max-card-matching-online

Trees

```
class tree:
    # tree represents a rooted tree
    def __init__(self, branches):
        # construct a rooted tree by its branches
        m1=0
        self.m0=1
        self.order=1

        for branch in branches:
            m1=m1*branch.m+self.m0*branch.m0
            self.m0*=branch.m
            self.order+=branch.order

        branch_types=union(map(lambda x:x.tree_type,branches))
        if branch_types in ([], ["B"]):
            self.tree_type="A"
            self.m=self.m0+m1
        elif branch_types==["A"]:
            self.tree_type="B"
            self.m=m1
        else:
            raise NotImplementedError("branches have types %s" %
branch_types)


    def __repr__(self):
        return("Tree of order %s with (m,m0)=(%s,%s)" %
(self.order, self.m, self.m0))

    def rho(self):
        return(self.m0/self.m)

def abtree(branches):
    # shorthand notation
    return(tree([tree(branches)]))

L=tree([])
F=abtree([L,L])
# L and F as in the paper


def C(length, T):
```

```
    assert length in ZZ
    assert length>= 0
    if length>0:
        return(abtree([L,F,C(length-1,T)]))
    return(T)
```

Replacement Rules

Important Rooted Trees (Figure 8)

```
A_3=abtree([L])
A_6=abtree([L,L,L,L])
A_7=abtree([L,F])
A_10=abtree([L,A_7])
A_14=abtree([F,F,F])
A_24=abtree([F,F,A_14])
```

```
assert (L.order,L.rho())==(1,1)
assert (A_3.order,A_3.rho())==(3,1/2)
assert (F.order,F.rho())==(4,2/3)
assert (A_6.order,A_6.rho())==(6,4/5)
assert (A_7.order,A_7.rho())==(7,5/8)
assert (A_10.order,A_10.rho())==(10,13/21)
assert (A_14.order,A_14.rho())==(14,2/3)
assert (A_24.order,A_24.rho())==(24,2/3)
```

Replacements for some alpha (Table 3)

```
def check_replacement_alpha(T1,T2, alphamin, alphamax, leftopen):
    # assert that T1.m+T1.m0*alpha<T2.m+T2.m0*alpha for
alphamin<alpha<alphamax (if leftopen==True)
    # or alphamin<=alpha<alphamax (if leftopen==False)
    assert T1.order==T2.order
    assert T1.tree_type==T2.tree_type
    if (alphamin,alphamax,leftopen)==(0,+Infinity,False):
        assert T1.m<T2.m and T1.m0<=T2.m0
    elif (alphamin,leftopen)==(0,False):
        assert T2.m0<T1.m0 and
T1.m+alphamax*T1.m0==T2.m+alphamax*T2.m0
    elif (alphamax,leftopen)==(+Infinity,True):
        assert T2.m0>T1.m0 and
T1.m+alphamin*T1.m0==T2.m+alphamin*T2.m0
    else:
```

```
        raise NotImplementedError("alphamin=%s, alphamax=%s,
leftopen=%s" %
            (alphamin, alphamax,leftopen))
```

```
check_replacement_alpha(tree([A_3]),
        tree([L, L, L]),
        0,2,False)
check_replacement_alpha(tree([L, A_3]),
        tree([L, L, L, L]),
        0,1,False)
check_replacement_alpha(tree([A_3, A_3]),
        tree([L, abtree([L, L, L])]),
        0,+Infinity,False)
check_replacement_alpha(tree([A_3, A_3, A_3]),
        tree([L, F, F]),
        0,+Infinity,False)
check_replacement_alpha(abtree([F, F, A_24]),
        abtree([L, C(1,L), abtree([L, C(1,L), abtree([L, L,
C(1,L)])])]),
        50/2473,+Infinity,True)
check_replacement_alpha(abtree([F, A_14, A_14]),
        abtree([L, C(1,L), abtree([L, C(1,L), abtree([L, L,
C(1,L)])])]),
        50/2473,+Infinity,True)
check_replacement_alpha(abtree([F, A_14, A_24]),
        C(1,abtree([L, C(1,L), abtree([L, C(1,L), C(2,L)])])),
        0,+Infinity,False)
check_replacement_alpha(abtree([F, A_24, A_24]),
        C(2,abtree([L, C(2,L), C(3,L)])),
        0,+Infinity,False)
check_replacement_alpha(abtree([A_14, A_14, A_14]),
        C(1,abtree([L, C(1,L), abtree([L, C(1,L), C(2,L)])])),
        0,+Infinity,False)
check_replacement_alpha(abtree([A_14, A_14, A_24]),
        C(2,abtree([L, C(2,L), C(3,L)])),
        0,+Infinity,False)
check_replacement_alpha(abtree([A_14, A_24, A_24]),
        C(9,L),
        0,+Infinity,False)
check_replacement_alpha(abtree([A_24, A_24, A_24]),
        C(10,F),
        0,+Infinity,False)
check_replacement_alpha(tree([L, A_10]),
        tree([L, L, abtree([L, L, abtree([L, L, L])])]),
        0,7/6,False)
check_replacement_alpha(tree([L, A_14]),
        tree([L, L, abtree([L, L, abtree([L, L, abtree([L, L,
L])])])]),
        0,18/25,False)
```

```
check_replacement_alpha(tree([L, A_24]),
        tree([L, L, abtree([L, L, abtree([L, C(1,L), C(1,L)])])]),
        0,295/318,False)
check_replacement_alpha(tree([A_10, A_7, A_7]),
        tree([L, abtree([L, L, abtree([L, C(1,L), C(1,L)])])]),
        0,+Infinity,False)
check_replacement_alpha(tree([A_10, A_10, A_7]),
        tree([F, C(1,L), C(2,L)]),
        0,+Infinity,False)
check_replacement_alpha(tree([A_10, A_10, A_10]),
        tree([F, C(1,L), C(2,F)]),
        0,+Infinity,False)
check_replacement_alpha(tree([A_7, F, F]),
        tree([L, L, abtree([L, L, abtree([L, L, abtree([L, L,
L])])])]),
        0,3/4,False)
check_replacement_alpha(tree([A_7, A_7, F]),
        tree([L, L, abtree([L, L, abtree([L, L, C(1,L)])])]),
        0,50/39,False)
check_replacement_alpha(tree([A_7, A_7, A_7]),
        tree([L, abtree([L, L, abtree([L, L, abtree([L, L,
C(1,L)])])])]),
        0,+Infinity,False)
check_replacement_alpha(tree([L, L, L, abtree([L, L, L])]),
        tree([L, A_7]),
        1/2,+Infinity,True)
check_replacement_alpha(tree([L, L, abtree([L, L, abtree([L, L,
abtree([L, L, abtree([L, L, L])])])])]),
        tree([L, F, A_14]),
        2/17,+Infinity,True)
check_replacement_alpha(tree([L, F, C(1,A_14)]),
        tree([L, F, abtree([F, F, C(1,F)])]),
        1/3,+Infinity,True)
check_replacement_alpha(tree([L, F, C(1,A_24)]),
        tree([L, F, abtree([F, F, abtree([F, F, C(1,F)])])]),
        1/3,+Infinity,True)
```

Replacements for alpha=0 (Table 2)

```
def check_replacement(T1,T2):
    assert T1.order==T2.order
    assert T1.m<T2.m, "T1=%s, T2=%s" % (T1,T2)
```

```
check_replacement(tree([L, L, L, L, abtree([L, L, L])]),
        tree([F, L, F]))
check_replacement(tree([L, L, L, L, A_6]),
        C(1,F))
```

```
check_replacement(tree([A_7, F, F, L]),
        tree([F, F, F, F]))
check_replacement(abtree([L, L, abtree([L, L, abtree([L, L,
abtree([L, L, L])])])]),
        tree([F, F, F, F]))
```

Setup for chains (cf. Lemma 6.2)

```
var('k')

A=matrix([[8,3],[5,3]])
NF=QQ.extension(A.charpoly(),'alpha')
alpha=NF.gen()
# alpha here corresponds to lambda in the paper, as lambda is a
# reserved word in python
alphabar=9/alpha
# alphabar is the conjugate of alpha
assert A.charpoly()(alpha)==0
assert A.charpoly()(alphabar)==0
assert alpha.complex_embedding(i=1)>10
assert alphabar.complex_embedding(i=1)<1



MNF=MatrixSpace(NF,2,2)
D,T=MNF(A).eigenmatrix_right()
assert A==T*D*T.inverse()
assert D==Matrix([[alpha,0],[0,alphabar]])

NFP.<alphabar_k,alpha_k>=PolynomialRing(NF,2)
# alpha_k=alpha^k
# alphabar_k=alphabar^k
NFRF=NFP.fraction_field()

def val(expression,kk):
    # substitute k=kk in the expression, in particular substitute
alpha_k and alphabar_k
    return(expression.subs({alpha_k:alpha^kk,alphabar_k:
(9/alpha)^kk}))

D_k=Matrix([[alpha_k,0],[0,alphabar_k]])
# D_k = D^k
assert val(D_k,1)==D

def coefficients_k(exponent):
    # determine pair (c0,c1) of integers such that
exponent=c0+c1*k
```

```
    if exponent in ZZ:
        return((exponent,0))
    c1=exponent.coefficient(k,1)
    c0=exponent.coefficient(k,0)
    assert exponent==c1*k+c0, "%s is not a affine linear function
of k" % exponent
    assert c1 in ZZ, "Coefficient of k in %s is not an integer" %
exponent
    assert c0 in ZZ, "Constant term in %s is not an integer" %
exponent
    return(c0,c1)

def alphapower(exponent):
    # compute alpha^exponent where exponent is an affine linear
function in k
    (c0,c1)=coefficients_k(exponent)
    return( alpha_k^c1*alpha^c0 )

def Apower(exponent):
    # compute A^exponent where exponent is an affine linear
function in k
    (c0,c1)=coefficients_k(exponent)
    return( T*D_k^c1*T.inverse()*A^c0 )

for a in range(-1,1):
    for b in range(-1,1):
        for kk in range(3):
            assert val(Apower(a*k+b),kk)==A^(a*kk+b)
```

```
class chain(tree):
    # chain represents a rooted tree which arises by a chain
operation
    def __init__(self, length, T):
        # construct C^length T for some rooted tree T
        self.m,self.m0= Apower(length)*vector([T.m,T.m0])
        self.order=T.order+7*length
        assert T.tree_type=="A"
        self.tree_type="A"


def CL(length):
    # C^length L
    return(chain(length,L))

def CF(length):
    # C^length F
    return(chain(length,F))
```

Lemma 6.7

Preparations

```
def is_positive(nfelement):
    # returns nfelement>0  for a nfelement in NF
    if nfelement in QQ:
        return(QQ(nfelement)>0)
    assert nfelement.parent()==NF
    [c0,c1]=nfelement.list()
    compare=-c0/c1
    if c1>0:
        return(not(compare>10 and A.characteristic_polynomial()
(compare)>0))
    else:
        return(compare>10 and A.characteristic_polynomial()
(compare)>0)

def is_negative(nfelement):
    # returns nfelement<0 for a nfelement in NF
    return(is_positive(-nfelement))

def absNF(nfelement):
    # returns abs(nfelement)
    if is_negative(nfelement):
        return(-nfelement)
    else:
        return(nfelement)

def minNF(l):
    # returns min(l) for a list l of elements of NF
    if len(l)==1:
        return(l[0])
    else:
        a=l[0]
        b=minNF(l[1:])
        if is_positive(a-b):
            return(b)
        else:
            return(a)

def sign_of_k_polynomial(what):
    # returns (sign,kmin) if sign(what)==sign for k>=kmin; here
what in NFP
    assert what.parent()==NFP
    assert what.is_homogeneous()

    degree=what.degree()
```

```
    degree_alpha_k=what.degree(alpha_k)
    leading_coefficient=what.monomial_coefficient(
        alpha_k^degree_alpha_k*alphabar_k^(degree-degree_alpha_k))
    kk=0
    while True:
        extra=add([absNF(

what.monomial_coefficient(alpha_k^i*alphabar_k^(degree-i)))*
            (alpha^(kk*(i-degree_alpha_k))*alphabar^(kk*
(degree_alpha_k-i)))
            for i in range(degree_alpha_k)])
        if is_positive(absNF(leading_coefficient)-extra):
             break
        kk+=1
    if is_positive(leading_coefficient):
        sign=1
    elif is_negative(leading_coefficient):
        sign=-1
    return(sign,kk)

def sign_of_fraction(what):
    # returns (sign,kmin) if sign(what)==sign for k>=kmin;
    # here what in Quotient Field of NFP
    signs=[sign_of_k_polynomial(what.numerator()),
        sign_of_k_polynomial(what.denominator())]
    return(signs[0][0]*signs[1][0],max(signs[0][1],signs[1][1]))

def k_limit(rational_function):
    # limit of a rational function in alpha_k, alphabar_k, alpha
for k to infinity
    num=rational_function.numerator()
    den=rational_function.denominator()
    assert num.parent()==NFP
    assert den.parent()==NFP
    assert num.is_homogeneous()
    assert den.is_homogeneous()
    assert num.degree(alpha_k)==den.degree(alpha_k)

    d=num.degree(alpha_k)

return(num.monomial_coefficient(alpha_k^d)/den.monomial_coefficient(a

def find_minimum(sequence):
    # For a sequence in NFRF, find its minimum over all k>=0.
    # The sequence has to be monotonic for almost all k and
convergent.
    if sequence in NF:
        return(sequence)

    assert sequence in NFRF
```

```
    lim=k_limit(sequence)
    sign_of_difference,k_stable_sign=sign_of_fraction(

sequence.subs({alpha_k:alpha*alpha_k,alphabar_k:alphabar*alphabar_k})
    candidates=[val(sequence,kk) for kk in range(k_stable_sign+1)]
    candidates.append(lim)
    return(minNF(candidates))

rhomax=ceil((sqrt(3)-1)*10000)/10000
assert rhomax>sqrt(3)-1

def replace_subtree(old,new):
    # Let old and new be rooted trees, possibly depending on k.
    # Returns a pair (q,new.order-old.order) such that
    # m(Tnew)/m(Told)>= q for all trees Told fulfilling the LC
    # containing old and the tree Tnew arises from Told
    # by replacing one occurrence of old in Told by Tnew
    rhomin=2/3

min_rhomin=find_minimum((new.m+new.m0*rhomin)/(old.m+old.m0*rhomin))

min_rhomax=find_minimum((new.m+new.m0*rhomax)/(old.m+old.m0*rhomax))
    return((minNF([min_rhomin, min_rhomax]),new.order-old.order))

def minNFCount(list):
    # Given a list of pairs (q_i, d_i) (for q_i in NF), determine
(min(q_i), union(d_i))
    Qmin=minNF(map(lambda x:x[0],list))
    if Qmin in QQ:
        Qmin_rounded=floor(QQ(Qmin)*10000)/10000
    else:

Qmin_rounded=floor(Qmin.complex_embedding(i=1)*10000)/10000
    assert is_positive(Qmin-Qmin_rounded)
    return(Qmin, Qmin_rounded.n(),union(map(lambda x:x[1],list)))



def combine_replacements(list):
    return(is_positive(prod(map(lambda x:x[0],list))-1),
        prod(map(lambda x:x[1],list)), add(map(lambda x:x[2]
[0],list)))
```

Verification of the items

```
r1=minNFCount([replace_subtree(
    tree([CF(2*k+floor((s)/3)),CF(2*k+floor((s+1)/3)),
```

```
CF(2*k+floor((s+2)/3))]),
    tree([CL(3*k+floor((s+2)/2)),CL(3*k+floor((s+3) /2)),L]))
    for s in range(6)])
r1
```

(-378197654099/48936229893*alpha + 453166604188/5437358877, 5.21100000000000, [5])

```
r2=minNFCount([ replace_subtree(
    tree([CF(2*k+floor((s)/3)),CF(2*k+floor((s+1)/3)),
CF(2*k+floor((s+2)/3))]),
    tree([CL(3*k+floor((s+1)/2)),CL(3*k+floor((s+2)/2)),L]))
    for s in range(6)])
r2
```

(-453166604188/48936229893*alpha + 511848332441/5437358877, 0.515400000000000, [-2])

```
r3=minNFCount([replace_subtree(
    tree([CF(k+floor(s/3)),CF(k+floor((s+1)/3)),CF(k+floor((s+2)
/3))]),
    tree([CF(3*k+s),F,L]))
    for s in range(3)])
r3
```

(19473509095/16312076631*alpha - 21199304045/1812452959, 0.372600000000000, [-3])

```
r4=minNFCount([replace_subtree(
    tree([CF(k+floor((jj+1)/3)),CF(k+floor((jj+2)/3)),
        chain(k,
        abtree([CF(k+floor((ii+1)/4)),CF(k+floor((ii+2)
/4)),CF(k+floor((ii+3)/4))]))]),
    tree([L,F,chain(2*k+ii+jj,abtree([CL(2*k+1),CL(2*k+1),L]))]))
    for ii in range(4) for jj in range(3)]);
r4
```

(11194822060325/92721277692*alpha - 3138801763960/2575591047, 1.94380000000000, [2])

```
r5=minNFCount([replace_subtree(
    tree([CL(k+1),CL(k+j),L]),
    tree([L,F,CF(2*k+j)]))
    for j in [0,1]])
r5
```

(5/81*alpha + 8/81, 0.722800000000000, [-1])

```
r6=minNFCount([replace_subtree(
    tree([CL(3*k+floor(s/2)),CL(3*k+floor((s+1)/2)),L]),
    tree([CF(2*k+floor(s/3)),CF(2*k+floor((s+1)
/3)),CF(2*k+floor((s-1)/3))]))
    for s in range(1,7)])
r6
```

(27/14, 1.92850000000000, [2])

```
r6a=minNFCount([replace_subtree(
    tree([CL(3*k+floor(s/2)),CL(3*k+floor((s+1)/2)),L]),
    tree([CF(2*k+floor(s/3)),CF(2*k+floor((s+1)
/3)),CF(2*k+floor((s-1)/3))]))
    for s in range(2,8)])
r6a
```

```
(2187/1133, 1.93020000000000, [2])
```

```
r7=minNFCount([replace_subtree(
    tree([CL(k+2+t),L,chain(k+1,
        abtree([CL(k+2+floor((s+1)/3)),CL(k+2+floor((s+2)
/3)),L]))]),
    tree([F,L,chain(k+floor((s+t+2)/4),
        abtree([CF(k+1+floor((s+t+3)/4)),CF(k+1+floor((s+t+4)/4)),
            CF(k+1+floor((s+t+5)/4))]))]))
    for s in range(3) for t in range(2)])
r7
```

```
(57962385/111863416, 0.518100000000000, [-2])
```

```
r8=minNFCount([replace_subtree(
    tree([CL(t),L,abtree([CL(floor(s/2)),CL(floor((s+1)/2)),L])]),
    tree([CF(floor((s+t-1)/3)),CF(floor((s+t)/3)),CF(floor((s+t+1)
/3))]))
    for s in range(1,5) for t in range (0,3)])
r8
```

```
(2187/4235, 0.516400000000000, [-2])
```

```
r9=minNFCount([replace_subtree(
   tree([CL(t),L,abtree([CL(floor(s/2)),CL(floor((s+1)/2)),L])]),
   tree([L,F,CF(s+t)]))
   for s in range(1,5) for t in range(3)]);
r9
```

```
(86496057/44259488, 1.95420000000000, [2])
```

Applications of Lemma 6.7

Lemma 6.10

```
combine_replacements([r1,r2,r3])
```

```
(True, 1.00071022644000, 0)
```

Lemma 6.11

```
combine_replacements([r2,r4])
```

```
(True, 1.00183452000000, 0)
```

Lemma 6.14

```
combine_replacements([r5,r5,r6])
```

```
    (True, 1.00752523144000, 0)
```

Lemma 6.16

```
combine_replacements([r6a,r7])
```

```
    (True, 1.00003662000000, 0)
```

```
combine_replacements([r9,r7])
```

```
    (True, 1.01247102000000, 0)
```

```
combine_replacements([r8,r9])
```

```
    (True, 1.00914888000000, 0)
```

Optimal Trees

```
optimal_tree_symbolic=[None for kk in range(7)]
# optimal_tree_symbolic[case] represents the "generic" case of
T_n^*
# for order n>=37 with case = n mod 7
# optimal_tree_symbolic[case] is a list containing various
subcases,
# as we resolve expressions of the shape floor( (r*k+j)/r )
# explicitly before storing the result.

optimal_tree_symbolic[1]=[CL(k)]

optimal_tree_symbolic[2]=
[tree([CL(0),CL(k+1+floor((j+1)/5)),CL(k+1+floor((j+3)/5)),
    chain(k+floor(j/5),
    abtree([CL(0),CL(k+1+floor((j+2)/5)),CL(k+1+floor((j+4)
/5))]))]) for j in range(5)]

optimal_tree_symbolic[3]=
[tree([CF(k+floor((j+0)/4)),CF(k+floor((j+1)/4)),
    CF(k+floor((j+2)/4)),CF(k+floor((j+3)/4))])
    for j in range(4)]

optimal_tree_symbolic[4]=[CF(k)]

optimal_tree_symbolic[5]=
[tree([CL(0),CL(k+floor((j+0)/3)),CL(k+floor((j+1)/3)),
    CL(k+floor((j+2)/3))]) for j in range(3)]

optimal_tree_symbolic[6]=
[tree([CF(k+floor((j+1)/7)),CF(k+floor((j+3)/7)),
    CF(k+floor((j+5)/7)),
```

```
    chain(k+floor(j/7),
    abtree([CF(k+floor((j+2)/7)),CF(k+floor((j+4)
/7)),CF(k+floor((j+6)/7))]))])
    for j in range(7)]

optimal_tree_symbolic[0]=[tree([CL(0),CL(0),CF(k)])]
```

```
def optimal_m(n):
    # determine m(T_n^*) for n>=37 for a specific n
    case=n % 7
    # compute relevant subcase and corresponding value of k
    offset=optimal_tree_symbolic[case][0].order.subs({k:0})
    num_subcases=len(optimal_tree_symbolic[case])
    subcase=int(((n-offset)/7)) % num_subcases
    kk = (n-offset-7*subcase)/(7*num_subcases)
    assert kk>=0, "T_%d^* is not a generic case " % n
    assert n==optimal_tree_symbolic[case]
[subcase].order.subs({k:kk})

    return(ZZ(val(optimal_tree_symbolic[case][subcase].m,kk)))
```

Lemma 6.16, case k_0=k_5=0, s=1

```
open_cases_6_16=[tree([CL(0),CL(floor(s/2)),CL(floor((s+1)/2)),
    abtree([CL(0),CL(t),
    abtree([CL(0),CL(floor(u/2)),CL(floor((u+1)/2))])])])
    for t in range(3) for s in range(1,5) for u in range(1,5)]
for t in open_cases_6_16:
    assert optimal_m(t.order)>t.m
```

Asymptotics of m(T_n^*)

```
def asymptotic_coefficient(case):
    # returns c_case such that m(T_n^*) ~ c_case lamba^floor(n/7)
for all n
    # with case = n mod 7.
    result=union([ k_limit(T.m/alphapower((T.order()-case)/7))
        for T in optimal_tree_symbolic[case] ])
    assert len(result)==1
    return(result[0])
```

```
def numeric_asymptotic_coefficient(case):
    # returns c_case_numeric such that m(T_n^*) ~ c_case
```

```
lamba^(n/7)
    # for all n with case = n mod 7.
    return(asymptotic_coefficient(case).complex_embedding(i=1)/
        (alpha.complex_embedding(i=1))^(case/7))
```

```
numeric_asymptotic_coefficients=map(numeric_asymptotic_coefficient,
range(7))
def optimal_m_approx(n):
    return(numeric_asymptotic_coefficients[ n %
7]*alpha.complex_embedding(i=1)^(n/7))
```

```
approx_ratios=[ optimal_m_approx(nn)/optimal_m(nn) for nn in
range(1000,1100)]
(min(approx_ratios),max(approx_ratios))
```

```
(0.999999999999957, 1.00000000000007)
```

```
[asymptotic_coefficient(jj) for jj in range(7)]
```

```
[67/765*alpha - 71/765, 11/85*alpha - 18/85, 101047/614125*alpha
90171/614125, 4996/21675*alpha - 4448/21675, 27/85*alpha - 21/85
3209/7225*alpha - 2817/7225, 6451616/10440125*alpha -
5743408/10440125]
```

```
[numeric_asymptotic_coefficient(jj) for jj in range(7)]
```

```
[0.792620574273610, 0.787947762616490, 0.783080426542439,
0.788434032505851, 0.790280714748050, 0.785510324593434,
0.784269603628599]
```

```
```