# Chow Polynomial of Braid-5

First, we initialize the matroid Braid-5 and then obtain the lattice of flats. We relabel the flats for simple presentation.

[543]:
```
n = 5
edgelist = sorted(graphs.CompleteGraph(n).edges(labels=False))
matroid = Matroid(graph=edgelist, groundset=edgelist)
flats = [list(matroid.flats(i)) for i in range(n)]
all_elements = sum(flats, [])  # Flatten the list of flats
labels = {element: idx for idx, element in enumerate(all_elements)}
matroid.lattice_of_flats().plot(
    element_labels = labels, element_color = "white",
    figsize= 12, cover_color = "grey")
```

[543]:
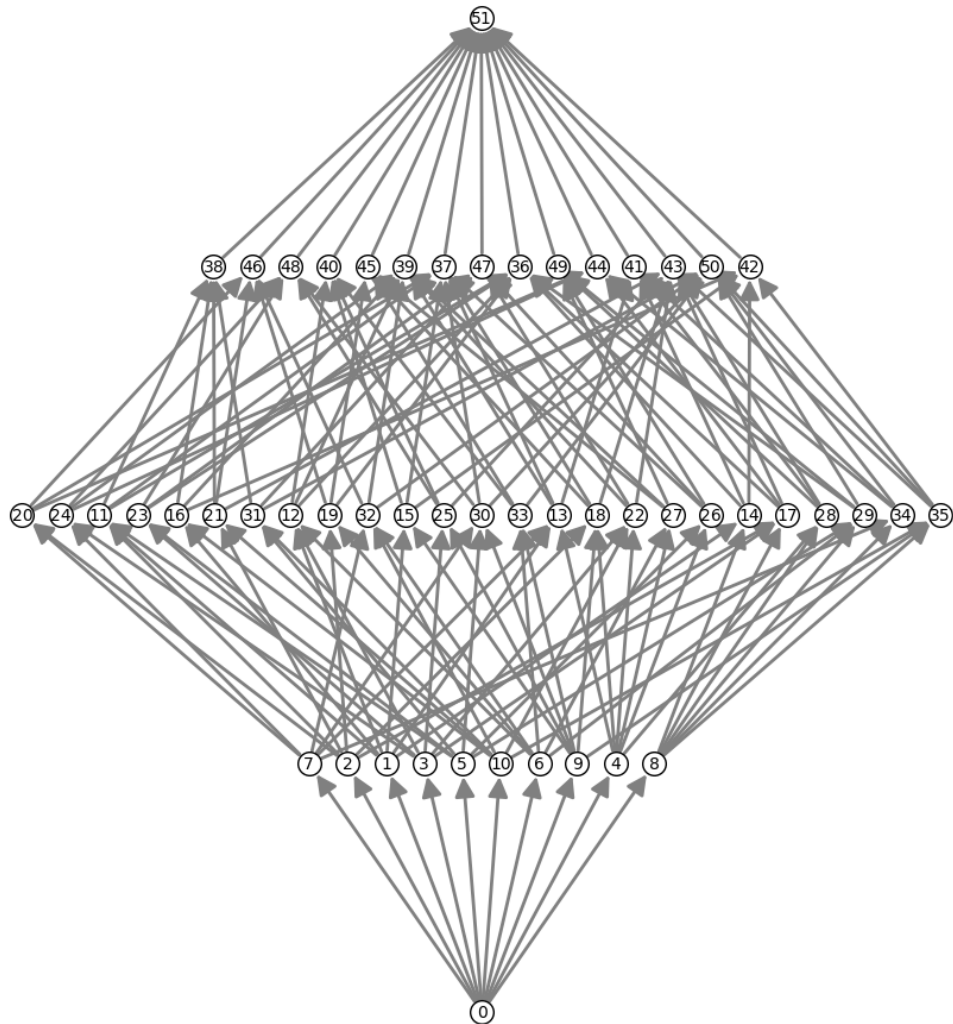
We now generate the possible degrees that a flat can have in a monomial. For example, if

1

$[x_1, x_2, x_3, x_4]$ is a chain and $[0, 1, 0, 2]$ is a weight, then we say that $x_2 x_4^2$ is an fy-monomial.

```
[544]: def generate_weights(rank):
           weights = set()
           for i in range(1, rank):
               for j in range(rank):
                   weight = [0] * rank

                   if i >= j:
                       weight[i] = j
                       weights.add(tuple(weight))

                   if rank - (i+1) > 1:
                       # Recursion to get the complete list of weights...
                       y = generate_weights(rank - (i + 1))
                       for x in y:
                           temp_weight = weight.copy()
                           weights.add(tuple(temp_weight[:i+1] + x))

           return [list(w) for w in weights]

       rank = matroid.rank()
       weights = generate_weights(rank)
       weights
```

```
[544]: [[0, 0, 0, 1],
        [0, 1, 0, 1],
        [0, 0, 1, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 3],
        [0, 0, 0, 0],
        [0, 0, 2, 0],
        [0, 0, 0, 2]]
```

Next, we compute all the possible fy-monomials in the example matroid.

```
[545]: rflats = flats[1:] #empty flat is not required.
       fy_monomials_list = [[] for _ in range(rank)]

       def generate_monomials(weight, flats):
           if sum(weight) == 0: return [[]]
           # Find the first non-zero weight
           start_index = next(i for i in range(len(weight)) if weight[i] != 0)
           result = []

           for initial_flat in flats[start_index]:
               initial_monomial = [initial_flat] * weight[start_index]
               potential_combinations = [initial_monomial]
```

```
            for i in range(start_index + 1, len(weight)):
                if weight[i] != 0:
                    new_combinations = []
                    for flat in flats[i]:
                        for combo in potential_combinations:
                            if combo[-1].issubset(flat):
                                new_combo = combo + [flat] * weight[i]
                                new_combinations.append(new_combo)
                    potential_combinations = new_combinations

            result.extend(potential_combinations)

    return result

for weight in weights:
    fy_monomials_list[sum(weight)].extend(generate_monomials(weight, rflats))

#example fy-monomial
print(fy_monomials_list[2][0])
```

```
[frozenset({(0, 1), (0, 2), (1, 2)}), frozenset({(0, 1), (2, 4), (1, 2), (0, 4),
(3, 4), (0, 3), (1, 4), (2, 3), (0, 2), (1, 3)})]
```

We write some simplification functions to make fy-monomials look simpler. Then we write out all fy-monomials of braid-5 in this simple form.

[546]:
```
def simplify(monomial):
    return tuple(sorted([labels[x] for x in monomial]))

def set_simplify(monomial_set):
    return set(sorted([simplify(x) for x in monomial_set]))
```

The symmetric group $G = S_5$ acts on the example matroid. We now set up the appropriate functions required to compute the actions on the vertex set and obtain the set of orbits and their stabilizer groups.

[547]:
```
G = SymmetricGroup(range(n))

def action_on_flats(g, m):
    def action_on_groundset(g, x):
        return tuple(sorted(g(y) for y in x))
    return frozenset(sorted([action_on_groundset(g,x) for x in m]))

def action_on_braidfymonomials(g, monomial):
    return tuple(sorted([action_on_flats(g,m) for m in monomial]))

def stab(G, m, action):
```

```
        return G.subgroup(set(g for g in G if action(g, m) == tuple(m)))

def orbit(G, m, action):
    return frozenset(sorted(action(g, m) for g in G))

def orbits(G, X, action):
    return set(orbit(G, x, action) for x in X)
```

Finally, we compute the orbits of the fy-monomials under the action of G.

```
[548]: fy_monomials_orbits = [orbits(G, fy_monomials_list[i],␣
       ↪action_on_braidfymonomials)
           for i in range(rank)
           ]

       orbits_dict = {}
       for idx, orbit_set in enumerate(fy_monomials_orbits):
           print(f"\nrank: {idx}")
           for x in orbit_set:
               orbits_dict[x] = len(orbits_dict)
               print(f"{orbits_dict[x]}: {set_simplify(x)}\n")
```

```
rank: 0
0: {()}


rank: 1
1: {(41,), (47,), (40,), (49,), (46,), (42,), (45,), (48,), (38,), (44,)}

2: {(28,), (31,), (15,), (21,), (34,), (24,), (14,), (27,), (33,), (23,), (20,),
(26,), (16,), (19,), (25,)}

3: {(12,), (18,), (35,), (11,), (30,), (17,), (29,), (13,), (32,), (22,)}

4: {(51,)}

5: {(37,), (50,), (43,), (36,), (39,)}


rank: 2
6: {(51, 51)}

7: {(46, 46), (45, 45), (49, 49), (38, 38), (44, 44), (48, 48), (41, 41), (47,
47), (40, 40), (42, 42)}

8: {(29, 51), (35, 51), (11, 51), (30, 51), (17, 51), (12, 51), (18, 51), (13,
51), (32, 51), (22, 51)}
```

4

```
9: {(25, 51), (34, 51), (21, 51), (16, 51), (19, 51), (14, 51), (24, 51), (20,
51), (15, 51), (23, 51), (26, 51), (27, 51), (28, 51), (31, 51), (33, 51)}

10: {(43, 43), (36, 36), (39, 39), (37, 37), (50, 50)}


rank: 3
11: {(51, 51, 51)}
```

Now, we compute the stabilizer of each orbit fy_monomials_orbits(rank) for all ranks.

```
[549]: for orbits in fy_monomials_orbits:
           fn = ClassFunction(G, [0] * len(G.conjugacy_classes()))
           orbits_simplify = {orbits_dict[orbit] for orbit in orbits}

           for orbit in orbits:
               orbit_num = orbits_dict[orbit]
               orbit_stab = stab(G, tuple(orbit)[0], action_on_braidfymonomials)
               print(f"Orbit {orbit_num} is stabilized by a subgroup of order␣
        ↪{orbit_stab.order()}")
               fn += orbit_stab.trivial_character().induct(G)

           print(f"\nThe permutation representation for {orbits_simplify} is")
           for x, y in fn.decompose():
               print(f"{x} copies of {list(y.values())}")
           print("\n")
```

```
Orbit 0 is stabilized by a subgroup of order 120

The permutation representation for {0} is
1 copies of [1, 1, 1, 1, 1, 1, 1]


Orbit 1 is stabilized by a subgroup of order 12
Orbit 2 is stabilized by a subgroup of order 8
Orbit 3 is stabilized by a subgroup of order 12
Orbit 4 is stabilized by a subgroup of order 120
Orbit 5 is stabilized by a subgroup of order 24

The permutation representation for {1, 2, 3, 4, 5} is
5 copies of [1, 1, 1, 1, 1, 1, 1]
4 copies of [4, 2, 0, 1, -1, 0, -1]
1 copies of [5, -1, 1, -1, -1, 1, 0]
3 copies of [5, 1, 1, -1, 1, -1, 0]
```

```
Orbit 6 is stabilized by a subgroup of order 120
Orbit 7 is stabilized by a subgroup of order 12
Orbit 8 is stabilized by a subgroup of order 12
Orbit 9 is stabilized by a subgroup of order 8
Orbit 10 is stabilized by a subgroup of order 24

The permutation representation for {6, 7, 8, 9, 10} is
5 copies of [1, 1, 1, 1, 1, 1, 1]
4 copies of [4, 2, 0, 1, -1, 0, -1]
1 copies of [5, -1, 1, -1, -1, 1, 0]
3 copies of [5, 1, 1, -1, 1, -1, 0]


Orbit 11 is stabilized by a subgroup of order 120

The permutation representation for {11} is
1 copies of [1, 1, 1, 1, 1, 1, 1]
```