# Lemma 3.1 Proof Code

Austin Roberts

2/13/2016

We begin by introducing the key functions needed to generate the desired graphs.

```
%cython
from sage.all import Graph, copy, Partitions

######################################
##Standard Dual Equivalence Functions##
######################################

###Elementary Dual Equivalence###
        #####only defined for permutations####
def DualEq(list x, int i):
    cdef list y,w

    y=copy(x)
    a=y.index(i-1)
    b=y.index(i)
    c=y.index(i+1)

    if a<b<c or a>b>c:
            return y

    elif b<c<a or b>c>a:
            del(y[b]); y.insert(b,i-1)
            del(y[a]); y.insert(a,i)
            return y

    else:     #y[i-1]<y[i-2]<y[i] or y[i-1]>y[i-2]>y[i]
            del(y[b]); y.insert(b,i+1)
            del(y[c]); y.insert(c,i)
            return y



########################################################
```

```
#####D Version of Dual Equivalence for Assaf grapns######
##########################################################

##Common error: all inputs must be permutations, not just a words.
###Non-Standard Def!. 'content' defines when to use d^~, when all 3 \
    entries are in a range [j,content[j]]###

def DualEq_D(list x,int i,list content):
    cdef list y

    y=copy(x)
    a=y.index(i-1)
    b=y.index(i)
    c=y.index(i+1)

    if a<b<c or a>b>c:
            return y

    elif b<c<a or b>c>a:
        if min(content[a],content[b]) <= max(a,b):
            del(y[b]); y.insert(b,i-1)
            del(y[a]); y.insert(a,i)
            return y

        else:
            del(y[c]); y.insert(c,i-1)
            del(y[b]); y.insert(b,i+1)
            del(y[a]); y.insert(a,i)
            return y

    else:
        if min(content[b],content[c]) <= max(b,c):
            del(y[b]); y.insert(b,i+1)
            del(y[c]); y.insert(c,i)
            return y

        else:
            del(y[c]); y.insert(c,i)
            del(y[b]); y.insert(b,i-1)
            del(y[a]); y.insert(a,i+1)
            return y



#####################################################
#######Graph from permutation and edge function######
#####################################################
```

```
def Graph_From_Function(Edge, Vertex, Label=copy, int Degree=3):

    ## This function makes graphs. It takes a function to define \
   edges and a vertex. There are further options about how the graph\
    is labeled and the degree of the graph.

    cdef list New_Vertices

    ##With Doubled Edges From a Vertex##
    G=Graph(multiedges= True); G.add_vertex(Label(Vertex))
    New_Vertices=[Vertex]
    while len(New_Vertices) > 0:
        Length = len(New_Vertices)
        for X in New_Vertices:
            for i in range(2,Degree):
                u=Edge(X,i)
                v=Label(u)
                w=Label(X)
                if v not in G.vertices():
                    New_Vertices.append(u)
                    G.add_edge(v,w,i)
                if (w, v, i) not in G.edges() and (v, w, i) not in G\
   .edges():
                    G.add_edge(v,w,i)
        for i in range(1,Length+1):
            del(New_Vertices[0])
    return G


####################
#Graphing Functions##
####################

def DEG(list x):
    return Graph_From_Function(DualEq,Vertex=x,Label=tuple,Degree=\
   len(x))

def DEG_D(list x,list content):
    def F(x,i):
        return DualEq_D(x,i,content)
    return Graph_From_Function(F,Vertex=x,Label=tuple,Degree=len(x))


#\
```

```
    #######################################################################\

##Functions for creating representative words of equivalence classes\
    ##
#\
    #######################################################################\


def Rep_Word(P):    #Input Partition, return maximal word (reading \
    word of U_\lambda).
     cdef list Word
     cdef int Size, Length

     Word=[]; Size=sum(P); Length=len(P)
     for i in range(1, len(P)+1):
         for k in range(Size-P[Length-i]+1, Size+1):
             Word.append(k)
         Size=Size-P[Length-i]
     return Word

def All_Rep_Words(int n):  ##Gives all representative reduced words \
    for each partition on n
     cdef list Words
     Words=[]
     for x in Partitions(n):
         Words.append(Rep_Word(x))
     return Words
```

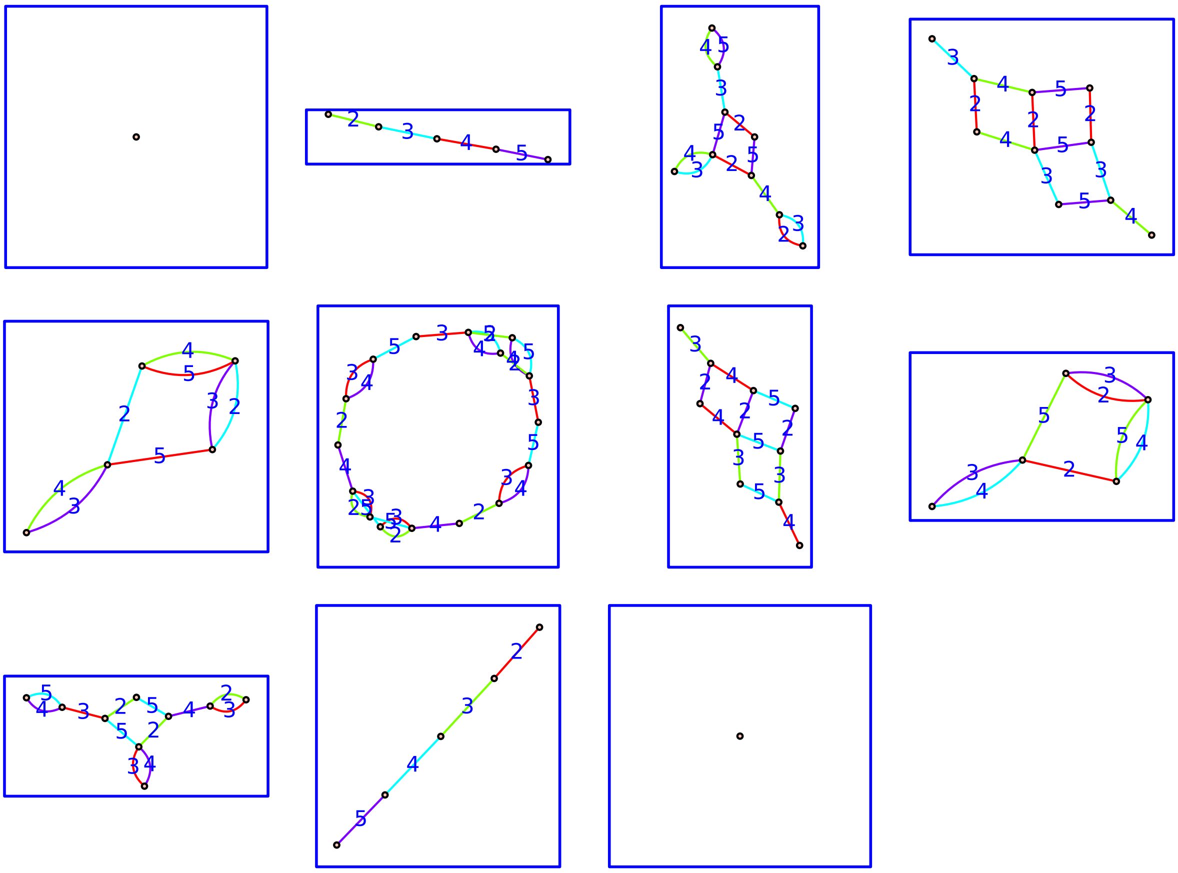
Auto-generated code...

Next we generate all standard dual equivalence graphs whose partition type has size 6. To do this, it suffices to generate the graphs including the reading words of U_\lambda. The function All_Rep_Words generates these reading words.

```
DEGs6=[]
for w in All_Rep_Words(6):
        G=DEG(w)
        DEGs6.append(G)
```

These standard dual equivalence graphs are shown below.

```
graphs_list.show_graphs(DEGs6, color_by_label=true, edge_labels=true\
    , layout='spring', vertex_size=6)
```



Next, we need to create a function to detect if a permutation contains a strict pattern. To do this, we again allow 'content' to encode which entries share a pistol.

```
%cython
def has_strict_pattern(list w, list content):
    cdef list pat

    #Check two shorter patterns 1342, 2341
    for i in range(1,4):
        pat=[w.index(i), w.index(i+2), w.index(i+3), w.index(i+1)]
        if pat[0]<pat[1]<pat[2]<pat[3]:
            if content[pat[0]] >= pat[3]:
                return True
        pat=[w.index(i+1), w.index(i+3), w.index(i+2), w.index(i)]
        if pat[0]<pat[1]<pat[2]<pat[3]:
            if content[pat[0]] >= pat[3]:
                return True

    #Check second pattern 12543, 34521
    for i in range(1,3):
        pat=[w.index(i),w.index(i+1),w.index(i+4),w.index(i+3),w.\
```

```
    index(i+2)]
          if pat[0]<pat[1]<pat[2]<pat[3]<pat[4]:
               if content[pat[0]]>=pat[3] and content[pat[1]]>=pat[4] \
    and content[pat[0]]<pat[4]:
                    return True
          pat=[w.index(i+2),w.index(i+3),w.index(i+4),w.index(i+1),w.\
    index(i)]
          if pat[0]<pat[1]<pat[2]<pat[3]<pat[4]:
               if content[pat[0]]>=pat[3] and content[pat[1]]>=pat[4] \
    and content[pat[0]]<pat[4]:
                    return True

     return False
```

Auto-generated code...

Finally, we may go through all all possible permutations of size 6 and 'content' to check if they contain any of our strict patterns. If they do not, we may generate the graph and ensure that it is a dual equivlalence graph by checking it against the list we made above. Notice that, without loss of generality, we have assumed pistols contain at least two cells.

```
for i in range(1,6):
     if i==4:
          break
     p
```

```
Bad_perms=[]
for p in Permutations(6):
     p=list(p)
     for a in range(2,7): #generate all contents [a,b,c,d,6,6]
          for b in range(max(3,a),7):
               for c in range(max(4,b),7):
                    for d in range(max(5,c),7):
                         if has_strict_pattern(p,[a,b,c,d,6,6])==False:
                              G=DEG(p)
                              Check=False
                              for H in DEGs6:
                                   if G.is_isomorphic(H)==True:
                                        Check=True
                                        break
                              if Check==False:
                                   Bad_perms.append(p)
                                   print "The lemma is false."

if Bad_perms==[]: print "The lemma is True."
```

```
The lemma is True.
```