AIMS DMG Sage Demonstration 20
AIMS 2013-14: Designs, Matroids and Graphs

Rob Beezer
University of Puget Sound

Nancy Neudauer
Pacific University

January 20, 2014

In this chapter, we begin with the complete graph on six vertices, K_6 and construct three diverse and remarkable combinatorial structures. This material follows Chapter 6, "A property of the number six" of "Designs, Graphs, Codes and their Links" by P.J. Cameron and J.H. van Lint, who in turn credit lectures of G. Higman. We begin with the vertices and edges of the complete graph, and then create sets of three edges known as 1-factors. Collections of 1-factors become factorizations of the original complete graph. Vertices, edges, factors and factorizations then all combine to create our three ultimate combinatorial structures.

1 Factors of K_6

A **factor** of a graph, or more precisely a **1-factor**, is a subgraph which includes every vertex and is regular of degree 1. For K_6 this is a collection of 3 edges with no common vertices.

We build K_6 first. We will work with sets of sets, so we convert the edges of K_6 into two-element sets. Note that the vertices are integers, *starting with zero*. Also, we use the Sage constructor `Set()` rather than the Python version (`set()`), since these behave differently and we will have an easier time with the Sage version. Of course, K_6 has 6 vertices and $\binom{6}{2} = 15$ edges. We will see these numbers repeatedly throughout this chapter.

```
K6 = graphs.CompleteGraph(6)
vertices = K6.vertices()
vertices
```

```
[0, 1, 2, 3, 4, 5]
```

```
edges = [Set(e) for e in K6.edges(labels=False)]
edges
```

```
[{0, 1}, {0, 2}, {0, 3}, {0, 4}, {0, 5},
{1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3},
{2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}]
```

We will give each of the 15 edges a name (an arbitrary lowercase letter) which will make some of our graphs a bit more tidy when displaying them. A Python dictionary (mapping) is a natural way to specify this.

```
edge_names = dict(zip(edges, [chr(i) for i in range(ord('a'), ord('a')+15)]))
edge_names  # not tested
```

Since K_6 includes every possible edge, a factor is partition of the vertex set into a set of three 2-sets. We discover 15 factors.

```
factors = SetPartitions(vertices,[2,2,2]).list()
factors = [Set(list(A)) for A in factors]
factors
```

```
[{{0, 1}, {2, 3}, {4, 5}},
 {{2, 4}, {3, 5}, {0, 1}},
 {{3, 4}, {2, 5}, {0, 1}},
 {{1, 3}, {0, 2}, {4, 5}},
 {{0, 2}, {1, 4}, {3, 5}},
 {{3, 4}, {1, 5}, {0, 2}},
 {{1, 2}, {0, 3}, {4, 5}},
 {{1, 4}, {0, 3}, {2, 5}},
 {{2, 4}, {1, 5}, {0, 3}},
 {{1, 2}, {0, 4}, {3, 5}},
 {{1, 3}, {0, 4}, {2, 5}},
 {{0, 4}, {2, 3}, {1, 5}},
 {{1, 2}, {0, 5}, {3, 4}},
 {{1, 3}, {2, 4}, {0, 5}},
 {{0, 5}, {2, 3}, {1, 4}}]
```

```
len(factors)
```

```
15
```

We can easily visualize the 15 different factors by considering each factor as an edge-induced subgraph of K_6. We build a list of these subgraphs and use a convenient utility in Sage to display them all simply and compactly. These 15 objects will be critical in what follows.

```
factor_graphs = [Graph([e.list() for e in f]) for f in factors]
graphs_list.show_graphs(factor_graphs)
```

The edge-factor incidence graph is a remarkable graph. The Tutte 8-cage is the cubic graph (i.e. regular of degree 3) with girth 8 and having the fewest possible number of vertices. It therefore also qualifies as a Moore graph. Since it is distance-transitive, it is also distance-regular.

We use the set of edges and the set of factors as the vertex set of a graph. We join an edge to a factor if the edge is a component of the factor.

```
T = Graph([(e, f) for e in edges for f in factors if e in f])
T.num_verts(), T.num_edges(), T.is_bipartite(), T.is_regular(), T.average_degree(), T.girth
```

```
(30, 45, True, True, 3, 8)
```

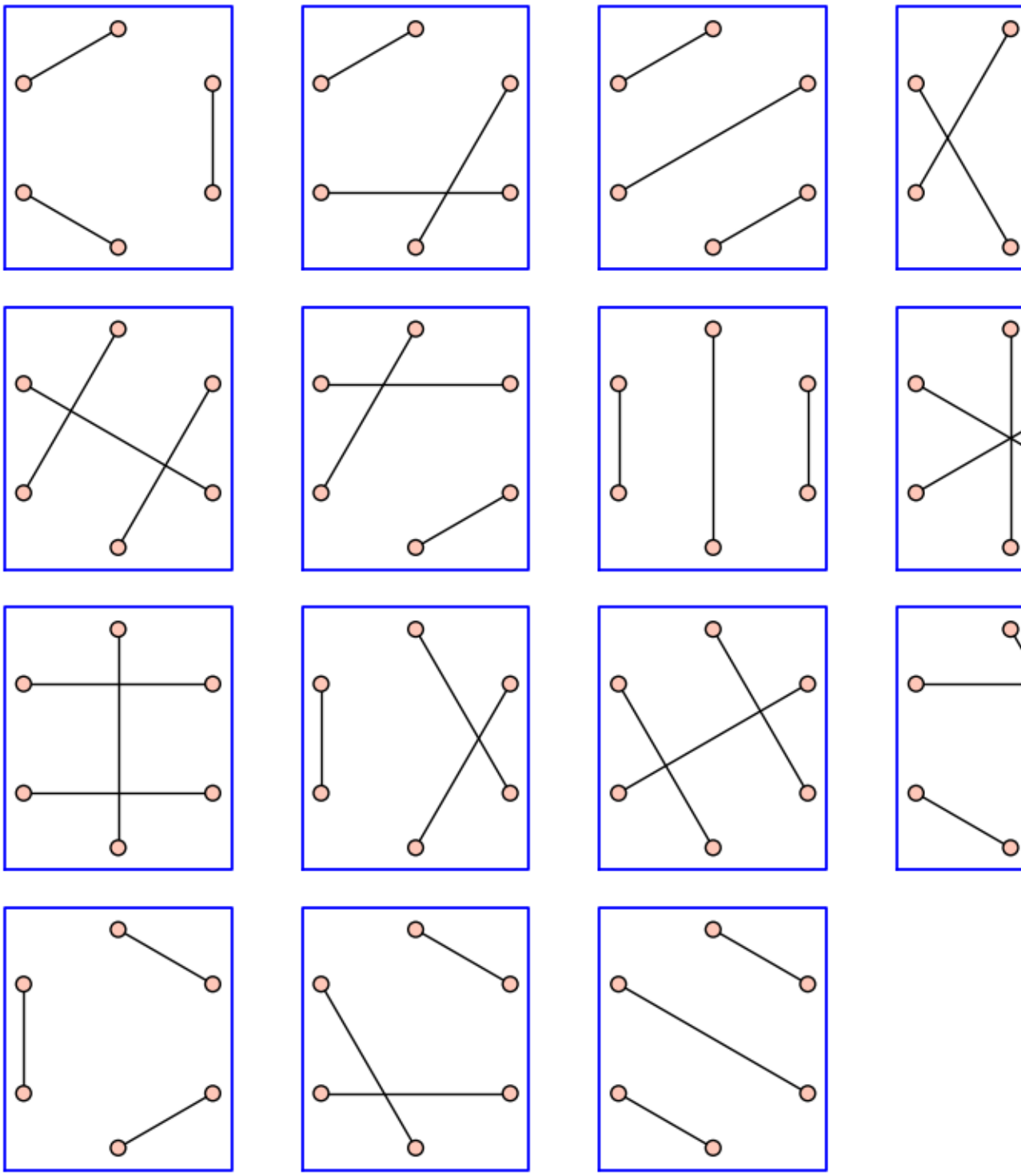

Figure 1: Fifteen one-factors of K_6

```
T.is_isomorphic(graphs.TutteCoxeterGraph())
```

```
True
```

Since this graph is built into Sage, we can get a much more pleasing drawing from the embedding that is hard-coded. If we give our factors names (uppercase letters from the start of the alphabet), then we can relabel the vertices of a copy of the graph and use Sage's `BipartiteGraph` class to get another informative drawing.

```
graphs.TutteCoxeterGraph().plot()
```

```
factor_names = dict(zip(factors, [chr(i) for i in range(ord('A'), ord('a')+15)]))
factor_names  # not tested
```

```
TB = copy(T)
TB.relabel(edge_names)
TB.relabel(factor_names)
BipartiteGraph(TB).plot()
```

2 Factorizations of K_6

A **factorization**, or more precisely a **1-factorization**, is a partition of the edge set of a graph into 1-factors. For K_6 this is a partition of the $\binom{6}{2} = 15$ edges into 5 factors with 3 edges each.

We will see that K_6 has six distinct factorizations, each isomorphic to the other. Furthermore, each factor is contained in exactly two of these factorizations, and any pair of factorizations have exactly one factor in common. So given a factor we can determine a pair of factorizations and given a pair of factorizations we can determine a factor. These observations will be critical in the following, especially the section on duality. These claims can be verified analytically — we will instead use Sage experimentally.

We begin by employing a graph to help us find the factorizations. The vertices of the graph are the factors of K_6, and we join two vertices if the factors are disjoint. Then a factorization of K_6 is a clique of size 5 in this graph, and there is no possibility of any larger clique because the edge set is exhausted by 5 disjoint factors. So factorizations of K_6 are maximal cliques in this graph, which we record as Sage sets.

```
factor_graph = Graph([(f1, f2) for f1, f2 in Subsets(factors, 2) if f1.intersection(f2).car
factorizations = factor_graph.cliques_maximum()
factorizations = [Set(f) for f in factorizations]
factorizations
```

```
[{{{2, 4}, {1, 5}, {0, 3}}, {{0, 1}, {2, 3}, {4, 5}}, {{0, 2}, {1, 4}, {3, 5}},
 {{1, 3}, {0, 4}, {2, 5}}, {{1, 2}, {0, 5}, {3, 4}}},
 {{{1, 2}, {0, 3}, {4, 5}}, {{0, 2}, {1, 4}, {3, 5}}, {{1, 3}, {2, 4}, {0, 5}},
 {{0, 4}, {2, 3}, {1, 5}}, {{3, 4}, {2, 5}, {0, 1}}},
 {{{1, 4}, {0, 3}, {2, 5}}, {{2, 4}, {3, 5}, {0, 1}}, {{1, 3}, {0, 2}, {4, 5}},
 {{0, 4}, {2, 3}, {1, 5}}, {{1, 2}, {0, 5}, {3, 4}}},
 {{{1, 4}, {0, 3}, {2, 5}}, {{0, 1}, {2, 3}, {4, 5}}, {{3, 4}, {1, 5}, {0, 2}},
 {{1, 3}, {2, 4}, {0, 5}}, {{1, 2}, {0, 4}, {3, 5}}},
```

```
{{{0, 5}, {2, 3}, {1, 4}}, {{2, 4}, {3, 5}, {0, 1}}, {{1, 3}, {0, 4}, {2, 5}},
{{3, 4}, {1, 5}, {0, 2}}, {{1, 2}, {0, 3}, {4, 5}}},
{{{2, 4}, {1, 5}, {0, 3}}, {{3, 4}, {2, 5}, {0, 1}}, {{1, 3}, {0, 2}, {4, 5}},
{{1, 2}, {0, 4}, {3, 5}}, {{0, 5}, {2, 3}, {1, 4}}}]
```

To visualize factorizations, it helps to look at a drawing of K_6 with edges colored according to the factor they belong to. We build an ad-hoc set of fifteen contrasting colors as RGB triples and assign them to the factors with a Python dictionary. We will consistently use this color assignment in the remainder.

```
twentyseven_colors = [triple for triple in CartesianProduct([0,0.5, 1], [0,0.5, 1], [0,0.5,
contrast_colors = [tuple(twentyseven_colors[i]) for i in [0,1,3,5,7,8,9,11,13,18,19,21,22,2
factor_colors = dict(zip(factors, contrast_colors))
```

We now build and draw a copy of K_6 for each factorization, using the provided color for the edges of each factor in the factorization. We will employ the `one_K6_graphic()` function twice, so be sure to execute it first. We exercise some of the graphics primitives for a little more control over the design of our diagrams.

```
def one_K6_graphic(center, factorization):
    r'''Draw a single factorization of K6 at a center point'''
    global factor_colors
    radius = 300
    v_radius = 20
    points = [(center[0]+radius*cos(i*pi/3.0), center[1]+radius*sin(i*pi/3.0)) for i in ran
    vertices = [circle(p, v_radius, fill=True, facecolor='black', edgecolor='black', zorder
    edges = [line([points[e[0]], points[e[1]]], color=factor_colors[f], thickness=3, zorder
    mask = polygon2d(points, fill=True, rgbcolor=(1,1,1), zorder=2) # for combined plot
    return sum([mask] + edges + vertices) # in z-order
```

```
six_plots = [one_K6_graphic((0,0), f) for f in factorizations]
array = graphics_array(six_plots,3,2)
array.show(axes=false)
```

We made two claims above about factors and factorizations: any single factor is contained in exactly two factorizations, and any two factorizations share a single common factor. For each factor, we form a list of factorizations containing that factor, and then simply check that the size (length) of that list is 2 in each case. Then we examine all pairs of factorizations, determining the size of the set of common factors for each pair.

```
map(len, [[fz for fz in factorizations if f in fz] for f in factors])
```

```
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
map(len, [fz1.intersection(fz2) for fz1, fz2 in Subsets(factorizations, 2)])
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

As any two factorizations share a single factor, we can make the 6 factorizations the vertices of a graph, add every possible edge (a complete graph) and color each edge with the color of the factor shared by the two factorizations at the endpoints.

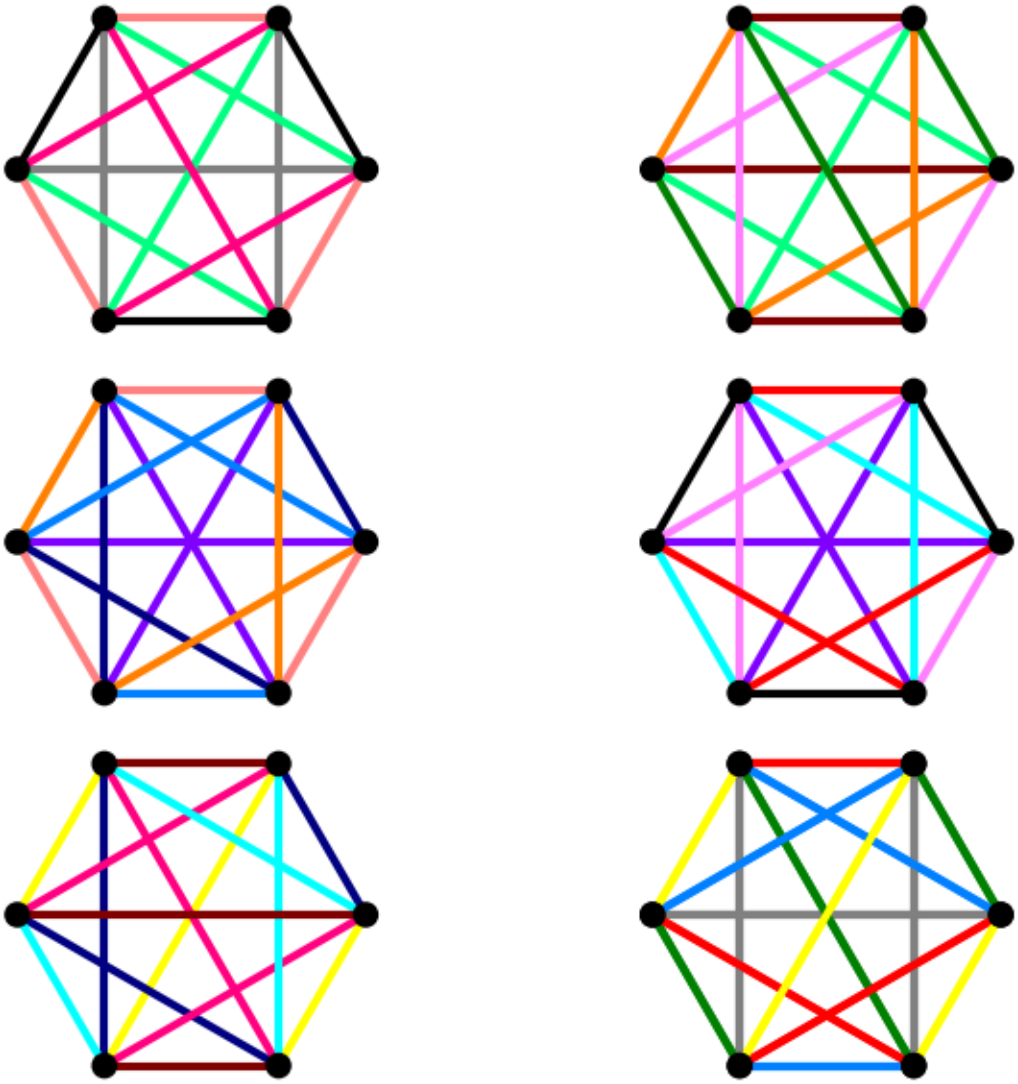

Figure 2: Factorizations of K_6

```
points = [(1000*cos(i*pi/3.0), 1000*sin(i*pi/3.0)) for i in range(6)]
nodes = [one_K6_graphic(p, f) for (p,f) in zip(points, factorizations)]
arcs = []
for i in range(6):
    for j in range(i+1, 6):
        common_factor = (factorizations[i].intersection(factorizations[j]))[0]
        color = factor_colors[common_factor]
        arcs.append(line([points[i], points[j]], thickness=5, color=color, zorder=1))
show(sum(arcs + nodes), axes=False) # in z-order
```

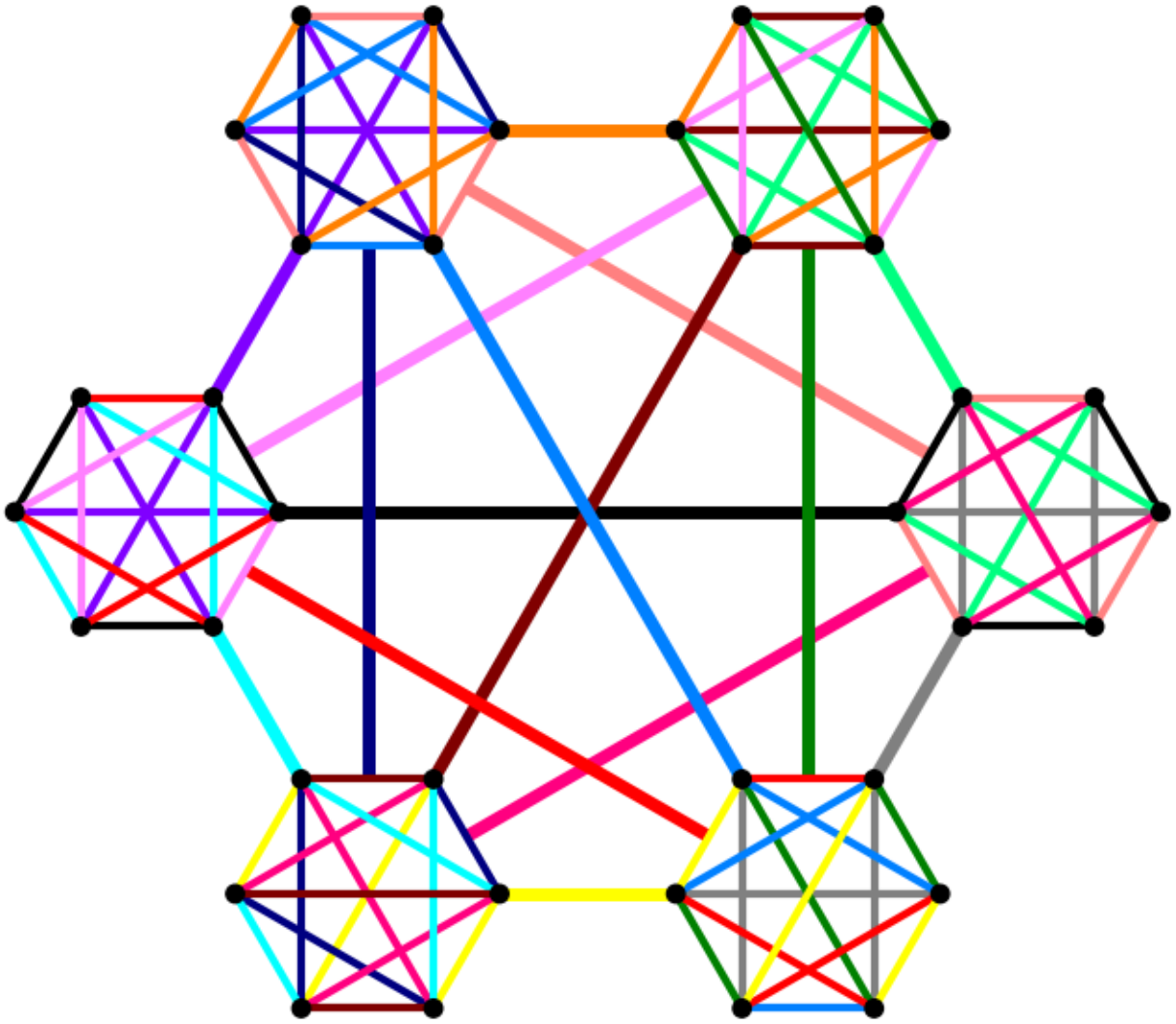

Figure 3: Factorization diagram for K_6

We will find it convenient later to have names for the factorizations — we will use uppercase letters from the end of the alphabet.

```
factorization_names = dict(zip(factorizations, [chr(i) for i in range(ord('U'), ord('a')+6)
factorization_names  # not tested
```

3 A Faithful Action on Factorizations

Automorphisms of a graph are permutations of the vertex set, which when applied to pairs of vertices, take edges to edges, and take non-edges to non-edges. Because every possible edge is present in K_6, *every* permutation of the vertices is an automorphism.

```
aut = SymmetricGroup(vertices)
aut
```

```
Symmetric group of order 6! as a permutation group
```

In this section we will show that the group of automorphisms of K_6, as a group action on the set of factorizations, is a faithful group action. This means that no nontrivial element of the group fixes all of the factorizations, or equivalently, the mapping from an automorphism to a permutation of the factorizations is injective. First, we define a group action on the edges of K_6, use it to define a group action on factors of K_6, and then define a group action on factorizations of K_6.

```
def permute_edge(sigma, e):
    '''Action of a vertex permutation on an edge.'''
    return Set([sigma(v) for v in e])

def permute_factor(sigma, f):
    '''Action of a vertex permutation on a factor.'''
    return Set([permute_edge(sigma, e) for e in f])

def permute_factorization(sigma, fz):
    '''Action of a vertex permutation on a factorization.'''
    return Set([permute_factor(sigma, f) for f in fz])
```

We give an illustration of this action with a 6-cycle from the automorphism group acting on the third factorization, with the result of the action being the fourth factorization. We accumulate the various items as strings in a list, then join them together to print in a readable way.

```
cycle = aut("(0,1,2,3,4,5)")
display = [str(cycle), "\n-----\n"]
fz = factorizations[2]
display += [str(fz), "\n-----\n"]
result = permute_factorization(cycle, fz)
display += [str(result), "\n-----\n"]
result_match = (result == factorizations[3])
display += [str(result_match)]
print ''.join(display)
```

```
(0,1,2,3,4,5)
-----
{{{1, 4}, {0, 3}, {2, 5}}, {{2, 4}, {3, 5}, {0, 1}},
```

```
{{1, 3}, {0, 2}, {4, 5}}, {{0, 4}, {2, 3}, {1, 5}},
{{1, 2}, {0, 5}, {3, 4}}}
-----
{{{1, 4}, {2, 5}, {0, 3}}, {{1, 2}, {0, 4}, {3, 5}},
{{3, 4}, {1, 5}, {0, 2}}, {{2, 4}, {1, 3}, {0, 5}},
{{0, 1}, {2, 3}, {4, 5}}}
-----
True
```

We can apply an automorphism to each factorization, creating a permutation of the factorizations. Employing the full permutation group of the factorizations, we check that the action only creates the identity permutation of the factorizations in the case of the identity automorphism. In other words, the mapping from the automorphism group of the graph to the permutation group of the factorizations is injective, which we describe by saying the action is faithful, or by saying the automorphism group acts "faithfully" on the factorizations.

We apply each of the $6! = 720$ automorphisms to each factorization, creating an ordered list of images which we use to define the resulting element of the permutation group of the factorizations. This permutation is checked to see if it is the identity permutation, and if so it is saved in the list. We expect to see only the identity automorphism. This command may take a little while to run, but it is instructive. There are more sensible ways to check computationally if the action is faithful (such as the result of the action for a set of generators of the group).

```
SFZ = SymmetricGroup(factorizations)
identity = SFZ("()")
[sigma for sigma in aut if SFZ([permute_factorization(sigma, fz) for fz in factorizations])
```

```
[()]
```

4 Constructions

From vertices, edges, factors and factorizations we can stitch together some interesting combinatorial configurations.

4.1 Projective Plane of Order 4

A **projective plane** of order n is a $2-(n^2+n+1, n+1, 1)$ design. Such a design is therefore a collection of sets (called **blocks**), each of size $n+1$, chosen from a universal set of size n^2+n+1, with the property that any 2 elements chosen from the universal set is contained in exactly 1 of the blocks.

As a finite geometry a projective plane of order n is a set of **points** and **lines**, together with an incidence relation, such that:

- Exactly one line passes through any two points.
- Any two lines pass through exactly one common point.
- Every point has exactly $n+1$ lines pasing through the point.
- Every line passes through exactly $n+1$ points.

Here the notion of a line "passing through" a point is more precisely given by the incidence relation. One common such relation is that lines are sets of points, and a point and line are incident if the point is an element of the line.

A finite projective plane of order 4 is equivalent to a set of three mutually orthogonal Latin Squares (see Chapter 12 of Combinatorial Theory, by Marshall Hall, Jr). The impossibility of the projective plane of order 10 was only settled in the late 1980s by Clement Lam.

We will build a design from our factorizations by first building a graph that describes an incidence relation. The points of the geometry will be the six vertices of K_6 together with the fifteen factors of K_6. The lines of the geometry will be the fifteen edges of K_6 along with the six factorizations of K_6. So we have 21 points and 21 lines. The graph will be bipartite with points in one set of the bipartition and lines in the other set. A vertex will be incident to an edge which contains it, a factor will be incident to an edge if the factor contains the edge, and a factor will be incident to factorization which contains it.

The resulting biparite graph is regular of degree 5 on 42 vertices with girth 6.

```
P = Graph()
P.add_edges([(v, e)  for v in vertices for e in edges if v in e])
P.add_edges([(f, e)  for f in factors for e in edges if e in f])
P.add_edges([(f, fz) for f in factors for fz in factorizations if f in fz])
P.num_verts(), P.is_bipartite(), P.is_regular(), P.average_degree(), P.girth()
```

```
(42, True, True, 5, 6)
```

We can relabel the vertices of a copy of the graph, and then have Sage recognize the bipartite structure before producing a drawing.

```
R = copy(P)
R.relabel(edge_names)
R.relabel(factor_names)
R.relabel(factorization_names)
B = BipartiteGraph(R)
B.plot()
```

If we call Sage's relabeling function with no arguments, the points will be labeled with the integers 0 to 20 and the lines will be labeled with the integers 21 to 41. We can create the blocks of a block design by creating lists of the points on each line.

```
C = copy(B)
C.relabel()
blocks = [[] for i in range(21)]
for e in C.edges():
    blocks[e[1]-21].append(e[0])
D = BlockDesign(21, blocks)
D.is_block_design()
```

```
(True, [2, 21, 5, 1])
```

So the parameters are those of a $2-(21,5,1)$ design, a projective plane of order 4.

We can list all of the 21 blocks, if we wish.

```
D.blocks()
```

```
[[0, 1, 6, 7, 8],
 [0, 2, 9, 10, 11],
 [0, 3, 12, 13, 14],
 [0, 4, 15, 16, 17],
 [0, 5, 18, 19, 20],
 [1, 2, 12, 15, 18],
 [1, 3, 9, 16, 19],
 [1, 4, 10, 13, 20],
 [1, 5, 11, 14, 17],
 [2, 3, 6, 17, 20],
 [2, 4, 7, 14, 19],
 [2, 5, 8, 13, 16],
 [3, 4, 8, 11, 18],
 [3, 5, 7, 10, 15],
 [4, 5, 6, 9, 12],
 [6, 10, 14, 16, 18],
 [6, 11, 13, 15, 19],
 [7, 9, 13, 17, 18],
 [7, 11, 12, 16, 20],
 [8, 9, 14, 15, 20],
 [8, 10, 12, 17, 19]]
```

Notice that this subsection is made possible by the arithmetic: $6 + 15 = 21 = 4^2 + 4 + 1$.

4.2 Hoffman-Singleton Graph

The Hoffman-Singleton Graph is an amazing structure, which can be created many different ways. It is best known as a Moore graph: regular of degree 7 with odd girth 5, on 50 vertices, which is the minimum conceivable number.

Our construction begins with two new vertices, which Cameron and van Lint call "zero" and "infinity" — we will shorten the label on the latter to simply "inf." Zero and infinity will be adjacent to each other, zero will be adjacent to each vertex of K_6 and infinity will be adjacent to each factorization of K_6.

```
G = Graph()
G.add_edge('inf', 'zero')
G.add_edges([(v, 'zero')for v in vertices])
G.add_edges([(fz, 'inf') for fz in factorizations])
```

We have 14 vertices in our graph so far, we get another 36 from the Cartesian product of the set of 6 K_6 vertices with the set of 6 factorizations. Each pair is joined to the K_6 vertex of the pair and the factorization in the pair. At this point the vertices from the Cartesian product have just degree 2, the other vertices have reached degree 7.

```
G.add_edges([(v,  (v,fz)) for v in vertices for fz in factorizations])
G.add_edges([(fz, (v,fz)) for v in vertices for fz in factorizations])
```

Among vertices of the Cartesian product a pair (a, x) is adjacent to the pair (b, y) if x and y are different factorizations and the edge $\{a, b\}$ is contained in the factor common to x and y. (Recall that every pair of factorizations have a single factor in common.)

```
for v1, v2 in Subsets(vertices, 2):
    for fz1, fz2 in Subsets(factorizations, 2):
        common_factor = fz1.intersection(fz2)[0]
        if Set([v1, v2]) in common_factor:
            G.add_edge((v1, fz1), (v2, fz2))
            G.add_edge((v2, fz1), (v1, fz2))
```

Construction complete, we can check that the graph has the requiste properties to be the Moore graph of degree 7 and girth 5.

```
G.num_verts(), G.is_regular(), G.average_degree(), G.girth()
```

```
(50, True, 7, 5)
```

Since these properties are enough to characterize the graph, we know we have it. However, we will let Sage do a brutish check. Our graph will not plot very nicely, while the implementation of the graph in Sage has a very novel feature. Each time you create the Hoffman-Singleton graph, you get a different layout, always displaying some of the inherent symmetry. So you can run the construct-and-plot command below repeatedly to experience different layouts.

```
HS = graphs.HoffmanSingletonGraph()
G.is_isomorphic(HS)
```

```
True
```

```
graphs.HoffmanSingletonGraph().plot()
```

Notice that this subsection is made possible by the arithmetic: $50 = 2 + 6 + 6 + 6 \cdot 6$.

4.3 The Steiner System $S(5, 6, 12)$

The Steiner system, $S(5, 6, 12)$ is a collection of 132 6-sets ("blocks"), chosen from a 12-set (the "points"), such that every set of 5 points chosen from the 12-set is contained in exactly one of the blocks. In other words, a $S(5, 6, 12)$ Steiner system is a 5-$(12, 6, 1)$ design.

The 12 points for our construction will be the 6 vertices of K_6 and the 6 factorizations of K_6. The construction of designs in Sage is not as robust as those of graphs and permutation groups, and require our points to be integers, starting at zero. We are in good shape with our vertices, and by necessity, we will use a dictionary to map factorizations to the integers 6 through 11.

```
factorization_numbers = dict(zip(factorizations, range(6, 12)))
```

The first two blocks of the construction will be the set of all six vertices and the set of all six factorizations. We will create the blocks as sorted lists. The sorting is only for the sake of appearance.

```
blocks = []
blocks.append(sorted(vertices))
```

```
blocks.append(sorted(factorization_numbers.values()))
```

The next 90 blocks come from considering each of the 15 pairs of distinct factorizations. We know that each such pair has a single factor in common. This factor has three edges, each edge providing two vertices as endpoints. For each of the $15 \cdot 3 = 45$ factorization-pair/vertex-pair combinations we build two blocks. Start the first block with the six vertices, remove the two vertices of the pair, and add in the two factorizations of the pair. Similarly, start the second block with the six factorizations, remove the two factorizations of the pair, and add in the two vertices of the pair.

```
for fz1, fz2 in Subsets(factorizations, 2):
    common_factor = fz1.intersection(fz2)[0]
    for v1, v2 in common_factor:
        ablock = copy(vertices)
        ablock.remove(v1)
        ablock.remove(v2)
        ablock.append(factorization_numbers[fz1])
        ablock.append(factorization_numbers[fz2])
        blocks.append(sorted(ablock))

        ablock = factorization_numbers.values()
        ablock.remove(factorization_numbers[fz1])
        ablock.remove(factorization_numbers[fz2])
        ablock.append(v1)
        ablock.append(v2)
        blocks.append(sorted(ablock))
```

The final 40 blocks require more care. Each of the $\binom{6}{3} = 20$ sets of triples can be used to create a permutation of the vertices that is a 3-cycle. While there are two possible ways to create this permutation, either choice leads to the same result in the following procedure. As we did in the section on the faithful group action, this permutation of the vertices can be used to induce a permutation of the six factorizations. It is not obvious (details are in van Lint and Wilson), but the induced permutation is a product of two 3-cycles. We use these two 3-cycles merely to group the six factorizations into two sets of three. (This decrease in structure partially explains the flexibility in the choice of the initial 3-cycle.) For each of the two sets of three factorizations so determined, append the three vertices chosen initially, so as to create the final $20 \cdot 2 = 40$ blocks.

```
G = SymmetricGroup(vertices)
H = SymmetricGroup(factorizations)

for triple in Subsets(vertices, 3):
    vertex_perm = G(tuple(triple))
    fz_perm = [permute_factorization(vertex_perm, fz) for fz in factorizations]
    fz_perm_cycles = H(fz_perm).cycle_tuples()
    for fz_triple in fz_perm_cycles:
        ablock = triple.list()
        for fz in fz_triple:
            ablock.append(factorization_numbers[fz])
        blocks.append(sorted(ablock))
```

There are now 132 blocks, which we can examine if we wish.

```
blocks
```

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

We can, of course, verify analytically that this construction creates a Steiner system, but we let Sage do the work instead and see that the parameters returned are as expected.

```
B = BlockDesign(12, blocks)
B.is_block_design()
```

```
(True, [5, 12, 6, 1])
```

Note that the enabling arithmetic for this example is: $12 = 6 + 6$ and $132 = 2 + 15 \cdot 3 \cdot 2 + \binom{6}{3} \cdot 2$.

The automorphism group of a design (Steiner system) is a permutation of the points which carries blocks to blocks, and non-blocks to non-blocks. The Steiner system $S(5, 6, 12)$ is remarkable, in part because its automorphism group is the Mathieu group, M_{12}. This permutation group is highly transitive and is one of the first five sporadic simple groups discovered. In particular, M_{12} is 5-transitive, a fact you can verify in Sage by creating the group and examining a succession of orbits and stabilizers.

```
M12 = MathieuGroup(12)
M12
```

```
Mathieu group of degree 12 and order 95040 as a permutation group
```

Exercise 1. Verify that the Mathieu group, as implemented directly in Sage, is a 5-transitive permutation group.

Sage claims to be able to build the automorphism group of this design, but the routine is broken and would likely produce permutations on 144 symbols (all vertices of the point-block incidence graph of the design). The code below will alert us, through automated testing, when this situation improves.

```
B.automorphism_group()
```

```
Traceback (most recent call last):
...
ValueError: Invalid permutation vector: [1, 4, 3, 2, 9, 10, 8,
7, 5, 6, 11, 12, 36, 34, 35, 31, 32, 33, 38, 37, 41, 40, 39, 44,
45, 43, 42, 46, 48, 47, 16, 17, 18, 14, 15, 13, 20, 19, 23, 22,
21, 27, 26, 24, 25, 28, 30, 29, 51, 50, 49, 55, 54, 53, 52, 56,
57, 58, 59, 60, 66, 65, 64, 63, 62, 61, 72, 71, 70, 69, 68, 67,
73, 74, 75, 76, 77, 81, 80, 79, 78, 84, 83, 82, 104, 103, 105,
108, 109, 107, 106, 112, 111, 110, 114, 113, 120, 118, 119, 115,
116, 117, 86, 85, 87, 91, 90, 88, 89, 94, 93, 92, 96, 95, 100,
101, 102, 98, 99, 97, 121, 122, 127, 128, 126, 125, 123, 124,
131, 130, 129, 132]
```

Exercise 2. Analyze all $12! = 479\,001\,600$ permutations of the 12 points of the $S(5, 6, 12)$ Steiner system and identify the 95 040 permutations which preserve the blocks. This may take a long time (perhaps an unreasonable amount of time). To be more efficient, perhaps use the Sage combinatorics routines to build the permutations quickly (rather than the permutation group code), and try to move on to testing the next permutation just as soon as the current one fails to preserve some block.

Exercise 3. Build the point-block incidence graph of the $S(5, 6, 12)$ Steiner system and construct the group of graph automorphisms. For these permuations, isolate the permutations of the points of the design which are automorphisms of the design. Use the experience to debug and fix Sage's automorphism group routine for designs.

5 Duality

Shortly after we built the six factorizations of K_6, we created a graphic which was another version of K_6 with factorizations as vertices and factors labeling (coloring) edges. This was possible, since as we also demonstrated above, every pair of factorizations have a single factor in common, and each factor is contained in exactly two factorizations. This version of K_6 is "dual" to the original K_6, as we will describe in this section.

Build a graph, Q, whose vertices are the factorizations of K_6, with two vertices adjacent if the factorizations have a factor in common. The code below is a bit overblown, as we expect this graph to be isomorphic to K_6. To emphasize the situation, we will label the vertices of Q (factorizations) by using their names, the uppercase letters from the end of the alphabet.

```
Q = Graph([(f1, f2) for f1, f2 in Subsets(factorizations, 2) if f1.intersection(f2).cardina
Q.relabel(factorization_names)
Q.is_isomorphic(graphs.CompleteGraph(6))
```

```
True
```

```
Q.plot(layout='circular')
```

By construction, the factorizations of K_6 are the vertices of Q. The factors of K_6, by the discussion above, are in a bijective correspondence with the edges of Q (pairs of factorizations). We illustrate these ideas with functions that accept the factorizations and factors of K_6 (respectively) and return the vertices and edges of Q. To emphasize the nature of the situation, the trivial function that takes factorizations to vertices will return the factorization names provided by the Python dictionary `factorization_names`. We test these functions exhaustively. Also, notice how the second function employs the first.

```
def factorization2vertex(afactorization):
    '''Returns a vertex of the dual graph determined by a factorization of the original gra
    return factorization_names[afactorization]
```

```
[factorization2vertex(fz) for fz in factorizations]
```

```
['U', 'V', 'W', 'X', 'Y', 'Z']
```

```
def factor2edge(afactor):
    '''Returns an edge of the dual graph determined by a factor of the original graph'''
    endpoints = [fz for fz in factorizations if afactor in fz]
    return Set([factorization2vertex(avertex) for avertex in endpoints])
```

```
[factor2edge(f) for f in factors]
```

```
[{'X', 'U'},
 {'Y', 'W'},
 {'Z', 'V'},
 {'Z', 'W'},
 {'U', 'V'},
 {'Y', 'X'},
 {'Y', 'V'},
 {'X', 'W'},
 {'Z', 'U'},
 {'X', 'Z'},
 {'Y', 'U'},
 {'W', 'V'},
 {'U', 'W'},
 {'X', 'V'},
 {'Y', 'Z'}]
```

Now we describe a natural bijection between edges of K_6 and factors of Q. Given an edge of K_6, there remain four vertices not incident to this edge. The subgraph induced by these four vertices (a complete graph on 4 vertices) contains three pairs of disjoint edges, which may be combined with the first edge to form the three edges of a factor of K_6. Each of these three factors of K_6 is then associated with an edge of Q. Since these three factors are not pairwise disjoint, no two of the factors can be in the same factorization. So the three factors determine three pairs of factorizations, and these six factorizations are different. Thus, these three pairs of factorizations form a factor in the graph Q. So we begin with an edge of K_6 and determine a factor of the dual, Q.

Here is the function, and the test.

```
def edge2factor(anedge):
    '''Returns a factor of the dual graph determined by an edge of the original graph'''
    completions = [f for f in factors if anedge in f]
    return [factor2edge(f) for f in completions]
```

```
[edge2factor(e) for e in edges]
```

```
[[{'X', 'U'}, {'Y', 'W'}, {'Z', 'V'}],
 [{'Z', 'W'}, {'U', 'V'}, {'Y', 'X'}],
 [{'Y', 'V'}, {'X', 'W'}, {'Z', 'U'}],
 [{'X', 'Z'}, {'Y', 'U'}, {'W', 'V'}],
 [{'U', 'W'}, {'X', 'V'}, {'Y', 'Z'}],
 [{'Y', 'V'}, {'X', 'Z'}, {'U', 'W'}],
 [{'Z', 'W'}, {'Y', 'U'}, {'X', 'V'}],
 [{'U', 'V'}, {'X', 'W'}, {'Y', 'Z'}],
 [{'Y', 'X'}, {'Z', 'U'}, {'W', 'V'}],
 [{'X', 'U'}, {'W', 'V'}, {'Y', 'Z'}],
 [{'Y', 'W'}, {'Z', 'U'}, {'X', 'V'}],
 [{'Z', 'V'}, {'X', 'W'}, {'Y', 'U'}],
 [{'Z', 'V'}, {'Y', 'X'}, {'U', 'W'}],
 [{'Y', 'W'}, {'U', 'V'}, {'X', 'Z'}],
 [{'X', 'U'}, {'Z', 'W'}, {'Y', 'V'}]]
```

Finally, you might now expect a natural bijection between vertices of K_6 and factorizations of Q. We will not disappoint. Given a vertex of K_6, there are five edges incident to this vertex. No factor of K_6 can contain two of these edges, since all the edges have a vertex in common. Each edge determines a factor of Q as above. These five factors of Q are disjoint, for if not, an edge common to two factors of Q would be associated with a factor of K_6 that contained two of the five original edges of K_6. So a vertex of K_6 determines five disjoint factors of Q, a factorization of the dual, Q.

```
def vertex2factorization(avertex):
    '''Returns the factorization of the dual graph determined by a vertex of the original g
    incident = [e for e in edges if avertex in e]
    return [edge2factor(e) for e in incident]
```

```
[vertex2factorization(v) for v in vertices]
```

```
[[[{'X', 'U'}, {'Y', 'W'}, {'Z', 'V'}], [{'Z', 'W'}, {'U', 'V'}, {'Y', 'X'}], [{'Y', 'V'
  [{'X', 'Z'}, {'Y', 'U'}, {'W', 'V'}], [{'U', 'W'}, {'X', 'V'}, {'Y', 'Z'}]],
 [[{'X', 'U'}, {'Y', 'W'}, {'Z', 'V'}], [{'Y', 'V'}, {'X', 'Z'}, {'U', 'W'}], [{'Z', 'W'
  [{'U', 'V'}, {'X', 'W'}, {'Y', 'Z'}], [{'Y', 'X'}, {'Z', 'U'}, {'W', 'V'}]],
 [[{'Z', 'W'}, {'U', 'V'}, {'Y', 'X'}], [{'Y', 'V'}, {'X', 'Z'}, {'U', 'W'}], [{'X', 'U'
  [{'Y', 'W'}, {'Z', 'U'}, {'X', 'V'}], [{'Z', 'V'}, {'X', 'W'}, {'Y', 'U'}]],
 [[{'Y', 'V'}, {'X', 'W'}, {'Z', 'U'}], [{'Z', 'W'}, {'Y', 'U'}, {'X', 'V'}], [{'X', 'U'
  [{'Z', 'V'}, {'Y', 'X'}, {'U', 'W'}], [{'Y', 'W'}, {'U', 'V'}, {'X', 'Z'}]],
 [[{'X', 'Z'}, {'Y', 'U'}, {'W', 'V'}], [{'U', 'V'}, {'X', 'W'}, {'Y', 'Z'}], [{'Y', 'W'
  [{'Z', 'V'}, {'Y', 'X'}, {'U', 'W'}], [{'X', 'U'}, {'Z', 'W'}, {'Y', 'V'}]],
 [[{'U', 'W'}, {'X', 'V'}, {'Y', 'Z'}], [{'Y', 'X'}, {'Z', 'U'}, {'W', 'V'}], [{'Z', 'V'
  [{'Y', 'W'}, {'U', 'V'}, {'X', 'Z'}], [{'X', 'U'}, {'Z', 'W'}, {'Y', 'V'}]]]
```

So notice that we have computed the factors and factorizations of the dual graph Q by exploiting properties of vertices, edges, factors and factorizations in K_6. This is in contrast to computing factors and factorizations of Q with set partition routines, or maximal cliques in an appropriate graph. The following table summarizes what we mean by saying that Q is the dual of K_6.

K_6	Q	Function
Factorization	Vertex	`factorization2vertex`
Factor	Edge	`factor2edge`
Edge	Factor	`edge2factor`
Vertex	Factorization	`vertex2factorization`

Table 1: Factor and factorization correspondences

Cameron and van Lint state the following theorem.

Theorem 5.1. *Six is the only natural number n for which there is a construction of n isomorphic objects on an n-set A, invariant under all permutations of A, but not naturally in one-to-one correspondence with the points of A.*

So here A is the set of six original vertices of K_6, and the "n isomorphic objects" are the six factorizations of K_6. We saw in the section on group actions that each permutation of the vertex set of K_6 creates a permutation of the factorizations. While there is a one-to-one correspondence between vertices and factorizations, the correspondence is not trivial. This theorem can be explained several ways, but perhaps the most concise is the category-theoretic statement that

> The category whose objects are n-element sets and whose morphisms are bijections between them has a non-trivial functor to itself if and only if $n = 6$.

Exercise 4. Begin with the graph Q having vertices labeled "W" through "Z" and construct the vertices, edges, factors and factorizations directly, using the methods employed for K_6 at the start of this chapter. Use your results, and equality testing of sets in Sage, to verify the exhaustive tests provided here. (Notice that our "tests," as given, rely on visual inspection, while this exercise suggests a much more rigorous approach.)

Exercise 5. Create the graph Q again, along with its vertices, edges, factors and factorizations using the direct methods described in the previous exercise, but now begin with the vertices labeled with the actual factorizations of K_6. So, for example, an edge of this graph will be a 2-set, whose elements are factorizations of K_6 (each

a 5-set of 3-sets of 2-sets). Write and test the inverses of the four bijections above. These will accept as input the (complicated) structures of Q and output the simpler structures of K_6.