COMPUTATION OF ORBIT SPACE HOMOLOGY

JACKSON WALTERS

1. Introduction

We seek to compute the homology of a CW-complex with a G action where G is a finite group. In particular, we are interested in the case of a quotient space X/G, where a topological space $X \subset \mathbb{R}^N$ carries an action, $G \circlearrowright X$.

Note that in general we have an invariance property for the quotient space, $H_*(X/G) = H_*(X)^G$. If X is contractible, then we obtain that $H_*(X/G) = 0$ outside of degree 0.

It's still interesting to see exactly why this result holds, and along the way we'll compute a fundamental domain for the moduli space of networks. We'll also provide the general setup for computing the homology of a space (possibly non-contractible) with a G-action.

Computation

We will perform this computation by computing simplicial homology. The key difficulty is that we may have faces with fixed points. We will compute these fixed points using linear algebra, and then include them in the simplicial complex to obtain a refinement that no longer contains fixed interior points.

First we define n for the symmetric group S_n and define how the group acts on vectors, as well as how the embedding $\Sigma_n \subset \Sigma_N$ behaves.

```
n = 4
N=binomial(n,2)
#N=4

#list of unordered pairs of elements in {1,...,n}
def pairs(n): return flatten([[[i+1,j+1] for j in
        range(i+1,n)] for i in range(n-1)],max_level=1)

#sigma acts on an unordered pair
def act(sigma, p): return sorted([sigma(p[0]),sigma(p[1])])

#sigma acts as matrix in basis on a vector v
```

Date: March 28, 2026.

```
def act_vect(g,v,basis=identity_matrix(N)):
        return (basis.inverse()*g.matrix()*basis)*vector(v)

#permutation in Sigma_N which induced from perm in Sigma_n
def embed(sigma,n): return perm_from_sort([act(sigma,p) for p in pairs(n)])

#find permutation which orders a list L
def perm_from_sort(L): return Permutation([pair[0] for
        pair in sorted(enumerate(L, 1), key=lambda x: x[1])]).to_cycles()

#image of generators of Sigma_n under embedding into Sigma_N
def embed_gens(n): return [embed(gen,n) for gen in SymmetricGroup(n).gens()]
```

Next we define how to find the minimum between two vectors when iterating over the group, i.e. $\min_{\sigma\in G} \angle(x,\sigma y)$:

```
#return n when N=(n choose 2)
def invert_binomial(N):
    for n in range(floor(sqrt(2*N)),floor(sqrt(2*N)+2)):
        if binomial(n,2) == N:
            return n

#angle between two vectors
def angle(x,y): return float(vector(x)*vector(y)/(vector(x).norm()*vector(y).norm()))

#naive minimum finding acting over entire group
def find_min(x,y):
    assert len(x) == len(y)
    print(angle(x,y))
    n = invert_binomial(len(x))
    G = PermutationGroup(embed_gens(n))
    for sigma in G:
        if vector(x)*(sigma.matrix()*vector(y)) >= vector(x)*vector(y):
            y = sigma.matrix()*vector(y)
    print(angle(x,y))
    return y
```

Next, we determine how to find the fundamental domain. Let the distance between two vectors be given by $d_{quot}(x,y) = \min_{\sigma\in G} ||x-\sigma y||^2$. We can rewrite this as $d_{quot}(x,y) = \max_{\sigma\in G}\{x\cdot\sigma y\}$. Then we can define $F_x = \{y\in\mathbb{R}^N : d_{quot}(x,y) = d_{Euc}(x,y) \subset \mathbb{R}^N$. Note that our fundamental domain is "centered" at some vector $x\in\mathbb{R}^N$.

Note that the fundamental domain has a very simple geometry: it is a cone, centered on the vector x. This is because we can rewrite the distance formulas as

$$\begin{aligned} d_{Euc}(x,y) &= ||x-y||^2 = ||x||^2 + ||y||^2 - 2x\cdot y \\ &= d_{quot}(x,y) = \min_{\sigma\in G} ||x-\sigma y||^2 \end{aligned}$$

$$= \min_{\sigma \in G} ||x||^2 + ||y||^2 - 2x \cdot \sigma y$$

$$= ||x||^2 + ||y||^2 + \min_{\sigma \in G} -2x \cdot \sigma y$$

This reduces to

$$F_x = \{y \in \mathbb{R}^N : x \cdot y = \max_{\sigma \in G} x \cdot \sigma y\}$$

$$= \{y \in \mathbb{R}^N : \forall \sigma \in G x \cdot y \leq x \cdot \sigma y\}$$

$$= \bigcap_{\sigma \in G} \{y \in \mathbb{R}^N : x \cdot y \leq x \cdot \sigma y\}$$

This is a list of linear inequalities, which forms the list of constraints. In $\mathbb{R}^N$, a list of linear equalities will carve out a convex polyhedral region. This is the convex hull of the origin and a number of rays.

```
#fundamental domain given arbitrary distinct vector l and basis B
def fund_domain(center=[i for i in range(N)],basis=identity_matrix(N),
    br=QQ,group=MatrixGroup(identity_matrix(N))):

    #augmented matrix for half-plane ineqs
    A = [ [center[j] - act_vect(group[i],vector(center),basis)[j]
        for j in range(N)]
        for i in range(group.order())]
    b_1 = group.order()*[0]
    bA = matrix(br,b_1).transpose().augment(matrix(br,A))

    #augmented matrix for positivity ineqs
    pos = basis.inverse().transpose()
    b_2 = [0 for i in range(N)]
    bPos = matrix(br,b_2).transpose().augment(pos)

    #augmented matrix for fund. domain
    aug_l = bA.stack(bPos)

    #find convex polyhedral region, i.e. intersection of all ineqs
    poly = Polyhedron(ieqs=aug_l,base_ring=br)

    return([aug_l,poly])
```

We also want to be able to slice the fundamental domain by a hyperplane, e.g. $H = \{x \cdot (1, 1, \ldots, 1\} = 0\}$:

```
#slice the fundamental domain with the plane x_1 + ... + x_N = 1
def cross_section(region,slice_dir=[1 for i in range(N)],chi=1,br=None):
    if br is None: br=region.base_ring()
    return Polyhedron(ieqs=region.inequalities(),eqns=[[-1]+slice_dir],
        base_ring=br)
```

We also want to be able to compute the fixed points of a face. Note that if we have a face determined by vertices $\{v_0, v_1, \ldots, v_k\}$, then we can form the hyperplane $\text{span}_{\mathbb{Q}}\{v_0, v_1, \ldots, v_k\}$. Let $\sigma \in G$. Then we have a matrix $[g]$. We want to solve the equation $[g] \cdot v = v$. Conveniently, this is an eigenvalue problem with $\lambda = 1$.

```
#find fixed points under G in polyhedron F to include as new vertices
#find intersection of fixed point subspace with F as polyhedron
#and return those vertices
def fixed_verts(F,br=QQ):
    new_verts=set()
    for g in G:
        #get equations for fixed point subspace
        A=g.matrix() #matrix associated to group element g
        #eqns defining fixed pt subspace are (A-Id)x == 0
        B=A-identity_matrix(N)
        b = N*[0] #0 vector
        #form augmented matrix for equations defining subspace
        bB = matrix(br,b).transpose().augment(matrix(br,B))
        #list of equations defining F
        eqns=matrix(br,[list(eq) for eq in F.equations()])
        #augmented matrix including fixed point subspace equations
        eqns=eqns.stack(bB)
        #get list of inequalities for fundamental domain as convex polyhedron
        ieqs=matrix(br,[list(ieq) for ieq in F.inequalities()])
        #form intersection of fund_domain with fixed point subspace
        intersection_F_fixed_pt_subspace=Polyhedron(ieqs=ieqs,eqns=eqns)
        verts=intersection_F_fixed_pt_subspace.vertices()
        for vert in verts:
            new_verts.add(tuple(vert))
    return sorted(list(new_verts))
```

Next we have some functions regarding gluing of faces.

```
#action of group element g on a face
#if g.face is not in fund_domain defined by vertex list, return None
def act_face(g,face,vertices):
    try:
        return [vertices.index(tuple(act_vect(g,vertices[i]))) for
                        i in face]
    except ValueError:
        return None

#determine if face1 is glued to face2 by the action of G
#faces are given by a list vertex indices
def faces_glued(face1,face2,G,vertices):
    for g in G:
        g_face1 = act_face(g,face1,vertices)
        if g_face1 is not None and set(g_face1) == set(face2):
            return True
    return False
```

```
# for two glued faces face1, face2, determine if the gluing map
# preserves orientation
# if a gluing map g1 preserves orientation, then all do since g1*g2^(-1)
# is a self map and must be trivial
def gluing_preserves_orientation(face1,face2,vertices):
    S=SymmetricGroup(range(len(vertices))) #symmetric group on vertex indices
    for g in G:
        g_face1 = act_face(g,face1,vertices)
        if g_face1 is not None and set(g_face1) == set(face2):
            sigma_g=perm_from_sort(g_face1)
            #get permutation which orders g.v_{i_0}...g.v_{i_k}
            g_cycle=PermutationGroupElement(sigma_g,parent=S)
            return g_cycle.sign()
    return None

#determine if there is group element which glues face non-trivially to itself
def trivial_self_gluing(face,G,vertices):
    trivial = True
    for g in G:
        #act on vertices defining face
        g_face = act_face(g,face,vertices)
        if g_face is not None and set(g_face) == set(face):
            #check if the gluing is non-trivial
            if [g_face.index(i) for i in face] != [i for i in range(len(face))]:
                trivial = False
    return trivial

#toss out any facets with non-trivial self-gluings such as [0,3,4] or [1,6]
#build list of faces for each dim
import itertools
def faces(k): return [face for face in itertools.combinations(
        range(len(vertices)), k+1) if trivial_self_gluing(face,G,vertices)]

#find classes of vertices which are glued
def glued_faces(k):
    glued_verts = []
    for i in faces(k):
        found = False
        for equiv_class in glued_verts:
            if len(equiv_class) >= 1:
                if faces_glued(i,equiv_class[0],G,vertices):
                    equiv_class.append(i)
                    found=True
        if not found:
            glued_verts.append([i])
    return glued_verts
```

Finally, we have the computation of the boundary faces, the boundary map, and the chain complex:

```
#compute boundary of each face. keep track of orientation
def boundary(face,glued_faces,vertices):
    #face/simplex dimension is number of vertices in face-1
    dim = len(face)-1
    #initialize vector to count occurence boundary faces
    boundary = [0]*len(glued_faces[dim-1])
    for i in range(len(face)):
        face_remove_i = face[:i] + face[i+1:] #remove vertex at index i
        sign = (-1)^i
        #find representative to which face\{i} is glued to
        for glued_face in glued_faces[dim-1]:
            #use first face in glued_face list as representative
            face_rep = glued_face[0]
            face_index = None
            #check if face_remove_i is in gluing class
            if face_remove_i in glued_face:
                #get index of face in glued (k-1)-faces
                face_index = glued_faces[dim-1].index(glued_face)
                #determine if gluing map preserves orientation
                orient_preserve =
                    gluing_preserves_orientation(face_rep,
                        face_remove_i,vertices)
            if face_index is not None:
                #add up using index
                boundary[face_index] += orient_preserve*sign
    return boundary

#compute the boundary map matrix over \ZZ
def boundary_map(k,glued_face_list,vertices): return matrix(ZZ,[boundary(
    face[0],glued_face_list,vertices) for face in
    glued_face_list[k]]).transpose()

#define a chain complex, optionally truncated at max_degree
def chain_complex(max_degree=1): return ChainComplex({k:boundary_map(k,
    glued_face_list,vertices) for k in range(1,max_degree+1)},degree=-1)
```

Now we can begin computing things one at a time. First, let's compute a fundamental domain centered at $(1, 2, 3, 4, 5, 6.1)$:

```
#define the finite group
G = PermutationGroup(embed_gens(n)) #symmetric group \sigma_n as a subgroup of \sigma_N
#G = MatrixGroup(matrix(QQ,[[0,0,0,1],[1,0,0,0],[0,1,0,0],[0,0,1,0]]))
#G=SymmetricGroup(4)

#compute fundamental domain to get list of vertices
F=fund_domain(group=G,center=[1,2,3,5,6,8]); F[1]
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex and 16 rays (
```

We obtain a polyhedral region in $\mathbb{R}^N$ which is the convex hull of 1 vertex (the origin) and 16 rays. We can slice the fundamental domain by the hyperplane $x_1 + x_2 + \ldots + x_N = 1$:

```
# compute a cross section of the fundamental domain
# (sum of components is constant)
F2=cross_section(F[1])
[tuple(v) for v in F2.vertices()]
[(0, 0, 0, 1/3, 1/3, 1/3),
 (0, 0, 1/3, 0, 1/3, 1/3),
 (0, 0, 9/22, 25/66, 7/66, 7/66),
 (25/66, 0, 7/66, 0, 7/66, 9/22),
 (2/9, 0, 2/9, 2/9, 1/9, 2/9),
 (9/22, 0, 0, 7/66, 7/66, 25/66),
 (1/4, 0, 1/4, 1/4, 0, 1/4),
 (1/6, 1/6, 1/6, 1/6, 1/6, 1/6),
 (1/2, 0, 0, 0, 0, 1/2),
 (1/4, 1/4, 0, 0, 1/4, 1/4),
 (2/5, 0, 0, 0, 1/5, 2/5),
 (0, 0, 0, 0, 1/2, 1/2),
 (0, 2/5, 0, 0, 2/5, 1/5),
 (0, 0, 0, 0, 0, 1),
 (0, 2/9, 2/9, 2/9, 2/9, 1/9),
 (0, 0, 2/5, 2/5, 0, 1/5)]
```

We obtain a list of verticies now with no rays. We can compute the fixed points of the region F_2. We obtain a list of 16 verticies.

```
# compute the fixed points under G in polyhedron F to include as new vertices
vertices=fixed_verts(F2,QQ); vertices
[(0, 0, 0, 0, 0, 1),
 (0, 0, 0, 0, 1/2, 1/2),
 (0, 0, 0, 1/3, 1/3, 1/3),
 (0, 0, 1/3, 0, 1/3, 1/3),
 (0, 0, 2/5, 2/5, 0, 1/5),
 (0, 0, 9/22, 25/66, 7/66, 7/66),
 (0, 2/9, 2/9, 2/9, 2/9, 1/9),
 (0, 2/5, 0, 0, 2/5, 1/5),
 (1/6, 1/6, 1/6, 1/6, 1/6, 1/6),
 (2/9, 0, 2/9, 2/9, 1/9, 2/9),
 (1/4, 0, 1/4, 1/4, 0, 1/4),
 (1/4, 1/4, 0, 0, 1/4, 1/4),
 (25/66, 0, 7/66, 0, 7/66, 9/22),
 (2/5, 0, 0, 0, 1/5, 2/5),
 (9/22, 0, 0, 7/66, 7/66, 25/66),
 (1/2, 0, 0, 0, 0, 1/2)]
```

We can compute the 0-faces under gluing:

```
#compute glued 0-faces
zero_faces=glued_faces(0); zero_faces
```

```
[[(0,)],
 [(1,)],
 [(2,)],
 [(3,)],
 [(4,), (7,), (13,)],
 [(5,), (12,), (14,)],
 [(6,), (9,)],
 [(8,)],
 [(10,), (11,)],
 [(15,)]]
```

We can compute the 1-faces under gluing:

```
#compute glued 1-faces
one_faces=glued_faces(1); one_faces
[[(0, 1)],
 [(0, 2)],
 [(0, 3)],
 [(0, 4), (0, 7)],
 [(0, 5)],
 [(0, 6)],
 [(0, 8)],
 [(0, 9)],
 [(0, 10), (0, 11)],
 [(0, 12)],
 [(0, 13)],
 [(0, 14)],
 [(0, 15)],
 [(1, 2)],
 [(1, 3)],
 [(1, 4)],
 [(1, 5)],
 [(1, 6), (1, 9)],
 [(1, 7), (1, 13)],
 [(1, 8)],
 [(1, 10)],
 [(1, 11)],
 [(1, 12)],
 [(1, 14)],
 [(1, 15)],
 [(2, 3)],
 [(2, 4), (2, 7), (2, 13)],
 [(2, 5), (2, 14)],
 [(2, 6), (2, 9)],
 [(2, 8)],
 [(2, 10), (2, 11)],
 [(2, 12)],
 [(2, 15)],
 [(3, 4), (3, 7), (3, 13)],
 [(3, 5), (3, 12)],
```

```
 [(3, 6), (3, 9)],
 [(3, 8)],
 [(3, 10), (3, 11)],
 [(3, 14)],
 [(3, 15)],
 [(4, 5), (12, 13), (13, 14)],
 [(4, 6), (6, 7), (9, 13)],
 [(4, 8), (7, 8), (8, 13)],
 [(4, 9)],
 [(4, 10), (7, 11), (11, 13)],
 [(4, 11), (7, 10)],
 [(4, 12), (7, 12)],
 [(4, 13)],
 [(4, 14), (7, 14)],
 [(4, 15), (7, 15)],
 [(5, 6), (5, 9), (9, 12), (9, 14)],
 [(5, 7), (5, 13)],
 [(5, 8), (8, 12), (8, 14)],
 [(5, 10), (10, 12), (10, 14), (11, 12), (11, 14)],
 [(5, 11)],
 [(5, 15)],
 [(6, 8), (8, 9)],
 [(6, 10), (6, 11), (9, 11)],
 [(6, 12)],
 [(6, 13), (7, 9)],
 [(6, 14)],
 [(6, 15)],
 [(8, 10), (8, 11)],
 [(8, 15)],
 [(9, 10)],
 [(9, 15)],
 [(10, 13)],
 [(10, 15), (11, 15)],
 [(12, 15), (14, 15)],
 [(13, 15)]]
```

We can compute the 2-faces under gluing:
See Appendix **??**.
Now put the glued faces together:

```
# put glued faces together
glued_face_list=[zero_faces,one_faces,two_faces]
```

Finally, we compute the boundary map $d_1 : C_1 \to C_0$.

```
# compute the boundary map d_1: C_1 -> C_0
d1=boundary_map(1, glued_face_list, vertices); d1
10 x 70 dense matrix over Integer Ring (use the '.str()' method to see the entries)
```

We obtain a 10×70 dense matrix over $\mathbb{Z}$. We are also able to compute the boundary map $d_2 : C_2 \to C_1$:

```
# compute boundary map d_2
d2=boundary_map(2, glued_face_list, vertices); d2
70 x 381 dense matrix over Integer Ring (use the '.str()' method to see the entries)
```

We obtain a 70×381 dense matrix over $\mathbb{Z}$. Now we are set up to compute the first two homology groups:

```
#compute the homology of the chain complex
cc.homology()
{0: Z, 1: 0, 2: Z^320}
```

We find that $H_0(X) \cong \mathbb{Z}$, and $H_1(Z) \cong 0$. The $H_2(X)$ computation is an artifact of not having a map d_3 present and can be ignored. This is what we expect: the space is consistent with being contractibe.

References

[1] https://github.com/jacksonwalters/orbit-space-homology
[2] Averages of Unlabeled Networks and Geometric Characterization Eric D. Kolaczyk, Lizhen Lin, Steven Rosenberg, Jackson Walters, Jie Xu. Ann. Statist. 48(1): 514-538 (February 2020). DOI: 10.1214/19-AOS1820.

Appendix A. Glued 2-Face Data

```
# compute glued 2-faces
two_faces=glued_faces(2); two_faces
[[(0, 1, 2)],
 [(0, 1, 3)],
 [(0, 1, 4)],
 [(0, 1, 5)],
 [(0, 1, 6)],
 [(0, 1, 7)],
 [(0, 1, 8)],
 [(0, 1, 9)],
 [(0, 1, 10)],
 [(0, 1, 11)],
 [(0, 1, 12)],
 [(0, 1, 13)],
 [(0, 1, 14)],
 [(0, 1, 15)],
 [(0, 2, 3)],
 [(0, 2, 4), (0, 2, 7)],
 [(0, 2, 5)],
 [(0, 2, 6)],
 [(0, 2, 8)],
```

```
[(0, 2, 9)],
[(0, 2, 10), (0, 2, 11)],
[(0, 2, 12)],
[(0, 2, 13)],
[(0, 2, 14)],
[(0, 2, 15)],
[(0, 3, 4), (0, 3, 7)],
[(0, 3, 5)],
[(0, 3, 6)],
[(0, 3, 8)],
[(0, 3, 9)],
[(0, 3, 10), (0, 3, 11)],
[(0, 3, 12)],
[(0, 3, 13)],
[(0, 3, 14)],
[(0, 3, 15)],
[(0, 4, 5)],
[(0, 4, 6), (0, 6, 7)],
[(0, 4, 8), (0, 7, 8)],
[(0, 4, 9)],
[(0, 4, 10), (0, 7, 11)],
[(0, 4, 11), (0, 7, 10)],
[(0, 4, 12), (0, 7, 12)],
[(0, 4, 13)],
[(0, 4, 14), (0, 7, 14)],
[(0, 4, 15), (0, 7, 15)],
[(0, 5, 6)],
[(0, 5, 7)],
[(0, 5, 8)],
[(0, 5, 9)],
[(0, 5, 10)],
[(0, 5, 11)],
[(0, 5, 12)],
[(0, 5, 13)],
[(0, 5, 14)],
[(0, 5, 15)],
[(0, 6, 8)],
[(0, 6, 9)],
[(0, 6, 10), (0, 6, 11)],
[(0, 6, 12)],
[(0, 6, 13)],
[(0, 6, 14)],
[(0, 6, 15)],
[(0, 7, 9)],
[(0, 7, 13)],
[(0, 8, 9)],
[(0, 8, 10), (0, 8, 11)],
[(0, 8, 12)],
[(0, 8, 13)],
[(0, 8, 14)],
```

```
[(0, 8, 15)],
[(0, 9, 10)],
[(0, 9, 11)],
[(0, 9, 12)],
[(0, 9, 13)],
[(0, 9, 14)],
[(0, 9, 15)],
[(0, 10, 12), (0, 11, 12)],
[(0, 10, 13)],
[(0, 10, 14), (0, 11, 14)],
[(0, 10, 15), (0, 11, 15)],
[(0, 11, 13)],
[(0, 12, 13)],
[(0, 12, 14)],
[(0, 12, 15)],
[(0, 13, 14)],
[(0, 13, 15)],
[(0, 14, 15)],
[(1, 2, 3)],
[(1, 2, 4)],
[(1, 2, 5)],
[(1, 2, 6), (1, 2, 9)],
[(1, 2, 7), (1, 2, 13)],
[(1, 2, 8)],
[(1, 2, 10)],
[(1, 2, 11)],
[(1, 2, 12)],
[(1, 2, 14)],
[(1, 2, 15)],
[(1, 3, 4)],
[(1, 3, 5)],
[(1, 3, 6), (1, 3, 9)],
[(1, 3, 7), (1, 3, 13)],
[(1, 3, 8)],
[(1, 3, 10)],
[(1, 3, 11)],
[(1, 3, 12)],
[(1, 3, 14)],
[(1, 3, 15)],
[(1, 4, 5)],
[(1, 4, 6)],
[(1, 4, 7)],
[(1, 4, 8)],
[(1, 4, 9)],
[(1, 4, 10)],
[(1, 4, 11)],
[(1, 4, 12)],
[(1, 4, 13)],
[(1, 4, 14)],
[(1, 4, 15)],
```

```
[(1, 5, 6), (1, 5, 9)],
[(1, 5, 7), (1, 5, 13)],
[(1, 5, 8)],
[(1, 5, 10)],
[(1, 5, 11)],
[(1, 5, 12)],
[(1, 5, 14)],
[(1, 5, 15)],
[(1, 6, 7), (1, 9, 13)],
[(1, 6, 8), (1, 8, 9)],
[(1, 6, 10)],
[(1, 6, 11), (1, 9, 11)],
[(1, 6, 12)],
[(1, 6, 13), (1, 7, 9)],
[(1, 6, 14)],
[(1, 6, 15)],
[(1, 7, 8), (1, 8, 13)],
[(1, 7, 10)],
[(1, 7, 11), (1, 11, 13)],
[(1, 7, 12)],
[(1, 7, 14)],
[(1, 7, 15)],
[(1, 8, 10)],
[(1, 8, 11)],
[(1, 8, 12)],
[(1, 8, 14)],
[(1, 8, 15)],
[(1, 9, 10)],
[(1, 9, 12)],
[(1, 9, 14)],
[(1, 9, 15)],
[(1, 10, 11)],
[(1, 10, 12)],
[(1, 10, 13)],
[(1, 10, 14)],
[(1, 10, 15)],
[(1, 11, 12)],
[(1, 11, 14)],
[(1, 11, 15)],
[(1, 12, 13)],
[(1, 12, 14)],
[(1, 12, 15)],
[(1, 13, 14)],
[(1, 13, 15)],
[(1, 14, 15)],
[(2, 3, 4)],
[(2, 3, 5)],
[(2, 3, 6), (2, 3, 9)],
[(2, 3, 7), (2, 3, 13)],
[(2, 3, 8)],
```

```
[(2, 3, 10)],
[(2, 3, 11)],
[(2, 3, 12)],
[(2, 3, 14)],
[(2, 3, 15)],
[(2, 4, 5), (2, 13, 14)],
[(2, 4, 6), (2, 6, 7), (2, 9, 13)],
[(2, 4, 8), (2, 7, 8), (2, 8, 13)],
[(2, 4, 9)],
[(2, 4, 10), (2, 7, 11), (2, 11, 13)],
[(2, 4, 11), (2, 7, 10)],
[(2, 4, 12)],
[(2, 4, 13)],
[(2, 4, 14), (2, 7, 14)],
[(2, 4, 15), (2, 7, 15)],
[(2, 5, 6), (2, 5, 9), (2, 9, 14)],
[(2, 5, 7), (2, 5, 13)],
[(2, 5, 8), (2, 8, 14)],
[(2, 5, 10), (2, 10, 14), (2, 11, 14)],
[(2, 5, 11)],
[(2, 5, 12)],
[(2, 5, 15)],
[(2, 6, 8), (2, 8, 9)],
[(2, 6, 10), (2, 6, 11), (2, 9, 11)],
[(2, 6, 12)],
[(2, 6, 13), (2, 7, 9)],
[(2, 6, 14)],
[(2, 6, 15)],
[(2, 7, 12)],
[(2, 8, 10), (2, 8, 11)],
[(2, 8, 12)],
[(2, 8, 15)],
[(2, 9, 10)],
[(2, 9, 12)],
[(2, 9, 15)],
[(2, 10, 12)],
[(2, 10, 13)],
[(2, 10, 15), (2, 11, 15)],
[(2, 11, 12)],
[(2, 12, 13)],
[(2, 12, 14)],
[(2, 12, 15)],
[(2, 13, 15)],
[(2, 14, 15)],
[(3, 4, 5), (3, 12, 13)],
[(3, 4, 6), (3, 6, 7), (3, 9, 13)],
[(3, 4, 8), (3, 7, 8), (3, 8, 13)],
[(3, 4, 9)],
[(3, 4, 10), (3, 7, 11), (3, 11, 13)],
[(3, 4, 11), (3, 7, 10)],
```

```
[(3, 4, 12), (3, 7, 12)],
[(3, 4, 13)],
[(3, 4, 14)],
[(3, 4, 15), (3, 7, 15)],
[(3, 5, 6), (3, 5, 9), (3, 9, 12)],
[(3, 5, 7), (3, 5, 13)],
[(3, 5, 8), (3, 8, 12)],
[(3, 5, 10), (3, 10, 12), (3, 11, 12)],
[(3, 5, 11)],
[(3, 5, 14)],
[(3, 5, 15)],
[(3, 6, 8), (3, 8, 9)],
[(3, 6, 10), (3, 6, 11), (3, 9, 11)],
[(3, 6, 12)],
[(3, 6, 13), (3, 7, 9)],
[(3, 6, 14)],
[(3, 6, 15)],
[(3, 7, 14)],
[(3, 8, 10), (3, 8, 11)],
[(3, 8, 14)],
[(3, 8, 15)],
[(3, 9, 10)],
[(3, 9, 14)],
[(3, 9, 15)],
[(3, 10, 13)],
[(3, 10, 14)],
[(3, 10, 15), (3, 11, 15)],
[(3, 11, 14)],
[(3, 12, 14)],
[(3, 12, 15)],
[(3, 13, 14)],
[(3, 13, 15)],
[(3, 14, 15)],
[(4, 5, 6), (9, 12, 13), (9, 13, 14)],
[(4, 5, 7)],
[(4, 5, 8), (8, 12, 13), (8, 13, 14)],
[(4, 5, 9)],
[(4, 5, 10), (11, 12, 13), (11, 13, 14)],
[(4, 5, 11)],
[(4, 5, 12)],
[(4, 5, 13)],
[(4, 5, 14)],
[(4, 5, 15)],
[(4, 6, 8), (6, 7, 8), (8, 9, 13)],
[(4, 6, 9)],
[(4, 6, 10), (6, 7, 11), (9, 11, 13)],
[(4, 6, 11), (6, 7, 10)],
[(4, 6, 12), (6, 7, 12)],
[(4, 6, 13)],
[(4, 6, 14), (6, 7, 14)],
```

```
[(4, 6, 15), (6, 7, 15)],
[(4, 7, 9)],
[(4, 7, 10), (4, 7, 11)],
[(4, 7, 13)],
[(4, 8, 9)],
[(4, 8, 10), (7, 8, 11), (8, 11, 13)],
[(4, 8, 11), (7, 8, 10)],
[(4, 8, 12), (7, 8, 12)],
[(4, 8, 13)],
[(4, 8, 14), (7, 8, 14)],
[(4, 8, 15), (7, 8, 15)],
[(4, 9, 10)],
[(4, 9, 11)],
[(4, 9, 12)],
[(4, 9, 13)],
[(4, 9, 14)],
[(4, 9, 15)],
[(4, 10, 11), (7, 10, 11)],
[(4, 10, 12), (7, 11, 12)],
[(4, 10, 13)],
[(4, 10, 14), (7, 11, 14)],
[(4, 10, 15), (7, 11, 15)],
[(4, 11, 12), (7, 10, 12)],
[(4, 11, 13)],
[(4, 11, 14), (7, 10, 14)],
[(4, 11, 15), (7, 10, 15)],
[(4, 12, 13)],
[(4, 12, 14)],
[(4, 12, 15), (7, 12, 15)],
[(4, 13, 14)],
[(4, 13, 15)],
[(4, 14, 15), (7, 14, 15)],
[(5, 6, 7), (5, 9, 13)],
[(5, 6, 8), (5, 8, 9), (8, 9, 12), (8, 9, 14)],
[(5, 6, 10), (9, 11, 12), (9, 11, 14)],
[(5, 6, 11), (5, 9, 11)],
[(5, 6, 12)],
[(5, 6, 13), (5, 7, 9)],
[(5, 6, 14)],
[(5, 6, 15)],
[(5, 7, 8), (5, 8, 13)],
[(5, 7, 10)],
[(5, 7, 11), (5, 11, 13)],
[(5, 7, 12)],
[(5, 7, 14)],
[(5, 7, 15)],
[(5, 8, 10), (8, 10, 12), (8, 10, 14), (8, 11, 12), (8, 11, 14)],
[(5, 8, 11)],
[(5, 8, 15)],
[(5, 9, 10), (9, 10, 12), (9, 10, 14)],
```

```
[(5, 9, 15)],
[(5, 10, 11)],
[(5, 10, 13)],
[(5, 10, 15)],
[(5, 11, 12)],
[(5, 11, 14)],
[(5, 11, 15)],
[(5, 12, 13)],
[(5, 12, 14)],
[(5, 12, 15)],
[(5, 13, 14)],
[(5, 13, 15)],
[(5, 14, 15)],
[(6, 7, 9), (6, 9, 13)],
[(6, 7, 13), (7, 9, 13)],
[(6, 8, 10), (6, 8, 11), (8, 9, 11)],
[(6, 8, 12)],
[(6, 8, 13), (7, 8, 9)],
[(6, 8, 14)],
[(6, 8, 15)],
[(6, 9, 10)],
[(6, 9, 12)],
[(6, 9, 14)],
[(6, 9, 15)],
[(6, 10, 12), (6, 11, 12)],
[(6, 10, 13)],
[(6, 10, 14), (6, 11, 14)],
[(6, 10, 15), (6, 11, 15)],
[(6, 11, 13), (7, 9, 11)],
[(6, 12, 13)],
[(6, 12, 14)],
[(6, 12, 15)],
[(6, 13, 14)],
[(6, 13, 15)],
[(6, 14, 15)],
[(7, 9, 10)],
[(7, 9, 12)],
[(7, 9, 14)],
[(7, 9, 15)],
[(7, 10, 13)],
[(7, 12, 13)],
[(7, 12, 14)],
[(7, 13, 14)],
[(7, 13, 15)],
[(8, 9, 10)],
[(8, 9, 15)],
[(8, 10, 13)],
[(8, 10, 15), (8, 11, 15)],
[(8, 12, 15), (8, 14, 15)],
[(8, 13, 15)],
```

```
[(9, 10, 11)],
[(9, 10, 13)],
[(9, 10, 15)],
[(9, 11, 15)],
[(9, 12, 15), (9, 14, 15)],
[(9, 13, 15)],
[(10, 11, 13)],
[(10, 12, 13), (10, 13, 14)],
[(10, 12, 15), (10, 14, 15), (11, 12, 15), (11, 14, 15)],
[(10, 13, 15)],
[(11, 13, 15)],
[(12, 13, 15), (13, 14, 15)]]
```