A PARETO BASED GENETIC ALGORITHM FOR VIRTUAL LOCAL AREA NETWORK OPTIMIZATION

BY

JINKU CUI

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science

May 2020

Winston-Salem, North Carolina

Approved By:

David J. John, Ph.D., Advisor

Errin W. Fulp, Ph.D., Chair

William H. Turkett, Ph.D.

Acknowledgments

At first, I need to thank my advisor, Dr. John. He is more optimistic than me on my research. When I did not make any progress on the research, I sometimes feel frustrated. Dr. John encourages me to move forward and settle obstacles with me together. Without him, I cannot complete the thesis.

I really appreciate Dr. Fulp. He is really passionate on academic research. I have not decided to devote myself into research area. After several times of conversation with Dr. Fulp, I learned the spirit and motivation from him. Dr. Fulp is the one leads me to the gate of research fields. I will inherit his will and expand the boundary of academic realm.

Great thanks to my model, Dr. Turkett. He demonstrates how to be a good teacher. I might become a teacher in a university or college in the future, and teaching will be part of my life. A good scholar can not only teach well, but also stimulate students' interests, fostering them as successors. Besides, I applied the knowledge of parallelism from Dr. Turkett's class. It helps me accelerate my experiments!

Thanks to my guards, Lesley and Cody! Lesley arranges trivial affairs for me, like graduation form and defense schedule. Cody offers technical support for running experiments on Dr. Fulp's workstation. They help me focus on my thesis.

Special thanks to Sarah and Esteban. Sarah and Esteban proofread my thesis proposal, and they give me precious advice.

Thanks to the other students in this program, Patrick, Irina, and Ziqin. Their existences make me feel not so alone.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Jinku Cui

Given a network policy, it is a challenge to create a configuration, which optimizes the security and functionality of networked computers. Virtual Local Access Network (VLAN) is a widely adopted technology for secure interconnections, dividing locally connected computers into various groups, where each group can implement a distinct access policy. VLANs provide the elasticity to manage groups and constrain access privileges. Our goal is to design a configuration that reduces the number of VLANs and satisfies the network policy.

Instead of focusing on one system parameter, our goal is to simultaneously optimize three network parameters: minimize the number of VLANS, maximize the permitted connections, and maximize the forbidden connections. This is a classical multiobjective optimization problem. One often-used strategy is a Pareto approach.

We introduce a novel Pareto based genetic algorithm, focusing both on developing a fitness measure which ranks possible VLAN solutions, and on designing affiliated genetic algorithm operators. The simulation compares the proposed approach with other search techniques, such as the Pacheco search. Experimental results suggest the Pareto based approach consistently finds concise and secure VLANs under various conditions, including increased numbers of interconnected devices.

Chapter 1: Introduction

The origin of the Internet begins with the Advanced Research Projects Agency Network (ARPANET). At the initial stage of ARPANET, computers in the Pentagon still were using basic card readers and card punching machines. Furthermore, these computers were heavy and expensive. As a program manager and office director at the Advanced Research Projects Agency, Lawrence Roberts wanted to connect two computers to make them work together. The result is the prototype of Local Access Network (LAN). Nowadays, LAN is still widely used in our daily life. Figure 1.1 shows a switched local network connecting the computer science and the mathematics departments.

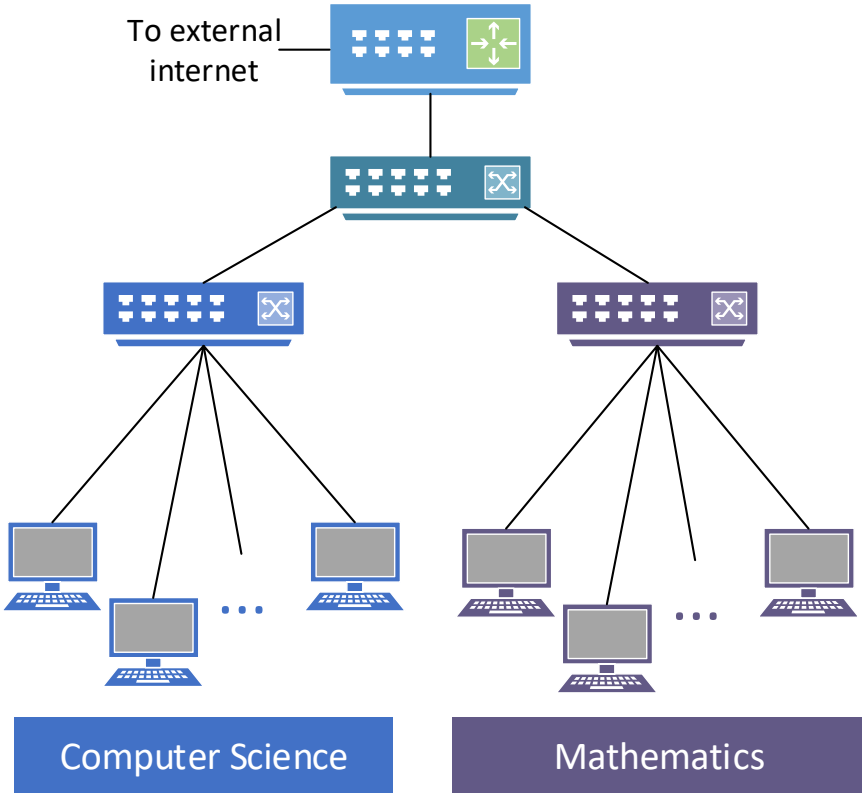
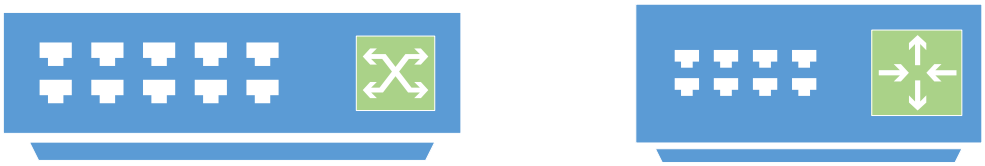


Figure 1.1: An institutional network connected together by three switches.



(a) A switch (b) A router

Figure 1.2: The router and switch icons. The left icon denotes a switch, and it has a sign containing two cross double end arrows, symbolizing the meaning of *switch*. The right icon represents a router. The router icon has a sign including four arrows. Two of the arrows direct toward inside, and the other two arrows points outside, implying the meaning of routing.

1.1 Switch and Router

The Open Systems Interconnection model (OSI model)[25] partitions a communication system into abstraction layers. The original version of the model had seven layers. Conventional switches (Figure 1.2a) works at the Data Link Layer (Layer 2 on the OSI model). Devices connected to the same switch compose a broadcast domain. If one of the devices send out a frame, the switch can broadcast the frame to all the other connected devices. However, the broadcast traffic sometimes contributes to network congestion[1]. Routers operates on the Network Layer (Layer 3 on the OSI model). They can send and receive messages across different kinds of networks. Moreover, routers can partition broadcast domain into several groups, and therefore a device can only broadcast messages to the devices in the same network segment. Switches are designed for exchanging data frames among devices in a local network, and therefore switches are widely adopted for constructing LANs [21].

1.2 Local Access Network

Computers, fax machines, printers and other kinds of devices in the computer science department are connected via a switch in Figure 1.1, as well as those devices in math-

[1]Network congestion is the reduced quality of service that occurs when a network node or link is carrying more data than it can handle.

ematics department. Another switch at higher hierarchy connects these two switched LANs. Devices in both the computer science department and the mathematics department can exchange information with each other. The hierarchy LANs access the external internet through the router.

The three switches can be substituted with routers in Figure 1.1. However, switches are generally cheaper than routers. Switches have more network interface and therefore can accommodate more devices than routers. Moreover, switches process frames more efficiently than routers. The hierarchically configured LANs in Figure 1.1 take advantages of switches, while there exist three disadvantages.

First, the broadcast domain are not isolated. If a new device, say 3-dimensional printer, connects to the LANs of the computer science department and another workstation needs to send a frame to the 3-dimensional printer, the switch will send a broadcast frame to discover the 3-dimensional printer, because the destination has not yet been learned by the switch. The broadcast frame will traverse not only the LAN of computer science department but also the LAN of mathematics department. If such broadcast traffic is constrained inside the computer science department scope, LAN performance would be improved. Furthermore, it is necessary to restrict the range of broadcast frames in LANs. For example, there exist some confidential files on the Chair's computer in computer science department. The network operator would rather not allow the Chair's frames reach a device maliciously running packet sniffing software.

Second, switches are not used efficiently. If number of devices in both computer science department and mathematics department were small, say 4, then a 10-port switch can suffice the connection requirement. If there are more departments are joint via switched LANs, devices can be integrated to a switch with more ports instead of distributing a switch to each department. However, there is still no traffic isolation

on a single switch merely with LANs.

Last, it is challenging to manage the users. If one of the computer in computer science department moves to the mathematics department, the physical network cable needs to be changed to another switch. The problem becomes more complicated if some device, say the 3-dimensional printer, is shared by both departments,

1.3 Virtual Local Area Network

A switch supporting **virtual local area networks (VLANs)** can handle the difficulties described in previous section. Multiple *virtual* local area networks can be configured over a single *physical* switch. Devices connected to the same VLAN compose a broadcast domain. The scope of a broadcast frame is limited in a VLAN. Devices with the same VLAN can communicate with each other directly. On the contrary, devices cannot communicate with devices in another VLAN directly.

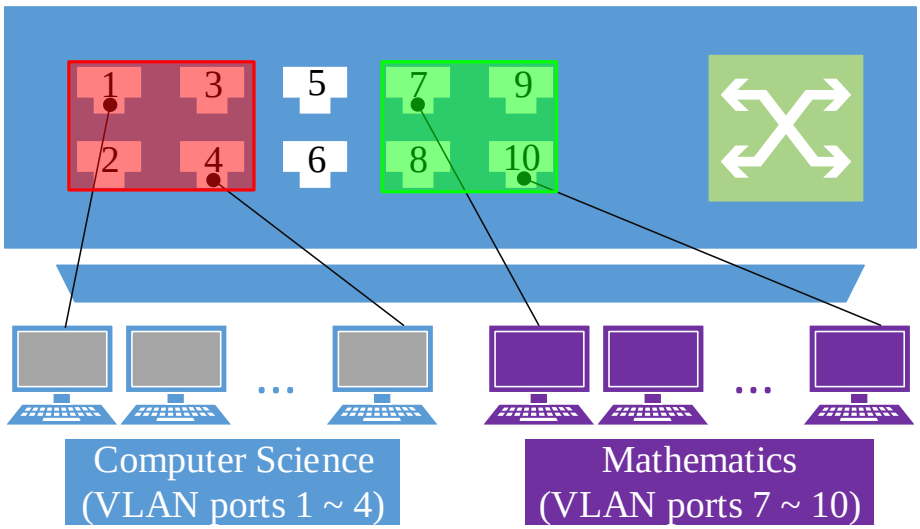


Figure 1.3: A single switch with 10 ports. Ports 1 to 4 are assigned to the computer science VLAN, while ports 7 to 10 belong to the mathematics VLAN. Ports 5 and 6 are unassigned. This VLAN solves all of the difficulties noted above—computer science and mathematics VLAN frames are isolated from each other, the two switches in Figure 1.1 have been replaced by a single switch.

VLANs can be partitioned based on switch ports. The switch ports in the same group constitutes a VLAN. There is a 10-ports switch in Figure 1.3. Ports from 1 to 4 are assigned to the computer science VLAN, while ports from 7 to 10 belong to the mathematics VLAN. Ports 5 and 6 are idle. Frames in the computer science department and mathematics department are isolated from each. Switches are utilized more efficiently by substituting the three switches in Figure 1.1 with a single switch. If a user in computer science department needs to be moved to the mathematics department, the network operator can simply declare the related port to the mathematics department so that the user's computer now belongs to the mathematics VLAN [21].

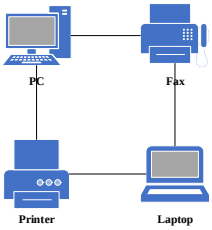
1.4 Security

However, the network becomes complicated in large companies or institutions. If the network operator simply configures a port to a user in some complex scenario, it can lead to information leakage, unprivileged access to some sensitive files. For example, a new staff member needs to be trained to become familiar with company's business. The staff member's computer connects to the VLAN that accesses only introductory videos, training manuals and the other non-sensitive media. In another VLAN, the department executive manager's computer interchanges information with secretary's computer. Assume after training, the new staff member is only qualified to work with the secretary. If the new staff member's computer is connected to the VLAN contains the executive department manager, he or she can access the shared files on department manager's computer. This introduces a hazardous confidentiality risk.

1.5 Flexibility

The connections between devices can be abstracted as Boolean matrices. In Figure 1.4a, four devices are connected in a ring topology. Their interconnections are denoted

in the table shown in Figure 1.4b. In the Boolean matrix, an entry of "1" means that the devices on the corresponding row and column connects with each other. An entry of "0" denotes that the related devices cannot communicate each other directly.



(a) Four devices connected with 4 VLANs.

	PC	Fax	Printer	Laptop
PC	1	1	1	0
Fax	1	1	0	1
Printer	1	0	1	1
Laptop	0	1	1	1

(b) Computer-Computer Boolean matrix.

	V_1	V_2	V_3	V_4
PC	1	1	0	0
Fax	1	0	1	0
Printer	0	1	0	1
Laptop	0	0	1	1

(c) Computer-VLAN Boolean matrix

Figure 1.4: A network containing 4 devices and its corresponding computer-computer Boolean matrix, as well as computer-VLAN Boolean matrix.

The four devices in Figure 1.4a can be connected with 4 VLANs. Similarly, we can also denote the relationship between computers and VLANs with Boolean matrix, where the rows represent devices, and the columns denote VLANs. As the name indicates, entries in the Boolean matrix are either 1's or 0's. If a device is in a VLAN, the entry at the intersection of device row and VLAN column is 1, otherwise, 0. As demonstrated in the table shown in Figure 1.4c, the personal computer connects with fax in VLAN_1, and connects with printer in VLAN_2. The laptop connects with fax and printer in VLAN_3 and VLAN_4, respectively.

In Figure 1.4a, we can see the personal computer (PC) cannot communicate with laptop directly. In Figure 1.5a, the network has been changed to allow the PC to exchange information with the laptop directly. The computer-computer Boolean Matrix and the computer-VLAN Boolean Matrix need to be updated as well.

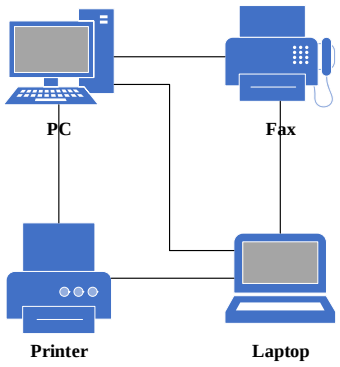
Apparently, we need to change some corresponding 0's in Boolean matrix into 1's. The updated computer-computer Boolean matrix corresponding to Figure 1.5a is shown in Figure 1.5b. There are two 0's in the computer-computer Boolean matrix, denoting the printer cannot exchange information with the fax directly, which is

consistent with the topological relationship shown in Figure 1.5a.

Similarly, the computer-VLAN Boolean matrix corresponding to Figure 1.5a is constructed in Table 1.1a. Focusing on the rows associated with the fax and printer, the Boolean entries cannot be 1 at the same time in the same column. This indicates the fax and the printer cannot communicate with each other directly. If we scrutinize the Table 1.1a thoroughly, we find the first column is the same as the third column, and the second column is the same as the forth column. In other words, we just need 2 VLANs instead of 4 VLANs to form the required network in Figure 1.5a.

After removing duplicate VLANs from Table 1.1a, the computer-VLAN matrix with least VLANs for the network in Figure 1.5a is described in Table 1.1b.

For convenience, we will refer to all kinds of interconnected devices, such as printers, laptops, fax machines as computers. The computer-computer Boolean matrix now is called the **policy** matrix, and the computer-VLAN Boolean matrix is referred to as the **configuration** matrix. In a policy matrix, 1 denotes the permitted connection between devices, and 0 indicates the forbidden exchange between devices. In



(a) The PC is connected with the laptop.

	PC	Fax	Printer	Laptop
PC	1	1	1	1
Fax	1	1	0	1
Printer	1	0	1	1
Laptop	1	1	1	1

(b) Computer-Computer Boolean matrix.

Figure 1.5: The PC and the laptop are connected in Figure 1.5a. The matrix entry at the lower corner in Figure 1.5b needs to be changed from 0 to 1, because the entry locates on the intersection of the row representing the laptop and the column denoting the PC. Similarly, the matrix entry at the upper corner needs to be updated as well. The computer-computer Boolean matrix is still symmetric.

	V_1	V_2	V_3	V_4
PC	1	1	1	1
Fax	1	0	1	0
Printer	0	1	0	1
Laptop	1	1	1	1

(a) Computer-VLAN Boolean matrix of Figure 1.5a

	V_1	V_2
PC	1	1
Fax	1	0
Printer	0	1
Laptop	1	1

(b) Simplified Computer-VLAN Boolean matrix of Figure 1.5a

Table 1.1: There are four entries in Table 1.1a being changed from 0's to 1's, the red 1's. Actually, we can merely change one of the four entries from 0 to 1, sufficing the connection requirements between the PC and the laptop. After updating all the four entries, there appears duplicate VLANs. The number of VLANs can be reduced by eliminating these duplication.

the thesis context, we define the number of 1's in a policy matrix as **feasibility**, and the number of 0's in a policy matrix as **confidentiality**.

Given a policy matrix, finding the configuration matrix with the least number of VLANs that meets the specified safety criteria is an NP-complete task [6]. The mathematical description of the problem is introduced in next chapter, and it establishes the relationship between the policy matrix and the configuration matrix.

The thesis is organized as follows. Mathematical description of the problem is stated in Chapter 2. General definition and terminology of typical genetic algorithm are explained in Chapter 3. In Chapter 4, we briefly summarize related work for exploring the same problem. Chapter 5 explores the problem from the view of maximum clique. Next, we present our ameliorated genetic algorithm in Chapter 6. The performance of our scheme and comparison among the other approaches are described in Chapter 7. Finally, conclusions and future work are routined in Chapter 8 and Chapter 9, respectively.

Chapter 2: Mathematical background

2.1 Boolean matrix multiplication

A mathematical formulation of the problem to be solved is presented here. Presume that there are n computers in a network. The interconnection between these computers (i.e. policy) is defined by a Boolean matrix $\mathbf{P}$, of size $n \times n$. If an entry $p_{ij} = 1$ then an exchange between computers i and j is permitted. Otherwise, the exchange is not possible.

Assume k VLANs constitute the network. Each of these VLANs permits communication between two or more computers. Let the matrix $\mathbf{C}$ be the configuration matrix, of size $n \times k$. If $c_{ij} = 1$ then the computer i belongs to the VLAN j. Otherwise, VLAN j does not include computer i.

The matrix $\mathbf{P}$ establishes the policy of the network. The matrix $\mathbf{C}$ is the configuration associated with the policy $\mathbf{P}$. Ideally, there exists the following relation between the Boolean matrices $\mathbf{P}$ and $\mathbf{C}$:

$$\mathbf{P} = \mathbf{C} \otimes \mathbf{C}^\top,$$

where $\mathbf{C}^\top$ is a transposed matrix $\mathbf{C}$, symbol $\otimes$ denotes Boolean matrix multiplication, which is a form of matrix multiplication based on the rules of Boolean algebra. Boolean matrix multiplication computes the entries of matrix $\mathbf{P}$ by the following expression: $p_{ij} = \bigvee_{j=1}^{n} (c_{ij} \wedge c_{ji})$. In other words, the Boolean matrix $\mathbf{P}$ can be calculated by substituting nonzero entries with 1 in $\mathbf{P}'$, where $\mathbf{P}'$ is the dot product of $\mathbf{C}$ and $\mathbf{C}^\top$, i.e. $\mathbf{P}' = \mathbf{C} \times \mathbf{C}^\top$.

Consider the following example. There are 5 computers in 3 VLANs. The relationship between the policy matrix $\mathbf{P}$ and the configuration matrix $\mathbf{C}$ are shown in

$$\mathbf{C} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \quad \mathbf{C}^\top = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

(a) Configuration matrix and its transpose

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Policy matrix $\mathbf{P}$

Figure 2.1: This figure shows the configuration matrix, its transpose and the policy matrix generated by Boolean matrix multiplication.

Figure 2.1. It is easy to see that matrix $\mathbf{P}$ is symmetric. Notice in this case, the configuration determines the policy. There are at most $\frac{n\times(n-1)}{2}$ pairwise connections among n computers. We can set a VLAN for each pair of computers. In other words, we can always find a configuration matrix that matches the policy without considering optimization. As shown in the previous example in chapter 1 (Figure 1.5a on page 7), there exist more than one valid configuration matrices that can generate same policy as the given one. The challenge is how to find the configuration matrix with the least number VLANs among all the reasonable candidates.

2.2 Isomorphism

There exist an important property in the Boolean matrix multiplication operation: a configuration matrix $\mathbf{C}$ generates the same policy matrix $\mathbf{P}$, no matter how the columns of $\mathbf{C}$ are permuted. The property is termed as **isomorphism**. For example, after switching the first column and the second column in Figure 2.1a, we get the matrix $\mathbf{C_p}$ and its transpose in Figure 2.2a. The matrix $\mathbf{C_p}$ is **isomorphic** with the matrix $\mathbf{C}$ in Figure 2.1a. After applying the Boolean matrix multiplication on matrix

$$\mathbf{C_p} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \quad \mathbf{C_p}^\top = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

(a) Configuration matrix and its transpose.

$$\mathbf{P_p} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Policy matrix produced by Boolean matrix multiplication

Figure 2.2: The configuration matrix after swapping the first and the second columns. The policy matrix stays the same no matter what the column order is.

$\mathbf{C_p}$ and its transpose, the policy matrix $\mathbf{P_p}$ in Figure 2.2b is the same as the one in Figure 2.1b.

The isomorphism property can be explained in two ways. First, the Boolean matrix multiplication contributes to the property. Without loss of generality, we select the entry, the red 0, at third row and first column in matrix $\mathbf{P_p}$ in Figure 2.2b. According to the principle of Boolean matrix multiplication, the red 0 is generated by the combination of the third row (red row in Figure 2.2b) in matrix $\mathbf{C_p}$, and the first column (red row in Figure 2.2b) in matrix $\mathbf{C_p}^\top$. The Boolean expression is: $(1 \wedge 0) \vee (0 \wedge 1) \vee (0 \wedge 1) = 0$. Before permutation, the corresponding entry is calculated as: $(0 \wedge 1) \vee (1 \wedge 0) \vee (0 \wedge 1) = 0$, the blue entry in Figure 2.1b on page 10. When the first column is switched with the second column in configuration matrix, the first two Boolean pairs, $(0 \wedge 1)$ and $(1 \wedge 0)$ are switched as well. These pairs are jointed with the operator $\vee$. The logical result stays the same because the operator $\vee$ has the commutative property. The other entries in the policy matrix can be analyzed in the similar way as well.

Second, the isomorphism property can be viewed from the network perspective.

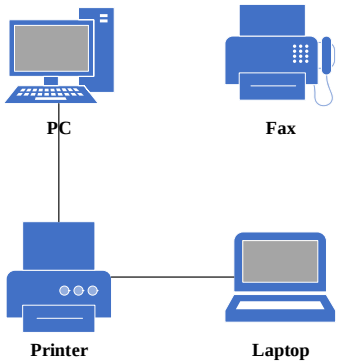
Each column in a configuration matrix represents a VLAN. The permutation of columns can be simply regarded as exchanging the labels of VLANs. The actual connection topology stays the same, and therefore there is no doubt that the policy matrix will stay the same.

If two or more configuration matrices are isomorphic, we regard these matrices as the same solution for the corresponding policy generated by Boolean matrix multiplication.

2.3 Problem boundary

From the discussion in Section 2.1 on page 9, there is a coarse upper boundary, $\frac{n(n-1)}{2}$, for how many VLANs a configuration can have given a policy containing n computers. In this section, we set reasonable constriction on the form of policy matrix, and give a smaller upper boundary for the number of VLANs in a configuration.

The policy matrix is used to describe the connection relationship between devices in a network. In some network, there may exist a device that has not been connected



(a) The fax is not connected with any other devices.

	PC	Fax	Laptop	Printer
PC	1	0	0	1
Fax	0	1	0	0
Laptop	0	0	1	1
Printer	1	0	1	1

(b) The row of fax only has one entry with value 1, and the column of fax only has one entry with value 1.

Figure 2.3: A deliberately constructed scenario: the fax is not connected with any other devices. The row of fax and the column of fax can be deleted from the policy matrix when we want to find the configuration matrix, because the fax does not belong to a VLAN.

with any other devices. Apparently, the individual device does not need to be in a VLAN. If a policy matrix contains such a device, there will be a row containing only one 1, as well as a column containing only one 1. For example, the fax in Figure 2.3a is not connected with any other devices. If we need to use policy matrix to describe the network, we can ignore such kind of devices. Now we have the constraint for a policy matrix: each row in a policy matrix has at least two 1 entries. In other words, a device directly connects with at least one other device in a network.

If all the computers are connected with each other, then only one VLAN is needed. If there are two computers connecting with each other but disconnecting with any other computers, then it requires a separate VLAN to hold these two computers. The more such individual pair of computers appears, the more number of VLANs a configuration matrix has. If there is no more than 2 computers connecting with each other directly in a network, the network forms a *complete bipartite graph*. In the mathematical field of graph theory, a complete bipartite graph or biclique is a special kind of bipartite graph where every vertex of the first set is connected to every vertex

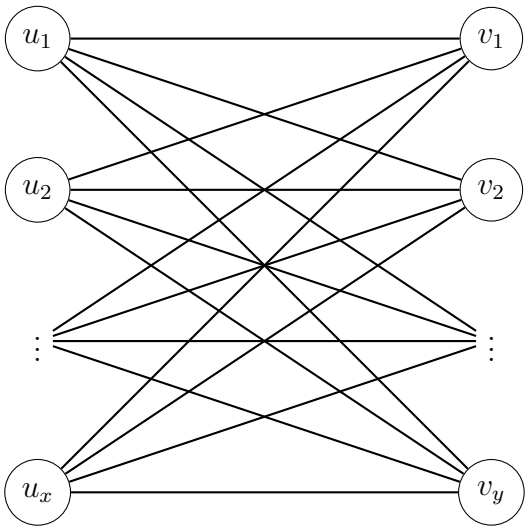


Figure 2.4: A complete bipartite graph. Vertices are split into two sets. Vertices in one set connect every vertex in another set. Vertices are not connected if they are in the same set.

of the second set [1].

In Figure 2.4, n vertices are in the complete bipartite graph. There are x vertices in the set $U = \{u_1, u_2, \cdots, u_x\}$. The other y of the vertices are in the set $V = \{v_1, v_2, \cdots, v_y\}$. Each vertex in the U set has y edges connected with itself. Totally, there are xy edges in the complete bipartite graph. We define the number of edges in a complete bipartite graph with n vertices as $f(n)$:

$$f(n) = xy, \begin{cases} x + y = n \\ 0 < x < n \\ 0 < y < n \end{cases} \tag{1}$$

Given n computers, each of them is connected with at least one computer. If the n computers and the connections between them form a complete bipartite graph, finding the upper bound of number of VLANs is the same as finding the maximum value of $f(n)$. All the $f_{max}(n)$ VLANs contains only two computers.

$$f_{max}(n) = \begin{cases} \frac{n^2}{4} & \text{n is even} \\ \frac{n^2-1}{4} & \text{n is odd} \end{cases} \tag{2}$$

If one pair of computers in the same partition set gets connected, then there are at least three computers connected with each other directly. At least two VLANs can merge together into one VLAN. In other words, the number of required VLANs can be reduced if the network with n computers cannot form a complete bipartite graph. Therefore, $f_{max}(n)$ is the upper bound for the number of VLANs in a network containning n computers. The upper bound is more precise than the coarse one, $\frac{n(n-1)}{2}$, though they are both $O(n^2)$.

With the acknowledgement of the upper bound, a brute force algorithm is developed in Algorithm 1. As Line 3 of Algorithm 1 indicates, when generating all the

	c_1	c_2
c_1	1	1
c_2	1	1

(a) Policy

	v
c_1	0
c_2	0

(b) 00

	v
c_1	0
c_2	1

(c) 01

	v
c_1	1
c_2	0

(d) 10

	v
c_1	1
c_2	1

(e) 11

Table 2.1: A network contains two computers, $n = 2$. The brute force algorithm generates configurations matrices with 2 rows and 1 column at the beginning. There are two entries in a configuration matrix. Each entry can either be 0 or 1, and therefore there are four possible configuration matrices of size 2×1.

possible configuration matrices of size $n \times i$ (n rows and i columns), where i is limited by $f_{max}(n)$. The Boolean entries changes from all 0's to all 1's. For example, there is a policy matrix contains two computers, $n = 2$, in Table 2.1a. The column number i of configuration matrix starts from 1. There are two Boolean entries in each configuration matrix of size 2×1. After concatenating the entries in each configuration matrix, we get 00, 01, 10 and 11. If we regard these string as binary numbers, their decimal values are 0, 1, 2 and 3, respectively.

Generally, for generating all possible configuration matrices of size $n \times i$, we can convert each decimal in the interval $[0, 2^{n \times i})$ into a binary string of length $n \times i$, and then fill each pre-allocated configuration matrix with the binary string.

From our experiments, when the computer number n is less than 6, the brute force algorithm can find a configuration matrix with least number of VLANs within 1 second. However, it takes more than 2 minutes to search for the optimal configuration matrix when there are 6 or more computers in a policy matrix. If the problem size is reasonably small, say $n < 6$, we can apply the above brute force algorithm to find the optimal configuration matrix for every possible policy matrix, and then store and compare the results, as has been done in Table 2.2. However, the number of possible policy matrices for n computers is $2^{\frac{n \times (n-1)}{2}}$. Hence, the time and space complexities for Algorithm 1 are exponential. Searching for the solution in this fashion is not computationally feasible.

Algorithm 1: Brute force method for finding the optimal configuration matrix

Data: Policy matrix $\mathbf{P}$
Result: Configuration matrix $\mathbf{C}$

$n \leftarrow$ first dimension of $\mathbf{P}$
$fn \leftarrow 0$ // upper bound for num of VLANs
if *n is even* **then**
  $fn \leftarrow \frac{n^2}{4}$
else
  $fn \leftarrow \frac{n^2-1}{4}$
end
for $i \leftarrow 1$ **to** fn **do** // column i changes from 1 to n
  size $\leftarrow n \times i$ // size of configuration matrix
  for $j \leftarrow 0$ **to** 2^{size} **do** // all possible Boolean combinations
    bits $\leftarrow$ DecimalToBinary(j, size)
  end
  $\mathbf{C} \leftarrow$ Reshape(bits, n, i) // arrange Boolean entries in matrix
  $\mathbf{P}' = \mathbf{C}' \otimes \mathbf{C}'^{\top}$ // Boolean matrix multiplication
  if $\mathbf{P}'$ *is the same as* $\mathbf{P}$ **then**
    return $\mathbf{C}$
  end
end

Heuristic methods have been explored for effectively solving this problem. One of these methods that has found wide application for solving similar optimization problems is **Genetic Algorithm** (GA). Terminologies and definitions of GA are reviewed in the following chapter.

Policy				Configuration			
	c_0	c_1	c_2		v_0	v_1	v_2
c_0	1	0	0	c_0	1	0	0
c_1	0	1	0	c_1	0	1	0
c_2	0	0	1	c_2	0	0	1
	c_0	c_1	c_2		v_0	v_1	
c_0	1	0	0	c_0	1	0	
c_1	1	1	1	c_1	1	1	
c_2	0	0	1	c_2	0	1	
	c_0	c_1	c_2		v_0	v_1	
c_0	1	0	1	c_0	1	0	
c_1	0	1	0	c_1	0	1	
c_2	1	0	1	c_2	1	0	
	c_0	c_1	c_2		v_0	v_1	
c_0	1	0	0	c_0	1	0	
c_1	0	1	1	c_1	0	1	
c_2	0	1	1	c_2	0	1	
	c_0	c_1	c_2		v_0	v_1	
c_0	1	1	1	c_0	1	1	
c_1	1	1	0	c_1	1	0	
c_2	1	0	1	c_2	0	1	
	c_0	c_1	c_2		v_0	v_1	
c_0	1	1	0	c_0	1	0	
c_1	1	1	1	c_1	1	1	
c_2	0	1	1	c_2	0	1	
	c_0	c_1	c_2		v_0	v_1	
c_0	1	0	1	c_0	1	0	
c_1	0	1	1	c_1	0	1	
c_2	1	1	1	c_2	1	1	
	c_0	c_1	c_2		v_0		
c_0	1	1	1	c_0	1		
c_1	1	1	1	c_1	1		
c_2	1	1	1	c_2	1		

Table 2.2: All possible policy matrices for 3 computers and related configuration matrices. The order of columns in configuration matrices can be permuted. If 2 configuration matrices are the same after permutation of their columns, they are isomorphic.

Chapter 3: Genetic Algorithm

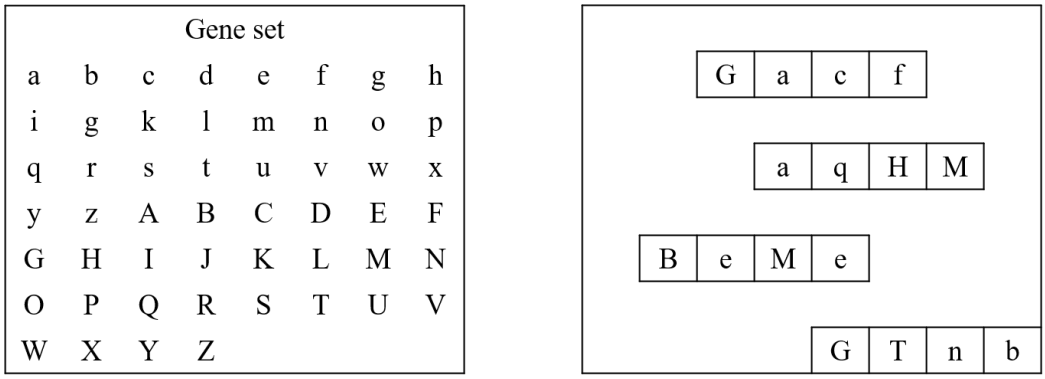
A genetic algorithm is one of the searching techniques for discovering good, sometimes optimal solutions to problems [3, 9, 10]. Often, the number of potential alternatives in the solution space of these problems is exponential. The genetic algorithm simulates the evolution process in biology by continuously generating new candidates, ranking them according to how well the candidates fit the problem's requirement, and improve the best candidates.

The candidate solution in genetic algorithm is regarded as an *individual*. The term *chromosome* is used to denote the encoded information related to the problem of an individual. Chromosomes are constituted by *genes*, where the value of genes can be different. All the possible values that a gene can have forms a *gene set*. Typical genetic algorithm designs a function, f, to produce a numerical score associated with each individual for evaluation. The value is called *fitness*, and the function is called *fitness function*. The better fitness a candidate possesses, the more similar to an optimal solution a candidate is. A typical genetic algorithm are composed of three process: *reproduction*, *crossover* and *mutation*. The overall fitness of generated individuals can be improved via the three procedures.

We introduce a simple example about guessing the word, *Gene*, with genetic algorithm to illustrate related terminologies.

3.1 Initialization

At the beginning, the genetic algorithm needs to initialize the first generation. Each candidate in the first generation represents a possible solution. The first generation is randomly initialized for enriching the diversity of candidates in the searching space.



(a) Gene set for guessing a word (b) The first generation

Figure 3.1: In the initialization process, the gene set includes both the lower case and upper case alphabets. There are four strings in the first generation.

More chances a genetic algorithm can find an optimal solution in a generation with larger population size. However, a genetic algorithm potentially consumes more computational resources, like memory, storage, power and time, etc., to search for a good solution in a large population.

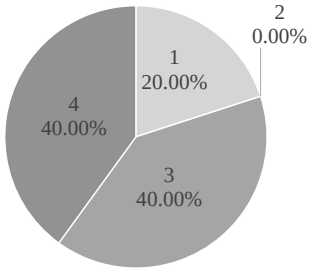
Assume the length of the word, *Gene*, can be obtained by the genetic algorithm. Possible genes are lower case letters and upper case letters. The gene set is shown in Figure 3.1a. The genetic algorithm generates strings containing four randomly selected letters from the gene set. There are four candidates in the first generation, Figure 3.1b. Besides generating new candidates, the fitness of each candidate is produced as well. In this example, the fitness value is the total number of identical letters in both the candidate and the intended word at the same position. The fitness values of the four candidates are: $f(\text{Gacf}) = 1$, $f(\text{aqHM}) = 0$, $f(\text{BeMe}) = 2$, $f(\text{GTnb}) = 2$.

3.2 Reproduction

Reproduction is a process that individuals with higher fitness values have a higher probability to be selected for producing one or more offspring in the next generation.

No.	Candidate	Fitness	% of Total
1	Gacf	1	20
2	aqHM	0	0
3	BeMe	2	40
4	GTnb	2	40
Total		5	100.0

(a) Candidates and their fitness percentage



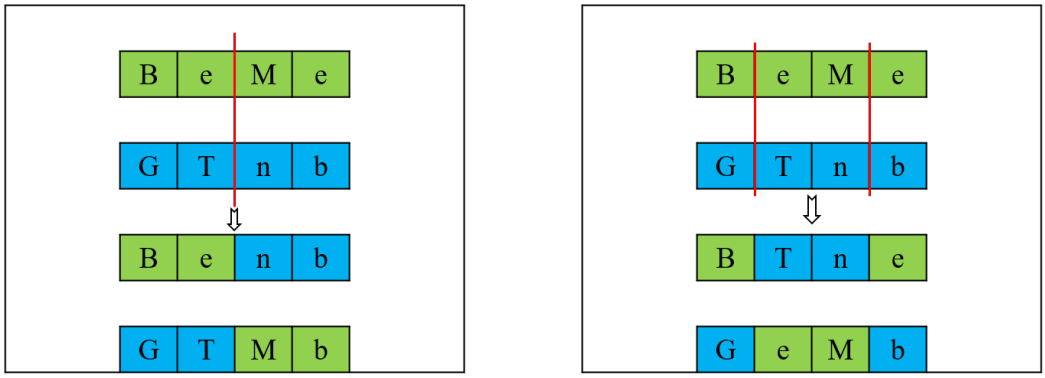
(b) Roulette wheel

Figure 3.2: The slots in the roulette wheel are sized in proportion to fitness. The sample wheel is constructed for the candidates in Table 3.2a

Reproduction operator simulates the natural selection introduced in the biological theory of evolution [2]. The fittest individual among all candidates will survive through the reproduction process. The typical way to implement the reproduction operator in algorithm is the Roulette Wheel method [3]. The overall fitness of the four strings in the first generation is 5. The four candidates' fitness and their percentages in the first generation are shown in the table inside Figure 3.2a. The sector area in roulette wheel shown in Figure 3.2b is commensurate with its corresponding fitness percentage. Once the genetic algorithm wants to reproduce, it just needs to spin the weighted roulette wheel. All the selected individuals enter the mating pool. In this way, the individual with a higher fitness are more likely to have more offspring in the next generation.

3.3 Crossover

Crossover is a process that new individuals are generated by combining the genetic information of two selected parent individuals from the mating pool in reproduction process. There is a probability threshold, p_c, controlling the frequency of the crossover process. Genetic information on chromosomes of two individuals can exchange at a single point, or at multiple points concurrently. As the name indicates, a single-point



(a) Single-point crossover (b) Multi-point crossover

Figure 3.3: The sing-point crossover operator swaps genetic information starting from a single randomly selected point, while the multi-point crossover operator swaps genetic information on segments between one or more pairs of randomly selected points.

crossover operator randomly selects a crossover point and then swaps the tail of two parents to produce new offspring. The multi-point crossover operator randomly selects several pairs of crossover points, and the segments within each pair of crossover points are swapped to generate new offspring [19]. The single-point crossover and multi-point crossover process are illustrated in Figure 3.3a and Figure 3.3b, respectively. Crossover operator can help produce individuals with new genetic sequences and therefore enrich the diversity of chromosomes in the next generation. In other words, crossover adds different candidates into the searching space.

3.4 Mutation

Mutation is a process that some gene values on individual's chromosomes are randomly changed to the other different values in the gene set. The mutation operator simulates the biological process that there may be errors during the replication, insertion or deletion of genetic information segments. The effect of mutation over an individual can be good, bad or neutral, but it is indispensable in the genetic algorithm. The mutation process is regarded as the ultimate source of genetic variation.

Chapter 4: Related work

The problem of finding a good or even optimal configuration matrix for a given policy matrix was categorized to be a variant of the Boolean matrix factorization (BMF) problem in [6], and the problem was proved to be a NP-complete problem in [7]. Among all the heuristic methods for solving the BMF problems, the bio-inspired genetic algorithm outperforms the other method [4]. A genetic algorithm was presented in [18], and the fitness function is associated with the Euclidean distance between the initial configuration matrices and the optimal matrices. Rezaei et al. [13] proposed a modified algorithm with a specialized mutation operator for finding solutions for nonnegative matrix factorization (NMF) problem. However, these algorithms cannot ensure finding a configuration matrix whose policy matrix is consistent with the intended policy matrix, though they are efficient in factorizing several matrices in BMF and NMF problems. In other words, the configuration matrix found by these algorithms cannot completely meet the security criterion, and therefore these algorithms are not suitable for optimizing the VLANs. The authors of [20] took several traffic conditions into account while designing VLANs, but they do not put the security criteria into consideration. In [8], the developed algorithm is security oriented, while the implementation needs to depend on some security mechanisms, like, IPSec[1].

Pacheco[11, 16] investigated the application of genetic algorithms to discover the minimum number of VLANs. Pacheco's thesis considered the associated access rights, which are necessary for connectivity and security in a computer network. Saenko et al.[17] defined their fitness function based on the Euclidean distance between the candidate configuration matrix and optimal configuration matrix. Pacheco proposed

[1]In computing, Internet Protocol Security (IPsec) is a secure network protocol suite that authenticates and encrypts the packets of data to provide secure encrypted communication between two computers over an Internet Protocol network.

a *Hybrid* fitness function in [11] by combining Pareto ranking and Euclidean distance. They both count how many entries are consistent between a candidate solution and the optimal solution. The fitness value is calculated by the linear combination of permitted exchange and forbidden exchange. In other words, neither of their works consider the feasibility and the confidentiality separately. When one of the feasibility and the confidentiality is improved, the other one is potentially decreased. If these two objective are joint as a whole in a fitness function, there exist two possible scenarios during the optimization process: (1) neither of the feasibility and the confidentiality becomes optimal; (2) one of them becomes optimal while the other one is almost 0.

We propose a Pareto based genetic algorithm in this thesis. Instead of depending on a single value — **fitness** — to select an individual in the population of each generation, we define a partial order on three attributes of an individual to optimize multiple objectives, i.e. feasibility, confidentiality, and the number of VLANs.

Chapter 5: Maximum Clique View

In the mathematical area of graph theory, a clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete [5]. If we regard computers as vertices, then computers in the same VLAN are adjacent with each other. In other words, a VLAN is a clique.

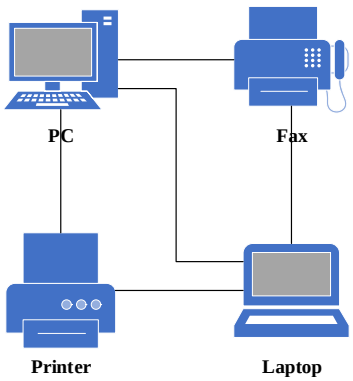
Naturally, this idea is an inspiration of another direction for designing the VLANs: the more computers a VLAN supports, the less VLANs a network needs. A greedy algorithm can be generally described as follows:

1. Find the maximum clique from current undirected graph
2. Put all the vertices from the maximum clique in a VLAN
3. Remove the maximum clique from current graph
4. If the graph is empty, stop; else, go to step 1

We start to implement the greedy algorithm from an simple example. As Figure 5.1a shows, it is easily seen that the size of the maximum clique is 3. Either the lower triangle or the upper triangle can form a clique.

5.1 Removing clique by vertices

If all the vertices of the upper triangle clique are removed from Figure 5.1a, the edges between these vertices will be removed automatically, then the residual graph only contains the printer vertex. The corresponding configuration would be as shown in Table 5.1. There is only one device, printer, in the second VLAN (i.e. V_2 in Table



(a) Maximum clique can be found by inspection.

	V_1	V_2
PC	1	1
Fax	1	0
Printer	0	1
Laptop	1	1

(b) Optimal configuration matrix.

Figure 5.1: A simple network and its configuration matrix.

5.1), and therefore the printer cannot communicate with the other devices. The configuration cannot satisfy the connection requirement in Figure 5.1a.

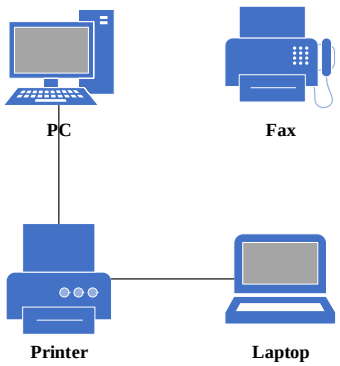
The example shows that, given a network, if cliques are removed by vertices, the greedy algorithm cannot generate a valid configuration matrix.

5.2 Removing clique by edges

If all the edges of the upper triangle clique are removed from Figure 5.1a, the residual graph is updated in Figure 5.2a. The maximum clique in Figure 5.2a can be either the $\{pc, printer\}$ or the $\{printer, laptop\}$ subgraph. The corresponding configuration matrix is shown in Figure 5.2b. The configuration satisfies the feasibility and confidentiality requirements in Figure 5.1a. However, as shown in Figure 5.1b, two VLANs

	V_1	V_2
PC	1	0
Fax	1	0
Printer	0	1
Laptop	1	0

Table 5.1: Incorrect resulting configuration matrix if maximum clique vertices are removed arbitrarily. Here the vertices should be kept when removing the maximum clique.



(a) Residual undirected graph after removing clique by edges.

	V_1	V_2	V_3
PC	1	1	0
Fax	1	0	0
Printer	0	1	1
Laptop	1	0	1

(b) The configuration matrix generated by continuously removing edges from maximum clique.

Figure 5.2: Maximum clique are removed from graph by edges. The generated configuration matrix has 3 VLANs, 1 more than the optimal answer in Figure 5.1b.

can suffice the network, instead of 3 VLANs in Figure 5.2b.

5.3 Retaining shared edges

Considering the process of the above removal methods, one may want to keep the edges if these edges are shared by another clique. For example, the edge between the PC and the laptop in Figure 5.1a on page 25 is shared by the upper clique and the lower clique. If the edge is kept while removing the upper triangle clique, then the next available clique is the lower triangle clique, and the algorithm will generate a configuration matrix with just 2 VLANs. It seems this strategy can find the optimal configuration matrix given a policy. However, if the shared edges are retained when removing the maximum clique, the algorithm runs forever on Figure 5.3, page 27. Vertices with different colors forms a clique in Figure 5.3, and the size of clique is 3. Every edge in a maximum clique is shared by another clique with the same size. If the shared edges are retained, then none of the maximum clique gets removed from the network.

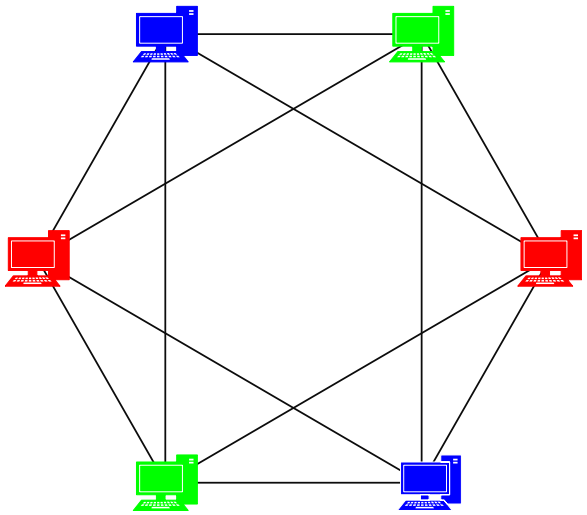


Figure 5.3: A tripartite network. The computers are partitioned into 3 sets. Computers in the same set have the same color, and they are net connected. Computers connected with the computers in another set directly.

5.4 Maximum clique removal algorithm

Among the three removal methods, removing clique by edges can generate a valid configuration matrix. The greedy algorithm adopts removing clique by edges is stated in Algorithm 2.

	c_1	c_2	c_3	c_4	c_5	c_6
c_1	1	1	1	0	1	1
c_2	1	1	1	1	0	1
c_3	1	1	1	1	1	0
c_4	0	1	1	1	1	1
c_5	1	0	1	1	1	1
c_6	1	1	0	1	1	1

(a) Policy matrix corresponding with the tripartite graph in Figure 5.3

	v_1	v_2	v_3	v_4
c_1	0	1	1	0
c_2	0	1	0	1
c_3	0	0	1	1
c_4	1	0	0	1
c_5	1	0	1	0
c_6	1	1	0	0

(b) Configuration matrix generated by the maximum clique removal algorithm

Table 5.2: Table 5.2a is the policy matrix for network in Figure 5.3. There are 8 cliques in the network. A configuration matrix can be generated by simply putting computers in each clique into a VLAN. The resulting configuration matrix would have 8 VLANs. However, with the maximum removal algorithm, a configuration matrix with 4 VLANs is produced in Table 5.2b.

.

Algorithm 2: Greedy Max Clique Algorithm

```
   Data: Policy matrix P
   Result: Configuration matrix C
 1 n = shape(P, 0)                                  // first dimension of matrix
 2 clq = MaxClique(P)                                    // an array of vertices
 3 while clq.size > 1 do
 4 |   common ← ∅
 5 |   for i changes from 1 to clq.size do
 6 |   |   degree = -1
 7 |   |   for column j changes from 1 to n do
 8 |   |   |   degree ← degree + P[clq[i]][j]
 9 |   |   end
10 |   |   if degree > clq.size - 1 then
11 |   |   |   common ← common ∩ clq[i]
12 |   |   end
13 |   end
14 |   for i changes from 1 to clq.size-1 do                  // remove max clique
15 |   |   for j changes from i + 1 to clq.size do
16 |   |   |   P[clq[i]][clq[j]] = 0
17 |   |   |   P[clq[j]][clq[i]] = 0
18 |   |   end
19 |   end
20 |   if clq.size > 1 then
21 |   |   vlan ← array(n, 0)
22 |   |   for i changes from 1 to clq.size do
23 |   |   |   vlan[clq[i]] = 1
24 |   |   end
25 |   |   C ← C ∩ vlan
26 |   end
27 |   clq = MaxClique(P)                              // Update the max clique
28 end
```

From the discussion in Section 5.2 on page 25, the greedy Algorithm 2 cannot assure generating the optimal configuration matrix, but it help find a good solution with less number of VLANs than just transferring each clique as a VLAN. For example, there are 8 cliques in the tripartite graph shown in Figure 5.3. If vertices in the tripartite graph represent computers, and the edges between vertices denote connections, the policy matrix for the network is presented in Table 5.2a. All the cliques in

Figure 5.3 are of size 3, any clique can be the maximum clique. An arbitrary method for generating a configuration matrix is to forming each clique in the tripartite graph as a VLAN. There would be 8 VLANs in the configuration matrix. However, after applying the greedy removal algorithm in 2, a configuration matrix with 4 VLANs is produced in Table 5.2b. It is because there is no need to put all the vertices belonging to a same clique into the same VLAN. The connection requirements between vertices in a same clique can be fulfilled in separate VLANs. If all the connections in a clique can be found in exist VLANs, then it is uncessary to create a new VLAN for forging the clique.

Chapter 6: Pareto based genetic algorithm

6.1 Pareto frontier

In this work, instead of minimizing on a single fitness value, we want to simultaneously optimize three parameters: minimize the number of VLANS, v, maximize the permitted connections, p, and maximize the forbidden connections, f. This is a classic multi-objective optimization problem. One often used strategy is a Pareto [3] approach.

The context in which we want to solve our multi-objective optimization problem is still a genetic algorithm. Instead of using fitness score to indicate the quality of a chromosome, we apply Pareto measure on tuples, $(p,\ f,\ v)$ to decide whether a

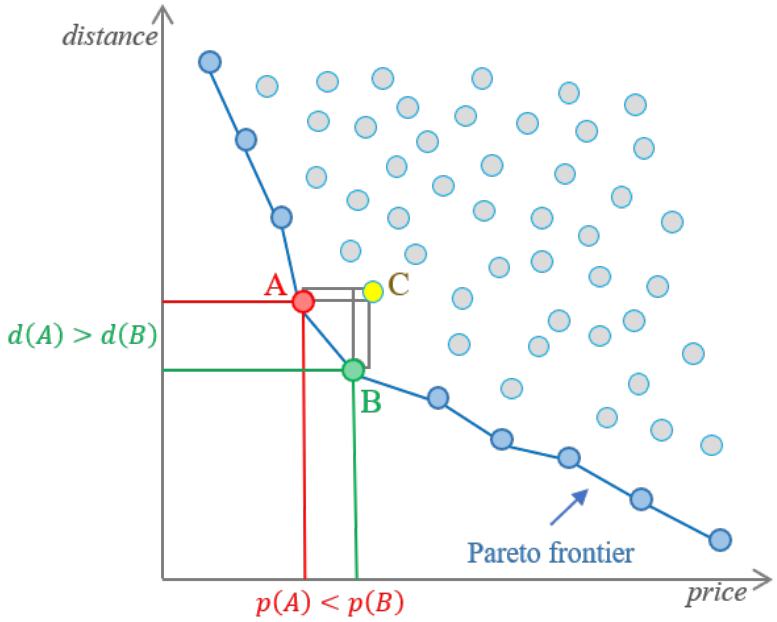


Figure 6.1: Pareto frontier of Myrtle Beach hotels. The horizontal axis represents price, the lower, the better. The vertical axis denotes the distance between beach and hotel, the lower, the better. Image adapted from wikipedia page.

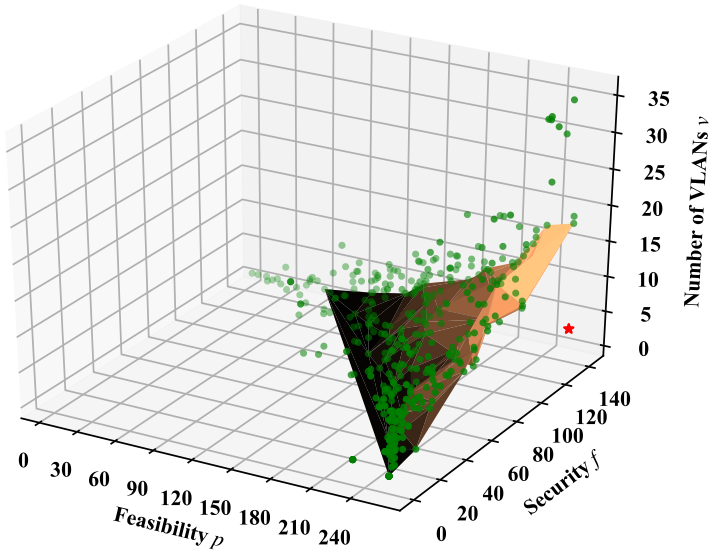


Figure 6.2: The surface with copper color is a 3-dimensional Pareto frontier. Each of the green points represent a configuration matrix. The red star denotes the optimal configuration matrix whose tuple is $(258, 142, 3)$.

chromosome can dominate another one. Domination in this respect refers to the quality of chromosomes: if object A is considered better than object B for each dimension in objects A and B, then object A *dominates* object B. For example, we can use two parameters to rank Myrtle Beach hotels: the distance to the beach, d, and price, p. The hotel is denoted by $H(d,\ p)$. Hotel A is better than hotel B if $d_A \leqslant d_B$ **and** $p_A \leqslant p_B$, which means hotel A is closer to the beach than B and A has a lower price than B. The two equations cannot hold at the same in a domination relationship. However, hotels are usually expensive if they are close to the beach, and hotels with a good price might be located several miles from the sea. The case hotel A located near the sea but expensive, $d_A \leqslant d_B$ and $p_A \geqslant p_B$, implies A and B cannot dominate each other, Figure 6.1. Moreover, both hotel A and B dominates hotel C in Figure 6.1.

Similarly, chromosome A representing one VLAN configuration is better than

chromosome B representing another VLAN configuration if $p_A \geqslant p_B$, $f_A \geqslant f_B$ and $v_A \leqslant v_B$, which means chromosome A has more permitted exchanges and forbidden exchanges than B, and chromosome A utilizes less VLANs than B, Figure 6.2. That is, if the related network is configured according to matrix A, it possesses more feasibility and security with consuming less resources than being configured by matrix B. The three equation cannot hold concurrently in a domination relationship. All the non-dominated chromosomes compose the Pareto frontier. The Pareto frontier is a subset of objects that cannot be dominated by the other objects in the set.

Using the Pareto frontier we modify the genetic algorithm operators of reproduction and mutation. We adopt a more flexible method for the reproduction process. One of our approaches is only choosing candidates from the Pareto frontier and putting them into the mating pool. The alternative plan is selecting parents from both the Pareto frontier and the non Pareto frontier, to generate the succeeding generation. We implement various mutation operators to accommodate fine-grained and coarse-grained adjustments.

6.2 Pruning the frontier

There are three parameters in a tuple (p, f, v). We present this kind of tuple in a 3-dimension coordinates in Figure 6.3. The proposed method takes a policy matrix as input, and therefore the number of permitted connections, p, and the number of forbidden connections, f, can be obtained by counting the 1s and 0s in the policy matrix. In other words, we can get the first two parameters in the tuple (p, f, v) of the optimal configuration matrix before we find it in search space.

Assume the point P on xy-plane has the same p and f as the optimal solution. All the solution matrices have the same p and f, but their number of VLANs can be different. The tuples of these solution matrices are located on the line PQ, as well as

the optimal configuration matrix, in Figure 6.3.

The Pareto frontier can be pruned by checking the cosine similarity between the candidate configuration matrix and the optimal configuration matrix. The value of p and f can be obtained from both given policy matrix and candidate configuration matrix, and therefore the cosine similarity between these two (p, f) pairs can be calculated. Let the threshold for cosine similarity be $cos(\theta)$. The candidates satisfying cosine similarity constraints are spread inside the cylinder $OEFGHR$. Moreover, the Pareto frontier can be pruned further. For example, assume a policy matrix containing 4 computers has 12 entries with value of 1, and 4 entries with value of 0. The values of first two parameters in the tuple are $p = 12$ and $f = 4$. A configuration matrix with $(p = 3, f = 1, v = 1)$ belongs to the Pareto frontier because it is not dominated

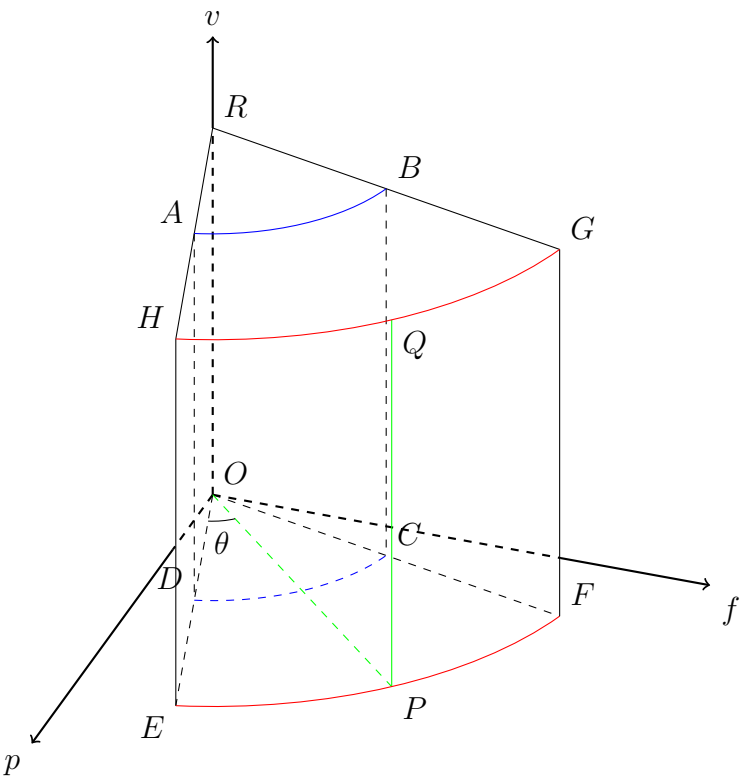


Figure 6.3: There is a 3-dimensional Cartesian coordinates system. The three dimensions denotes the parameters in tuple (p, f, v), respectively. The value θ is the angle degree between line OE and line OP.

Algorithm 3: Pareto frontier pruning algorithm

Data: Pareto frontier set and policy matrix
Result: Pruned Pareto frontier

```
(p_p, f_p) ← policy matrix
t_θ ← cos(θ)                                  // Cosine similarity threshold
t_p ← sqrt(p_p^2 + f_p^2)                     // Euclidean distance of the policy matrix
for configuration matrix in Pareto frontier set do
    (p_c, f_c) ← configuration matrix
    cs ← cosineDistance((p_p, f_p), (p_c, f_c))
    ed ← sqrt(p_c^2 + f_c^2)
    if cs < t_θ AND ed < t_p × t_e then        // t_e is Euclidean threshold
        remove the configuration matrix from the Pareto frontier set
    end
end
```

by the other configuration matrices for its number of VLANs, however, the Pareto frontier may not want to keep a candidate with such few permitted connections and forbidden disconnections. Another restriction based on Euclidean distance can be utilized to get rid of candidates whose p and f are far from the optimal solution. The notion (p_p, f_p) denotes the p and f values of the given policy matrix. The radius length of sector OEF represents the Euclidean distance between point $(p_p, f_p, 0)$ and the original point. The genetic algorithm needs to run on different policy matrices, and the Euclidean distances are possibly different. It is better to set a ratio threshold t_e instead of fixed value for the Euclidean distance. For example, if the value of t_e is 0.5 and the radius length in sector OCD is the half of radius length in sector OEF, then all the configuration matrices located in cylinder $ABCDOR$ are excluded from the Pareto frontier set. The residual candidates in the Pareto frontier are limited in the hollow cylinder $ABCDEFGH$, Figure 6.3. The pruning algorithm is detailed in Algorithm 3.

6.3 Initialization

The chromosome length is one of the objective we want to minimize, and therefore we do not want to generate matrices with a large dimension. The configuration matrices need to have same number of rows as the policy matrix does. As for the column dimension, Section 2.3, page 12, gives a boundary: a configuration matrix contains n computers can have at most $\frac{n^2}{4}$ VLANs if n is a even number, otherwise, $\frac{n^2-1}{4}$. With the access to policy matrix, we can check whether the policy is a bipartite graph, and then utilize the maximum clique removal algorithm to find a lower bound for the number of VLANs. Empirically, an network containing n computers has less than n VLANs. In the initialization process, the proposed algorithm generate matrices whose number of columns do not exceed the value n.

6.3.1 Randomization

Conventionally, it is reasonable to initialize the first population of a genetic algorithm using randomization methods. In our proposed approach, the random initiator utilizes

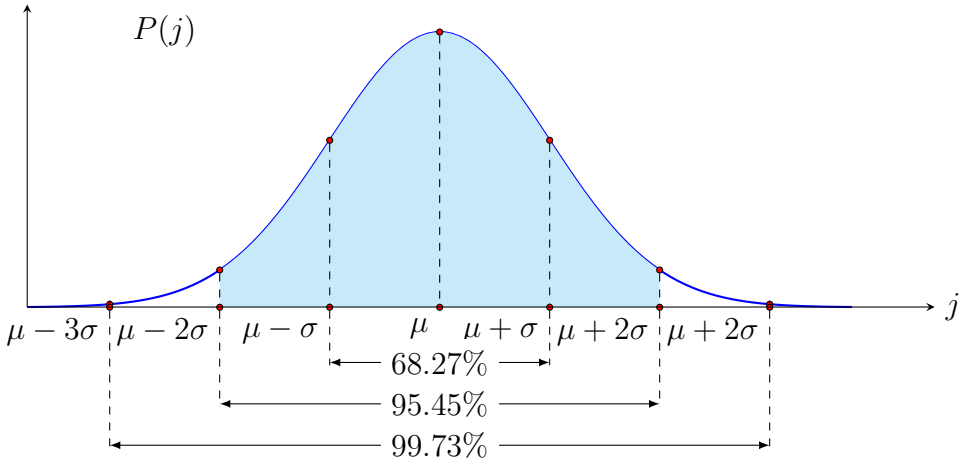


Figure 6.4: Generating matrices with their column number in normal distribution. The shadow area covers 95.45% possible values from 0 to n, where n is the number of computers. This figure is adapted from an image with *Creative Commons License*.

the fact that the number of columns in a configuration matrix seldom exceeds the number of its rows. It only generates matrices with dimension $n \times j$, where $j = 1, 2, \ldots, n$. Specifically, the random initiator varies the length to a normal distribution for assuring the diversity of chromosomes, Figure 6.4.

A normal distribution is a kind of probability function that describes how the values of a variable are distributed[24]. The probability density function of a normal distribution is

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2},$$

where the parameter μ is the mean of the normal distribution, the parameter σ is the standard deviation of the normal distribution, and the σ^2 is the variance of the distribution.

In our proposed method, for n computers, the parameter μ is set to be $(n+1)/2$, i.e. $\mu = \frac{(n+1)}{2}$. We make $n/5$ to be the standard deviation, i.e., $\sigma = \frac{n}{5}$. Moreover, the lower bound of chromosome length is restricted to $\mu - 2\sigma$, and the upper bound is limited to $\mu + 2\sigma$. According to the empirical rule (or 68–95–99.7 rule) in statistics [12], 95.45% of the values lies in the interval $[\mu - 2\sigma, \mu + 2\sigma]$. That is to say, the generated chromosomes suffice the diversity requirement.

6.3.2 Time complexity of Maximum Clique Removal Algorithm

Tarjan and Trojanowski presented an algorithm in [23] that finds a maximum clique in an n-vertex graph in time $O(2^{\frac{n}{3}}) = O(1.2599^n)$. Tarjan and Trojanowski's recursive backtracking method can be improved by eliminating some recursive calls when the found cliques in a call are suboptimal. Jian reduced the time complexity from $O(1.2599^n)$ to $O(2^{0.304n}) = O(1.2346^n)$ in [22]. Robson trades increased space

usage with decreased time, improving time to $O(2^{0.276n}) = O(1.2108^n)$ [14]. Robson's algorithm integrated a dynamic programming technique and a backtracking scheme. Maximum cliques in small graphs are found and stored in advance. Those precomputed solutions can help terminate some backtracking recursion calls early. The state-of-the-art algorithm by Robson [15] runs in time $O(2^{0.249n}) = O(1.1888^n)$. The time complexity is still exponential, $O(2^n)$.

The greedy max clique removal algorithm in Section 5.4 on page 27 needs to find a maximum clique in each iteration. As discussed in Section 2.3 on page 12, the number of VLANs in a network containing n computers can be $\frac{n^2}{4}$. In other words, there can be at most $\frac{n^2}{4}$ cliques in a n-vertex graph. The greedy maximum clique removal algorithm iterates at most $\frac{n^2}{4}$ times. In each iteration, the max clique find algorithm runs in time $O(2^n)$. The greedy maximum clique removal algorithm is limited by the upper bound $O(n^2 2^n)$.

The initialization process can leverage the greedy algorithm to set the maximal number of VLANs in generated configuration matrix. It is possible to produce a more tighter boundary than n, however, with the time complexity $O(n^2 2^n)$.

In our scheme, we adopt the randomization method in Section 6.3.1 on page 35 to initialize the number of VLANs for configuration matrices in the first generation. In the initialization process, N_{pop} configuration matrices are initialized with the value of 1 and 0 randomly. Next, the N_{pop} individuals are partitioned into 2 sets according to domination relationship with respect to their tuples, $(p,\ f,\ v)$. One set is composed of Pareto frontier, and the other set contains the rest of individuals.

6.4 Reproduction

The individuals in Pareto frontier have one or more good features of $(p,\ f,\ v)$. We want to keep and spread these features in the future generations. We try to select the

parents from Pareto frontier to generate a pair of individuals. However, if there are only few individuals in the Pareto frontier — in an extreme scenario, there are only 2 individuals in the Pareto frontier — then individuals in the next generation will always be produced by the 2 individuals. The diversity of next generation is limited if the mating pool contains few candidates.

We define a parameter to assure the reproduction process have sufficient candidates.

$$d_{pareto} = \frac{N_{pareto}}{N_{pop}}$$

The parameter d_{pareto} denotes the density of the Pareto frontier, and it is defined as the number of individuals in the Pareto frontier, N_{pareto} divided by the number of populations, N_{pop}.

If the density is high enough, greater than 0.4, then both the parents are selected from the Pareto frontier; otherwise, one of the parent from the Pareto frontier, and the other one from the non-frontier set. As for another extreme scenario, i.e., the Pareto frontier is empty, both parents are selected from the non-frontier. The selection process is expressed in Algorithm 4.

Algorithm 4: Reproduction

Data: Pareto frontier set and its complement
Result: 2 configuration matrices
if *paretoFrontier.size* $> N_{pop} \times d_{pareto}$ **then**
    parent1 ← randomly picked individual from Pareto frontier
    parent2 ← randomly picked individual from Pareto frontier
else if *paretoFrontier.size* > 0 **then**
    parent1 ← randomly picked individual from Pareto frontier
    parent2 ← randomly picked individual from non Pareto frontier
else
    parent1 ← randomly picked individual from non Pareto frontier
    parent2 ← randomly picked individual from non Pareto frontier
end

6.5 Crossover

Crossover generates new chromosomes by combining genetic sequences of the previously selected chromosomes Figure 3.3a, page 21. The combination process happens under the control of a probability, p_c, in Algorithm 7 on page 55. The crossover process shown in Figure 3.3a is a single point crossover. Our scheme adopts a multi-point crossover technique [19]. It consists of swapping random columns (VLANs) between the previously selected parents. As described in Section 6.2, columns represent the VLANs in a configuration matrix. Swapping VLANs between two chromosomes potentionally expands the search space by generating new configuration matrices with genetic sequences of both parents. Algorithm 5 shows a general implementation of the crossover process.

Algorithm 5: Crossover

```
   Data: Chromosome1 Chromosome2
   Result: Chromosome1 Chromosome2
   /* get available swap times                                    */
 1 swapTimes ← min(Chromosome1.colNum, Chromosome2.colNum)
   /* Generate swapTimes different numbers in the valid range     */
 2 columns1 ← random(swapTimes, Chromosome1.colNum)
   /* Generate swapTimes different numbers in the valid range     */
 3 columns2 ← random(swapTimes, Chromosome2.colNum)
 4 for i ← 1 to swapTimes do
 5 |   j ← columns1[i]
 6 |   k ← columns2[i]
 7 |   swap(Chromosome1[j], Chromosome2[k])
 8 end
```

The crossover can exchange the columns between the two previously selected chromosomes in any order, because the configuration matrix has the isomorphism property.

6.6 Mutation

For mutation, there is also a probability, p_m, associated with the process. The parameter denotes the possibility of how often this event will happen. The mutation process changes genetic information of a single chromosome. Three different kinds of mutators are introduced in our scheme. One of the three mutators will be applied according to the comparison between the probability p_m and a threshold t_m, in Algorithm 7 on page 55. The three mutators are illustrated in the following sub-sections.

6.6.1 Coin Flipping

The coin flipping mutator changes the Boolean entries in the configuration matrix by reverting the 1 to 0, or vice versa. The algorithm is quite straight and simple, as shown in Algorithm 6. There exists another threshold t_c to control the possibility on whether flip a Boolean entry or not. Otherwise, the mutator will flip all the entries, which is too aggressive.

Algorithm 6: Coin Flipping

```
   Data: Chromosome
   Result: Chromosome
 1 row ← Chromosome.rowNum
 2 col ← Chromosome.colNum
 3 for i ← 1 to row do
 4 |   for j ← 1 to col do
 5 |   |   prob ← random(0, 1) // The probability for changing this
   |   |       entry
 6 |   |   if prob > t_c then
   |   |   |   // Probability is greater than threshold
 7 |   |   |   Chromosome[i][j] = 1 - Chromosome[i][j]
 8 |   |   end
 9 |   end
10 end
```

6.6.2 Majority Voting

The majority voting mutator adds more control to the evolution direction. Once we get a configuration matrix, the corresponding policy matrix can be generated by Boolean matrix multiplication. According to the principle of Boolean matrix multiplication, if an entry in the configuration matrix, $\mathbf{C_{ij}}$, is changed (i.e., 1 becomes 0, or 0 becomes 1), then the entries in the i-th row and j-th column of related policy matrix are affected as well. The policy matrix is of size $n \times n$ for n computers network. The i-th row and j-th column of policy matrix contains total $2n-1$ entries, and they vote together to decide whether change the $\mathbf{C_{ij}}$ or not. For example, if $\mathbf{C_{ij}} = 1$ can make n or more (i.e. more than a half of $2n-1$) entries in the generated policy matrix the same as the corresponding entries in the required policy matrix, then n or more entries will vote to set the $\mathbf{C_{ij}}$ to be 1. On the contrary, if $\mathbf{C_{ij}} = 0$ can make n or more(i.e. more than a half of $2n-1$) entries in the generated policy matrix the same as the corresponding entries in the required policy matrix, then n or more entries will vote to set the $\mathbf{C_{ij}}$ to be 0. The majority voting mutator will mutate entries according to the votes. Because $2n-1$ is an odd number, there is no need to worry about a tie.

The majority voting method is detailed in Algorithm 7.

6.6.3 Column Cropping

The column cropping mutator is simple and straightforward. It randomly deletes a column from a chromosome, or generates a column, randomly filling each of its entries with 1 or 0, and then inserting the column into a chromosome.

Some programming languages, for example *python* and MATLAB™, have already implemented convenient interfaces or methods to complete the adding or deleting operation. The algorithmic details are shown in Algorithm 8.

Algorithm 7: Majority Voting

```
   Data: Chromosome
   Result: Chromosome
 1 row ← Chromosome.rowNum
 2 col ← Chromosome.colNum
 3 voteZero ← zeros(row, col)
 4 voteOne ← zeros(row, col)
   /* Approximate Policy matrix by Boolean matrix multiplication  */
 5 ap ← Chromosome ⊗ transpose(Chromosome)
 6 for i ← 1 to row do                                  // lower triangle
 7     for j ← 0 to i do
 8         if(1 == ap[i][j] and 0 == p[i][j])  for k ← 0 to col do
 9             if 1==chromosome[i][k] and 1==chromosome[j][k] then
10                 choice ← rand() % 2
11                 if 0 == choice then
12                     voteZero[i][k]++
13                 else
14                     voteZero[j][k]++
15                 end
16             end
17         end
18         if 0 == ap[i][j] and 1 == p[i][j] then
19             k ← = rand() % col
20             if 0 == chromosome[i][k] then
21                 voteOne[i][k]++
22             end
23             if 0 == chromosome[j][k] then
24                 voteOne[j][k]++
25             end
26         end
27     end
28 end
29 for i ← 0 to row do
30     for j ← 0 to row do
31         if voteOne[i][j] > voteZero[i][j] then
32             chromosome[i][j] ← 1
33         else if voteOne[i][j] < voteZero[i][j] then
34             chromosome[i][j] ← 0
35     end
36 end
```

Algorithm 8: Column Cropping

```
   Data: Chromosome
   Result: Chromosome
 1 row ← Chromosome.rowNum
 2 col ← Chromosome.colNum
 3 prob ← random()
 4 tmp ← ∅
 5 if prob ¿ 0.5 then
 6 |   c ← rand() % col
 7 |   tmp ← zeros(row, col-1)
 8 |   for i ← 1 to row do
 9 |   |   for j ← 1 to c do
10 |   |   |   tmp[i][j] == Chromosome[i][j]
11 |   |   end
12 |   |   for k ← c + 1 to col do
13 |   |   |   tmp[i][j] == Chromosome[i][k]
14 |   |   |   j ← j + 1
15 |   |   end
16 |   end
17 else
18 |   tmp ← zeros(row, col+1)
19 |   for i ← 0 to row do
20 |   |   tmp[i][col+1] = rand() % 2;
21 |   end
22 end
23 Chromosome ← tmp
```

Chapter 7: Test bed and Experiments

A test bed was developed to assess our proposed approach. The programming language C++ was adopted for implementations.

In this chapter, the performance of the proposed method will compare with Pacheco's hybrid method[11]. For fairness, the test bed is the same data set found in [11], also preserving the genetic algorithm parameters of population size and number of generations. From the last generation, we will choose the best solution from the Pareto frontier, and then compute its corresponding security coverage and record its VLAN number. Pacheco defines security coverage to evaluate the performance of a chromosome in [11]. We use the same measurement to compare Pacheco's Hybrid scheme and our Pareto method. The security coverage is calculated as the ratio of matches between generated policy matrix and the required matrix over the total number of matrix entries, i.e.,

$$S_c = \frac{matches}{n^2},$$

where n is the number of computers.

Table 7.1 shows the experimental setup for this research. The values of the genetic algorithm parameters are for selection, d_{pareto}, 0.4, for crossover probability, p_c, 0.8,

Configuration	Runs	Population Size	Generations
20 * 3	100	200	200
50 * 2	100	200	200
50 * 5	100	200	200
50 * 8	100	200	200
50 * 10	100	200	200
100 * 15	100	300	200

Table 7.1: Parameters configurations for experiments. In the first column, the $m * n$ denotes the dimension of the optimal configuration matrix. The second column denotes the experiments were run 100 times and the average result was recorded.

for mutation probability, p_m , 1.0, and for threshold, t_m, is 0.5. In the experiments, if the probability p_m is greater than t_m, the mutation process will do coin flipping, otherwise, it will do majority voting. Pacheco sets the crossover probability of her scheme to be 0.7, and the mutation probability to be 0.3. The test bed keeps Pacheco's setting on the crossover probability and the mutation probability.

7.1 Increasing Number of Computers

This set of experiments measures the performance of the algorithms as the number of the computers increases. Table 7.2 shows the number of VLANs, the security coverage and the number of generations of the Hybrid and the Pareto schemes.

From the security coverage in Table 7.2, the Pareto method is being able to find an acceptably good answer quickly without knowing what the right answer is, and the Pareto method outperforms the Hybrid method on all three configurations in security. From the first row of the Table 7.2, the Pareto method finds the optimal answer with 100% security coverage, while the solution found by the Hybrid method has 5% less security than optimal. Besides, the Pareto method runs less generations for finding the optimal solution. For the policy with 50 computers, with $2,500$ entries in a matrix, the Pareto method is still finding a more secure solution than the Hybrid. When the algorithm stops at the 200^{th} generation, the Pareto method achieves 93% security coverage, better than the Hybrid method. On the largest policy matrix among these three network, the Pareto method discovers the solution with 100% security coverage

Configuration	VLANs		Security		Generation	
	Hybrid	Pareto	Hybrid	Pareto	Hybrid	Pareto
20 * 3	3	3	95%	100%	25	13
50 * 5	2	18	86%	93%	35	200
100 * 15	2	15	96%	100%	25	24

Table 7.2: Performance for both algorithms of 3 different configurations

as well, 4% more than the Hybrid method.

7.2 Varying Number of VLANs

In this set of experiments, shown in Table 7.3, the Pareto method and the Hybrid method are tested on three different policies of 50 computers. The three configurations are of size 50×2, 50×8 and 50×10, respectively. For the 50×2 and 50×10 setting, the Pareto method finds good configurations satisfying security requirement, though the Pareto method uses 11 more VLANs than optimal in the 50×10 setting. For the 50×8 setting, the Pareto method finds a configuration with 96% security coverage within 200 generations, a little higher than the hybrid method. But the Pareto method required 25 more VLANs than the hybrid method to improve the security coverage.

Configuration	VLANs		Security		Generations	
	Hybrid	Pareto	Hybrid	Pareto	Hybrid	Pareto
50 * 2	2	2	98%	100%	195	14
50 * 8	2	27	92%	96%	30	200
50 * 10	2	19	94%	100%	30	187

Table 7.3: Performance for both algorithms of 3 different configurations

From the discussion above and Table 7.3, there is evidence that the Pareto method can find a solution with more security than the hybrid method in 200 generations. On the 6 experimental data sets the Pareto method can find solutions with 100% security coverage on 4 of them. Moreover, the Pareto method does not leverage any information from the optimal configuration matrix. In other words, the Pareto method only feeds the policy matrix into the genetic algorithm, while the Hybrid method needs both the policy matrix and its corresponding optimal configuration matrix.

7.3 Aiming at finding a solution

It appears that the Pareto method requires more generations to find a solution with 100% security coverage. Thus, the termination condition for the Pareto method is modified. Instead of stopping at 200 generations, the Pareto method halts when the best chromosome in Pareto frontier keeps the same for last 10 generations. In the experiment, we maintain a list to record the best chromosome in the Pareto frontier of last 10 generations. If the 10 chromosomes are the same, then the genetic algorithm terminates. The results from these experiments are shown in Table 7.4.

n computers	generation	configurations	security coverage	optimal solution
20	13	20 * 3	100%	20 * 3
20	18	20 * 5	100%	20 * 5
20	179	20 * 12	100%	20 * 8
20	262	20 * 18	100%	20 * 10
50	14	50 * 2	100%	50 * 2
50	237	50 * 5	100%	50 * 5
50	316	50 * 27	100%	50 * 8
50	187	50 * 19	100%	50 * 10
100	274	100 * 17	100%	100 * 8
100	657	100 * 29	100%	100 * 15

Table 7.4: Using the modified termination condition, the number of generations for the modified Pareto method are shown to achieve 100% security coverage.

In the experiments, we randomly generate a configuration matrix, and insure there are no duplicate VLANs in each configuration matrix. Next, the policy matrices are produced with these configuration by Boolean matrix multiplication. The Pareto method only takes the policy matrix as its input. Table 7.4 suggests the Pareto method can find a configuration which matches the required policy, though some of the configurations have more VLANs than the optimal ones.

7.4 The important mutator

We introduced several different mutators in the Section 6.6. This set of experiments aims at exploring which mutator makes a significant impact on the Pareto method. There are 3 mutators under consideration: coin flipping, majority voting and column cropping. At first, we only activate one of them in the mutation process. Then, we combine two of them. Finally, we apply all three mutators. Table 7.5 records the generations for finding a configuration. In the first column, *cf* is short for coin flipping. The *mv* denotes majority voting. Column cropping is abbreviated as *cc*.

In this set of experiments, the Pareto method runs at most 400 generations. If the top chromosomes in the last 10 Pareto frontier remains the same, the algorithm stops.

From Table 7.5, the column cropping mutator uses the least number of generations to meet the termination condition. However, as seen in Table 7.6, the column cropping mutator produces poor security coverage, on average 64.5%. From Table 7.5, comparing the generations between row of {*cf*} and {*cf, cc*}, or row of {*mv*} and {*mv, cc*}, we can see the column cropping mutator can help terminate the genetic algorithm in earlier generations. It is because the column cropping mutator can decrease the number of columns in configuration matrices. With activating the column cropping mutator, the algorithm can find the optimal chromosome with fewer generations.

Table 7.6 displays the security coverage of the best chromosome in the last Pareto frontier. If the Pareto method terminates at the 400^{th} generation, then the best chromosome is the top one over all 400 generations. Otherwise, the best chromosome is the top one in the last ten generations.

Examining Table 7.5 and Table 7.6, the Pareto method achieves average 99% security coverage by activating all three mutators. As the majority voting row in Table 7.6 shows, the Pareto method achieves 98.5% security coverage with only the

Generation	20 * 3	50 * 2	50 * 5	50 * 8	50 * 10	100 * 15	Average
cf	400	400	400	400	400	400	400
mv	101	49	289	389	218	400	241
cc	26	39	36	29	47	56	39
cf,mv	97	51	276	361	207	400	232
cf,cc	111	158	161	117	223	276	174
mv,cc	52	36	210	357	198	400	209
cf,mv,cc	13	14	237	316	187	400	195

Table 7.5: The stopping generations on 6 different combinations of three mutators. The Pareto method stops at the 400^{th} generation if the best chromosomes in last 10 Pareto frontier are not the same. In the first column, *cf* is short for coin flipping. The *mv* denotes majority voting. Column cropping is abbreviated as *cc*.

Security	20 * 3	50 * 2	50 * 5	50 * 8	50 * 10	100 * 15	Average
cf	95%	84%	79%	81%	77%	72%	81%
mv	100%	100%	100%	100%	100%	91%	98.5%
cc	73%	68%	71%	65%	59%	51%	64.5%
cf,mv	100%	100%	100%	100%	100%	91%	98.5%
cf,cc	76%	79%	75%	73%	63%	60%	71%
mv,cc	100%	100%	100%	100%	100%	92%	98.6%
cf,mv,cc	100%	100%	100%	100%	100%	94%	99%

Table 7.6: The security performance for the best chromosome in last Pareto frontier

majority voting mutator, 0.05% less than 99%. However, comparing the average generations in Table 7.5, the Pareto method consistently requires fewer generations with all three mutators than merely using the majority voting mutator.

From the analysis above, the majority voting mutator plays a significant role in the mutation process. The coin flipping mutator introduces randomness to the solution search space. The column cropping mutator is aggressive because it adds or deletes columns of entries in the configuration matrix. It can help accelerate the search process. The experiment shows it is a good choice to activate all three mutators in the mutation process.

Chapter 8: Conclusions

The thesis proposed a Pareto-based Genetic Algorithm to optimizing VLAN configurations. We explore the boundary of solution space, and discuss the problem from the maximum clique point of view. The VLAN design problem is a special kind of Boolean Matrix Factorization. The Boolean Matrix Factorization problem is well known as an NP-complete problem, as well as the maximum clique problem. For this reason, it is impossible to leverage the existing mathematical methods for solving this problem. Based on related work, it is proposed to use an improved genetic algorithm. The main contributes of the thesis are: (1) finding a tight upper bound of a configuration matrix size: $n \times f(n)$, where n is the number of computers, $f(n) = \frac{n^2-(n \mod 2)}{4}$; (2) considering the problem from the maximum clique view, and developing a greedy maximum clique removal algorithm to help with establishing an initial generation; (3) substitution of a single fitness value with multiple objectives to satisfy feasibility and confidentiality in VLAN design problem; (4) combination of Pareto based multiple objective optimization method and genetic algorithms; (5) implementation of three mutators operating the genetic information in various grain size; (6) examination of the significance of mutators and their combinations.

Experimental evaluation of the proposed genetic algorithm showed its competitive security. By comparing with Pacheco's results on the same data sets, the Pareto method can achieve better security coverage. Moreover, the experimental evaluation helps find the significant mutator, which improves the genetic algorithm.

Chapter 9: Future work

9.1 Parameters for GA

In the experimental evaluation, the parameters for the genetic algorithm N_{pop}, d_{pareto}, p_c, p_m, t_e are set to a value empirically. These values can affect the generations for finding a solution for the VLAN design problem. We will define a function: $f(n, N_{pop}, d_{pareto}, p_c, p_m, t_e)$ aims to find a good set of these parameters, and minimize the generations for finding solutions for a network with n computers.

9.2 Network structures

The network can be denoted as undirected graphs. The graphs can be dense or sparse. In our experimental evaluation, we did not intentionally run our proposed method on dense or sparse graphs individually. We will measure the performance of our algorithm on different kinds of graphs, and accommodate the genetic algorithm further in the future.

9.3 Directed network

The proposed algorithm focuses on undirected network. In reality, the computer connected to the network can send or receive data in just one way. The VLAN design problem becomes more complicated if the transmission between computers are not limited to, only send, receive, or both directions.

Bibliography

[1] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph Theory with Applications*. Macmillan, 1976.

[2] Charles Darwin. *On the Origin of Species by Means of Natural Selection Or the Preservation of Favoured Races in the Struggle for Life*. H. Milford; Oxford University Press, 1859.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman, Boston, MA, USA, 1st edition, 1989.

[4] Andreas Janecek and Ying Tan. Using population based algorithms for initializing nonnegative matrix factorization. In *Proceedings of the Second International Conference on Advances in Swarm Intelligence - Volume Part II*, ICSI'11, pages 307–316, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] R. Duncan Luce and Albert D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[6] P. Miettinen. Dynamic boolean matrix factorizations. In *2012 IEEE 12th International Conference on Data Mining*, pages 519–528, December 2012.

[7] Pauli Miettinen and Jilles Vreeken. Mdl4bmf: Minimum description length for boolean matrix factorization. *ACM Transactions Knowledge Discovery in Data*, 8(4), October 2014.

[8] Minli Zhu, M. Molle, and B. Brahmam. Design and implementation of application-based secure vlan. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 407–408, 2004.

[9] Melanie Mitchell. Genetic algorithms: An overview. *Complexity*, 1(1):31–39, 1995.

[10] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[11] Alina Pacheco. Using evolutionary algorithms to manage virtual local area networks. Master's thesis, Wake Forest University, Department of Computer Science, May 2019.

[12] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[13] Masoumeh Rezaei and Reza Boostani. Using the genetic algorithm to enhance nonnegative matrix factorization initialization. *Expert Systems*, 31(3):213–219, 2014.

[14] J. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425 – 440, 1986.

[15] John M Robson. Finding a maximum independent set in time $o(2^{n/4})$. Technical report.

[16] Alina Pacheco Rodriguez, Errin W. Fulp, David J. John, and Jinku Cui. Using evolutionary algorithms and Pareto ranking to identify secure local area networks. In *Proceedings of GECCO 2020*, accepted.

[17] Igor Saenko and Igor Kotenko. Design of virtual local area network scheme based on genetic optimization and visual analysis. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 5(4):86–102, December 2014.

[18] V. Snášel, J. Platoš, and P. Krömer. On genetic algorithms for boolean matrix factorization. In *2008 Eighth International Conference on Intelligent Systems Design and Applications*, volume 2, pages 170–175, 2008.

[19] William M. Spears and Kenneth A. DeJong. An analysis of multi-point crossover. In *Foundations of genetic algorithms*, volume 1, pages 301–315. Elsevier, 1991.

[20] X. Sun, Y. Sung, S. D. Krothapalli, and S. G. Rao. A systematic approach for evolving vlan designs. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.

[21] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Press, USA, 5th edition, 2010.

[22] Tang Jian. An $\mathcal{O}(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Transactions on Computers*, C-35(9):847–851, 1986.

[23] Robert Endre Tarjan and Anthony E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[24] Friedrich-Wilhelm Wellmer. *The Normal Distribution*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[25] Hubert Zimmermann. Osi reference model-the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

Appendix A: Pareto based Genetic Algorithm

Algorithm 9: Genetic Algorithm with Pareto Measure

Result: Chromosome matrix **C**

```
Randomly generate some chromosomes as the initial population;
while not reach the preset generation do
    Find the Pareto frontier of current generation;
    while population size of next generation < preset size do
        Selection: randomly select 2 chromosomes from Pareto frontier;
        probability ← random generator ;
        if probability > p_c then                    // p_c is the crossover rate
            Crossover 2 chromosomes;
        end
        probability ← random generator ;
        if probability > p_m then                    // p_m is the mutation rate
            if probability > t_c then                // t_c is coin clipping threshold
                Apply coin flipping on two chromosomes;
            else if probability > t_m then           // t_m is majority voting
             threshold
                Apply majority voting on two chromosomes;
            else
                Apply column cropping on tow chromosomes;
        end
        if chromosome 1 dominates chromosome 2 then
            Chromosome 1 goes to next generation;
        else if chromosome 2 dominates chromosome 1 then
            Chromosome 2 goes to next generation;
        else
            Both 2 chromosomes are added to next generation;
    end
    Shift to next generation;
end
```

Appendix B: C++ Code

```
#ifndef CHROMOSOME_H
#define CHROMOSOME_H


#include <vector>

class Chromosome {
private:
    std::vector<std::vector<int>> genes;
    int row;
    int col;
    int nperm;
    int nforb;
    int nvlan;

public:
    Chromosome();

    Chromosome(int row, int col);

    // bool operator==(const Chromosome &rhs) const;

    const std::vector<std::vector<int>> &getGenes() const;

    void setGenes(const std::vector<std::vector<int>> &genes);

    void coinFlip();

    void voteTune(int** p);

    void delColum();

    void trimGene();

    void updateMatrics(int **p);

    int getRow() const;
```

```
    void setRow(int row);

    int getCol() const;

    void setCol(int col);

    int getNperm() const;

    void setNperm(int nperm);

    int getNforb() const;

    void setNforb(int nforb);

    int getNvlan() const;

    void setNvlan(int nvlan);
};


#endif //CHROMOSOME_H
```

```
#include <cstdlib>
#include <random>
#include <algorithm>
#include <chrono>
#include "Chromosome.h"

Chromosome::Chromosome(int row, int col) : row(row), col(col) {

    srand(1);
    for (int i = 0; i < row; i++) {
        bool allZero = true;
        std::vector<int> tmp (col);
        for (int j = 0; j < col; j++) {
            tmp[j] = rand() % 2;
            if (1 == tmp[j]) {
                allZero == false;
            }
        }
        if (allZero) {
            int j = rand() % col;
```

```
            tmp[j] = 1;      // computer must be in a vlan
        }
        genes.push_back(tmp);
    }
}

Chromosome::Chromosome() {}

int Chromosome::getRow() const {
    return row;
}

void Chromosome::setRow(int row) {
    Chromosome::row = row;
}

int Chromosome::getCol() const {
    return col;
}

void Chromosome::setCol(int col) {
    Chromosome::col = col;
}

const std::vector<std::vector<int>> &Chromosome::getGenes() const {
    return genes;
}

int Chromosome::getNperm() const {
    return nperm;
}

void Chromosome::setNperm(int nperm) {
    Chromosome::nperm = nperm;
}

int Chromosome::getNforb() const {
    return nforb;
}

void Chromosome::setNforb(int nforb) {
    Chromosome::nforb = nforb;
```

```
}

int Chromosome::getNvlan() const {
    return nvlan;
}

void Chromosome::setNvlan(int nvlan) {
    Chromosome::nvlan = nvlan;
}

void Chromosome::setGenes(const std::vector<std::vector<int>> &genes)
   {
    Chromosome::genes = genes;
}

void Chromosome::coinFlip() {
    unsigned seed = std::chrono::system_clock::now().time_since_epoch
   ().count();
    std::default_random_engine gen (seed);
    std::uniform_real_distribution<double> rdis(0.0,1.0);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (rdis(gen) < 0.05) {
                genes[i][j] = 1 - genes[i][j];
            }
        }
    }
}

void Chromosome::voteTune(int** p) {

    int** ap = new int* [row];
    for (int i = 0; i < row; i++) {
        ap[i] = new int[row];
    }

    int** vzero = new int* [row];
    for (int i = 0; i < row; i++) {
        vzero[i] = new int [col];
    }

    int** vone = new int* [row];
```

```
for (int i = 0; i < row; i++) {
    vone[i] = new int [col];
}

for(int i = 0; i < row; i++) {
    for (int j = 0; j < row; j++) {
        for (int k = 0; k < col; k++) {
            ap[i][j] |= genes[i][k] & genes[j][k];
        }
    }
}

unsigned seed = std::chrono::system_clock::now().time_since_epoch
().count();
std::default_random_engine gen (seed);
std::uniform_int_distribution<int> idis(0, col-1); // [0, col-1]
for (int i = 0; i < row; i++) {
    for (int j = 0; j < row; j++) {
        if (i == j ) {
            if (0 == p[i][j]) {
                printf("Unrealistic policy\n");
                exit(-2);
            }
            else {
                if (0 == ap[i][j]) {
                    int k = idis(gen);
                    vone[i][k]++;
                }
            }
        }
        else {
            if (1==ap[i][j] && 0==p[i][j]) {
                for (int k = 0; k < col; k++) {
                    if (1==genes[i][k] && 1==genes[j][k]) {
                        int choice = rand()%2;
                        if (0 == choice) vzero[i][k]++;
                        else vzero[j][k]++;
                    }
                }
            }
            if (0==ap[i][j] && 1==p[i][j]) {
                int k = rand() % col;
```

```
                    // pairs could be (0, 0), (0, 1), (1, 0)
                    if (0==genes[i][k] && 0==genes[j][k]) {
                        vone[i][k]++;
                        vone[j][k]++;
                    }
                    else if (0==genes[i][k] && 1==genes[j][k]) {
                        vone[i][k]++;
                    }
                    else { //if (1==genes[i][k] && 0==genes[j][k])
                        vone[j][k]++;
                    }
                }
            }
        }
    }

    for (int i  = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (vone[i][j] > vzero[i][j]) genes[i][j] = 1;
            else if (vone[i][j] < vzero[i][j]) genes[i][j] = 0;
            else {
                if (0 != vone[i][j]) genes[i][j] = rand() % 2;
            }
        }
    }

    for (int i = 0; i < row; i++) {
        delete[] vone[i];
    }
    delete[] vone;

    for (int i = 0; i < row; i++) {
        delete[] vzero[i];
    }
    delete[] vzero;

    for (int i = 0; i < row; i++) {
        delete[] ap[i];
    }
    delete[] ap;
}
```

```
void Chromosome::delColum() {
    /*
    std::default_random_engine gen;
    std::uniform_real_distribution<double> rdis(0.0,1.0);
    double prob = rdis(gen);
    if (prob < 0.5) {
     */
    int ncol = rand() % col;
    std::vector<std::vector<int>> tmp;
    for (int i = 0; i < row; i++) {
        std::vector<int> r;
        for(int j = 0; j < col; j++) {
            if (j != ncol) {
                r.push_back(genes[i][j]);
            }
        }
        tmp.push_back(r);
    }
    genes = tmp;
}

void Chromosome::trimGene() {

    // transpose
    std::vector<std::vector<int>> t;
    for (int j = 0; j < col; j++) {
        std::vector<int> tmp;
        for (int i = 0; i < row; i++) {
            tmp.push_back(genes[i][j]);
        }
        t.push_back(tmp);
    }

    // remove duplicates
    std::vector<std::vector<int>>::iterator it;
    it = std::unique (t.begin(), t.end());
    t.resize(std::distance(t.begin(),it));

    // remove zero rows in transpose
    std::vector<int> zero(col, 0);
    it = std::find(t.begin(), t.end(), zero);
    if (it != t.end()) {
```

```
        t.erase(it);
    }

    row = t[0].size();
    col = t.size();
    std::vector<std::vector<int>> g;
    for (int j = 0; j < row; j++) {
        std::vector<int> tmp;
        for (int i = 0; i < col; i++) {
            tmp.push_back(t[i][j]);
        }
        g.push_back(tmp);
    }
    genes = g;
}

void Chromosome::updateMatrics(int **p) {
    // approximate policy matrix
    int** ap = new int* [row];
    for (int i = 0; i < row; i++) {
        ap[i] = new int[row];
    }

    nperm = 0;
    nforb = 0;
    for(int i = 0; i < row; i++) {
        for (int j = 0; j < row; j++) {
            for (int k = 0; k < col; k++) {
                ap[i][j] |= genes[i][k] & genes[j][k];
            }
            if (ap[i][j] == p[i][j]) {
                if (1 == ap[i][j]) {
                    nperm++;
                }
                else {
                    nforb++;
                }
            }
        }
    }
```

```
    for (int i = 0; i < row; i++) {
        delete[] ap[i];
    }
    delete[] ap;
}
```

```
#include <iostream>
#include <random>
#include <fstream>
#include <algorithm>
#include <cassert>
#include <chrono>
#include <array>
#include <omp.h>
#include "Chromosome.h"

using namespace std;

bool comp(Chromosome &a, Chromosome &b)
{
    return (a.getNperm()>b.getNperm() && a.getNforb()>b.getNforb() &&
   a.getCol()<b.getCol());
}

int main()
{

    const int ncomp = 20;
    // allocate space for policy matrix
    int **policy = new int *[ncomp];
    for (int i = 0; i < ncomp; i++)
    {
        policy[i] = new int[ncomp];
    }

    ifstream file;
    file.open("policy20x3.txt");
    if (!file.is_open())
    {
        cout << "Cannot open policy file" << endl;
        exit(-1);
    }
```

```
    else
    {
        for (int i = 0; i < ncomp; i++)
        {
            for (int j = 0; j < ncomp; j++)
            {
                file >> policy[i][j];
            }
        }
    }
    file.close();

    // diagonal must be 1
    for (int i = 0; i < ncomp; i++)
    {
        assert(1 == policy[i][i]);
    }

#ifdef PRINT
    for (int i = 0; i < ncomp; i++)
    {
        for (int j = 0; j < ncomp; j++)
        {
            printf("%2d␣", policy[i][j]);
        }
        printf("\n");
    }
#endif

    // generate population
    const int npopu = 400;
    // construct a trivial random generator engine from a time-based
   seed:
    unsigned seed = chrono::system_clock::now().time_since_epoch().
   count();
    default_random_engine gen(seed);
    double mu = ncomp / 2;
    double sigma = ncomp / 5;
    normal_distribution<> dis(mu, sigma);
    const int low = mu - 2 * sigma;
    const int high = mu + 2 * sigma;
```

```
    int *nvlans = new int[npopu];
    for (int i = 0; i < npopu; i++)
    {
        int nvlan = round(dis(gen));
        if (nvlan < low)
            nvlan = low;
        if (nvlan > high)
            nvlan = high;
        nvlans[i] = nvlan;
    }

    vector<Chromosome> population(npopu);
    for (int i = 0; i < npopu; i++)
    {
        population[i] = Chromosome(ncomp, nvlans[i]);
        population[i].updateMatrics(policy);
    }

// print the populations
#ifdef PRINT
    printf("print␣the␣populations\n");
    for (int i = 0; i < npopu; i++)
    {
        Chromosome temp = population[i];
        for (int r = 0; r < temp.getRow(); r++)
        {
            for (int c = 0; c < temp.getCol(); c++)
            {
                printf("%2d", temp.getGenes()[r][c]);
            }
            printf("\n");
        }
    }
#endif

    double start = omp_get_wtime();

    // number of generations or iterations
    array<Chromosome, 10> head;
    const int ngenn = 120;

    uniform_real_distribution<double> rdis(0.0, 1.0);
```

```
    for (int i = 0; i < ngenn; i++)
    {
        vector<Chromosome> paretoFrontier;
        vector<Chromosome> nonFrontier;
        while (!population.empty())
        {
            sort(population.begin(), population.end(), comp);
            Chromosome frontier = population.front();
            paretoFrontier.push_back(frontier);
            population.erase(population.begin());

            vector<Chromosome> rest;
            for (Chromosome chromosome : population)
            {
                if (!(frontier.getNperm() >= chromosome.getNperm() &&
                      frontier.getNforb() >= chromosome.getNforb() &&
                      frontier.getCol() <= chromosome.getCol()))
                {
                    rest.push_back(chromosome);
                }
                else
                {
                    nonFrontier.push_back(chromosome);
                }
            }
            population = rest;
        }

#ifdef PRINT
        int a = paretoFrontier.size();
        int c = nonFrontier.size();
        printf("%2d␣", a);
        printf("%2d\n", c);
#endif

#ifdef PRINT
    printf("Frontier\n");
    for (Chromosome chromosome : paretoFrontier)
    {
        printf("%3d%3d%3d\n", chromosome.getNperm(), chromosome.
   getNforb(), chromosome.getCol());
    }
```

```
    printf("nonFrontier\n");
    for (Chromosome chromosome : nonFrontier)
    {
        printf("%3d%3d%3d\n", chromosome.getNperm(), chromosome.
   getNforb(), chromosome.getCol());
    }
#endif

#ifdef LOG
        char *buffer = new char[30];
        sprintf(buffer, "./frontier/frontier%03d.txt", i);
        string path = buffer;
        ofstream outFile(path);
        for (Chromosome chromosome : paretoFrontier) {
            outFile << chromosome.getNperm() << "␣" << chromosome.
   getNforb() << "␣" << chromosome.getCol() << endl;
        }
        outFile.flush();
        outFile.close();
        delete[] buffer;
#endif

        vector<Chromosome> nextPopulation = paretoFrontier;
        while (nextPopulation.size() < npopu)
        {
            // start selection
            // if the size of pareto frontier is large enough, parents
    are chosen from the frontier
            // else if frontier is not emtpy, one from frontier, one
   from non-frontier
            // else, both from non frontier
            Chromosome chromosome1, chromosome2;
            double density = 0.40;
            if (paretoFrontier.size() > npopu * density)
            {
                chromosome1 = paretoFrontier.at(rand() %
   paretoFrontier.size());
                chromosome2 = paretoFrontier.at(rand() %
   paretoFrontier.size());
            }
            else if (paretoFrontier.size() > 0)
```

```
        {
            chromosome1 = paretoFrontier.at(rand() %
paretoFrontier.size());
            chromosome2 = nonFrontier.at(rand() % nonFrontier.size
());
        }
        else
        {
            chromosome1 = nonFrontier.at(rand() % nonFrontier.size
());
            chromosome2 = nonFrontier.at(rand() % nonFrontier.size
());
        }
        // end selection


        // start crossover
        // double prob = rdis(gen);
        vector<vector<int>> genes1 = chromosome1.getGenes();
        vector<vector<int>> genes2 = chromosome2.getGenes();
        int colRange = min(chromosome1.getCol(), chromosome2.
getCol());
        uniform_int_distribution<int> ndis(1, colRange);   // [1,
colRange]
        int ntimes = ndis(gen);
        for (int j = 0; j < ntimes; j++) {
            int nCol = ndis(gen) - 1;
            for (int k = 0; k < ncomp; k++) {
                swap(genes1[k][nCol], genes2[k][nCol]);
            }
        }
        chromosome1.setGenes(genes1);
        chromosome2.setGenes(genes2);
        // end crossover


        // start mutation
        double prob = rdis(gen);
        if (prob > 0.5) {
            chromosome1.coinFlip();
            chromosome2.coinFlip();
        }
```

```
            else if (prob > 0.3)
            {
                chromosome1.voteTune(policy);
                chromosome2.voteTune(policy);
            }
            // end mutation


            chromosome1.trimGene();
            chromosome2.trimGene();
            chromosome1.updateMatrics(policy);
            chromosome2.updateMatrics(policy);


            if (chromosome1.getNperm() > chromosome2.getNperm() &&
                chromosome1.getNforb() > chromosome2.getNforb() &&
                chromosome1.getCol() < chromosome2.getCol())
            {
                nextPopulation.push_back(chromosome1);
            }
            else if (chromosome2.getNperm() > chromosome1.getNperm()
&&
                     chromosome2.getNforb() > chromosome1.getNforb()
&&
                     chromosome2.getCol() < chromosome1.getCol())
            {
                nextPopulation.push_back(chromosome2);
            }
            else
            {
                nextPopulation.push_back(chromosome1);
                nextPopulation.push_back(chromosome2);
            }
        }

        population = nextPopulation;

    }
    double elapsed = omp_get_wtime() - start;
    printf("time: %f\n", elapsed);

    delete[] nvlans;
```

```
    for (int i = 0; i < ncomp; i++)
    {
        delete[] policy[i];
    }
    delete[] policy;

    return 0;
}
```

Curriculum Vitae

JINKU CUI

EDUCATION

Wake Forest University Aug 2018 - May 2020

Master of Science, Computer Science *Winston-Salem, NC*

Member of Upsilon Pi Epsilon

Nanjing University of Aero. and Astro. Sep 2013 - Jun 2018

Bachelor of Engineering, Computer Science and Technology *Nanjing, China*

Student Member of China Computer Federation

RESEARCH EXPERIENCE

Database Research Group Sep 2014 - Jun 2015

Software Engineer *Nanjing, China*

· Visiualized the searching process of a branch-and-bound skyline query algorithm on R-tree with C++ and Qt programming language

Cyber Security Lab Sep 2016 - Jun 2017

Research Assistant *Nanjing, China*

· Developed a scheme that reduced the storage cost, and improved the security and detection probability of password leakage

· Published work: Weiwei Jing, Jinku Cui, and Youwen Zhu. *A Honeyword Generation Method Based on Special Character Distance.* Hans Publisher, 2019.

WORK EXPERIENCE

China Unicom Jun 2016 - Sep 2016

Network Engineer Intern *Shangqiu, China*

- Planned optimal network routes based on available budge and network capacity
- Deployed and maintained network cables, routers, and switches
- Distributed network bandwidth dynamically with prediction algorithm

ADDITIONAL EXPERIENCE

Wake Forest University Aug 2018 - May 2020

Teacher Assistant *Wisnton-Salem, NC*

CSC 111 - Introduction to Computer Science

CSC 221 - Data Structures & Algorithms

TECHNICAL STRENGTHS

Programming Languages	C, C++, Java, C#, Python, JavaScript
Software & Tools	Microsoft Office, LaTeX, SAS, MATLAB

EXTRA-CIRRUCULAR

Outstanding volunteer in National Youth University Science Camp 2014

The second prize of the 10th National Software Competition 2015

Participated in software design competition for college students 2016

Outstanding Student Cadre of Student Association 2017

Outstanding Graduates 2018