Equivalence Classes and Set Partitions

Ernesto P. Adorio

1 Equivalence Classes

An equivalence relation $\equiv$ on a set is a relation which is reflexive (for all $x \in S, x \equiv x$), symmetric (for all $x, y \in S, if x \equiv y$, then $y \equiv x$) and transitve (for $x, y, z \in S$, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$). An equivalence relation on a set of n objects partitions the set into equivalence classes.

We implement equivalence relations on a set with n elements in Python using a simple list of class numbers in the module `equiv.py`. The class number of an element at position i is the index of the representative of the class which contains the given element.

To declare an equivalence relation on 10 objects and print the contents of the equivalence array, we write

```
>>> import equiv
>>> A = equiv.equiv(10, 0)
>>> print A
equiv[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that each element is initialized to be in its own equivalence class. An equivalence pair (i, j) means that objects having positions i and j must be put in the same equivalence class. Suppose we add the equivalence pairs $(3, 5), (3, 7)(2, 4)$ and $(8, 9)$.

```
>>> A.Add(3, 5)
3
>>> A.Add(3, 7)
3
>>> A.Add(2, 4)
2
>>> A.Add(8, 9)
8
>>> print A
equiv[0, 1, 2, 3, 2, 3, 6, 3, 8, 8]
```

The numbers printed by *Add*() routine is the index of the representative of the class. The above sequence of operations are equivalent to the series of commands

```
>>> A.AddList([3,5,7])
>>> A.AddList([2,4])
>>> A.AddList([8,9))
```

or to the crisp form

```
>>> A.AddLists([3,5,7], [2,4], [8,9])
```

To determine the number of classes, we call the $nClasses()$ method.

```
>>> A.nClasses()
6
```

To see the equivalence classes clearly:

```
>>> A.ClassList()
[[0], [1], [2, 4], [3, 5, 7], [6], [8, 9]]
```

If we want to convert the representation to a partition form:

```
>>> B = A.EquivToPartition()
>>> print B
part[0, 1, 2, 3, 2, 3, 4, 3, 5, 5]
```

We want to recover the original equivalence array:

```
>>> A = B.PartitionToEquiv()
>>> print A
equiv[0, 1, 2, 3, 2, 3, 6, 3, 8, 8]
```

Suppose we add the pair $(1, 2)$ to A.

```
>>> A.Add(1,2)
1
>>> print A
equiv [0, 1, 1, 3, 2, 3, 6, 3, 8, 8]
```

Notice that the member with position index 4, still has its class number set to 2. To set the class number to the class representative, we reduce the equivalence array:

```
>>> A.Reduce()
>>> print A
equiv[0, 1, 1, 3, 1, 3, 6, 3, 8, 8]
```

2 Set Partitions

Partitions have applications in combinatorial mathematics and in the study of symmetrical colorings. The entries in a partition array c satisfy the following properties:

P1. c[0] = 0
P2. c[i] ¡= 1 + max(c[0], c[1], ..., c[i-1]) for 0 ¡ i ¡ m
P3. max(c[i]) = m-1

We provide routines for generating partitions in a non-recursive manner. In the following session, we list all partitions of size 5 with 3 classes.

```
>>> A.FirstPartition(3)
K =  2
>>> i = 1
>>> while 1:
...   if not A.NextPartition(): break
...   print i, A
...   i += 1
...
1 part[0, 0, 1, 0, 2]
2 part[0, 0, 1, 1, 2]
3 part[0, 0, 1, 2, 0]
4 part[0, 0, 1, 2, 1]
5 part[0, 0, 1, 2, 2]
6 part[0, 1, 0, 0, 2]
7 part[0, 1, 0, 1, 2]
8 part[0, 1, 0, 2, 0]
9 part[0, 1, 0, 2, 1]
10 part[0, 1, 0, 2, 2]
11 part[0, 1, 1, 0, 2]
12 part[0, 1, 1, 1, 2]
13 part[0, 1, 1, 2, 0]
14 part[0, 1, 1, 2, 1]
15 part[0, 1, 1, 2, 2]
16 part[0, 1, 2, 0, 0]
17 part[0, 1, 2, 0, 1]
18 part[0, 1, 2, 0, 2]
19 part[0, 1, 2, 1, 0]
20 part[0, 1, 2, 1, 1]
21 part[0, 1, 2, 1, 2]
22 part[0, 1, 2, 2, 0]
23 part[0, 1, 2, 2, 1]
24 part[0, 1, 2, 2, 2]
```

There were 1 initial + 24 additional partitions generated, but beware! The total number of partitions of size n = 20 with k = 1,2, ... ,20 classes is given by the Bell number Bell(20) = 51,724,158,235,372. Formulas for combinatorial numbers such as Bell number and others will be discussed in a future article.

3 Using starting index of 1

The above example used a starting index of 0. Some users are more comfortable with a starting index of 1 and where the class numbers vary from 1 to n, where n is the maximum number of classes. The routines in equiv.py can also handle this case. Users will simply initialize an equivalence object by simply setting the base to 1 when initializing an object by calling equiv.equiv(size, base). When using a base of 1, the array position of 0 is not used. Here is the result of listing all the partitions of size 5 and with 3 classes using a base of 1.

```
>>> A = equiv.equiv(5, 1)
>>> A.FirstPartition(3)
K =  3
>>> i = 1
>>> while 1:
...     if not A.NextPartition(): break
...     print i, A
...     i += 1
...
1 part[1, 1, 2, 1, 3]
2 part[1, 1, 2, 2, 3]
3 part[1, 1, 2, 3, 1]
4 part[1, 1, 2, 3, 2]
5 part[1, 1, 2, 3, 3]
6 part[1, 2, 1, 1, 3]
7 part[1, 2, 1, 2, 3]
8 part[1, 2, 1, 3, 1]
9 part[1, 2, 1, 3, 2]
10 part[1, 2, 1, 3, 3]
11 part[1, 2, 2, 1, 3]
12 part[1, 2, 2, 2, 3]
13 part[1, 2, 2, 3, 1]
14 part[1, 2, 2, 3, 2]
15 part[1, 2, 2, 3, 3]
16 part[1, 2, 3, 1, 1]
17 part[1, 2, 3, 1, 2]
18 part[1, 2, 3, 1, 3]
19 part[1, 2, 3, 2, 1]
20 part[1, 2, 3, 2, 2]
21 part[1, 2, 3, 2, 3]
```

```
22 part[1, 2, 3, 3, 1]
23 part[1, 2, 3, 3, 2]
24 part[1, 2, 3, 3, 3]
```

4 Summary, To dos and Source codes

The `equiv.py` module for processing equivalences and generating partitions have the following routines available:

Add(i, j)	Adds relation i == j.
AddList(list)	Adds a list of relations.
AddLists(lists)	Adds a list of relatiion lists.
Auxiliary()	Returns a restricted growth function list.
Copy()	Returns a clone object.
ClassList()	Returns a list of all classes.
ClassNumber(i)	Returns the index of class representative.
equiv(size, base)	Returns an equivalence object of maximum size. using starting base index (either 0 or 1).
EquivToParition()	Converts an equiv class to a partition class.
nClasses()	Returns the number of classes.
PartitionToEquiv()	Converts a partition class to an equiv class.
Reduce()	Sets the class number to the class rep.
Representatives()	Returns list of class representatives.
FirstPartition(nclasses)	Generates the first partition with nclasses.
PreviousPartition()	Generates the previous partition.
NextPartition()	Generates the next partition.
LastPartition(nclasses)	Generates the last partition.

There are some things missing in this current work, the most important are ranking and unranking of partitions. We hope to remedy this in the future.

We now present the file equiv.py with all its warts and glory. Your constructive critical comments will be most welcome.

```
"""
equiv.py - module for processing equivalences and partitions

Author/programmer
  Ernesto P. Adorio, Ph.D.
  Department of Mathematics
  University of the Philippines

Email
   adorio@math.dontspam.upd.edu.ph

Version
   0.1.0 - jan/20/2004 released
```

```
   0.1.1 - feb/02/2004
           Added attribute baseindex field (0, or 1). This allows
           users who are comfortable with array indices 1 to n
           rather than 0 to n-1.

           feb/03/2004
           Add PreviousPartition() to traverse backwards.
           Revised NextPartition().

   There will be no concerted attempt to preserve copies of earlier versions.

References
   1. Horowitz and Sahni, "Fundamentals of Data Structures",
         Computer Science Press, 1976, pp. 256-257

   2. Ruskey, F. "Info About Set Partitions."
         http://www.theory.csc.uvic.ca/~cos/inf/setp/SetPartitions.html.

   2. Press, et al., 'Numerical Recipes'

   3. Adorio, "Colorings in Quotient Spaces", Ph.D. Dissertation,
         Diliman, University of the Philippines, 2000
"""

import copy

class equiv:
    """
    Routines for processing equivalence class and
    set partition arrays.
    """

    def __init__(self, n = 10, base = 0):
        """
        Initializes the class number of each element

          n -- maximum number of equivalence classes.
          base -- starting index (either 0 or 1).
        """
        self.baseindex = base
        if base == 0: self.c = range(n)
        else: self.c = range(n+1)

        # 0 - equiv class, 1 - a partition class
        self.whattype = 0
```

```
def __str__(self):
    """
    Returns the string representation of an equivalence
    class object.
    """
    if self.whattype == 0:
      return 'equiv' + str(self.c[self.baseindex:])
    else:
      return 'part' + str(self.c[self.baseindex:])

def Copy(self):
    """
    Clones self by performing a deepcopy.
    """
    return copy.deepcopy(self)

def ClassNumber(self, i):
    """
    Returns the index of the class representative of ith element.
    """
    while True:
       t = i
       i = self.c[i]
       if i == t:
           return t

def Representatives(self):
    """
    Returns a list of the class representatives.
    """
    reps   = []
    maxrep = 0
    for i in range(self.baseindex, len(self.c)):
        if self.c[i] > maxrep:
            reps.append(i)
            maxrep = self.c[i]
    return reps

def Add(self, i, j):
    """
    Process the equivalence pair (i, j). The elements with indices
    i and j are to be treated as belonging to the same
    equivalence class.

    Returns the class representative for the pair.
    """
```

```
        x = self.ClassNumber(i)
        y = self.ClassNumber(j)

        if x == y:
           return x

        minVal = min(x,y)

        while True:
            x = i
            i = self.c[x]
            self.c[x] = minVal
            if (i == self.c[i]):
                self.c[i] = minVal
                break

        while True:
            y = j
            j = self.c[y]
            self.c[y] = minVal
            if (j == self.c[j]):
                self.c[j] = minVal
                break

    def AddList(self, list):
        """
        Process a single list of equivalent objects.
        """
        if len(list) > 1:
            x = list[0]
            for y in list[1:]:
                self.Add(x, y)

    def AddLists(self, lists):
        """
        Process a list of lists of equivalent objects.
        """
        for x in lists:
            self.AddList(x)

    def Reduce(self):
        """
        Sets each element of self to its class representative
        which is the leading element (earliest position).
        """
        for i in range(self.baseindex, len(self.c)):
```

```
        self.c[i] = self.ClassNumber(i)

def EquivToPartition(self):
    """
    Returns a partition class object corresponding to
    an equivalence class object.

    Partition class numbers will NOT be interpreted as
    the indices to the class representatives.

    The maximum of self.c[i]s is the number of classes-1.
    This function always call Reduce().
    """
    D = self.Copy()
    D.Reduce()

    if D.whattype == 1:
      return D

    nClasses = 0
    for i in range(D.baseindex, len(D.c)):
        k = self.c[i]
        if (k > nClasses):
           nClasses = nClasses + 1
           j = i
           while j < len(self.c):
               if (D.c[j] == k):
                  D.c[j] = nClasses
               j = j + 1
    D.whattype = 1
    return D

def PartitionToEquiv(self):
    """
    Converts a partition class object to a reduced
    equivalence class object.
    """
    D = self.Copy()
    if self.whattype == 0:
        return D
    pseudo   = len(D.c)
    n = len(D.c)
    for i in range(D.baseindex, n):
        j = D.c[i]
        if j > i or D.c[j] != j:
            # temporarily replace each occurrence of i with pseudo
```

```
            for k in range(i, n):
                if D.c[k] == i:
                    D.c[k] = pseudo

            # replace all occurrence  of j with i
            for  k in  range(i, n):
                if D.c[k] == j:
                    D.c[k] = i
            pseudo = pseudo + 1
    D.whattype = 1
    return D

def nClasses(self):
    """
    Returns the number of equivalence classes defined.
    Does not require self to be in reduced form.
    """
    if len(self.c) == 0:
        return 0

    n = 1
    maxrep = self.c[self.baseindex]
    for i in range(self.baseindex + 1,len(self.c)):
        j = self.ClassNumber(i)
        if j > maxrep:
            n        = n + 1
            maxrep   = j
    return n

def ClassList(self):
    """
    Returns the list of equivalence classes
    """
    R = self.Copy()
    R.Reduce()
    C = []
    for i in range(self.baseindex, len(R.c)):
        x = R.c[i]
        if x != -1:
            S = []
            for j in range(i, len(R.c)):
                if R.c[j] == x:
                   S.append(j)
                   R.c[j] = -1
            C.append(S)
    return C
```

```
# ################ Partitions ####################
def FirstPartition(self, nclasses):
    """
    Computes the first partition with nclasses
      [0, 0, ..., nclasses-3, nclasses-2, nclasses-1] for baseindex = 0
      [0, 1, ...,             nclasses-2, nclasses-1,  nclasses]  for baseindex = 1

    nclasses is checked for correct values.
    """
    n = len(self.c)
    k = nclasses - (1 - self.baseindex)
    k = max(self.baseindex, k)
    k = min(n-1, k)
    print 'K = ', k

    for i in range(n-1,self.baseindex-1,-1):
        self.c[i] = max(self.baseindex, i - (n-k) + 1)
    self.whattype = 1

def PreviousPartition(self):
    """
    Returns True if the previous partition is generated,
    otherwise False. The permutation vector self.c is
    modified by this routine.

    feb/03/2004 - first version  by ernie
    """
    if self.whattype != 1:
      return False

    M = self.Auxiliary()             # restricted grouwth function array
    ifirst =  self.baseindex         # starting index
    ilast  = len(self.c) - 1         # last index value
    mink   = ifirst
    maxk   = M[ilast]

    for i in range (ilast, ifirst, -1):
        if self.c[i] > mink:
            if M[i-1] == maxk:
                self.c[i] -= 1
                for j in range(i+1, ilast+1):
                    self.c[j] = maxk
                return True
            elif i + (maxk - M[i-1] - 1) < ilast:
                self.c[i] -= 1
```

```
                M[i] = max(M[i-1], self.c[i])
                for j in range(i+1, ilast + 1):
                    M[j] = min(M[j-1] + 1, maxk)
                    self.c[j] = M[j]
                return True
        return False

    def NextPartition(self):
        """
        Returns True if the next partition is generated,
        otherwise False. The permutation vector self.c is
        modified by this routine.

        feb/03/2004 - second version by ernie for clarity
        """
        if self.whattype != 1:
          return False

        M = self.Auxiliary()             # restricted grouwth function array
        ifirst =  self.baseindex         # starting index
        ilast  = len(self.c) - 1         # last index value
        mink   = ifirst
        maxk   = M[ilast]

        for i in range (ilast, ifirst, -1):
            if self.c[i] < maxk:
                if M[i] == maxk:         # speed optimization
                    self.c[i] += 1
                    for j in range(i+1, ilast+1):
                        self.c[j] = mink
                    return True

                elif M[i-1]  >= self.c[i]:
                    self.c[i] += 1
                    m = max(M[i-1], self.c[i])

                    for j in range(i+1, ilast+1):
                        self.c[j] = mink

                    # Adjust trailing entries
                    j = ilast
                    while m < maxk:
                        self.c[j] = maxk
                        maxk = maxk-1
                        j = j - 1
                    return True
```

```
    return False

def LastPartition(self, nclasses):
    """
    Returns the last partition containing k classes.
    """

    if self.whattype != 1:
      return False

    n = len(self.c)
    k = nclasses - (1 - self.baseindex)
    k = max(self.baseindex, k)
    k = min(n-1, k)
    print 'K = ', k

    for i in range(self.baseindex, n):
        self.c[i] = min(k, i)

def Auxiliary(self):
    """
    Returns an auxiliary array M (Restricted growth function)
    """
    n = len(self.c)
    M = [self.baseindex] * n
    for i in range(self.baseindex, n):
        if self.c[i] > M[i-1]:
            M[i] = M[i-1] + 1
        else:
            M[i] = M[i-1]
    return M
```