, and the highest degree as d_M ($d_1 = d_{\min} < d_2 \cdots < d_M = d_{\max}$). n_1 nodes with degree d_1 are considered as set $S(d_1)$, n_2 nodes with degree d_2 are considered as set $S(d_2)$, n_i nodes with degree d_i are considered as set $S(d_i)$. Besides, we use $d_0 = 0$, $n_0 = 0$, and $S(d_0) = \emptyset$ to indicate that no node in the graph has a degree of 0. Consequently, we obtain M non-empty sets of nodes that do not overlap and collectively cover every node in graph G. We aim to find the node set $S(d_k)$ such that $\sum_{i=0}^{k-1} n_i < K \leq \sum_{i=0}^{k} n_i$. The node sets with degree lower than d_k is defined as the initial existing controllers set $S_C = \bigcup_{i=0}^{k-1} S(d_i)$, the controllers are placed on every node in this set due to their low degree. The number of remaining controllers is defined as $K' = K - \sum_{i=0}^{k-1} n_i$. The nodes in $S(d_k)$ are considered as potential locations for placing K' controllers according to the distance.

If $K \leq n_1$, $S_C = \emptyset$, the nodes in $S(d_1)$ are considered as potential locations to place $K' = K$

controllers. For each node in set $S(d_1)$, we compute the sum of its distances to other nodes in G. The node with the highest sum of distances is selected as the location for the first controller. From the placement of the second controller onwards, we select the node that has the longest distance to the existing controllers as the location for the next controller.

If $K > n_1$, the node set that we are going to place K' controllers is determined as follows: If $n_1 < K \leq n_1 + n_2$, $S_C = S(d_1)$, the nodes in $S(d_2)$ are considered as potential locations for placing $K' = K - n_1$ controllers. If $n_1 + n_2 < K \leq n_1 + n_2 + n_3$, $S_C = S(d_1) \cup S(d_2)$, the nodes in $S(d_3)$ are considered as potential locations for placing $K' = K - n_1 - n_2$ controllers, etc. Using this method, we can identify the locations of $K - K'$ controllers based on the node degree, and then place the remaining K' controllers based on the distance within a smaller set of nodes. Within this set, the distances between each node and the existing controllers are found. The node with the longest distance to the existing controllers is selected as the location for the next controller. This process is repeated until K nodes are identified.

Algorithm 2 Algorithm based on degree and distance

Input: network G, controllers' number K

Output: Set S_C

Define set S_C as the set of nodes with controllers
Define d_i as the i-th lowest degree
Define n_i as the number of nodes with degree d_i
Define $S(d_i)$ as the set of nodes with degree d_i
Define $n_0 = 0$, $S(d_0) = \emptyset$
Define $distance(u, v)$ as the shortest path length between u and v
Find the set $S(d_k)$ such that $\sum_{i=0}^{k-1} n_i < K \leq \sum_{i=0}^{k} n_i$
$S_C = \bigcup_{i=0}^{k-1} S(d_i)$
$K' = K - \sum_{i=0}^{k-1} n_i$
if $K < n_1$ **then**
    for $v \in S(d_1)$ **do**
        $SumDistance(v) = \sum_{u \in G, u \neq v}(distance(u, v))$
    end for
    Add the node v with the highest $SumDistance(v)$ into S_C
    $K' = K' - 1$
end if
while $K' > 0$ **do**
    for $v \in S(d_k)$ **do**
        $D(v) = \min(distance(u, v))$ where $u \in S_C$
    end for
    Add the node v with the highest $D(v)$ into S_C
    $K' = K' - 1$
end while
return Set S_C

5.2. Greedy algorithm

The greedy algorithm is an algorithm that always takes the best local solution while finding an answer. The main idea of this approach is to make a decision based on the current information and do not take the future into consideration. The greedy algorithm is known for its ability to find near-optimal solutions for certain optimization problems [31].

For the controller placement problem, the greedy algorithm will place controllers one by one. When placing a single controller on the network, the controller reachability remains the same regardless of its location. However, the initial placement has a significant impact on determining the subsequent placement of the second controller. Hence, it becomes necessary to investigate how to select the location for the first controller. In this study, we consider four different approaches in our attempt to determine a nice starting point.

- Randomly choose a node
- Randomly choose a node with the lowest degree
- Use algorithm 2 to determine the first node
- Enumerate all possible $K = 2$ placements, and choose the optimal solution as the first and second controllers

Not surprisingly, randomly choosing a node performs not well when placing the first few controllers, but it gradually approaches the performance of other methods. Enumeration over 2 placements performs the best, but it is so time consuming. Method 2 and 3 will both choose a node with the lowest degree, but method 3 also takes distance into consideration. So we finally decide to use algorithm 2 to determine the first node. This process only takes a second, which is almost negligible compared with the calculation time of the whole greedy algorithm.

Starting from the second controller, the greedy algorithm will go through every possible node (the nodes without placed controllers) and choose the node that brings the highest controller reachability improvement. This process is repeated until K controllers are placed.

5.3. Genetic algorithm

In order to address the controller placement problem, classic GA and heuristic GA [1] are implemented. In the first part, the introduction to the genetic algorithm is presented to help understanding the role of different operators. Two GA models are explained afterwards.

5.3.1. Introduction to genetic algorithm

The genetic algorithm is a well-known meta-heuristic algorithm inspired by the biological evolution process. At each generation, it selects individuals from the current population to be parents and uses them to produce children and a new population. The best individual in the population is gradually moving towards the optimal solution. The chromosome representation,

Algorithm 3 Greedy algorithm

Input: network G, controllers' number K
Output: Set S_C

1: Define set S_C as the set of nodes with controllers
2: The first controller is chosen based on Algorithm 2 and added into set S_C
3: $K = K - 1$
4: **while** $K > 0$ **do**
5:     **for** node $v \notin S_C$ **do**
6:         Compute the controller reachability $P_{cr}(v)$ if controllers are placed at node v and nodes in S_C
7:     **end for**
8:     Add node v with the highest $P_{cr}(v)$ into set S_C
9:     $K = K - 1$
10: **end while**
11: **return** Set S_C

selection, crossover, mutation, and fitness function computation are the key elements of the genetic algorithm [13]. Selection, mutation, and crossover are also called biological-inspired operators.

- Selection: At each iteration, some individuals of the old population are selected to reproduce a new population. The fitness function measures the quality of the individuals and gives fitness value. The selection operator selects individuals on the basis of their fitness value. Usually, the individuals with higher fitness values are likely to be selected, and the individuals with lower fitness values are unlikely to be selected. Roulette wheel, rank, tournament, Boltzmann, and stochastic universal sampling are common selection methods.
- Crossover: The crossover operator is used to generate new solutions from two parents. The most simple crossover technique is single point crossover, which selects a random crossover point and swaps the information of two parents behind that crossover point. Methods derived from this are two-point and k-point crossover. Other common crossover techniques are uniform, order, and partially matched crossover.
- Mutation: The mutation operator is used to maintain the genetic diversity of the population. It can avoid the algorithm converging to a locally optimal solution. Inversion, bit string mutation, and displacement are common mutation methods.

5.3.2. Encoding and initializing in controller placement problem

Encoding

The individual is represented as a sequence of K genes where each gene corresponds to the node that we choose to place a controller.

Initializing the population

The initial population is generated with the method used in [1], which ensures that each gene has a similar frequency of occurrence in the population. Specifically, the occurrence frequency f is determined by the following equation,

$$f = max\left\{2, \left\lceil \frac{N}{100} \cdot \frac{ln(s)}{d} \right\rceil \right\} \tag{5.1}$$

where N is the number of nodes, K is the number of controllers, $s = \binom{N}{K}$ is the number of possible solutions, $d = \lceil N/K \rceil$ is the rounded-up density of the problem. This equation makes sure that each gene will be present at least twice. The initial population size is $f \cdot d$, which can be considered as f sets of solutions with each set comprising d solutions.

After determining the frequency and the population size, nodes are assigned to each solution. For the first set of d solutions, nodes $1, 2, \cdots K$ are assigned to the first solution, nodes $K+1, K+2, \cdots 2K$ are assigned to the second solution. Repeat this process until all nodes are assigned to solutions and d solutions are obtained. If N/K is not an integer, random non-repeat genes are selected to fill the last solution. For the second set of d solutions, nodes $1, 3, \cdots 2K-1$ are assigned to the first solution, nodes $2K+1, 2K+3, \cdots 4K-1$ are assigned to the second solution, etc. This process is repeated until f sets of solutions are obtained. By adjusting the increment of nodes when generating different sets of solutions, we ensure that there are no repeated solutions in the initial population.

The fitness value for this problem is the controller reachability with respect to K controllers.

5.3.3. Classic GA

Selection operator

Tournament selection is used in classic GA. Compared with the roulette wheel selection, tournament selection converges faster and is easier to implement [36]. Tournament selection randomly chooses a few individuals from the population and runs tournaments. Only the fittest individual will be chosen and continue with the crossover. If the tournament size is larger, weak individuals have a smaller chance of getting selected since it has to compete with more individuals. In this thesis, a binary tournament is used to ensure the diversity of the population. An example of binary tournament selection is shown in Fig. 5.1.

Crossover operator

Partial mapped crossover (PMX) is the most commonly used crossover operator for permutation of encoded chromosomes. It can generate two offspring without duplicate genes and performs better than most of the other crossover operators [13]. The crossover rate is $p_c = 0.8$. The steps of partial mapped crossover [6] are as follows,

- Cut two substrings of the same size on each parent at the same position

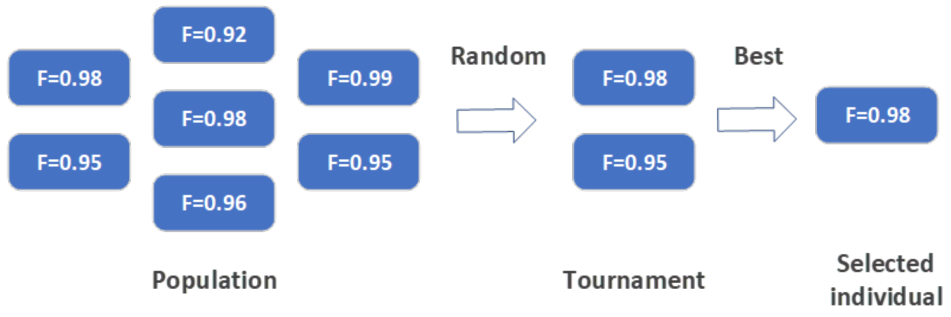


Figure 5.1: Example of binary tournament selection. In this example, two individuals are randomly chosen from the population to run the tournament. Comparing their fitness values, the individual with highest fitness value is selected.

- Exchange two substrings
- Determine the mapping relationship based on selected substrings
- Use the mapping relationship to replace the duplicate genes

An example of PMX is shown in Fig. 5.2.

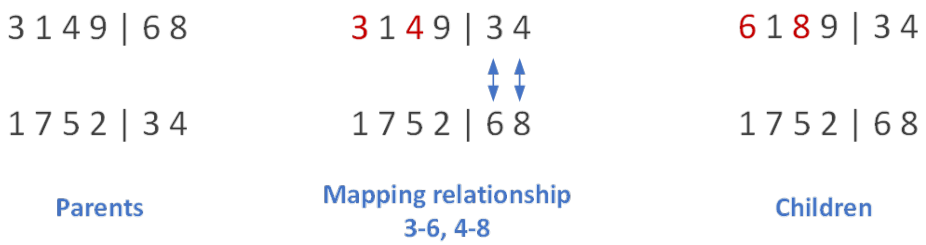


Figure 5.2: Example of partial mapped crossover. Two parents exchange the substrings with the same size and two draft offsprings with duplicate genes are obtained. The mapping relationship between gene 3 and gene 6, as well as between gene 4 and gene 8 are observed. The duplicate genes are replaced according to the mapping relationship to obtain two children.

Mutation operator

The mutation in this algorithm is very simple. This operator will randomly choose a gene and replace it with a gene that is not present in this individual to ensure the offspring has no duplicate genes. The mutation rate is $p_m = 0.1$.

5.3.4. Heuristic GA

This method is based on the genetic algorithm proposed by O.Alp [1] to solve the facility location problem, which selects the location of K facilities to serve N demand points with minimal total travel time. Different from traditional crossover which uses two individuals and exchanges genes to reproduce two offsprings, the heuristic GA firstly takes a union of genes of two parents to obtain a draft solution. Then greedy deletion heuristic is used to decrease

Algorithm 4 Classic genetic algorithm

Input: network G, controllers' number K, max number of iteration MAX
Output: Set S_C

```
Define set S_C as the set of nodes with controllers
Determine the population size P
Initialize the population
Compute the fitness value of each individual
Set iteration counter t = 0
while t < MAX do
    Select P individuals from the population using tournament selection.
    Apply crossover on P/2 pairs of individuals with crossover probability.
    Apply mutation on the offspring with mutation probability.
    New population with size P is generated
    t = t + 1
end while
Add nodes in the best solution at the last iteration to set S_C
return Set S_C
```

the number of genes until the solution has K genes. Also, this algorithm does not produce a new population at each iteration, but continues to update the initial population. The heuristic genetic algorithm shows good performance in many optimization problems. It is applied to the controller placement problem to compare with the performance of classic GA.

Crossover operator

Two individuals are randomly selected as parents. A temporary offspring can be generated by combining genes of two parents. This offspring can not be passed to the next step since it contains $2K$ genes. The redundant genes need to be removed to generate an offspring with K genes. The removal follows two rules:

- The gene that is present twice is kept.
- The gene that contributes the least to the improvement of controller reachability is removed.

One example is shown in Fig. 5.3.

This kind of crossover increases the time demands, but also improves the quality of the offspring.

Update population

Every time a new individual is generated by crossover, it is compared with other members in the populations. If the generated individual is not identical to the existing members and its fitness value is better than the worst fitness value in the population, then the newly generated

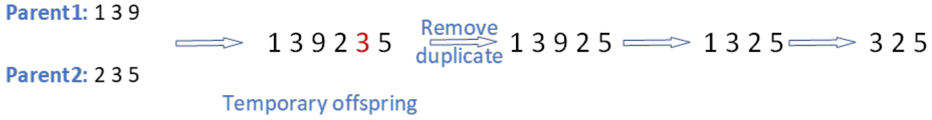


Figure 5.3: Example of greedy heuristic crossover. Two parents (1 3 9) and (2 3 5) are selected to generate a new solution. By simply combining them, a temporary offspring (1 3 9 2 3 5) is obtained. Firstly, the duplicate gene is removed, and node 3 is marked as kept gene according to rule 1 when removing genes. Then, the fitness value of (1 3 9 2), (1 3 9 5), (1 3 2 5), and (3 9 2 5) is computed. Among these 4 solutions, (1 3 2 5) has the highest fitness value, which means that the removal of node 9 influences the least on overall performance. Therefore, node 9 is removed at this step. Similarly, by comparing the fitness value of (1 3 2), (1 3 5), and (3 2 5), the node 1 is removed. Finally, a new offspring with length K is generated.

individual will replace the worst one. With this kind of replacement, the average quality of the entire population is improved at each iteration. Also, the population always has good diversity due to the non-duplicate individuals.

The best fitness value of each iteration is recorded to determine the termination. GA will terminate if the best fitness value in N successive iterations is unchanged.

Algorithm 5 Heuristic genetic algorithm

Input: network G, controllers' number K
Output: Set S_C
Define set S_C as the set of nodes with controllers
Determine the population size P
Initialize the population
Compute the fitness value of each individual and store the best/worst value
Set iteration counter $t = 0$
while $t < n$ **do**
  Select two individuals from population randomly
  Apply crossover and generate one offspring
  if offspring not in population **then**
    compute the fitness value of offspring
    if fitness value > worst value **then**
      Update population
    else
      t=t+1
    end if
  else
    t=t+1
  end if
end while
Add nodes in the best solution to set S_C
return Set S_C

6

Result and analysis

In this chapter, the quality of different placements is analysed based on real-world graphs from the Topology Zoo. Section 6.1 presents the analysis of Internet2 OS3E. Section 6.2 presents the analysis of more real-world graphs. In section 6.3, different controller placement strategies are compared for the Erdős–Rényi model, the Barabási–Albert model, and the real-world graphs.

6.1. Analysis of Internet2 OS3E

Internet2 OS3E is a topology that was used in many controller placement problems since Heller *et al.* first used it for analysis [10]. Therefore, OS3E network is chosen as an example in this section. In Section 6.1.1, the optimal placements of OS3E when $K = 2,3,4,5$ are presented. In Section 6.1.2, a comparison is conducted to assess the improvement of controller reachability achieved by adding a controller under various link operational probabilities p.

6.1.1. How does placement affect controller reachability

OS3E is a network with 34 nodes and 42 links. There are $\binom{34}{2}$ ways to place 2 controllers in this network. The controller reachability polynomial of each placement is computed by the path decomposition algorithm. The optimal placement is (9,19) and the worst placement is (1,29) as shown in Fig. 6.1a. If we focus on link operational probability p that ranges from 0.99 to 1, the controller reachability of different placements is shown in Fig. 6.1b and error bars are shown in Fig. 6.1c.

Not surprisingly, the controller reachability of different placements varies widely. For instance, when $p = 0.99$, the optimal controller reachability is 0.99686, the worst network controller reachability is 0.97691, the average controller reachability is 0.97843. The optimal value is far away from the average value for every link operational probability p, which indicates that the network performance can be effectively improved if we can find a close-to-optimal placement. Similar results are obtained when placing 3, 4, and 5 controllers (Fig. 6.2,

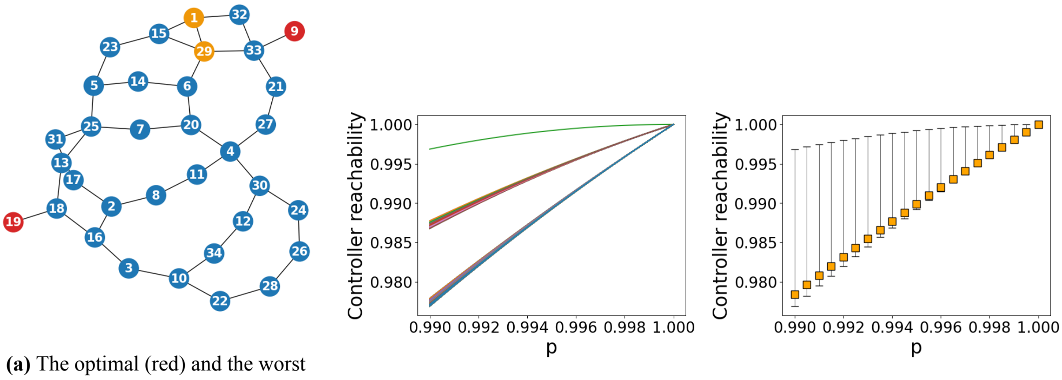
Fig. 6.3, Fig. 6.4).

The optimal placements and the worst placements of OS3E for different K at $p = 0.99$ are shown in Table 6.1

K	Optimal placement	Worst placement
2	9 19	1 29
3	9 19 28	1 15 29
4	9 19 12 28	4 6 20 25
5	9 8 19 28 34	4 5 6 20 25

Table 6.1: The optimal placement and the worst placement of OS3E for $K = 2,3,4,5$ at $p = 0.99$.

Besides, we can observe that the controller reachability curves for all possible K placements can be separated into three groups, where the highest performance group contains all placements that have both node 9 and 19, the middle performance group contains all placements that have either node 9 or node 19, the lowest performance group contains all placements that do not have node 9 or node 19. Node 9 and node 19 are the only two nodes that have degree 1 in OS3E, which reflects the relationship between node degree and controller reachability discussed Chapter 4.

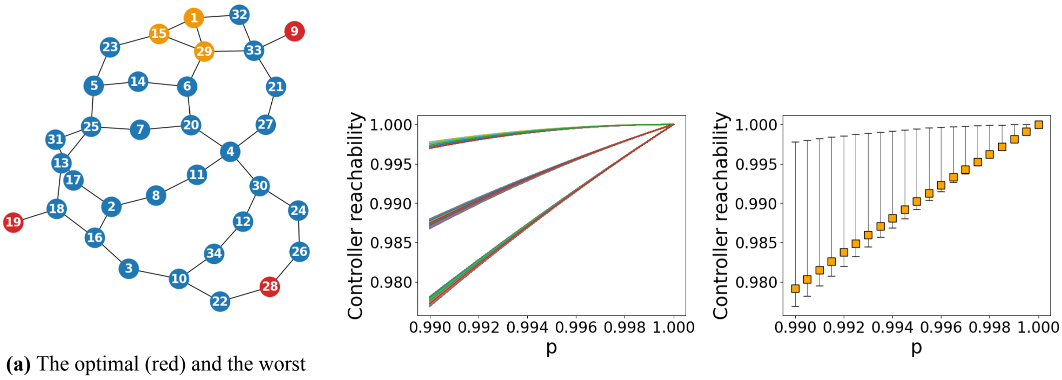


(a) The optimal (red) and the worst (yellow) placements at $p = 0.99$ **(b)** Controller reachability curves **(c)** Error bars of controller reachability

Figure 6.1: Controller reachability of OS3E (K=2). (a) shows the optimal and the worst placements when placing 2 controllers. (b) shows the controller reachability curves for all possible placements, where each curve represents a kind of placement. (c) shows the max-min error bars of controller reachability with all possible placements when placing 2 controllers.

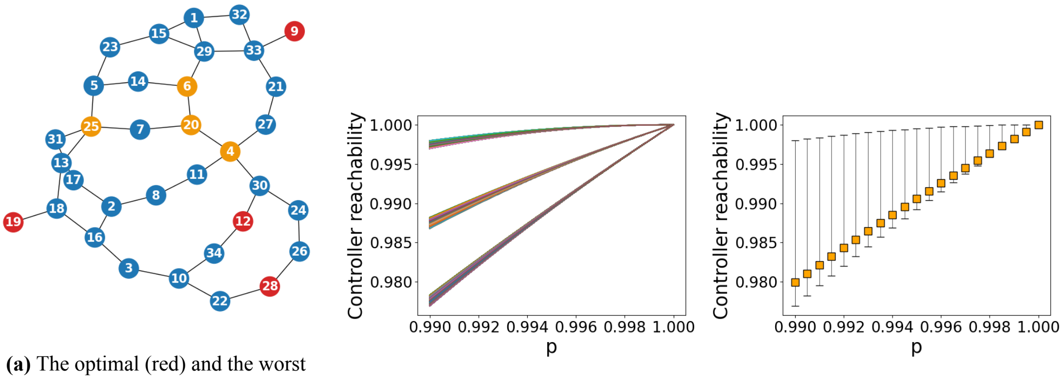
6.1.2. How many controllers are needed

Another important question is how many controllers are needed. We want to know how much improvement can be obtained by adding a controller. To do this, optimal K placement is employed. Since it is a problem that also depends on link probability, $p = 0.9, 0.99, 0.999$ are chosen to show the differences. In Fig. 6.5, the controller reachability with $K = 1,2,3,4,5$ is shown.



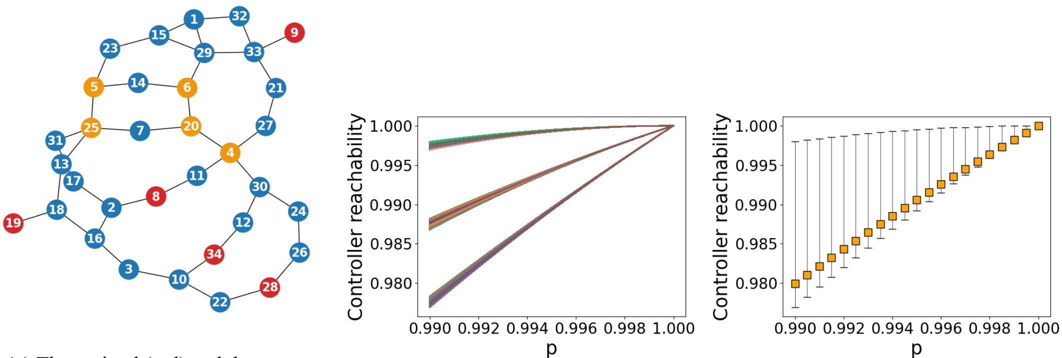
(a) The optimal (red) and the worst (yellow) placements at $p = 0.99$ **(b)** Controller reachability curves **(c)** Error bars of controller reachability

Figure 6.2: Controller reachability of OS3E (K=3). (a) shows the optimal and the worst placements when placing 3 controllers. (b) shows the controller reachability curves for all possible placements, where each curve represents a kind of placement. (c) shows the max-min error bars of controller reachability with all possible placements when placing 3 controllers.



(a) The optimal (red) and the worst (yellow) placements at $p = 0.99$ **(b)** Controller reachability curves **(c)** Error bars of controller reachability

Figure 6.3: Controller reachability of OS3E (K=4). (a) shows the optimal and the worst placements when placing 4 controllers. (b) shows the controller reachability curves for all possible placements, where each curve represents a kind of placement. (c) shows the max-min error bars of controller reachability with all possible placements when placing 4 controllers.



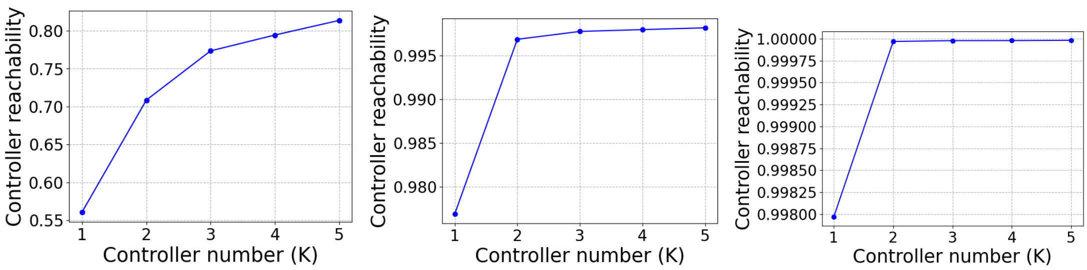
(a) The optimal (red) and the worst (yellow) placements at $p = 0.99$ **(b)** Controller reachability curves **(c)** Error bars of controller reachability

Figure 6.4: Controller reachability of OS3E (K=5). (a) shows the optimal and the worst placements when placing 5 controllers. (b) shows the controller reachability curves for all possible placements, where each curve represents a kind of placement. (c) shows the max-min error bars of controller reachability with all possible placements when placing 5 controllers.

When $p = 0.9$, even if we add controllers until $K = 5$, the improvement at each step is still large. When $p = 0.99$, two controllers ensure the controller reachability larger than 0.995 and we can observe slight improvement if K keeps increasing. When $p = 0.999$, the controller reachability with a single controller is already larger than 0.995. We can barely observe improvement if continuing adding the controllers after $K = 2$.

If we want to place controllers in OS3E such that the controller reachability is larger than 0.995, 5 controllers are still not enough when $p = 0.9$, 2 controllers meet the requirement when $p = 0.99$, a single controller is needed when $p = 0.999$. It is consistent with the intuition that if p is higher, less controllers are needed to meet a certain controller reachability.

For specific networks, the number of needed controllers should be considered according to the actual situation.



(a) $p = 0.9$ **(b)** $p = 0.99$ **(c)** $p = 0.999$

Figure 6.5: The controller reachability of OS3E with $K = 1,2,3,4,5$ at $p = 0.9, 0.99, 0.999$. The x-axis denotes the number of controllers, the y-axis denotes the controller reachability.

6.2. Analysis of more topologies

In this section, the analysis is based on the optimal placements ($K = 2,3,4,5$, $p = 0.99$) of 100 small size graphs ($11 \leq n \leq 30$) from the Topology Zoo.

6.2.1. How does placement affect controller reachability

The controller reachability of all possible K controller placements is computed. The maximum, minimum, and average values are plotted in the form of error bars in Fig. 6.6. From this figure, we can see that only a few networks can achieve close-to-optimal controller reachability when placing controllers randomly. For most networks, the average values are far away from optimal.

Min-max scaling is applied to overall data to quantify the performance of random placement. Min-max scaling is a normalization technique that scales the data values to a range between 0 and 1, using the minimum and maximum values of the original data. This process can be expressed as $x' = \frac{x - \min(x)}{\max(x) - \min(x)}$. The range (0, 1) is partitioned into four subintervals: (0, 0.25), (0.25, 0.5), (0.5, 0.75), and (0.75, 1). Table 6.2 presents the number of networks categorized based on their scaled average controller reachability within each respective interval.

K	(0, 0.25)	(0.25, 0.5)	(0.5, 0.75)	(0.75, 1)
2	31	52	11	6
3	25	48	21	6
4	17	50	27	6
5	8	54	31	7

Table 6.2: The number of networks in each interval.

We can see that more networks are in the intervals (0.5, 0.75) and (0.75,1) as the number of controllers K is larger. However, for most networks, the controller reachability of the random placement is still far away from the controller reachability of the optimal placement. Therefore, for most networks, it is necessary to use some strategies to place controllers such that close-to-optimal placement is achieved.

6.2.2. How many controllers are needed

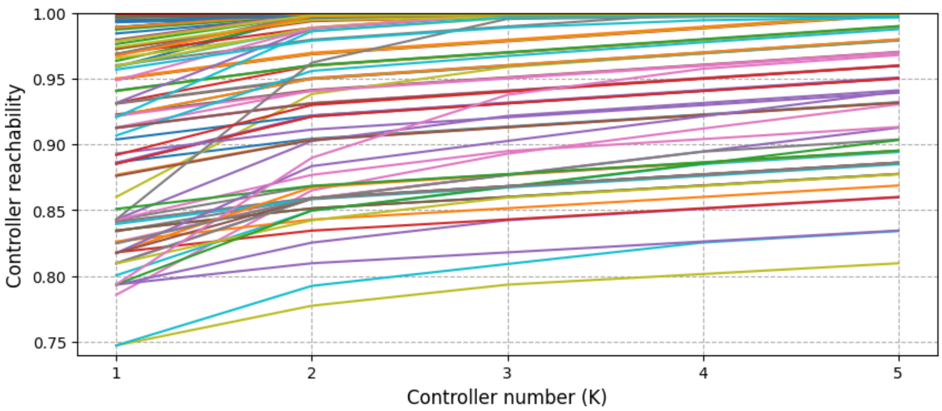


Figure 6.7: The optimal controller reachability of 100 small real-world graphs. In this figure, the x-axis denotes the number of controllers, the y-axis denotes the controller reachability. Each curve represents the optimal controller reachability of a graph with $K = 1,2,3,4,5$ at $p = 0.99$.

If K controllers are always placed at its optimal placement, we can find out the minimum number of needed controllers. Fig. 6.7 plots the optimal controller reachability of 100 small real-world graphs with $K = 1,2,3,4,5$ at $p = 0.99$. Although the networks' sizes are very close, the controller reachability of different graphs with a single controller varies in a large range from 0.75 to 0.9995.

If we continue to use 0.995 as the controller reachability requirement, 6 networks meet the requirement with a single controller, 26 networks meet the requirement with 2 placed controllers, 10 networks meet the requirement with 3 placed controllers, 1 network meets the requirement with 4 placed controllers, 4 networks meet the requirement with 5 placed controllers, and 53 networks cannot meet the 0.995 requirement, even with 5 placed controllers.

Upon examining the network topologies exhibiting high and low controller reachability, a

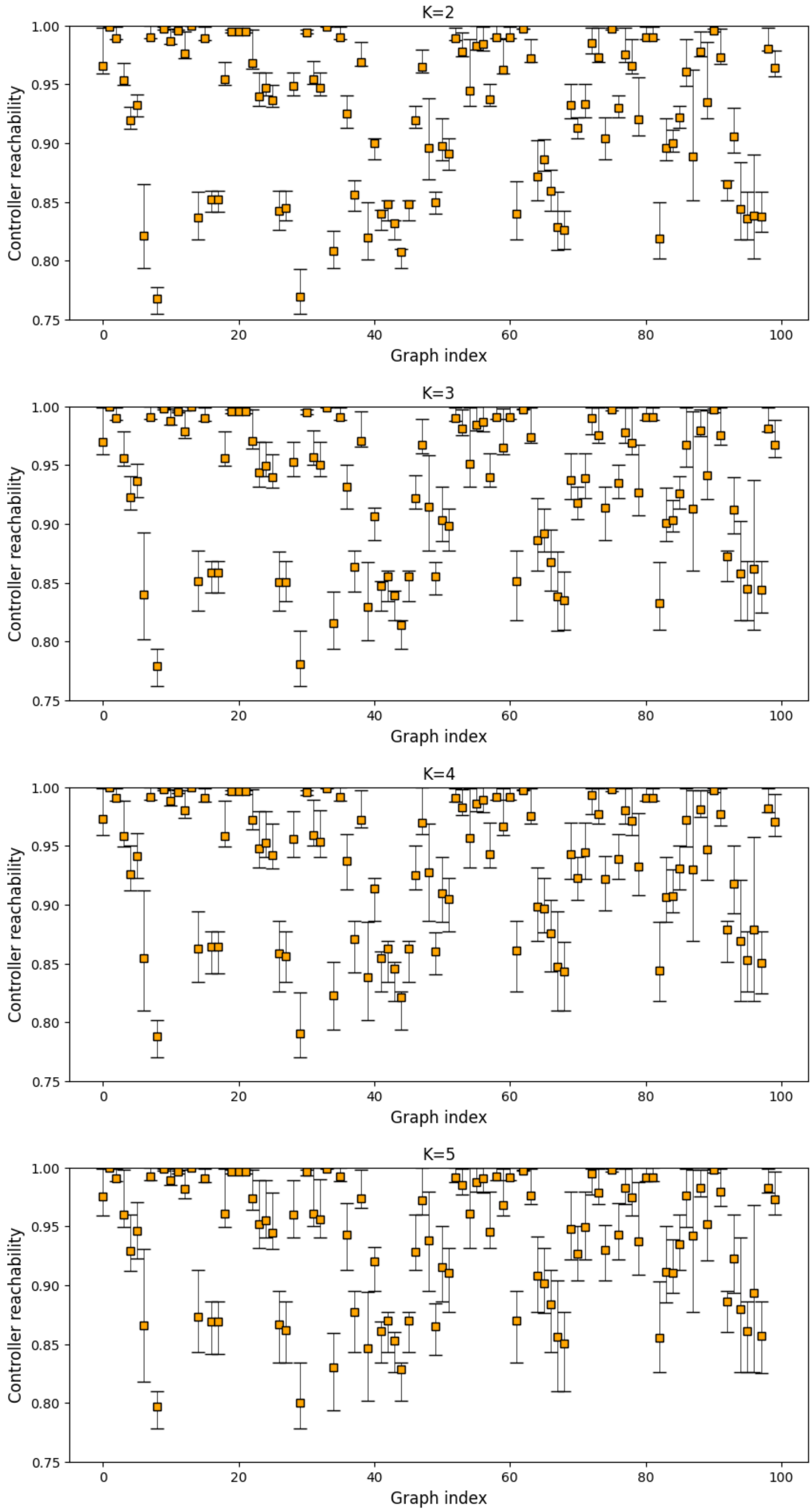


Figure 6.6: Controller reachability error bars of 100 small real-world graphs. In these figures, the x-axis denotes the index of graph, the y-axis denotes the controller reachability. Each error bar represents the maximim, minimum, and average controller reachability of all possible placements of K controllers.

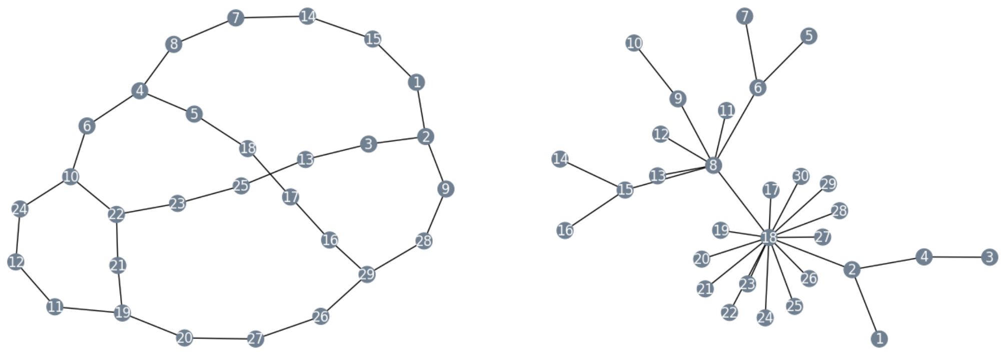


Figure 6.8: Two example graphs. The first one exhibits the highest controller reachability among 100 small real-world graphs. The second one exhibits the lowest controller reachability among 100 small real-world graphs. The controller reachability is computed when $K = 1$ and $p = 0.99$.

notable relationship emerges between controller reachability and the network topology. Specifically, networks lacking degree 1 nodes demonstrate remarkably high controller reachability, whereas networks characterized by star and tree topologies featuring a substantial number of degree 1 nodes exhibit notably low controller reachability. This observation underscores the significant influence of network topology on controller reachability.

The topology exhibiting the highest controller reachability and the topology exhibiting the lowest controller reachability among 100 small real-world graphs are shown in Fig. 6.8.

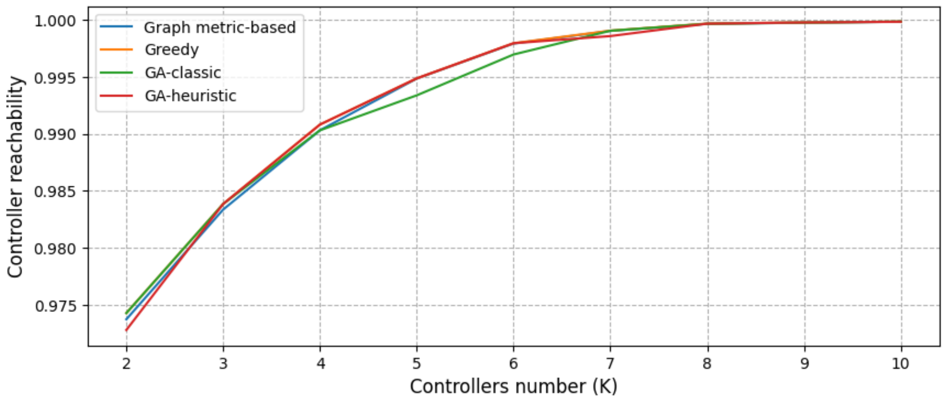
The number of needed controllers should be determined according to the targeted controller reachability, the link probability p as well as the network topology. We cannot conclude a number that applies to all networks. However, we can observe that the deployment of multi-controllers obviously improves the controller reachability for most networks. Especially when $K = 2$, all the curves have obvious inflection points, which indicates that the deployment of two controllers already improves controller reachability a lot.

6.3. Comparison of different placement strategies

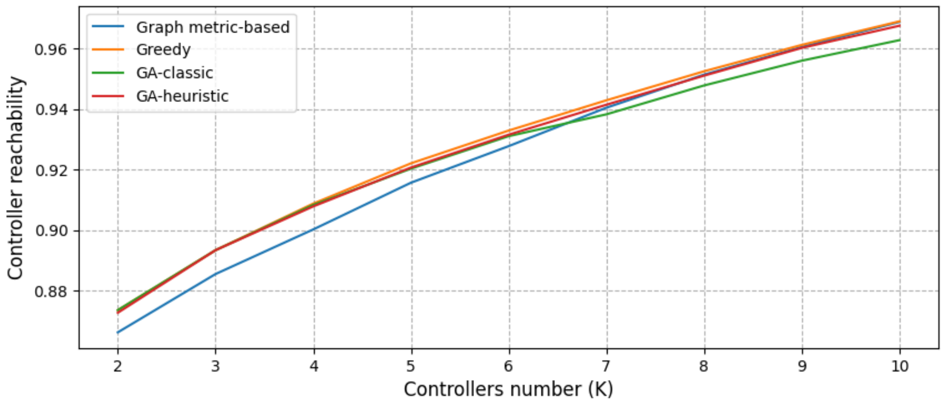
In this section, we evaluate the performance of the four controller placement strategies introduced in Chapter 5 across three network categories: ER random graphs, BA random graphs, and 155 real-world networks. Our objective is to identify the most effective strategy, which can find the placement with high controller reachability.

6.3.1. ER random graph

In order to assess the performance of the four controller placement strategies, we conducted experiments on two sets of ER random graphs. The first set consists of 50 ER random graphs with 25 nodes, while the second set comprises 20 ER random graphs with 50 nodes. For all the graphs in both sets, the link probability p is set to 0.99. The average of controller reachability



(a) ER(25,0.1)



(b) ER(50,0.05)

Figure 6.9: Comparison of placement strategies on Erdős–Rényi model. 4 strategies are applied to place up to 10 controllers with $p = 0.99$. The x-axis denotes the number of controllers, the y-axis denotes the controller reachability. The blue curve represents the average controller reachability of the placements found by graph metric based strategy (degree and distance). The orange curve represents the average controller reachability of the placements found by the greedy algorithm. The green and red curves represent the average controller reachability of the placements found by classic GA and heuristic GA, respectively.

obtained from different placement strategies are presented in Fig. 6.9.

In the case of the ER random graphs with 25 nodes, the performance of the four placement strategies shows overlap. By placing 5 controllers, the controller reachability can achieve a high value of 0.995. However, further improvements in controller reachability are marginal when adding controllers beyond $K = 6$.

On the other hand, for the ER random graphs with 50 nodes, the performance of the placement strategy based on graph metrics initially lags behind the other strategies. However, as the number of controllers increases, the performance of the graph metric based strategy gradually converges towards the other methods. Notably, this method proves to be the most time-efficient among the strategies. Examining the plotted curve, it is obvious that placing 10 controllers has not yet reached a saturation point, which suggests that further improvements are possible by adding more controllers.

6.3.2. BA random graph

The evaluation of the four placement strategies was conducted on three sets of BA random graphs: 50 graphs with parameters $n = 25$ and $m = 1$, 20 graphs with parameters $n = 25$ and $m = 2$, and 50 graphs with parameters $n = 50$ and $m = 1$. The link probability p is set to 0.99, and the average results are presented in Fig 6.10.

Across all BA graph sets, the curves representing the four placement strategies exhibit significant overlap. This observation indicates that the placement strategy based on graph metrics performs the best, primarily due to its lower time consumption. Notably, the differences in controller reachability between BA graphs with $m = 1$ and BA graphs with $m = 2$ are substantial, supporting the notion that nodes with degree 1 significantly contribute to the lower controller reachability values.

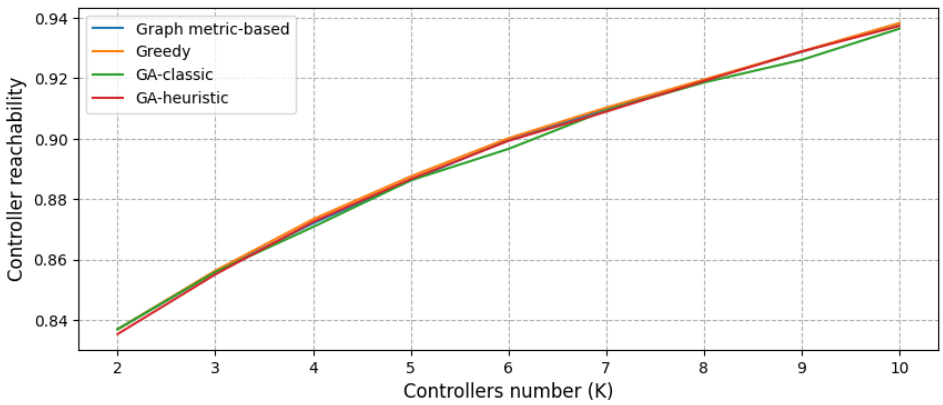
6.3.3. Real-world graph

In order to evaluate the effectiveness of various controller placement strategies, we selected a dataset comprising of 150 connected graphs with small size ($11 \leq n \leq 50$) and 5 connected graphs with middle/large size ($50 \leq n$). The networks chosen for analysis are characterized as sparse networks, exhibiting average node degrees ranging from 1.875 to 4.48. Among the selected networks, the smallest network consists of 11 nodes and 14 edges, the largest network consists of 197 nodes and 243 edges.

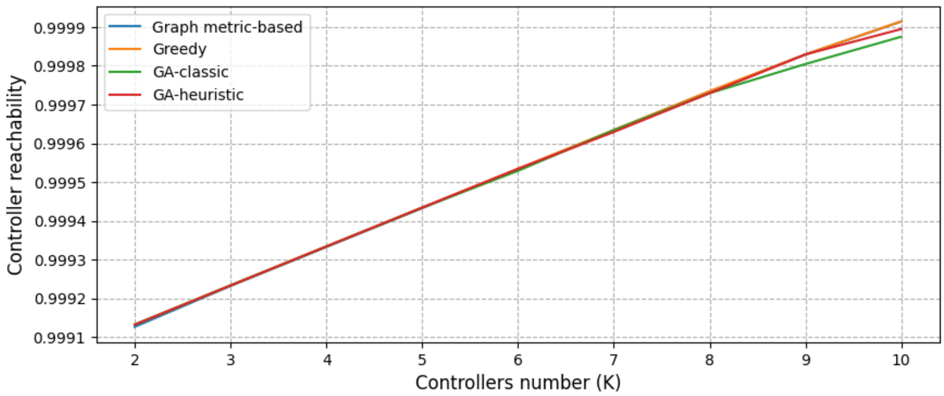
Small sized network

The four placement strategies are evaluated on 150 small sized real-world networks sourced from the Topology Zoo. The link probability p is set to 0.99, and the average results are depicted in Fig. 6.11.

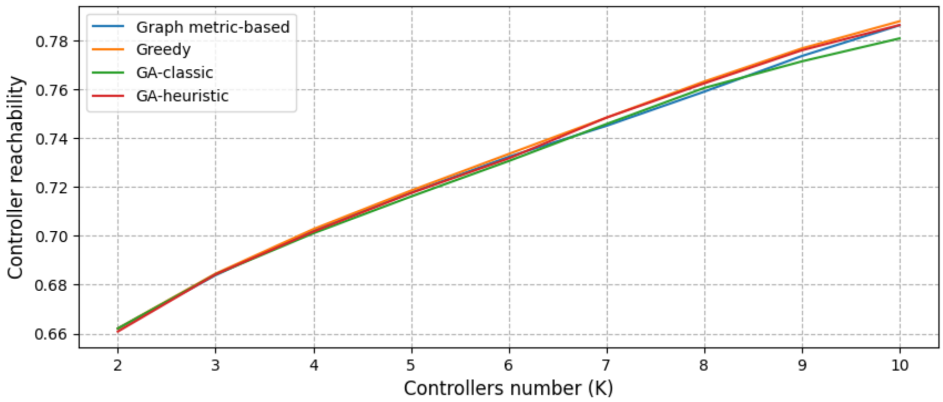
Across all the real-world graphs, the curves representing the four placement strategies ex-



(a) BA(25,1)



(b) BA(25,2)



(c) BA(50,1)

Figure 6.10: Comparison of placement strategies on Barabási–Albert model. 4 strategies are applied to place up to 10 controllers with $p = 0.99$. The x-axis denotes the number of controllers, the y-axis denotes the controller reachability. The blue curve represents the average controller reachability of the placements found by graph metric based strategy (degree and distance). The orange curve represents the average controller reachability of the placements found by the greedy algorithm. The green and red curves represent the average controller reachability of the placements found by classic GA and heuristic GA, respectively.

hibit significant overlap. The attained controller reachability by employing different placement strategies displays a close proximity, indicating that the quality of the placements obtained through these four strategies is similar. This observation again suggests that the placement strategy based on graph metrics outperforms the others due to its lower time consumption.

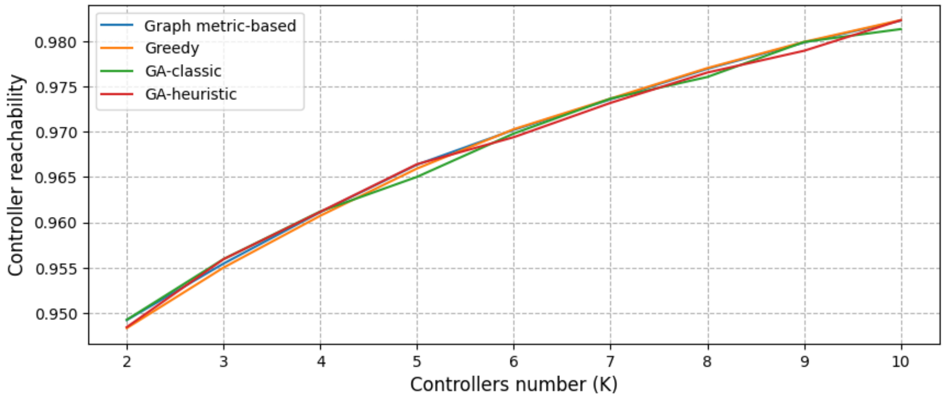


Figure 6.11: Comparison of placement strategies on 150 graphs from the Topology Zoo. 4 strategies are applied to place up to 10 controllers with $p = 0.99$. The x-axis denotes the number of controllers, the y-axis denotes the controller reachability. The blue curve represents the average controller reachability of the placements found by graph metric based strategy (degree and distance). The orange curve represents the average controller reachability of the placements found by the greedy algorithm. The green and red curves represent the average controller reachability of the placements found by classic GA and heuristic GA, respectively.

Middle and large sized networks

Table 6.3 presents the characteristics of the 5 selected networks, including the number of nodes, the number of edges, the average degree, and the number of nodes with a degree lower than the average degree. Given the large number of nodes in each graph and the observation from Section 4.1 that the optimal placement tends to involve low degree nodes, we consider the nodes with degrees lower than the average degree as a potential set of nodes for the controller placement.

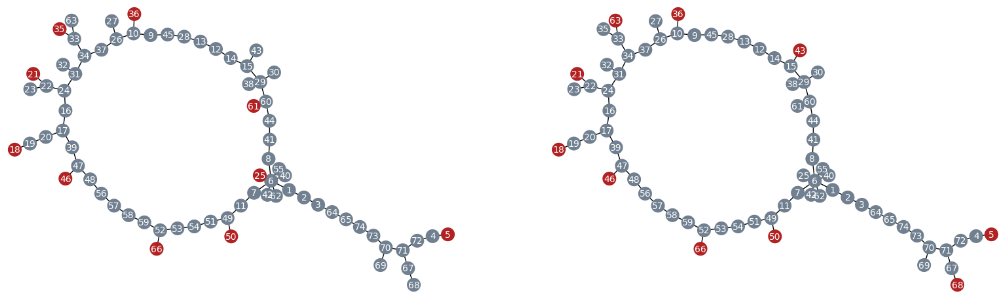
	N	L	$E[D]$	$d = 1$	$d = 2$
HinerniaGlobal	55	81	2.945	1	20
Syringa	74	74	2	23	34
Interoute	110	146	2.655	8	53
Cogentco	197	243	2.467	22	95
GtsCe	149	193	2.591	12	80

Table 6.3: Properties of 5 middle/large sized real-world networks from the Topology Zoo.

In addition to the four placement strategies mentioned earlier, we also incorporated a random placement approach, which involved 10,000 times simulation. For the HinerniaGlobal network and the Syringa network, we also obtained the optimal placements for different values of K within the potential nodes set, comprising nodes with degrees equal to 1 and 2. The outcomes of these 5 networks are illustrated in Fig 6.13. It is observed that all four placement strategies performed well on HinerniaGlobal, Interoute, and GtsCe. However, the graph metric based strategy exhibited relatively poorer performance compared to the other methods on Syringa and Cogentco. This discrepancy can be attributed to the influence of network topology.

Taking Syringa as an example, the placement of 10 controllers using both the greedy algorithm and the graph metric based strategy is depicted in Fig 6.12. Both the greedy algorithm and graph metric-based strategy select node 5 as the initial controller. However, as $K = 2$, the subsequent controller selections differ between the two strategies. The greedy algorithm selects node 18 as the location for the second controller, whereas the graph metric-based method selects node 21. Upon examining the network topology, it becomes apparent that the greedy algorithm makes a better choice. Although node 21 exhibits the greatest distance from node 5 with a distance of 31, node 18 is more susceptible to disconnection due to it connects to the "ring" portion with a path whose length is 3, which is higher than the path length between node 21 and the "ring" portion. In Syringa network, this "ring" portion can be considered as a portion which is relatively more reliable. Compared with node 21, node 18 is farther from that reliable "ring". In scenarios where similar situations arise, graph metric based strategy perform cannot achieve a near-optimal solution. This example highlights the limitations of the graph metric based strategy in certain topologies.

Based on the above comparison, it can be concluded that the strategy based on graph metrics, specifically degree and distance, effectively places controllers in terms of controller reachability. This method proves to be the less time-consuming approach and yields placements comparable to other heuristic methods that are generally more time-consuming for most networks. While it exhibits limitations in certain topologies, overall it demonstrates satisfactory performance. The greedy algorithm performs well across all tested sets of networks, consistently identifying placements with a high controller reachability. Although it is slightly more time-consuming, it consistently delivers favorable results. The performance of the classic GA heavily depends on factors such as population size and iteration times. It exhibits less stability compared to the heuristic GA, and often requires longer execution times. Both classic GA and heuristic GA are the most time-consuming strategies, yet they do not outperform the greedy algorithm.



(a) Graph metric based **(b)** Greedy

Figure 6.12: Syringa: Placement for 10 controllers found by graph metric based and greedy strategy.

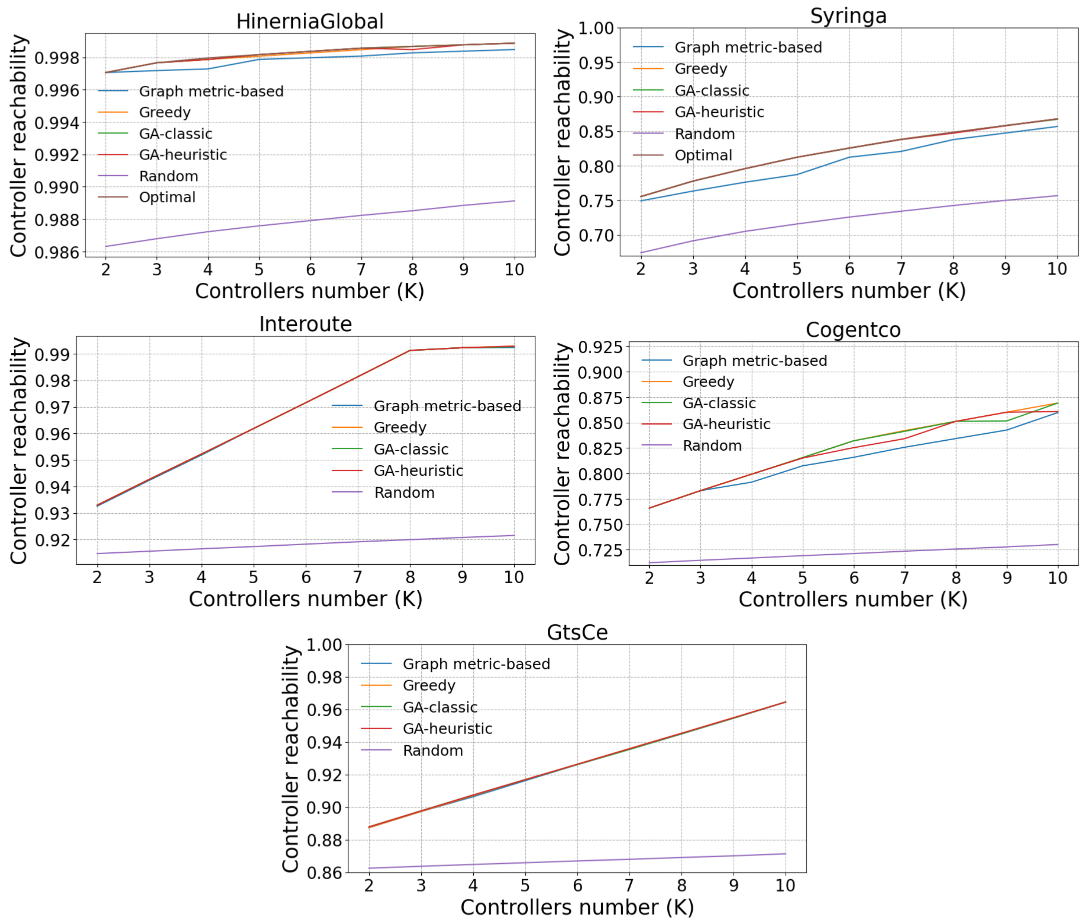


Figure 6.13: Comparison of placement strategies on 5 middle/large sized real-world networks. 4 strategies are applied to place up to 10 controllers with $p = 0.99$. The x-axis denotes the number of controllers, the y-axis denotes the controller reachability. The blue curve represents the average controller reachability of the placements found by graph metric based strategy (degree and distance). The orange curve represents the average controller reachability of the placements found by the greedy algorithm. The green and red curves represent the average controller reachability of the placements found by classic GA and heuristic GA, respectively. The purple curve represents the average value of random placement. For HinerniaGlobal and Syringa, the optimal placements are represented as brown curves.

7

Conclusion

In this thesis, our research focuses on the controller placement problem, with controller reachability as the primary performance metric. We begin by employing the path decomposition algorithm to evaluate the controller reachability. Through a comprehensive analysis of over 40,000 graphs from three distinct graph classes and 100 real-world networks, we identify two influential graph metrics: degree and distance. These metrics exhibit a significant impact on controller reachability. Subsequently, we propose a controller placement strategy based on graph metrics. Additionally, we introduce three widely used heuristic algorithms for determining controller placement. To explore the impact of controller placement on controller reachability and determine the required number of controllers, we analyze and identify the optimal placement for 100 real-world graphs considering different number of controllers K ranging from 2 to 5. We conduct comprehensive evaluations by testing different placement strategies on Erdős–Rényi random graphs, Barabási–Albert random graphs, and a set of 155 real-world graphs obtained from the Topology Zoo dataset. This extensive experimentation enables us to assess the performance and effectiveness of the placement strategies across a wide range of network topologies, including both randomly generated graphs and real-world networks.

In chapter 3, the path decomposition algorithm is employed to evaluate the controller reachability. Through our research, we establish a correlation between the controller reachability and the all-terminal reliability by consolidating nodes where controllers are co-located. Furthermore, we introduce the principle of path decomposition and propose an improved approach for determining path decomposition. We also prove that controller reachability is not a submodular function.

In chapter 4, we identify two influential graph metrics: degree and distance. We conduct an exhaustive enumeration of optimal placements for all connected non-isomorphic graphs belonging to classes $\Omega(7,9)$, $\Omega(10,12)$, and $\Omega(9,18)$ as well as 100 small size networks from the Topology Zoo, considering different link probabilities p. Our analysis reveals a consistent tendency in the optimal placement selection, which considers nodes that are most susceptible to disconnection as the preferred locations for the controllers. This preference is evident through the evaluation of two significant graph metrics, namely node degree and distance between the selected node and the controller. Furthermore, we discover that certain graphs exhibit changes in their optimal placement as the link probability p varies, resulting in an intersection of controller reachability. To explore this phenomenon, we investigate the Aarnet network and analyze an additional set of five graphs from $\Omega(7,9)$. Our findings indicate that the shifting optimal placement is influenced by the network topology. We observe that these intersections do not significantly impact overall performance, as the optimal placements obtained at higher link probabilities remain close to optimal at lower link probabilities.

In chapter 5, four controller placement strategies are introduced. We propose a novel controller placement strategy based on the graph metrics of degree and distance. Our approach prioritizes nodes with the lowest degree for controller placement while simultaneously maximizing the distance between each controller. Additionally, we introduce the greedy algorithm and the genetic algorithm, which are commonly employed in optimization problems, to address the controller placement problem.

In chapter 6, we focus on investigating the impact of placements on controller reachability and determining the number of controllers required in a network. We examine the impact of different placements on controller reachability by analyzing the OS3E network and 100 additional networks. The result shows that random placement is far from optimal, which indicates that it is necessary to find a strategy to place controller such that a close-to-optimal placement is achieved. The number of needed controllers is depends on the network topology, link operational probability p, and the controller reachability requirement. Different networks perform differently and we cannot conclude a number that applies to all networks. Furthermore, 4 placement strategies (graph metric based, greedy, classic GA, heuristic GA) are applied to place up to 10 controllers on Erdős–Rényi random graphs, Barabási–Albert random graphs, and 155 real-world graphs from the Topology Zoo. The result shows that, for most networks, the performance of these 4 methods is almost the same. The strategy based on graph metrics proves to be the less time-consuming approach and yields placements comparable to other heuristic methods. Despite some limitations in certain topologies, its overall performance is satisfactory. The greedy algorithm performs well across all tested sets of networks, consistently identifying placements with a high controller reachability. The classic GA exhibits less stability compared to the heuristic GA. Both classic GA and heuristic GA are the most time-consuming strategies, yet they do not outperform the greedy algorithm. Therefore, considering the performance and algorithm complexity, the strategy based on graph metrics and the greedy algorithm are the most suitable methods to address the controller placement problem.

For future work, we would like to address the following aspects: 1) Consider the controller placement problem where nodes also fail. The path decomposition algorithm we used can also extend to a new algorithm which can compute the controller reachability when nodes and edges are both operational with a probability. 2) Consider other performance metrics like node-controller reachability which assesses the probability of a specific node being able to establish communication with at least one controller. The worst or the average node-controller reachability can be used as a performance metric. 3) Consider the controller placement problem where the controllers are not co-located with switches. A controller is not placed at a node, but connects to a node, which means the connections between controllers and switches might also fail. 2) Improve the strategy based on graph metrics by considering additional topology properties, such as the existence of multiple paths between nodes and controllers.

References

[1] Osman Alp, Erhan Erkut, and Zvi Drezner. “An efficient genetic algorithm for the p-median problem”. In: *Annals of Operations research* 122 (2003), pp. 21–42.

[2] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks”. In: *Science* 286.5439 (1999), pp. 509–512.

[3] James L Beck and Konstantin M Zuev. “Rare event simulation”. In: *arXiv preprint arXiv:1508.05047* (2015).

[4] Jacques Carlier and Corinne Lucet. “A decomposition algorithm for network reliability evaluation”. In: *Discrete Applied Mathematics* 65.1-3 (1996), pp. 141–156.

[5] Masaki Chujyo and Yukio Hayashi. “Adding links on minimum degree and longest distance strategies for improving network robustness and efficiency”. In: *PLOS ONE* 17.10 (2022), e0276733.

[6] Kusum Deep and Hadush Mebrahtu. “Variant of partially mapped crossover for the travelling salesman problems”. In: *International Journal of Combinatorial Optimization Problems and Informatics* 3.1 (2012), pp. 47–69.

[7] Paul Erdős, Alfréd Rényi, et al. “On the evolution of random graphs”. In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1 (1960), pp. 17–60.

[8] Vaibhav Gaur et al. “A literature review on network reliability analysis and its engineering applications”. In: *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 235.2 (2021), pp. 167–181.

[9] Jennifer Golbeck. *Introduction to social media investigation: A hands-on approach*. Syngress, 2015.

[10] Brandon Heller, Rob Sherwood, and Nick McKeown. “The controller placement problem”. In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 473–478.

[11] Tao Hu et al. “Multi-controller Based Software-Defined Networking: A Survey”. In: *IEEE Access* 6 (2018), pp. 15980–15996.

[12] Yannan Hu et al. “On reliability-optimized controller placement for software-defined networks”. In: *China Communications* 11.2 (2014), pp. 38–54.

[13] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. “A review on genetic algorithm: past, present, and future”. In: *Multimedia Tools and Applications* 80 (2021), pp. 8091–8126.

[14] Simon Knight et al. “The Internet Topology Zoo”. In: *IEEE Journal on Selected Areas in Communications* 29.9 (2011), pp. 1765–1775.

[15] Andreas Krause and Daniel Golovin. “Submodular function maximization.” In: *Tractability* 3 (2014), pp. 71–104.

[16] Andreas Krause et al. “Efficient sensor placement optimization for securing large water distribution networks”. In: *Journal of Water Resources Planning and Management* 134.6 (2008), pp. 516–526.

[17] Diego Kreutz et al. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[18] Abha Kumari and Ashok Singh Sairam. “Controller placement problem in software-defined networking: A survey”. In: *Networks* 78.2 (2021), pp. 195–223.

[19] Brendan D McKay and Adolfo Piperno. “Practical graph isomorphism, II”. In: *Journal of symbolic computation* 60 (2014), pp. 94–112.

[20] Elchanan Mossel and Sebastien Roch. “On the submodularity of influence in social networks”. In: *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 2007, pp. 128–134.

[21] Willem Pino, Teresa Gomes, and Robert Kooij. “A comparison between two all-terminal reliability algorithms”. In: *Journal of Advances in Computer Networks* 3.4 (2015), pp. 284–290.

[22] André Pönitz and Peter Tittmann. *Computing network reliability in graphs of restricted pathwidth*. 2001. URL: `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.7220`.

[23] Francisco Javier Ros and Pedro Miguel Ruiz. “Five nines of southbound reliability in software-defined networks”. In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 31–36.

[24] Arnie Rosenthal. “Computing the reliability of complex networks”. In: *SIAM Journal on Applied Mathematics* 32.2 (1977), pp. 384–393.

[25] Afrim Sallahi and Marc St-Hilaire. “Optimal model for the controller placement problem in software defined networks”. In: *IEEE Communications Letters* 19.1 (2014), pp. 30–33.

[26] Appajosyula Satyanarayana and R Kevin Wood. “A linear-time algorithm for computing K-terminal reliability in series-parallel networks”. In: *SIAM Journal on Computing* 14.4 (1985), pp. 818–832.

[27] Manohar Shamaiah, Siddhartha Banerjee, and Haris Vikalo. “Greedy sensor selection: Leveraging submodularity”. In: *49th IEEE Conference on Decision and Control (CDC)*. 2010, pp. 2572–2577.

[28] Anh Khoa Tran, Md Jalil Piran, and Chuan Pham. “SDN controller placement in IoT networks: An optimized submodularity-based approach”. In: *Sensors* 19.24 (2019), p. 5474.

[29] Remco Van Der Hofstad. *Random graphs and complex networks*. Vol. 43. Cambridge university press, 2016.

[30] Piet Van Mieghem. *Performance analysis of complex networks and systems*. Cambridge University Press, 2014.

[31] Andrew Vince. “A framework for the greedy algorithm”. In: *Discrete Applied Mathematics* 121.1-3 (2002), pp. 247–260.

[32] Guodong Wang et al. “The controller placement problem in software defined networking: A survey”. In: *IEEE Network* 31.5 (2017), pp. 21–27.

[33] Gerhard J Woeginger. “Exact algorithms for NP-hard problems: A survey”. In: *Combinatorial Optimization—Eureka, You Shrink! Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001 Revised Papers*. Springer. 2003, pp. 185–207.

[34] Gang Yu and Jian Yang. “On the robust shortest path problem”. In: *Computers & Operations Research* 25.6 (1998), pp. 457–468.

[35] Yuan Zhang et al. “A survey on software defined networking with multiple controllers”. In: *Journal of Network and Computer Applications* 103 (2018), pp. 101–118.

[36] Jinghui Zhong et al. “Comparison of performance between different selection strategies on simple genetic algorithms”. In: *International conference on computational intelligence for modelling, control and automation and international conference on intelligent agents, web technologies and internet commerce (CIMCA-IAWTIC’06)*. Vol. 2. IEEE. 2005, pp. 1115–1121.

[37] Qinghong Zhong et al. “A min-cover based controller placement approach to build reliable control network in SDN”. In: *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2016, pp. 481–487.

[38] Wenzhu Zou et al. “On the availability of networks”. In: *Proc. of BroadBand Europe* (2007).

A

Path decomposition algorithm

This appendix presents the code of the path decomposition algorithm.

```
import numpy as np
import matplotlib.pyplot as plt
import random
import networkx as nx
from decimal import *
print(getcontext())
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])
getcontext().prec = 50

def FindPath2(network,n):
    nodes=np.array(range(n))
    minAN=n
    minANtotal=n*n
    minPathDecomposition=0
    for node in nodes:
        maxAN=0
        ANtotal=0
        pathDecomposition=[]
        pathDecomposition.append(node) # node is the first node to activate
        remainNodes=nodes[nodes!=node]

        while len(remainNodes)!=0:
            a=np.nonzero(network[:,pathDecomposition])
            Neighborhood=list(set(a[0]).intersection(set(remainNodes)))
            random.shuffle(Neighborhood)
            tempAN=n

            for neighbor in Neighborhood:
                tempPD=pathDecomposition.copy()
                tempPD.append(neighbor)
                tempRN=remainNodes[remainNodes!=neighbor]
                ActivateNode=len(tempPD)
```

```
                for i in tempPD:
                    if network[i,tempRN].any()==0:
                        ActivateNode=ActivateNode-1

                if ActivateNode < tempAN:
                    tempAN=ActivateNode
                    chosen=neighbor

            if tempAN==len(tempPD):
                connection=0
                for neighbor in Neighborhood:
                    tempConnection=sum(network[pathDecomposition,neighbor])
                    if tempConnection>connection:
                        connection=tempConnection
                        chosen=neighbor
            pathDecomposition.append(chosen)
            remainNodes=remainNodes[remainNodes!=chosen]
            ANtotal=ANtotal+tempAN

            if tempAN > maxAN:
                maxAN=tempAN

        if maxAN <= minAN and ANtotal<=minANtotal:
            minAN=maxAN
            minANtotal=ANtotal
            minPathDecomposition=pathDecomposition

    return minAN,minPathDecomposition

def FindPath(network,n):  # This the method used in "Computing network reliability in graphs of restricted pathwidth" to find upper bound pathwidth
    nodes=np.array(range(n))
    minPathwidth=n
    minPathDecomposition=0
    for node in nodes:
        maxVertexSep=0
        pathDecomposition=[]
        pathDecomposition.append(node)
        remainNodes=nodes[nodes!=node]

        while len(remainNodes)!=0:
            a=np.nonzero(network[:,pathDecomposition])
            Neighborhood=list(set(a[0]).intersection(set(remainNodes)))
            random.shuffle(Neighborhood)
            tempVertexSep=n

            for neighbor in Neighborhood:
                tempPD=pathDecomposition.copy()
                tempPD.append(neighbor)
                tempRN=remainNodes[remainNodes!=neighbor]
                VertexSep=np.count_nonzero(sum(network[tempPD][:,tempRN]))

                if VertexSep < tempVertexSep:
                    tempVertexSep=VertexSep
```

```
                    chosen=neighbor

            pathDecomposition.append(chosen)
            remainNodes=remainNodes[remainNodes!=chosen]

            if tempVertexSep > maxVertexSep:
                maxVertexSep=tempVertexSep

        if maxVertexSep < minPathwidth:
            minPathwidth=maxVertexSep
            minPathDecomposition=pathDecomposition

    return minPathwidth,minPathDecomposition
# find the decomposition series (activate node/deactivate node/activate link)
def FindSeries(network,Path):
    path=np.add(Path,1) # with node numbering start from 1
    for i in path:
        int(i)
    series=[]
    DelNodes=[]
    i=0
    for node in path:
        i=i+1
        # activate node
        series.append(node)
        # activate egde
        a=np.nonzero(network[:,node-1])
        Neighbors=np.add(a[0],1) # all neighbors of node
        for Neighbor in Neighbors:
            if Neighbor in path[0:i]:
                series.append([Neighbor,node])

                # deactivate node
                NeighborColumn=network[:,Neighbor-1]
                NeighborColumn=NeighborColumn[path[i:]-1]  # if neighbor is connected to the remain part
                neighborSepSet=sum(NeighborColumn)
                if neighborSepSet==0 and (Neighbor not in DelNodes):
                    series.append(-Neighbor)
                    DelNodes.append(Neighbor)

        # deactivate node
        nodeColumn=network[:,node-1]
        nodeColumn=nodeColumn[path[i:]-1]
        nodeSepSet=sum(nodeColumn)
        if nodeSepSet==0 and (node not in DelNodes):
            series.append(-node)
            DelNodes.append(node)

    # print('The decomposition series is',series)  # with node numbering start from 1
    return series
def add_poly(L1,L2):
    R=[]
    if len(L1)>len(L2):
```

```
        L1,L2=L2,L1
    i=0
    while i<len(L1):
        R.append(L1[i]+L2[i])
        i+=1
    R=R+L2[len(L1):len(L2)]
    return R

def multiply_poly(L1,L2):
    if len(L1)>len(L2):
        L1,L2=L2,L1
    zero=[];R=[]
    for i in L1:
        T=zero[:]
        for j in L2:
            T.append(i*j)
        R=add_poly(R,T)
        zero=zero+[0]
    return R
def Decomposition(network,n,series,Link_to_add):
    # Reliability from decomposition method
    # Polynomial is stored as [a_0,a_1,...a_n]
    # classes are stored as following example:
    # node class1 class2
    #  1     0      0
    #  2     0      0
    #  3     1      1
    #  4     2      1
    #  5     0      0
    #  6     0      0
    # Meaning: 3 and 4 are activated node, two ways to group them  (3/4)
       or (34)

    R=[]
    R.append([1,0]) # initial reliability (cannot use [1] at here)
    X=np.zeros((n,2))
    X[:,0]=range(1,n+1) # Create classes storage array
    X[series[0]-1,1]=1 # First step and activate first node

    for s in range(1,len(series)):  # Start from second step in series
        step=series[s]
        if isinstance(step, (int, np.integer)):    # If it is int number,
           thus is node process
            if s==len(series)-1:
                # print(R)
                z=1
            else:
                if step > 0:
                    # activation of node
                    # print('This step is activation of node',step)
                    for Class in range(1,X.shape[1]):
                        New_class=len(set(X[:,Class]))
                        X[step-1,Class]=New_class
                    # print(X,R)
                else:
                    # deactivation of node
```

```
                    # print('This step is deactivation of node',step)
                    ColumnDel=[]  # record which class make node disconnected
                    for Class in range(1,X.shape[1]):
                        ClassNum_del=X[abs(step)-1,Class]
                        X[abs(step)-1,Class]=0 # deactivate node
                        if ClassNum_del not in X[:,Class]:
                            # this deactivated node is disconnected
                            ColumnDel.append(Class)
                        else:
                            # renumbering
                            ClassNum=len(set(X[:,Class]))
                            b=range(1,ClassNum)
                            X[:,Class]
                            j=0
                            for i in range(len(X[:,Class])):
                                if X[i,Class]>0:
                                    X[:,Class][X[:,Class]==X[i,Class]]=-b[j]
                                    j=j+1
                            X[:,Class]=abs(X[:,Class])

                    X = np.delete(X, ColumnDel, axis=1)
                    RowDel=[i-1 for i in ColumnDel]
                    RowDel.sort(reverse=True)
                    for i in RowDel:
                        del R[i]


                    # Merge the class with same partition
                    for Class in range(1,X.shape[1]):
                        ColumnMerge=[]
                        ColumnMerge.append(Class)
                        for remainClass in range(Class+1,X.shape[1]):
                            if (X[:,Class]==X[:,remainClass]).all():
                                ColumnMerge.append(remainClass)
                        if len(ColumnMerge)>=2:
                            X = np.delete(X, ColumnMerge[1:], axis=1)
                            RowMerge=[i-1 for i in ColumnMerge]
                            R_merge=[]
                            for i in RowMerge:
                                R_merge.append(R[i])
                            R_new=R_merge[0].copy()
                            for i in range(1,len(R_merge)):
                                R_new=add_poly(R_new,R_merge[i])
                            a=RowMerge[1:]
                            a.sort(reverse=True)
                            for i in a:
                                del R[i]
                            R[RowMerge[0]]=R_new
                    # print(X,R)
        else:
            # activation of edge
            # print('This step is activation of edge',step)
            node1,node2=step   # neighbor, node
            ClassNum_old=X.shape[1]
```

```
            SpecialLink=0
            for i in range(len(Link_to_add)):
                if set((node1,node2))==set(Link_to_add[i]):
                    SpecialLink=1
            if SpecialLink==1:
                for Class in range(1,ClassNum_old):
                    Class_change=X[:,Class].copy()

                    a=Class_change[node1-1]
                    b=Class_change[node2-1]
                    if a<b:
                        Class_change[Class_change==b]=Class_change[node1-1]
                    else:
                        Class_change[Class_change==a]=Class_change[node2-1]
                    a=Class_change.copy()
                    a=np.insert(a, 0, values=0, axis=0)
                    ClassNum=len(set(a))

                    for number in range(1,ClassNum+1):
                        if number not in set(a):
                            for j in range(n):
                                if Class_change[j]>number:
                                    Class_change[j]=Class_change[j]-1

                    X[:,Class]=Class_change
            else:
                R_new=[]
                for Class in range(1,ClassNum_old):
                    # Get new X (insert new class in case edge is connected)
                    Class_add=X[:,Class*2-1].copy()
                    a=Class_add[node1-1]
                    b=Class_add[node2-1]
                    if a<b:
                        Class_add[Class_add==b]=Class_add[node1-1]
                    else:
                        Class_add[Class_add==a]=Class_add[node2-1]

                    # Make group nummbering continuous here, ex: change 0,1,3,1,0,0 to 0,1,2,1,0,0
                    a=Class_add.copy()
                    a=np.insert(a, 0, values=0, axis=0)
                    ClassNum=len(set(a))

                    for number in range(1,ClassNum+1):
                        if number not in set(a):
                            for j in range(n):
                                if Class_add[j]>number:
                                    Class_add[j]=Class_add[j]-1
                    X=np.insert(X, Class*2, values=Class_add, axis=1)

                    # Get new R
                    R_new.append(multiply_poly(R[Class-1],[1,-1]))  # mutiply to 1-p
```

```
                    R_new.append(multiply_poly(R[Class-1],[0,1]))   #
                        mutiply to p
                R=R_new


            # Merge the class with same partition
            for Class in range(1,X.shape[1]):
                ColumnMerge=[]
                ColumnMerge.append(Class)
                for remainClass in range(Class+1,X.shape[1]):
                    if (X[:,Class]==X[:,remainClass]).all():
                        ColumnMerge.append(remainClass)
                if len(ColumnMerge)>=2:
                    X = np.delete(X, ColumnMerge[1:], axis=1)
                    RowMerge=[i-1 for i in ColumnMerge]
                    R_merge=[]
                    for i in RowMerge:
                        R_merge.append(R[i])
                    R_new=R_merge[0].copy()
                    for i in range(1,len(R_merge)):
                        R_new=add_poly(R_new,R_merge[i])
                    a=RowMerge[1:]
                    a.sort(reverse=True)
                    for i in a:
                        del R[i]
                    R[RowMerge[0]]=R_new
            # print(X,R)
    return R
def Link_between_sensor(Sensors,G):
    NumSensor=len(Sensors)
    Link_to_add=[]

    check_G=G.subgraph(Sensors)
    listCC = [len(c) for c in sorted(nx.connected_components(check_G), key
        =len, reverse=True)]
    List=sorted(nx.connected_components(check_G)) # components with node
    minD_node=np.zeros(len(listCC))
    for i in range(len(listCC)):
        component=G.subgraph(List[i])
        degree=np.array(G.degree(List[i]))
        degree=degree[np.lexsort(degree.T)]
        minD_node[i]=degree[0,0]
        if len(component.edges()) !=0:
            for i in component.edges():
                Link_to_add.append(i)
    for i in range(len(minD_node)-1):
        Link_to_add.append((minD_node[i],minD_node[i+1]))

    Link_to_add=np.array(Link_to_add)
    return Link_to_add

def R_poly(G,Sensors):
    g=G.copy()
    n=G.number_of_nodes()
    Link_to_add=Link_between_sensor(Sensors,G)
    for j in Link_to_add:
```

```
        g.add_edge(j[0],j[1])

    Adj=nx.adjacency_matrix(g)
    A=Adj.todense()
    network=A.copy()
    PathWidth,Path=FindPath2(network,n)
    series=FindSeries(network,Path)
    R=Decomposition(network,n,series,Link_to_add)
    R=R[0] # The all - terminal reliability when placed two sensor
    return R

def main():
    # import network at here
    s='Real1.txt'
    lines=[]
    # with open('/home/ranxu/TopologyZooNetworks/'+s, 'r') as f:
    with open('D:/TUD/Code/python/Thesis/TopologyZooNetworks/'+s, 'r') as
        f:
        for line in f.readlines():
            line = line.replace('\n','').replace('\t',' ')
            lines.append(line)

    edges=[]
    for i in range(len(lines)):
        a=list(map(int,lines[i].split()))
        edges.append(a)
    edges=np.array(edges)
    G=nx.Graph()
    for edge in edges:
        G.add_edge(edge[0], edge[1])
    G=nx.convert_node_labels_to_integers(G,first_label=1) # Cannot delete
        this line !!!
    nx.draw(G, pos=nx.kamada_kawai_layout(G),node_size=300,with_labels =
        True)
    plt.show()

    Controllers=[1,2]
    reliability=R_poly(G,Controllers)
    print(reliability)

if __name__ == "__main__":
    main()
```

B

Graph metric-based strategy

This appendix presents the code of graph metric-based strategy. Only the functions responsible for placement are shown. See GitHub for complete code.
https://github.com/Amyxuran/Controller-placement.git

```
def MaxDistance_placement(G,K):
    # find two node far away from othernode
    paths=list(nx.shortest_path_length(G))
    D=dict(nx.shortest_path_length(G))

    n=G.number_of_nodes()
    L=G.number_of_edges()
    AvgD=np.floor(2*L/n)
    degree=np.array(list(G.degree()))
    NodeSet=list(G.degree())
    numMin=0
    S=sorted(set(degree[:,1]))
    for i in degree:
        if i[1]==S[0]:
            numMin=numMin+1
    PlaceKSensor=[]
    sortedNodeSet=np.array(sorted(NodeSet,key=lambda x:x[1]))
    if K<=numMin:
        SumDistance=[]
        NewNodeSet=sortedNodeSet[:numMin,0].copy()
        NewNodeSet_len=len(NewNodeSet)
        for i in range(NewNodeSet_len):
            SumDistance.append(sum(dict.values(paths[NewNodeSet[i]-1][1]))
                )
        PlaceKSensor.append(NewNodeSet[np.argmax(SumDistance)])
        for i in range(K-1):
            Slength=np.zeros((NewNodeSet_len,2))
            for j in range(NewNodeSet_len):
                Slength[j,1]=max(SumDistance)
                Slength[j,0]=j
            for sensor in PlaceKSensor:
                for j in range(NewNodeSet_len):
```

```
                    Slength[j,1]=min(Slength[j,1],D[sensor][NewNodeSet[j]])
            sortedSlength=sorted(Slength,key=lambda x:x[1],reverse=True)
            for node in range(NewNodeSet_len):
                NewNode=NewNodeSet[int(sortedSlength[node][0])]
                if NewNode not in PlaceKSensor:
                    PlaceKSensor.append(NewNode)
                    break
    else:
        include_num=numMin
        for s in S[1:]:
            count=0
            for i in sortedNodeSet[include_num:]:
                if i[1]==s:
                    count=count+1
            if include_num+count>=K:
                K=K-include_num
                part1=sortedNodeSet[:include_num,0]
                part2=sortedNodeSet[include_num:(include_num+count),0]
                NewNodeSet=part2.copy()
                NewNodeSet_len=len(NewNodeSet)
                PlaceKSensor=list(part1.copy())
                for i in range(K):
                    Slength=np.zeros((NewNodeSet_len,2))
                    for j in range(NewNodeSet_len):
                        Slength[j,1]=999
                        Slength[j,0]=j
                    for sensor in PlaceKSensor:
                        for j in range(NewNodeSet_len):
                            Slength[j,1]=min(Slength[j,1],D[sensor][NewNodeSet[j]])
                    sortedSlength=sorted(Slength,key=lambda x:x[1],reverse=True)
                    for node in range(NewNodeSet_len):
                        NewNode=NewNodeSet[int(sortedSlength[node][0])]
                        if NewNode not in PlaceKSensor:
                            PlaceKSensor.append(NewNode)
                            break
                # print('PlaceKSensor: 111 ',PlaceKSensor)
                return PlaceKSensor
            else:
                include_num=include_num+count
    # print('PlaceKSensor: ',PlaceKSensor)
    return PlaceKSensor
```

C

Greedy algorithm

This appendix presents the code of greedy algorithm. Only the functions responsible for placement are shown. See GitHub for complete code.
https://github.com/Amyxuran/Controller-placement.git

```
def greedy(G,PlacedSensor,K,p,R_record,NodeSet):
    PlaceKSensor=[]
    for i in NodeSet:
        if i not in PlacedSensor:
            a=PlacedSensor.copy()
            a.append(i)
            PlaceKSensor.append(a)
    R_all=[]
    for i in range(len(PlaceKSensor)):
        R_all.append(R(G,PlaceKSensor[i],p))

    R_all=np.array(R_all)
    a=np.argsort(R_all)[-1]
    print(PlaceKSensor[a],'probability at p=',p,'is',R_all[a])

    R_record=np.append(R_record,R_all[a])
    sensor_record=PlaceKSensor[a]
    K-=1
    if K>1:
        R_record,sensor_record=greedy(G,PlaceKSensor[a],K,p,R_record,
            NodeSet)
    return R_record,sensor_record
```

D

Classic genetic algorithm

This appendix presents the code of classic genetic algorithm. Only the functions responsible for initializing, selection, crossover and mutation are shown. See GitHub for complete code. https://github.com/Amyxuran/Controller-placement.git

```
def initial_population(G,K,p,NodeSet):
    NodeSet=list(NodeSet)
    n=len(NodeSet)
    d=int(np.ceil(n/K))
    S=factorial(n)/(factorial(K)*factorial(n-K))
    group_num=max((2,int(np.ceil(n/100*np.log(S)/d))))
    population_size=group_num*d
    population_index=[]
    for i in range(1,group_num+1):
        sequence=[]
        for j in range(i):
            sequence=sequence+list(range(j+1,n+1,i))
        empty_slots_num=int(d*K-n)
        sequence=sequence+random.sample(list(set(range(1,n+1)).difference(
            set(sequence[-(K-empty_slots_num):]))),empty_slots_num)
        for j in range(d):
            population_index.append(sequence[j*K:(j*K+K)])
    population=[]
    for individual_index in population_index:
        individual=[]
        for index in individual_index:
            individual.append(NodeSet[index-1])
        population.append(individual)
    dic1={}
    dic2={}
    for k in range(len(population)):
        dic1[k]=sorted(population[k])
        dic2[k]=R(G,population[k],p)

    best_R=max(dic2.items(),key=lambda x:x[1])[1]
    worst_R=min(dic2.items(),key=lambda x:x[1])[1]
    return best_R,worst_R,dic1,dic2
```

```
def select_cross_mutate(G,p,K,best_R,worst_R,dic1,dic2,NodeSet,
    crossover_rate,mutation_rate,elitism_num):
    NodeSet=list(NodeSet)
    population_size=len(dic1)-elitism_num
    remain_placement=[]
    remain_R=[]
    a=sorted(dic2.items(),key=lambda x:x[1],reverse=True)
    for i in range(elitism_num):
        remain_placement.append(dic1[a[i][0]])
        remain_R.append(a[i][1])

    # select: tournament selection
    pop1={}
    pop2={}
    for i in range(population_size):
        a,b=np.random.choice(range(population_size),2,False)
        choice1,choice2=dic1[a],dic1[b]
        choice1_R,choice2_R=dic2[a],dic2[b]
        if choice1_R>=choice2_R:
            pop1[i]=choice1
            pop2[i]=choice1_R
        else:
            pop1[i]=choice2
            pop2[i]=choice2_R

    # cross: single point crossover
    new_pop1={}
    new_pop2={}
    cross_pairs_num=round(population_size*crossover_rate/2)*2
    k=0
    for i in range(0,population_size,2):
        if i <=cross_pairs_num:
            parent1=pop1[i]
            parent2=pop1[i+1]
            cross_point=np.random.randint(1,K)
            child1_temp=parent1.copy()
            child2_temp=parent2.copy()
            child1_temp[cross_point:]=parent2[cross_point:].copy()
            child2_temp[cross_point:]=parent1[cross_point:].copy()
            ## repeat detect
            a=child1_temp[:cross_point]
            b=child1_temp[cross_point:]
            child1=b.copy()
            c=child2_temp[:cross_point]
            d=child2_temp[cross_point:]
            child2=d.copy()
            for i in a:
                while i in b:
                    i = d[b.index(i)]
                child1.append(i)
            for i in c:
                while i in d:
                    i = b[d.index(i)]
                child2.append(i)
        else:
```

```
            child1=pop1[i]
            child2=pop1[i+1]
        if np.random.rand()< mutation_rate:
            while True:
                a=np.random.randint(0,K)
                b=np.random.randint(0,len(NodeSet))
                if NodeSet[b] not in child1:
                    child1[a]=NodeSet[b]
                    break
        if np.random.rand()< mutation_rate:
            while True:
                a=np.random.randint(0,K)
                b=np.random.randint(0,len(NodeSet))
                if NodeSet[b] not in child2:
                    child2[a]=NodeSet[b]
                    break

        new_pop1[k]=child1
        new_pop1[k+1]=child2
        new_pop2[k]=R(G,child1,p)
        new_pop2[k+1]=R(G,child2,p)
        k=k+2
    for i in range(elitism_num):
        new_pop1[k]=remain_placement[i]
        new_pop2[k]=remain_R[i]
        k=k+1

    best_R=max(new_pop2.items(),key=lambda x:x[1])[1]
    worst_R=min(new_pop2.items(),key=lambda x:x[1])[1]
    return best_R,worst_R,new_pop1,new_pop2
```

E

Heuristic genetic algorithm

This appendix presents the code of heuristic genetic algorithm. Only the functions responsible for initializing and crossover are shown. See GitHub for complete code.
https://github.com/Amyxuran/Controller-placement.git

```
def initial_population(G,K,p,NodeSet):
    n=len(NodeSet)
    d=int(np.ceil(n/K))
    S=factorial(n)/(factorial(K)*factorial(n-K))
    group_num=max((2,int(np.ceil(n/100*np.log(S)/d))))
    population_size=group_num*d
    population_index=[]
    for i in range(1,group_num+1):
        sequence=[]
        for j in range(i):
            sequence=sequence+list(range(j+1,n+1,i))
        empty_slots_num=int(d*K-n)
        sequence=sequence+random.sample(list(set(range(1,n+1)).difference(
            set(sequence[-(K-empty_slots_num):]))),empty_slots_num)
        for j in range(d):
            population_index.append(sequence[j*K:(j*K+K)])
    population=[]
    for individual_index in population_index:
        individual=[]
        for index in individual_index:
            individual.append(NodeSet[index-1])
        population.append(individual)

    dic1={}
    dic2={}
    for k in range(len(population)):
        dic1[k]=sorted(population[k])
        dic2[k]=R(G,population[k],p)

    best_R=max(dic2.items(),key=lambda x:x[1])[1]
    worst_R=min(dic2.items(),key=lambda x:x[1])[1]

```

```
    return best_R,worst_R,dic1,dic2

def selection(K,best_R,worst_R,dic1,dic2):

    population=list(dic1.values())
    population_size=len(population)

    # random select
    a,b=random.sample(range(population_size),2)

    #####
    parent1,parent2=population[a],population[b]
    fix_gene=list(set(parent1).intersection((set(parent2))))
    free_gene=list(set(parent1).difference(set(parent2)))+list(set(parent2).difference(set(parent1)))
    child_draft=list(set(parent1+parent2))
    num_drop=len(child_draft)-K
    return sorted(child_draft),sorted(fix_gene),sorted(free_gene),num_drop

def GreedyDeletion(G,p,child_draft,fix_gene,free_gene,num_drop):
    if num_drop==0:
        return sorted(child_draft)
    compare_sensor=[]
    for gene in free_gene:
        sensor=child_draft.copy()
        sensor.remove(gene)
        compare_sensor.append(sensor)

    R_value=[]
    for i in compare_sensor:
        R_value.append(R(G,i,p))
    ind=np.argmax(np.array(R_value))
    child_draft=compare_sensor[ind]
    new_free_gene=list(set(compare_sensor[ind]).difference(set(fix_gene)))
    num_drop-=1
    if num_drop==0:
        return sorted(child_draft)
    else:
        return GreedyDeletion(G,p,child_draft,fix_gene,new_free_gene,
            num_drop)

def OneGeneration(G,p,K,best_R,worst_R,dic1,dic2):
    child_draft,fix_gene,free_gene,num_drop=selection(K,best_R,worst_R,
        dic1,dic2)
    new_population=GreedyDeletion(G,p,child_draft,fix_gene,free_gene,
        num_drop)
    if new_population in list(dic1.values()):
        return best_R,worst_R,dic1,dic2
    new_R=R(G,new_population,p)
    if new_R>worst_R:
        for key,value in dic2.items():
            if value==worst_R:
                dic2[key]=new_R
                dic1[key]=new_population
                break

```

```
        best_R=max(dic2.items(),key=lambda x:x[1])[1]
        worst_R=min(dic2.items(),key=lambda x:x[1])[1]
    return best_R,worst_R,dic1,dic2
```