Assignment Two

Abanob Tawfik
z5075490

March 2019

1 Problem 1

Use Natural Deduction to show the following: (10 marks)

$$\vdash P(a) \rightarrow \forall x(P(x) \vee \neg(x = a))$$

Solution:

1. $\neg(P(a) \rightarrow \forall x(P(x) \vee \neg(x = a))$	
2. P(a)	
3. $\neg(P(b) \vee \neg(b = a))$	
4. $\neg P(b)$	
5. b = a	
6. $\neg P(a)$	$=$**Elim:** 4, 5
7. $\bot$	$\bot$**Intro:** 2, 6
8. $\neg(b = a)$	$\neg$**Intro:** 5–7
9. P(b) $\vee \neg(b = a)$	$\vee$ Intro-2: 8
10. $\bot$	$\bot$**Intro:** 3, 9
11. $\neg\neg P(b)$	$\neg$**Intro:** 4–10
12. P(b)	Double Negation Elim: 11
13. P(b) $\vee \neg(b = a)$	$\vee$ Intro-1: 12
14. $\bot$	$\bot$**Intro:** 3, 13
15. $\neg\neg(P(b) \vee \neg(b = a))$	$\neg$**Intro:** 3–14
16. (P(b) $\vee \neg(b = a))$	Double Negation Elim: 15
17. $\forall x(P(x) \vee \neg(x = a))$	$\forall$**Intro:** 16
18. P(a) $\rightarrow \forall x(P(x) \vee \neg(x = a))$	$\rightarrow$**Intro:** 2–17
19. $\bot$	$\bot$**Intro:** 1, 18
20. P(a) $\rightarrow \forall x(P(x) \vee \neg(x = a))$	IP: 1-19

Since i had used a derived rule "Double Negation Elim" i will also provide a proof for this rule
on the next page.

Proof for "Double Negation Elim"

$$\neg\neg\psi(a) \vdash \psi(a))$$

Solution:

1. $\neg\neg\psi(a)$

   2. $\neg\psi(a)$

   3. $\bot$ $\bot$ **Intro:** 1, 2

4. $\psi(\mathsf{a})$ IP: 1-3

Explanation of the proof:

This proof has no premises, so in order to perform a proof of such we need an indirect proof which can also be seen as proof by contradiction. First we assume the opposite of what we are trying to prove, in this case the negation of our consequence:
$$\neg(P(a) \to \forall x(P(x) \vee \neg(x = a)))$$

Next we want to try to arrive to the contradiction somewhere down the proof. We use modus ponen by assuming $P(a)$ and trying to reach the conclusion in the subproof $\forall x(P(x) \vee \neg(x = a))$ allowing us to introduce implication and arrive to a direct contradiction:
$$P(a) \to \forall x(P(x) \vee \neg(x = a))$$

We want to make the assumption of the negation of the conclusion we are trying to reach using variable b, as when b is no longer a free variable in x, in other words we are no longer working under the assumption $P(b)$ and b is arbitrary we can perform a $\forall$ introduction.

Since the consequence $\forall x(P(x) \vee \neg(x = a))$ contains $\vee$, we only need one side of the predicate to perform a $\vee$ introduction. To do this properly and allow use of the $\forall$ introduction we need to make b an arbitrary variable. We make the assumption $\neg P(b)$ and under that we make another assumption b = a. By doing this we can perform an equals elimination (substitution) to arrive at $\neg P(a)$ which is a direct contradiction to our first assumption $P(a)$. This gives us $\bot$ which implies $\neg(b = a)$ and this can also be done to imply $P(b)$ similarly.

By having the above under the assumption $\neg(P(b) \vee \neg(b = a))$ we can perform a $\vee$ introduction on $P(b)$ above to introduce $\neg(b = a)$ arriving at P(b) $\vee \neg(b = a)$ which is again another contradiction. Since this subproof concludes to $\bot$ we can assume the negation of our initial assumption for the subproof which is $P(b) \vee \neg(b = a)$. Because b is an arbitrary variable, $\neg P(b)$ was discharged, we can simply perform a $\forall$ introduction and arrive to the conclusion:
$$P(a) \to \forall x(P(x) \vee \neg(x = a))$$

This is a direct contradiction to our initial assumption $\neg(P(a) \to \forall x(P(x) \vee \neg(x = a)))$, giving us $\neg\neg(P(a) \to \forall x(P(x) \vee \neg(x = a)))$. Performing one more double negation elimination (proven in the second natural deduction proof) gives us the following conclusion by indirect proof:

$$\vdash P(a) \to \forall x(P(x) \vee \neg(x = a))$$

The following images below were taken from the following natural deduction verification tool,
https://proofs.openlogicproject.org/, which was used to validate the following natural deduction
that was performed in the problem and the proof of the derived rule for double negation
elimination. Note that rule DNE was used on the image, this was a derived rule that was proven
and verified in the next image.

Construct a proof for the argument: $\therefore Pa \rightarrow \forall x(Px \vee \neg x = a)$

1	$\neg(Pa \rightarrow \forall x(Px \vee \neg(x = a)))$	
2	Pa	
3	$\neg(Pb \vee \neg(b = a))$	
4	$\neg Pb$	
5	$b = a$	
6	$\neg Pa$	=E 4, 5
7	$\bot$	$\neg$E 2, 6
8	$\neg(b = a)$	$\neg$I 5–7
9	$Pb \vee \neg(b = a)$	$\vee$I 8
10	$\bot$	$\neg$E 3, 9
11	$\neg\neg Pb$	$\neg$I 4–10
12	Pb	DNE 11
13	$Pb \vee \neg(b = a)$	$\vee$I 12
14	$\bot$	$\neg$E 3, 13
15	$\neg\neg(Pb \vee \neg(b = a))$	$\neg$I 3–14
16	$(Pb \vee \neg(b = a))$	DNE 15
17	$\forall x(Px \vee \neg(x = a))$	$\forall$I 16
18	$Pa \rightarrow \forall x(Px \vee \neg(x = a))$	$\rightarrow$I 2–17
19	$\bot$	$\neg$E 18, 1
20	$Pa \rightarrow \forall x(Px \vee \neg(x = a))$	IP 1–19

☐ NEW LINE   ☐ NEW SUBPROOF

☺ Congratulations! This proof is correct.

Figure 1: Verification tool returning that my proof is correct for main proof

Below is the image showing the proof for the derived rule DNE which states that $\neg\neg P = P$.

Proof:

Construct a proof for the argument: $\neg\neg Pa \therefore Pa$

1	$\neg\neg Pa$	
2	$\quad \neg Pa$	
3	$\quad \bot$	$\neg$E 1, 2
4	Pa	IP 2–3

[+ NEW LINE]　　　[+ NEW SUBPROOF]

☺ Congratulations! This proof is correct.

[CHECK PROOF]　　[START OVER]

Figure 2: Verification tool returning that my proof of the derived rule DNE is correct

2 Problem 2

Please see the appendix for the full java code that can be copied and pasted and run to display the model. the code that was used inside this question is a bit too split up and cannot be easily copied and pasted however do give a good view of what i am trying to say.

Recall from Assignment 1 the problem of assigning channels to wifi networks to avoid interference. The task in this assignment is to set up a logical framework that can handle the same problem in a more general setting: i.e. with more than two channels and a wider variety of proximity topologies. As before we have some standard requirements:

  I A network uses one, and only one channel.

 II Networks within close proximity cannot both use the same channel.

III "Close proximity" is an anti-reflexive, symmetric relation.

(a) Carefully define a specification language (that is, a vocabulary or set of propositional variables) necessary to formally specify the requirements laid out above. You may:

   - consider the problem in propositional logic or predicate logic;
   - fix the number of channels and/or networks (e.g. assume there are three channels and four networks).

   However, you will be assessed on how extensible your definitions are – in other words, how easy it is to extend the specification language to cover more general settings (e.g. more channels and/or more networks). (4 marks)

   Solution:
   For this problem we will be defining a specification language in the form of a vocabulary to satisfy the above requirements. To define a vocabulary we need to first define our domain of discourse and our set of:

   - Predicate symbols
   - Function symbols
   - Constant symbols

   Our two objects inside our domain of discourse are channels and networks which can be viewed as the following:
   Channel:

   - Integer id (represents the channel state e.g, 0 = low, 1 = medium, 2 = high)

   Network:

   - String name
   - Channel connected
   - List<String> close_proximity_networks (String was used since it can be the identifier of Network assuming unique naming)

5

Using these definitions we can construct the following predicates for our vocabulary:

Network(name, channel,close_proximity_networks) which will be used to return true if the tuple values match the entry for the network.

Channel(id) which will be our unary predicate that returns true if the channel with the corresponding id exists, otherwise false.

is_close_proximity(network1, network2) which resolves to true if the two networks are in close proximity (the networks are in each others close proximity list) or false if otherwise.

valid_channel_configuration(network1, network2) which will resolves to true if the networks have no interference (network 1's channel is different to network 2's channel).

is_connected(network, channel) which will resolve to true if the network is connected to that channel (we can have the channel connected to be -1 before assignment to indicate it is yet to be assigned to a channel.

Some other unary predicates to restrict our domain of discourse would include:
is_integer(value) which will be used to make sure valid entries for channel ids.
is_String(value) which will be used to make sure valid entries for Network names.

For this problem it is mostly relational and predicate based so there are no function symbols and constant symbols and these can be considered an empty set. Our vocabulary V can be seen as:
V = {Network, Channel, is_close_proximity, valid_channel_configuration, is_connected, is_integer, is_String}

With this vocabulary we are able to setup a solution to any configuration (if there is one) with n networks and m channels by making sure each network is connected, and making sure that each network in close proximity are on different channels.

(b) In your specification language, and subject to the assumptions you have made, define the requirements with logical formulas:

Please note for this question i will use the variable X to represent all networks and x to represent individual networks. similarly i will be using the variable Y to represent all channels and y to represent individual channels. A quick explanation of the predicate formulas will also be provided in code and in propositional logic aswell.!

  (i) A network uses one, and only one channel. (2 marks)

    Solution:
    $\psi_1 = \exists x_1 \in X, \exists y_1 \in Y (P(x_1, y_1) \wedge \forall y_2 \in Y (is\_connected(x_1, y_2)) \rightarrow y_2 = y_1)$

    Essentially what we are saying is that a network is connected to a channel, and if any other channel satisfies that connection, than the channels are equivalent.
    The following predicate can be seen in propositional logic as the following:

Assuming there are n channels and A_i represents the proposition that network A is connected to channel i, then we can construct the proposition that a network uses one and only channel in the following:

$\sigma = A_1 \vee A_2 \vee A_3 \vee A_4 \vee ... \vee A_n$ (network is connected to any channel)

$\phi_1 = \neg(A_1 \wedge A_2) \wedge \neg(A_1 \wedge A_3) \wedge ... \wedge \neg(A_1 \wedge A_n)$ (no multiple connections on channel 1 but this needs to be done for all channels) $\phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge ... \wedge \phi_{n-1}$

so our proposition is $\sigma \wedge \phi$

Essentially ϕ says that for all possible combinations of connections with collisions we make sure that none of these are true, for example if there were three channels we would say

$\phi_1 = \neg(A_1 \wedge A_2) \wedge \neg(A_1 \wedge A_3)$

$\phi_2 = \neg(A_2 \wedge A_3)$ (note $A_2 \wedge A_1$ is captured in ϕ_1)

$\phi = \phi_1 \wedge \phi_2$

  $= \neg(A_1 \wedge A_2) \wedge \neg(A_1 \wedge A_3) \wedge \neg(A_2 \wedge A_3)$

σ says that we are connected to any network, so we must be connected to one network, for example if there were three channels we would say

$\sigma = A_1 \vee A_2 \vee A_3$

Putting these together we have the predicate function for 3 channels (note this can be extended to any amount of channels

$is\_connected(network, channel) = (channel_1 \vee channel_2 \vee channel_3) \wedge \neg(channel_1 \wedge channel_2) \wedge \neg(channel_1 \wedge channel_3) \wedge \neg(channel_2 \wedge channel_3)$

(ii) Networks within close proximity cannot both use the same channel.   (2 marks)

Solution:

$\psi_2 = \forall x_1 \in X (\forall x_2 \in X (is\_close\_proximity(x1, x2) \rightarrow valid\_configuration(x1, x2)))$

We are saying that we will check each network with every other network and for all combinations (nested for loop style) if two networks are in close proximity that implies they are in a valid configuration (using different channels). By using the same amount of channels as the question above and this time using channel A and channel B in close proximity we can set up the propositional formula, that if the channels A and B are in close proximity they are on different channels in the following way:

$\phi_1 = is\_close\_proximity(A, B)$

let ϕ_2 be the propositional statement that close proximity networks don't use the same channel. For n amount of networks this can be represented as:

$\phi_2 = \neg(A_1 \wedge B_1) \wedge \neg(A_2 \wedge B_2) \wedge \neg(A_3 \wedge B_3) \wedge ... \wedge \neg(A_n \wedge b_n)$

so our requirement ψ_2 becomes the following:

$\psi_2 = \phi_1 \rightarrow \phi_2$ or in expanded form

$\psi_2 = (is\_close\_proximity(A, B)) \rightarrow (\neg(A_1 \wedge B_1) \wedge \neg(A_2 \wedge B_2) \wedge \neg(A_3 \wedge B_3) \wedge ... \wedge \neg(A_n \wedge b_n))$

Note is_close_proximity(A, B) is simply a predicate which will check if the two networks are close. This predicate formula can also be displayed in code in the following way.

Class for channel

```java
public class Channel{

    public int id;

    public Channel(int ID){
        this.id = ID;
    }

    public int get_Id(){
        return this.id;
    }
}
```

class for Network

```java
import java.util.*;

public class Network{

    public String name;
    public Channel connected;
    public ArrayList<String> close_proximity_networks;

    public Network(String name, Channel connected, ArrayList<String>
         close_proximity_networks){
        this.name = name;
        this.connected = connected;
        this.close_proximity_networks = close_proximity_networks;
    }

    public String get_name(){
        return this.name;
    }

    public Channel get_channel(){
        return this.connected;
    }

    public ArrayList<String> get_close_proximity_networks(){
        return this.close_proximity_networks;
    }
}
```

General solution for this problem and the predicates in code!

```java
import java.util.*;

public class Predicates{

    public ArrayList<Network> all_networks;
    public ArrayList<Channel> all_channels;

    public boolean is_close_proximity(Network n1, Network n2){
        //if the network name is apart of the other networks close proximity
            network list then return true
        if(n1.get_close_proximity_networks().contains(n2.get_name())){
            return true;
        }
        //return false otherwise
        return false;
    }

    public boolean valid_configuration(Network n1, Network n2){
        //if the two networks are in close proximity then we check the channels
            are different to return true
        if(is_close_proximity(n1, n2){
            //both networks use different channels
            return n1.get_channel().get_id() != n2.get_channel().get_id();
        }
        //if the channels are not in close proximity then it is valid eitherway
        return true;
    }

    //Now we can construct our final predicate
    public boolean all_configurations_valid(){
        //we want to check all possible configurations
        for(Network n1 : all_networks){
            for(Network n2 : all_networks){
                //anti-reflexive property as this will lead to invalid configuration
                if(n1 == n2){
                    continue;
                }
                //if ANY configuration is invalid we return false as the setup is
                    invalid
                if(!valid_configuration(n1, n2){
                    return false;
                }
            }
        }
        //return true if it does not exit the loop with return false
        return true;
    }
}
```

Note the following predicates for the vocabulary have been defined in the java code above.

(iii) "Close proximity" is an anti-reflexive, symmetric relation. (2 marks)

Solution:
$\psi_3 = \forall x1 \in X(\neg is\_close\_proximity(x1, x1))$
$\psi_4 = \forall x1 \in X(\forall x2 \in X(is\_close\_proximity(x1, x2) \rightarrow is\_close\_proximity(x2, x1))$

ψ_3 is the logical formula which states for all networks inside our network domain, they cannot be in close proximity with themselves which is an anti-reflexive relation.

ψ_4 is the logical formula which states for all networks if network A is in close proximity to network B, then that also implies network B is in close proximity with network A which a symmetric relation.

"Close proximity" must be an anti-reflexive and symmetric relation in order for valid configurations. This is because if a Network can be in close proximity with itself that means the valid_configuration would return false since it will show up as both networks using the same channel even though its the same network. "Close proximity" must also be a symmetric relation, since if A is in close proximity with B, that means B must be in close proximity with A. The above formulas slightly modify the code we have written above the following
re-defining the code to allow close_proximity to be an anti-reflexive and symmetric.

```java
public boolean is_close_proximity(Network n1, Network n2){
    //anti-reflexive property by not letting n1 = n2 be true
    //i.e. if the two networks are the same return false!
    if(n1 == n2){
        return false;
    }
    //adding the symmetric relation by making n1 close proximity to n2
    if(n2.get_close_proximity_networks().contains(n1.get_name())){
        return true;
    }
    //if the network name is apart of the other networks close proximity network
        list then return true
    if(n1.get_close_proximity_networks().contains(n2.get_name())){
        return true;
    }
    //return false otherwise
    return false;
}
```

```
//this allows use to now modify all_valid configuration to the following
//since we now have anti-reflexive inside our close proximity rather than in the
    predicate all_configurations_valid we can just simply call
    valid_configuration(n1,n2)
public boolean all_configurations_valid(){
      //we want to check all possible configurations
      for(Network n1 : all_networks){
         for(Network n2 : all_networks){
             //if ANY configuration is invalid we return false as the setup is
                 invalid
             if(!valid_configuration(n1, n2){
                 return false;
             }
         }
      }
      //return true if it does not exit the loop with return false
      return true;
}
```

Note that since we have "close proximity" as a symmetric relation, that means if network A is inside of network B's close proximity list, that also means that network B is also inside of network A's close proximity list, this can be shown in code with the following.

```
//suppose this is the method for making two networks within close proximity
public void make_close_proximity(Network A, Network B){
    //this shows the symmetry since A is close proximity to B
    //also means B is close proximity to A
    //since we make them BOTH close proximity to each other.
    A.get_close_proximity_networks.add(B.get_name());
    B.get_close_proximity_networks().add(A.get_name());
}
```

.

Consider the following set-up: we have exactly three channels: hi, med, lo; and (at least) four networks Alpha, Bravo, Charlie, Delta with close proximity as defined in the following diagram:
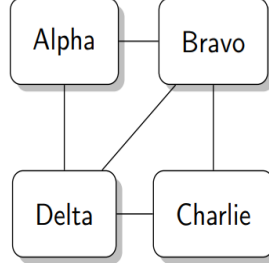

Figure 3: Network configuration between all four networks and three channels

We now want to refine our specification (i.e. add requirements) so that any model which satisfies all the requirements will necessarily have the structure detailed here.

(c) Extending your specification language if necessary, provide a set of requirements which, together with your answer to (b), ensures any satisfying model has the structure detailed here. (10 marks)

Solution:
Since in part (b) we gave requirements for a more general solution which will solve for any configuration (if there is a solution) we simply just have to add requirements so that all models have the structure shown in Figure 3. The following extra requirements can be added to achieve this:

(1) we have 4 networks and 3 channels

(2) two UNIQUE networks are in "close proximity" to every other network

(3) the other two networks are ONLY in close proximity to the networks in (2)

By saying that the set of all networks is X and the set of all channels is Y these requirements can be shown in predicate formulas in the following:

(1) $\psi_5 = (|X| = 4) \wedge (|Y| = 3)$
This essentially says that the size of our network set is 4 and the size of our channel set is 3

(2) $\psi_6 = (x1, x2 \exists X (\forall n1 \in X (is\_close\_proximity(x1, n1) \wedge (is\_close\_proximity(x2, n1)))) \wedge \neg(x1 = x2)$
This essentially says that there exists two networks x1 and x2 in the set of networks which is in close proximity to ALL other networks and those two networks are not the same network.

(3) $\psi_7 = (x1, x2 \exists X (\forall n1 \in X (is\_close\_proximity(x1, n1) \wedge (is\_close\_proximity(x2, n1)))) \wedge \neg(x1 = x2) \wedge$
$(u1, u2 \exists X (is\_close\_proximity(u1, x1) \wedge is\_close\_proximity(u1, x2) \wedge$
$is\_close\_proximity(u2, x1) \wedge is\_close\_proximity(u2, x2) \wedge$
$\neg is\_close\_proximity(u1, u2) \wedge \neg(u1 = u2) \wedge \neg(u1 = x1) \wedge$
$\neg(u1 = x2) \wedge \neg(u2 = x1) \wedge \neg(u2 = x2))$
This seems like a much more complicated predicate formula, however this is because we needed to write out ψ_6 to reference the networks x1 and x2, essentially it states that two networks, x1 and x2, are in close proximity to all other networks and that there are two

distinct networks u1 and u2 that are in close proximity to those networks and not in close proximity to each other and they are not the same network.

We will name our following predicates ψ_5 ψ_6 ψ_7 as correct_sizes, two_connected_to_all and two_unique_connected respectively These new requirements can also be added to the previous predicates ψ_{1-4} to give us our final requirement ψ as:
$\psi = \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5 \wedge \psi_6 \wedge \psi_7$
This can also be shown in the code above in part (b) along with the following additions below to introduce our new requirements.

```java
//we will also be adding new class variables
public Network connected_to_all1 = null;
public Network connected_to_all2 = null;
public Network uniq1 = null;
public Network uniq2 = null;
public boolean correct_sizes(){
    return (all_networks.size() == 4 && all_channels.size() == 3);
}
//our predicate to make sure we have only two networks which are in close proximity to
    all other networks
public boolean two_connected_to_all(){
    //this flag will be used to be a check if the network is
    //in close proximity to all other networks
    //if a network is not in close proximity to any network we want to go to the next
    //network as this one is not connected to ALL networks. we reset the flag at each
        iteration
    //if the flag is false.
    boolean flag = true;
    //count how many networks are connected to ALL networks
    int count = 0;
    //scan through all combinations of the networks
    for(Network n1 : all_networks){
        for(Network n2 : all_networks){
            //if n1 is not in close proximity to all other networks we want to set flag
                as false
            //and go to the next network
            if(!is_close_proximity(n1, n2) && n1 != n2){
                flag = false;
                break;
            }
        }
        //if the flag is false that means we dont want to increase the counter or update
        //our class vairables
        if(flag == false){
            flag = true;
            continue;
        }
        //otherwise we want to set the class variables in order
        if(connected_to_all2 == null && connected_to_all1 != null){
            connected_to_all2 = n1;
        }
```

```java
        if(connected_to_all1 == null){
            connected_to_all1 = n1;
        }
        //and increment the count
        count++;
    }
    //return the statement that the count must be exactly 2
    //and that the two networks connected to all the other networks
    //are not the same network
    return (count == 2) && (connected_to_all1 != connected_to_all2);
}


//our predicate to make sure that the other two networks are also connected to the
     other networks
//that connect to everything and are not connected to any other node
public boolean two_unique_connected(){
    //if we don't have two networks which are connected to all other networks
    if(null == connected_to_all1 || null == connected_to_all2) {
        return false;
    }
    //otherwise we want to isolate the other 2 networks and make sure they are
    //1. connected to the connected_to_all networks AND
    //2. those two networks are not connected to each other
    //3. those two networks are not the same network
    Network n1 = null;
    Network n2 = null;
    //find the two networks which are not the ones connected to the rest of the networks
    for(Network n : all_networks){
        if(n1 != null && (n != connected_to_all1 && n != connected_to_all2) && n2 ==
            null){
            n2 = n;
        }
        if((n != connected_to_all1 || n != connected_to_all2) && n1 == null){
            n1 = n;
        }
    }
    //if those two networks are NOT in close proximity with ONLY the two networks that
         are connected
    //to the rest, we return false
    if(n1.get_close_proximity_networks().size() != 2 &&
        n2.get_close_proximity_networks().size() != 2){
        return false;
    }
    //if those two networks are in close proximity to each other
    //return false
    if(is_close_proximity(n1, n2)){
        return false;
    }
    //if the network is not in close proximity with the two networks
    //connected to the all networks, return false
    if(!n1.get_close_proximity_networks().contains(connected_to_all2.get_name()) &&
            !n1.get_close_proximity_networks().contains(connected_to_all1.get_name())){
```

```
        return false;
    }
    if(!n2.get_close_proximity_networks().contains(connected_to_all2.get_name()) &&
            !n2.get_close_proximity_networks().contains(connected_to_all1.get_name())) {
        return false;
    }
    //otherwise return true since it has passed all checks
    unique1 = n1;
    unique2 = n2;
    return true;
}
```

With our new predicates which restrict all models to the configuration in Figure 3 we can finally modify all_configurations_valid which achieved ψ in the following code

```
//this allows use to now modify all_valid configuration to the following
//since we now have anti-reflexive inside our close proximity rather than in the
    predicate all_configurations_valid we can just simply call
    valid_configuration(n1,n2)
public boolean all_configurations_valid(){
    //we want to check all possible configurations
    for(Network n1 : all_networks){
        for(Network n2 : all_networks){
            //if ANY configuration is invalid we return false as the setup is invalid
            if(!valid_configuration(n1, n2)){
                return false;
            }
        }
    }
    //return true if it does not exit the loop with return false
    return correct_sizes() && two_connected_to_all() && two_unique_connected();
}
```

from these predicates we can also make the observation that since two networks are connected to every other networks, those networks must run on different channels, and since the other two networks are not connected to each other, they must run on the same channel (this will be proven in (c) and (d). The following is a satisfying model to this problem.

 (a) A = channel medium

 (b) B = channel High

 (c) C = channel medium

 (d) D = channel low

When this is run with the following test code given the same configuration, we receive the following output

```java
import java.util.*;

public class test {
    public static void main(String args[]) {
        Channel low = new Channel(0);
        Channel medium = new Channel(1);
        Channel high = new Channel(2);

        ArrayList<String> close_proximity_to_a = new ArrayList<String>();
        close_proximity_to_a.add("B");
        close_proximity_to_a.add("D");
        Network A = new Network("A", medium, close_proximity_to_a);

        ArrayList<String> close_proximity_to_b = new ArrayList<String>();
        close_proximity_to_b.add("A");
        close_proximity_to_b.add("C");
        close_proximity_to_b.add("D");
        Network B = new Network("B", high, close_proximity_to_b);

        ArrayList<String> close_proximity_to_c = new ArrayList<String>();
        close_proximity_to_c.add("B");
        close_proximity_to_c.add("D");
        Network C = new Network("C", medium, close_proximity_to_c);

        ArrayList<String> close_proximity_to_d = new ArrayList<String>();
        close_proximity_to_d.add("A");
        close_proximity_to_d.add("B");
        close_proximity_to_d.add("C");
        Network D = new Network("D", low, close_proximity_to_d);

        ArrayList<Channel> channels = new ArrayList<Channel>();
        channels.add(low);
        channels.add(medium);
        channels.add(high);

        ArrayList<Network> networks = new ArrayList<Network>();
        networks.add(A);
        networks.add(B);
        networks.add(C);
        networks.add(D);

        Predicates check = new Predicates(networks, channels);
        System.out.println("Correct size of configuration 4 networks 3 channels- : " +
            check.correct_sizes());
        System.out.println("Only two networks connected to all other networks: " +
            check.two_connected_to_all());
        System.out.println("Two networks connected to the ones above but not to each
            other: " + check.two_unique_connected());
```

```
System.out.println("Final valid configuration (main requirements): " +
    check.all_configurations_valid());
System.out.println("A connected to : " + A.get_close_proximity_networks());
System.out.println("B connected to : " + B.get_close_proximity_networks());
System.out.println("C connected to : " + C.get_close_proximity_networks());
System.out.println("D connected to : " + D.get_close_proximity_networks());
System.out.println("The two channels which were connected to all other channels
    satisfying psi_6 are : "
        + check.connected_to_all1.get_name() + " and "
        + check.connected_to_all2.get_name());
System.out.println("The two channels which are unique satisfying psi_7 are : "
        + check.unique1.get_name() + " and " + check.unique2.get_name());

    }
}
```

When the above code is run we get the following output

Figure 4: code output with the configuration above with same structure in Figure 3

Note this code produces the correct output and can be copied and pasted from the appendix, and tested with different models. I did not use the predicate make_close_proximity in my testing as i already implemented close proximity with the symmetric nature of adding both networks to each other's close proximity list.

(d) How would you use a SAT-solver (for First Order Logic) such as Z3 or IDP to show that in any
model satisfying your requirements, Alpha and Charlie use the same channel?   (5 marks)

Solution:
Similar to the lecture example for solving a simpler version of the zebra puzzle, we would load
each of our variables in so that each channel will be loaded in as an integer, and we would set
these channels to constant values. So high = 2, medium = 1 and low = 0. Then we would load
in our Networks as integers which range from 0-2 (indicating their channels). Next we would
assert the conditions that our defined close proximity networks cannot have equal values. So in
the defined model above, we have the following:

- Alpha != Bravo
- Alpha != Delta
- Bravo != Alpha
- Bravo != Charlie
- Bravo != Delta
- Charlie != Bravo
- Charlie != Delta
- Delta != Alpha
- Delta != Bravo
- Delta != Charlie

all these conditions enforce the close proximity in the given

- Alpha is in close proximity to Bravo and Delta
- Bravo is in close proximity to Alpha, Charlie and Delta
- Charlie is in close proximity to Bravo and Delta
- Delta is in close proximity to Alpha, Bravo and Charlie

which follows our structure in Figure 3 as we are also using 3 channels low, medium and high.
Next we would try to get all models by adding more conditions and exhausting all possibilities
to verify that all satisfying models have Alpha and Charlie on the same Network.

to get all models we would first run it with those conditions in place, then after we check
satisfiability and get the model, we will now make it so alpha cannot be channel 0 and repeat
till we have exhausted all possible channels giving all possible models.

This will also what we will be using for the bonus to implement the solution using Z3.

Bonus: Implement your solution to the previous question in a SAT-solver to show that in any model satisfying the requirements, Alpha and Charlie use the same channel. Either include a link to your solution or provide the code so it can be easily copy-pasted and run. (10* marks)

Solution:

```
;declaring our channels as integers
(declare-const low Int)
(declare-const medium Int)
(declare-const high Int)

;set low = 0, medium = 1, high = 2
(assert (= low 0))
(assert (= medium 1))
(assert (= high 2))

;now we want to declare our networks as integers
;showing which channel they are connected to by their integer value
(declare-const alpha Int)
(declare-const bravo Int)
(declare-const charlie Int)
(declare-const delta Int)

;assert the range of values, so all networks are connected from 0-2
;which is our range of channels 0, 1 and 2
(assert (and (< -1 alpha)(< alpha 3)))
(assert (and (< -1 bravo)(< bravo 3)))
(assert (and (< -1 charlie)(< charlie 3)))
(assert (and (< -1 delta)(< delta 3)))


;now we want to add our close proximity AND the rule that close proximity
;networks cannot use the same channel. note close proximity is anti-reflexive and
    symmetric!

;alpha is in close proximity to bravo and delta
(assert (not(= alpha bravo)))
(assert (not(= alpha delta)))

;bravo is in close proximity to alpha, charlie and delta
(assert (not(= bravo alpha)))
(assert (not(= bravo charlie)))
(assert (not(= bravo delta)))

;charlie is in close proximity to bravo and delta
(assert (not(= charlie bravo)))
(assert (not(= charlie delta)))

;delta is in close proximity to alpha, bravo and charlie
(assert (not(= delta alpha)))
(assert (not(= delta bravo)))
```

```
(assert (not(= delta charlie)))

;now we want to get any model satisfying this
(check-sat)
(get-model)

;to get all models we need to add restrictions by exhausting all possible channels alpha
;can be connected to. since alpha MUST be connected to a network, if we keep removing
    options
;e.g. if we say alpha can't connect to 0, then it cant connect to 1 then it cant
    connect to 2, then we have
;exhausted all options and gotten every model that satisfies this configuration

;restrict alpha so it cannot be on channel 1 and get the model
(assert (not(= alpha 1)))
(check-sat)
(get-model)

;restrict alpha so it cannot be on channel 0 and get the model
(assert (not(= alpha 0)))
(check-sat)
(get-model)

;restrict alpha so it cannot be on channel 2 and get the model

(assert (not(= alpha 2)))
(check-sat)
(get-model)
;at this point we have gotten all output possible so the output is the following
;sat
;(model
;  (define-fun bravo () Int
;    0)
;  (define-fun charlie () Int
;    1)
;  (define-fun alpha () Int
;    1)
;  (define-fun delta () Int
;    2)
;  (define-fun high () Int
;    2)
;  (define-fun medium () Int
;    1)
;  (define-fun low () Int
;    0)
;)
;sat
;(model
;  (define-fun bravo () Int
;    1)
;  (define-fun delta () Int
;    2)
```

```
;  (define-fun medium () Int
;     1)
;  (define-fun high () Int
;     2)
;  (define-fun charlie () Int
;     0)
;  (define-fun alpha () Int
;     0)
;  (define-fun low () Int
;     0)
;)
;sat
;(model
;  (define-fun bravo () Int
;     1)
;  (define-fun delta () Int
;     0)
;  (define-fun medium () Int
;     1)
;  (define-fun high () Int
;     2)
;  (define-fun charlie () Int
;     2)
;  (define-fun alpha () Int
;     2)
;  (define-fun low () Int
;     0)
;)
;unsat
;Z3(47, 10): ERROR: model is not available
```

As can be seen in the following output from the SAT solver Z3, all models that solve the configuration in Figure 3 have alpha and charlie using the same channel, because alpha and charlie have the same value, and bravo and delta use different channels to alpha and charlie and each other.

3   Problem 3

Recall the Fibonacci sequence defined as:

- F(0) = 0

- F(1) = 1

- F(n + 2) = F(n + 1) + F(n) for n ≥ 0

(a) Write a program FIB in $\mathcal{L}$ that computes the n-th Fibonacci number (for n ≥ 1). That is, if n ≥ 1 at the start of execution, then variable r has value F(n) at the end of execution (if Fib terminates).   (12 marks)

Solution:

We will use the variable r to store our fib(i) and i will eventually be equal to n assuming the program terminates. We will use the variable x to keep the value of f(i - 1). Note counting starts from 0 since we are also considering fib(0) = 0, the 0th Fibonacci number is 0, the first Fibonacci number is 1, the second Fibonacci number is 1, the third Fibonacci number is 2 and so on.

```
//variable that holds the previous value of f(i) -> (f(i - 1)) this is f(0)
x := 0;
//variable that holds f(i) -> (note this is (f(1))
r := 1;
//counter for loop
i := 1;
//while the counter is less than our value n
//note on each iteration (loop invariant we have)
//x = fib(i - 1)
//r = fib(i)
while i < n do
    //set f(i) = f(i - 2) + f(i - 1)
    //this is because we are using assignment and we increment i to i + 1
    //old r becomes f(i - 1) and old x becomes fib(i - 2)
    r := r + x;
    //update x to be f(i - 1) for next iteration
    //since r is updated to f(i) we have f(i) - f(i - 2)
    //giving f(i - 1) e.g. 1, 1 , 2 ,3, 5
    //if we have f(i) as 5, then f(i - 1) is 3
    //f(i - 2) is 2
    //and 5 - 2 = 3 which is f(i - 1) correctly updating for next iteration
    x := r - x;
    //increment loop counter
    i := i + 1;
od
```

Note that fib(n) = fib(n - 1) + fib(n - 2) is also equivalent to fib(n + 2) = fib(n + 1) + fib(n). This is because fib(1) and fib(0) is already defined for us so we don't end up with negative fib numbers and undefined behaviour. We exit the loop with our invariant being r = f(i) and i ≤ n. Our exit condition is when i ≥ n, that means i = n, giving us r = f(n)

(b) Prove, using Hoare Logic, that $\vdash \{n \geq 1\}$ Fib $\{r = F(n)\}$ (10 marks)

Solution:

For this solution conditions have been compacted in the following:

{init} = {n ≥ 1}

{invariant} = {(f(i) = r) ∧ (f(i - 1) = x) ∧ (i ≤ n) ∧ ({init})}

{updateRpre} = {(f(i + 1) = r + x) ∧ (f(i) = r) ∧ (i ≤ n) ∧ (n ≥ 1)} note f(i) = (r + x) - x

{updateRpost} = {(f(i + 1) = r) ∧ (f(i) = r - x) ∧ (i ≤ n) ∧ (n ≥ 1)}

{updateXpost} = {(f(i + 1) = r) ∧ (f(i) = x) ∧ (i ≤ n) ∧ (n ≥ 1)}

Below this proof we will also prove our proof obligations. The invariant is very similair to the one given in the lecture for the power program example. The overall proof is a mix between the lecture example for factorial and power function.

1. {init} (premise)

2. {f(0) = 0} x := 0 {x = f(0)} (assignment inference rule)

3. {f(1) = 1} r := 1 {r = f(1)} (assignment inference rule)

4. {r = f(1)} i := 1 {r = f(i)} (assignment inference rule)

5. {f(1) = 1} r := 1; i := 1 {r = f(i)} (sequence inference rule: 3, 4)

6. {updateRpre} r := r + x {updateRpost} (assignment inference rule)

7. {updateRpost} x := r - x {updateXpost} (assignment inference rule)

8. {updateXpost} i := i + 1 {invariant} (assignment inference rule)

9. {updateRpre} r := r + x; x := r - x {updateXpost} (sequence inference rule: 6, 7)

10. {updateRpre} r := r + x; x := r - x; i := i + 1 {invariant} (sequence inference rule: 9, 8)

11. (invariant ∧ (i < n)) → (updateRpre) (proof obligation, definition of Fibonacci)

12. {invariant ∧ (i < n)} r := r + x; x := r - x; i := i + 1{invariant}
    (consequence inference rule: 10, 11)

13. {invariant ∧ (i < n)} while ... od {invariant ∧ (r = f(i)) ∧ (i ≥ n)} (loop inference rule: 12)

14. {(0 = f(0)) ∧ (1 = f(1)) ∧ (init)} Fib {invariant ∧ (r = f(i)) ∧ (i ≥ n)} (seq)

15. (TRUE ∧ (n ≥ 1)) → (init ∧ 1 = f(1) ∧ 0 = f(0)) (proof obligation)

16. {TRUE ∧ (n ≥ 1)} FIB {invariant ∧ (r = f(i)) ∧ (i ≥ n)}(consequence inference rule: 14, 15)

17. (invariant ∧ (r = f(i)) ∧ (i ≥ n)) → r = f(n) (proof obligation)

18. {TRUE ∧ (n ≥ 1)} Fib {r = f(n)} (consequence inference rule: 16, 17)

Proof obligations used in the Hoare logic derivation:

Note our substitutions are the following:
$\{init\} = \{n \geq 1\}$
$\{invariant\} = \{(f(i) = r) \wedge (f(i - 1) = x) \wedge (i \leq n) \wedge (\{init\})\}$
$\{updateRpre\} = \{(f(i + 1) = r + x) \wedge (f(i) = r) \wedge (i \leq n) \wedge (n \geq 1)\}$

Proof for Line 11 - $(invariant \wedge (i < n)) \rightarrow (updateRpre)$

$(invariant \wedge (i < n)) \rightarrow ((f(i + 1) = r + x) \wedge (f(i) = r) \wedge (i \leq n) \wedge (n \geq 1))$

So our proof becomes:

$(f(i) = r) \wedge (f(i - 1) = x) \wedge (i \leq n) \wedge (n \geq 1)) \rightarrow ((f(i + 1) = r + x) \wedge (f(i) = r) \wedge (i \leq n) \wedge (n \geq 1))$

with the overlap $i < n$ and $i \leq n$ we can simplify that to $i < n$ because $(i < n) \rightarrow (i \leq n)$. allowing us only needing to prove:

$((r = f(i)) \wedge (x = f(i -1)) \wedge (i < n) \wedge (n \geq 1)) \rightarrow ((f(i + 1) = r + x) \wedge (f(i) = x) \wedge (i \leq n) \wedge (n \geq 1))$

To prove this we essentially need a proof that in the Fibonacci sequence:

If $f(i) = r$ and $f(i - 1) = x$ then that means $f(i + 1) = r + x$ for all Fibonacci numbers greater than or equal to 1.

Essentially we are proving $f(i + 1) = f(i) + f(i - 1)$ however, since $i < n$ and $f(0)$ is defined aswell as $f(1)$ we can substitute $i - 1$ into the expression to achieve $f(i) = f(i - 1) + f(i - 2)$ which is our definition of the Fibonacci sequence.

Proof for Line 15 - $(TRUE \wedge (n \geq 1)) \rightarrow (init \wedge (1 = f(1)) \wedge (0 = f(0)))$

$(TRUE \wedge (n \geq 1)) \rightarrow (init \wedge (1 = f(1)) \wedge (0 = f(0)))$

So our proof becomes:

show $1 = f(1)$ and $0 = f(0)$ is true. As can be seen init and $n \geq 1$ are equivalent. From our definition of the Fibonacci sequence, $f(0) = 0$ and $f(1) = 1$. So this is true.

Proof for Line 17 - $(invariant \wedge (r = f(i)) \wedge (i \geq n)) \rightarrow r = f(n)$

$(invariant \wedge (r = f(i)) \wedge (i \geq n)) \rightarrow r = f(n)$

So our proof becomes:

Show that $i = n$ by the end of the loop to get $f(n)$ by substitution. We have $i \geq n$ and our invariant states that $r = f(i)$ and $x = f(i - 1)$ and $i \leq n$. By combining the fact that $i \leq n$ and that $i \geq n$ in our first part of the implication this can be shown to be true. Using a variation of squeezes theorem, if $n \geq i \leq n$ then $i = n$. in a less formal way to put it, i is less than or equal to a number and greater than or equal to the same number, then i is that number. This gives us $i = n$, and $r = f(n)$, allowing for post condition weakening.

Therefore our proof is complete, since we have derived $\{TRUE \wedge n \geq 1\}$ Fib $\{r = f(n)\}$. Assuming our program Fib terminates we get the following deduction:
Since this was deduced with no presumptions that means everything deduces to the following
$$\vdash \{n \geq 1\} \text{ Fib } \{r = F(n)\}$$

(c) Annotate your code with the necessary assertions so that your proof can be reconstructed.

(3 marks)

For this solution conditions have been compacted in the following:
{init} = {n ≥ 1}
{invariant} = {(f(i) = r) ∧ (f(i - 1) = x) ∧ (i ≤ n) ∧ ({init})}
{updateRpre} = {(f(i + 1) = r + x) ∧ (f(i) = r) ∧ (i ≤ n) ∧ (n ≥ 1)} note f(i) = (r + x) - x
{updateRpost} = {(f(i + 1) = r) ∧ (f(i) = r - x) ∧ (i ≤ n) ∧ (n ≥ 1)}
{updateXpost} = {(f(i + 1) = r) ∧ (f(i) = x) ∧ (i ≤ n) ∧ (n ≥ 1)}

Using our derivation from our Hoare logic above we can annotate the code in the following (without the comments from (a)):

```
                    //{True n ≥ 1}
x := 0;             //{f(0) = 0}
r := 1;             //{f(1) = 1}
i := 1;             //{r = f(i)}
while y < n do      //{invariant}
                    //{updateRpre}
    r := r + x;     //{updateRpost}
    x := r - x;     //{updateXpost}
    i := i + 1;     //{invariant}
od                  //{r = f(i) ∧ invariant ∧ i ≥ n}
                    //{r = f(n)}
```

Bonus: Adjust your proof to show that your code terminates.   (3* marks)

Solution:
To show that the code terminates we can use a variant V which is a formula that computes to a value that begins as positive, and we need to show that this value keeps decreasing on each iteration of the loop. a suitable variant to achieve this would be V := (n - i). By using this we can change the annotations on the code above to prove termination (note that since the annotated code is a direct application of the Hoare logic derivation, this can also be applied to the Hoare logic proof).

```
                    //{True n ≥ 1}
x := 0;             //{f(0) = 0}
r := 1;             //{f(1) = 1}
i := 1;             //{r = f(i)}
while i < n do      //{invariant ∧ V = (n - i)}
                    //{updateRpre ∧ V = (n - i)}
    r := r + x;     //{updateRpost ∧ V = (n - i)}
    x := r - x;     //{updateXpost ∧ V = (n - i)}
    i := y + 1;     //{invariant ∧ V < (n - i)}
od                  //{r = f(i) ∧ invariant ∧ i ≥ n}
                    //{r = f(n)}
```

However we have two proof obligations before this proof can be complete

show (V := (n - i)) > 0 before entering loop

Given that n $\geq$ 1 from our initial conditions and that i $\leq$ n from our loop invariant then (n - i) $\geq$ 0 since n is larger than i, and n is also positive.

show [V := n - i] i = i + 1 [(V := < (n - i)]

on the precondition we get [v := n - (i + 1)]. Expanding out we get [v := (n - i - 1)]. as can be seen (n - i - 1) < (n - i)

Another way to phrase our variant, is that since v := n - i and our value i increments by one on each loop iteration, i < n and n does not change, we reach a state where v = 0 after i iterations. This completes the proof showing in fact our program Fib will terminate if n $\geq$ 1.

4 appendix

To run this code used in Section 2

1. Copy and paste all the code into files an editor of choice using the same class names

2. Make sure all files are in the same directory!

3. javac *.java

4. java Test

   Channel.java (name the file Channel.java)

```java
public class Channel{

    public int id;

    public Channel(int ID){
        this.id = ID;
    }

    public int get_Id(){
        return this.id;
    }
}
```

   Network.java (name the file Network.java)

```java
import java.util.*;

public class Network {

    public String name;
    public Channel connected;
    public ArrayList<String> close_proximity_networks;

    public Network(String name, Channel connected, ArrayList<String>
         close_proximity_networks) {
        this.name = name;
        this.connected = connected;
        this.close_proximity_networks = close_proximity_networks;
    }

    public String get_name() {
        return this.name;
    }

    public Channel get_channel() {
        return this.connected;
    }

    public ArrayList<String> get_close_proximity_networks() {
        return this.close_proximity_networks;
```

```
    }
}
```

Predicate.java (name the file Predicate.java)

```java
import java.util.*;

public class Predicates{

    public Network connected_to_all1 = null;
    public Network connected_to_all2 = null;
    public Network unique1 = null;
    public Network unique2 = null;
    public ArrayList<Network> all_networks;
    public ArrayList<Channel> all_channels;
    public Predicates(ArrayList<Network> networks, ArrayList<Channel> channels){
        this.all_networks = networks;
        this.all_channels = channels;
    }

    public boolean is_close_proximity(Network n1, Network n2){
        //anti-reflexive property by not letting n1 = n2 be true
        //i.e. if the two networks are the same return false!
        if(n1 == n2){
            return false;
        }
        //adding the symmetric relation by making n1 close proximity to n2
        if(n2.get_close_proximity_networks().contains(n1.get_name())){
            return true;
        }
        //if the network name is apart of the other networks close proximity network list
            then return true
        if(n1.get_close_proximity_networks().contains(n2.get_name())){
            return true;
        }
        //return false otherwise
        return false;
    }

    public boolean valid_configuration(Network n1, Network n2){
        //if the two networks are in close proximity then we check the channels are
            different to return true
        if(is_close_proximity(n1, n2)){
            //both networks use different channels
            return n1.get_channel().get_Id() != n2.get_channel().get_Id();
        }
        //if the channels are not in close proximity then it is valid eitherway
        return true;
    }

    //this allows use to now modify all_valid configuration to the following
    //since we now have anti-reflexive inside our close proximity rather than in the
```

```java
    predicate all_configurations_valid we can just simply call
    valid_configuration(n1,n2)
public boolean all_configurations_valid(){
    //we want to check all possible configurations
    for(Network n1 : all_networks){
        for(Network n2 : all_networks){
            //if ANY configuration is invalid we return false as the setup is invalid
            if(!valid_configuration(n1, n2)){
                return false;
            }
        }
    }
    //return true if it does not exit the loop with return false
    return correct_sizes() && two_connected_to_all() && two_unique_connected();
}


//our predicate to make sure we have the correct amount of networks and channels
public boolean correct_sizes(){
    return (all_networks.size() == 4 && all_channels.size() == 3);
}


//our predicate to make sure we have only two networks which are in close proximity to
    all other networks
public boolean two_connected_to_all(){
    //this flag will be used to be a check if the network is
    //in close proximity to all other networks
    //if a network is not in close proximity to any network we want to go to the next
    //network as this one is not connected to ALL networks. we reset the flag at each
        iteration
    //if the flag is false.
    boolean flag = true;
    //count how many networks are connected to ALL networks
    int count = 0;
    //scan through all combinations of the networks
    for(Network n1 : all_networks){
        for(Network n2 : all_networks){
            //if n1 is not in close proximity to all other networks we want to set flag
                as false
            //and go to the next network
            if(!is_close_proximity(n1, n2) && n1 != n2){
                flag = false;
                break;
            }
        }
        //if the flag is false that means we dont want to increase the counter or update
        //our class vairables
        if(flag == false){
            flag = true;
            continue;
        }
        //otherwise we want to set the class variables in order
        if(connected_to_all2 == null && connected_to_all1 != null){
```

```java
            connected_to_all2 = n1;
        }

        if(connected_to_all1 == null){
            connected_to_all1 = n1;
        }
        //and increment the count
        count++;
    }
    //return the statement that the count must be exactly 2
    //and that the two networks connected to all the other networks
    //are not the same network
    return (count == 2) && (connected_to_all1 != connected_to_all2);
}


//our predicate to make sure that the other two networks are also connected to the
    other networks
//that connect to everything and are not connected to any other node
public boolean two_unique_connected(){
    //if we don't have two networks which are connected to all other networks
    if(null == connected_to_all1 || null == connected_to_all2) {
        return false;
    }
    //otherwise we want to isolate the other 2 networks and make sure they are
    //1. connected to the connected_to_all networks AND
    //2. those two networks are not connected to each other
    //3. those two networks are not the same network
    Network n1 = null;
    Network n2 = null;
    //find the two networks which are not the ones connected to the rest of the networks
    for(Network n : all_networks){
        if(n1 != null && (n != connected_to_all1 && n != connected_to_all2) && n2 ==
            null){
            n2 = n;
        }
        if((n != connected_to_all1 || n != connected_to_all2) && n1 == null){
            n1 = n;
        }
    }
    //if those two networks are NOT in close proximity with ONLY the two networks that
         are connected
    //to the rest, we return false
    if(n1.get_close_proximity_networks().size() != 2 &&
        n2.get_close_proximity_networks().size() != 2){
        return false;
    }
    //if those two networks are in close proximity to each other
    //return false
    if(is_close_proximity(n1, n2)){
        return false;
    }
    //if the network is not in close proximity with the two networks
```

```java
        //connected to the all networks, return false
        if(!n1.get_close_proximity_networks().contains(connected_to_all2.get_name()) &&
                !n1.get_close_proximity_networks().contains(connected_to_all1.get_name())){
            return false;
        }
        if(!n2.get_close_proximity_networks().contains(connected_to_all2.get_name()) &&
                !n2.get_close_proximity_networks().contains(connected_to_all1.get_name())) {
            return false;
        }
        //otherwise return true since it has passed all checks
        unique1 = n1;
        unique2 = n2;
        return true;
    }

}
```

Test.java (name the file Test.java)

```java
import java.util.*;

public class Test {
    public static void main(String args[]) {
        Channel low = new Channel(0);
        Channel medium = new Channel(1);
        Channel high = new Channel(2);

        ArrayList<String> close_proximity_to_a = new ArrayList<String>();
        close_proximity_to_a.add("B");
        close_proximity_to_a.add("D");
        Network A = new Network("A", medium, close_proximity_to_a);

        ArrayList<String> close_proximity_to_b = new ArrayList<String>();
        close_proximity_to_b.add("A");
        close_proximity_to_b.add("C");
        close_proximity_to_b.add("D");
        Network B = new Network("B", high, close_proximity_to_b);

        ArrayList<String> close_proximity_to_c = new ArrayList<String>();
        close_proximity_to_c.add("B");
        close_proximity_to_c.add("D");
        Network C = new Network("C", medium, close_proximity_to_c);

        ArrayList<String> close_proximity_to_d = new ArrayList<String>();
        close_proximity_to_d.add("A");
        close_proximity_to_d.add("B");
        close_proximity_to_d.add("C");
        Network D = new Network("D", low, close_proximity_to_d);

        ArrayList<Channel> channels = new ArrayList<Channel>();
        channels.add(low);
        channels.add(medium);
```

```
        channels.add(high);

        ArrayList<Network> networks = new ArrayList<Network>();
        networks.add(A);
        networks.add(B);
        networks.add(C);
        networks.add(D);

        Predicates check = new Predicates(networks, channels);
        System.out.println("Correct size of configuration 4 networks 3 channels : " +
            check.correct_sizes());
        System.out.println("Only two networks connected to all other networks: " +
            check.two_connected_to_all());
        System.out.println("Two networks connected to the ones above but not to each other:
            " + check.two_unique_connected());
        System.out.println("Final valid configuration (main requirements): " +
            check.all_configurations_valid());
        System.out.println("A connected to : " + A.get_close_proximity_networks());
        System.out.println("B connected to : " + B.get_close_proximity_networks());
        System.out.println("C connected to : " + C.get_close_proximity_networks());
        System.out.println("D connected to : " + D.get_close_proximity_networks());
        System.out.println("The two channels which were connected to all other channels
            satisfying psi_6 are : "
                + check.connected_to_all1.get_name() + " and "
                + check.connected_to_all2.get_name());
        System.out.println("The two channels which are unique satisfying psi_7 are : "
                + check.unique1.get_name() + " and " + check.unique2.get_name());
    }
}
```