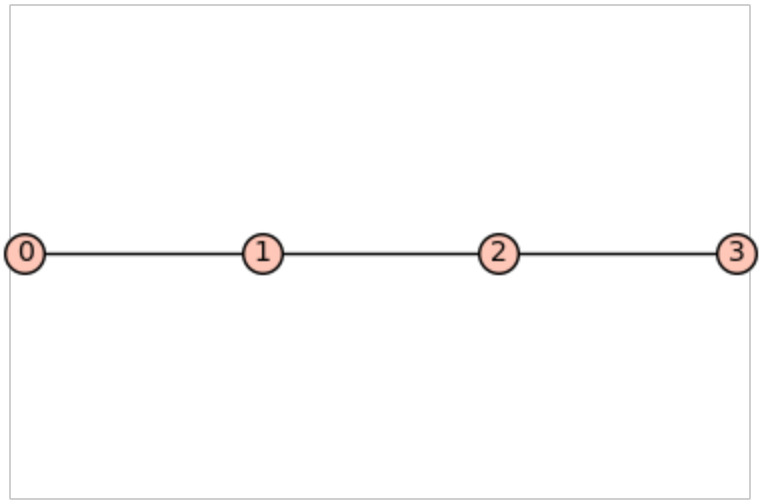
In [ ]:
```
###################################
# This Sage worksheet is written by Bryan Curtis and Leslie Hogben hogben@ai
# using code written by numerous various people.
# It includes computations for the paper
# Zero forcing irredundant sets
# by
# Bryan Curtis, Leslie Hogben, and Riana Roux
#
# To run computations, you must first enter the function F- cells at the end
###################################
```

In [ ]:
```
# Verify ZIR(P_4) (Observation 3.4)
# up_low_ZIR(G) returns [ZIR(G), zir(G)]
```

In [16]:
```
g=graphs.PathGraph(4)
show(g)
up_low_ZIR(g)
```



Out[16]: [2, 1]

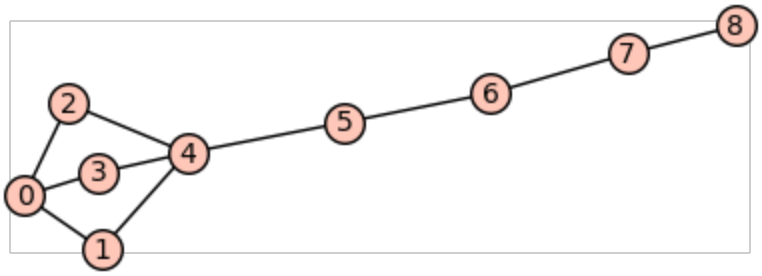
In [ ]:
```
# ZIRsets(G) redurns all maximal ZIr-sets of G
```

In [17]:
```
ZIRsets(g)
```

Out[17]: [{1, 2}, {3}, {0}]

In [ ]:
```
# Verify zir(H(3,5)), Z(H(3,5)), Zbar(H(3,5)), ZIR(H(3,5)) (comment before F
```

In [7]:
```
g=Graph({0:[1,2,3],4:[1,2,3,5],6:[5,7],8:[7]})
show(g)
```



In [8]:
```
up_low_ZIR(g)
```

Out[8]: [5, 2]

In [9]:
```
Z(g)
```

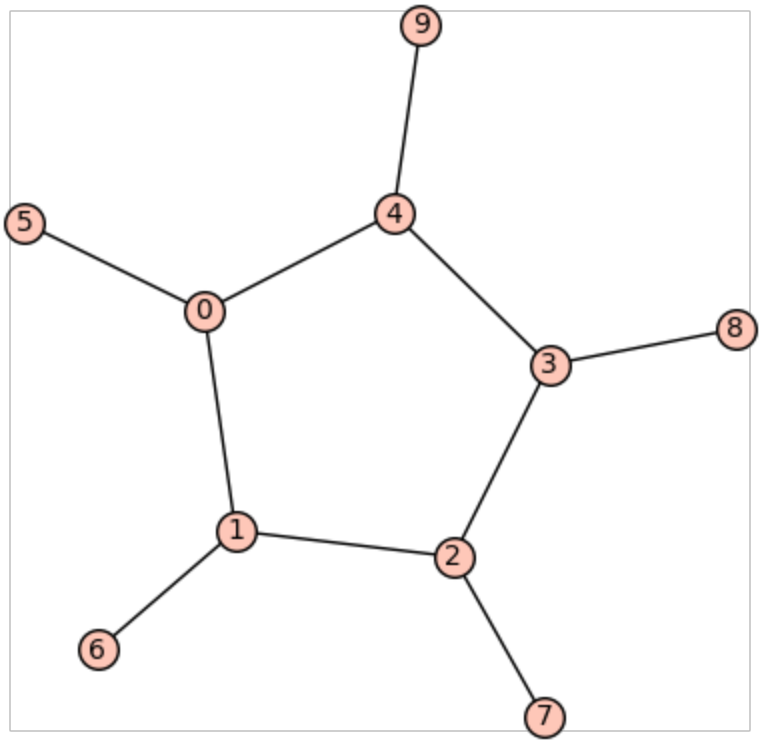
Out[9]: 3

In [10]:
```
zbar(g)
```

Out[10]: 4

In [ ]:
```
# Example 7.2: Verify that zir(G)=3 for the graph G in Figure 7.2 (the penta
```

In [18]:
```
g=Graph({0:[1,5],1:[2,6],2:[3,7],3:[4,8],4:[0,9]})
show(g)
```

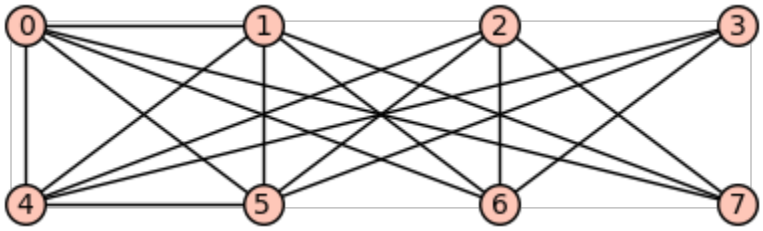


In [19]:
```
up_low_ZIR(g)
```

Out[19]: [5, 3]

In [ ]:
```
# Example 7.4: Verify that zir(G)=4 for the graph G in Figure 7.2
```

In [3]:
```
g=graphs.CompleteBipartiteGraph(4,4)
g.add_edges([(0,1),(5,4)])
g.delete_edge([3,7])
show(g)
```



In [7]:
```
up_low_ZIR(g)
```

Out[7]:
```
[5, 4]
```

In [8]:
```
g.treewidth()
```

Out[8]:
```
5
```

In [4]:
```
# Cell F-ZIR1
# load main minimum rank/maximum nullity and zero forcing functions
# developed by S. Butler, L. DeLoss, J. Grout, H.T. Hall, J. LaGrange, J.C.-
# updated and maintained by Jephian C.-H. Lin
load("https://raw.githubusercontent.com/jephianlin/minrank_aux/master/load_a
load_all(mr_JG=True, minrank_aux=False, timeout=5, load_func="loadurl")
```

```
/usr/lib/python3/dist-packages/sage/misc/remote_file.py:46: DeprecationWarni
ng: ssl.SSLContext() without protocol argument is deprecated.
  content = urlopen(req, timeout=1, context=SSLContext())
/usr/lib/python3/dist-packages/sage/misc/remote_file.py:46: DeprecationWarni
ng: ssl.PROTOCOL_TLS is deprecated
  content = urlopen(req, timeout=1, context=SSLContext())
Compiling /home/jupyter-hogben@iastate.edu-c55cc/.sage/temp/sagemath2/46443
3/tmp_w27tzsmg.pyx...
xrange test failed: define xrange = range
Loading Zq_c.pyx...
Loading Zq.py...
Loading zero_forcing_64.pyx...
Compiling /home/jupyter-hogben@iastate.edu-c55cc/.sage/temp/sagemath2/46443
3/tmp_ph4ft_98.pyx...
Compiling /home/jupyter-hogben@iastate.edu-c55cc/.sage/temp/sagemath2/46443
3/tmp_90gk2z3z.pyx...
Loading zero_forcing_wavefront.pyx...
Loading minrank.py...
Loading inertia.py...
```

In [5]:
```
# Cell F-ZIR2
# Useful tools


# function: is_fort(G,F)
```

```
# input: graph G and set F of vertices
# output: True if F is a fort, False if F is not a fort

def is_fort(G,F):
    # initialize flag isFort
    if len(F) > 0:
        isFort = True
    else:
        isFort = False
    F = set(F)
    V = set(G.vertices())
    Z = V - F
    # check if a vertex in the complement of F is adjacent to exactly 1 vert
    for v in Z:
        if len(set(G.neighbors(v)) & F) == 1:
            isFort = False
            break
    return isFort


# function: kforts(G,k)
# input: a graph G and a positive integer k
# output: a list of all forts of G of size k

def kforts(G,k):
    V = G.vertices()
    S = Subsets(V,k)
    forts = []
    for f in S:
        if is_fort(G,f):
            forts.append(set(f))
    return forts


# function: all_forts(G)
# input: a graph G
# output: a list of all forts of G

def all_forts(G):
    V = set(G.vertices())
    S = Subsets(V)
    forts = []
    for f in S:
        if is_fort(G,f):
            forts.append(set(f))
    return forts


# function: is_ZIRset(G,T)
# input: graph G and set of vertices T with at least 1 element
# output: True if T is a ZIR set and False otherwise

def is_ZIRset(G,T):
    T = set(T)
    S = set([])
    ZIR = False
```

```
    forts = all_forts(G)
    for F in forts:
        F_int = F.intersection(T)
        if len(F_int) == 1:
            S = S.union(F_int)
        if len(S) == len(T):
            ZIR = True
            break
    return ZIR


# function: ZIRset_cert(G,T)
# input: graph G and set T
# output: False if T is not a ZIR set, otherwise private forts

def ZIRset_cert(G,T):
    T = set(T)
    S = set([])
    ZIR = False
    cert = []
    forts = all_forts(G)
    for F in forts:
        F_int = set(F).intersection(T)
        if len(F_int) == 1:
            cert.append(F)
            S = S.union(F_int)
        if len(S) == len(T):
            ZIR = True
            break
    return cert if ZIR == True else ZIR


# function: mult_ZIR_cert(G,T,forts)
# input: graph G, set T, and list of all forts
# output: False if T is not a ZIR set, otherwise private forts

def mult_ZIRset_cert(G,T,forts):
    T = set(T)
    S = set([])
    ZIR = False
    cert = []
    for F in forts:
        F_int = set(F).intersection(T)
        if len(F_int) == 1:
            cert.append(F)
            S = S.union(F_int)
        if len(S) == len(T):
            ZIR = True
            break
    return cert if ZIR == True else ZIR
```

In [6]:

```
# Cell F-ZIR3
# Function for finding ZIR sets

# Possible improvements: Carry fort data with vertices - would reduce checki
#                        Incorporating bounds on lower zir number (min degre
```

```
# function: is_ZIRset_mult(G,T,forts)
# input: graph G, set of vertices T with at least 1 element, list of all for
# output: True if T is a ZIR set and False otherwise

def is_ZIRset_mult(G,T,forts):
    T = set(T)
    S = set([])
    ZIR = False
    for F in forts:
        F_int = F.intersection(T)
        if len(F_int) == 1:
            S = S.union(F_int)
        if len(S) == len(T):
            ZIR = True
            break
    return ZIR


# function: extend(G,L,k)
# input: graph G, list L, and integer k
# output: add vertices to L to increase size of ZIR set

def extend(G,L,forts):
    n = G.order()
    k = L[-1:][0]    # last element of L
    new_L = list(L)
    for i in range(k+1,n+1):    # add each vertex greater than k to L in con
        if is_ZIRset_mult(G,new_L + [i],forts):
            new_L.append(i)
    return new_L


# function: update(L,ord)
# input: list L, positive integer ord
# output: updates L to the next list that needs to be checked

# function: update(L,ord)
# input: list L, positive integer ord
# output: updates L to the next list that needs to be checked

def update(L,ord):
    n = len(L)
    flag = False
    if ord - L[-1:][0] > 1:                                  # check if L conta
        new_L = [x for x in L if x < L[-1:][0]]              # remove largest v
        new_L.append(L[-1:][0] + 1)                          # add vertex v+1 t
        flag = True
    else:                                                    # runs the followi
        for s,t in zip(reversed(L),reversed(L[:n-1])):       # zip L and copy o
            if s - t > 1:                                    # check for non co
                flag = True
                new_L = [x for x in L if x < t]              # once non consecu
                new_L.append(t+1)                            # append t+1
```

```
                break
    return new_L if flag == True else False


# function: max_subsets(sets)
# input: list of lists
# output: removes subsets

def max_subsets(sets):
    sets = reversed(sorted(map(set, sets), key=len))
    maximal_subsets = []
    for s in sets:
        if not any(s.issubset(maximal_subset) for maximal_subset in maximal_
            maximal_subsets.append(s)
    return maximal_subsets


# function: ZIRsets(G)
# input: graph G
# output: all maximal ZIR sets

def ZIRsets(G):
    ord = G.order()
    forts = all_forts(G)
    L = [0]
    zsets = []    # initialize all maximal ZIR sets
    flag1 = True
    while flag1 == True:
        L = extend(G,L,forts)    # extend L to obtain maximal ZIR set
        zsets.append(L)
        flag2 = True
        while flag2 == True:    # update L to next ZIR set (still needs to b
            L = update(L,ord)
            if L == False:
                flag1 = False
                flag2 = False
            elif is_ZIRset_mult(G,L,forts):
                flag2 = False
    zsets = max_subsets(zsets)     # remove ZIR sets that are not maximal
    return zsets


# function: up_low_ZIR(G)
# input: graph G
# output: upper and lower ZIR number

def up_low_ZIR(G):
    zsets = ZIRsets(G)
    return [len(zsets[0]),len(zsets[-1])]
```

In [4]:

```
# Cell F-ZIR4 Zero forcing sets
#
# Zero forcing number Z(G)
def Z(g):
    z = len(zero_forcing_set_bruteforce(g))
    return z
```

```
# All zero forcing sets of G of size k
# input: a graph G and a positive integer k
# output: a list of all zero forcing sets of G of size k
def ZFsets(G,k):
    ord=G.order()
    V = G.vertices()
    AllSubs = set(Subsets(V,k))
    ZF=[]
    for s in AllSubs:
        A = zerosgame(G,s)
        if len(A) == ord:
            ZF.append(s)
    return ZF

def all_ZFsets(G,k):
    S = []
    for i in [Z(G)..k]:
        ZFS_size_i = ZFsets(G,i)
        S = S + ZFS_size_i
    return S
```

In [5]:
```
# Cell F-ZIR5
#
# Input: list of lists/sets/tuples
# Output: list without super-lists/sets/tuples

def get_minimal_subsets(sets):
    out = [x for x in sets if not any(set(x).issuperset(set(y)) and len(x)>l
    return out


# Input: graph G
# Output: all minimal zero forcing sets

def ZF_min_sets(G):
    ord = G.order()
    ZFS = all_ZFsets(G,ord)
    mZFS = get_minimal_subsets(ZFS)
    return mZFS


# Input: graph G
# Output: the zero forcing number and upper minimal zero forcing number

def zbar(G):
    min_list = ZF_min_sets(G)
    Zb = max(len(x) for x in min_list)
    return Zb
```

In [ ]:
```

```