In [ ]:

```
from collections import defaultdict
from itertools import chain, product
import logging
```

In [ ]:

```
# export

class dict_nGmap:

    def __init__(self, n):

        self.n = n

        self.darts = set()
        self.marks = defaultdict(lambda:defaultdict(lambda:False))
        # self.marks[index][dart]

        ##### Carmine #####

        self.alpha = [dict() for _ in range(n + 1)]
        """
        Variable to keep trace of levels that may be created applying removal/contraction operations.
        In my opinion, I think it is possible to store only one variable like that to keep trace, for
        each alpha, of the current level.
        """
        self.level = 0

        """
            The real number of the pyramid is obtained if We know how much worth the reduction factor.
            For the tests, I have set a random value.
        """
        self.n_layers = 7

        ##### Carmine #####

        self.taken_marks = {-1}
        self.lowest_unused_dart_index = 0

    @classmethod
    def from_string(cls, string):
        lines = string.strip().split("\n")
        lines = [[int(k) for k in l.split(" ") if k != ""] for l in lines]

        return cls.from_list_of_lists(lines)

    @classmethod
    def from_list_of_lists(cls, ll):
        n = len(ll) - 1
        d = len(ll[0])

        darts = set(ll[0])

        assert all(set(l) == darts for l in ll)

        my_nGmap = cls(n)
        my_nGmap.darts.update(darts)

        #for alpha, l in zip(my_nGmap.alpha, ll):
        for alpha, l in zip(my_nGmap.alpha, ll):
            for a, b in zip(sorted(darts), l):

                ##### Carmine #####
                """
                    Using my implementation there are not more two int as key and value,
                    but I need to have an int key and as value, a list to store the history
                    of the pyramid. So, thus, I initialize the gmap in that way as follow.
                """
                alpha[a] = [b]
```

```
                ##### Carmine #####

        my_nGmap.lowest_unused_dart_index = max(darts) + 1

        return my_nGmap

    @property
    def is_valid(self):
        """
        checks condition 2 and 3 from definition 1
        (condition 1 is always true as a computer probably has finite memory)
        """
        for dart in self.darts:
            for alpha in self.alpha:
                if alpha[alpha[dart]] != dart:
                    return False
            for i in range(0, self.n - 1): # n-1 not included
                alpha1 = self.alpha[i]
                for j in range(i + 2, self.n + 1): # n+1 again not included
                    alpha2 = self.alpha[j]
                    if alpha1[alpha2[alpha1[alpha2[dart]]]] != dart:
                        return False
        return True


    ##### Carmine #####
    """
        I have implemented a custom version of the is_valid method to adapt it
        to the new data structure used.
    """
    @property
    def is_valid_custom(self):

        for dart in self.darts:
            for alpha in self.alpha:


                """
                    The following if it is useful because I have to initialize a certain size of
                    the list with the None value to use it like an array, otherwise, all the
                    insert operations will be like the append method and we can not have
                    the index of the level. In that case, I can insert the
                    element in the list at self.level position in set_ai. Considering that each list ha
s to be initialized with
                    log(n) * None.
                """
                if not alpha:
                    continue


                try:
                    x = next(item for item in alpha[dart] if item is not None)

                    #if dart not in alpha[x]:
                    if alpha[x][-1] != dart:
                        return False
                except KeyError:
                    logging.debug(f'Key {x} has been removed. Of course there is not!')
                    continue


            for i in range(0, self.n - 1): # n-1 not included
                alpha1 = self.alpha[i]
                for j in range(i + 2, self.n + 1): # n+1 again not included
                    alpha2 = self.alpha[j]

                    y = alpha2[dart][0] # -> int, for instance, [2] -> [2][0] -> 2
                    z = alpha1[y] # -> list, for instance, [5, 1]

                    """
                        z is a list. So, we should iterate on that to use all the items.
                    """
                    for a in z:
                        # a -> item of the list -> a = 5, 1
                        """
                            The try-except statement is used to handle the KeyError that is
```

```
                        The try except statement is used to handle the KeyError that is
                        raised when we try to use a value as a key and that key is not
                        in the dict.
                    """
                    try:

                        """
                            The try-except block, I think, is right because in that case we can hav
e
                            keys that are removed from the list because we have removed the corresp
onding
                            dart of the Gmap.
                            For example, we can have that situation:
                            BEFORE -> dict (1 : [2], 2 : [5])
                            AFTER REMOVE DART 1 -> dict (2 : [5, 1])
                        """

                        k = alpha2[a] # list, for instance, [5] or [1] -> [5][0] or [1][0] -> 5 or
1
                    except KeyError:
                        logging.debug(f'Key {a} has been removed. Of course there is not in the lis
t!')
                        continue


                """
                    Eventually, instead of alpha1[k] != dart in the version with interger
                    values, now we check if the dart is in the list.
                    If the dart is not in the list, so, the new gmap is not valid,
                    otherwise, yes.
                """

                if dart not in alpha1[k[0]]:
                    return False

        return True

    ##### Carmine #####

    def reserve_mark(self):
        """
        algorithm 2
        """
        i = max(self.taken_marks) + 1
        self.taken_marks.add(i)
        return i

    def free_mark(self, i):
        """
        algorithm 3
        also includes deleting all these marks, making it safe to call everytime
        """
        del self.marks[i]
        self.taken_marks.remove(i)

    def is_marked(self, d, i):
        """
        algorithm 4
        d ... dart
        i ... mark index
        """
        return self.marks[i][d]

    def mark(self, d, i):
        """
        algorithm 5
        d ... dart
        i ... mark index
        """
        self.marks[i][d] = True

    def unmark(self, d, i):
        """
        algorithm 6
        d ... dart
        i ... mark index
        """
```

```
        # same as bla = False
        del self.marks[i][d]

    def mark_all(self, i):
        for d in self.darts:
            self.mark(d, i)

    def unmark_all(self, i):
        del self.marks[i]

    def ai(self, i, d):
        return self.alpha[i][d][0]

    def set_ai(self, i, d, d1):

        assert 0 <= i <= self.n

        ##### Carmine #####

        """
        I am inserting to position (self.level - 1) within self.levels, the dart d1.
        """
        #self.levels.insert(self.level - 1, d1)
        #print(f'\ncurrent level: {self.level} in the function set_ai')
        """
        With alpha[i] we are selecting the alpha we have to modify by inserting the new information.
        """
        if d in self.alpha[i]:
            self.alpha[i][d].insert(self.level - 1, d1)
        else:
            """ the next line of the code is necessary because, otherwise, i cannot insert
                in a certain position within the list. So, I had to create a list with None values
                of size n. The size is n because at the moment I do not have idea of how to set
                the fixed size. Basically, the size of the list should be log(n), at most.
            """
            self.alpha[i][d] = [None] * self.n_layers
            self.alpha[i][d].insert(self.level - 1, d1)


    def _remove_dart(self, d):
        self.darts.remove(d)
        for i in self.all_dimensions:
            del self.alpha[i][d]

    def orbit(self, sequence, d):
        """
        algorithm 7
        sequence ... valid list of dimensional indices
        d ... dart
        """
        ma = self.reserve_mark()
        self.mark_orbit(sequence, d, ma)
        orbit = self.marks[ma].keys()
        self.free_mark(ma)
        return orbit

    def mark_orbit(self, sequence, d, ma):
        """
        used as in algorithm 7 and 8 etc...
        sequence ... valid list of dimension indices
        d ... dart
        ma ... mark to use
        """


        P = [d]
        self.mark(d, ma)
        while P:
            cur = P.pop()
            for i in sequence:
                ##### Carmine #####

                try:
                    other = self.alpha[i][cur][-1]
                    if not self.is_marked(other, ma):
```

```
                    self.mark(other, ma)
                    P.append(other)
            except KeyError:
                logging.debug(f'Key {cur} has been removed/contracted.')
                continue

            ##### Carmine #####

    def cell_i(self, i, dart):
        """iterator over i-cell of a given dart"""
        return self.orbit(self.all_dimensions_but_i(i), dart)

    def cell_0(self, dart):
        return self.cell_i(0, dart)
    def cell_1(self, dart):
        return self.cell_i(1, dart)
    def cell_2(self, dart):
        return self.cell_i(2, dart)
    def cell_3(self, dart):
        return self.cell_i(3, dart)
    def cell_4(self, dart):
        return self.cell_i(4, dart)

    def no_i_cells (self, i=None):
        """
        Counts
            i-cells,             if 0 <= i <= n
            connected components if i is None
        """
        assert i is None or 0 <= i <= self.n
        # return more_itertools.ilen (self.darts_of_i_cells(i))
        return sum ((1 for d in self.darts_of_i_cells(i)))

    @property
    def no_0_cells (self): return self.no_i_cells (0)
    @property
    def no_1_cells (self): return self.no_i_cells (1)
    @property
    def no_2_cells (self): return self.no_i_cells (2)
    @property
    def no_3_cells (self): return self.no_i_cells (3)
    @property
    def no_4_cells (self): return self.no_i_cells (4)
    @property
    def no_ccs     (self): return self.no_i_cells ( )

    def darts_of_i_cells(self, i = None):
        """
        algorithm 8
        """
        ma = self.reserve_mark()
        try:
            for d in self.darts:
                if not self.is_marked(d, ma):
                    yield d
                    self.mark_orbit(self.all_dimensions_but_i(i), d, ma)
        finally:
            self.free_mark(ma)

    def all_i_cells(self, i = None):
        for d in self.darts_of_i_cells(i):
            yield self.cell_i(i, d)

    def all_conected_components(self):
        return self.all_i_cells()

    def darts_of_i_cells_incident_to_j_cell(self, d, i, j):
        """
        algorithm 9
        """
        assert i != j
        ma = self.reserve_mark()
        try:
            for e in self.orbit(self.all_dimensions_but_i(j), d):
                if not self.is_marked(e, ma):
```

```
                yield e
                self.mark_orbit(self.all_dimensions_but_i(j), e, ma)
    finally:
        self.free_mark(ma)

def darts_of_i_cells_adjacent_to_i_cell(self, d, i):
    """
    algorithm 10
    """
    ma = self.reserve_mark()
    try:
        for e in self.orbit(self.all_dimensions_but_i(i), d):
            f = self.alpha[i][e]
            if not self.is_marked(f, ma):
                yield f
                self.mark_orbit(self.all_dimensions_but_i(i), f, ma)
    finally:
        self.free_mark(ma)

def all_dimensions_but_i (self, i=None):
    """Return a sorted sequence [0,...,n], without i, if 0 <= i <= n"""
    assert i is None or 0 <= i <= self.n
    return [j for j in range (self.n+1) if j != i]

@property
def all_dimensions(self):
    return self.all_dimensions_but_i()

def is_i_free(self, i, d):
    """
    definiton 2 / algorithm 11
    """
    return self.alpha[i][d] == d

def is_i_sewn_with(self, i, d):
    """
    definiton 2
    """
    d2 = self.alpha[i][d]
    return d != d2, d2

def create_dart(self):
    """
    algorithm 12
    """
    d = self.lowest_unused_dart_index
    self.lowest_unused_dart_index += 1
    self.darts.add(d)

    ##### Carmine #####
    """
        I did not take care about that method, but if I need to use it I can adapt it
        as already done with the set_ai method. Both methods are similar.
    """
    ##### Carmine #####
    for alpha in self.alpha:
        alpha[d] = d
    return d

def remove_isolated_dart(self, d):
    """
    algorithm 13
    """
    assert self.is_isolated(d)
    self.remove_isolated_dart_no_assert(d)

def remove_isolated_dart_no_assert(self, d):
    self.darts.remove(d)
    for alpha in self.alpha:
        del alpha[d]

def is_isolated(self, d):
    for i in range(self.n + 1):
        if not self.is_i_free(i, d):
            return False
```

```
        return True

    def increase_dim(self):
        """
        algorithm 15 in place
        """
        self.n += 1
        self.alpha.append(dict((d,d) for d in self.darts))

    def decrease_dim(self):
        """
        algorithm 16 in place
        """
        assert all(self.is_i_free(self.n, d) for d in self.darts)
        self.decrease_dim_no_assert()

    def decrease_dim_no_assert(self):
        del self.alpha[self.n]
        self.n -= 1

    def index_shift(self, by):
        self.darts = {d + by for d in self.darts}
        self.alpha = [{k + by : v + by for k, v in a.items()} for a in self.alpha]

        for mark in self.marks:
            new_dict = {key + by : value for key, value in self.marks[mark].items()}
            self.marks[mark].clear()
            self.marks[mark].update(new_dict) #this is done to preserve default dicts
        self.lowest_unused_dart_index += by

    def merge(self, other):
        """
        algorithm 17 in place
        """
        assert self.n == other.n
        self.taken_marks.update(other.taken_marks)
        shift = max(self.darts) - min(other.darts) + 1
        other.index_shift(shift)

        self.darts.update(other.darts)
        print(f'alpha: {self.alpha}\nother: {other.alpha}\n')
        for sa, oa in zip(self.alpha, other.alpha):
            sa.update(oa)
            print(f'sa: {sa}\n')
            print(f'oa: {oa}\n')
        for mk in other.marks:
            self.marks[mk].update(other.marks[mk])
        self.taken_marks.update(other.taken_marks)
        self.lowest_unused_dart_index = other.lowest_unused_dart_index

    def restrict(self, D):
        """
        algorithm 18
        """
        raise NotImplementedError #boring

    def sew_seq(self, i):
        """
        indices used in the sewing operations
        (0, ..., i - 2, i + 2, ..., n)
        """
        return chain(range(0, i - 1), range(i + 2, self.n + 1))

    def sewable(self, d1, d2, i):
        """
        algorithm 19
        tests wether darts d1, d2 are sewable along i
        returns bool
        """
        if d1 == d2 or not self.is_i_free(i, d1) or not self.is_i_free(i, d2):
            return False
        try:
            f = dict()
            for d11, d22 in strict_zip(self.orbit(self.sew_seq(i), d1), self.orbit(self.sew_seq(i), d2)
, strict = True):
```

```
                f[d11] = d22
                for j in self.sew_seq(i):
                    if self.alpha[j][d11] in f and f[self.alpha[j][d11]] != self.alpha[j][d22]:
                        return False
        except ValueError: #iterators not same length
            return False
        return True

    def sew(self, d1, d2, i):
        """
        algorithm 20
        """
        assert self.sewable(d1, d2, i)
        self.sew_no_assert(d1, d2, i)

    def sew_no_assert(self, d1, d2, i):
        for e1, e2 in strict_zip(self.orbit(self.sew_seq(i), d1), self.orbit(self.sew_seq(i), d2), stri
ct = True):
            self.alpha[i][e1] = e2
            self.alpha[i][e2] = e1

    def unsew(self, d, i):
        """
        algorithm 21
        """
        assert not self.is_i_free(i, d)
        for e in self.orbit(self.sew_seq(i), d):
            f = self.alpha[i][e]
            self.alpha[i][f] = f
            print(self.alpha[i][f])
            self.alpha[i][e] = e
            print(self.alpha[i][e])

    def incident (self, i, d1, j, d2):
        """
        checks wether the i-cell of d1 is incident to the j-cell of d2
        """
        for e1, e2 in product(self.cell_i(i, d1), self.cell_i(j, d2)):
            if e1 == e2:
                return True
        return False

    def adjacent(self, i, d1, d2):
        """
        checks wether the i-cell of d1 is adjacent to the i-cell of d2
        """
        first_cell = self.cell_i(i, d1)
        second_cell = set(self.cell_i(i, d2))
        for d in first_cell:
            ##### Carmine #####
            """
                Only adpat the terms in the if statement to work with a list.
            """
            if self.alpha[i][d][-1] in second_cell:
                ##### Carmine #####
                return True
        return False

    # Contractablilty & Removability

    def _is_i_removable_or_contractible(self, i, dart, rc):
        """
        Test if an i-cell of dart is removable/contractible:

        i    ... i-cell
        dart ... dart
        rc   ... +1 => removable test, -1 => contractible test
        """

        assert dart in self.darts
        assert 0 <= i <= self.n
        assert rc in {-1, +1}

        if rc == +1:  # removable test
            if i == self.n  : return False
            if i == self.n-1: return True
```

```
            if i == self.n-1: return True
        if rc == -1:  # contractible test
            if i == 0: return False
            if i == 1: return True

        ##### Carmine #####
        """
            I have just adpated the code to work with a list instead of int values.
        """
        for d in self.cell_i(i, dart):
            if self.alpha[i+rc][self.alpha[i+rc+rc][d][0]] != self.alpha[i+rc+rc][self.alpha[i+rc][d][0
]]:
                return False
        return True
        ##### Carmine #####

    def is_i_removable(self, i, dart):
        """True if i-cell of dart can be removed"""
        return self._is_i_removable_or_contractible(i, dart, rc=+1)

    def is_i_contractible(self, i, dart):
        """True if i-cell of dart can be contracted"""
        return self._is_i_removable_or_contractible(i, dart, rc=-1)

    def _i_remove_contract(self, i, dart, rc, skip_check=False):
        """
        Remove / contract an i-cell of dart
        d  ... dart
        i  ... i-cell
        rc ... +1 => remove, -1 => contract
        skip_check ... set to True if you are sure you can remove / contract the i-cell
        """
        logging.debug (f'{"Remove" if rc == 1 else "Contract"} {i}-Cell of dart {dart}')

        if not skip_check:
            assert self._is_i_removable_or_contractible(i, dart, rc),\
                f'{i}-cell of dart {dart} is not {"removable" if rc == 1 else "contractible"}!'

        i_cell = set(self.cell_i(i, dart))  # mark all the darts in ci(d)
        logging.debug (f'\n{i}-cell to be removed {i_cell}')
        for d in i_cell:
            d1 = self.ai (i,d) # d1 ← d.Alphas[i];
            if d1 not in i_cell:  # if not isMarkedNself(d1,ma) then
                # d2 ← d.Alphas[i + 1].Alphas[i];
                d2 = self.ai (i+rc,d)
                d2 = self.ai (i   ,d2)
                while d2 in i_cell: # while isMarkedNself(d2,ma) do
                    # d2 ← d.Alphas[i + 1].Alphas[i];
                    d2 = self.ai (i+rc,d2)
                    d2 = self.ai (i   ,d2)
                logging.debug (f'Modifying alpha_{i} of dart {d1} from {self.ai (i,d1)} to {d2}')


                ##### Carmine #####
                """
                    We can increase the 'level' variable here because the check is skipped.
                    So, it means that the operation can be done and it is useful to insert
                    the new element in the right position within the array.
                """
                self.level += 1
                ##### Carmine #####

                self.set_ai(i,d1,d2) # d1.Alphas[i] ← d2;
        for d in i_cell:  # foreach dart d' ∈ ci(d) do
            self._remove_dart (d)  # remove d' from gm.Darts;


    ##### Carmine #####
    """
        These are utily methods to check if the items are at the right position in the data structure.
    """
    def print_alpha(self, alpha_index, key_index, level_index):
        return self.alpha[alpha_index][key_index][level_index - 1]

    def print_all(self):
        i = -1
```

```
        for a in self.alpha:
            i += 1
            print(f'Alpha{i} -> {a}\n')

    ##### Carmine #####

    def _remove(self, i, dart, skip_check=False):
        """Remove i-cell of dart"""
        self._i_remove_contract(i, dart, rc=+1, skip_check=skip_check)

    def _contract(self, i, dart, skip_check=False):
        """Contract i-cell of dart"""
        self._i_remove_contract(i, dart, rc=-1, skip_check=skip_check)

    def __repr__(self):
        out = f"{self.n}dGmap of {len(self.darts)} darts:\n"
        for i in range(self.n + 1):
            out += f" {i}-cells: {self.no_i_cells(i)}\n"
        out += f" ccs: {self.no_ccs}"
        return out

def strict_zip(arg1, arg2, strict = False):
    """
    strict keyword for zip is only avaliable in python 3.10 which is still in alpha :(
    """
    assert strict == True
    arg1 = list(arg1)
    arg2 = list(arg2)
    if len(arg1) == len(arg2):
        return zip(arg1, arg2)
    else:
        raise ValueError
```

Tests using G2_TWO_TRIANGLES_1

In [ ]:

```
from combinatorial.notebooks.combinatorial.zoo import G2_HOUSE_1, G2_TWO_TRIANGLES_1, G2_345_BOUNDED_1


m = dict_nGmap.from_string(G2_TWO_TRIANGLES_1)

m
```

Out[ ]:

```
2dGmap of 12 darts:
 0-cells: 6
 1-cells: 6
 2-cells: 2
 ccs: 2
```

In [ ]:

```
# Meaning of the parameters of the remove method: type of i-cell, dart
m._remove(0, 1)
m.is_valid_custom
```

Out[ ]:

```
True
```

In [ ]:

```
m._remove(0, 2)
m.is_valid_custom
```

Out[ ]:

Out[ ]:

True

In [ ]:

```
m._contract(1, 8)
m.is_valid_custom
```

Out[ ]:

True

In [ ]:

```
m
```

Out[ ]:

```
2dGmap of 6 darts:
 0-cells: 3
 1-cells: 3
 2-cells: 2
 ccs: 2
```

Test using G2_butterfly_16

In [ ]:

```
#    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16    17 18 19 20 21 22    23 24 25 26 27 28
G2_butterfly_16 = """\
     2  1  4  3  6  5  8  7 10  9 12 11 14 13 16 15    18 17 20 19 22 21    24 23 26 25 28 27
    16  3  2  5  4  7  6  9  8 11 10 13 12 15 14  1    22 19 18 21 20 17    28 25 24 27 26 23
    10  9 23 24 25 26 27 28  2  1 22 21 20 19 18 17    16 15 14 13 12 11     3  4  5  6  7  8
"""

Bridge = dict_nGmap.from_string (G2_butterfly_16)
Bridge
```

Out[ ]:

```
2dGmap of 28 darts:
 0-cells: 6
 1-cells: 7
 2-cells: 3
 ccs: 1
```

In [ ]:

```
Bridge._contract(1,1)
Bridge.is_valid_custom
```

Out[ ]:

True

In [ ]:

```
Bridge
```

Out[ ]:

```
2dGmap of 24 darts:
 0-cells: 5
 1-cells: 6
 2-cells: 3
 ccs: 1
```