In [ ]:

```
Kandidatnummer: 10001 og 10146
```

In [1]:

```
from graph_utils.graph import Graph, nx, plt
from graph_utils.grid_graph import GridGraph
from graph_utils.constructed_graph import ConstructedGraph
from graph_utils.watts_strogatz import WattsStrogatz
from graph_utils.barabasi_albert import BarabasiAlbert
from graph_utils.real_network_graph import RealNetworkGraph
from graph_utils.vdes_graph import VDESGraph
import matplotlib
import numpy as np
import operator
import matplotlib.pyplot as plt
```

Del 1: Introduksjon

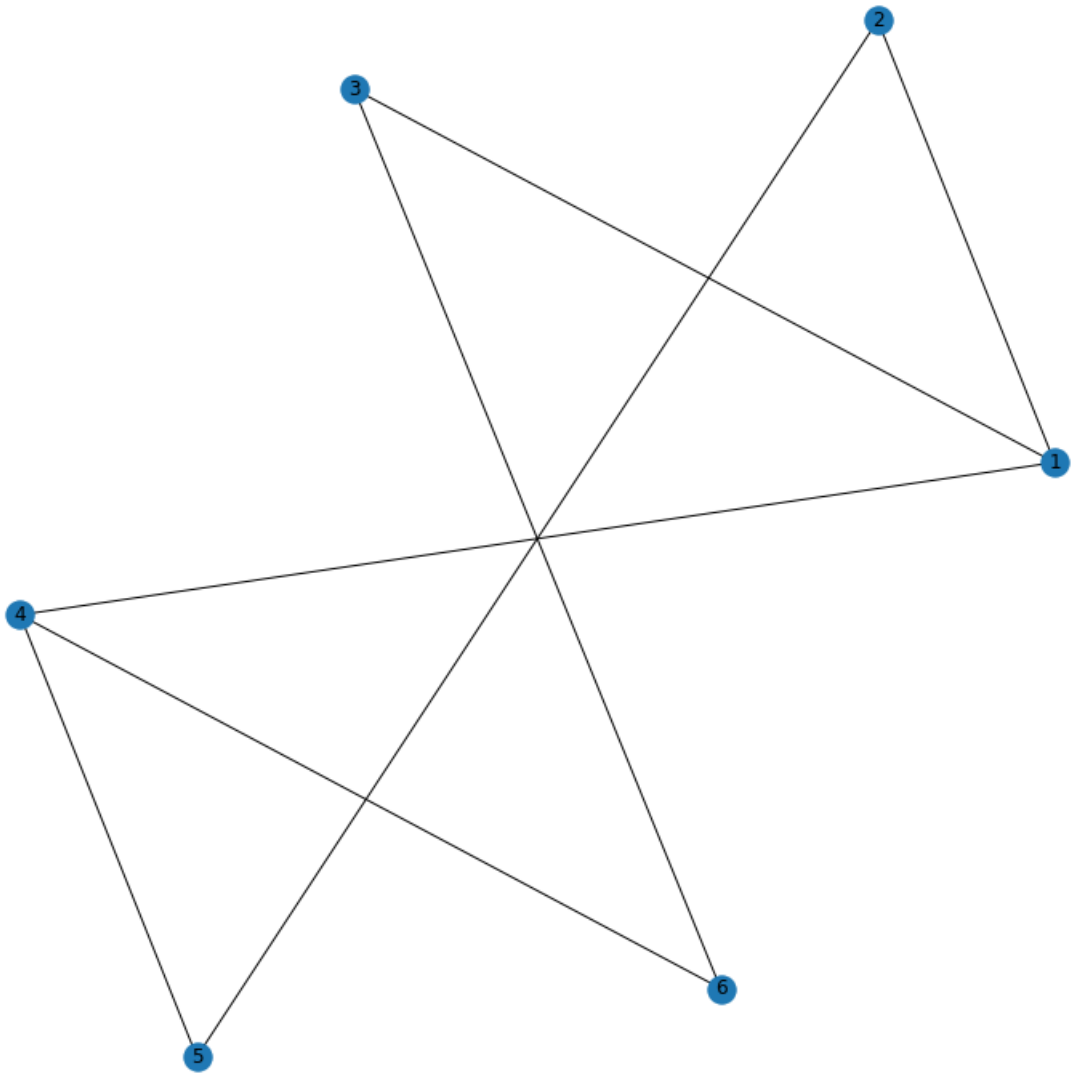
I denne delen skal vi bli kjent med hvordan man plotter grafer.

Oppgave 1.1

Lag en graf ved å først opprette et objekt av klassen `Graph`. Legg deretter til 5-10 noder ved å bruke metoden `add_node()` eller `add_nodes_from()`. Legg så til kanter mellom de nodene du ønsker ved å bruke `add_edge()` eller `add_edges_from()`. Bruk til slutt metoden `draw()` for å tegne grafen.

In [2]:

```
graf = Graph(seed=1234)
graf.add_node(1)
graf.add_nodes_from([2,3,4,5,6])
graf.add_edge(1,2)
graf.add_edges_from([(1,3),(1,4),(2,5),(4,6),(3,6),(4,5)])
graf.draw()
```



Oppgave 1.2

Man kan finne antall noder ved å bruke metoden `number_of_nodes()`. Prøv den ut på grafobjektet du opprettet i forrige oppgave og print resultatet.

In [3]:

```
print(graf.number_of_nodes())
```

```
6
```

Oppgave 1.3

På samme måte som med noder kan man finne antallet kanter i grafen med metoden `number_of_edges()`. Bruk den på grafen fra oppgave 1.1 til å finne antall kanter i grafen og print resultatet.

In [4]:

```
print(graf.number_of_edges())
```

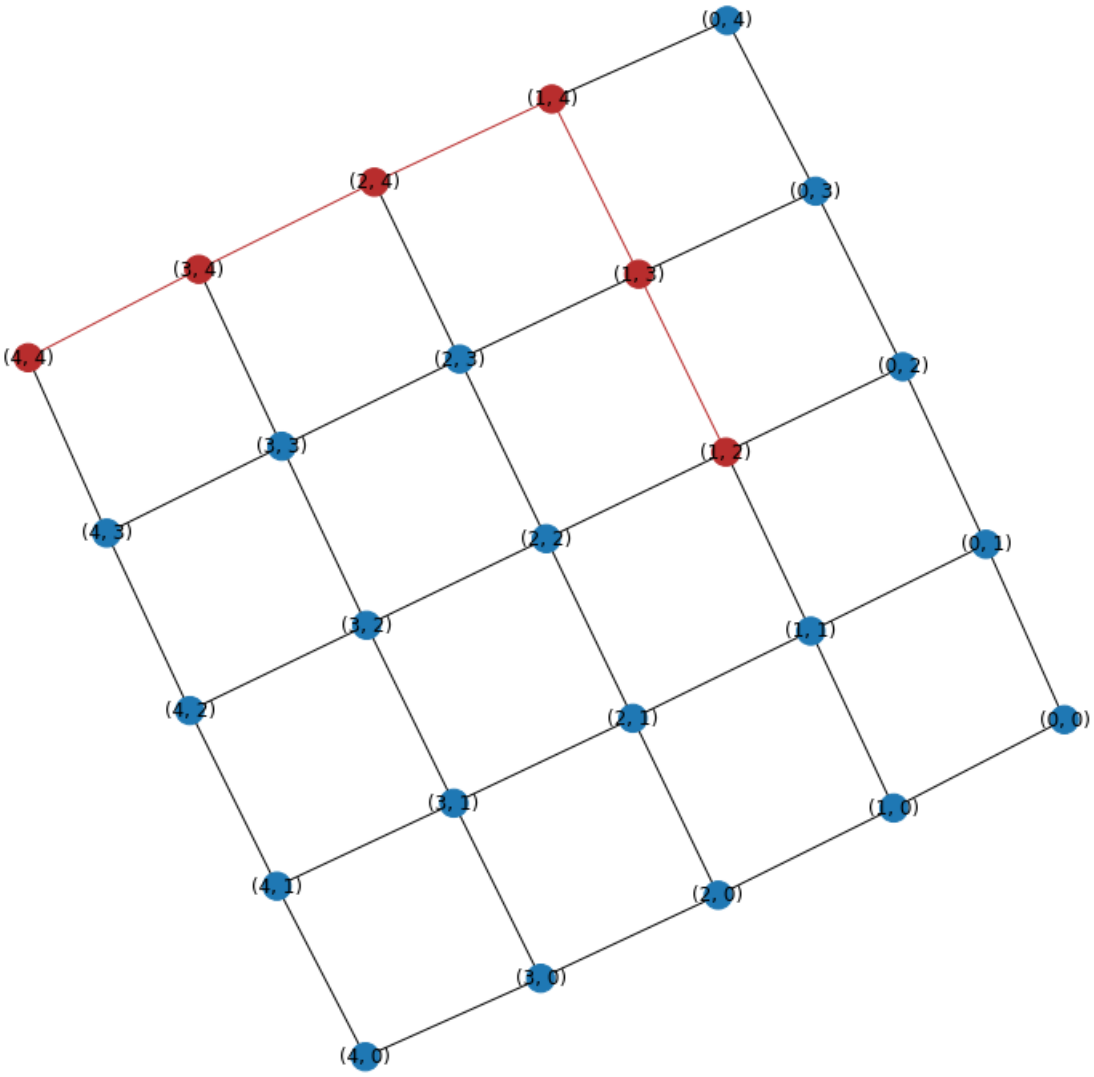
```
7
```

Oppgave 1.4

Opprett et `GridGraph` -objekt med 5 x 5 noder. Tegn korteste vei mellom node (4, 4) og (1, 2) med metoden `mark_shortest_path()`.

In [10]:

```
grid = GridGraph(5,5)
grid.mark_shortest_path((4, 4),(1,2))
```

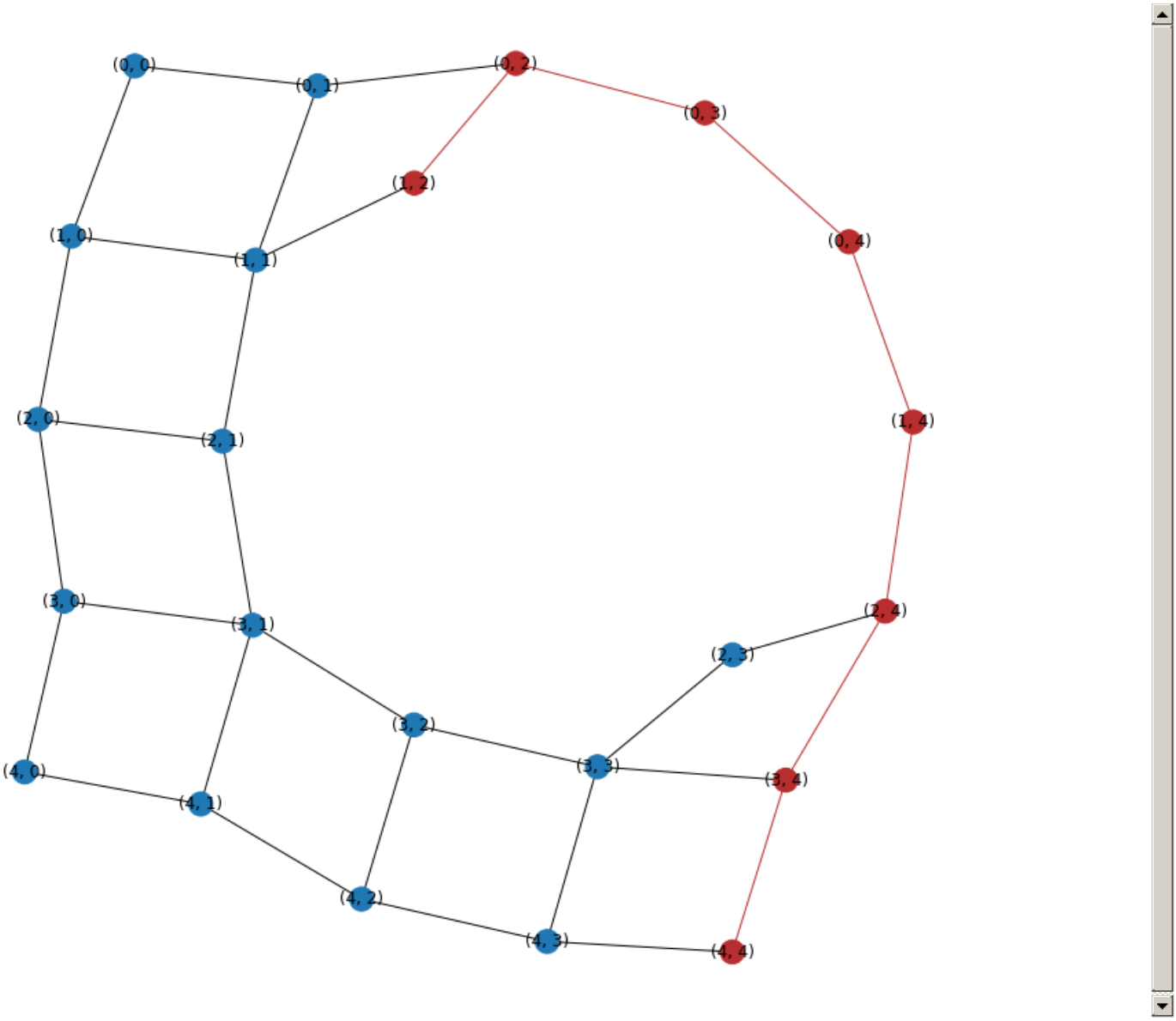


Oppgave 1.5

Vi skal forsette å bruke grafen fra forrige oppgave. Fjern node (1, 3) og (2, 2) ved å bruke metoden `remove_node()` . Finn nå korteste vei mellom node (4, 4) og (1, 2) og tegn resultatet. (Merk at nodene ikke nødvendigvis blir tegnet på samme steder som i forrige oppgave, men den tegner en isomorf graf)

In [11]:

```
grid.remove_node((1,3))
grid.remove_node((2,2))
grid.mark_shortest_path((4, 4),(1,2))
```

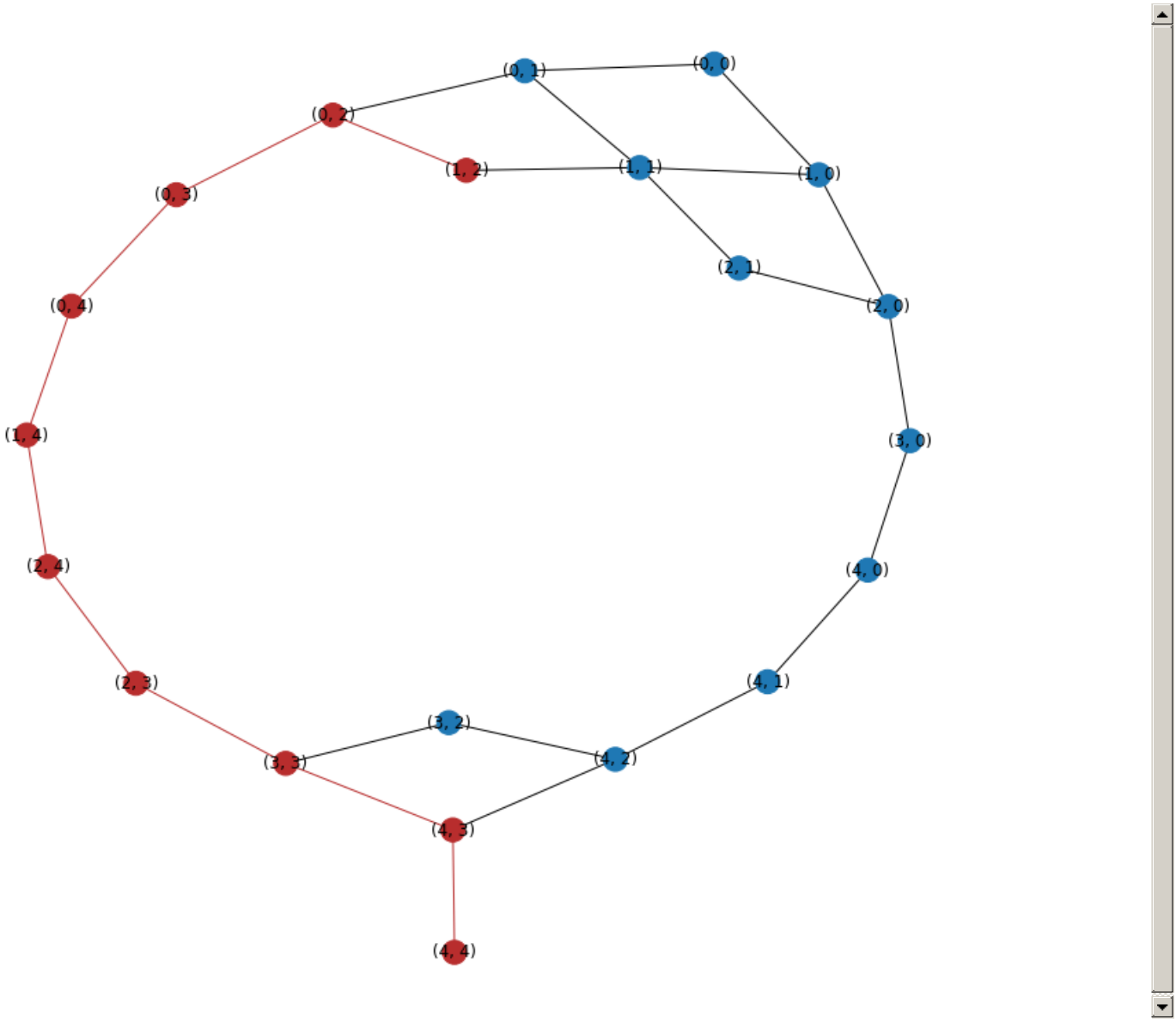


Oppgave 1.6

Hvor mange noder må minst slettes for at korteste vei mellom node (4, 4) og (1, 2) skal bli lenger enn den var i oppgave 1.5, men at det fremdeles finnes en vei?

In [12]:

```
grid.remove_node((3,1))
grid.remove_node((3,4))
grid.mark_shortest_path((4, 4),(1,2))
```



Det må slettes minst to noder. Per nå finnes det to veier som har vekt 7. Dersom du fjerner en node fra hver av de veiene, har ny korteste vei vekt 9
Hvor mange noder må slettes for at det ikke skal eksistere noen vei mellom node (4, 4) og (1, 2), med utgangspunkt i grafen slik den er i oppgave 1.5?

In [8]:

```
# Kodecelle hvis du vil teste
```

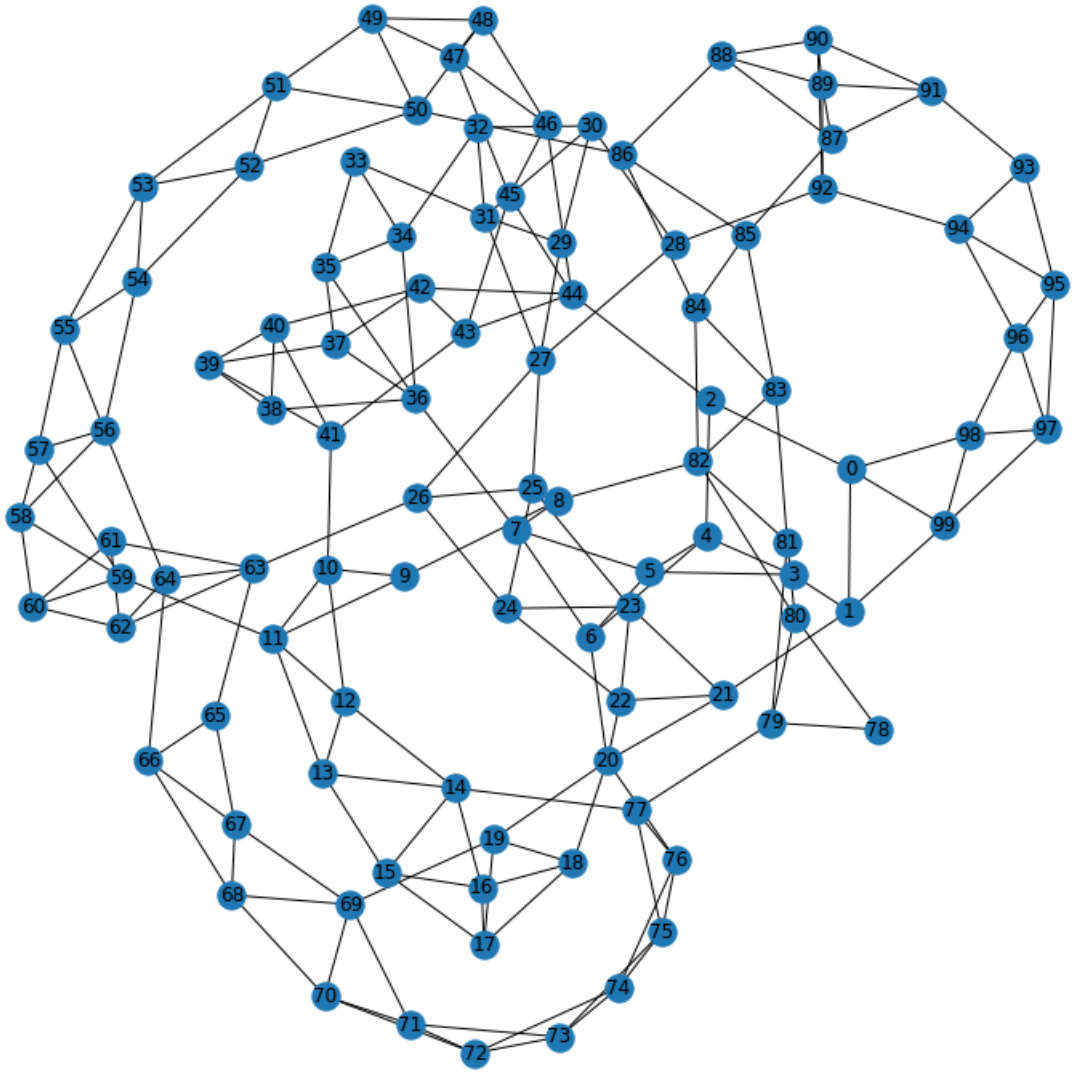
Det holder å slette to noder - enten (4,3) og (3,4) eller (1,1) og (0.2)

Oppgave 1.7

Opprett et `WattsStrogatz` -objekt med parametre n=100, k=4 og p=0.1. Tegn så grafen, og finn korteste vei mellom node 53 og 75.

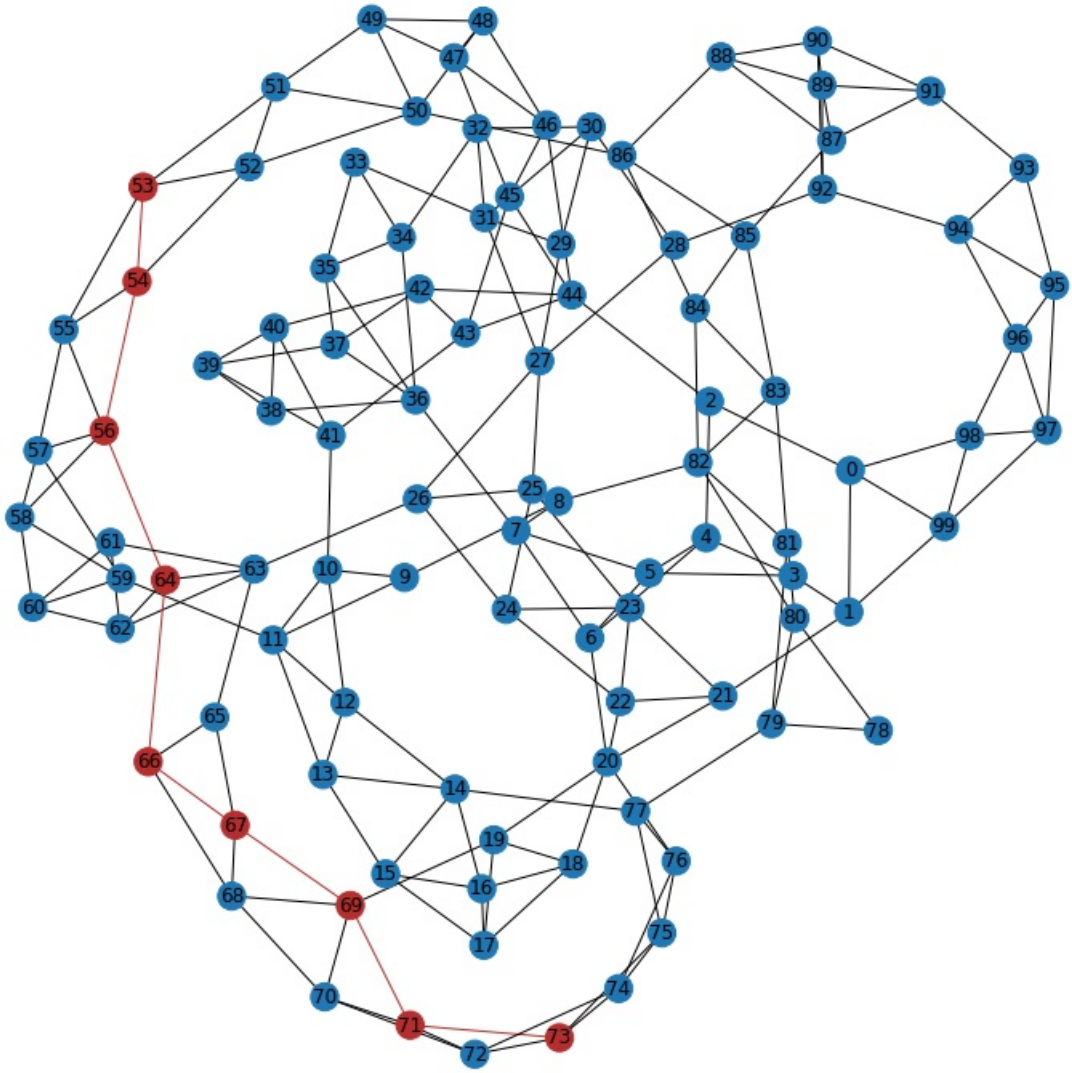
In [13]:

```
ws1 = WattsStrogatz(n=100, k=4, p=0.1, seed=69)
ws1.draw()
```



In [14]:

```
ws1.mark_shortest_path(53,73)
```



Del 2: Strukturanalyse

Oppgave 2.1

Lag følgende 4 grafer, alle med 100 noder:

- Graf 1: En Barabasi Albert graf med parameter m=1
- Graf 2: En Barabasi Albert graf med parameter m=2
- Graf 3: En Watts Strogatz graf med parametre k=2 og p=0.1
- Graf 4: En Watts Strogatz graf med parametre k=4 og p=0.1

Du trenger ikke tegne grafene, bare opprett et objekt for hver graf. Det kan likevel være lurt å tegne grafen for din egen del, så du vet hvordan den ser ut.

In [15]:

```
ba1 = BarabasiAlbert(n=100, m=1)
ba2 = BarabasiAlbert(n=100, m=2, seed=ba1.seed)
ws2 = WattsStrogatz(n=100, k=2, p=0.1, seed=69)
ws3 = ws1
```

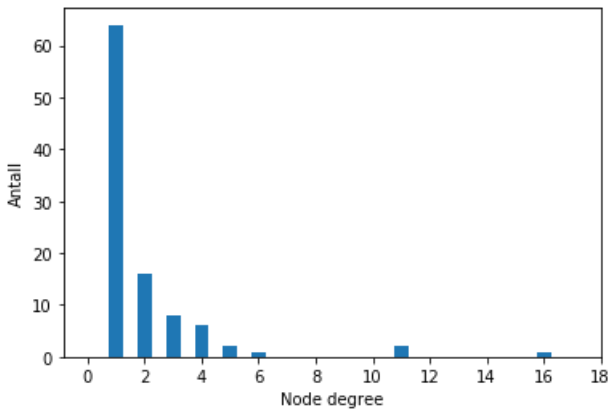
Oppgave 2.2

For hver av de fire grafene: lag et histogram som viser degree distribution.

- Bruk metoden `histogram()` for å gjøre dette.
- Kommenter så kort hva du diagrammet illustrerer.

In [16]:

```
ba1.histogram()
```

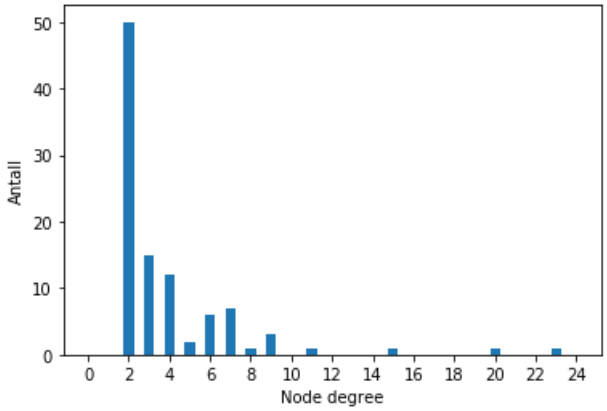


Out[16]:

```
[0, 64, 16, 8, 6, 2, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1]
```

In [17]:

```
ba2.histogram()
```

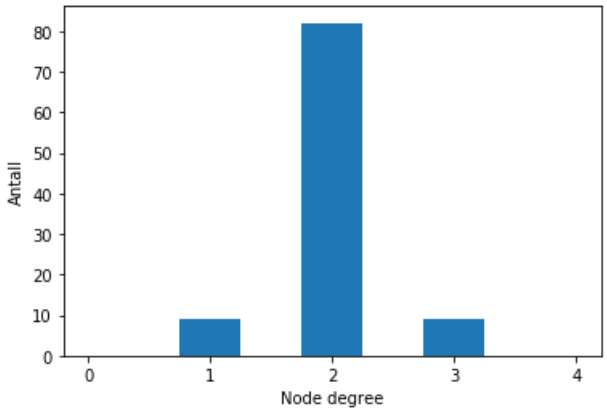


Out[17]:

```
[0, 0, 50, 15, 12, 2, 6, 7, 1, 3, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1]
```

In [18]:

```
ws2.histogram()
```

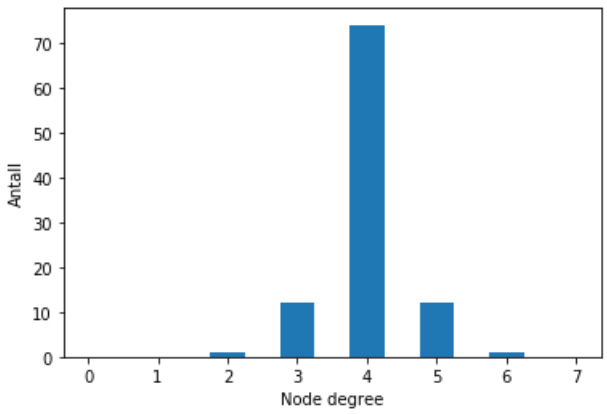


Out[18]:

```
[0, 9, 82, 9]
```

In [19]:

```
ws3.histogram()
```



Out[19]:

```
[0, 0, 1, 12, 74, 12, 1]
```

Oppgave 2.3

Kommenter resultatene med hensyn på sårbarheter for hver av de 4 grafene i forrige oppgave.

For ba1, så er det åpenbart at hele grafen er sårbar dersom de "innerste" nodene blir svikter. Det er mange avhengigheter som er knyttet til de innerste. Det er kun løvnodene som kan fjernes uten at det setter andre noder ut av spill. Det er node degree som forteller oss hvor mange kanter noden har. ba2 har litt flere noder med høy node degree, som gjør hele systemet (som grafen skal illustrere) mindre sårbart. Den er mindre "hierarkisk", som øker "attack resistance". Til forskjell fra ba1 er den mindre avhengig av et fåtall noder ws2 er samme som ba1: mister tilgang til resten av treet når en node svikter. Forskjellen her er at grafen ikke har like mange løvnoder, dvs. svikt av en ikke-løvnode vil kutte av deler av grafen. ws3 har nesten alle noder med degree 3 eller høyere. Fordelen med dette er at hvis en node svikter, er det lite sjans for at hele/deler av grafen svikter. ws3 har jevnere fordelt kanter. Dvs. at hvis en tilfeldig node svikter blir ikke den potensielle omveien så stor.

Oppgave 2.4

Hvordan endrer degree distribution seg for en Barabasi Albert graf seg når m øker fra 1 til 2? Forklar hvorfor.

1) Det sprer seg mer ut. Flere noder får høyere antall kanter. 2) Typetallet (og minsteverdien) byttes fra 1 til 2 Begge observasjonene forklares logisk ved at m er "antall kanter som nye noder skal få til en vilkårlig eksisterende node"

Oppgave 2.5

Hvordan endrer degree distribution seg for en Watts Strogats graf seg når k øker fra 2 til 4? Forklar hvorfor.

Som beskrevet tidligere: mange noder med 2 kanter resulterer i en svak graf ettersom de ender i en løvnode. Det er dermed ingen loop og hvis en node svikter kan en hel gren bli utilgjengelig. Når k blir 4 blir grafen derimot mye mer robust ettersom det oftere er mulig å finne en vei fra a til b (og grafen blir bygd opp slik at omveien ikke blir veldig lang).

Oppgave 2.6

Hvordan endrer degree distribution seg for en Watts Strogats graf dersom p øker? Test det ut og forklar hvorfor.

In [16]:

```
# Eventuell kode her
```

P er sannsynligheten for at en ny kant skal ta en annen kants plass. degre distrubution er det samme. Men hvis P = 0 har alle noder automatisk kanter = k

Oppgave 2.7

Hva sier degree distribution om nettverket?

Degree annoterer hvor mange kanter en gitt node har til andre noder. Degree distrubution er den såkalte "probibility distribution" av disse nodene over hele nettverket. Vi kan definere degree distribution slik: for en gitt n noder og nk av dem har k kanter , så er degree distribution for grafen: P(k)= nk/n.

Oppgave 2.8

- Hent ut de 5 viktigste nodene i forhold til closeness centrality for graf 1 og graf 2 og finn deres verdi.
- Marker så nodene på grafene og kommenter viktigheten av de 5.
- Sammenlign mellom grafene hvor disse nodene spiller størst rolle.

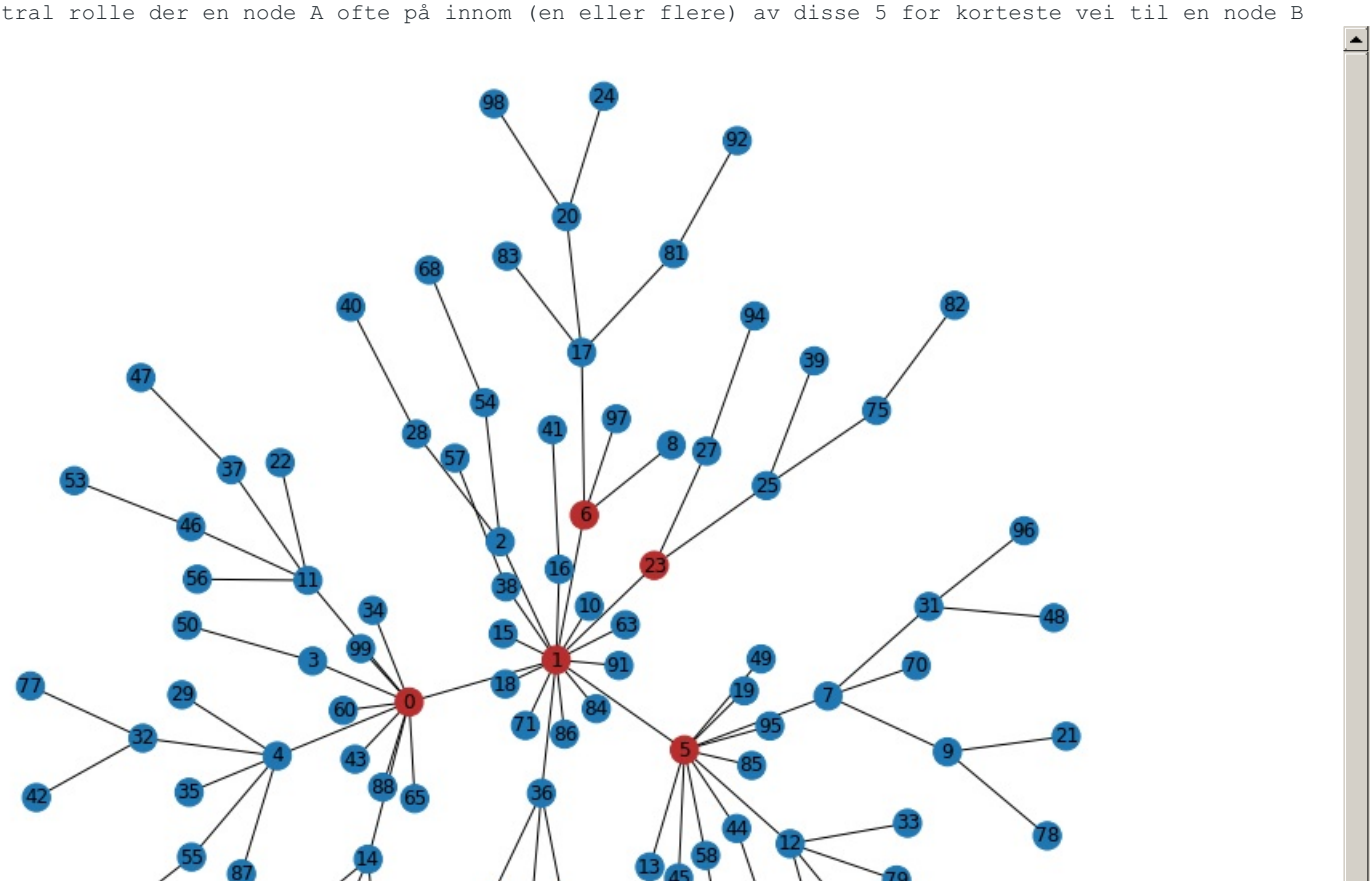
In [20]:

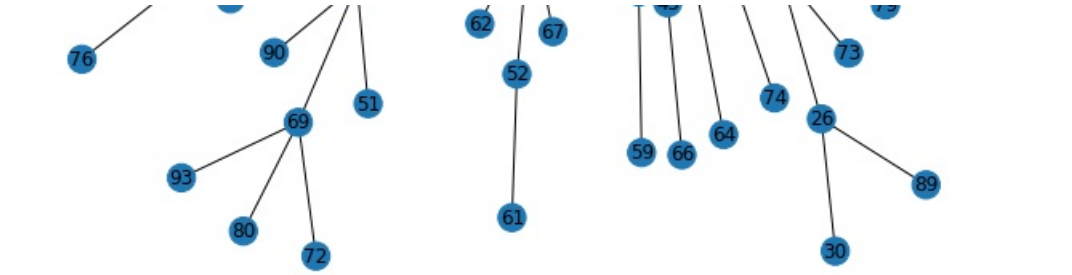
```
# Assume the task is about the graphs in task 2.1
myGraf1 = ba1.closeness_centrality()
myGraf2 = ba2.closeness_centrality()
graf1High = []
```

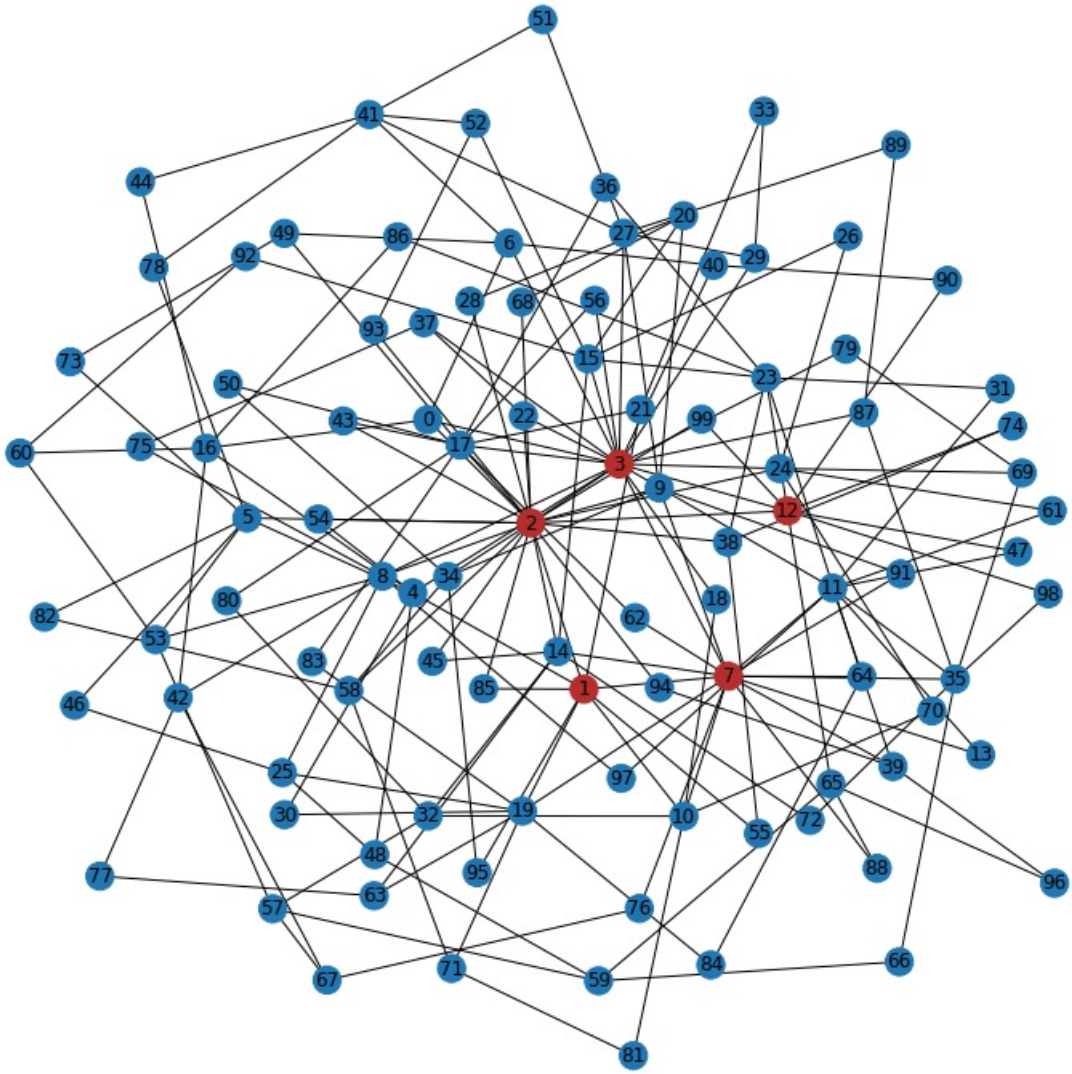
```
graft2High = []

print("\n-- graf 1 --\nNode: Verdi")
# graf 1
for x in range(0,5):
    k = max(myGraf1.items(), key=operator.itemgetter(1))[0]
    graft1High.append(k)
    v = max(myGraf1.items(), key=operator.itemgetter(1))[1]
    print (k," : ", v)
    del myGraf1[k]

print("\n-- graf 2 --\nNode: Verdi")
# graf 2
for x in range(0,5):
    k = max(myGraf2.items(), key=operator.itemgetter(1))[0]
    graft2High.append(k)
    v = max(myGraf2.items(), key=operator.itemgetter(1))[1]
    print (k," : ", v)
    del myGraf2[k]

kommentar = "\ncloseness_centrality er summen av kortest vei fra en node til alle andre noder, dvs. at  d
print(kommentar)



ba1.mark_nodes(graft1High)
ba2.mark_nodes(graft2High)
```

```
-- graf 1 --
Node: Verdi
1  :  0.3944223107569721
0  :  0.34494773519163763
5  :  0.33559322033898303
6  :  0.2990936555891239
23  :  0.29376854599406527

-- graf 2 --
Node: Verdi
3  :  0.49748743718592964
2  :  0.495
7  :  0.44
1  :  0.4125
12  :  0.4074074074074074
```

closeness_centrality er summen av kortest vei fra en node til alle andre noder, dvs. at de spiller en sentral rolle der en node A ofte på innom (en eller flere) av disse 5 for korteste vei til en node B





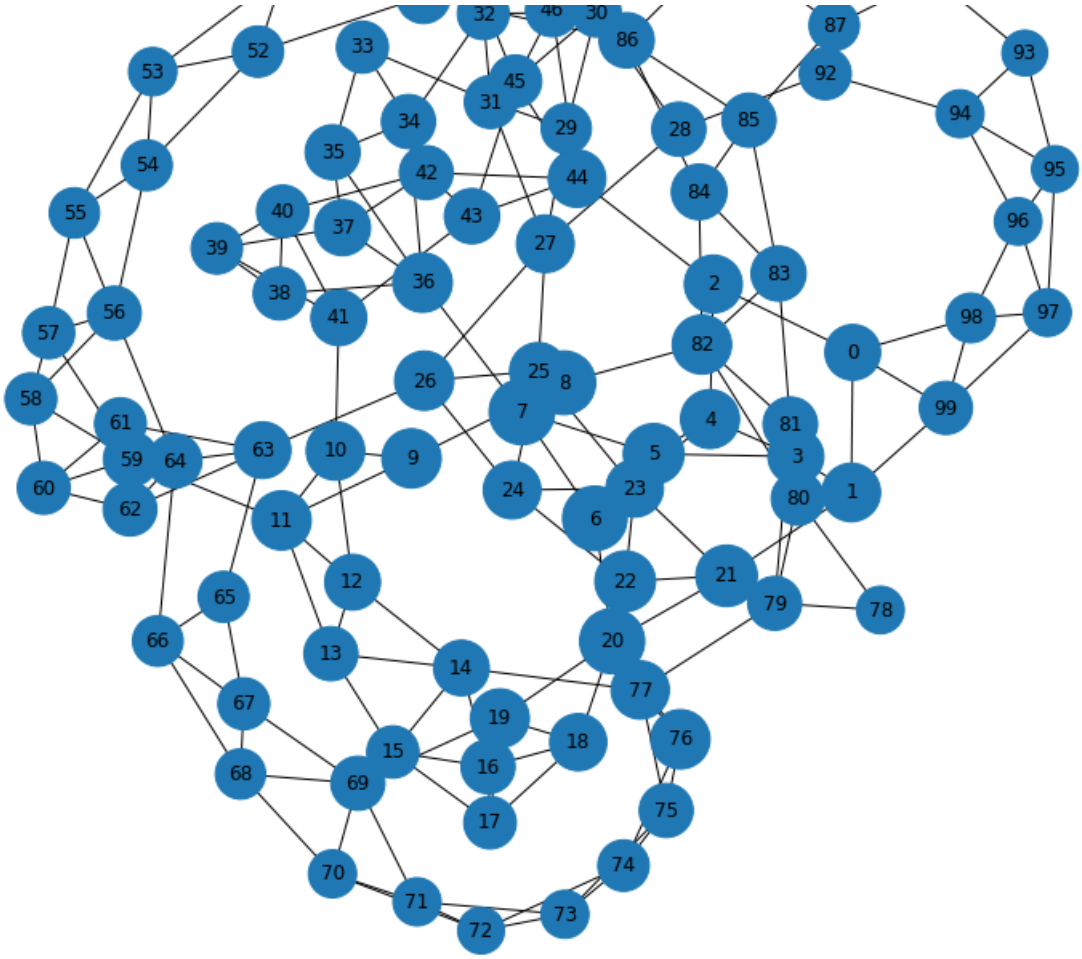


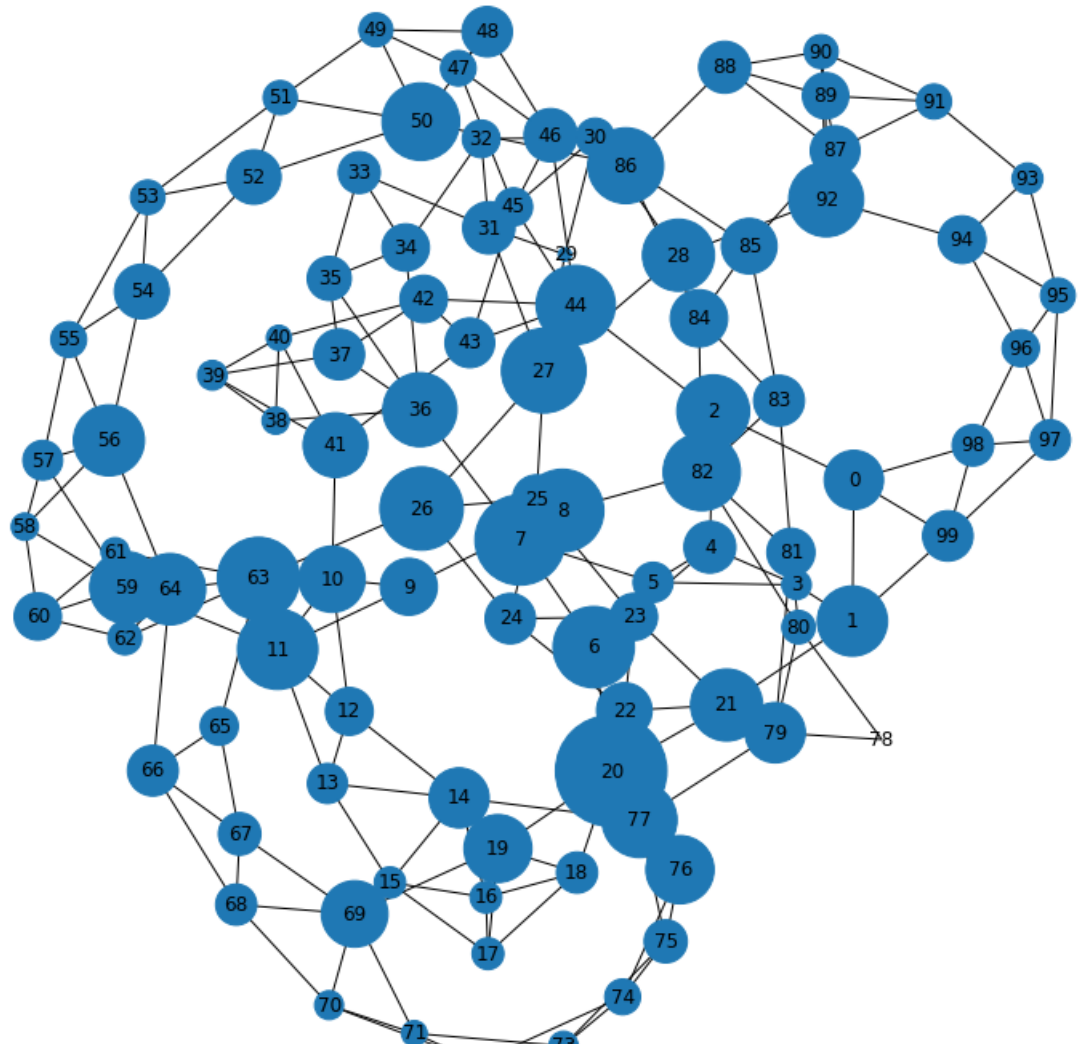
Oppgave 2.9

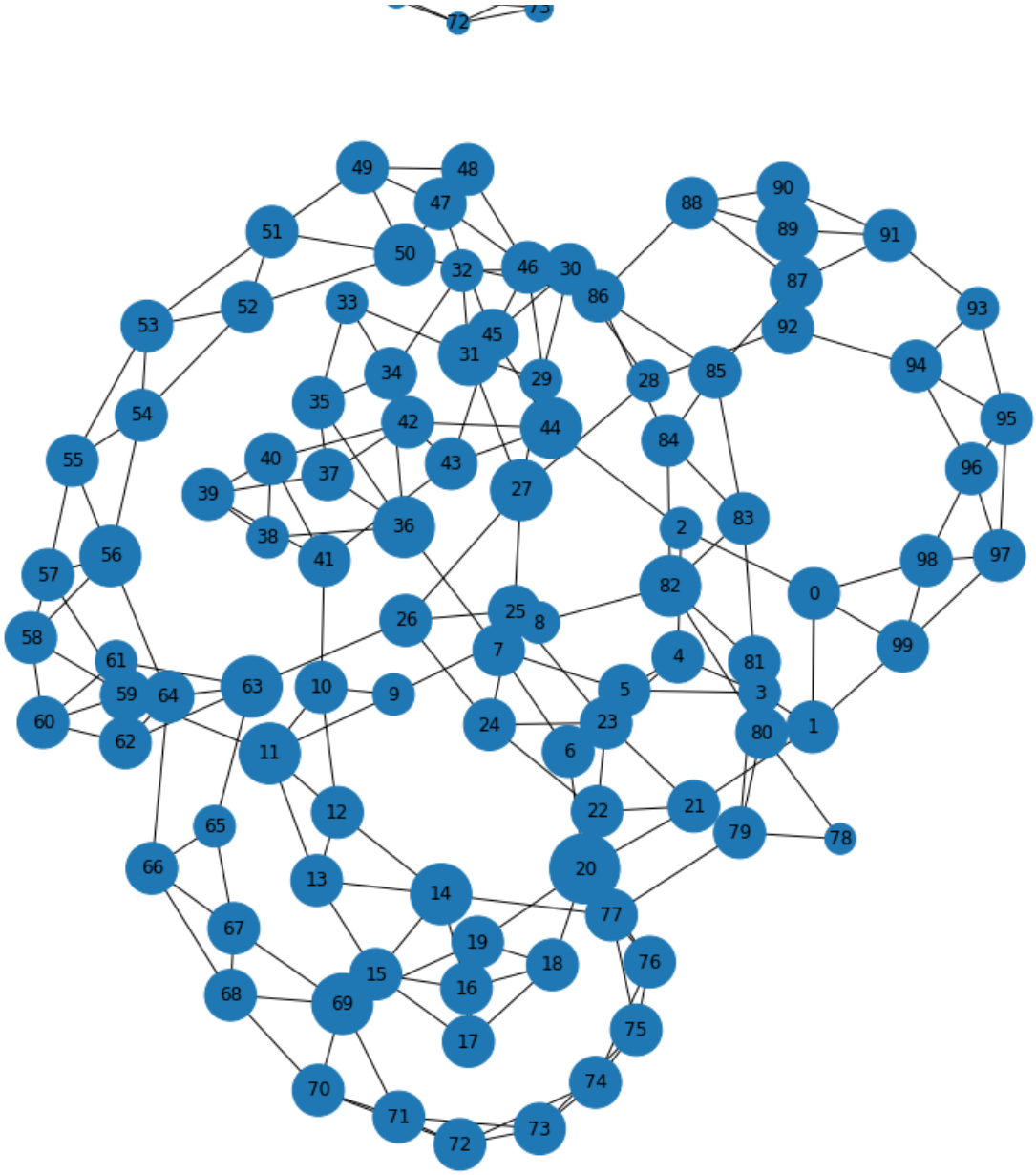
For graf 4: Identifiser noder som har høy verdi av en indeks, men lav av en annen indeks. Tips: tegning illustreringen av centrality indexene for å se det ut ifra grafen. Hva blir konsekvensene dersom disse nodene feiler?

In [24]:

```
ws3.draw_closeness_centrality(1137)
ws3.draw_betweenness_centrality(1137)
ws3.draw_degree_centrality(1137)

# Node 19 og 7 har f.eks ganske høy betweenness og lav degree, som betyr at det er mange av
# grafens "korteste veier" som går via 19 og 7, men at de ikke har så mange kanter tilknyttet andre node
# følgelig kan vi påstå at nettverket vil få ganske redusert ytelse dersom en av disse feiler
```


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99



Oppgave 2.10

Konstruer et nettverk bestående av mellom 9 og 15 noder med en node som har høyest closeness centrality, og en av de laveste degree centrality verdiene. Tips: Siden grafen ikke er så stor kan det være lettere å hardkode noder og edges enn å generere de ved hjelp av en `for` loop.

In [25]:

```
networkGraph = Graph(seed=69420)
networkGraph.add_nodes_from([1,2,3,4,5,6,7,8,9])
networkGraph.add_edges_from([(5,4),(5,6),(4,3),(3,2),(2,1),(1,4),(6,7),(7,8),(8,9),(9,6)])

closeNetwork = networkGraph.closeness_centrality()
degreeNetwork = networkGraph.degree_centrality()

k1 = max(closeNetwork.items(), key=operator.itemgetter(1))[0]
v1 = max(closeNetwork.items(), key=operator.itemgetter(1))[1]
v2 = min(degreeNetwork.items(), key=operator.itemgetter(1))[1]

def findMinDegree():
    minDegree = {}
    for x in degreeNetwork:
        if degreeNetwork[x] == v2:
            minDegree[x] = degreeNetwork[x]
    print("lowest degree:\n",minDegree)


findMinDegree()
print("\nCloseness\n",k1,":",v1)
```

```
print("\n- Vi kan se at node 5 har størst closeness centrality, og delt laveste degree centrality verdier
```

```
lowest degree:
 {1: 0.25, 2: 0.25, 3: 0.25, 5: 0.25, 7: 0.25, 8: 0.25, 9: 0.25}

Closeness
 5 : 0.5

- Vi kan se at node 5 har størst closeness centrality, og delt laveste degree centrality verdiene -
```

Del 3: Analyse av Reelle nettverk

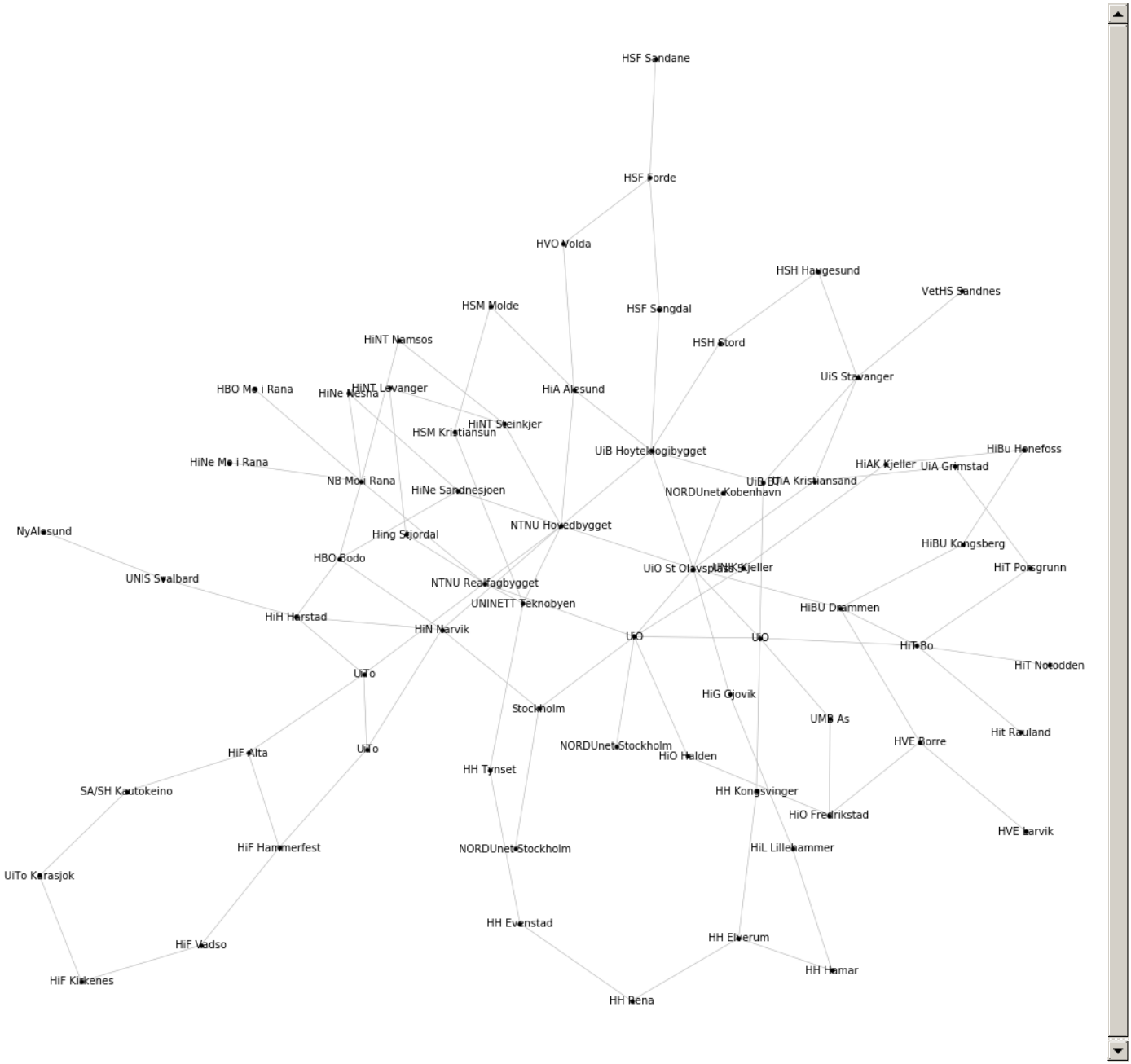
Her skal vi analysere og diskutere reelle/ simulerte nettverk.

Oppgave 3.1

I denne oppgaven skal vi analysere Uninett sitt nettverk slik det var i 2011. Klassen `RealNetworkGraph` tar inn en url av en fil med filtypen *.graphml*. Lag et objekt for kjernenettet i Norge og tegn det. Filene som kan analyseres finnes på nettsiden Topology Zoo.

In [26]:

```
# Assuming we have to ctrl + F for uninett and copy link location for the 2011 result (graphml).
# Yielding this link: http://www.topology-zoo.org/files/Uninett2011.graphml
core_network = RealNetworkGraph("http://www.topology-zoo.org/files/Uninett2011.graphml")
core_network.draw()
```

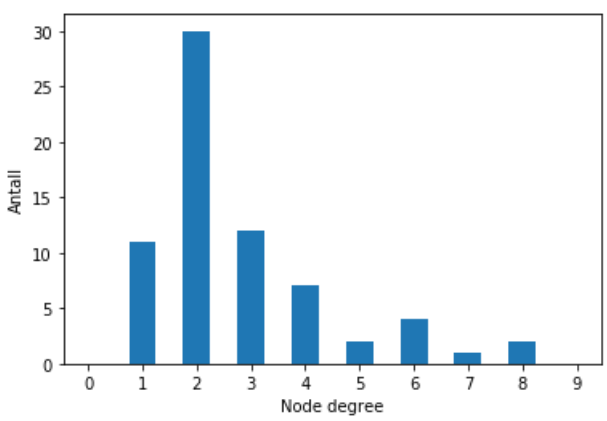


Oppgave 3.2

Plot et histogram over degree distribution for nettverket til Uninett

In [21]:

```
core_network.histogram()
```



Out[21]:

```
[0, 11, 30, 12, 7, 2, 4, 1, 2]
```

Oppgave 3.3

For hver av de tre centrality indeksene, print gjennomsnittlig centrality i nettverket til Uninett.

In [22]:

```
# Declaring the three centrality indexes mentioned in the lecure. Returns key/value pairs for each node
cn_close = core_network.closeness_centrality()
cn_between = core_network.betweenness_centrality()
cn_degree = core_network.degree_centrality()

def average(g):
    total = 0
    # Loop through nodes, find avg of the values
    for key in g:
        total += g[key]
    return total/len(g)

print(f"Average closeness: {round(average(cn_close), 3)}")
print(f"Average betweenness: {round(average(cn_between), 3)}")
print(f"Average degree: {round(average(cn_degree), 3)}")
```

```
Average closeness: 0.244
Average betweenness: 0.049
Average degree: 0.041
```

Oppgave 3.4

Diskuter resultatene fra oppgave 3.2 og 3.3. Er nettverket robust? Forklar.
Hvilket av nettverkene fra del 2 ligner Uninett sitt mest på?

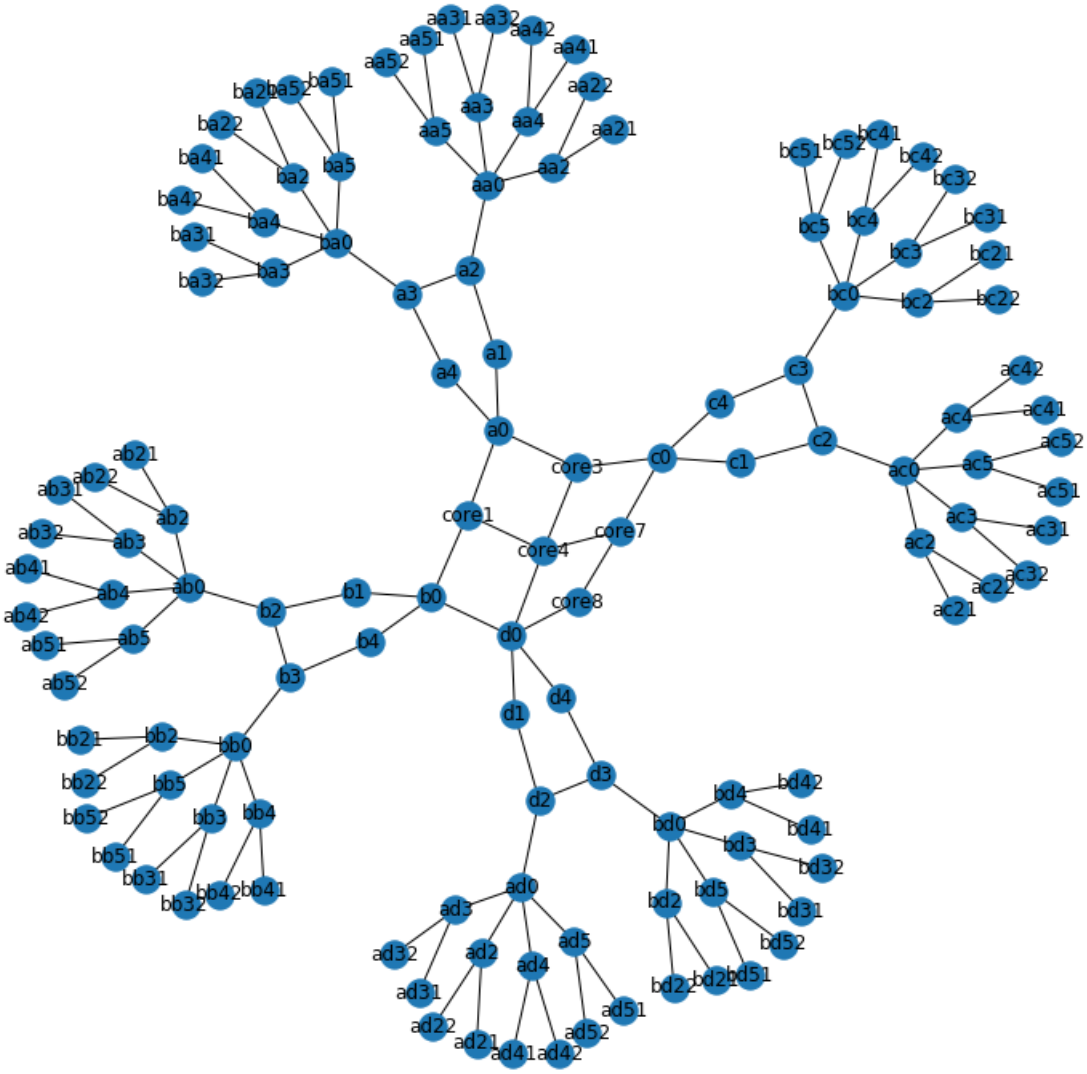
Vi ser at de fleste nodene har grad i området 1-4, hvor 2 er typetallet. Som nevnt i forelesning er det større risiko tilknyttet sikkerheten og robustheten til et nettverk dersom et angrep klarer å "ta ut" de nodene med høyest node-degree. Jo høyere gjennomsnittlig centrality indeks en node har, desto "viktigere" kan de sies å være for nettverket. Dermed vil en graf med lav score på avg degree (hvorav det er et fåtall noder som har mye høyere grad enn de andre) være mindre robust. Følgelig vil jeg påstå at dette nettverket er nokså robust, men med klare svakheter dersom man klarer å ta u f.eks de 10 nodene med høyest node degree. (Jeg har ikke så mye å sammenligne med her, og "hvor robust noe er" trenger litt kontekst - jeg føler i hvert fall ikke at det er noe åpenbart svar på dette uten noe videre kontekst) Den er nok likest ba2 (ganske lik ba1 også - men det har flertall node med node degree >= 2).

Oppgave 3.5

Constructed graph simulerer et reelt netverk. Den består av et kjernenett med en grid struktur, et regionalnett og et tettbebyggd aksessnett. Bruk klassen `ConstructedGraph` og tegn så grafen.

In [27]:

```
constructedGraph = ConstructedGraph()
constructedGraph.draw()
```



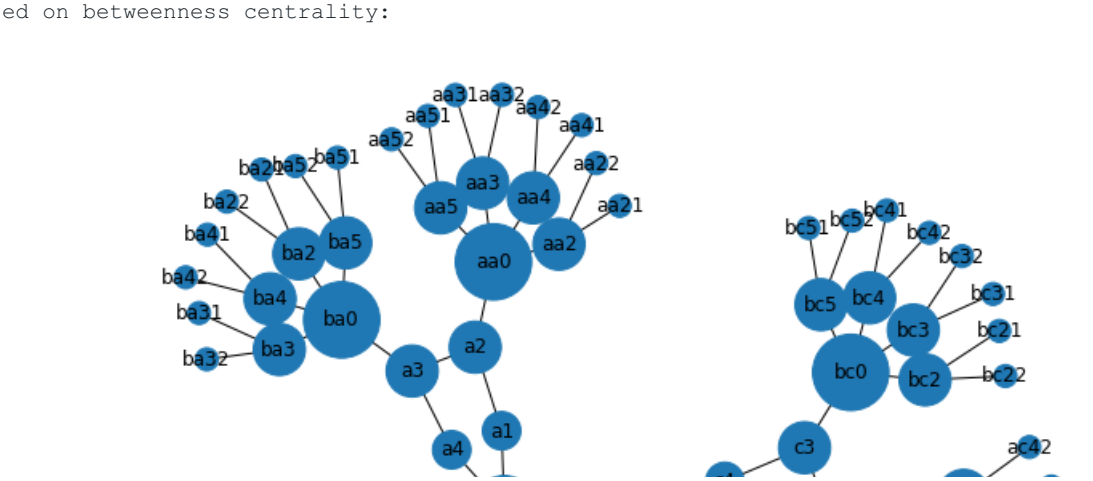
Oppgave 3.6

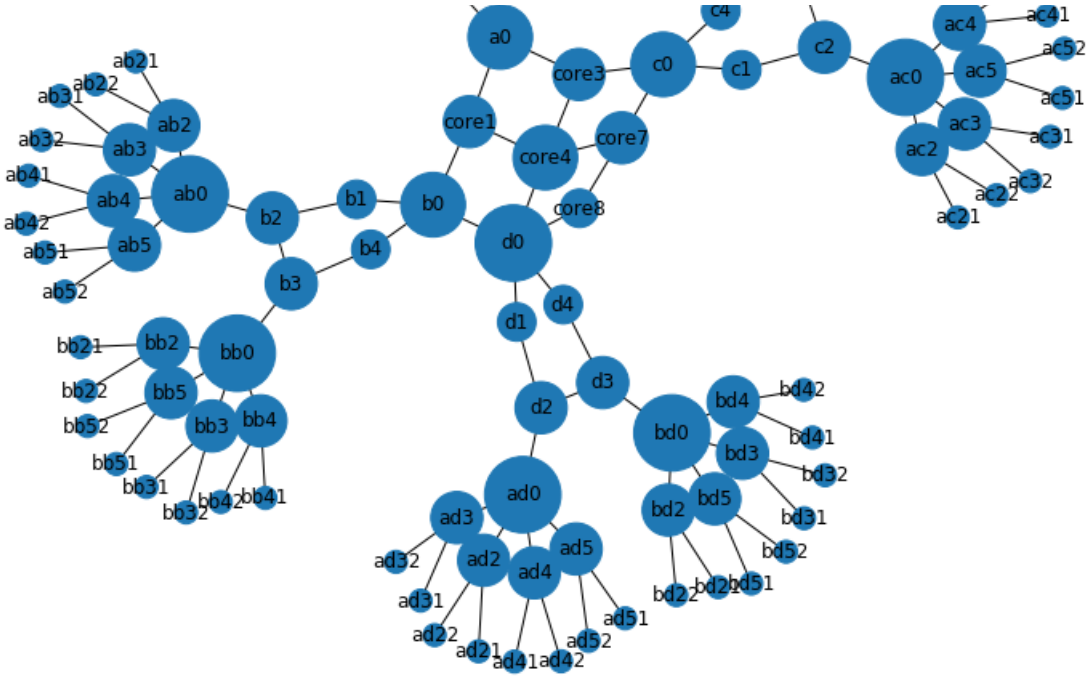
Illustrer på grafen verdien av de tre centrality indeksene. Tips: bruk de metodene som er ferdiglagde.

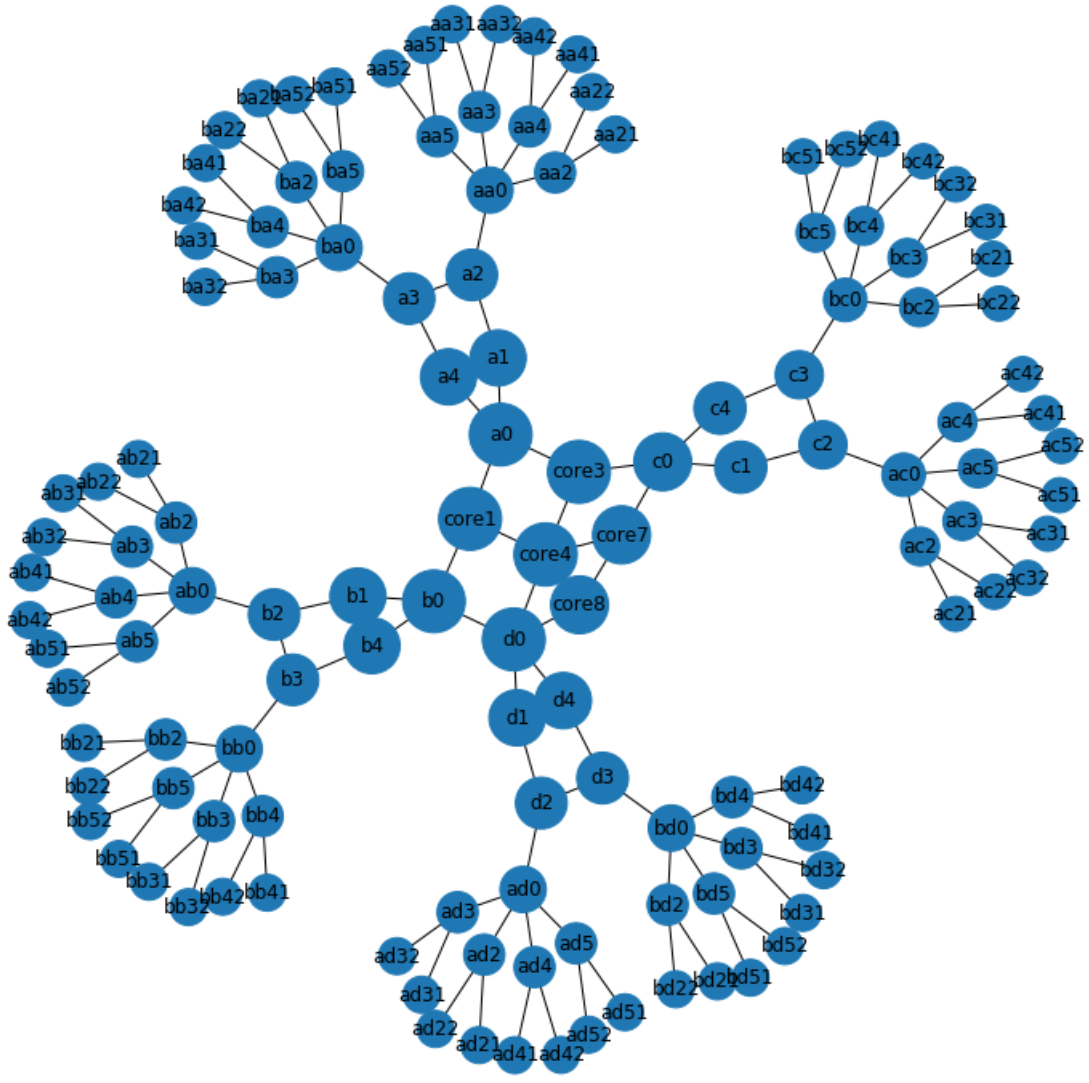
In [28]:

```
print("Based on degree centrality: ")
constructedGraph.draw_degree_centrality(avg_size=700)
print("Based on closeness centrality: ")
constructedGraph.draw_closeness_centrality(avg_size=700)
print("Based on betweenness centrality: ")
constructedGraph.draw_betweenness_centrality(avg_size=700)
# Det viser seg at det hender maskinen printer alt før grafene blir tegnet, men da ser man i hvert fall .
```

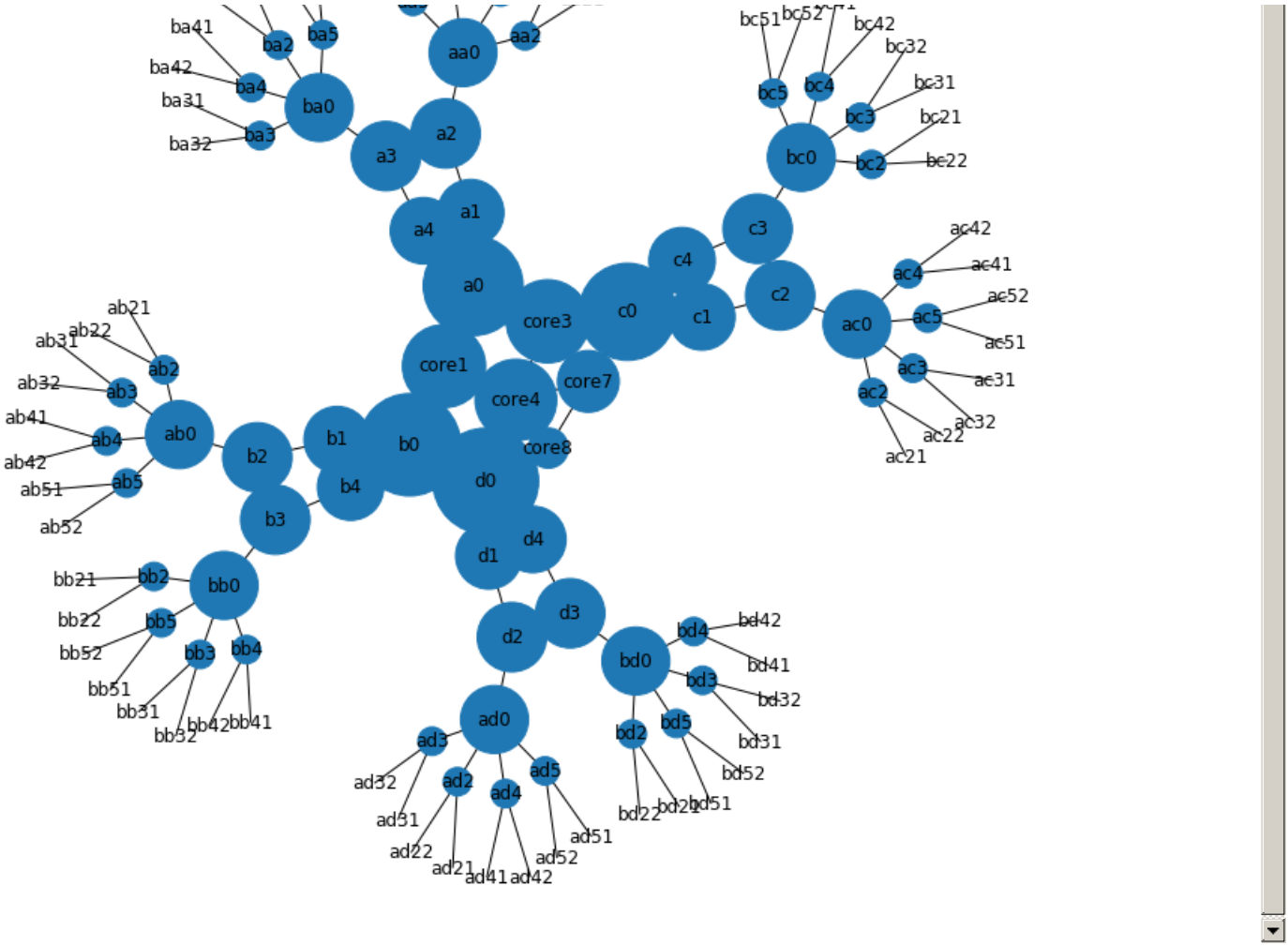
```
Based on degree centrality:
Based on closeness centrality:
Based on betweenness centrality:
```




a0
core3
c0
c1
c2
c4
ac4
ac41
ac52
ac5
ac0
ac51
ac3
ac31
ac2
ac32
ac22
ac21
ab21
ab22
ab31
ab2
ab32
ab3
ab41
ab4
ab0
ab42
ab5
ab51
ab52
b2
b1
b0
b4
b3
core1
core4
core7
core8
d0
d1
d4
d2
d3
bb21
bb2
bb0
bb22
bb5
bb52
bb3
bb4
bb51
bb31
bb32
bb42
bb41
ad0
ad3
ad2
ad4
ad5
ad32
ad31
ad22
ad21
ad41
ad42
ad51
bd0
bd4
bd42
bd41
bd3
bd32
bd2
bd5
bd31
bd52
bd22
bd21
bd51


aa31
aa32
aa42
aa51
aa41
aa52
aa22
aa3
aa4
aa5
aa21
aa0
aa2
ba21
ba52
ba51
ba22
ba41
ba2
ba5
ba42
ba4
ba31
ba0
ba32
ba3
a3
a2
a4
a1
a0
bc51
bc52
bc41
bc42
bc32
bc5
bc4
bc31
bc3
bc21
bc0
bc2
bc22
c3
c4
c2
c0
c1
ac42
ac4
ac41
ac52
ac0
ac5
ac51
ac3
ac31
ac2
ac32
ac22
ac21
core3
core1
core4
core7
core8
ab21
ab22
ab31
ab2
ab32
ab3
ab41
ab4
ab0
ab42
ab5
ab51
ab52
b2
b1
b0
b4
b3
d0
d1
d4
d2
d3
bb21
bb2
bb0
bb22
bb5
bb52
bb3
bb4
bb51
bb31
bb32
bb42
bb41
ad0
ad3
ad2
ad4
ad5
ad32
ad31
ad22
ad21
ad41
ad42
ad51
bd0
bd4
bd42
bd41
bd3
bd32
bd2
bd5
bd31
bd52
bd22
bd21
bd51


aa31
aa32
aa42
aa51
aa41
aa52
aa22
aa3
aa4
aa21
ba21
ba52
ba51
ba22
bc41



Oppgave 3.7

Diskuter viktigheten og robustheten til regionalnettet.

Antar at det med "regionalnett" menes de nodene som kun er navngitt med èn bokstav og ett siffer - typ "a2". Det fremkommer ved første øyekast at viktigheten til den noden som i hvert av regionalnettene har kanter til kjernenettet er meget høy (de med siffer = 0). Følgelig vil nettverkets avhengighet av disse nodene gå negativt ut over robustheten til nettverket totalt sett. Innad i selve regionalnettverket er det heller ikke så god robusthet. Her kan vi maksimalt fjerne en node før koblingen til ett av de to aksessnettverkene mistes. (F.eks vil regionalnettverket fortsatt fungere dersom "b4" fjernes, men opphøre dersom det fjernes ytterligere noder) Ved å legge til flere kanter til kjernenettverket fra andre noder i regionalnettverket, samt øke antall kanter innad i regionalnettverket vil vi oppnå et mer robust regionalnettverk.

Del 4: Feil og Angrep på nettverk

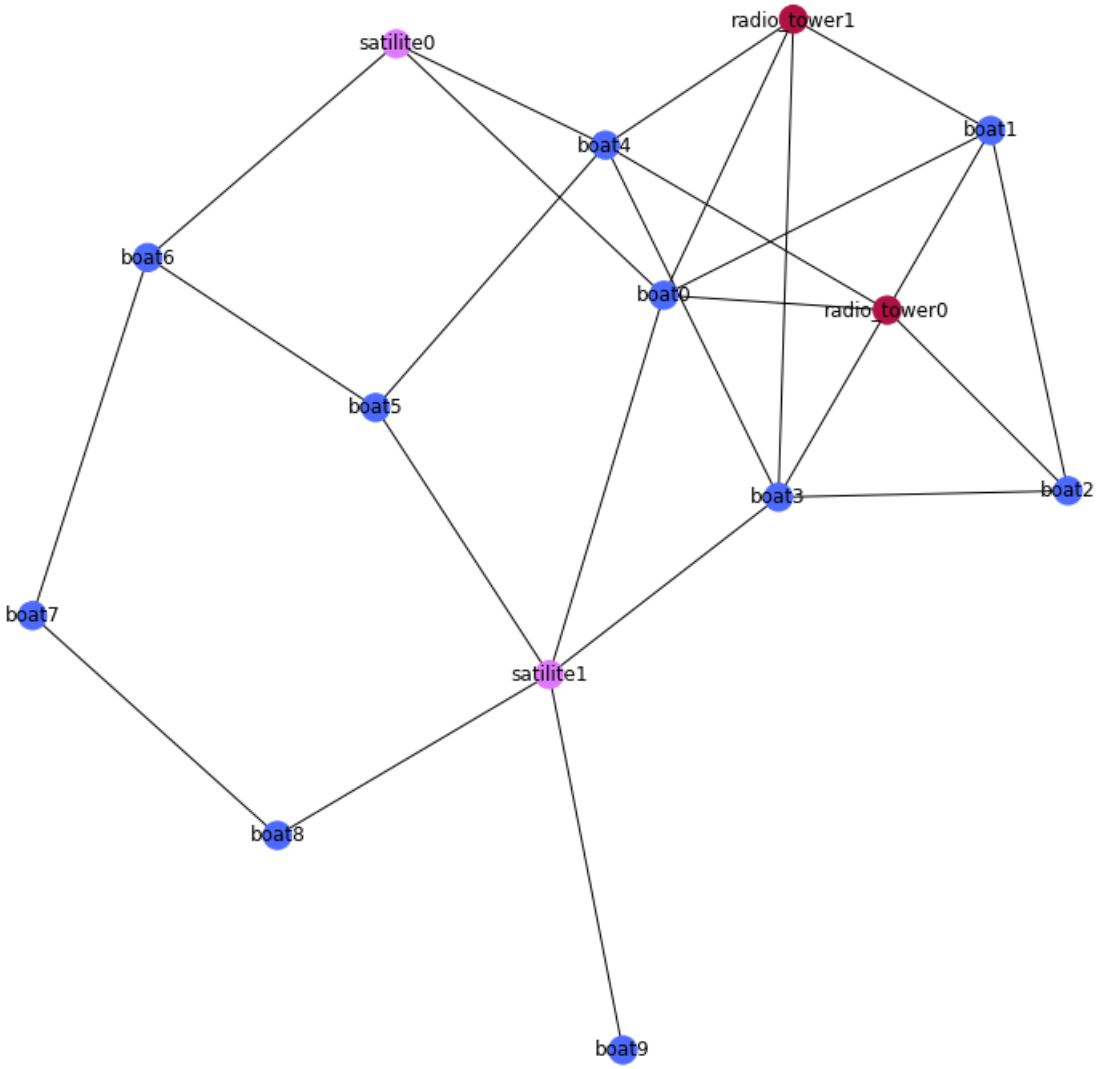
CASE: Du er en terrorist er ute etter å utføre mest mulig skade. Vi skal nå hjelpe deg med å se på hvor det er mest interessant å angripe.

Nettverket som angripes

Bruk objektet i denne oppgaven. Grafen er et utsnitt av et VDES nettverk i aksjon der båtene snakker med hverandre, satelittene og de landbaserte radiotårnene.

In [29]:

```
VDES = VDESGraph()
VDES.draw()
```



4.1 Analyse av nettverket

4.1.1 Sentraliteter

For å vite hvor vi skal angripe er det en fordel å vite hvor vi kan gjøre mest mulig skade. Bruk de numeriske verdiene for de forskjellige centrality målene du lærte i de tidligere delene for ut hvilken node som er den viktigste i nettverket.

In [30]:

```
closenessVDES = VDES.closeness_centrality()
degreeVDES = VDES.degree_centrality()
betweenessVDES = VDES.betweenness_centrality()

# Hjelpemetoder for å finne høyeste verdi (og hvis det er flere med like verdier, finne alle)
def findMaxDegree():
    maxDegree = {}
    v2 = max(degreeVDES.items(), key=operator.itemgetter(1))[1]
    for x in degreeVDES:
        if degreeVDES[x] == v2:
            maxDegree[x] = degreeVDES[x]
    print("\nhighest degree:\n",maxDegree)

def findMaxClosenness():
    maxClosenness = {}
    v1 = max(closenessVDES.items(), key=operator.itemgetter(1))[1]
    for x in closenessVDES:
        if closenessVDES[x] == v1:
            maxClosenness[x] = closenessVDES[x]
    print("\nhighest closeness:\n",maxClosenness)

def findMaxbetweeness():
    maxBetweeness = {}
```

```
    v3 = max(betweenessVDES.items(), key=operator.itemgetter(1))[1]
    for x in betweenessVDES:
        if betweenessVDES[x] == v3:
            maxBetweeness[x] = betweenessVDES[x]
    print("\nhighest betweeness:\n",maxBetweeness)

# for å enkelt kunne kalle alle på en gang
def findAllMaxCategories():
    findMaxDegree()
    findMaxClosenness()
    findMaxbetweeness()

findAllMaxCategories()

print("\n- Vi ser at 'satalite1' er har høyest verdi på alle centrality målene -")
```

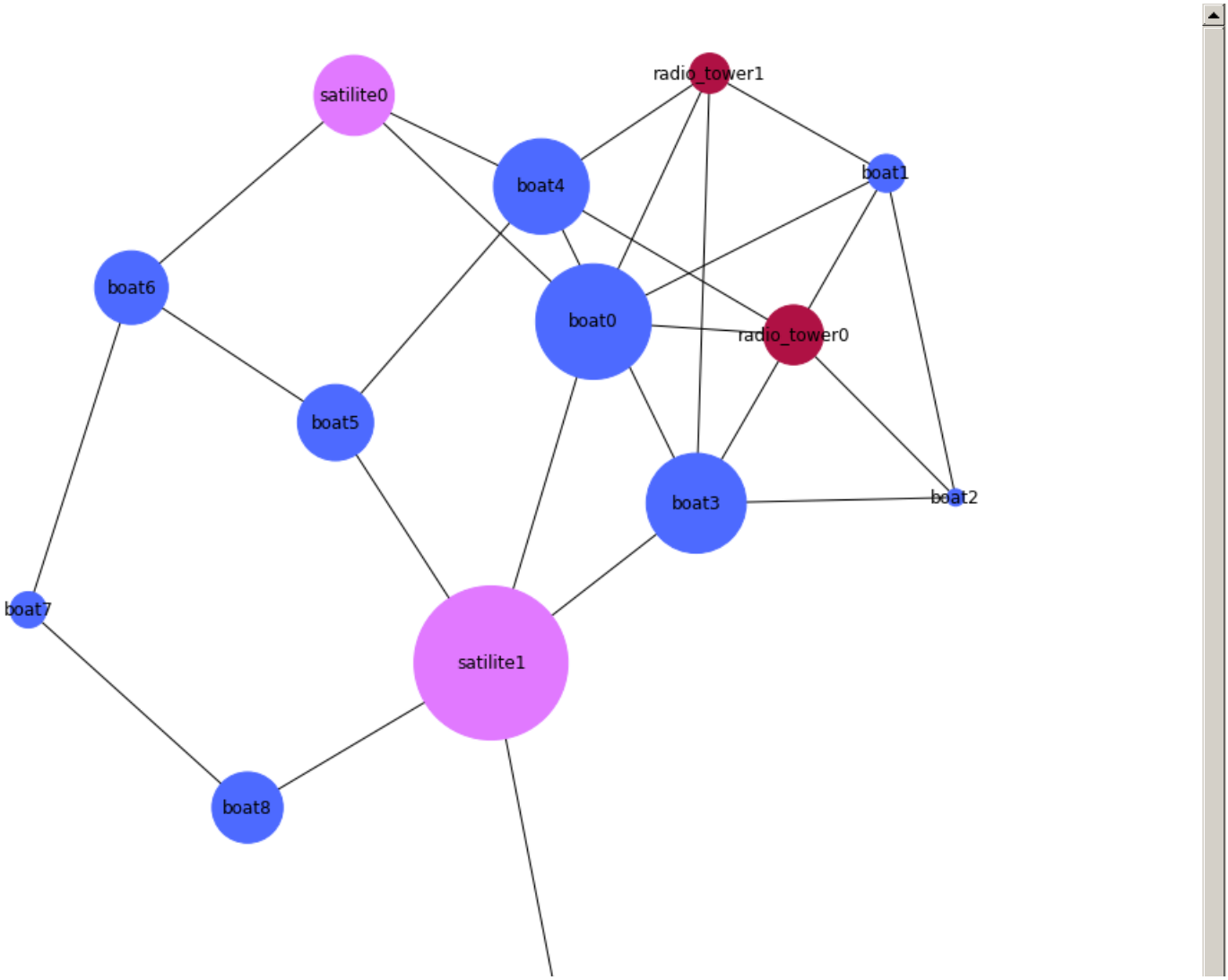
```
highest degree:
 {'boat0': 0.38461538461538464, 'boat3': 0.38461538461538464, 'boat4': 0.38461538461538464, 'satilite1':
0.38461538461538464, 'radio_tower0': 0.38461538461538464}

highest closeness:
 {'satilite1': 0.6190476190476191}

highest betweeness:
 {'satilite1': 0.3496947496947496}

- Vi ser at 'satalite1' er har høyest verdi på alle centrality målene -
```

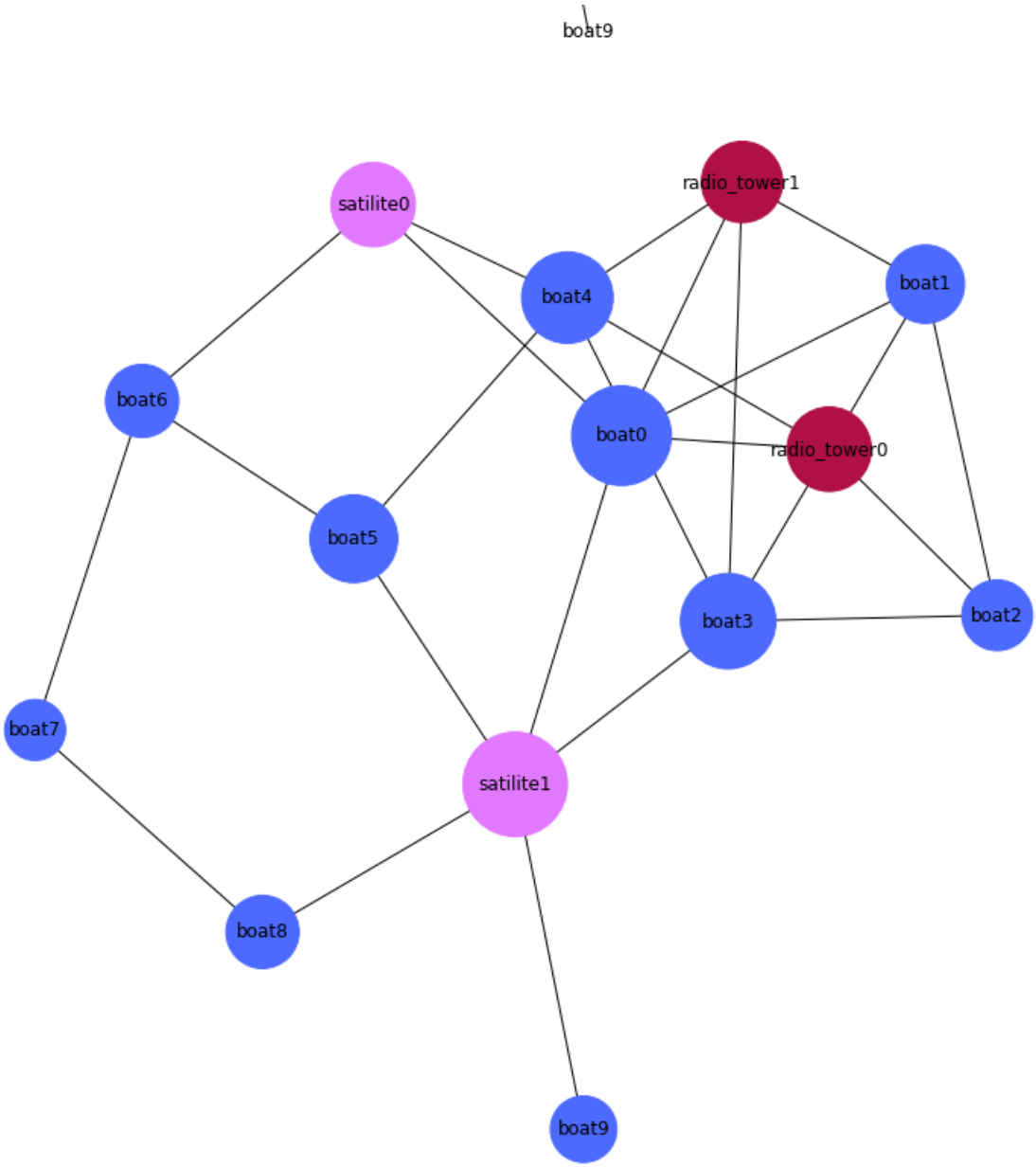
4.1.2 Sentraliteter del 2

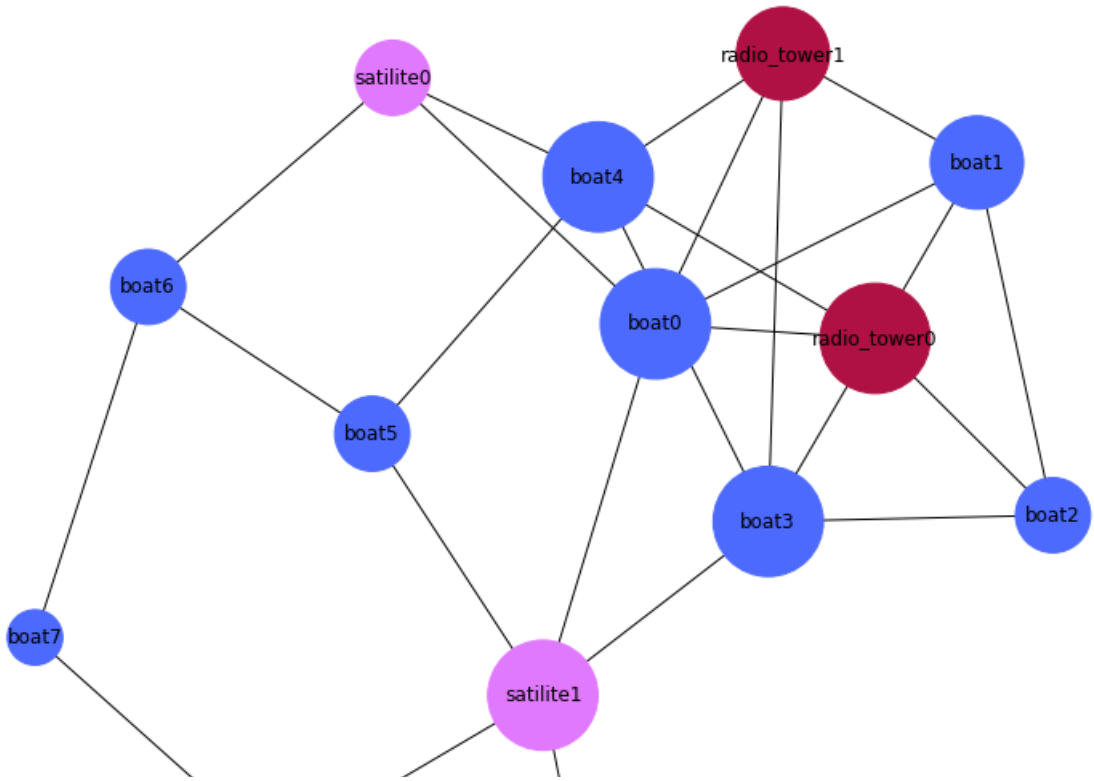
- Illustrer sentralitetene grafisk.
- Diskuter om nodene med lavest verdier på sentralitsmålene alltid er de minst viktige nodene i nettverket.

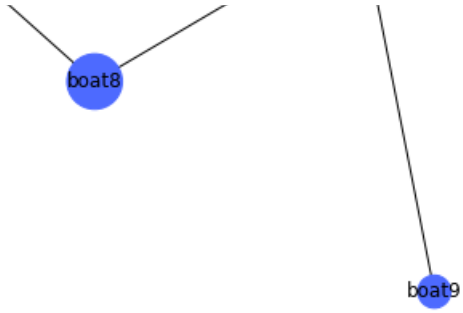
In [31]:

```
VDES.draw_betweenness_centrality(avg_size=3000)
VDES.draw_closeness_centrality(avg_size=3000)
VDES.draw_degree_centrality(avg_size=3000)
```




boat9
satilite0
radio_tower1
boat4
boat1
boat6
boat0
radio_tower0
boat5
boat3
boat2
boat7
satilite1
boat8
boat9


satilite0
radio_tower1
boat4
boat1
boat6
boat0
radio_tower0
boat5
boat3
boat2
boat7
satilite1



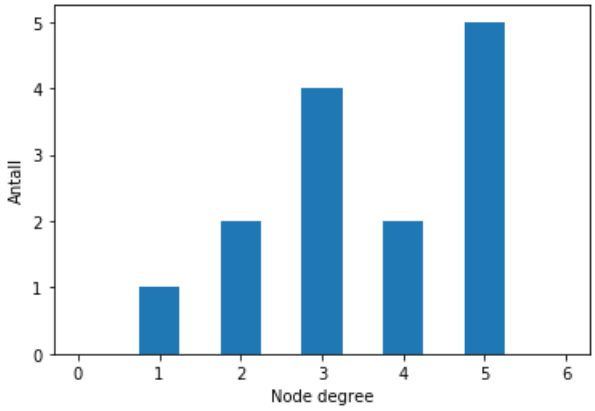
Hvis vi tar utgangspunkt i de 3 minste nodene i hver kategori. betweenness: båt 2, båt 7, båt 9 closeness: båt 2, båt 7, båt 9 degree: båt 7, båt 8, båt 9 Vi kan helt tydelig se at node 9 har minst betydning og skårer lavest i alle kategorier. Det er nok pga. det er en løvnode. Deretter kommer node 7. Grunnen til det er at selv om den har like mange node som f.eks. node 8, så er den mindre sentral og har lenger vei til andre noder (totalt sett). Til slutt har vi node 2 og 8 som bytter litt på for hver "render". Men oppsummert kan vi se at de to laveste nodene totalt sett er node 7 og 9.

4.1.3 Annen analyse

Analyser nettverket med degree distribution og kanter per node. Diskuter robustheten bassert på de nevnte målene.

In [32]:

```
VDES.histogram()
```



Out[32]:

```
[0, 1, 2, 4, 2, 5]
```

Fordelen med å kun ha en node med degree 1, er at det er bare i to tilfeller hvor man ikke klarer å aksessere denne noden. Det vil selvfølgelig være store konsekvenser, men sjanser for det skjer er lav (1/7). Videre er nodene jevnt fordelt med mange kanter som fører til at en omvei ikke blir stor hvis det skal skje et angrep.

4.2 Angrep av nettverket/feil med nettverket

4.2.1 Tilfeldige feil

Bruk metoden `delete_random_nodes` for å angripe grafene med tilfeldige angrep.

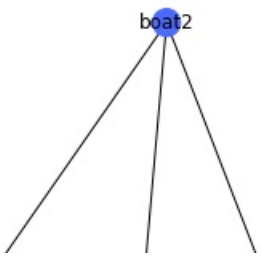
- Fjern én node. Tegn så grafen.
- Fjern tre noder. Tegn så grafen

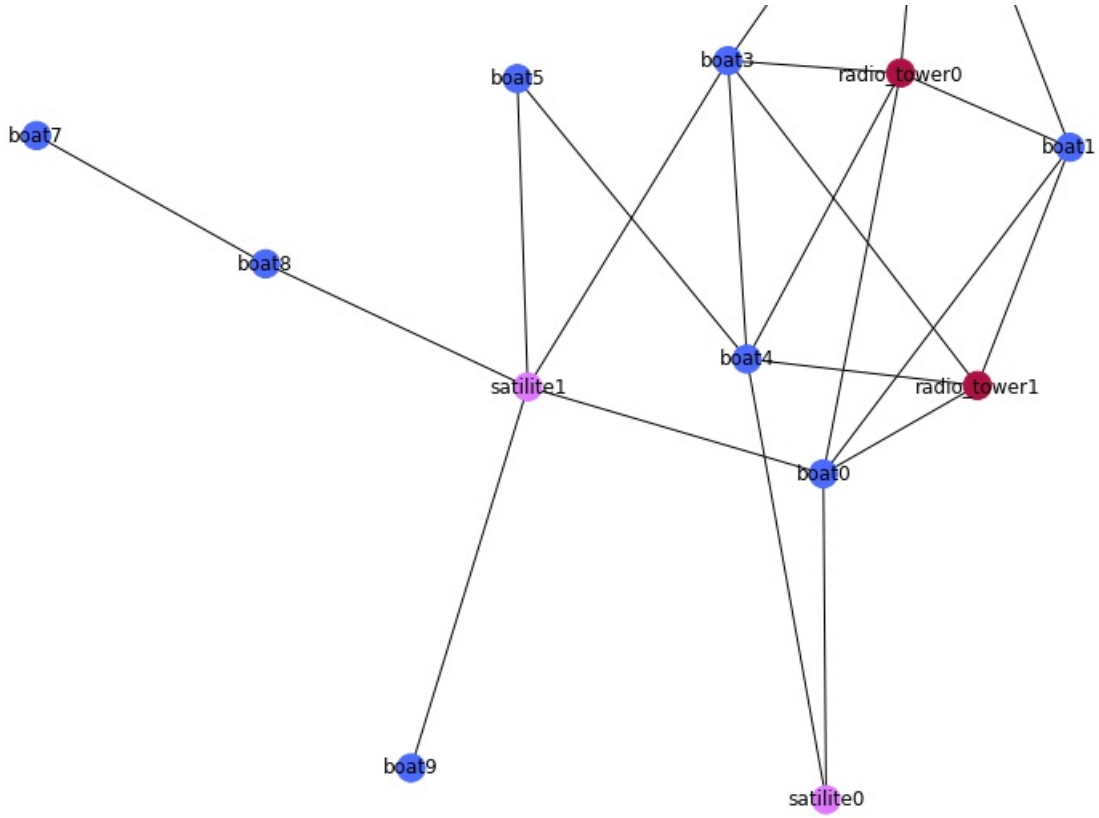
Random metoden her er seedet, altså vil den gi ut samme random hver gang.

In [33]:

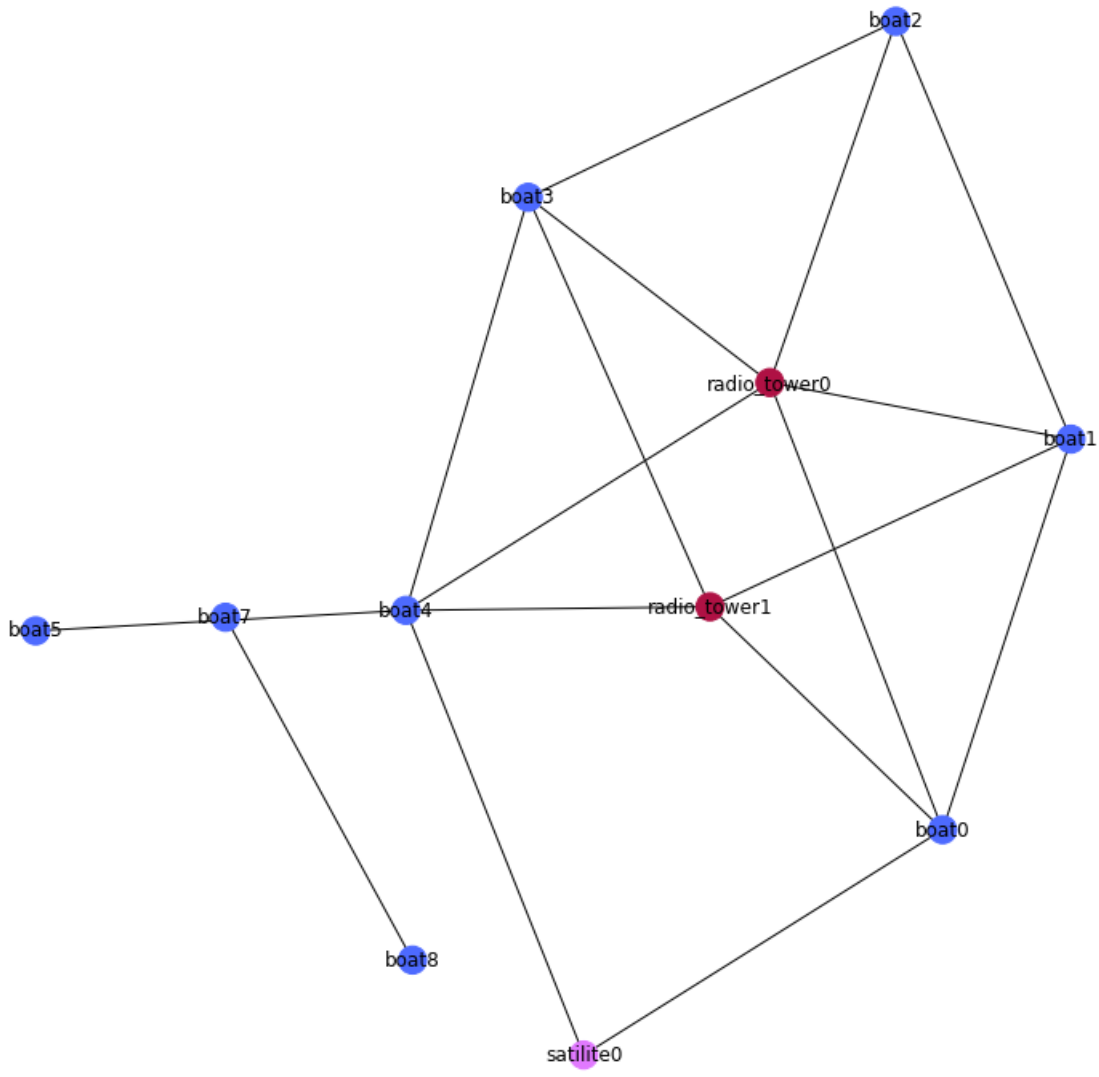
```
attackVDES = VDES.delete_random_nodes(1, print_result=True)
attackVDES.draw()
attackVDESrandom = VDES.delete_random_nodes(3, print_result=True)
attackVDESrandom.draw()
```

```
Removed node boat6 using random_fault
```






```
Removed node boat6 using random_fault
Removed node satilite1 using random_fault
Removed node boat9 using random_fault
```



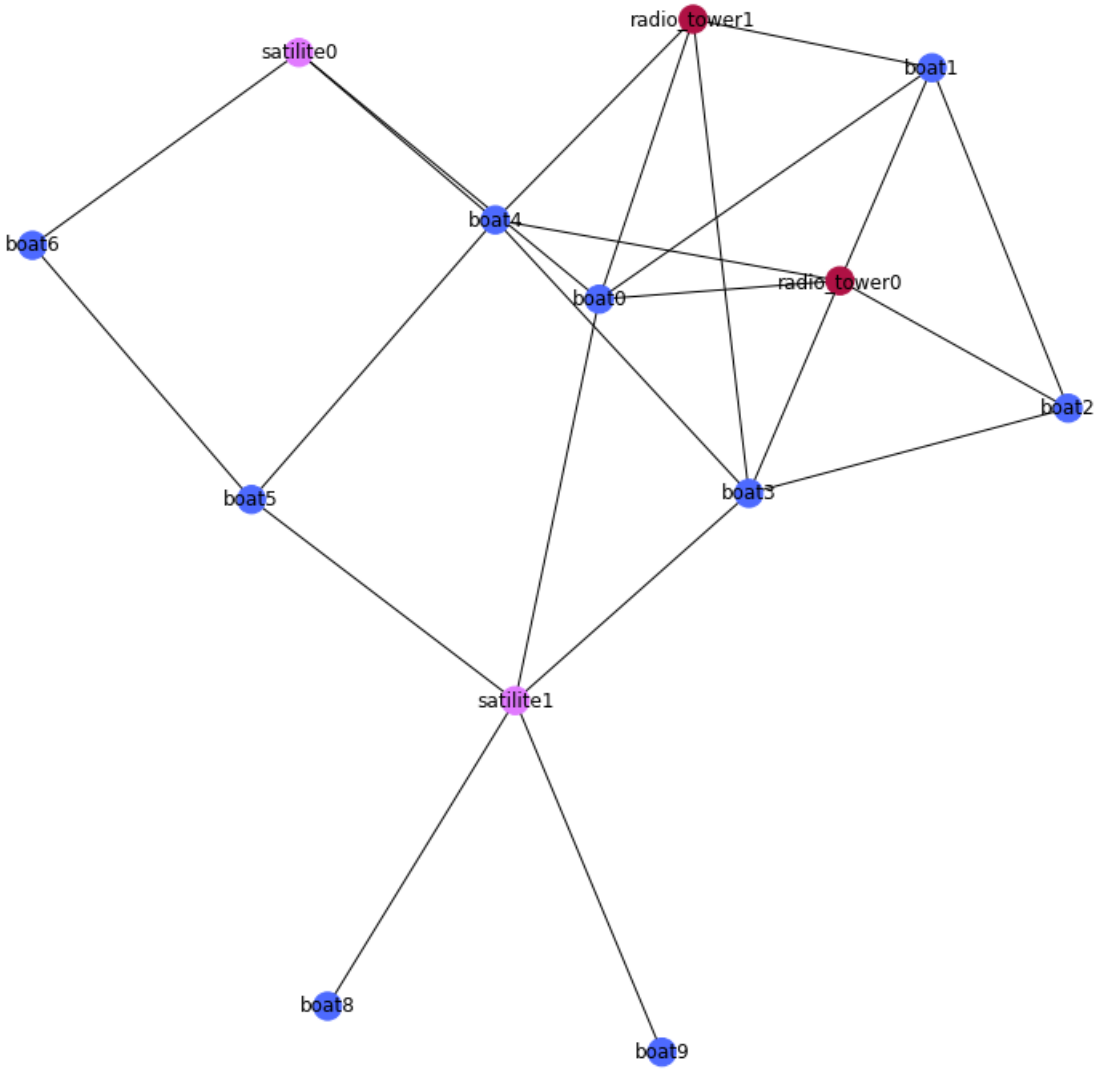
4.2.2 Enkelt angrep

Utfør ett enkelt målrettet angrep. Velg her den metoden som gjør mest skade og begrunn hvorfor.

In [30]:

```
attackRandomVDES = VDES.delete_nodes_attack(n=1, centrality_index="closeness", print_result=True)
print("- Fra oppg 4.1.1, får vi at Satalite1 har høyest verdi i alle kategoriene og vi utgjør dermed mest
attackVDES.draw()
```

```
Removed node satilite1 using closeness_centrality
- Fra oppg 4.1.1, får vi at Satalite1 har høyest verdi i alle kategoriene og vi utgjør dermed mest skade
ved å angripe denne -
```



4.2.3 Flere angrep

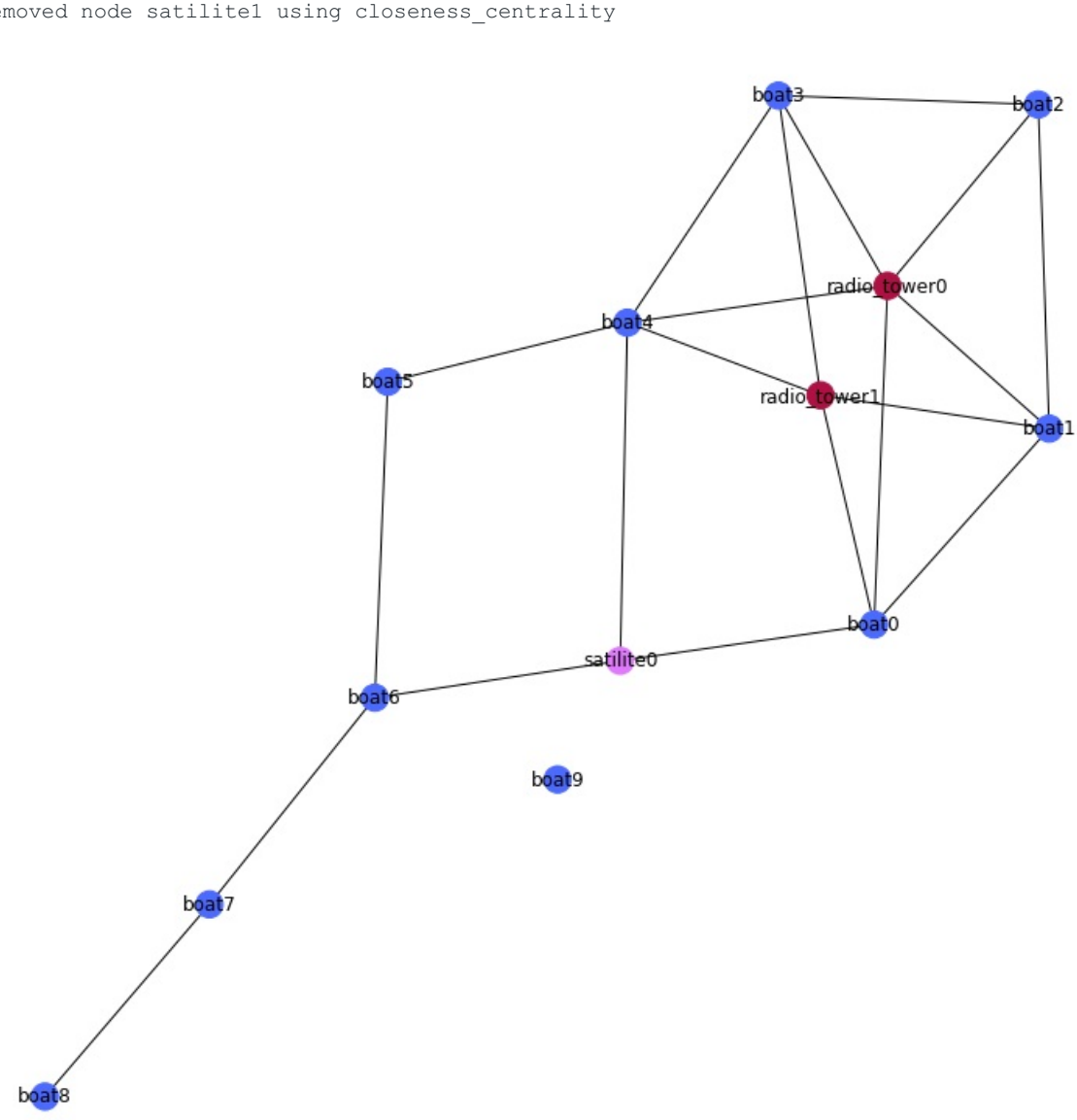
Utfør 3 angrep totalt og utfør mest mulig skade. Sammenlign med like mange tilfeldige feil og forklar resultatet.

In [31]:

```
# attack 1
# vi har gjort utregning for mest skade ved første node i oppgaven over
print("\n- Angrep 1 -")
attackVDES = VDES.delete_nodes_attack(n=1, centrality_index="closeness", print_result=True)
attackVDES.draw()

# attack 2
closenessVDES = attackVDES.closeness_centrality()
degreeVDES = attackVDES.degree_centrality()
betweenessVDES = attackVDES.betweenness_centrality()
print("- Angrep 2 -")
findAllMaxCategories()
print("\nser at båt 4 skårer høyest på degree og closeness")
```

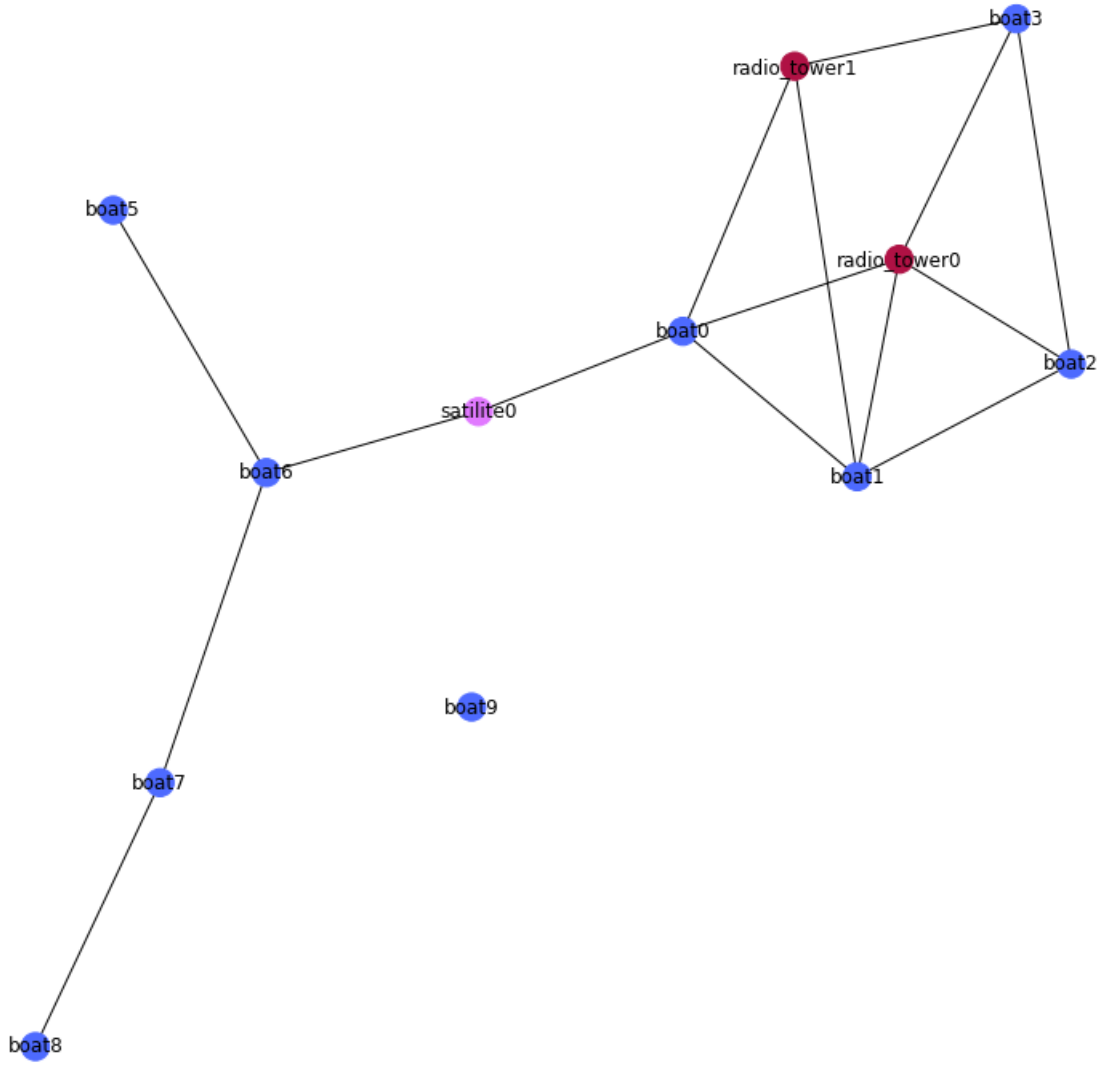
```
attackVDES2 = attackVDES.delete_nodes_attack(n=1, centrality_index="closeness", print_result=True)
attackVDES2.draw()

# attack 3
closenessVDES = attackVDES2.closeness_centrality()
degreeVDES = attackVDES2.degree_centrality()
betweenessVDES = attackVDES2.betweenness_centrality()
print("- Angrep 3 -")
findAllMaxCategories()
print("\nser at båt 0 skårer høyest på alle kategorier")
attackVDES3 = attackVDES2.delete_nodes_attack(n=1, centrality_index="closeness", print_result=True)
attackVDES3.draw()

# after attack 3
closenessVDES = attackVDES3.closeness_centrality()
degreeVDES = attackVDES3.degree_centrality()
betweenessVDES = attackVDES3.betweenness_centrality()
print("- Resultat etter angrep 3 -")
findAllMaxCategories()

# attack random nodes
closenessVDES = attackRandomVDES.closeness_centrality()
degreeVDES = attackRandomVDES.degree_centrality()
betweenessVDES = attackRandomVDES.betweenness_centrality()
print("\n\n- Random angrep -")
findAllMaxCategories()
```

```
- Angrep 1 -
Removed node satilite1 using closeness_centrality
```

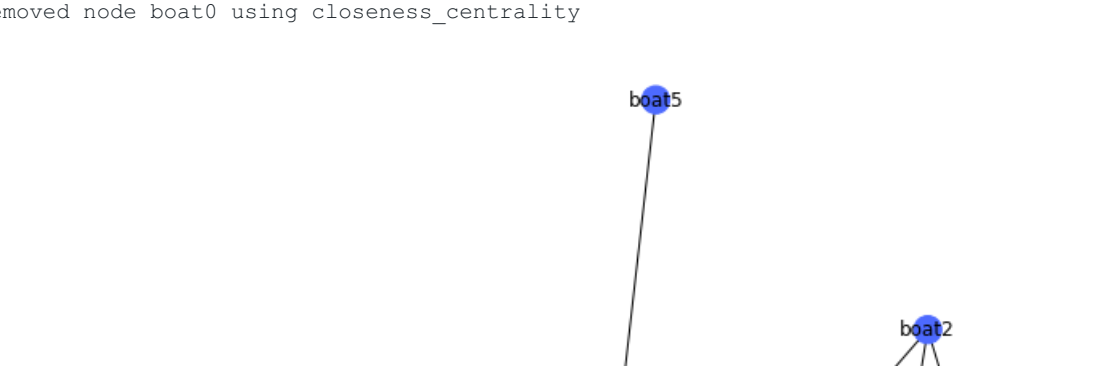


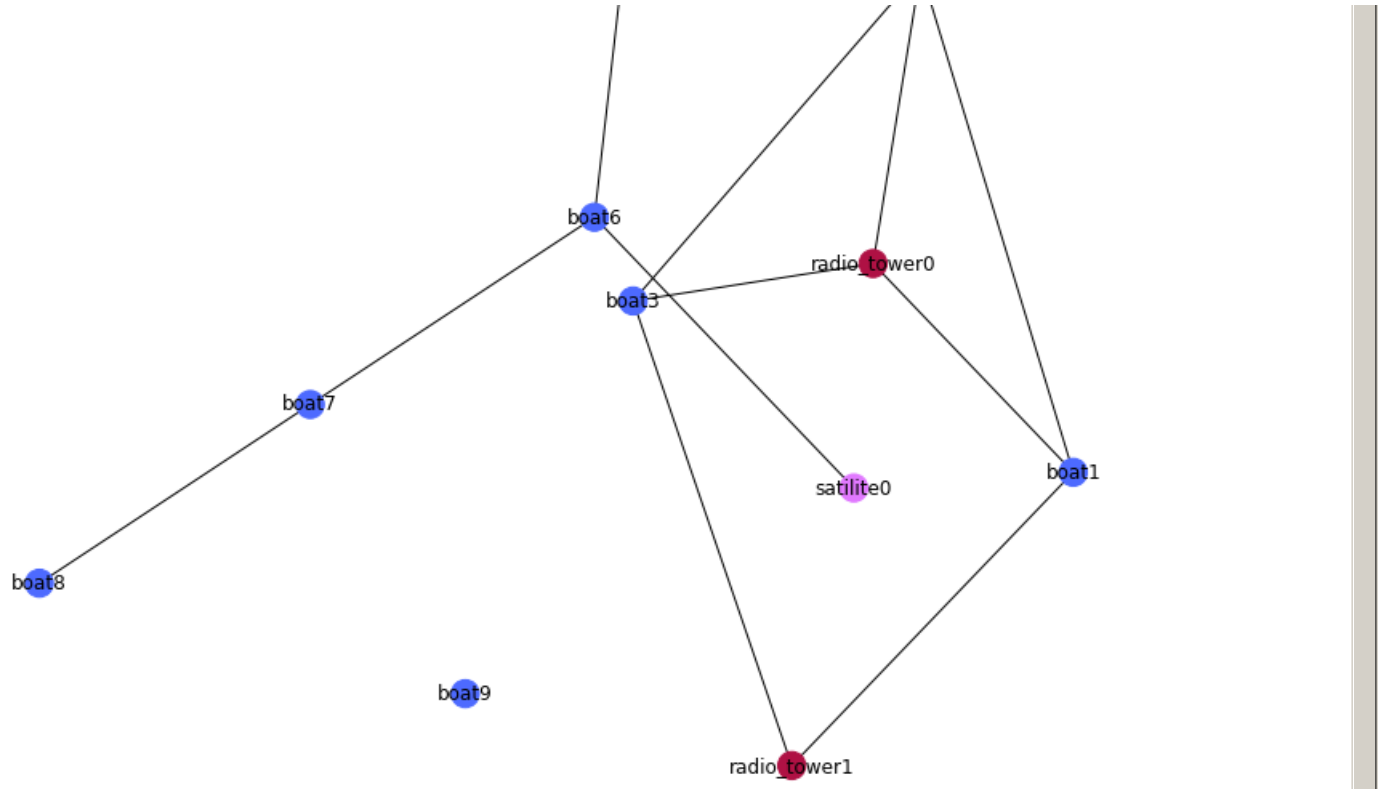

```
- Angrep 2 -

highest degree:
 {'boat4': 0.41666666666666663, 'radio_tower0': 0.41666666666666663}
```

```
highest closeness:
 {'boat4': 0.5041666666666667}

highest betweeness:
 {'boat6': 0.2840909090909091}

ser at båt 4 skårer høyest på degree og closeness
Removed node boat4 using closeness_centrality
```




```
- Angrep 3 -

highest degree:
 {'boat0': 0.36363636363636365, 'boat1': 0.36363636363636365, 'radio_tower0': 0.36363636363636365}

highest closeness:
 {'boat0': 0.45454545454545453}

highest betweeness:
 {'boat0': 0.46060606060606063}

ser at båt 0 skårer høyest på alle kategorier
Removed node boat0 using closeness_centrality
```






```
- Resultat etter angrep 3 -

highest degree:
 {'boat1': 0.30000000000000004, 'boat2': 0.30000000000000004, 'boat3': 0.30000000000000004, 'boat6':
0.30000000000000004, 'radio_tower0': 0.30000000000000004}

highest closeness:
 {'boat1': 0.32000000000000006, 'boat2': 0.32000000000000006, 'boat3': 0.32000000000000006, 'boat6':
0.32000000000000006, 'radio_tower0': 0.32000000000000006}

highest betweeness:
 {'boat6': 0.11111111111111112}


- Random angrep -

highest degree:
 {'boat4': 0.41666666666666663, 'radio_tower0': 0.41666666666666663}

highest closeness:
 {'boat4': 0.5041666666666667}

highest betweeness:
 {'boat6': 0.2840909090909091}
```

Vi kan se betydelig forskjell på tilfeldige angrep sammenlignet med angrep som går etter mest skade. Alle kategoriene er lavere med taktisk angrep. Dette kan jo være litt tilfeldig, men med taktisk angrep der man regner ut verdiene etter hvert angrep vil alltid få lavere eller likt resultat som et tilfeldig angrep.

4.2.4 Evaluering

Prøv å evaluere angrepene som du har gjort nå med metoder fra tidligere seksjoner.

In [32]:

```
# Din kode her
```

Slik vi tolket oppgave 4.2.3 er å gjøre beregninger for å utføre mest mulig skade. Dermed brukte vi metoder for å finne hvilke noder som hadde høyest verdi i de ulike Centrality metodene: (de som skal utføres her) degree, closeness og betweeness. Det er selvfølgelig mulig for argumentasjon for hvilken type centrality som er viktigst, men nodene vi angrpe hadde høyest verdi i flere kategorier og dermed naturlig å angripe disse. Etter første angrep oppnådde vi å gjøre båt 9 utigjenglig. Etter andre angrep oppnådde vi å gjøre nettverket veldig svakt ettersom det kun er en kant som deler de to delene av grafen. Samtidig hadde båt 4 utrolig mange kanter som fører til en stor omvei ved Shortest path. Etter tredje angrep klarer vi å dele grafen i to, slik at de er utilgjenglige for hverandre.

4.2.5 Evaluering part 2

For å videre evaluere angrepene vil vi introdusere "noder i største partisjon" som mål. Beregn størrelsen av største partisjon og sammenlign tilfeldig angrep med målrettet ved tre noder fjernet.

In [33]:

```
print("calculated attack:",attackVDES3.get_largest_components_size())
print("random attack:",attackVDESrandom.get_largest_components_size())

print("\nVi kan her se at random attack ikke klarte å gjøre noen noder utilgjenglig for hverandre (utenor
print("Sammenlignet med target attack hvor vi delte grafen i to, samtidig som vi gjorde boat9 utilgjenge
print("Skadene er derfor utrolig forskjellig")
```

```
calculated attack: 5
random attack: 11

Vi kan her se at random attack ikke klarte å gjøre noen noder utilgjenglig for hverandre (utenom de som
er fjernet).
Sammenlignet med target attack hvor vi delte grafen i to, samtidig som vi gjorde boat9 utilgjengelig ved
å fjerne satal0ite1
Skadene er derfor utrolig forskjellig
```

4.2.6 Evaluering part 3

Diskuter fordeler og ulemper med noder i største partisjon som pålitelighetsmål. Er dette et fornuftig mål i vårt tilfelle?

Fordeler: Fremtille noe abstrakt på en veldig konkret måte. Når nettverket ikke ikke gir mye informasjon kan centrality: degree, closeness og betweeness i kombinasjon med histrogram gi utrolig mye informasjon om komplektisteten. Vi kan også se hvor et angrep til utføre mest mulig skade. Fungerer bra i et lite netverk. Med 14 noder kan vi påføre mye skade (som sett over). det lønner seg derfor i vårt tilfelle. Ulemper: Det fungerer ikke bra i et utrolig stort nettverk. Gitt et nettverk med 4000 noder må man utføre veldig mange angrep før man begynner å se et resultat.

4.2.7 Evaluering part 4

Drøft kort viktigheten av informasjonen rundt en nettverkstopologi.

Viktig informasjon er: størrelse til nettverket, hvor komplekst nettverket er (anatall kanter) og hvor man kan utføre mest skade. Et eksempel er et lite nettverk, men hvis alle noder går til alle er det vil et angrep utgjøre minimalt med skade. Resultatet er at hvis vi vet denne informasjonenn, kan vi vurderer om vi vil bruke tid til å angripe eller ikke. Derfor er denne informasjonen viktig å holde skjult.

4.3 Sikring av nettverket

4.3.1 Flere kanter

Legg til tre kanter for å sikre flest mulig noder mot angrep. Diskuter så hvorfor du la til de kantene du gjorde og om det nå er mer robust mot angrep.

In [35]:

```
# En bug gjør at jeg må fjerne kantene jeg legger til, men de viser fortsatt feil verdier
# VDES2.remove_edge("boat9","boat8")
# VDES2.remove_edge("boat7","boat5")
# VDES2.remove_edge("boat9","satilite0")
VDES2 = VDES

def findMinDegreeVDES():
    minDegree = {}
    degreeVDES = VDES2.degree_centrality()
    v2 = min(degreeVDES.items(), key=operator.itemgetter(1))[1]
    for x in degreeVDES:
        if degreeVDES[x] == v2:
            minDegree[x] = degreeVDES[x]
    print("lowest degree:\n",minDegree)


# kant 1
print("\n- adding edge 1 -")
findMinDegreeVDES()
VDES2.add_edge("boat9","boat8")
print("Adding edge between boat9 and boat8")

# kant 2
print("\n- adding edge 2 -")
findMinDegreeVDES()
VDES.add_edge("boat7","boat5")
print("Adding edge between boat7 and boat5")
```

```
# kant 3
print("\n- adding edge 3 -")
findMinDegreeVDES()
VDES.add_edge("boat9","satilite0")
print("Adding edge between boat9 and satilite0")



print("\nGrunnen til at jeg har lagt til disse kantene er for å gjøre noder med lav degree mer robuste. `
```

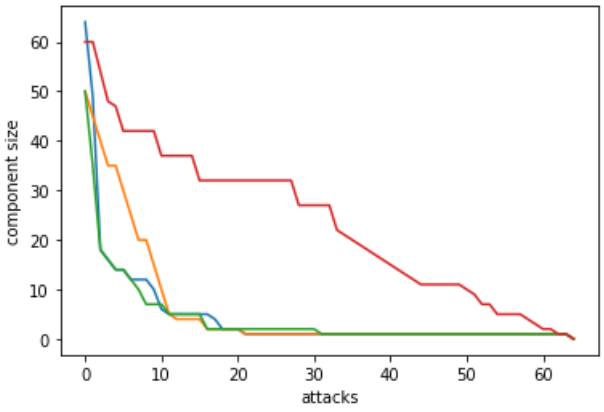
```
- adding edge 1 -
lowest degree:
 {'boat9': 0.07692307692307693}
Adding edge between boat9 and boat8

- adding edge 2 -
lowest degree:
 {'boat7': 0.15384615384615385, 'boat9': 0.15384615384615385}
Adding edge between boat7 and boat5

- adding edge 3 -
lowest degree:
 {'boat9': 0.15384615384615385}
Adding edge between boat9 and satilite0

Grunnen til at jeg har lagt til disse kantene er for å gjøre noder med lav degree mer robuste.
I tilegg hvis flere noder har samme degree kobler jeg til noden lengst unna, slik at Shortest path
(closenness og betweenness øker) og omveien ikke blir så stor.
```

4.3.2 Plotting av robusthet

Antall noder i største partisjon er et eksempel på et mål for hvor mye av et nettverk som har overlevd. Plott et linjediagram (en graf) med antall noder i største komponent langs y_aksen, antall noder fjernet på x_aksen og fjern en etter en node med de 4 forskjellige angrepene.
Lag en funksjon for denne grafen.
Bruk ConstructedGraph nettverket og plott grafen.

In [45]:

```
def allFourAttacks(graph):
    nodes = graph.number_of_nodes()

    # 4 forskjellige angrep
    graphs = [graph, graph, graph, graph]
    values = [[0 for _ in range(nodes)] for _ in graphs]

    # angriper grafen med forskjellig angrep.
    for attackType in range(0,nodes-1):
        graphs[0] = graphs[0].delete_nodes_attack(n=1, centrality_index="closeness", print_result=False)
        graphs[1] = graphs[1].delete_nodes_attack(n=1, centrality_index="degree", print_result=False)
        graphs[2] = graphs[2].delete_nodes_attack(n=1, centrality_index="betweenness", print_result=False)
        graphs[3] = graphs[3].delete_random_nodes(1, print_result=False)

        # finne største komponenet for vært angrep
        for number, graph in enumerate(graphs):
            values[number][attackType] = graphs[number].get_largest_components_size()

    plt.xlabel("attacks")
    plt.ylabel("component size")
    # legge til i grafen
    for z in range(4):
        plt.plot(values[z])


allFourAttacks(ConstructedGraph(expanded=False))
```



4.3.3 Sammenligning av angrepene

Er noen av angrepene bedre enn andre? Hvordan gjør tilfeldige feil det?

Det skal sier at random kan bli bedre, men når jeg har kjørt grafen flere ganger er den oftest dårligst. Her kan vi tydelig se at med et tilfeldig angrep til det ta lang tid før man får et resultat, sammenlignet med target attack. De tre centrality angrepene er ganske like, men vi kan utifra grafen se at closeness og betweenness er nesten identiske. Degree er derimot litt treigere.

4.3.4 Beregning av kostnad

Kostnaden kan måles med antall kanter per node i nettverket. Beregn kanter per node for VDES med ekstra kanter og VDES uten.

In [36]:

```
# Hadde problemer med noen bugs, lager derfor VDES på nytt
VDES3 = VDESGraph()

print("- Uten ekstra kanter -")
for node in VDES3:
    print(node,":",len(VDES3.edges(node)))

print("\n- Med ekstra kanter -")
for node in VDES2:
    print(node,":",len(VDES2.edges(node)))
```

```
- Uten ekstra kanter -
boat0 : 5
boat1 : 4
boat2 : 3
boat3 : 5
boat4 : 5
boat5 : 3
boat6 : 3
boat7 : 2
boat8 : 2
boat9 : 1
satilite0 : 3
satilite1 : 5
radio_tower0 : 5
radio_tower1 : 4

- Med ekstra kanter -
boat0 : 5
boat1 : 4
boat2 : 3
boat3 : 5
boat4 : 5
boat5 : 4
boat6 : 3
boat7 : 3
boat8 : 3
boat9 : 3
satilite0 : 4
satilite1 : 5
radio_tower0 : 5
radio_tower1 : 4
```

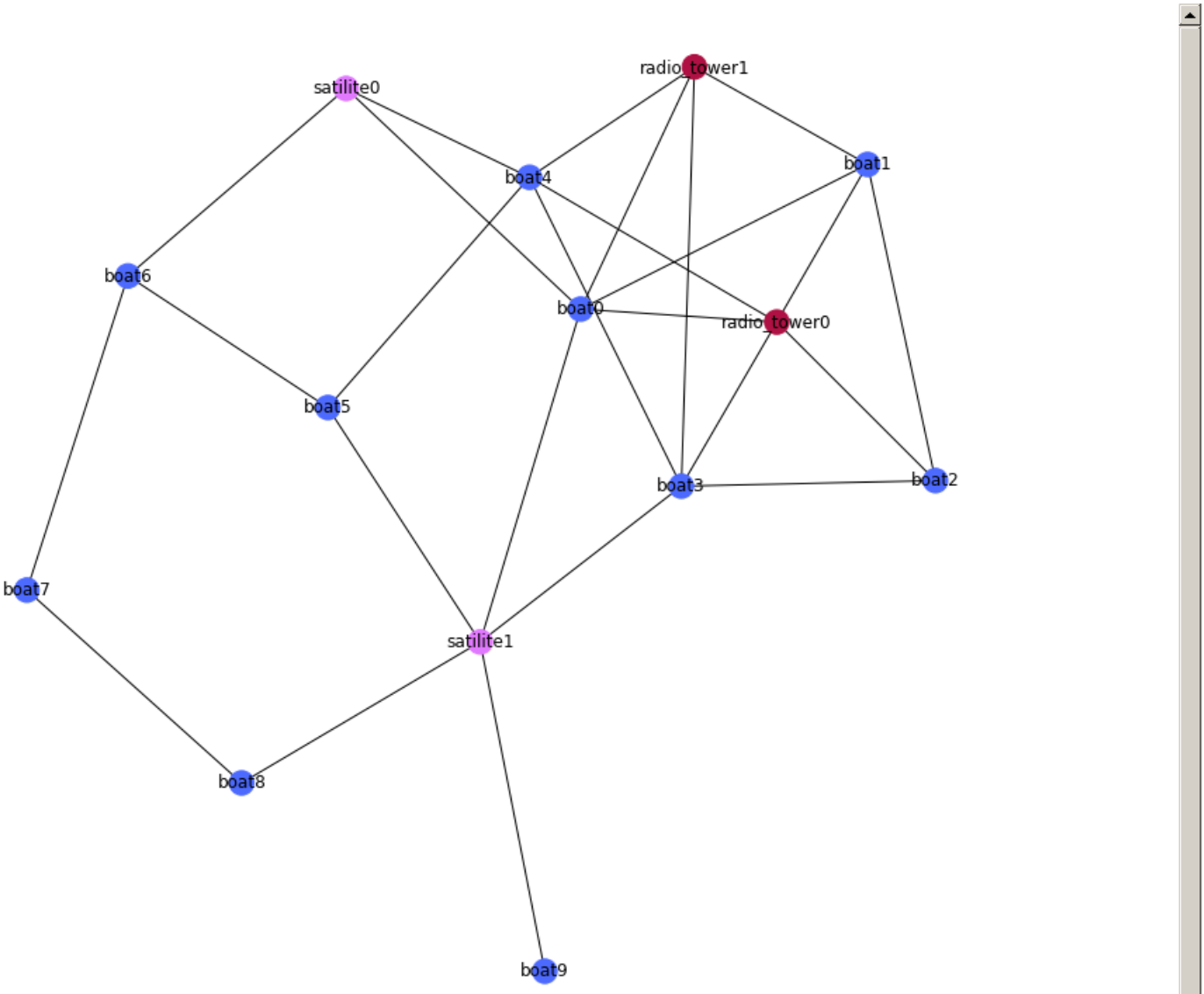
4.3.5 Sammenligning med ekstra redundans

- Sammenlign det originale VDES nettverket med VDES nettverket med ekstra redundans ved å plotte en lik graf som i 4.3.2 for begge.
- Ble nettverket sikrere?
- Er det verdt det?
- Er det alltid verdt det?

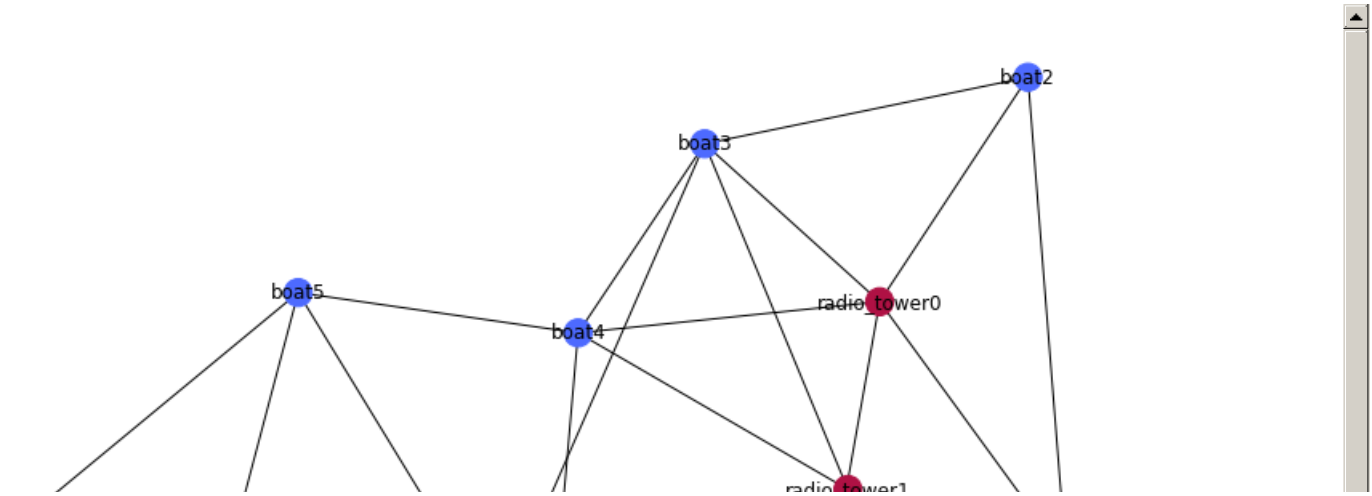
In [37]:

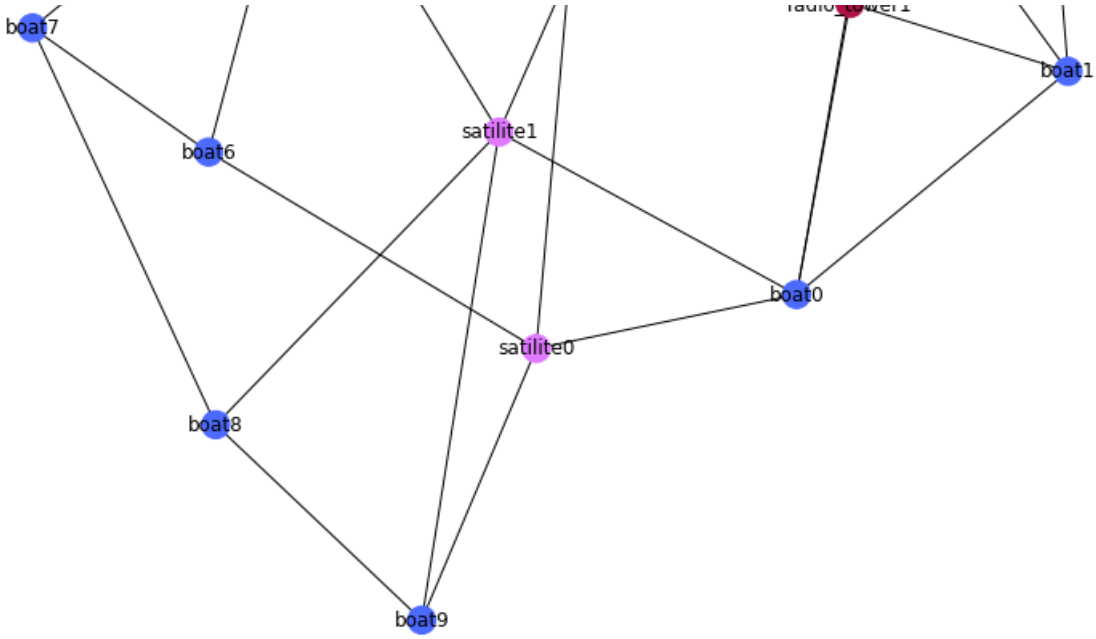
```
print("Orginal VDES")
VDES3.draw()
print("VDES med ekstra kanter")
VDES2.draw()
```

```
Orginal VDES
```




```
VDES med ekstra kanter
```





Nettverket ble absolutt sikrere med redundans. Nå er det ingen noder med mindre enn 3 kanter, dvs. at hvis en node svikter, er det ikke sikkert det vil påvirke nettverket i noen grad. I tilegg la jeg til kanter mellom noder langt unna hverandre (boat9 og satalite0) dvs. at jeg oppretter ny Shortest path ikke bare for de nodene, men alle nodene rundt. Hvis vi har et nettverk med 7 noder, der alle går til alle, vil det være litt overkill. Med andre ord kommer vi til et punkt for redundans ikke påvirker i stor grad bortsett fra kostnad og øker kompleksiteten i nettverket. Man burde stille spørsmålet om det egentlig er nødvendig.

4.3.6 Kostnad

Diskuter hvor mye burde man sikre et nettverk med hensyn på redundans?

Som i forrige oppgave, rendundans er kjempefint for å få ekstra funksjonalitet i tilfelle en node svikter, men nettverk blir fort kompleks med høy degree per node. Dvs. at man burde være forsiktig med å ikke gjøre nettverket for komplekst enn det trenger å være. Man må derfor finne en balanse mellom sikkerhet og kompleksistet/ kostnad.

In [ ]: