Project Report for CS542

Fangzhou Xiong A20376382
Zhiquan Li A20381063
Bo Liang A20356451

Abstract

This report analysis the final traffic distribution caused by applying the strategy in the network, and the deviation of the simulated control variable. It introduces the algorithm and data structure used in simulation, and verify the simulation result.

Problem analysis

In a network defined by the project_f16_network_matrix(show below), we tried to find the TCV(traffic control variable) α, β and γ, which is the variable to choose the path to the destination for every packet, with the decision strategy that each source selects the transmission path with lowest cost from the source to the destination for every packet. The cost is defined by the utilization of XTY(u_{XTY}) which is:

$$u_{XTY} = u_{XT} + u_{TY}$$

Where u_{XT} is defined as:

$$u_{XT} = L_{total}/S_{XT}$$

Where S_{XT} is defined as the capacity of the link and L_{total} is the total load of the link, all the loads come from the intensity of three source-destination pair: A-C, B-D, C-A.

Distribution

The control variables we tried to find is utilized to allocate the traffic in each router that for every packet router routing to the path with lowest cost.
If we keep sending packets to the router, the router will always route the packets to lower path until eventually every path have almost the same cost, and the traffic flow for each path will be stable, which means the proportion for packets sending to each Path will be stable. And this proportion can use as traffic control variables.
Hence, by simulating the transmission in this network with the routing strategic, we can have the cumulative packets distribution in each router, and then we can calculate the control variables.

For a single router, if the number of all packet going through is SUM_{total}, the number of packets routing to each two path is N_{XTY} and N_{XNY}, then we have control variables for each path as, SUM_{XTY}/ SUM_{total} and SUM_{XNY}/ SUM_{total}.
Finally, we can get the Traffic control variable as:

$$\boldsymbol{\alpha} = SUM_{ABC}/ (SUM_{ABC} + SUM_{ABC})$$
$$\boldsymbol{\beta} = SUM_{BCD}/ (SUM_{BCD} + SUM_{BAD})$$
$$\boldsymbol{\gamma} = SUM_{CDA}/ (SUM_{CDA} + SUM_{CBA})$$

Deviation

Ideally, consider the traffic as flow, the traffic can evenly be divided to any degree, so we can find the optimized distribution. However, if we simulate the transmission with packets, we will have quantized difference between different path.
For a router with two routing path, simulating with 400 packets from source to destination **in a unit time (400 packets/unit time)**, the maximum deviation to the optimized answer is 1/40. Since every packets chose the path by calculating the cost, we cannot determine the accuracy beyond the deviation for the final distribution. However, the final packets distribution is directly determined by the total number of packets we sent, even though a single choice is not accuracy, with many attempts, we can have average result.
Therefore, if we want to control the accuracy of the result, first we should make sure the number of packets transmitting in a unit of time (N_{flow}) is lager then 1/error:

$$N_{flow} >= 1/error$$

For example, if the error is 0.1, we should transmit more than 10 packets in a unit of time.
And then we can send more packets to get a better result.

Intensity

Before we simulate the packets transmission, we need to clarify the intensity from the project. We know intensity is $\frac{aL}{R}$, which “a” is the average arrival rate of packets, “L” is the average packet length and ”R” is the transmission rate. If the intensity is greater than 1,it means the average arrival rate is $\frac{aL}{R}$ times greater than the transmission rate of the source and need queuing delay to deal with it. Since we do not need to consider the delay in the project, we could assume the intensity as the packet transmission rate, which is the number of packets need to be transited in a unit time.

Timeslot Queue

To simulating the packet transmission, we introduce a queue to simulate the traffic flow and control the deviation to the control variable result.
In the queue we called timeslot queue, each elements represent a timeslot, in each timeslot, there could be maximum 3 packets (A-C, B-D, C-A). Consider the flow of traffic can be described as number of packets in a unit time, the queue implies the intensity of the flow correlate with the number of packets of the queue.
To ensure the the intensity of source A, B and C is 4, 3 and 2, we consider the packets are evenly transmitted, we the packet of A, B and C should have different gap between two same neighbor packets.
For example, in the time queue below, the length of the queue is 12.

ABC			A	B		AC	B		A	B	
0	1	2	3	4	5	6	7	8	9	10	11

Graph 1 Timeslot queue

We enqueue the packet A-C in every 3 elements, packet B-D in every 4 elements, and packet C-A in every 6 elements. In this case, in a unit time, there is: 4 packet in A-C, 3 packets in B-D, and 2 packets in C-A respectively. This proportion is same as the given intensities.
Additionally, if the length of queue is increasing, and we keep the gap (every 3 packets for A-C, every 4 packets for B-D, every 6 packet for C-A), there will be more packets in the path. For example, assume the length of queue is 1200 means there 400 packet from A to C and 300 packets from B to D, and 200 packets from C to A.
So we can simulate flow with higher speed by longer the length of queue and keep the gap.

Simulation algorithm

To track the packets distribution in the network for every single unit time, we set Path Tractor (described as N_{XTY})to represent the number of packets in this path right now, for example, we can use: N_{ABC}, N_{ADC} to represent the number of packets in transit path {AB, BC} and {AD, DC} respectively.
To calculate the number of all the packets routing to transit paths, we set Counter SUM_{XTY} to represent the number of all the packets routing to this path.
Also, we label the packets with the transit path as P_{XY}. For example, the packet from source A to destination C is label as P_{AC} or A-C Packet. So, there are three types of packets: A-C, B-D, and C-A.

This is the algorithm used in simulation:

1. **Initialize the timeslot queue, enqueue a A-C Packet every 3 timeslot(element), a B-D Packet every 4 timeslot(element), and a C-A Packet every 6 timeslot(element) until the queue is full.**

2. **Keep do these steps until the number of sent packet is equal to what we set.**
    a) **Dequeue a timeslot element from the queue.**
    b) **Generate a new timeslot element.**
    c) **If the timeslot contains one or several packets, we call them arrived packets and do:**
        i. **Generate one new packets with same type corresponding to each packets from time slot.**
        ii. **For each new packet, calculate the cost of the two path, chose the path with smaller cost for the packet, and add 1 to the corresponding Path Tractor N_{XTY} and add 1 to the Counter SUM_{XTY}.**
        iii. **For each arrived packets, find their Path minus 1 to corresponding Path Tractor N_{XTY}.**
        iv. **Put the new packets into the new timeslot element.**
    d) **Enqueue the new timeslot element.**

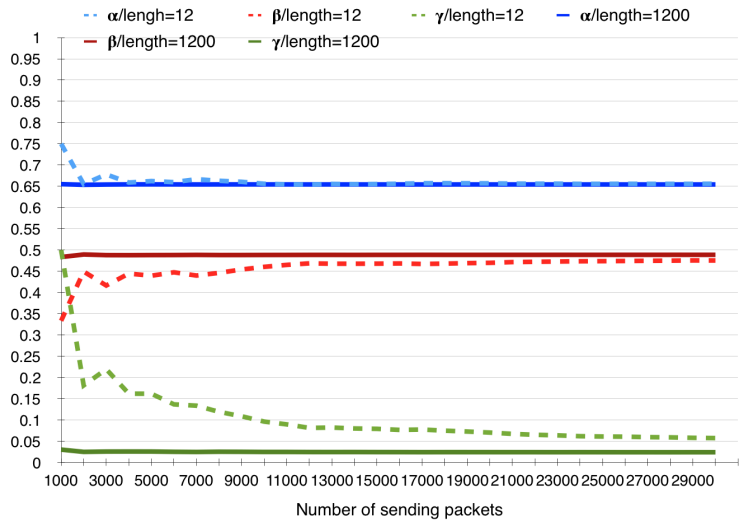
To calculate the cost of the path, we need to do these operation:

1. Calculate the traffic intensity distribution for each source-destination pair.
2. Calculate the utilization of each link with intensity of each traffic path and capacity of each link
3. Combine the utilization like to get cost of each path.

The algorithm we use determines the length of queue must can be multiple of 12, other wise the intensity of three sour-destination pair cannot be satisfied.

Result and analysis

In the simulation, we sent different number of packet when the length is 12 and 12000. The accuracy for when length is 12 is much lower then the other, so it is obviously optimized when the packets number is increasing. Since the **γ** is much lower then accuracy of this case, it is harder to get closer to the right answer then the case when the length is 1200.



Graph 2 control variable in different sending packets number and length of queue

Sending 30000 packets, we get several result with different accuracy below:

Length of the queue	Original α	Error(α)	α	Original β	Error(β)	β	γ	Error(γ)
600000	0.654183333	0.000005	0.64183	0.488848889	6.66667E-06	0.48884	0.023413333	0.00001
60000	0.65415	0.00005	0.6541	0.488866667	6.66667E-05	0.4888	0.02345	0.0001
6000	0.654107843	0.0005	0.654	0.488666667	0.000666667	0.488	0.024372549	0.001
600	0.657235529	0.005	0.65	0.481836327	0.006666667	0.48	0.039361277	0.01

Length of the queue	Original γ	Error(γ)	γ
600000	0.023413333	0.00001	0.02341
60000	0.02345	0.0001	0.0234
6000	0.024372549	0.001	0.023
600	0.039361277	0.01	0.02

Choosing the result of {**α = 0.6541, β = 0.4888, γ = 0.0234**}, we have answer for 2nd and 3rd question below:

Link	Traffic intensity of link	Utilization of link
AB	2.6166	0.8722
BA	3.4865	0.871625
BC	4.0832	0.81664
CB	1.9531	0.651033333
CD	1.5135	1.5135
DC	1.3834	0.230566667
DA	0.0469	0.00938
AD	2.9168	1.4584

Path	cost of the path
ABC	1.685289428
ADC	1.695270558
BCD	2.332355286
BAD	2.334565859
CBA	1.517557673
CDA	1.529748047

Additional Verify

As we discuss above, in ideal situation, we can divide the intensity to any degree, since the strategy eventually lead to a traffic distribution that the cost of two path for a source should all be same, we can calculate the control variable from these equations:

$Cost_{ABC}=4*\alpha/S_{AB}+(4*\alpha+3*\beta)/S_{BC}$

$Cost_{ADC}=(4*(1-\alpha)+3*(1-\beta))/S_{AD}+4*(1-\alpha)/S_{DC}$

$Cost_{BCD}=(4*\alpha+3*\beta)/S_{BC}+(2*\gamma+3*\beta)/S_{CD}$

$Cost_{BAD}=(3*(1-\beta)+2*(1-\gamma))/S_{BA}+(4*(1-\alpha)+3*(1-\beta))/S_{AD}$

$Cost_{CBA}=2*(1-\gamma)/S_{CB}+(2*(1-\gamma)+3*(1-\beta))/S_{BA}$

$Cost_{CDA}=(2*\gamma+3*\beta)/S_{CD}+2*\gamma/S_{DA}$

Since the all the cost of two path for a source is equal

$Cost_{ABC} = Cost_{ADC}$

$Cost_{BCD} = Cost_{BAD}$

$Cost_{CBA} = Cost_{CDA}$

So we have:

$\alpha=0.654183$

$\beta=0.488849$

$\gamma=0.023405$

Comparing to the the simulation result which has 4 decimal places, we can verify that the simulation result is reasonable.

Conclusion

This report analysis the eventual traffic distribution in the network by applying the strategy, the deviation in simulating. And then it introduction logarithm and data structure using in the simulation, analysis and verify the final result.
To have 3 decimal palaces for result, we have: **$\alpha = 0.6541$, $\beta = 0.4888$, $\gamma = 0.0234$.**
Using this control variables, we can have traffic load for each link:
l_{AB}= 2.616733333
l_{BA}= 3.486626667
l_{BC}= 4.08328
l_{CB}= 1.953173333
l_{DC}= 1.383266667
l_{CD}= 1.513373333
l_{DA}= 0.046826667
l_{AD}= 2.91672

Traffic utilization for each link is:
U_{AB}= 0.872244444
U_{BA}= 0.871656667
U_{BC}= 0.816656
U_{CB}= 0.651057778
U_{DC}= 0.230544444
U_{CD}= 1.513373333
U_{DA}= 0.009365333
U_{AD}= 1.45836

Cost of 6 considered traffic path is:
U_{ABC}=1.688900444
U_{ADC}=1.688904444
U_{BCD}=2.330029333
U_{BAD}=2.330016667
U_{CBA}=1.522714444
U_{CDA}=1.522738667

Source code:

Simulator

```
package edu.iit.cs542;

public class Simulator {

protected static CS542Queue<TimeSlot> timeStream = new
CS542Queue<TimeSlot>(12);

    protected static NetworkGraph network = new NetworkGraph();

    protected static long numberOfTimeSlot = 30000;
    protected static int lengthOfqueue = 12000;

    protected static double cost = 1000000;
    protected static double alfa = 1;
    protected static double beta = 1;
    protected static double gamma = 1;

    /**
     * This Simulator following these steps:
     * 1. Dequeue a time slot from each
     * three queues
     * 2. If the time slot contains a packets, get info for new
     * packet
     * 3. Generate time slot and fill with the info of packet
     * 5, enqueue the packet
     */
    public static void main(String[] args) {
        initQueue();

        for (int i = 0; i < numberOfTimeSlot; i++) {
            TimeSlot newSlot = new TimeSlot(null, null, null);
            TimeSlot passSlot = timeStream.remove();
            // dnqueue a timeslot

            for (int j = 0; j < 3; j++) {// check all the packets in
```

```
                                        // timeslot
                if (passSlot.packets[j] != null) {
                    // If a packet arrive in destination, a new packet
                    // send
                    // out from source
                    newSlot.packets[j] =
sendNewPacket(passSlot.packets[j].path);
                    // reduce one from the number of corresponding path
                    // packets
                    int pi =
Path_Type.get_path_index(passSlot.packets[j].path);
                    network.distribution[pi]--;
                }
            }
            timeStream.add(newSlot);
        }

        alfa = (double) network.counter[0] / (network.counter[0] +
network.counter[1]);
        beta= ((double) network.counter[2] / (network.counter[2] +
network.counter[3]));
        gamma = (double) network.counter[4] / (network.counter[4] +
network.counter[5]);
        System.out.println("For " + numberOfTimeSlot + " time slots we
have:\n\nafa is:" + alfa + "\ngamma is:" + gamma
                + "\nbeta is:" + beta+"\n\n");

        System.out.println(Strategic.getNetworkInfor(network, lengthOfqueue,
alfa, beta , gamma));

    }

    /**
     * Initial the time stream queue
     */
    private static void initQueue() {
        for (int i = 0; i < lengthOfqueue; i++) {
            TimeSlot timeslot = new TimeSlot(null, null, null);

            if (i % 3 == 0) {// send an packet from A to C
                timeslot.packets[0] = sendNewPacket(Path_Type.ABC);
            }
            if (i % 4 == 0) {// send an packet from B to D
                timeslot.packets[1] = sendNewPacket(Path_Type.BCD);
            }
            if (i % 6 == 0) {// send an packet from C to A
                timeslot.packets[2] = sendNewPacket(Path_Type.CDA);
            }
            timeStream.add(timeslot);
        }
    }

    /**
     * Sent a new packet from the same source of arrived packet using the
     * strategic of chosing the smaller utilization
     *
     * @param type
     *          -the path of arrived packet
```

```
     * @return -the sent packet
     */
    protected static Packet sendNewPacket(Path_Type type) {
        Packet packet = new Packet();
        Strategic startegic = new Strategic();
        // packet.type = type;
        // packet_index of the arrived packet
        int packet_index = Path_Type.get_path_index(type);
        // pccket_index of the sent packet
        packet_index = (packet_index / 2) * 2 +
startegic.choseThePath(network, packet_index);

        packet.path = Path_Type.get_path_name(packet_index);
        network.counter[packet_index]++;
        network.distribution[packet_index]++;

        return packet;
    }

}
```

Strategic

```
package edu.iit.cs542;

public class Strategic {
    public Strategic(){}

    /**
     * Chose the path whose utilization is lower than the others
     * @param network -the network graph
     * @param pindex -the start vertex
     * @return return 0 for chose the clockwise direction, and 1 for
anticlockwise direction
     */
    public int choseThePath(NetworkGraph network, int pindex){
        return Utilization(pindex,network);
    }

    public int Utilization(int pi, NetworkGraph network) {
        double alfa = (double) network.distribution[0] /
(network.distribution[0] + network.distribution[1]) ;
        double beta = ((double) network.distribution[2] /
(network.distribution[2] + network.distribution[3]) );
        double gamma = (double) network.distribution[4] /
(network.distribution[4] + network.distribution[5]) ;

        double  abc = 4*alfa,
                adc = 4-abc,
                bcd = 3*beta,
                bad = 3-bcd,
                cda = 2*gamma,
```

```
                cba = 2-cda;

        double  Uab = abc / 3,
                Ubc = (abc + bcd) / 5,
                Uad = (adc + bad) / 2,
                Udc = adc / 6,
                Uba = (bad + cba) / 4,
                Ucb = cba / 3,
                Uda = cda / 5,
                Ucd = bcd + cda,
                Uabc = Uab + Ubc,
                Uadc = Uad + Udc,
                Ubcd = Ubc + Ucd,
                Ubad = Uba + Uad,
                Ucba = Ucb + Uba,
                Ucda = Ucd + Uda;

        if ((pi == 0 || pi == 1) &&Uabc >= Uadc) {
            return 1;
        } else if ((pi == 2 || pi == 3) && Ubcd >= Ubad) {
            return 1;
        } else if ((pi == 4 || pi == 5) && Ucda >= Ucba) {
            return 1;
        } else {
            return 0;
        }
    }
    public String toString(NetworkGraph network) {
        double alfa = (double) network.distribution[0] /
(network.distribution[0] + network.distribution[1]) ;
        double beta = ((double) network.distribution[2] /
(network.distribution[2] + network.distribution[3]) );
        double gamma = (double) network.distribution[4] /
(network.distribution[4] + network.distribution[5]) ;

        double  abc = 4*alfa,
                adc = 4-abc,
                bcd = 3*beta,
                bad = 3-bcd,
                cda = 2*gamma,
                cba = 2-cda;

        double Uab = abc / 3, Ubc = (abc + bcd) / 5, Uad = (adc + bad) / 2,
Udc = adc / 6, Uba = (bad + cba) / 4,
                Ucb = cba / 3, Uda = cda / 5, Ucd = bcd + cda, Uabc = Uab +
Ubc, Uadc = Uad + Udc, Ubcd = Ubc + Ucd,
                Ubad = Uba + Uad, Ucba = Ucb + Uba, Ucda = Ucd + Uda;

        return "{"+alfa+", "+gamma+", "+beta+"}";
    }

    public static String getNetworkInfor(NetworkGraph network, int
lengthOfqueue, double alfa, double beta, double gamma){
        double  abc = 4*alfa,
                adc = 4-abc,
                bcd = 3*beta,
                bad = 3-bcd,
                cda = 2*gamma,
```

```
                cba = 2-cda;


        double  Iab = abc,
                Ibc = (abc + bcd) ,
                Iad = (adc + bad),
                Idc = adc ,
                Iba = (bad + cba) ,
                Icb = cba ,
                Ida = cda ,
                Icd = bcd + cda,
                Uab = abc / 3,
                Ubc = (abc + bcd) / 5,
                Uad = (adc + bad) / 2,
                Udc = adc / 6,
                Uba = (bad + cba) / 4,
                Ucb = cba / 3,
                Uda = cda / 5,
                Ucd = bcd + cda,
                Uabc = Uab + Ubc,
                Uadc = Uad + Udc,
                Ubcd = Ubc + Ucd,
                Ubad = Uba + Uad,
                Ucba = Ucb + Uba,
                Ucda = Ucd + Uda;

        return "This is the linke info:\n"
                + "{Iab:"+Iab+", Uab="+Uab+"}\n"+
        "{Iba:"+Iba+", Uba="+Uba+"}\n"+
        "{Ibc:"+Ibc+", Ubc="+Ubc+"}\n"+
        "{Icb:"+Icb+", Ucb="+Ucb+"}\n"+
        "{Icd:"+Icd+", Ucd="+Ucd+"}\n"+
        "{Idc:"+Idc+", Udc="+Udc+"}\n"+
        "{Ida:"+Ida+", Uda="+Uda+"}\n"+
        "{Iad:"+Iad+", Uad="+Uad+"}\n"+
        "\n\nThis is all the transit paths info:\n"+
        "{Uabc:"+Uabc+"}\n"+"{Uadc:"+Uadc+"}\n"+
        "{Ubcd:"+Ubcd+"}\n"+"{Ubad:"+Ubad+"}\n"+
        "{Ucba:"+Ucba+"}\n"+"{Ucda:"+Ucda+"}\n"+
        "\nThe total cost:"+(Uabc+Uadc+Ubcd+Ubad+Ucba+Ucda);
    }

}
```

NetworkGraph

```
package edu.iit.cs542;

public class NetworkGraph {
    protected int[] distribution = { 0, 0, 0, 0, 0, 0, };// Each
elements
```

```
                                                    // represent the
                                                    // number of
                                                    // packet in
                                                    // ABC, ADC,
                                                    // BCD, BAD,
                                                    // CBA, CDA
    protected int[] counter = { 0, 0, 0, 0, 0, 0, };// Each elements
represents
                                                  // the counter of
all
                                                  // arriving packets
in
                                                  // ABC, ADC, BCD,
BAD,
                                                  // CBA, CDA

    public NetworkGraph() {

    }

    public String toString() {
        return "{ AC : ABC[" + distribution[0] + "]," + "ADC:[" +
distribution[1] + "]}" + ";{ BD :BCD["
                + distribution[2] + "]" + ",BAD[" + distribution[3] +
"]}" + ";{ CA :CDA[" + distribution[4] + "]"
                + ",CBA[" + distribution[5] + "]}";

    }
}
```

TimeSlot

```
package edu.iit.cs542;

public class TimeSlot {
    protected Packet[] packets = {null,null,null};// packets array,
packets[0] is packet in AC,
                            // packets[1] is packet in BD,
                            // packets[2] is packet in CA

    public TimeSlot(Packet pac, Packet pbd, Packet pca) {
        packets[0] = pac;
        packets[1] = pbd;
        packets[2] = pca;

    }

    public String toString() {
```

```
        return "[packetAC:" + (packets[0] == null ? "null" :
packets[0].toString()) + "]; " + "[packetBD:"
                + (packets[1] == null ? "null" : packets[1].toString())
+ "]; " + "[packetCA:"
                + (packets[2] == null ? "null" : packets[2].toString()
+ "]; ");
    }
}
```

```
package edu.iit.cs542;

public class Packet {
//  protected Packet_Type type;
    protected Path_Type path;

    public String toString(){
        return "path:"+ path+" ";//"Type:"+type+", "+
    }
}
```

```
package edu.iit.cs542;

public class TimeSlot {
    protected Packet[] packets = {null,null,null};// packets array,
packets[0] is packet in AC,
                            // packets[1] is packet in BD,
                            // packets[2] is packet in CA

    public TimeSlot(Packet pac, Packet pbd, Packet pca) {
        packets[0] = pac;
        packets[1] = pbd;
        packets[2] = pca;

    }

    public String toString() {
        return "[packetAC:" + (packets[0] == null ? "null" :
packets[0].toString()) + "]; " + "[packetBD:"
                + (packets[1] == null ? "null" : packets[1].toString())
+ "]; " + "[packetCA:"
                + (packets[2] == null ? "null" : packets[2].toString()
+ "); ");
    }
}
```

Packet

```
package edu.iit.cs542;

public class Packet {
//  protected Packet_Type type;
    protected Path_Type path;
```

```
    public String toString(){
        return "path:"+ path+" ";//"Type:"+type+", "+
    }
}
```

Packet_Type

```
package edu.iit.cs542;

public enum Packet_Type {AC, BD, CA}
```

Path_Type

```
package edu.iit.cs542;

public enum Path_Type {
    ABC, ADC, BAD, BCD, CBA, CDA;
    public static int get_path_index(Path_Type name) {
        int v = -1;

        switch (name) {
        case ABC:
            v = 0;
            break;
        case ADC:
            v = 1;
            break;
        case BCD:
            v = 2;
            break;
        case BAD:
            v = 3;
            break;
        case CDA:
            v = 4;
            break;
        case CBA:
            v = 5;
            break;
        default:
            System.out.println("get_path_index: invalid name");
            break;
        }
        return v;
    }

    public static Path_Type get_path_name(int index) {
        Path_Type v = null;
        switch (index) {
```

```
        case 0:
            v = ABC;
            break;
        case 1:
            v = ADC;
            break;
        case 2:
            v = BCD;
            break;
        case 3:
            v = BAD;
            break;
        case 4:
            v = CDA;
            break;
        case 5:
            v = CBA;
            break;
        default:
            System.out.println("get_path_name: invalid index");
            break;
        }
        return v;
    }
}
```

CS542Queue

```
package edu.iit.cs542;

/*
 * Fixed front implementation of a Queue using arrays */

public class CS542Queue<E>  {
    private E[] data;
    private int front, back;
    private int capacity;

    /**
     * Constructor
     *
     * @param num_elems
     */
    public CS542Queue(int num_elems) {
        capacity = num_elems;
        data = (E[]) new Object[capacity];
        front = back = 0;
    }
```

```
    /**
     * Add the new element into the end of the queue Enqueue
operation
     *
     * @param element
     */
    public void add(E element) {
        if (is_full()) {
            capacity *= 2;
            E[] newData = (E[]) new Object[capacity];
            for (int i = 0; i < capacity / 2; i++) {
                newData[i] = data[i];
            }
            data = newData;
            data[back] = element;
        } else {
            data[back] = element;
        }
        back++;
    }

    /**
     * Remove the frist element in the front of the queue Dequeue
operation
     *
     * @return
     */
    public E remove() {
        if (is_empty()) {
            return null;
        } else {
            E oe = data[0];
            for (int i = 1; i < back; i++) {
                data[i - 1] = data[i];
            }
            back--;
            return oe;
        }

    }

    /**
     * Return the first element value in the font of the queue
without remove
     * it.
     *
     * @return
     */
    public E peek() {
        if (is_empty()) {
            return null;
```

```
    } else {
        return data[0];
    }
}

/**
 * True if the queue is empty
 *
 * @return
 */
public boolean is_empty() {
    if (back == 0)
        return true;
    else
        return false;
}

/**
 * True if the queue is full
 *
 * @return
 */
public boolean is_full() {
    if (back == capacity) {
        return true;
    } else {
        return false;
    }
}

/**
 * Return the capacity of the array inside. [For test use]
 *
 * @return
 */
protected int getCapacity() {
    return capacity;
}

/**
 * Return the front pointer of the array inside. [For test use]
 *
 * @return
 */
protected int getFront() {
    return front;
}

/**
 * Return the back pointer of the array inside. [For test use]
 *
 * @return
```

```
     */
    protected int getBack() {
        return back;
    }

}
```