RELAZIONE DEL PROGETTO “ALLOCAZIONE DELLE RISORSE” DI RETI DI CALCOLATORI

Studenti: **Matricole:**

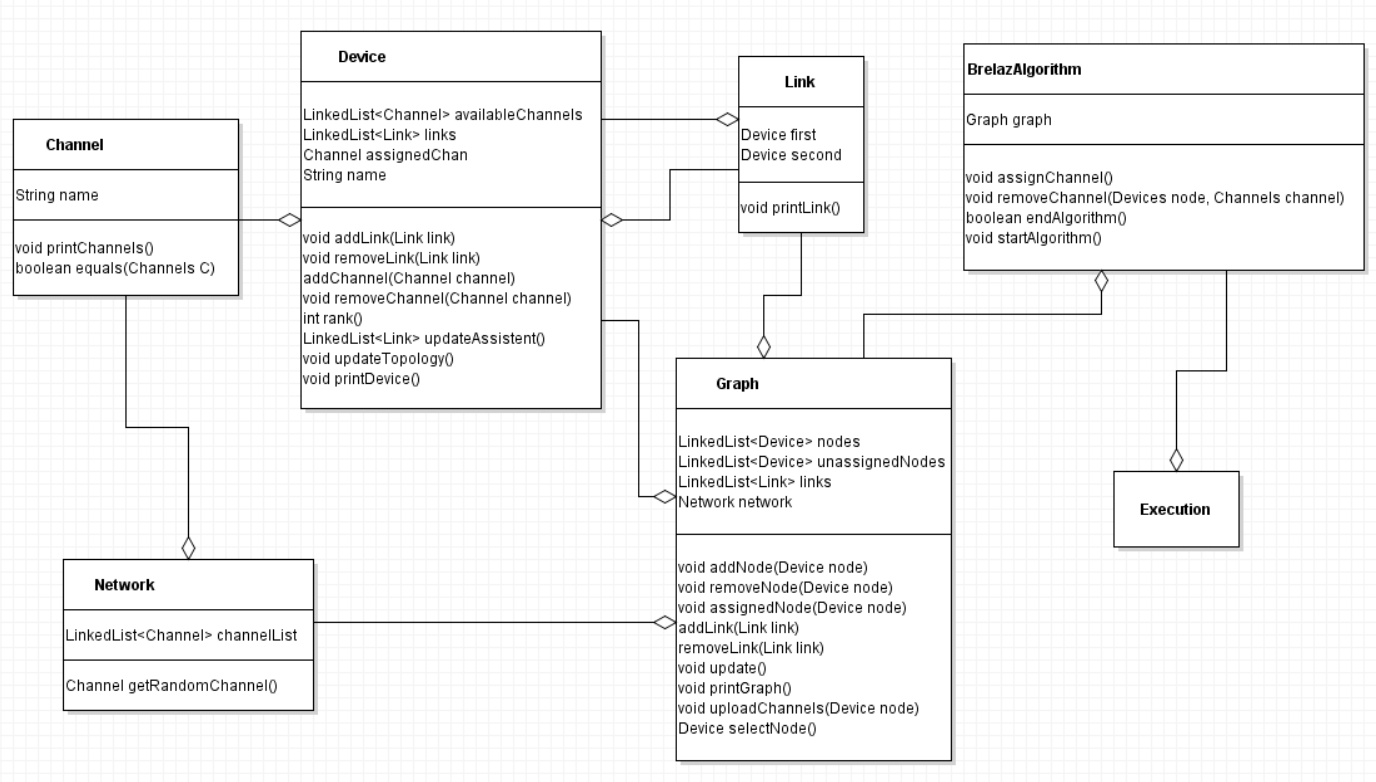
Massimiliano Sirgiovanni 6357449
Luca Donzelli 6355924

E-mail:

massimiliano.sirgiovanni@stud.unifi.it
luca.donzelli1@stud.unifi.it

DESCRIZIONE SOLUZIONE ADOTTATA

Il progetto assegnatoci è basato sull’**allocazione di** un numero K di **risorse**, **tramite l’algoritmo di Brélaz**, ad un numero N di dispositivi. **Il linguaggio di programmazione da noi selezionato** per l’attuazione del progetto **è Java**, il quale, grazie alla caratteristica di essere orientato agli oggetti, ha permesso di analizzare il problema nelle sue sottoparti e, successivamente, di sintetizzarle per ottenere il risultato desiderato. Tramite l’utilizzo di un **diagramma UML**, è possibile osservare il programma nella sua completezza.



Creazione del Grafo

Dovendo noi applicare un determinato algoritmo ad un grafo, il primo punto da analizzare era la sua realizzazione e memorizzazione.
Un grafo è composto da un insieme di **nodi** e da un insieme di **archi**. Per questo sono state create due classi concrete, *"Link"* e *"Device"*, che rappresentano, rispettivamente, gli archi ed i nodi.

Classe Link:

Partiamo dalla più semplice delle due, la classe *"Link"*, la quale **contiene al suo interno due attributi privati** *"Device"*. **Tali attributi rappresentano i due nodi collegati dall'arco**. I metodi implementati in questa classe sono, esclusivamente, i **getters** dei due nodi ed un metodo ***printLink()***, utile per la rappresentazione *"grafica"* del grafo.

Classe Device:

Proseguiamo ora con la classe che rappresenta le fondamenta di tale progetto, la classe *"Device"*. Come già precedentemente accennato, essa rappresenta un nodo del grafo su cui si vuole lavorare. Tale classe contiene **quattro attributi privati**:

1. *name*
2. *availableChannels*
3. *assignedChan*
4. *links*

Il primo, *"name"*, è una **stringa contenente il nome** che si vuole assegnare al nodo concreto. Tale stringa viene passata al costruttore della classe ed ha la funzionalità di identificare il nodo *(per questo sarebbe necessario dare nomi diversi ai nodi di uno stesso grafo)* e di rendere la stampa del grafo più leggibile per l'utente.

Costruttore della classe Device:

```
public Devices(String name) {
        this.name = name;
}
```

Passiamo ora al secondo attributo, *"availableChannels"*, il quale è fondamentale per la realizzazione dell'algoritmo. Si tratta di una **LinkedList**, nella quale **vengono memorizzati i canali** (oggetti di classe *"Channel"*) che possono essere utilizzati dal nodo. Simile è l'attributo *"links"*, il quale è una LinkedList, come *"availableChannels"*, però questa

contiene gli archi collegati al nodo in questione. Infine, l'ultimo attributo è *"assignedChan"*, il quale **indica il canale** (risorsa) **che è stato assegnato ad un dispositivo**. Ovviamente **se tale attributo è pari a null**, **significherà che nessun canale è stato assegnato al nodo**, nel quale è contenuto l'attributo.
Questi sono gli attributi che sono contenuti nella classe *"Device"*, ora visioniamo i metodi e le azioni che può compiere tale classe. Naturalmente i primi metodi ad essere implementati sono i **getters ed i setters**, quando necessari, degli attributi. Passiamo poi agli **add e remove di tutte le LinkedList** della classe (*availableChannels e links*).
E' presente poi **un metodo** per il **calcolo del rango**. Tale metodo, attraverso l'uso di uno **Stream**, verifica quanti archi sono presenti nella **LinkedList** ***"links"***. Per fare ciò utilizza il metodo ***count()*** degli **Stream**, come si può vedere dal codice scritto sotto.

```
public int rank() {
        int n = 0;
        Stream<Arch> stream = arch.stream();
        n = (int) stream.count();
        return n;
}
```

Proseguendo oltre, metodo ***equals(Devices node)*** **verifica se due nodi sono uguali**. Tale **controllo viene effettuato tramite la verifica del nome**, il quale dovrebbe essere unico per ogni nodo del grafo. D'altra parte, il metodo ***printDevice()*** non fa altro che **stampare le informazioni principali** del nodo stesso, richiamando poi, tramite iteratore, per ogni arco presente nella LinkedList *links,* il metodo ***printLink()***.
Gli ultimi due metodi rimanenti, sono tra loro inscindibili ed hanno il compito di **aggiornare la topologia del nodo**. Ovviamente questi metodi saranno utilizzati per aggiornare la topologia dell'intero grafo, ma ciò verrà esposto successivamente. I due metodi sono ***updateAssistent()*** ed ***updateTopology()***. Il primo metodo, attraverso l'utilizzo di un **buffer di supporto**, **memorizza tutti gli archi che devono essere cancellati**, secondo le regole dettate dall'algoritmo, per fare ciò si utilizza un iterator sul buffer di supporto. Dopo aver memorizzato tutti gli archi nel **buffer**, quest'ultimo viene **restituito come output**.

```
public LinkedList<Arch> updateAssistent() {
        LinkedList<Arch> buffer = new LinkedList<Arch>();
        Iterator<Arch> iterator = arch.iterator();
        Arch arch = null;
        while (iterator.hasNext()) {
        arch = iterator.next();
        if (arch.getB().getAssignedChan() != null && arch.getA().getAssignedChan() != null
                && arch.getA().getAssignedChan().equals(arch.getB().getAssignedChan())) {
                        buffer.add(arch);
                }
```

```
        }
        return buffer;
}
```

Questo metodo viene richiamato dal metodo ***updateTopology()***, il quale memorizza il buffer di supporto con gli archi da rimuovere e successivamente procede a rimuoverli dalla LinkedList *links*.
Ovviamente prima dell'esecuzione di tali istruzioni, viene effettuato un controllo per evitare eccezioni prevedibili (*NullPointerException*) o per evitare si eseguire istruzioni inutili su una lista vuota. In entrambi i casi non è necessario effettuare l'aggiornamento.

```
public void updateTopology() throws Exception {
        if (getAssignedChan() != null && !arch.isEmpty()) {
                LinkedList<Arch> buffer = updateAssistent();
                Iterator<Arch> iterator = buffer.iterator();
                while (iterator.hasNext()) {
                        this.removeArch(iterator.next());
                }
        }
}
```

Avendo ora visto il funzionamento e le potenzialità della classe Device, possiamo analizzare la classe che ci permette di unire nodi ed archi al fine di creare un vero e proprio grafo, *"Graph"*.

Classe Graph:

La classe **Graph** ha il compito di rappresentare il grafo effettivo memorizzando i nodi e gli archi da cui sono collegati.
Per tale compito la classe contiene *quattro attributi*:

1. *nodes*
2. *unassignedNotes*
3. *links*
4. *network*

Il primo attributo è una **LinkedList**, nodes, che contiene oggetti di classe **Device**, ovvero i nodi del grafo. Questi verranno aggiunti dall'utente tramite un metodo addNodes().
L'attributo *unassignedNodes*, anch'esso una **LinkedList** di **Device**, permette di **tenere traccia dei nodi a cui non sono stati attribuiti canali**. Ciò permette di evitare controlli successivi, in fase di scelta del nodo a cui assegnare un canale o durante il controllo per

determinare la fine dell'algoritmo *(che verrà affrontato successivamente)*, semplificando la scrittura del codice e permettendo di salvare tempo di esecuzione.
La **LinkedList** *links* ha il compito di memorizzare tutti gli archi presenti all'interno del grafo. La sua funzione è quella di ***"ponte"*** tra l'utente e la classe device, infatti ogni volta che un arco viene aggiunto dall'utente, questo viene memorizzato all'interno dei due Device che collega. Questo permette di **diminuire la percentuale di errori commessi dall'utente** nell'aggiunta degli archi, permettendo all'utente di inserire una sola volta un arco invece di inserirlo in ogni Device coinvolto.
L'attributo *network*, la cui classe vedremo più avanti, è responsabile della memorizzazione dei canali di comunicazione che verranno via via assegnati ai vari **Device**.
Questi quattro attributi, privati, sono accessibili tramite comuni getters.

Il metodo ***addNode()*** aggiunge il **Device** passato come passato alle due liste di nodi del grafo. I nodi vengono aggiunti prima dell'esecuzione dell'algoritmo non dovrebbero avere un canale assegnato, ma dato che è possibile cambiare il canale assegnato del nodo anche all'infuori dell'algoritmo, per evitare errori viene effettuato un semplice controllo. Su tale device verrà inizializzata la lista di canali disponibili, tramite il metodo ***uploadChannels(node)***, la quale sarà inizialmente corrispondente a quella del Network.

```
public void addNode(Device node) throws Exception {
        nodes.add(node);
        if (node.getAssignedChan() == null) {
                unassignedNodes.add(node);
                uploadChannels(node);
        }
}

public void removeNode(Devices node) throws Exception {
        nodes.remove(node);
}
```

Il metodo ***assignedNodes(Device device)*** verrà chiamato quando un canale dovesse essere assegnato ad un nodo, rimuovendo tale nodo dalla lista *unassignedNodes*

```
public void assignedNode(Device node) throws Exception {
        unassignedNodes.remove(node);
}
```

Passiamo ora all'*addLink()*, la quale riceve in ingresso un **oggetto Link** e lo aggiunge alla lista *links* del grafo e ad ognuno dei nodi connessi da tale arco.

Per aggiungere il link al nodo corrispondente viene iterata la lista di **Device**, nodes, e si verifica, per ogni nodo, se questo è uno dei due nodi collegati dall'arco.

```
public void addLink(Link link) throws Exception {
	links.add(link);
	Iterator<Devices> iterator = nodes.iterator();
	Devices node = null;
	while (iterator.hasNext()) {
		node = iterator.next();
		if (link.getFirst().equals(node) || link.getSecond().equals(node)) {
			node.addLink(link);
		}
	}
}
```

La ***removeLink()*** è una semplice remove sulla lista *links*.

```
public void removeLink(Link link) throws Exception {
	links.remove(link);
}
```

Continuiamo ad analizzare i metodi della classe Graph, osservando ***update()***. Questo è il metodo usato per aggiornare la topologia del grafo, nel quale viene iterata la lista *nodes* e viene chiamato il metodo ***updateTopology()***, che abbiamo già visto nella classe Device. Questo metodo verrà chiamato ad ogni ciclo dell'algoritmo dal metodo *assignChannel()* della classe *BrelazAlgorithm*.

```
public void update() throws Exception {
	Iterator<Device> iterator = nodes.iterator();
	while (iterator.hasNext()) {
		iterator.next().updateTopology();
	}
}
```

Il metodo ***printGraph()*** stampa su console una descrizione del grafo, iterando la lista dei nodi e chiamando, su ogni nodo, il metodo ***printDevice()***, già descritto in precedenza.

```
public void printGraph() {
	Iterator<Device> iterator = nodes.iterator();
	while (iterator.hasNext()) {
		iterator.next().printDevice();
	}}
```

Il metodo ***uploadChannels(Devices node)*** ha la funzione di inizializzare la lista *availableChannels* dell'oggetto *Device* passato come argomento. Il compito viene **svolto iterando la lista di canali del network** e **aggiungendo via via i canali alla lista del device**, così che *availableChannels* sia inizializzato con tutti i canali del network.

```
public void uploadChannels(Device node) throws Exception {
        Iterator<Channel> iterator = network.getChannelList().iterator();
        while (iterator.hasNext()) {
                node.addChannel(iterator.next());
        }
}
```

Il metodo ***selectNode()*** si occupa di **selezionare**, dalla lista di nodi senza un canale assegnato, **il device di grado più alto**. Il grado, ovvero il numero di archi connessi ad un nodo, **viene calcolato col metodo *node.rank()***. Iterando la lista *unassignedNodes*, per ogni device viene calcolato il grado e se questo è maggiore il nodo corrente diventa quello da restituire. **Una volta iterata la lista viene restituito il nodo con grado massimo**.

```
public Device selectNode() {
        Iterator<Device> iterator = getUnassignedNodes().iterator();
        Device node = iterator.next();
        int max = node.rank();
        Device temp = null;
        while (iterator.hasNext()) {
                temp = iterator.next();
                if (max < temp.rank()) {
                        node = temp
                        max = temp.rank();
                }
        }
        return node;
}
```

Creazione delle risorse

Dopo aver realizzato le classi abilitate alla creazione del grafo, **passiamo ora alla realizzazione delle risorse da assegnare ai vari dispositivi**, i canali.

Classe Channel:

Dato che il focus del progetto è l'implementazione di un algoritmo di assegnazione risorse, non abbiamo la necessità di implementare un canale di comunicazione effettivamente funzionante.
Perciò **la classe Channel**, la quale rappresenta appunto le risorse da distribuire ai dispositivi, **è** semplicemente **definita** dall'**unico attributo** che la costituisce, una stringa (*name*). I metodi sono quelli di base, il costruttore e un ***getName()***, oltre a **un metodo *printChannels()* per la rappresentazione grafica del canale** (non fa altro che stampare il nome dell'oggetto) e a un metodo ***equals(Channel C)*** per confrontarlo con altri oggetti Channel. In questo caso due Channel vengono considerati uguali se hanno lo stesso name.

Classe Network:

La classe **network rappresenta la rete di telecomunicazioni**, è quindi **adibita allo 'storage' dei canali** che, come precedentemente accennato, sono le risorse che andranno assegnate come obiettivo del progetto.
La classe ha **un solo attributo**, *channelsList,* che nient'altro è se non una **LinkedList** di Channel. E' possibile accedere ad essa tramite un metodo getter, ***getChannelList()*** che **restituisce** un oggetto **LinkedList<Channel>**.
Essendo necessario in particolari casi avere un canale random da assegnare ad un device, è stato scritto un metodo ***getRandomChannel()*** che **restituisce un oggetto Channel casualmente estratto dalla lista channelList**.

```
public Channel getRandomChannel() {
        return channelList.get((int) (Math.random() * channelList.size()));
}
```

Per fare ciò viene usata la funzione **Math.random()** da cui **viene generato un double casuale tra 0 e 0.999**.. che **viene moltiplicato per la dimensione della lista** e convertito in intero. Così facendo **abbiamo un indice casuale tra 0 e N-1**, **dove N è la dimensione della lista**. Verrà restituito il channel con tale indice nella lista.

Applicazione dell'Algoritmo

Ora che abbiamo creato il grafo e le risorse, possiamo organizzare l'applicazione dell'algoritmo. Quest'ultimo segue, ovviamente, i passaggi richiesti dal testo del progetto.

Classe BrelazAlgorithm:

Questa è la classe, BrelazAlgorithm, che **contiene** tutte **le informazioni ed i metodi** necessari **per l'applicazione dell'algoritmo di Brélaz**. Procediamo con ordine.
Al costruttore della classe **BrelazAlgorithm**, durante la creazione di un oggetto di tale classe, **viene passato** solo **un oggetto Graph** (un grafo). Tale grafo rappresenta anche l'unico attributo privato della classe. Il primo metodo che andremo ad analizzare è *assignChannel()*, il quale, come suggerisce il nome stesso, **assegna un canale ad un nodo**.
Il nodo viene scelto richiamando il metodo *selectNode()* della classe Graph. In questo modo, come già visto in precedenza, **selezioniamo il nodo con grado maggiore**, ovvero con il maggior numero di archi collegati, ed al quale non sia già stato assegnato un canale. **Una volta scelto il nodo vi sono due possibilità**. La prima è che il nodo abbia, **all'interno della sua lista di canali**, un possibile canale da utilizzare. La seconda possibilità implica che tale la lista all'interno del nodo sia vuota, quindi **il canale viene selezionato**, in modo aleatorio, **dalla lista di canali contenuta nel Network**. In entrambi i casi, **il canale viene eliminato dalla lista di tutti i nodi adiacenti** tramite il metodo *removeChannel(Device device, Channel channel)* e, successivamente, **si notifica al grafo che al nodo è stato assegnato il canale** e si **effettua un aggiornamento della topologia**.

```
public void assignChannel() throws Exception {
        Device node = graph.selectNode();
        Channel channel = null;
        if (!node.getAvailableChannels().isEmpty()) {
                channel = node.getAvailableChannels().pop();
                node.setAssignedChan(channel);
        } else {
                channel = graph.getNetwork().getRandomChannel();
                node.setAssignedChan(channel);
        }
        removeChannel(node, channel);
        graph.assignedNode(node);
        graph.update();
}
```

Proseguendo, analizziamo il, già precedentemente citato, metodo *removeChannel*, il quale **riceve in input un oggetto di classe Device e uno di classe Channel**. Il metodo **elimina**, da tutti i Device adiacenti a quello passato in input, **il canale passato in input al metodo**. Per fare ciò **sfrutta la lista di archi** del nodo su cui sta lavorando. Per fare ciò **itera sulla LinkedList** *links* e, per ogni arco, rimuove il canale dalle liste dei due nodi collegati all'arco.

```
public void removeChannel(Device node, Channel channel) throws Exception {
        Iterator<Link> iterator = node.getLinks().iterator();
        while (iterator.hasNext()) {
                Link link = iterator.next();
                link.getFirst().removeChannel(channel);
                link.getSecond().removeChannel(channel);
        }
}
```

Infine, gli ultimi due metodi hanno la funzione di ***"start&stop"*** per l'algoritmo. Si tratta di *endAlgorithm()*, il quale **restituisce true se la lista di nodi non assegnati sul grafo è vuota** (*Ovvero se tutti i dispositivi hanno un canale assegnato*) e *startAlgorithm()*. Quest'ultimo, tramite un **ciclo while**, richiama, finché *endAlgorithm()* non restituisce true, il metodo *assignChannel()*.

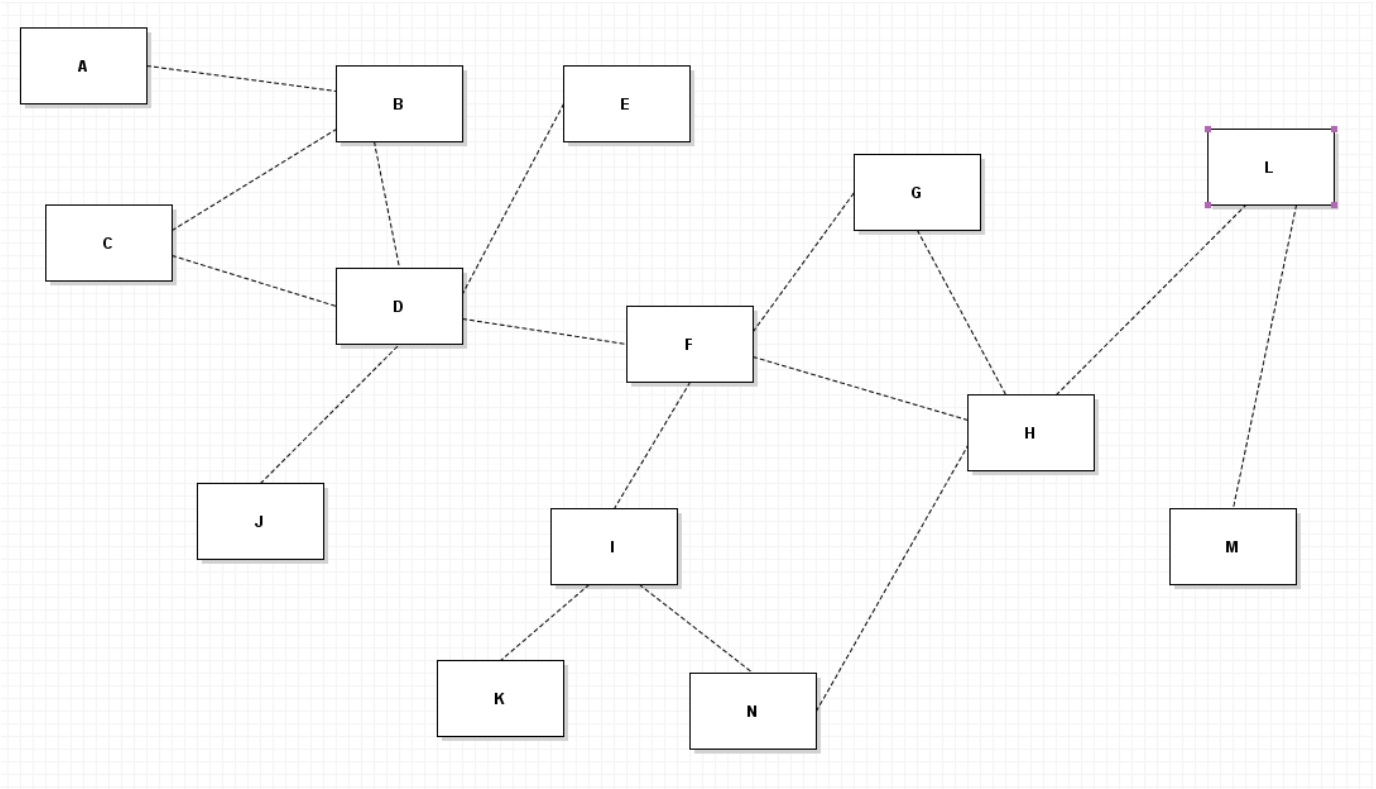
```
public boolean endAlgorithm() {
        return graph.getUnassignedNodes().isEmpty();
}

public void startAlgorithm() throws Exception {
        while (!endAlgorithm()) {
                assignChannel();
        }
}
```

Testing

Osserviamo ora l'applicazione del programma in alcuni casi completi, in modo tale da verificare la correttezza del programma. Prenderemo in considerazione tre casi, usufruendo della classe **Execution** nella quale è contenuto il ***main***, trattando un caso *"speciale"* e due casi medi. Il grafo da cui partiremo per effettuare i nostri test è identico a

quello presente su testo del progetto. Per ragioni pratiche sono stati assegnati dei nomi ai nodi del grafo, nel nostro caso il grafo ha assunto tale forma:



Per quanto riguarda la rappresentazione grafica all'interno del programma, il grafo viene stampato in questo modo:

```
Node: A
Assigned Channel: There is no assigned channel
The arch connects the following nodes: A->B
Node: B
Assigned Channel: There is no assigned channel
The arch connects the following nodes: A->B
The arch connects the following nodes: B->C
The arch connects the following nodes: B->D
Node: C
Assigned Channel: There is no assigned channel
The arch connects the following nodes: B->C
The arch connects the following nodes: C->D
Node: D
Assigned Channel: There is no assigned channel
The arch connects the following nodes: F->D
The arch connects the following nodes: B->D
The arch connects the following nodes: C->D
The arch connects the following nodes: E->D
The arch connects the following nodes: J->D
Node: E
Assigned Channel: There is no assigned channel
The arch connects the following nodes: E->D
Node: F
Assigned Channel: There is no assigned channel
The arch connects the following nodes: F->H
The arch connects the following nodes: I->F
The arch connects the following nodes: G->F
The arch connects the following nodes: F->D
Node: G
Assigned Channel: There is no assigned channel
The arch connects the following nodes: G->H
The arch connects the following nodes: G->F
```

```
Node: H
Assigned Channel: There is no assigned channel
The arch connects the following nodes: H->L
The arch connects the following nodes: H->N
The arch connects the following nodes: G->H
The arch connects the following nodes: F->H
Node: I
Assigned Channel: There is no assigned channel
The arch connects the following nodes: I->N
The arch connects the following nodes: I->K
The arch connects the following nodes: I->F
Node: J
Assigned Channel: There is no assigned channel
The arch connects the following nodes: J->D
Node: K
Assigned Channel: There is no assigned channel
The arch connects the following nodes: I->K
Node: L
Assigned Channel: There is no assigned channel
The arch connects the following nodes: L->M
The arch connects the following nodes: H->L
Node: M
Assigned Channel: There is no assigned channel
The arch connects the following nodes: L->M
Node: N
Assigned Channel: There is no assigned channel
The arch connects the following nodes: H->N
The arch connects the following nodes: I->N
```

Partiamo con il primo caso intermedio, al nostro grafo, composto da **quattordici nodi**, assegnamo un Network contenente **cinque canali**. *Quali sono le conseguenze della nostra scelta?*

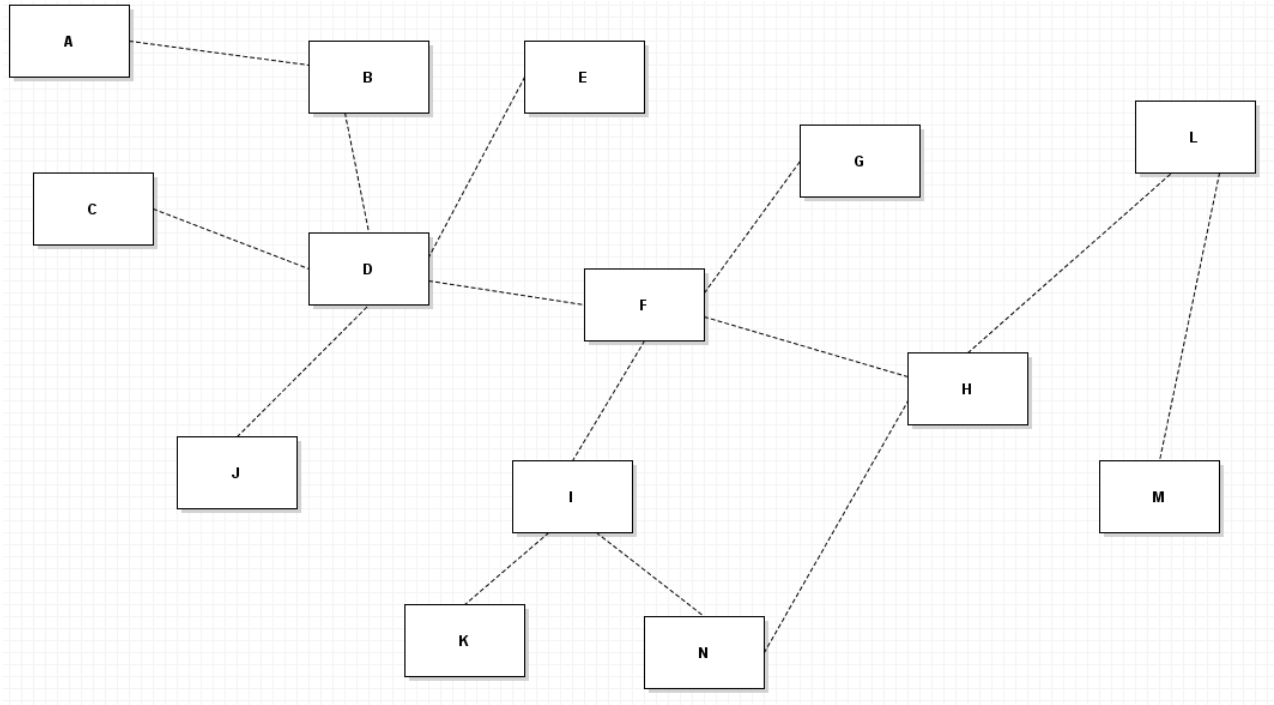
```
Node: A
Assigned Channel: C0
The arch connects the following nodes: A->B
Node: B
Assigned Channel: C1
The arch connects the following nodes: A->B
The arch connects the following nodes: B->C
The arch connects the following nodes: B->D
Node: C
Assigned Channel: C2
The arch connects the following nodes: B->C
The arch connects the following nodes: C->D
Node: D
Assigned Channel: C0
The arch connects the following nodes: F->D
The arch connects the following nodes: B->D
The arch connects the following nodes: C->D
The arch connects the following nodes: E->D
The arch connects the following nodes: J->D
Node: E
Assigned Channel: C1
The arch connects the following nodes: E->D
Node: F
Assigned Channel: C1
The arch connects the following nodes: F->H
The arch connects the following nodes: I->F
The arch connects the following nodes: G->F
The arch connects the following nodes: F->D
Node: G
Assigned Channel: C2
The arch connects the following nodes: G->H
The arch connects the following nodes: G->F
```

```
Node: H
Assigned Channel: C0
The arch connects the following nodes: H->L
The arch connects the following nodes: H->N
The arch connects the following nodes: G->H
The arch connects the following nodes: F->H
Node: I
Assigned Channel: C0
The arch connects the following nodes: I->N
The arch connects the following nodes: I->K
The arch connects the following nodes: I->F
Node: J
Assigned Channel: C1
The arch connects the following nodes: J->D
Node: K
Assigned Channel: C1
The arch connects the following nodes: I->K
Node: L
Assigned Channel: C1
The arch connects the following nodes: L->M
The arch connects the following nodes: H->L
Node: M
Assigned Channel: C0
The arch connects the following nodes: L->M
Node: N
Assigned Channel: C1
The arch connects the following nodes: H->N
The arch connects the following nodes: I->N
```

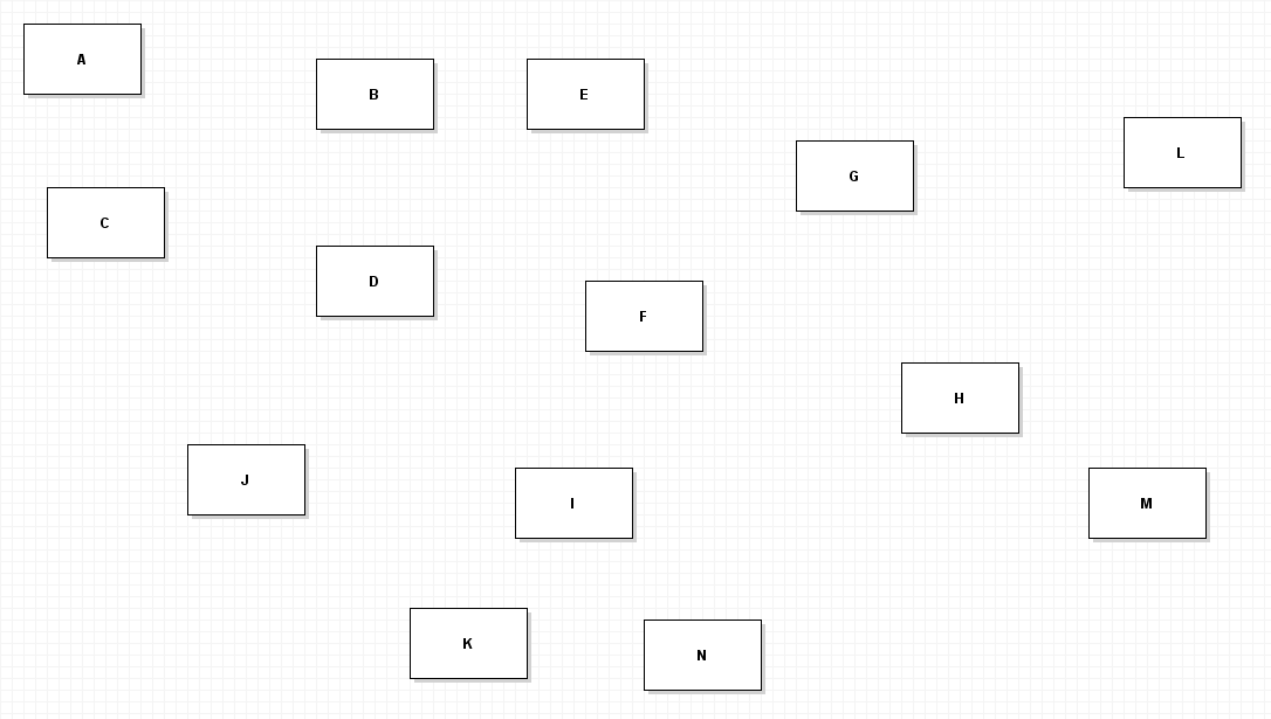
Dato che le risorse sono tante quante il rango del nodo con rango maggiore, queste **basteranno per coprire l'intero fabbisogno del grafo, senza eliminare alcun arco**. Verranno assegnate le risorse considerando sempre i nodi vicini a quello esaminato e, di conseguenza, verrà allocata una risorsa non utilizzata dai nodi adiacenti. In realtà, come si può osservare dall'immagine precedente, non sono nemmeno necessarie cinque risorse, infatti le ultime due risorse non vengono mai usate.
Dato che con tre risorse non otterremmo alcun cambiamento nella topologia del grafo, **consideriamo il caso in cui le risorse assegnate siano solo due**. In questo caso, **è necessaria l'eliminazione di alcuni archi**, la quale però avviene in maniera imprevedibile, dato che quando un nodo termina i canali disponibili, quest'ultimo gli viene assegnato casualmente dal Network.

```
Node: A
Assigned Channel: C0
The arch connects the following nodes: A->B
Node: B
Assigned Channel: C1
The arch connects the following nodes: A->B
The arch connects the following nodes: B->D
Node: C
Assigned Channel: C1
The arch connects the following nodes: C->D
Node: D
Assigned Channel: C0
The arch connects the following nodes: F->D
The arch connects the following nodes: B->D
The arch connects the following nodes: C->D
The arch connects the following nodes: E->D
The arch connects the following nodes: J->D
Node: E
Assigned Channel: C1
The arch connects the following nodes: E->D
Node: F
Assigned Channel: C1
The arch connects the following nodes: F->H
The arch connects the following nodes: I->F
The arch connects the following nodes: G->F
The arch connects the following nodes: F->D
Node: G
Assigned Channel: C0
The arch connects the following nodes: G->F
Node: H
Assigned Channel: C0
The arch connects the following nodes: H->L
The arch connects the following nodes: H->N
The arch connects the following nodes: F->H
Node: I
Assigned Channel: C0
The arch connects the following nodes: I->N
The arch connects the following nodes: I->K
The arch connects the following nodes: I->F
Node: J
Assigned Channel: C1
The arch connects the following nodes: J->D
Node: K
Assigned Channel: C1
The arch connects the following nodes: I->K
Node: L
Assigned Channel: C1
The arch connects the following nodes: L->M
The arch connects the following nodes: H->L
Node: M
Assigned Channel: C0
The arch connects the following nodes: L->M
Node: N
Assigned Channel: C1
The arch connects the following nodes: H->N
The arch connects the following nodes: I->N
```

In questo caso specifico, è possibile osservare come effettivamente alcuni **archi vengano eliminati**. Ad esempio l'arco che collega C e B viene eliminato, sia dal nodo C che dal nodo B, questo perché B e C hanno lo stesso canale assegnato, ovvero C1. Questo vale anche per l'arco che collega G e H. Di conseguenza il grafo risulta essere differente, come si può osservare qui in basso.



Effettuati questi due test, **passiamo al caso "speciale"**, ovvero il caso in cui il **network contiene una sola risorsa**. Razionalmente, si può intuire che, in tal modo, **a tutti i nodi verrà assegnato lo stesso canale**, e di conseguenza **nessun canale può essere adiacente all'altro**. Questo trasforma il grafo in una serie di nodi scollegati tra loro, come si può vedere dal grafo qui sotto.



Anche il programma stampa una serie di nodi, senza archi, con lo stesso canale assegnato

```
Node: A
Assigned Channel: C0
Node: B
Assigned Channel: C0
Node: C
Assigned Channel: C0
Node: D
Assigned Channel: C0
Node: E
Assigned Channel: C0
Node: F
Assigned Channel: C0
Node: G
Assigned Channel: C0
Node: H
Assigned Channel: C0
Node: I
Assigned Channel: C0
Node: J
Assigned Channel: C0
Node: K
Assigned Channel: C0
Node: L
Assigned Channel: C0
Node: M
Assigned Channel: C0
Node: N
Assigned Channel: C0
```