Marked Graph Compression

Generated by Doxygen 1.8.13

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 b_graph Class Reference

simple unmarked bipartite graph

`#include <bipartite_graph.h>`

Public Member Functions

- b_graph (vector< vector< int > > list, vector< int > right_deg)
  a constructor
- b_graph (vector< vector< int > > list)
  a constructor
- vector< int > get_adj_list (int v) const
  returns the adjacency list of a given left vertex
- int get_right_degree (int v) const
  returns the degree of a right vertex v
- int get_left_degree (int v) const
  returns the degree of a right vertex v
- vector< int > get_right_degree_sequence () const
  return the right degree sequence
- vector< int > get_left_degree_sequence () const
  return the left degree sequence
- int nu_left_vertices () const
  returns the number of left vertices
- int nu_right_vertices () const
  returns the number of right vertices

Private Attributes

- int n
  the number of left vertices
- int np
  the number of right vertices
- vector< vector< int > > adj_list
  adjacency list for left vertices, where for $0 \leq v < n$, adj_list[v] is a sorted list of right vertices connected to v.
- vector< int > left_deg_seq
  degree sequence for left vertices, where left_deg_seq[v] is the degree of the left node v
- vector< int > right_deg_seq
  degree sequence for right vertices, where left_deg_seq[v] is the degree of the right node v

Friends

- ostream & operator<< (ostream &o, const b_graph &G)

  printing the graph to the output
- bool operator== (const b_graph &G1, const b_graph &G2)

  comparing two graphs for equality
- bool operator!= (const b_graph &G1, const b_graph &G2)

  comparing for inequality

3.1.1 Detailed Description

simple unmarked bipartite graph

A simple unmarked bipartite graph with n left nodes and np right nodes. There are two ways to define such an object.

1. through adjacency list which is a vector<vector<int> > of size n (number of left nodes) where each element is a vector of adjacent right vertices (does not have to be sorted). Note that both left and right vertex indices are 0 based. For instance, (in c++11 notation), if list = {{0},{1},{0,1}}, the graph has 3 left nodes and 2 right nodes, left node 0 is connected to right node 0, left node 1 is connected to right node 1, and left node 2 is connected to right nodes 0 and 1.

```
vector<vector<int> > list = {{0},{1},{0,1}};
b_graph G(list);
```

2. through adjacency list and right degree vector. Adjacency list is as explained above, and the extra information of right degree vector is just to help construct the object more easily. For instance, with list = {{0},{1},{0,1}}, we have right_deg = {1,2}, which means that the degree of the right node 0 is 1 while the degree of the right node 1 is 2.

```
vector<vector<int> > list = {{0},{1},{0,1}};
vector<int> right_deg = {1,2}
b_graph G(list, right_deg);
```

3.1.2 Constructor & Destructor Documentation

3.1.2.1 b_graph() [1/2]

```
b_graph::b_graph (
            vector< vector< int > > list,
            vector< int > right_deg )
```

a constructor

This constructor takes the list of adjacent vertices and the right degree sequence, and constructs an object.

Parameters

list	list[v] for a left node v is the list of right nodes w connected to v. This list does not have to be sorted
right_deg	right_deg[v] is the degree of the right node v

```
{
  n = list.size();
  np = right_deg.size(); // the number of right nodes
  adj_list = list;
  left_deg_seq.resize(n);
  // sorting the list
   for (int v=0; v<n; v++){
     sort(adj_list[v].begin(), adj_list[v].end());
     left_deg_seq[v] = adj_list[v].size();
   }
   right_deg_seq = right_deg;
}
```

3.1.2.2 b_graph() [2/2]

```
b_graph::b_graph (
              vector< vector< int > > list )
```

a constructor

This constructor takes the list of adjacent vertices

Parameters

list	list[v] for a left node v is the list of right nodes w connected to v. This list does not have to be sorted

```
{
  // goal: finding right degrees and calling the above constructor
  // first, we find the number of right nodes
  np = 0; // the number of right nodes
  n = list.size();
  adj_list = list;
  left_deg_seq.resize(n);
  for (int v=0;v<adj_list.size();v++){
    sort(adj_list[v].begin(), adj_list[v].end());
    if (adj_list[v][adj_list[v].size()-1] > np)
      np = list[v][adj_list[v].size()-1];
    left_deg_seq[v] = adj_list[v].size();
  }
  np++; // node indexing is zero based

  right_deg_seq.resize(np);
  fill(right_deg_seq.begin(), right_deg_seq.end(), 0); // make all elements 0
  for (int v=0;v<list.size();v++)
    for (int i=0;i<list[v].size();i++)
      right_deg_seq[list[v][i]]++;
}
```

3.1.3 Member Function Documentation

3.1.3.1 get_adj_list()

```
vector< int > b_graph::get_adj_list (
              int v ) const
```

returns the adjacency list of a given left vertex

```
{
  if (v < 0 or v >= n)
    cerr << "b_graph::get_adj_list, index v out of range" << endl;
  return adj_list[v];
}
```

3.1.3.2 get_left_degree()

```
int b_graph::get_left_degree (
            int v ) const
```

returns the degree of a right vertex v

```
{
  if (v < 0 or v >= n)
    cerr << "b_graph::get_left_degree, index v out of range" << endl;
  return left_deg_seq[v];
}
```

3.1.3.3 get_left_degree_sequence()

```
vector< int > b_graph::get_left_degree_sequence ( ) const
```

return the left degree sequence

```
{
  return left_deg_seq;
}
```

3.1.3.4 get_right_degree()

```
int b_graph::get_right_degree (
            int v ) const
```

returns the degree of a right vertex v

```
{
  if (v < 0 or v >= n)
    cerr << "b_graph::get_right_degree, index v out of range" << endl;
  return right_deg_seq[v];
}
```

3.1.3.5 get_right_degree_sequence()

```
vector< int > b_graph::get_right_degree_sequence ( ) const
```

return the right degree sequence

```
{
  return right_deg_seq;
}
```

3.1.3.6 nu_left_vertices()

```
int b_graph::nu_left_vertices ( ) const
```

returns the number of left vertices

```
{
  return n;
}
```

3.1.3.7 nu_right_vertices()

```
int b_graph::nu_right_vertices ( ) const
```

returns the number of right vertices

```
{
  return np;
}
```

3.1.4 Friends And Related Function Documentation

3.1.4.1 operator"!=

```
bool operator!= (
             const b_graph & G1,
             const b_graph & G2 )   [friend]
```

comparing for inequality

```
{
  return !(G1 == G2);
}
```

3.1.4.2 operator<<

```
ostream& operator<< (
             ostream & o,
             const b_graph & G )  [friend]
```

printing the graph to the output

```
{
  int n = G.nu_left_vertices();
  vector<int> list;
  for (int i=0;i<n;i++){
    list = G.get_adj_list(i);
    o << i << " -> ";
    for (int j=0;j<list.size();j++){
      o << list[j];
      if (j < list.size()-1)
        o << ", ";
    }
    o << endl;
  }
  return o;
}
```

3.1.4.3 operator==

```
bool operator== (
             const b_graph & G1,
             const b_graph & G2 )  [friend]
```

comparing two graphs for equality

```
{
  int n1 = G1.nu_left_vertices();
  int n2 = G2.nu_left_vertices();

  int np1 = G1.nu_right_vertices();
  int np2 = G2.nu_right_vertices();
  if (n1!= n2 or np1 != np2)
    return false;
  vector<int> list1, list2;
  for (int v=0; v<n1; v++){
    list1 = G1.get_adj_list(v);
    list2 = G2.get_adj_list(v);
    if (list1 != list2)
      return false;
  }
  return true;
}
```

3.1.5 Member Data Documentation

3.1.5.1 adj_list

```
vector<vector<int> > b_graph::adj_list  [private]
```

adjacency list for left vertices, where for $0 \leq v < n$, adj_list[v] is a sorted list of right vertices connected to v.

3.1.5.2 left_deg_seq

```
vector<int> b_graph::left_deg_seq [private]
```

degree sequence for left vertices, where left_deg_seq[v] is the degree of the left node v

3.1.5.3 n

```
int b_graph::n [private]
```

the number of left vertices

3.1.5.4 np

```
int b_graph::np [private]
```

the number of right vertices

3.1.5.5 right_deg_seq

```
vector<int> b_graph::right_deg_seq [private]
```

degree sequence for right vertices, where left_deg_seq[v] is the degree of the right node v

The documentation for this class was generated from the following files:

- bipartite_graph.h
- bipartite_graph.cpp

3.2 b_graph_decoder Class Reference

Decodes a simple unmarked bipartite graph.

```
#include <bipartite_graph_compression.h>
```

Public Member Functions

- b_graph_decoder (vector< int > a_, vector< int > b_)

  constructor
- void init ()

  initializes x as empty list of size n, beta as b, U with b and W with a
- pair< mpz_class, mpz_class > decode_node (int i, mpz_class tN)

  decodes the connectivity list of a left node $0 \leq i < n$ *given* $\tilde{N}_{i,i}$
- pair< mpz_class, mpz_class > decode_interval (int i, int j, mpz_class tN)

  decodes the connectivity list of left vertices $i \leq v \leq j$ *given* $\tilde{N}_{i,j}$
- b_graph decode (mpz_class f)

  decodes the bipartite graph given the encoded integer

Private Attributes

- int n

  number of left vertices
- int np

  number of right vertices
- vector< int > a

  left degree sequence
- vector< int > b

  right degree sequence
- vector< vector< int > > x

  the adjacency list of left nodes for the decoded graph
- reverse_fenwick_tree U

  reverse Fenwick tree initialized with the right degree sequence b, and after decoding vertex i*, for* $0 \leq v < n'$*, we have* $U_v = \sum_{k=v}^{n'-1} b_k(i)$
- reverse_fenwick_tree W

  keeping partial sums for the degree sequence a. More precisely, for $0 \leq v < n$*, we have* $W_v = \sum_{k=v}^{n-1} a_k$
- vector< int > beta

  the sequence $\vec{\beta}$*, where before decoding vertex* i*, for* $0 \leq v < n'$*, we have* $\beta_v = b_v(i)$

3.2.1 Detailed Description

Decodes a simple unmarked bipartite graph.

Decodes a simple bipartite graph given its encoded integer. We assume that the decoder knows the left and right degree sequences of the encoded graph, hence these sequences must be given when a decoder object is being constructed. For instance, borrowing the degree sequences of the example we used to explain the b_graph_encoder class:

```
vector<int> a = {1,1,2};
vector<int> b = {2,2};
b_graph_decoder D(a,b);
```

Then, if variable f of type mpz_class is obtained from a b_graph_encoder class, we can reconstruct the graph using f:

```
b_graph Ghat = D.decode(f);
```

Then, the graph Ghat will be equal to the graph G. Here is a full example showing the procedure of compression and decompression together:

```
vector<int> a = {1,1,2}; // left degree sequence
vector<int> b = {2,2}; // right degree sequence

b_graph G({{0},{1},{0,1}}); // defining the graph

b_graph_encoder E(a,b); // constructing the encoder object
mpz_class f = E.encode(G);

b_graph_decoder D(a, b);
b_graph Ghat = D.decode(f);

if (Ghat == G)
  cout << " we successfully reconstructed the graph! " << endl;
```

3.2.2 Constructor & Destructor Documentation

3.2.2.1 b_graph_decoder()

```
b_graph_decoder::b_graph_decoder (
            vector< int > a_,
            vector< int > b_ )
```

constructor

```
{
  a = a_;
  b = b_;
  n = a.size();
  np = b.size();
  init();
}
```

3.2.3 Member Function Documentation

3.2.3.1 decode()

```
b_graph b_graph_decoder::decode (
            mpz_class f )
```

decodes the bipartite graph given the encoded integer

Parameters

f	which is $\lceil N(G)/\prod b_v! \rceil$

Returns

the decoded bipartite graph G

```
{
  mpz_class prod_b_factorial = prod_factorial(b, 0, np-1);
  mpz_class tN = f * prod_b_factorial;
  decode_interval(0,n-1,tN);
  return b_graph(x, b);
}
```

3.2.3.2 decode_interval()

```
pair< mpz_class, mpz_class > b_graph_decoder::decode_interval (
            int i,
            int j,
            mpz_class tN )
```

decodes the connectivity list of left vertices $i \leq v \leq j$ given $\tilde{N}_{i,j}$

Parameters

i,j	endpoints of the interval
tN	$\tilde{N}_{i,j}$

Returns

decodes the connectivity list of vertices in the range and updated member x. Furthermore, returns a pair where the first component is $N_{i,j}(G)$ and the second is $l_{i,j}(G)$

```
{
  if (i==j)
    return decode_node(i,tN);
  int k = (i+j)/ 2; // midpoint to break
  int Wk = W.sum(k+1);
  int Wj = W.sum(j+1);
  mpz_class rkj = compute_product(Wk, Wk - Wj, 1) /
  prod_factorial(a, k+1, j); // r_{t+1, j}
  mpz_class tNik = tN / rkj; // \tilde{N}_{i,k}
  pair<mpz_class, mpz_class> ans; // to keep the return for each subinterval

  // calling the left subinterval
  ans = decode_interval(i,k,tNik);

  // preparing for the right subinterval
  mpz_class Nik = ans.first;
  mpz_class lik = ans.second;
  mpz_class tNkj = (tN - Nik * rkj) / lik; // \tilde{N}_{k+1, j}

  // calling the right subinterval
  ans = decode_interval(k+1, j, tNkj);
  mpz_class Nkj = ans.first;
  mpz_class lkj = ans.second;
  mpz_class Nij = Nik * rkj + lik * Nkj;
  mpz_class lij = lik * lkj;
  return pair<mpz_class, mpz_class> (Nij, lij);
}
```

3.2.3.3 decode_node()

```
pair< mpz_class, mpz_class > b_graph_decoder::decode_node (
            int i,
            mpz_class tN )
```

decodes the connectivity list of a left node $0 \leq i < n$ given $\tilde{N}_{i,i}$

Parameters

i	the vertex to be decoded
tN	$\tilde{N}_{i,i}$

Returns

decodes the connectivity list and updates the x member, and returns a pair, where the first component is $N_{i,i}(G)$ and the second component is $l_i(G)$

```
{
  mpz_class li = 1;
  mpz_class Ni = 0;
  int f, g; // endpoints of the interval for binary search
  int v;
  mpz_class y; // helper
  x[i].clear(); // make sure nothing is in the list to be decoded
  for (int k=0;k<a[i];k++){
    // finding x[i][k]
    if (k==0)
      f = 0;
    else
      f = 1 + x[i][k-1];
    g = np-1;
    while (g > f){
      v = (f+g)/2;
      if (binomial(U.sum(1+v) , a[i] - k) <= tN)
        g = v;
      else
        f = v + 1;
    }
    x[i].push_back(f); // decoded the kth connection of vertex i
    y = binomial(U.sum(1+x[i][k]), a[i] - k);
    tN = (tN - y) / beta[x[i][k]];
    Ni += li * y;
    li *= beta[x[i][k]];
    beta[x[i][k]] --;
    U.add(x[i][k], -1);
  }
  return pair<mpz_class, mpz_class>(Ni, li);
}
```

3.2.3.4 init()

```
void b_graph_decoder::init ( )
```

initializes x as empty list of size n, beta as b, U with b and W with a

```
{
  x.clear();
  x.resize(n);
  beta = b;
  U = reverse_fenwick_tree(b);
  W = reverse_fenwick_tree(a);

}
```

3.2.4 Member Data Documentation

3.2.4.1 a

```
vector<int> b_graph_decoder::a [private]
```

left degree sequence

3.2.4.2 b

```
vector<int> b_graph_decoder::b [private]
```

right degree sequence

3.2.4.3 beta

```
vector<int> b_graph_decoder::beta [private]
```

the sequence $\vec{\beta}$, where before decoding vertex i, for $0 \leq v < n'$, we have $\beta_v = b_v(i)$

3.2.4.4 n

```
int b_graph_decoder::n [private]
```

number of left vertices

3.2.4.5 np

```
int b_graph_decoder::np [private]
```

number of right vertices

3.2.4.6 U

`reverse_fenwick_tree b_graph_decoder::U [private]`

reverse Fenwick tree initialized with the right degree sequence b, and after decoding vertex i, for $0 \leq v < n'$, we have $U_v = \sum_{k=v}^{n'-1} b_k(i)$

3.2.4.7 W

`reverse_fenwick_tree b_graph_decoder::W [private]`

keeping partial sums for the degree sequence a. More precisely, for $0 \leq v < n$, we have $W_v = \sum_{k=v}^{n-1} a_k$

3.2.4.8 x

`vector<vector<int> > b_graph_decoder::x [private]`

the adjacency list of left nodes for the decoded graph

The documentation for this class was generated from the following files:

- bipartite_graph_compression.h
- bipartite_graph_compression.cpp

3.3 b_graph_encoder Class Reference

Encodes a simple unmarked bipartite graph.

`#include <bipartite_graph_compression.h>`

Public Member Functions

- b_graph_encoder (vector$<$ int $>$ a_, vector$<$ int $>$ b_)

  constructor
- void init (const b_graph &G)

  initializes beta and U
- pair$<$ mpz_class, mpz_class $>$ compute_N (int i, int j, const b_graph &G)

  computes $N_{i,j}(G)$
- mpz_class encode (const b_graph &G)

  encodes the given bipartite graph G and returns an integer in the specified range

Private Attributes

- vector< int > beta

  when compute_N is called for $i \leq j$*, for* $i \leq v \leq n$*, we have beta[v] =* $b_v(i)$
- vector< int > a

  the degree sequence for the left nodes
- vector< int > b

  the degree sequence for the right nodes
- reverse_fenwick_tree U

  a Fenwick tree which encodes the degree of right nodes. When compute_N is called for $i \leq j$*, for* $i \leq v \leq n$*, we have U.sum[v]* $= \sum_{k=v}^{n} b_k(i)$.

3.3.1 Detailed Description

Encodes a simple unmarked bipartite graph.

Encodes a simple bipartite graph in the set of bipartite graphs with given left degree sequence a and right degree sequence b. Therefore, to construct an encoder object, we need to specify these two degree sequences as vectors of int. For instance (in c++11)

```
vector<int> a = {1,1,2};
vector<int> b = {2,2};
b_graph_encoder E(a,b);
```

constructs an encode object E which is capable of encoding bipartite graphs having 3 left nodes with degrees 1, 1, 2 (in order) and 2 right nodes with degrees 2,2 (in order). Hence, assume that we have defined such a bipartite graph by giving adjacency list:

```
b_graph G({{0},{1},{0,1}});
```

Note that G has left and right degree sequences which are equal to a and b, respectively. Then, we can use E to encode G as follows:

```
mpz_class f = E.encode(G);
```

In this way, the encode converts G to an integer stored in f. Later on, we can use f to decode G.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 b_graph_encoder()

```
b_graph_encoder::b_graph_encoder (
            vector< int > a_,
            vector< int > b_ )  [inline]
```

constructor

```
40 : a(a_), b(b_) {}
```

3.3.3 Member Function Documentation

3.3.3.1 compute_N()

```
pair< mpz_class, mpz_class > b_graph_encoder::compute_N (
            int i,
            int j,
            const b_graph & G )
```

computes $N_{i,j}(G)$

Parameters

i,j	the interval for which we compute $N_{i,j}(G)$
G	reference to the bipartite graph for which we compute N

Returns

A pair, where the first component is $N_{i,j}(G)$, and the second component is $l_{i,j}(G)$

```
{
  mpz_class Nij = 0;
  mpz_class lij = 1;
  if (i == j){
    vector<int> x = G.get_adj_list(i); // the adjacency list of the vertex
    for (int k=0;k<a[i];k++){
      Nij += lij * binomial(U.sum(1+x[k]), a[i] - k);
      lij *= beta[x[k]];
      beta[x[k]] --;
      U.add(x[k],-1);
    }
    return pair<mpz_class, mpz_class> (Nij, lij);
  }else{
    int t = (i+j)/ 2;
    mpz_class Nit, lit; // for the left subinterval i, j
    mpz_class Ntj, ltj; // for the right subinterval t+1, j
    mpz_class Nij, lij; // to return
    int St; // S_{t+1}
    int Sj; // S_{j+1}

    pair<mpz_class, mpz_class> ans; // for collecting the results from subintervals

    // left subinterval
    ans = compute_N(i,t, G);
    Nit = ans.first;
    lit = ans.second;
    St = U.sum(0);

    // right subinterval
    ans = compute_N(t+1, j, G);
    Ntj = ans.first;
    ltj = ans.second;
    Sj = U.sum(0);

    mpz_class rtj; // r_{t+1, j}
    mpz_class prod_afac = prod_factorial(a, t+1, j);; // the product of a_k! for t + 1 <= k <= j


    rtj = compute_product(St, St - Sj, 1) / prod_afac;

    Nij = Nit * rtj + lit * Ntj;
    lij = lit * ltj;
    return pair<mpz_class, mpz_class>(Nij, lij);
  }
}
```

3.3.3.2 encode()

```
mpz_class b_graph_encoder::encode (
              const b_graph & G )
```

encodes the given bipartite graph G and returns an integer in the specified range

```
{
  if (a != G.get_left_degree_sequence() or b != G.get_right_degree_sequence())
    cerr << " WARNING b_graph_encoder::encoder : vectors a and/or b do not match with the degree sequences of the given bipartite graph  " << endl;
  init(G); // initialize U and beta for G
  pair<mpz_class, mpz_class> ans = compute_N(0,G.nu_left_vertices()-1, G);
  mpz_class prod_b_factorial = prod_factorial(b, 0, b.size()-1); // \prod_{i=0}^{n-1} b_i
  bool ceil = false;
  if (ans.first % prod_b_factorial != 0)
    ceil = true;
  ans.first /= prod_b_factorial;
  if (ceil)
    ans.first ++;
  return ans.first;
}
```

3.3.3.3 init()

```
void b_graph_encoder::init (
              const b_graph & G )
```

initializes beta and U

```
{
  // initializing beta
  beta = G.get_right_degree_sequence();
  // initializing the Fenwick tree
  U = reverse_fenwick_tree(beta);
  if (a != G.get_left_degree_sequence() or b != G.get_right_degree_sequence())
    cerr << " WARNING b_graph_encoder::init : vectors a and/or b do not match with the degree sequences of the given bipartite graph  " << endl;
}
```

3.3.4 Member Data Documentation

3.3.4.1 a

```
vector<int> b_graph_encoder::a  [private]
```

the degree sequence for the left nodes

3.3.4.2 b

```
vector<int> b_graph_encoder::b  [private]
```

the degree sequence for the right nodes

3.3.4.3 beta

```
vector<int> b_graph_encoder::beta  [private]
```

when compute_N is called for $i \leq j$, for $i \leq v \leq n$, we have beta[v] = $b_v(i)$

3.3.4.4 U

```
reverse_fenwick_tree b_graph_encoder::U  [private]
```

a Fenwick tree which encodes the degree of right nodes. When compute_N is called for $i \leq j$, for $i \leq v \leq n$, we have U.sum[v] $= \sum_{k=v}^{n} b_k(i)$.

The documentation for this class was generated from the following files:

- bipartite_graph_compression.h
- bipartite_graph_compression.cpp

3.4 colored_graph Class Reference

this class defines a colored graph, which is obtained from a simple marked graph and the color of edges come from the type of edges

```
#include <graph_message.h>
```

Public Member Functions

- colored_graph (const marked_graph &graph, int depth, int max_degree)

  constructor from a graph, depth and maximum degree parameters
- void init ()

  initializes other variables

Public Attributes

- const marked_graph & G

    the marked graph from which this is created
- int h

    the depth up to which look at edge types
- int Delta

    the maximum degree threshold
- graph_message M

    we use the message passing algorithm of class graph_message to find out edge types
- int nu_vertices

    the number of vertices in the graph.
- vector$<$ vector$<$ pair$<$ int, pair$<$ int, int $>$ $>$ $>$ $>$ adj_list

    adj_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, color component towards i, color component towards other endpoint). Therefore, the color of an edge between v and its ith neighbor is of the form (adj_list[v][i].second.first, adj_list[v][i].second.second)
- vector$<$ map$<$ int, int $>$ $>$ adj_location

    adj_location[v] for $0 \leq v < n$, is a map, where adj_location[v][w] denotes the index in adj_list[v] where the information regarding the edge between v and w is stored. Hence, adj_location[v][w] does not exist if w is not adjacent to v, and adj_list[v][adj_location[v][w]] is the edge between v and w
- vector$<$ vector$<$ int $>$ $>$ ver_type

    vertex mark and the colored degree matrix of each vertex. For a vertex v, D[v] is a vector of size $1 + L \times L$, where the first entry is the vertex mark, and the rest is the colored degree matrix row by row. Here, L denotes the number of colors.
- map$<$ vector$<$ int $>$, int $>$ ver_type_dict

    the dictionary mapping vertex types to integers, obtained from the ver_type array defined above
- vector$<$ vector$<$ int $>$ $>$ ver_type_list

    the list of all distinct vertex types, obtained from the ver_type array. This is constructed in such a way that ver_type↩ _list[ver_type_dict[x]] = x
- vector$<$ int $>$ ver_type_int

    vertex type converted to integers, using the ver_type_dict map, i.e. ver_type_int[v] = ver_type_dict[ver_type[v]]

3.4.1 Detailed Description

this class defines a colored graph, which is obtained from a simple marked graph and the color of edges come from the type of edges

quick member overview:

- There is a reference to a marked_graph object G,
- h and Delta are parameters that determine depth and maximum degree to form edge types,
- M is a member with type graph_message that is used to form edge types,
- nu_vertices: number of vertices in the graph
- adj_list: the adjacency list of vertices, which also includes edge colors
- adj_location: map for finding where neighbors of vertices are in the adjacency list
- ver_type: a vector for each vertex, containing mark + vectorized degree matrix
- ver_type_dict: dictionary mapping vertex mark + degree matrix to integer
- ver_type_list: list of "distinct" vertex types

- ver_type_int: vertex types converted to integers

Sample Usage

```
marked_graph G;
... //define G
int h = 10;
int Delta = 5;
colored_graph C(G, h, Delta);
```

3.4.2 Constructor & Destructor Documentation

3.4.2.1 colored_graph()

```
colored_graph::colored_graph (
              const marked_graph & graph,
              int depth,
              int max_degree )  [inline]
```

constructor from a graph, depth and maximum degree parameters

```
                                                              : G(graph),
M(graph, depth, max_degree), h(depth), Delta(max_degree)
{
  init(); // initialize other variables
}
```

3.4.3 Member Function Documentation

3.4.3.1 init()

```
void colored_graph::init ( )
```

initializes other variables

- updates messages for M
- updates adj_list
- updates ver_type, ver_type_dict, ver_type_list, ver_type_int
- to make sure, checks whether the sum of degree matrices is symmetric

```
{
  nu_vertices = G.nu_vertices;
  adj_location = G.adj_location; // neighborhood structure is the same as the given graph
  // assigning edge colors based on the messages given by M
  M.update_messages();
  adj_list.resize(nu_vertices);

  // updating adj_list
  int w, my_location, color_v, color_w;
  for (int v=0;v<nu_vertices;v++){
    adj_list[v].resize(G.adj_list[v].size()); // the same number of neighbors here
    for (int i=0;i<G.adj_list[v].size();i++){
      w = G.adj_list[v][i].first; // the ith neighbor, the same as in G
      my_location = G.adj_location[w].at(v); // where v stands among the neighbors of w
      color_v = M.message_dict[M.messages[v][i][h-1]]; // the color towards v corresponds to the message v sends to w
      color_w = M.message_dict[M.messages[w][my_location][h-1]]; // the color towards w is the message w sends towards v
      adj_list[v][i] = pair<int, pair<int, int> >(w, pair<int, int>(color_v, color_w)); // add w as a neighbor, in the same order as in G, and add the colors towards v and w
    }
  }

  // updating the vertex type sequence, dictionary and list, i.e. variables ver_type, ver_type_dict and ver_type_list
  // we also update ver_type_int

  int L = M.message_list.size(); // the number of messages
  ver_type.resize(nu_vertices);
  ver_type_int.resize(nu_vertices);
  for (int v=0;v<nu_vertices;v++){
    ver_type[v].resize(1 + L * L);
    ver_type[v][0] = G.ver_mark[v];
    for (int i=0;i<adj_list[v].size();i++){
      //if (adj_list[v][i].second.first < M.message_list.size()){ // equivalently, the edge is not * typed, since all * typed messages are after L by sorting
      ver_type[v][1 +  adj_list[v][i].second.first * L + adj_list[v][i].second.second] ++;
      //}
    }
    if (ver_type_dict.find(ver_type[v]) == ver_type_dict.end()){
      ver_type_list.push_back(ver_type[v]);
      ver_type_dict[ver_type[v]] = ver_type_list.size() -1;
    }
    ver_type_int[v] = ver_type_dict[ver_type[v]];
  }

  // checking whether the sum of degrees is symmetric
  vector<int> sum;
  sum.resize(1 + L * L);
  for (int v=0;v<nu_vertices;v++)
    for (int i=0;i<1 + L * L;i++)
      sum[i] += ver_type[v][i];
  for (int i=0;i<L;i++){
    for (int j=0;j<L;j++){
      cout << sum[1+i*L + j] << " ";
      if (sum[1+i*L + j] != sum[1+j*L+i])
        cout << " DANGER! the sum matrix is not symmetric" << endl;
    }
    cout << endl;
  }
}
```

3.4.4 Member Data Documentation

3.4.4.1 adj_list

`vector<vector<pair<int, pair<int, int> > > > colored_graph::adj_list`

adj_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, color component towards i, color component towards other endpoint). Therefore, the color of an edge between v and its ith neighbor is of the form (adj_list[v][i].second.first, adj_list[v][i].second.second)

3.4.4.2 adj_location

```
vector<map<int,int> > colored_graph::adj_location
```

adj_location[v] for $0 \leq v < n$, is a map, where adj_location[v][w] denotes the index in adj_list[v] where the information regarding the edge between v and w is stored. Hence, adj_location[v][w] does not exist if w is not adjacent to v, and adj_list[v][adj_location[v][w]] is the edge between v and w

3.4.4.3 Delta

```
int colored_graph::Delta
```

the maximum degree threshold

3.4.4.4 G

```
const marked_graph& colored_graph::G
```

the marked graph from which this is created

3.4.4.5 h

```
int colored_graph::h
```

the depth up to which look at edge types

3.4.4.6 M

```
graph_message colored_graph::M
```

we use the message passing algorithm of class graph_message to find out edge types

3.4.4.7 nu_vertices

```
int colored_graph::nu_vertices
```

the number of vertices in the graph.

3.4.4.8 ver_type

```
vector<vector<int> > colored_graph::ver_type
```

vertex mark and the colored degree matrix of each vertex. For a vertex v, D[v] is a vector of size $1 + L \times L$, where the first entry is the vertex mark, and the rest is the colored degree matrix row by row. Here, L denotes the number of colors.

3.4.4.9 ver_type_dict

```
map<vector<int>, int > colored_graph::ver_type_dict
```

the dictionary mapping vertex types to integers, obtained from the ver_type array defined above

3.4.4.10 ver_type_int

```
vector<int> colored_graph::ver_type_int
```

vertex type converted to integers, using the ver_type_dict map, i.e. ver_type_int[v] = ver_type_dict[ver_type[v]]

3.4.4.11 ver_type_list

```
vector<vector<int> > colored_graph::ver_type_list
```

the list of all distinct vertex types, obtained from the ver_type array. This is constructed in such a way that ver_↩
type_list[ver_type_dict[x]] = x

The documentation for this class was generated from the following files:

- graph_message.h
- graph_message.cpp

3.5 fenwick_tree Class Reference

Fenwick tree class.

```
#include <fenwick.h>
```

Public Member Functions

- fenwick_tree ()
  default constructor
- fenwick_tree (vector< int >)
  constructor, which takes a vector of values and initializes
- int size ()
  the size of the array, which is sums.size()-1, since sums is one based
- void add (int k, int val)
- int sum (int k)

Private Attributes

- vector< int > sums
  a one based vector containing sum of values

3.5.1 Detailed Description

Fenwick tree class.

this class computes the partial sums of an array. More precisely, we feed it a vector of integers, and it can compute the sum of values up to a certain index efficiently. Moreover, we can change the value of an index. Both these operations are done in $O(\log n)$ where n is the size of the array.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 fenwick_tree() [1/2]

```
fenwick_tree::fenwick_tree ( )  [inline]
```

default constructor

```
{
  sums.resize(0);
}
```

3.5.2.2 fenwick_tree() [2/2]

```
fenwick_tree::fenwick_tree (
              vector< int > vals )
```

constructor, which takes a vector of values and initializes

```
{
  int n = vals.size();
  sums.resize(n+1);
  // initializes at zero
  for (int i=1;i<=n;i++)
    sums[i] = 0;
  for (int i=0;i<n;i++)
    add(i,vals[i]); // add values one by one
}
```

3.5.3 Member Function Documentation

3.5.3.1 add()

```
void fenwick_tree::add (
            int k,
            int val )
```

gets a (zero based) index k, and add to that value

Parameters

k	the index to be modified, this is zero based
val	the value to be added to the above index

```
{
  k = k +1; // the sums vector is one based while the index k was zero based
  while (k < sums.size()){
    sums[k] += val;
    k += (k & -k);
  }
}
```

3.5.3.2 size()

```
int fenwick_tree::size ( ) [inline]
```

the size of the array, which is sums.size()-1, since sums is one based

```
  {
    return sums.size() - 1;
  }
```

3.5.3.3 sum()

```
int fenwick_tree::sum (
            int k )
```

returns the sum of values from 0 to k

Parameters

k	the index up to which (including) the sum is computed

```
{
  k = k +1; // the sums vector is one based while the index k was zero based
  int sum_computed = 0;
  while (k > 0){
    sum_computed += sums[k];
    k -= (k & -k); // reduce the lsb bit
  }
  return sum_computed;
}
```

3.5.4 Member Data Documentation

3.5.4.1 sums

`vector<int> fenwick_tree::sums [private]`

a one based vector containing sum of values

sums[k] contains the sum of values in the interval (k-lsb(k), k]. Here lsb(k) denotes the rightmost one in k.

The documentation for this class was generated from the following files:

- fenwick.h
- fenwick.cpp

3.6 graph Class Reference

simple unmarked graph

`#include <simple_graph.h>`

Public Member Functions

- graph (vector< vector< int > > list, vector< int > deg)

  a constructor
- vector< int > get_forward_list (int v) const

  returns the forward adjacency list of a given vertex
- int get_forward_degree (int v) const

  returns the forward degree of a vertex v
- int get_degree (int v) const

  returns the overall degree of a vertex
- vector< int > get_degree_sequence () const

  returns the whole degree sequence
- int nu_vertices () const

  the number of vertices in the graph

Private Attributes

- int n

  the number of vertices in the graph
- vector< vector< int > > forward_adj_list

  for a vertex $0 \leq v < n$, forward_adj_list[v] is a vector containing vertices w such that are adjacent to v and also $w > v$, i.e. the adjacent vertices in the forward direction. For such v, forward_adj_list[v] is sorted increasing.
- vector< int > degree_sequence

  the degree sequence of the graph, where the degree of a vertex is the number of all edges connected to it (not just the ones with greater index).

Friends

- ostream & operator<< (ostream &o, const graph &G)

  printing the graph to the output
- bool operator== (const graph &G1, const graph &G2)

  comparing two graphs for equality
- bool operator!= (const graph &G1, const graph &G2)

  comparing for inequality

3.6.1 Detailed Description

simple unmarked graph

3.6.2 Constructor & Destructor Documentation

3.6.2.1 graph()

```
graph::graph (
            vector< vector< int > > list,
            vector< int > deg )
```

a constructor

This constructor takes the list of adjacent vertices and the degree sequence, and constructs an object.

Parameters

list	list[v] is the list of vertices w adjacent to v such that $w > v$. However, this list does not have to be sorted.
deg	deg[v] is the overall degree of the vertex (not only the ones with greater index).

```
{
  n = list.size();
  forward_adj_list = list;
  // sorting the list
  for (int v=0; v<n; v++)
```

```
    sort(forward_adj_list[v].begin(), forward_adj_list[v].end());
  degree_sequence = deg;
}
```

3.6.3 Member Function Documentation

3.6.3.1 get_degree()

```
int graph::get_degree (
              int v ) const
```

returns the overall degree of a vertex

```
                              {
  return degree_sequence[v];
}
```

3.6.3.2 get_degree_sequence()

```
vector< int > graph::get_degree_sequence ( ) const
```

returns the whole degree sequence

```
                                          {
  return degree_sequence;
}
```

3.6.3.3 get_forward_degree()

```
int graph::get_forward_degree (
              int v ) const
```

returns the forward degree of a vertex v

```
                                      {
  if (v < 0 or v >= n)
    cerr << "graph::get_forward_degree, index v out of range" << endl;
  return forward_adj_list[v].size();
}
```

3.6.3.4 get_forward_list()

```
vector< int > graph::get_forward_list (
            int v ) const
```

returns the forward adjacency list of a given vertex

```
                                            {
  if (v < 0 or v >= n)
    cerr << "graph::get_forward_list, index v out of range" << endl;
  return forward_adj_list[v];
}
```

3.6.3.5 nu_vertices()

```
int graph::nu_vertices ( ) const
```

the number of vertices in the graph

```
                              {
  return n;
}
```

3.6.4 Friends And Related Function Documentation

3.6.4.1 operator"!=

```
bool operator!= (
            const graph & G1,
            const graph & G2 )   [friend]
```

comparing for inequality

```
{
  return !(G1 == G2);
}
```

3.6.4.2 operator<<

```
ostream& operator<< (
            ostream & o,
            const graph & G )   [friend]
```

printing the graph to the output

```
{
  int n = G.nu_vertices();
  vector<int> list;
  for (int i=0;i<n;i++){
    list = G.get_forward_list(i);
    o << i << " -> ";
    for (int j=0;j<list.size();j++){
      o << list[j];
      if (j < list.size()-1)
        o << ", ";
    }
    o << endl;
  }
  return o;
}
```

3.6.4.3 operator==

```
bool operator== (
            const graph & G1,
            const graph & G2 )   [friend]
```

comparing two graphs for equality

```
{
  int n1 = G1.nu_vertices();
  int n2 = G2.nu_vertices();
  if (n1!= n2)
    return false;
  vector<int> list1, list2;
  for (int v=0; v<n1; v++){
    list1 = G1.get_forward_list(v);
    list2 = G2.get_forward_list(v);
    if (list1 != list2)
      return false;
  }
  return true;
}
```

3.6.5 Member Data Documentation

3.6.5.1 degree_sequence

```
vector<int> graph::degree_sequence   [private]
```

the degree sequence of the graph, where the degree of a vertex is the number of all edges connected to it (not just the ones with greater index).

3.6.5.2 forward_adj_list

`vector<vector<int> > graph::forward_adj_list [private]`

for a vertex $0 \leq v < n$, forward_adj_list[v] is a vector containing vertices w such that are adjacent to v and also $w > v$, i.e. the adjacent vertices in the forward direction. For such v, forward_adj_list[v] is sorted increasing.

3.6.5.3 n

`int graph::n [private]`

the number of vertices in the graph

The documentation for this class was generated from the following files:

- simple_graph.h
- simple_graph.cpp

3.7 graph_decoder Class Reference

Decodes a simple unmarked graph.

`#include <simple_graph_compression.h>`

Public Member Functions

- graph_decoder (vector< int > a_)

  constructor given the degree sequence
- graph decode (mpz_class f, vector< int > tS_)

  given $\tilde{N}$ and a vector $\tilde{S}$, decodes the graph and returns an object of type graph
- pair< mpz_class, mpz_class > decode_node (int i, mpz_class tN)

  decode the node i
- pair< mpz_class, mpz_class > decode_interval (int i, int j, int I, mpz_class tN, int Sj)

  decodes the interval $[i, j]$ with interval index I.

Private Attributes

- vector< int > a

  the degree sequence of the graph.
- int n

  the number of vertices, which is a.size()
- int logn2

  the integer part of $\log^2 n$
- vector< vector< int > > x

  the forward adjacency list of the decoded graph
- vector< int > beta

  the sequence $\vec{\beta}$, where after decoding vertex i, for $i \leq v \leq n$ we have $\beta_v = d_v(i)$.
- reverse_fenwick_tree U

  a Fenwick tree initialized with the degree sequence a, and after decoding vertex i, for $i \leq v$, we have $U_v = \sum_{k=v}^{n-1} d_k(i)$.
- vector< int > tS

  the $\tilde{S}$ vector, which stores the partial sums for the midpoints of intervals with length more than $\log^2 n$.

3.7.1 Detailed Description

Decodes a simple unmarked graph.

This class received a number $\tilde{N}$ and finds a simple unmarked graph. We assume that the degree sequence of the graph is known as well.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 graph_decoder()

```
graph_decoder::graph_decoder (
            vector< int > a_ )
```

constructor given the degree sequence

```
{
  a = a_;
  n = a.size();
  double logn = log(n);
  logn2 = int(logn * logn);
  x.resize(n);
  beta = a;
  U = reverse_fenwick_tree(a);
}
```

3.7.3 Member Function Documentation

3.7.3.1 decode()

```
graph graph_decoder::decode (
            mpz_class f,
            vector< int > tS_ )
```

given $\tilde{N}$ and a vector $\tilde{S}$, decodes the graph and returns an object of type graph

```
{
  tS = tS_;
  //mpz_class prod_a_factorial = 1; // \prod_{i=1}^n a_i!
  //for (int i=0; i<a.size();i++)
  //  prod_a_factorial *= compute_product(a[i], a[i], 1);

  mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=0}^{n-1} a_i!
  mpz_class tN = f * prod_a_factorial;
  decode_interval(0,n-1,1,tN,0);
  return graph(x, a);
}
```

3.7.3.2 decode_interval()

```
pair< mpz_class, mpz_class > graph_decoder::decode_interval (
            int i,
            int j,
            int I,
            mpz_class tN,
            int Sj )
```

decodes the interval $[i, j]$ with interval index I.

Parameters

i,j	intervals endpoints
I	the index of the interval
tN	$\tilde{N}_{i,j}$
Sj	S_{j+1}

Returns

a pair $N_{i,j}, l_{i,j}$ where $N_{i,j} = N_{i,j}(G)$ and $l_{i,j} = l_{i,j}(G)$

```
{
  //cerr << " decode interval " << i << " " << j << " tN " << tN << endl;
  if (i == j)
    return decode_node(i, tN);

  // sweeping for zero nodes

  int t; // place to break
  int St; // S_{t+1}
  if ((j-i) > logn2){
    //cerr << " long interval I = " << I << endl;
    t = (i+j) / 2; // break at middle, since we have \tilde{S}
    St = tS[I]; // looking at the \f$\tilde{S}\f$ vector
  }else{
    //cerr << " short interval " << endl;
    t = i;
    St = U.sum(i) - 2 * beta[i];
  }

  //cerr << " decode interval " << i << " " << j << " t " << t << " St " << St << " Sj " << Sj << endl;
  mpz_class rtj; // \f$t_{t+1, j}\f$
  mpz_class tNit; // \f$\tilde{N}_{i,t}\f$ for the left decoder
  mpz_class tNtj; // \f$\tilde{N}_{t+1, j}\f$ for the right decoder
  mpz_class Nit; // the true N_{i,t} returned by the left decoder
  mpz_class lit; // the true l_{i,t} returned by the left decoder
  mpz_class Ntj; // the true N_{t+1, j} returned by the right decoder
  mpz_class ltj; // the true l_{t+1,j} returned by the right decoder
  mpz_class Nij; // the true N_{i,j} to return
  mpz_class lij; // the true l_{i,j} to return

  pair<mpz_class, mpz_class> ans; // returned by subintervals


  rtj = compute_product(St - 1, (St - Sj)/2, 2);
  //cerr << " interval " << i << " " << j << " t " << t << " St " << St << " rtj " << rtj << endl;
  tNit = tN / rtj;

  // calling the left decoder
  ans = decode_interval(i,t,2*I,tNit, St);
  Nit = ans.first;
  lit = ans.second;

  // reducing the contribution of the left decoder to prepare for the right decoder
  tNtj = (tN - Nit * rtj) / lit;

  // calling the right decoder
  ans = decode_interval(t+1, j, 2*I + 1, tNtj, Sj);
  Ntj = ans.first;
  ltj = ans.second;

  // preparing Nij and lij to return
  Nij = Nit * rtj + lit * Ntj;
  lij = lit * ltj;
  return pair<mpz_class, mpz_class> (Nij, lij);
}
```

3.7.3.3 decode_node()

```
pair< mpz_class, mpz_class > graph_decoder::decode_node (
            int i,
            mpz_class tN )
```

decode the node i

Parameters

i	the vertex index
tN	$\tilde{N}_{i,i}$

Returns

a pair $(N_{i,i}, l_i)$ where $l_i = l_i(G)$ and $N_{i,i} = N_{i,i}(G)$

```
{
  //cerr << " decode node " << i << " tN " << tN << endl;
  //cerr << " beta[i] " << beta[i] << endl;
  //cerr << " beta " << endl;
  //for (int k = i; k< n;k++)
  //  cerr << k << " " << beta[k] << endl;
  //cerr << " U " << endl;
  //for (int k=i;k<n;k++)
  //  cerr << k << " " << U.sum(k) << endl;

  if (beta[i] == 0)
    return pair<mpz_class, mpz_class> (0,1);

  mpz_class li = 1; // l_i(G)
  mpz_class Ni = 0; // N_{i,i}(G)
  int f, g; // endpoints for the binary search
  int t; // midpoint for the binary search
  mpz_class zik, lik;
  for (int k=0;k<beta[i];k++){
    if (k==0)
      f = i+1;
    else
      f = x[i][k-1]+1;
    g = n-1;
    while(g > f){
      //cerr << " f , g " << f << " " << g << endl;
      t = (f+g)/2;
      // binary search:
      if(compute_product(U.sum(t+1), beta[i] - k, 1) <= tN)
        g = t;
      else
        f = t+1;
    }
    x[i].push_back(f);
    zik = compute_product(U.sum(x[i][k]+1), beta[i] - k, 1);
    Ni += li * zik;
    lik = (beta[i] - k) * beta[x[i][k]];
    li *= lik;
    tN -= zik;
    tN /= lik;
    U.add(x[i][k],-1);
    beta[x[i][k]] --;
  }
  //cerr << " decoded for " << i << " x: " << endl;
  //for (int j=0;j<x[i].size(); j++)
  //  cerr << x[i][j] << " " ;
  //cerr << endl;
  return pair<mpz_class, mpz_class> (Ni, li);
}
```

3.7.4 Member Data Documentation

3.7.4.1 a

`vector<int> graph_decoder::a [private]`

the degree sequence of the graph.

3.7.4.2 beta

```
vector<int> graph_decoder::beta [private]
```

the sequence $\vec{\beta}$, where after decoding vertex i, for $i \leq v \leq n$ we have $\beta_v = d_v(i)$.

3.7.4.3 logn2

```
int graph_decoder::logn2 [private]
```

the integer part of $\log^2 n$

3.7.4.4 n

```
int graph_decoder::n [private]
```

the number of vertices, which is a.size()

3.7.4.5 tS

```
vector<int> graph_decoder::tS [private]
```

the $\tilde{S}$ vector, which stores the partial sums for the midpoints of intervals with length more than $\log^2 n$.

3.7.4.6 U

```
reverse_fenwick_tree graph_decoder::U [private]
```

a Fenwick tree initialized with the degree sequence a, and after decoding vertex i, for $i \leq v$, we have $U_v = \sum_{k=v}^{n-1} d_k(i)$.

3.7.4.7 x

```
vector<vector<int> > graph_decoder::x [private]
```

the forward adjacency list of the decoded graph

The documentation for this class was generated from the following files:

- simple_graph_compression.h
- simple_graph_compression.cpp

3.8 graph_encoder Class Reference

Encodes a simple unmarked graph.

```
#include <simple_graph_compression.h>
```

Public Member Functions

- graph_encoder (const graph &Gin)

  constructor
- void init ()

  initializes beta and U, logn2, and resizes Stilde.
- pair< mpz_class, mpz_class > compute_N (int i, int j, int l)

  computes $N_{i,j}(G)$
- pair< mpz_class, vector< int > > encode ()

  Encodes the graph and returns N together with Stilde.

Private Attributes

- const graph & G

  the simple unmarked graph which is going to be encoded
- vector< int > beta

  When compute_N is called for $i \leq j$*, for* $i \leq v \leq n$*, we have* $\beta_v = d_v(i)$.
- reverse_fenwick_tree U

  a Fenwick tree which encodes the forward degrees to the right. When compute_N is called for $i \leq j$*, for* $i \leq v$*, we have* $U_v = \sum_{k=v}^{n} d_k(i)$.
- vector< int > Stilde

  Summation of forward degrees at $n/\log^2 n$ *many points.*
- int logn2

  the integer part of $\log^2 n$ *where* n *is the number of vertices in* G.

3.8.1 Detailed Description

Encodes a simple unmarked graph.

This class has a reference to a simple unmarked graph, and encodes it using the counting algorithm which counts the number of configurations resulting in a graph lexicographically smaller than the reference graph. It is assumed that both the encode and the decode know the number of vertices and also the degree profile of the graph.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 graph_encoder()

```
graph_encoder::graph_encoder (
            const graph & Gin ) [inline]
```

constructor

```
                                : G(Gin)
{
  init();
}
```

3.8.3 Member Function Documentation

3.8.3.1 compute_N()

```
pair< mpz_class, mpz_class > graph_encoder::compute_N (
            int i,
            int j,
            int I )
```

computes $N_{i,j}(G)$

Parameters

i,j	the interval for which we compute $N_{i,j}(G)$
I	The integer index corresponding to the current interval (follows a heap convention).

Returns

A pair, where the first component is $N_{i,j}(G)$ and the second component is $l_{i,j}(G)$.

```
{
  //cerr << " i, j " << i << " , " << j << endl;
  if (i==j){
    mpz_class zi, li, zik, lik;
    zi = 0;
    li = 1;
    vector<int> x = G.get_forward_list(i); // the forward adjacency list of vertex i
    if (beta[i] != x.size())
      cerr << " DANGER! beta[i] is not the same as x.size()!!" << endl;
    for (int k=0;k<x.size();k++){
      zik = compute_product(U.sum(1+x[k]), beta[i] - k, 1); // we are zero based here, so instead of -k + 1, we have -k
      zi += li * zik;
      lik = (beta[i] - k) * beta[x[k]]; // we are zero based here, so instead of -k + 1, we have -k
      li *= lik;
      beta[x[k]] --;
      U.add(x[k], -1);
    }
    //cerr << " returning (" << i << " , " << j << ") N " << zi << " l " << li << endl;
    return pair<mpz_class, mpz_class>(zi, li);
  }else{
    int t = (i+j) / 2;
    mpz_class Nit, lit; // \f$N_{i,t}\f$ and \f$l_{[i:t]}\f$
    mpz_class Ntj, ltj; // \f$N_{t+1,j}\f$ and \f$l_{[t+1:j]}\f$
    mpz_class Nij, lij; // \f$N_{i,j}\f$ and \f$l_{[i:j]}\f$
    int St, Sj; // \f$S_{t+1}\f$ and \f$S_{j+1}\f$
```

```
    mpz_class rtj; // \f$r_{[t+1:j]}\f$

    pair<mpz_class, mpz_class> return_left = compute_N(i,t, 2*I); // calling for the interval i,t
    Nit = return_left.first;
    lit = return_left.second;
    St = U.sum(t+1);

    if (j - i > logn2){// we should save the midpoint sum St
      //cerr << " i " << i << " j " << j << " I " << I << " storing St " << St << endl;
      if (I >= Stilde.size() ){
        cerr << " BAD: I out of range I " << I << " Stilde.size() "<< Stilde.size() << endl;
        cerr << " i " << i << " j " << j << " logn2 " << logn2 << endl;
      }
      Stilde[I] = St;
    }

    pair<mpz_class, mpz_class> return_right = compute_N(t+1, j, 2*I + 1); // calling for the
    interval t+1, j
    Ntj = return_right.first;
    ltj = return_right.second;
    Sj = U.sum(j+1);
    rtj = compute_product(St-1, (St - Sj)/2, 2);
    //cerr << "( " << i << " , " << j << ") St " << St << " Sj " << Sj << " rtj " << rtj << endl;
    Nij = Nit * rtj + lit * Ntj ;
    lij = lit * ltj;
    //cerr << " returning (" << i << " , " << j << ") N " << Nij << " l " << lij << endl;
    return pair<mpz_class, mpz_class> (Nij, lij);
  }
}
```

3.8.3.2 encode()

`pair< mpz_class, vector< int > > graph_encoder::encode ( )`

Encodes the graph and returns N together with Stilde.

Returns

A pair, where the first component is $\lceil N(G)/\prod_{i=1}^{n} a_i! \rceil$ where $N(G) = N_{0,n-1}(G)$ and a is the degree sequence of the graph, and the second component is the vector Stilde which stores partial mid sum of intervals and has length roughly $n/\log^2 n$

```
                                                 {
  pair<mpz_class, mpz_class> ans  = compute_N(0,G.nu_vertices()-1,1);
  vector<int> a = G.get_degree_sequence();  // the graph degree sequence
  //mpz_class prod_a_factorial = 1; // \prod_{i=1}^n a_i!
  //for (int i=0; i<a.size();i++)
  //  prod_a_factorial *= compute_product(a[i], a[i], 1);
  mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=1}^n a_i!
  // we need the ceiling of the ratio of ans.first and prod_a_factorial
  bool ceil = false; // if true, we will add one to the integer division
  if (ans.first % prod_a_factorial != 0)
    ceil = true;
  ans.first /= prod_a_factorial;
  if (ceil)
    ans.first ++;
  return pair<mpz_class, vector<int> > (ans.first, Stilde);
}
```

3.8.3.3 init()

```
void graph_encoder::init ( )
```

initializes beta and U, logn2, and resizes Stilde.

```
{
  // initializing the beta sequence
  beta = G.get_degree_sequence();

  //beta.resize(G.nu_vertices());
  //for (int v=0;v<G.nu_vertices();v++)
  //  beta[v] = G.get_degree(v);

  // initializing the Fenwick Tree
  U = reverse_fenwick_tree(beta);

  // initializing logn2
  int n = G.nu_vertices();
  double logn = log(n);
  logn2 = int(logn * logn);

  //initializing the partial sum vector Stilde
  Stilde.resize(4 * n / logn2); // TO CHECK,
    2018-10-18_self-compression_Stilde-size-required-2nlogn2.pdf
  Stilde[0] = 0;
}
```

3.8.4 Member Data Documentation

3.8.4.1 beta

```
vector<int> graph_encoder::beta [private]
```

When compute_N is called for $i \leq j$, for $i \leq v \leq n$, we have $\beta_v = d_v(i)$.

3.8.4.2 G

```
const graph& graph_encoder::G [private]
```

the simple unmarked graph which is going to be encoded

3.8.4.3 logn2

```
int graph_encoder::logn2 [private]
```

the integer part of $\log^2 n$ where n is the number of vertices in G.

3.8.4.4 Stilde

```
vector<int> graph_encoder::Stilde [private]
```

Summation of forward degrees at $n/\log^2 n$ many points.

3.8.4.5 U

```
reverse_fenwick_tree graph_encoder::U [private]
```

a Fenwick tree which encodes the forward degrees to the right. When compute_N is called for $i \leq j$, for $i \leq v$, we have $U_v = \sum_{k=v}^{n} d_k(i)$.

The documentation for this class was generated from the following files:

- simple_graph_compression.h
- simple_graph_compression.cpp

3.9 graph_message Class Reference

this class takes care of message passing on marked graphs.

```
#include <graph_message.h>
```

Public Member Functions

- graph_message (const marked_graph &graph, int depth, int max_degree)

    constructor, given reference to a graph
- void update_messages ()

    performs the message passing algorithm and updates the messages array accordingly
- void update_message_dictionary ()

    update message_dict and message_list

Public Attributes

- const marked_graph & G

    reference to the marked graph for which we do message passing
- int h

    the depth up to which we do message passing (the type of edges go through depth h-1)
- int Delta

    the maximum degree threshold
- vector< vector< vector< vector< int > > > > messages

    messages[v][i][t] is the message at time t from vertex v towards its ith neighbor (in the order given by adj_list of vertex i in graph G). Messages will be useful to find edge types
- map< vector< int >, int > message_dict

    the message dictionary (at depth t=h-1), which maps each message to its corresponding index in the dictionary
- vector< vector< int > > message_list

    the list of messages present in the graph, stored in an order consistent with message_dict, i.e. for a message m, if messsage_dict[m] = i, then message_list[i] = m.

3.9.1 Detailed Description

this class takes care of message passing on marked graphs.

This graph has a reference to a marked_graph object for which we perform message passing to find edge types. The edge types are discovered up to depth h-1, and with degree parameter Delta, where h and Delta are member objects.

Sample Usage

```
marked_graph G;
... //define G
int h = 10;
int Delta = 5;
graph_message M(G, h, Delta);
M.update_messages();
```

3.9.2 Constructor & Destructor Documentation

3.9.2.1 graph_message()

```
graph_message::graph_message (
            const marked_graph & graph,
            int depth,
            int max_degree )  [inline]
```

constructor, given reference to a graph

```
                                                            : G(graph){
    h = depth;
    Delta = max_degree;
  }
```

3.9.3 Member Function Documentation

3.9.3.1 update_message_dictionary()

```
void graph_message::update_message_dictionary ( )
```

update message_dict and message_list

The message_list is sorted in reverse order so that all $*$ messages (those messages starting with -1) go to the end of the list.

```
{
  vector<int> message;
  for (int v=0;v<G.nu_vertices;v++){
    for (int i=0;i<G.adj_list[v].size();i++){
      message = messages[v][i][h-1];
      if(message_dict.find(message) == message_dict.end()){
        // the message does not exist in the dictionary, hence add it
        message_dict[message] = message_list.size(); // so that it points to the last element in message_list which is going to be added in the next line, this assures that if message_dict[m] = i, then message_list[i] = m
        message_list.push_back(message); // add the message to the list
      }
    }
  }

  // we want all the * messages to be together so that later we can easily distinguish between * messages and normal messages.
  // in order to do this, we simply sort the message list
  sort (message_list.begin(), message_list.end());
  // but, since we want the * messages which start by -1 to go to the end of the list, after sorting, we reverse the vector as well
  reverse(message_list.begin(), message_list.end());
  // then, we update message_dict accordingly
  // at the same time, we count the number of non * messages, i.e. L
  //L = 0;
  for (int i=0;i<message_list.size();i++){
    message_dict[message_list[i]] = i;
    //if (message_list[i][0] != -1)
    //L++;
  }
}
```

3.9.3.2 update_messages()

```
void graph_message::update_messages ( )
```

performs the message passing algorithm and updates the messages array accordingly

The structure of messages is as follows. To simplify the notation, we use $M_k(v,w)$ to denote the message sent from v towards w at time step k, this is in fact messages[v][i][t] where i is the index of w among neighbors of v.

- For $k = 0$, we have $M_0(v,w) = (\tau_G(v), 0, \xi_G(w,v))$ where $\tau_G(v)$ is the mark of vertex v and $\xi_G(w,v)$ denotes the mark of the edge between v and w towards v.
- For $k > 0$, if the degree of v is bigger than Delta, we have $M_k(v,w) = (-1, \xi_G(w,v))$.
- Otherwise, we form the list $(s_u : u \sim_G v, u \neq w)$, where for $u \sim_G v, u \neq w$, we set $s_u = (M_{k-1}(u,v), \xi_G(u,v))$.
- If for some $u \sim_G v, u \neq w$, the sequence s_u starts with a -1, we set $M_k(v,w) = (-1, \xi_G(w,v))$.
- Otherwise, we sort the sequences s_u nondecreasingly with respect to the lexicographic order and set s to be the concatenation of the sorted list. Finally, we set $M_k(v,w) = (\tau_G(v), \deg_G(v) - 1, s, \xi_G(w,v))$.

```
{
 int nu_vertices = G.nu_vertices;

 messages.resize(nu_vertices);

 // initialize the messages
 for (int v=0;v<nu_vertices;v++){
   messages[v].resize(G.adj_list[v].size());
   for (int i=0;i<G.adj_list[v].size();i++){
     // the message from v towards the ith neighbor (lets call is w) at time 0 has a mark component which is \xi(v,w) and a subtree component which is a single root with mark \tau(v). This is encoded as a message vector with size 3 of the form (\tau(v), 0,\xi(v,w)) where the last 0 indicates that there is no offspring.
     messages[v][i].resize(h);
     // initialize messages to be empty
     for (int t=0;t<h;t++)
```

```
        messages[v][i][t].resize(0);

      vector<int> m;
      m.push_back(G.ver_mark[v]);
      m.push_back(0);
      m.push_back(G.adj_list[v][i].second.first);
      messages[v][i][0] = m; // the message at time 0
    }
  }

  // updating messages
  for (int t=1;t<h;t++){
    for (int v=0;v<nu_vertices;v++){
      if (G.adj_list[v].size() <= Delta){
        // the degree of v is no more than Delta
        // do the standard message passing by aggregating messages from neighbors
        // stacking all the messages from neighbors of v towards v
        vector<pair<vector<int>, int> > neighbor_messages; // the first component is the message and the second is the name of the neighbor
        // the second component is stored so that after sorting, we know the owner of the message

        // the message from each neighbor of v, say w,  towards v is considered, the mark of the edge between w and v towards v is added to it, and then all these objects are stacked in neighbor_messages to be sorted and used afterwards
        for (int i=0;i<G.adj_list[v].size();i++){
          int w = G.adj_list[v][i].first; // what is the name of the neighbor I am looking at now, which is the ith neighbor of vertex v
          int my_location = G.adj_location[w].at(v); // where is the place of node v among the list of neighbors of the ith neighbor of v
          vector<int> previous_message = messages[w][my_location][t-1]; // the message sent from this neighbor towards v at time t-1
          previous_message.push_back(G.adj_list[v][i].second.first); // adding the mark towards v to the list
          neighbor_messages.push_back(pair<vector<int> , int> (previous_message, w));
        }

        sort(neighbor_messages.begin(), neighbor_messages.end(), pair_compare);
        for (int i=0;i<G.adj_list[v].size();i++){
          // let w be the current ith neighbor of v
          int w = G.adj_list[v][i].first;
          // first, start with the mark of v and the number of offsprings in the subgraph component of the message
          messages[v][i][t].push_back(G.ver_mark[v]); // mark of v
          messages[v][i][t].push_back(G.adj_list[v].size()-1); // the number of offsprings in the subgraph component of the message
          // stacking messages from all neighbors of v expect for w towards v at time t-1
          for (int j=0;j<G.adj_list[v].size();j++){
            if (neighbor_messages[j].second != w){
              if (neighbor_messages[j].first[0] == -1){
                // this means that one of the messages that should be aggregated is * typed, therefore the outgoing messages should also be * typed
                // i.e. the message has only two entries: (-1, \xi(w,v)) where \xi(w,v) is the mark of the edge between v and w towards v
                // since after this loop, the mark \xi(w,v) is added to the message (after the comment starting with 'finally'), we only add the initial -1 part
                messages[v][i][t].resize(0);
                messages[v][i][t].push_back(-1);
                break; // the message is decided, we do not need to go over any of the other neighbor messages, hence break
              }
              // this message should be added to the list of messages
              messages[v][i][t].insert(messages[v][i][t].end(), neighbor_messages[j].first.begin(), neighbor_messages[j].first.end());
            }
          }
          // if we break, we reach at this point and message is (-1), otherwise the message is of the form (\tau(v), \deg(v) - 1, ...) where ... is the list of all neighbor messages towards v except for w.
          // finally, the mark of the edge between v and w towards v, \xi(w,v), should be added to this list
          messages[v][i][t].push_back(G.adj_list[v][i].second.first);
        }
      }else{
        // if the degree of v is bigger than Delta, the message towards all neighbors is of the form *
        // i.e. message of v towards a neighbor w is of the form (-1, \xi(w,v)) where \xi(w,v) is the mark of the edge between v and w towards v
        for (int i=0;i<G.adj_list[v].size();i++){
          messages[v][i][t].resize(2);
          messages[v][i][t][0] = -1;
          messages[v][i][t][1] = G.adj_list[v][i].second.first;
        }
      }
    }
  }

  // now, we should update messages at time h-1 so that if the message from v to w is *, i.e. is of the form (-1,x), then the message from w to v is also of the similar form, i.e. it is (-1,x') where x' = \xi(v,w)
  for (int v=0;v<nu_vertices;v++){
```

```
        for (int i=0;i<G.adj_list[v].size();i++){
          if (messages[v][i][h-1][0] == -1){
            // it is of the form *
            int w = G.adj_list[v][i].first; // the other endpoint of the edge
            int my_location = G.adj_location[w].at(v); // so that adj_list[w][my_location].first = v
            messages[w][my_location][h-1].resize(2);
            messages[w][my_location][h-1][0] = -1;
            messages[w][my_location][h-1][1] = G.adj_list[v][i].second.second; // the mark towards w
          }
        }
      }
      update_message_dictionary(); // update the variables message_dict and message_list
}
```

3.9.4 Member Data Documentation

3.9.4.1 Delta

`int graph_message::Delta`

the maximum degree threshold

3.9.4.2 G

`const marked_graph& graph_message::G`

reference to the marked graph for which we do message passing

3.9.4.3 h

`int graph_message::h`

the depth up to which we do message passing (the type of edges go through depth h-1)

3.9.4.4 message_dict

`map<vector<int>, int> graph_message::message_dict`

the message dictionary (at depth t=h-1), which maps each message to its corresponding index in the dictionary

3.9.4.5 message_list

```
vector<vector<int> > graph_message::message_list
```

the list of messages present in the graph, stored in an order consistent with message_dict, i.e. for a message m, if messsage_dict[m] = i, then message_list[i] = m.

3.9.4.6 messages

```
vector<vector<vector<vector<int> > > > graph_message::messages
```

messages[v][i][t] is the message at time t from vertex v towards its ith neighbor (in the order given by adj_list of vertex i in graph G). Messages will be useful to find edge types

The documentation for this class was generated from the following files:

- graph_message.h
- graph_message.cpp

3.10 marked_graph Class Reference

simple marked graph

```
#include <marked_graph.h>
```

Public Member Functions

- marked_graph ()
  default constructor
- marked_graph (int n, vector< pair< pair< int, int >, pair< int, int > > > edges, vector< int > vertex_marks)
  constructs a marked graph based on edges lists and vertex marks.

Public Attributes

- int nu_vertices
  number of vertices in the graph
- vector< vector< pair< int, pair< int, int > > > > adj_list
  adj_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, mark towards i, mark towards other endpoint)
- vector< map< int, int > > adj_location
  adj_location[v] for $0 \leq v < n$, is a map, where adj_location[v][w] denotes the index in adj_list[v] where the information regarding the edge between v and w is stored. Hence, adj_location[v][w] does not exist if w is not adjacent to v, and adj_list[v][adj_location[v][w]] is the edge between v and w
- vector< int > ver_mark
  ver_mark[i] is the mark of vertex i

3.10.1 Detailed Description

simple marked graph

This class stores a simple marked graph where each vertex carries a mark, and each edge carries two marks, one towards each of its endpoints. The mark of each vertex and each edge is a nonnegative integer.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 marked_graph() [1/2]

marked_graph::marked_graph ( ) [inline]

default constructor

```
{
  nu_vertices = 0;
}
```

3.10.2.2 marked_graph() [2/2]

```
marked_graph::marked_graph (
            int n,
            vector< pair< pair< int, int >, pair< int, int > > > edges,
            vector< int > vertex_marks )
```

constructs a marked graph based on edges lists and vertex marks.

Parameters

n	the number of vertices in the graph
edges	a vector, where each element is of the form $((i,j),(x,y))$ where $i \neq j$ denotes the endpoints of the edge, x is the mark towards i and y is the mark towards j
vertex_marks	is a vector of size n, where vertex_marks[i] is the mark of vertex i

```
{
  nu_vertices = n;
  adj_list.resize(n);
  adj_location.resize(n);
  for (int k=0; k<edges.size(); k++){
    // (i,j) are endpoints if the edge
     // (x,y) are marks, x towards i and y towards j
     int i = edges[k].first.first;
     int j = edges[k].first.second;
     int x = edges[k].second.first;
     int y = edges[k].second.second;
     if (i <0 || i >= n || j < 0 || j >= n || i ==j)
       cerr << " ERROR: graph::graph(n, edges) received an invalid pair of edges with n = " << n << " : (" <
    < i << " , " << j << " )" << endl;
```

```
    adj_list[i].push_back(pair<int, pair<int, int> > (j, pair<int, int> (x,y)));
    adj_location[i][j] = adj_list[i].size() - 1;
    adj_list[j].push_back(pair<int, pair<int, int> > (i, pair<int, int> (y,x)));
    adj_location[j][i] = adj_list[j].size() - 1;
  }
  ver_mark = vertex_marks;
}
```

3.10.3 Member Data Documentation

3.10.3.1 adj_list

`vector<vector<pair<int, pair<int, int> > > > marked_graph::adj_list`

adj_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, mark towards i, mark towards other endpoint)

3.10.3.2 adj_location

`vector<map<int,int> > marked_graph::adj_location`

adj_location[v] for $0 \leq v < n$, is a map, where adj_location[v][w] denotes the index in adj_list[v] where the information regarding the edge between v and w is stored. Hence, adj_location[v][w] does not exist if w is not adjacent to v, and adj_list[v][adj_location[v][w]] is the edge between v and w

3.10.3.3 nu_vertices

`int marked_graph::nu_vertices`

number of vertices in the graph

3.10.3.4 ver_mark

`vector<int> marked_graph::ver_mark`

ver_mark[i] is the mark of vertex i

The documentation for this class was generated from the following files:

- marked_graph.h
- marked_graph.cpp

3.11 reverse_fenwick_tree Class Reference

similar to the fenwick_tree class, but instead of prefix sums, this class computes suffix sums.

```
#include <fenwick.h>
```

Public Member Functions

- reverse_fenwick_tree ()
  default constructor
- reverse_fenwick_tree (vector< int >)
  constructor which receives values and initializes
- void add (int k, int val)
- int size ()
  the number of elements in the original array
- int sum (int k)

Private Attributes

- fenwick_tree FT
  member of type fenwick_tree, which saves the partial sums for the reversed array.

3.11.1 Detailed Description

similar to the fenwick_tree class, but instead of prefix sums, this class computes suffix sums.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 reverse_fenwick_tree() [1/2]

```
reverse_fenwick_tree::reverse_fenwick_tree ( )  [inline]
```

default constructor

```
58 {}
```

3.11.2.2 reverse_fenwick_tree() [2/2]

```
reverse_fenwick_tree::reverse_fenwick_tree (
            vector< int > vals )
```

constructor which receives values and initializes

```
{
  reverse(vals.begin(),vals.end()); // reverse the array and then use the previously defined fenwick_tree
    class
  FT = fenwick_tree(vals);
}
```

3.11.3 Member Function Documentation

3.11.3.1 add()

```
void reverse_fenwick_tree::add (
            int k,
            int val )
```

gets a (zero based) index k, and add to that value

Parameters

k	the index to be modified, this is zero based
val	the value to be added to the above index

```
{
  FT.add(FT.size() - 1 - k, val);
}
```

3.11.3.2 size()

```
int reverse_fenwick_tree::size ( ) [inline]
```

the number of elements in the original array

```
            {
    return FT.size();
  }
```

3.11.3.3 sum()

```
int reverse_fenwick_tree::sum (
            int k )
```

returns the sum of values from index k until the end of the array

Parameters

k	the index from which (including) the sum is computed

```
{
  if (k >= size())
    return 0;
  return FT.sum(FT.size() - 1 - k);
}
```

3.11.4 Member Data Documentation

3.11.4.1 FT

```
fenwick_tree reverse_fenwick_tree::FT  [private]
```

member of type fenwick_tree, which saves the partial sums for the reversed array.

The documentation for this class was generated from the following files:

- fenwick.h
- fenwick.cpp

Chapter 4

File Documentation

4.1 bipartite_graph.cpp File Reference

```
#include "bipartite_graph.h"
```

Functions

- ostream & operator<< (ostream &o, const b_graph &G)
- bool operator== (const b_graph &G1, const b_graph &G2)
- bool operator!= (const b_graph &G1, const b_graph &G2)

4.1.1 Function Documentation

4.1.1.1 operator"!=()

```
bool operator!= (
            const b_graph & G1,
            const b_graph & G2 )
```

```
{
  return !(G1 == G2);
}
```

4.1.1.2 operator<<()

```
ostream& operator<< (
        ostream & o,
        const b_graph & G )
```

```
{
  int n = G.nu_left_vertices();
  vector<int> list;
  for (int i=0;i<n;i++){
    list = G.get_adj_list(i);
    o << i << " -> ";
    for (int j=0;j<list.size();j++){
      o << list[j];
      if (j < list.size()-1)
        o << ", ";
    }
    o << endl;
  }
  return o;
}
```

4.1.1.3 operator==()

```
bool operator== (
        const b_graph & G1,
        const b_graph & G2 )
```

```
{
  int n1 = G1.nu_left_vertices();
  int n2 = G2.nu_left_vertices();

  int np1 = G1.nu_right_vertices();
  int np2 = G2.nu_right_vertices();
  if (n1!= n2 or np1 != np2)
    return false;
  vector<int> list1, list2;
  for (int v=0; v<n1; v++){
    list1 = G1.get_adj_list(v);
    list2 = G2.get_adj_list(v);
    if (list1 != list2)
      return false;
  }
  return true;
}
```

4.2 bipartite_graph.h File Reference

```
#include <iostream>
#include <vector>
```

Classes

- class b_graph

  simple unmarked bipartite graph

4.3 bipartite_graph_compression.cpp File Reference

```
#include "bipartite_graph_compression.h"
```

4.4 bipartite_graph_compression.h File Reference

```
#include <iostream>
#include <vector>
#include "compression_helper.h"
#include "bipartite_graph.h"
#include "fenwick.h"
```

Classes

- class b_graph_encoder

  Encodes a simple unmarked bipartite graph.
- class b_graph_decoder

  Decodes a simple unmarked bipartite graph.

4.5 compression_helper.cpp File Reference

```
#include "compression_helper.h"
```

Functions

- mpz_class compute_product (mpz_class N, mpz_class k, int s)

  This function computes the product of consecutive integers separated by a given iteration.
- mpz_class binomial (const mpz_class n, const mpz_class m)

  computes the binomial coefficient n choose m = n! / m! (n-m)!
- mpz_class prod_factorial (const vector< int > &a, int i, int j)

  computes the product of factorials in a vector given a range

4.5.1 Function Documentation

4.5.1.1 binomial()

```
mpz_class binomial (
            const mpz_class n,
            const mpz_class m )
```

computes the binomial coefficient n choose m = n! / m! (n-m)!

Parameters

n	integer
m	integer

Returns

the binomial coefficient n! / m! (n-m)!. If n <= 0, or m > n, or m <= 0, returns 0

```
{
  if (n <= 0 or m > n or m <= 0)
    return 0;
  return compute_product(n, m, 1) / compute_product(m, m, 1);
}
```

4.5.1.2 compute_product()

```
mpz_class compute_product (
              mpz_class N,
              mpz_class k,
              int s )
```

This function computes the product of consecutive integers separated by a given iteration.

Parameters

N	The first term in the product
k	the number of terms in the product
s	the iteration

Returns

the product N $N \times (N-s) \times (N-2s) \times \ldots \times (N-(k-1)s)$

```
                                                         {
 //cerr << " compute_product  N " << N << " k " << k << " s " << s << endl;

 if (k==1)
   return N;
 if (k == 0) // TO CHECK because there are no terms to compute product
    return 1;

  if (k < 0){
    cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s << endl;
    return 1;
  }
  if (N - (k-1) * s <= 0) // the terms go negative
    return 0;

  // we do this by dividing the terms into two parts
  mpz_class m = k / 2; // the middle point
  mpz_class left, right; // each of the half products
  left = compute_product(N, m, s);
  right = compute_product(N-m * s, k-m, s);
  return left * right;
}
```

4.5.1.3 prod_factorial()

```
mpz_class prod_factorial (
            const vector< int > & a,
            int i,
            int j )
```

computes the product of factorials in a vector given a range

Parameters

a	vector of integers
i,j	endpoints of the interval

Returns

$\prod_{v=i}^{j} a_v!$

```
{
  if (i == j){
    return compute_product(a[i], a[i], 1);
  }else{
    int k = (i+j)/2;
    mpz_class x = prod_factorial(a, i, k);
    mpz_class y = prod_factorial(a, k+1, j);
    return x * y;
  }
}
```

4.6 compression_helper.h File Reference

```
#include <iostream>
#include <gmpxx.h>
#include <vector>
```

Functions

- mpz_class compute_product (mpz_class N, mpz_class k, int s)

  This function computes the product of consecutive integers separated by a given iteration.
- mpz_class binomial (const mpz_class n, const mpz_class m)

  computes the binomial coefficient n choose m = n! / m! (n-m)!
- mpz_class prod_factorial (const vector< int > &a, int i, int j)

  computes the product of factorials in a vector given a range

4.6.1 Function Documentation

4.6.1.1 binomial()

```
mpz_class binomial (
            const mpz_class n,
            const mpz_class m )
```

computes the binomial coefficient n choose m = n! / m! (n-m)!

Parameters

n	integer
m	integer

Returns

the binomial coefficient n! / m! (n-m)!. If n <= 0, or m > n, or m <= 0, returns 0

```
{
  if (n <= 0 or m > n or m <= 0)
    return 0;
  return compute_product(n, m, 1) / compute_product(m, m, 1);
}
```

4.6.1.2 compute_product()

```
mpz_class compute_product (
              mpz_class N,
              mpz_class k,
              int s )
```

This function computes the product of consecutive integers separated by a given iteration.

Parameters

N	The first term in the product
k	the number of terms in the product
s	the iteration

Returns

the product N $N \times (N - s) \times (N - 2s) \times \ldots \times (N - (k - 1)s)$

```
                                                          {
 //cerr << " compute_product  N " << N << " k " << k << " s " << s << endl;

 if (k==1)
   return N;
 if (k == 0) // TO CHECK because there are no terms to compute product
    return 1;

  if (k < 0){
    cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s << endl;
    return 1;
  }
  if (N - (k-1) * s <= 0) // the terms go negative
    return 0;

  // we do this by dividing the terms into two parts
  mpz_class m = k / 2; // the middle point
  mpz_class left, right; // each of the half products
  left = compute_product(N, m, s);
  right = compute_product(N-m * s, k-m, s);
  return left * right;
}
```

4.6.1.3 prod_factorial()

```
mpz_class prod_factorial (
            const vector< int > & a,
            int i,
            int j )
```

computes the product of factorials in a vector given a range

Parameters

a	vector of integers
i,j	endpoints of the interval

Returns

$\prod_{v=i}^{j} a_v!$

```
{
  if (i == j){
    return compute_product(a[i], a[i], 1);
  }else{
    int k = (i+j)/2;
    mpz_class x = prod_factorial(a, i, k);
    mpz_class y = prod_factorial(a, k+1, j);
    return x * y;
  }
}
```

4.7 fenwick.cpp File Reference

```
#include "fenwick.h"
```

4.8 fenwick.h File Reference

```
#include <vector>
```

Classes

- class fenwick_tree

    Fenwick tree class.

- class reverse_fenwick_tree

    similar to the fenwick_tree class, but instead of prefix sums, this class computes suffix sums.

4.9 graph_message.cpp File Reference

```
#include "graph_message.h"
```

Functions

- bool pair_compare (const pair< vector< int >, int > &a, const pair< vector< int >, int > &b)

    used for sorting messages

4.9.1 Function Documentation

4.9.1.1 pair_compare()

```
bool pair_compare (
            const pair< vector< int >, int > & a,
            const pair< vector< int >, int > & b )
```

used for sorting messages

```
                                                                              {
  return a.first < b.first;
}
```

4.10 graph_message.h File Reference

```
#include <vector>
#include <map>
#include "marked_graph.h"
```

Classes

- class graph_message

    this class takes care of message passing on marked graphs.

- class colored_graph

    this class defines a colored graph, which is obtained from a simple marked graph and the color of edges come from the type of edges

Functions

- bool pair_compare (const pair< vector< int >, int > &, const pair< vector< int >, int > &)

    used for sorting messages

4.10.1 Function Documentation

4.10.1.1 pair_compare()

```
bool pair_compare (
            const pair< vector< int >, int > & ,
            const pair< vector< int >, int > &  )
```

used for sorting messages

```
                                                                                  {
  return a.first < b.first;
}
```

4.11 marked_graph.cpp File Reference

```
#include "marked_graph.h"
```

Functions

- istream & operator>> (istream &inp, marked_graph &G)

  inputs a marked_graph

4.11.1 Function Documentation

4.11.1.1 operator>>()

```
istream& operator>> (
            istream & inp,
            marked_graph & G )
```

inputs a marked_graph

The input format is as follows: 1) number of vertices 2) a list of vertex marks as nonnegative integers 3) number of edges 4) for each edge: write $ijxy$, where i and j are the endpoints (here, $0 \leq i,j \leq n-1$ with n being the number of vertices), x is the mark towards i and y is the mark towards j (both nonnegative integers) Example: 2 1 2 1 0 1 1 2 which is a graph with 2 vertices, the mark of vertex 0 is 1 and the mark of vertex 1 is 2, there is one edge between these two vertices with mark 1 towards 0 and mark 2 toward s 1

```
{
  int nu_vertices;
  inp >> nu_vertices;

  vector<int> ver_marks;
  ver_marks.resize(nu_vertices);
  for (int i=0;i<nu_vertices;i++)
    inp >> ver_marks[i];

  int nu_edges;
  inp >> nu_edges;
  vector<pair< pair<int, int> , pair<int, int> > > edges;
  edges.resize(nu_edges);
  for (int i=0;i<nu_edges;i++)
    inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >> edges[i].second.second
  ;

  G = marked_graph(nu_vertices, edges, ver_marks);

  return inp;
}
```

4.12 marked_graph.h File Reference

```
#include <iostream>
#include <vector>
#include <map>
#include <fstream>
```

Classes

- class marked_graph

  simple marked graph

Functions

- istream & operator>> (istream &inp, marked_graph &G)

  inputs a marked_graph

4.12.1 Function Documentation

4.12.1.1 operator>>()

```
istream& operator>> (
            istream & inp,
            marked_graph & G )
```

inputs a marked_graph

The input format is as follows: 1) number of vertices 2) a list of vertex marks as nonnegative integers 3) number of edges 4) for each edge: write $ijxy$, where i and j are the endpoints (here, $0 \leq i,j \leq n-1$ with n being the number of vertices), x is the mark towards i and y is the mark towards j (both nonnegative integers) Example: 2 1 2 1 0 1 1 2 which is a graph with 2 vertices, the mark of vertex 0 is 1 and the mark of vertex 1 is 2, there is one edge between these two vertices with mark 1 towards 0 and mark 2 toward s 1

```
{
  int nu_vertices;
  inp >> nu_vertices;

  vector<int> ver_marks;
  ver_marks.resize(nu_vertices);
  for (int i=0;i<nu_vertices;i++)
    inp >> ver_marks[i];

  int nu_edges;
  inp >> nu_edges;
  vector<pair< pair<int, int> , pair<int, int> > > edges;
  edges.resize(nu_edges);
  for (int i=0;i<nu_edges;i++)
    inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >> edges[i].second.second
   ;

  G = marked_graph(nu_vertices, edges, ver_marks);

  return inp;
}
```

4.13 simple_graph.cpp File Reference

```
#include "simple_graph.h"
```

Functions

- ostream & operator<< (ostream &o, const graph &G)
- bool operator== (const graph &G1, const graph &G2)
- bool operator!= (const graph &G1, const graph &G2)

4.13.1 Function Documentation

4.13.1.1 operator"!=()

```
bool operator!= (
            const graph & G1,
            const graph & G2 )
```

```
{
  return !(G1 == G2);
}
```

4.13.1.2 operator<<()

```
ostream& operator<< (
            ostream & o,
            const graph & G )
```

```
{
  int n = G.nu_vertices();
  vector<int> list;
  for (int i=0;i<n;i++){
    list = G.get_forward_list(i);
    o << i << " -> ";
    for (int j=0;j<list.size();j++){
      o << list[j];
      if (j < list.size()-1)
        o << ", ";
    }
    o << endl;
  }
  return o;
}
```

4.13.1.3 operator==()

```
bool operator== (
            const graph & G1,
            const graph & G2 )
```

```
{
  int n1 = G1.nu_vertices();
  int n2 = G2.nu_vertices();
  if (n1!= n2)
    return false;
  vector<int> list1, list2;
  for (int v=0; v<n1; v++){
    list1 = G1.get_forward_list(v);
    list2 = G2.get_forward_list(v);
    if (list1 != list2)
      return false;
  }
  return true;
}
```

4.14 simple_graph.h File Reference

```
#include <iostream>
#include <vector>
```

Classes

- class graph

  simple unmarked graph

4.15 simple_graph_compression.cpp File Reference

```
#include "simple_graph_compression.h"
```

4.16 simple_graph_compression.h File Reference

```
#include <vector>
#include <math.h>
#include "simple_graph.h"
#include "compression_helper.h"
#include "fenwick.h"
```

Classes

- class graph_encoder

  Encodes a simple unmarked graph.
- class graph_decoder

  Decodes a simple unmarked graph.

4.17 test.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
#include "fenwick.h"
#include "simple_graph.h"
#include "simple_graph_compression.h"
#include "bipartite_graph.h"
#include "bipartite_graph_compression.h"
```

Functions

- ostream & operator<< (ostream &o, const vector< int > &v)
- int main ()

4.17.1 Function Documentation

4.17.1.1 main()

```
int main ( )
```

```
        {
//marked_graph G;
//ifstream inp("star_graph.txt");
//inp >> G;
//graph_message M(G, 10, 2);
//M.update_messages();
vector<int> a = {1,1,2}; // left degree sequence
vector<int> b = {2,2}; // right degree sequence

b_graph G({{0},{1},{0,1}}); // defining the graph

b_graph_encoder E(a,b); // constructing the encoder object
mpz_class f = E.encode(G);

b_graph_decoder D(a, b);
b_graph Ghat = D.decode(f);

if (Ghat == G)
  cout << " successfully decoded the graph! " << endl;


return 0;
// vector<vector<int> > list = {{}, {}, {}};

// b_graph G({{0},{1},{0,1}});
// // cout << G << endl;
// // cout << G.nu_left_vertices() << endl;
// // cout << G.nu_right_vertices() << endl;
// // cout << G.get_left_degree_sequence() << endl;
// // cout << G.get_right_degree_sequence() << endl;
// vector<int> a = G.get_left_degree_sequence();
// vector<int> b = G.get_right_degree_sequence();


// b_graph_encoder  E(a,b);
// mpz_class m = E.encode(G);
// cout << "encoded: " << m << endl;
```

```
  // b_graph_decoder D(a, b);
  // b_graph Ghat = D.decode(m);
  // cout << "decoded graph: " << endl << Ghat << endl;

  // if (Ghat == G)
  //   cout << " equal " << endl;
  // return 0;

}
```

4.17.1.2 operator<<()

```
ostream& operator<< (
              ostream & o,
              const vector< int > & v )
```

```
{
  for (int i=0;i<v.size();i++){
    o << v[i];
    if (i < v.size()-1)
      o << ", ";
  }
  return o;
}
```

Index