Power Domination Reconfiguration

last edited Sep 30, 2021, 11:05:54 AM by reinh196


```
####################################
# This Sage worksheet is written by Chassidy Bozeman and Carolyn Reinhart
# using code written by numerous other people.
# The focus of this worksheet is token addition and removal (TAR) and token jumping (TJ)
# reconfiguration graphs for power domination in support of the paper
# Power domination reconfiguration
# by
# Beth Bjorkman, Chassidy Bozeman, Daniela Ferrero, Mary Flagg,
# Cheryl Grood, Leslie Hogben, Bonnie Jacob, Carolyn Reinhart
#
# To run computations, you must first enter the function F- cells at the beginning
####################################
```

```
# Functions
```

```
# F-1 load main minimum rank/maximum nullity and zero forcing functions
# zero forcing, PSD forcing, and minimum rank code
# developed by S. Butler, L. DeLoss, J. Grout, H.T. Hall, J. LaGrange, J.C.-H. Lin,T. McKay,
J. Smith, G. Tims
# updated and maintained by Jephian C.-H. Lin
URL='https://raw.githubusercontent.com/jephianlin/mr_JG/py2/'
files=['Zq_c.pyx','Zq.py','zero_forcing_64.pyx','zero_forcing_wavefront.pyx','minrank.py',
'inertia.py']
for f in files:
    print("Loading %s..."%f);
    load(URL+f)
```

```
    Loading Zq_c.pyx...
    Compiling
    /home/sageuser/3/.sage/temp/sage.math.iastate.edu/22821/tmp_WmHttP.pyx..\
    .
    Loading Zq.py...
    Loading zero_forcing_64.pyx...
    Compiling
    /home/sageuser/3/.sage/temp/sage.math.iastate.edu/22821/tmp_rENy_W.pyx..\
    .
    Loading zero_forcing_wavefront.pyx...
    Compiling
    /home/sageuser/3/.sage/temp/sage.math.iastate.edu/22821/tmp_JlJf6S.pyx..\
    .
    Loading minrank.py...
    Loading inertia.py...
```

```
# F-2 Power domination functions
# by Brian Wissman

# input: a graph G and a subset of its vertices V
# output: true/false depending if V is a power dominating set of G

def isPowerDominatingSet(G,V):
    N=[]
    for i in V:
        N+=G.neighbors(i)
    NVert=uniq(N+list(V))

    A=zerosgame(G,NVert)
    if len(A)==G.order():
        return True
    else:
        return False

# input: a graph G
# output: a power dominating set of G that achieves the power domination number
def minPowerDominatingSet(G):
    V = G.vertices()
    for i in range(1,len(V)+1):
        S = subsets(V,i)
        for s in S:
            if isPowerDominatingSet(G,s):
                return s

# input: a graph G
# output: the power domination number of G
def PowerDom(G):
    V = G.vertices()
    for i in range(1,len(V)+1):
        S = subsets(V,i)
        for s in S:
            if isPowerDominatingSet(G,s):
                p=len(s)
                return p
```

```
# F-3 power dominating sets
# input: a graph G and a positive integer k
# output: a list of all power dominating sets G of size k
def PDsets(G,k):
    ord=G.order()
    V = G.vertices()
    S = subsets(V,k)
    PDS=[]
    for s in S:
        if isPowerDominatingSet(G,s):
            PDS.append(s)
    return PDS
```

```
# F-4 Token Addition and TAR and
# by Chassidy Bozeman
```

```
# by Chassidy Bozeman



# input: A graph G and a positive integer k
#output: All power dominating sets G up to size k


def PDS_up_to_size_k(G,k):
    S=[]
    for i in range(1,k+1):
        PDS_size_i=PDsets(G,i)
        S=S+PDS_size_i
    return S


#input: A graph G and a positive integer k
#output: The TAR PD reconfiguration graph on power dominating set up to size k.

def TAR_reconfig(G,k):

    S=PDS_up_to_size_k(G,k)
    #creates a list of all dominating sets up to size k.

    H=Graph()
    #creates empty graph


    H.add_vertices(S)
    # add power dom sets up to size k to the vertices of H


    # The following part determines if the size of two power dominating sets differ
    # by one and if one is a subset of the other. If both are true, an edge is added.


    for i in range(len(S)):
        for j in range(i+1, len(S)):
            if len(S[i])-len(S[j]) in {-1,1}:
                    if set(S[i]).issubset(set(S[j])):
                        H.add_edge(S[i],S[j])
                    else:
                        if set(S[j]).issubset(set(S[i])):
                            H.add_edge(S[i],S[j])
    return H


# Token Addition, Remova, Exchange TARE(Exchange=Jumping)
#input: A graph G and integer k
#output: The TARE PD reconfiguration graph on PDS of size up to k


def TARE_reconfig(G,k):
    # The first part of this code is the same as that given for TAR above.

    S=PDS_up_to_size_k(G,k)
    H=Graph()
    H.add_vertices(S)
    for i in range(len(S)):
        for j in range(i+1,len(S)):
            if len(S[i])-len(S[j]) in {-1,1}:
```

```
            if len(S[i])-len(S[j]) in {-1,1}:
                    if set(S[i]).issubset(set(S[j])):
                        H.add_edge(S[i],S[j])
                    else:
                        if set(S[j]).issubset(set(S[i])):
                            H.add_edge(S[i],S[j])

                # The part below is for token exchange. If two power dominating sets have the
                # same size and they differ by one element, then an edge is added between
them.

            if len(S[i])==len(S[j]):
                    if len(set(S[i]).difference(set(S[j])))==1:
                        H.add_edge(S[i],S[j])
    return H

# Token Jumping = Token Exchange
def TE_reconfig(G,k):

    S=PDsets(G,k)
    #creates a list of all dominating sets of size k.

    H=Graph()
    #creates empty graph


    H.add_vertices(S)
    # add power dom sets of size k to the vertices of H

    # The part below is for token exchange. If two power dominating sets have the
    # same size and they differ by one element, then an edge is added between them.

    for i in range(len(S)):
        for j in range(i+1, len(S)):
            if len(S[i])==len(S[j]):
                if len(set(S[i]).difference(set(S[j])))==1:
                        H.add_edge(S[i],S[j])

    return H
```

```
# Token Addition and Removal Reconfiguration and Complete Bipartite Graphs
```

```
def TAR_reconfig_comparison(G):
    examples = set([])
    n=G.order()
    H=TAR_reconfig(G,n)
    for g in graphs(n):
        K=TAR_reconfig(g,n)
        if H.is_isomorphic(K):
            if not g.is_isomorphic(G):
                print 'Found one'
                examples.add(g.graph6_string())
    return examples
```

```
G=graphs.CompleteBipartiteGraph(3,3)
TAR_reconfig_comparison(G)
```

```
    set()
```

```
G=graphs.CompleteBipartiteGraph(3,4)
TAR_reconfig_comparison(G)
```

```
    set()
```

```
G=graphs.CompleteBipartiteGraph(3,5)
TAR_reconfig_comparison(G)
```

```
    set()
```

```
G=graphs.CompleteBipartiteGraph(4,4)
TAR_reconfig_comparison(G)
```

```
    set()
```

```
G=graphs.CompleteBipartiteGraph(4,5)
TAR_reconfig_comparison(G)
```

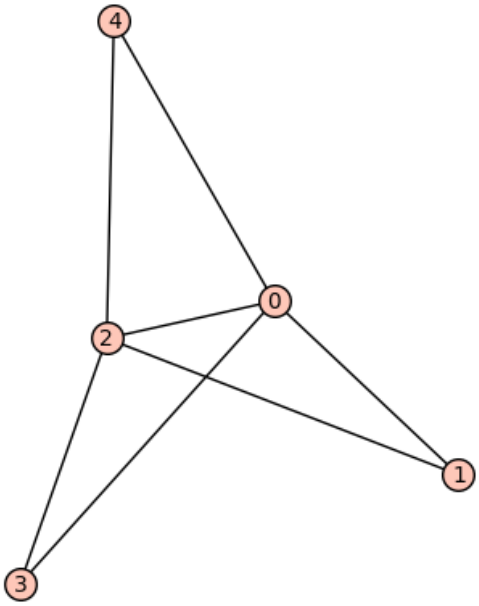
```
# As a comparision, this program finds the one graph G=K_2,3+e that is not nonisomorphic to
K_2,3
# but has TAR(G) isomorphic to TAR(K_2,3).
```

evaluate

```
G=graphs.CompleteBipartiteGraph(2,3)
TAR_reconfig_comparison(G)
```

```
    Found one
    {'D|g'}
```

```
g=Graph("D|g")
show(g)
```




```
# Token Jumping Reconfiguration and Grid Graphs
```

```
G5=graphs.GridGraph([5,12])
H5=TE_reconfig(G5,2)
H5.is_connected()
```

```
False
```

```
H5.connected_components()
```

```
WARNING: Output truncated!
full_output.txt


[[((0, 0), (0, 3)),
  ((0, 0), (1, 2)),
  ((0, 0), (2, 1)),
  ((0, 0), (3, 0)),
  ((0, 0), (4, 1)),
  ((0, 1), (0, 3)),
  ((0, 1), (0, 4)),
  ((0, 1), (1, 2)),
  ((0, 1), (1, 3)),
  ((0, 1), (2, 1)),
  ((0, 1), (2, 2)),
  ((0, 1), (3, 0)),
  ((0, 1), (3, 1)),
  ((0, 1), (4, 0)),
  ((0, 1), (4, 1)),
  ((0, 2), (1, 2)),
  ((0, 2), (2, 1)),
  ((0, 2), (2, 3)),
  ((0, 3), (1, 0)),
  ((0, 3), (1, 1)),
```

```
     ((0, 3), (1, 2)),
     ((0, 3), (2, 2)),
     ((0, 4), (1, 0)),
     ((0, 4), (1, 2)),
     ((0, 4), (2, 3)),
     ((1, 0), (1, 2)),
     ((1, 0), (1, 3)),
     ((1, 0), (2, 1)),
     ((1, 0), (2, 2)),
     ((1, 0), (3, 0)),
     ((1, 0), (3, 1)),
     ((1, 0), (4, 0)),
     ((1, 0), (4, 1)),
     ((1, 1), (1, 2)),
     ((1, 1), (2, 1)),
     ((1, 1), (2, 3)),
     ((1, 1), (3, 0)),
     ((1, 1), (3, 2)),
     ((1, 1), (4, 1)),
     ((1, 2), (1, 3)),
     ((1, 2), (2, 0)),
     ((1, 2), (2, 4)),
     ((1, 2), (3, 1)),
     ((1, 3), (2, 1)),
     ((1, 4), (2, 2)),
     ((1, 5), (2, 3)),
     ((2, 0), (2, 1)),
     ((2, 0), (3, 2)),
     ((2, 1), (3, 0)),
     ((2, 1), (3, 1)),
     ((2, 1), (3, 3)),
     ((2, 1), (4, 0)),
     ((2, 1), (4, 1)),
     ((2, 1), (4, 2)),
     ((2, 2), (3, 0)),
     ((2, 2), (3, 4)),
     ((2, 2), (4, 1)),
     ((2, 2), (4, 3)),
     ((2, 3), (3, 1)),

...

     ((0, 10), (4, 10)),
     ((0, 10), (4, 11)),
     ((0, 11), (1, 9)),
     ((0, 11), (2, 10)),
     ((0, 11), (3, 11)),
     ((0, 11), (4, 10)),
     ((1, 6), (2, 8)),
     ((1, 7), (2, 9)),
     ((1, 8), (1, 9)),
     ((1, 8), (1, 11)),
     ((1, 8), (2, 10)),
     ((1, 9), (1, 10)),
     ((1, 9), (1, 11)),
     ((1, 9), (2, 7)),
     ((1, 9), (2, 11)),
     ((1, 9), (3, 10)),
     ((1, 10), (2, 8)),
```

```
      ((1, 10), (2, 10)),
      ((1, 10), (3, 9)),
      ((1, 10), (3, 11)),
      ((1, 10), (4, 10)),
      ((1, 11), (2, 9)),
      ((1, 11), (2, 10)),
      ((1, 11), (3, 10)),
      ((1, 11), (3, 11)),
      ((1, 11), (4, 10)),
      ((1, 11), (4, 11)),
      ((2, 7), (3, 9)),
      ((2, 8), (3, 6)),
      ((2, 8), (3, 10)),
      ((2, 8), (4, 7)),
      ((2, 8), (4, 9)),
      ((2, 9), (3, 7)),
      ((2, 9), (3, 11)),
      ((2, 9), (4, 8)),
      ((2, 9), (4, 10)),
      ((2, 10), (2, 11)),
      ((2, 10), (3, 8)),
      ((2, 10), (3, 10)),
      ((2, 10), (3, 11)),
      ((2, 10), (4, 9)),
      ((2, 10), (4, 10)),
      ((2, 10), (4, 11)),
      ((2, 11), (3, 9)),
      ((3, 8), (3, 9)),
      ((3, 8), (3, 11)),
      ((3, 8), (4, 10)),
      ((3, 9), (3, 10)),
      ((3, 9), (3, 11)),
      ((3, 9), (4, 7)),
      ((3, 9), (4, 8)),
      ((3, 9), (4, 9)),
      ((3, 9), (4, 10)),
      ((3, 9), (4, 11)),
      ((3, 10), (4, 8)),
      ((3, 11), (4, 7)),
      ((3, 11), (4, 8)),
      ((4, 7), (4, 10)),
      ((4, 8), (4, 10)),
      ((4, 8), (4, 11))]]
```

full_output.txt

```
G6=graphs.GridGraph([6,12])
H6=TE_reconfig(G6,2)
H6.is_connected()
```

```
False
```

```
H6.connected_components()
```

```
[[((0, 1), (0, 4)),
  ((0, 1), (1, 3)),
  ((0, 1), (2, 2)),
  ((0, 1), (3, 1)),
  ((0, 1), (4, 0)),
  ((0, 1), (5, 1)),
```

```
      ((0, 2), (2, 3)),
      ((0, 3), (2, 2)),
      ((0, 4), (1, 0)),
      ((0, 4), (1, 2)),
      ((0, 4), (2, 3)),
      ((1, 0), (1, 3)),
      ((1, 0), (2, 2)),
      ((1, 0), (3, 1)),
      ((1, 0), (4, 0)),
      ((1, 0), (5, 1)),
      ((1, 1), (2, 3)),
      ((1, 1), (3, 2)),
      ((1, 2), (1, 3)),
      ((1, 2), (2, 4)),
      ((1, 2), (3, 1)),
      ((1, 3), (2, 1)),
      ((1, 4), (2, 2)),
      ((1, 5), (2, 3)),
      ((2, 0), (3, 2)),
      ((2, 1), (3, 1)),
      ((2, 1), (4, 0)),
      ((2, 1), (4, 2)),
      ((2, 1), (5, 1)),
      ((2, 2), (3, 0)),
      ((2, 2), (4, 1)),
      ((3, 1), (4, 3)),
      ((3, 2), (4, 0)),
      ((3, 2), (4, 4)),
      ((3, 2), (5, 1)),
      ((3, 2), (5, 3)),
      ((3, 3), (4, 1)),
      ((3, 3), (4, 5)),
      ((3, 3), (5, 2)),
      ((3, 3), (5, 4)),
      ((3, 4), (4, 2)),
      ((4, 0), (4, 3)),
      ((4, 0), (5, 4)),
      ((4, 2), (4, 3)),
      ((4, 2), (5, 4)),
      ((4, 3), (5, 1)),
      ((5, 1), (5, 4))],
     [((0, 7), (0, 10)),
      ((0, 7), (1, 9)),
      ((0, 7), (1, 11)),
      ((0, 7), (2, 8)),
      ((0, 8), (2, 9)),
      ((0, 9), (2, 8)),
      ((0, 10), (1, 8)),
      ((0, 10), (2, 9)),
      ((0, 10), (3, 10)),
      ((0, 10), (4, 11)),
      ((0, 10), (5, 10)),
      ((1, 6), (2, 8)),
      ((1, 7), (2, 9)),
      ((1, 8), (1, 9)),
      ((1, 8), (1, 11)),
      ((1, 8), (2, 10)),
      ((1, 9), (2, 7)),
      ((1, 9), (3, 10)),
```

```
      ((1, 10), (2, 8)),
      ((1, 10), (3, 9)),
      ((1, 11), (2, 9)),
      ((1, 11), (3, 10)),
      ((1, 11), (4, 11)),
      ((1, 11), (5, 10)),
      ((2, 9), (3, 11)),
      ((2, 9), (4, 10)),
      ((2, 10), (3, 10)),
      ((2, 10), (4, 9)),
      ((2, 10), (4, 11)),
      ((2, 10), (5, 10)),
      ((2, 11), (3, 9)),
      ((3, 7), (4, 9)),
      ((3, 8), (4, 6)),
      ((3, 8), (4, 10)),
      ((3, 8), (5, 7)),
      ((3, 8), (5, 9)),
      ((3, 9), (4, 7)),
      ((3, 9), (4, 11)),
      ((3, 9), (5, 8)),
      ((3, 9), (5, 10)),
      ((3, 10), (4, 8)),
      ((4, 8), (4, 9)),
      ((4, 8), (4, 11)),
      ((4, 8), (5, 10)),
      ((4, 9), (5, 7)),
      ((4, 11), (5, 7)),
      ((5, 7), (5, 10))]]
```

```
G7=graphs.GridGraph([7,12])
H7=TE_reconfig(G7,2)
H7.is_connected()
```

```
False
```

```
H7.connected_components()
```

```
[[((0, 7), (2, 8)),
  ((0, 9), (2, 8)),
  ((1, 6), (2, 8)),
  ((1, 10), (2, 8)),
  ((1, 10), (3, 9)),
  ((2, 11), (3, 9)),
  ((3, 9), (4, 11)),
  ((3, 9), (5, 10)),
  ((4, 8), (5, 6)),
  ((4, 8), (5, 10)),
  ((4, 8), (6, 7)),
  ((4, 8), (6, 9))],
 [((0, 2), (2, 3)),
  ((0, 4), (2, 3)),
  ((1, 1), (2, 3)),
  ((1, 1), (3, 2)),
  ((1, 5), (2, 3)),
  ((2, 0), (3, 2)),
  ((3, 2), (4, 0)),
  ((3, 2), (5, 1)),
  ((4, 3), (5, 1)),
```

```
      ((4, 3), (5, 5)),
      ((4, 3), (6, 2)),
      ((4, 3), (6, 4))],
     [((2, 1), (4, 2)), ((4, 2), (5, 4))],
     [((1, 7), (2, 9)), ((2, 9), (4, 10))],
     [((2, 10), (4, 9)), ((4, 9), (5, 7))],
     [((1, 4), (2, 2)), ((2, 2), (4, 1))],
     [((1, 2), (2, 4))],
     [((4, 4), (5, 2))],
     [((4, 7), (5, 9))],
     [((1, 9), (2, 7))]]
```

```
G8=graphs.GridGraph([8,12])
H8=TE_reconfig(G8,2)
H8.is_connected()
```

```
False
```

```
H8.connected_components()
```

```
[[((5, 4), (6, 2))],
 [((1, 9), (2, 7))],
 [((5, 8), (6, 6))],
 [((2, 1), (4, 2))],
 [((3, 9), (5, 10))],
 [((1, 5), (2, 3))],
 [((1, 6), (2, 8))],
 [((3, 2), (5, 1))],
 [((2, 10), (4, 9))],
 [((5, 3), (6, 5))],
 [((1, 2), (2, 4))],
 [((5, 7), (6, 9))]]
```

```
```