Marked Graph Compression

Generated by Doxygen 1.9.8

Chapter 1

Main Page

This is an implementation of the low complexity graph compression algorithm developed and published by Payam Delgosha and Venkat Anantharam.

Relevant Publications

    Delgosha, Payam, and Venkat Anantharam. "A universal low complexity
    compression algorithm for sparse marked graphs." IEEE Journal on
    Selected Areas in Information Theory (2023).

    Delgosha, Payam, and Venkat Anantharam. "A Universal Low Complexity
    Compression Algorithm for Sparse Marked Graphs." arXiv preprint ar↩
    Xiv:2301.05742 (2023).

Installation

Prerequisites

- c++ compiler
- Boost

Compile

`make all`

Usage

In what follows, `gcomp.o` is the compiled executable generated by the makefile.

Compression

The graph should be given as an input text file in the edge list format described as follows.

- The first line specified the numebr of vertices, say `n`.

- The second line includes the mark of `n` vertices each as an integer. If the graph is unmarked, the second line inlcudes `n` zeros.

- The third line includes the number of edges in the graph, say `m`.

- Then we have `m` lines each for an edge represented as `i j x y`, where `i` and `j` are the indices of the end-point, and `x` and `y` are marks (as integers) representing the mark of the edge towards `i` and `j`, respectively.

To compress such a marked graph stored in the file `<input graph>`, use the following command. Here, `h` and `d` are the two height and degree hyperparameters of the compression algorithm (see the paper for details, and the following note regarding these hyperparameters). Also, `<compressed graph file name>` is the name of the file where the compressed graph is going to be stored.
```
gcomp.o -i <input graph> -h <h> -d <d> -o <compressed graph file name>
```

Additional logging options

The following are optional.

- `-v`: verbose mode, which prints the compression / decompression progress.

- `-V <log file>`: if specified, the log is stored in `<log file>`.

Decompression

To decompress a compressed graph stored in file `<compressed graph file name>`, run the following command:
```
gcomp.o -u -i <compressed graph file name> -o <reconstructed graph file name>
```

Here, the `-u` option specifies the decomperssion mode, and the reconstructed graph will be stored in the file `<reconstructed graph file name>`, in the same edge list format explained in the compression phase.

A note on hyperparameters

Regrading the height and degree hyperparameters (`h` and `d`, respectively), please see the paper for more details. In theory, the optimal asymptotic behavior is obtained as both `h` and `d` go to infinity. However, in practice, usually best performacne is obtained for `h` in the range 1, 2, 3. The choice of `d` very much depends on the degree distribution of the graph, but usually a trial with small values such as 2, 5, 10, 20 should yield satisfactory results.

Documentation

Check the html documentation generated by Doxygen in `html/index.html` Also, a pdf version is available in `latex/refman.pdf`.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 helper_vars Namespace Reference

Variables

- mpz_class mul_1
- mpz_class mul_2

  helper variables in order to avoid initialization
- vector< mpz_class > return_stack
- vector< mpz_class > mpz_vec
- vector< mpz_class > mpz_vec2

5.1.1 Variable Documentation

5.1.1.1 mpz_vec

```
vector< mpz_class > helper_vars::mpz_vec  [extern]
```

5.1.1.2 mpz_vec2

```
vector< mpz_class > helper_vars::mpz_vec2  [extern]
```

5.1.1.3 mul_1

```
mpz_class helper_vars::mul_1  [extern]
```

5.1.1.4 mul_2

```
mpz_class helper_vars::mul_2
```

helper variables in order to avoid initialization

5.1.1.5 return_stack

```
vector< mpz_class > helper_vars::return_stack  [extern]
```

Chapter 6

Class Documentation

6.1 b_graph Class Reference

simple unmarked bipartite graph

```
#include <bipartite_graph.h>
```

Public Member Functions

- b_graph ()

    default constructor
- b_graph (const vector< vector< int > > &list, const vector< int > &left_deg, const vector< int > &right_deg)

    a fast constructor getting adjacency list and both left and right degree sequences
- b_graph (const vector< vector< int > > &list, const vector< int > &right_deg)

    a constructor
- b_graph (const 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

6.1.1 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);
    ```

6.1.2 Constructor & Destructor Documentation

6.1.2.1 b_graph() [1/4]

```
b_graph::b_graph ( ) [inline]
```

default constructor
```
00033 : n(0), np(0) {}
```

6.1.2.2 b_graph() [2/4]

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

a fast constructor getting adjacency list and both left and right degree sequences

This constructor takes the adjacency list of left vertices assuming it is sorted, together with left and right degree sequences.

Parameters

list	list[v] is an increasingly sorted list of right nodes adjacent to the left node v
left_deg	left_deg[v] is the degree of the left node v
right_deg	right_deg[w] is the degree of the right node w

```
00004 {
00005   n = left_deg.size();
00006   np = right_deg.size();
00007   adj_list = list;
00008   left_deg_seq = left_deg;
00009   right_deg_seq = right_deg;
00010 }
```

6.1.2.3 b_graph() [3/4]

```
b_graph::b_graph (
              const vector< vector< int > > & list,
              const 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

```
00013 {
00014   n = list.size();
00015   np = right_deg.size(); // the number of right nodes
00016   adj_list = list;
00017   left_deg_seq.resize(n);
00018   // sorting the list
00019   for (int v=0; v<n; v++){
00020     sort(adj_list[v].begin(), adj_list[v].end());
00021     left_deg_seq[v] = adj_list[v].size();
00022   }
00023   right_deg_seq = right_deg;
00024 }
```

6.1.2.4 b_graph() [4/4]

```
b_graph::b_graph (
              const 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

```
00027 {
00028   // goal: finding right degrees and calling the above constructor
00029   // first, we find the number of right nodes
00030   np = 0; // the number of right nodes
00031   n = list.size();
00032   adj_list = list;
00033   left_deg_seq.resize(n);
00034   for (int v=0;v<adj_list.size();v++){
00035     //cerr << " v " << v << endl;
00036     sort(adj_list[v].begin(), adj_list[v].end());
00037     if (adj_list[v].size() > 0 and adj_list[v][adj_list[v].size()-1] > np)
00038       np = adj_list[v][adj_list[v].size()-1];
00039     left_deg_seq[v] = adj_list[v].size();
00040   }
00041   np++; // node indexing is zero based
00042
00043   right_deg_seq.resize(np);
00044   fill(right_deg_seq.begin(), right_deg_seq.end(), 0); // make all elements 0
00045   for (int v=0;v<list.size();v++)
00046     for (int i=0;i<list[v].size();i++)
00047       right_deg_seq[list[v][i]]++;
00048 }
```

6.1.3 Member Function Documentation

6.1.3.1 get_adj_list()

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

returns the adjacency list of a given left vertex
```
00051 {
00052   if (v < 0 or v >= n)
00053     cerr << "b_graph::get_adj_list, index v out of range" << endl;
00054   return adj_list[v];
00055 }
```

6.1.3.2 get_left_degree()

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

returns the degree of a right vertex v
```
00066 {
00067   if (v < 0 or v >= n)
00068     cerr << "b_graph::get_left_degree, index v out of range" << endl;
00069   return left_deg_seq[v];
00070 }
```

6.1.3.3 get_left_degree_sequence()

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

return the left degree sequence
```
00078 {
00079   return left_deg_seq;
00080 }
```

6.1.3.4 get_right_degree()

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

returns the degree of a right vertex v

```
00059 {
00060   if (v < 0 or v >= n)
00061     cerr « "b_graph::get_right_degree, index v out of range" « endl;
00062   return right_deg_seq[v];
00063 }
```

6.1.3.5 get_right_degree_sequence()

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

return the right degree sequence

```
00073 {
00074   return right_deg_seq;
00075 }
```

6.1.3.6 nu_left_vertices()

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

returns the number of left vertices

```
00084 {
00085   return n;
00086 }
```

6.1.3.7 nu_right_vertices()

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

returns the number of right vertices

```
00089 {
00090   return np;
00091 }
```

6.1.4 Friends And Related Symbol Documentation

6.1.4.1 operator"!=

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

comparing for inequality

```
00131 {
00132   return !(G1 == G2);
00133 }
```

6.1.4.2 operator<<

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

printing the graph to the output

```
00095 {
00096   int n = G.nu_left_vertices();
00097   vector<int> list;
00098   for (int i=0;i<n;i++){
00099     list = G.get_adj_list(i);
00100     o « i « " -> ";
00101     for (int j=0;j<list.size();j++){
00102       o « list[j];
00103       if (j < list.size()-1)
00104         o « ", ";
00105     }
00106     o « endl;
00107   }
00108   return o;
00109 }
```

6.1.4.3 operator==

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

comparing two graphs for equality

```
00112 {
00113   int n1 = G1.nu_left_vertices();
00114   int n2 = G2.nu_left_vertices();
00115
00116   int np1 = G1.nu_right_vertices();
00117   int np2 = G2.nu_right_vertices();
00118   if (n1!= n2 or np1 != np2)
00119     return false;
00120   vector<int> list1, list2;
00121   for (int v=0; v<n1; v++){
00122     list1 = G1.get_adj_list(v);
00123     list2 = G2.get_adj_list(v);
00124     if (list1 != list2)
00125       return false;
00126   }
00127   return true;
00128 }
```

6.1.5 Member Data Documentation

6.1.5.1 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.

6.1.5.2 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

6.1.5.3 n

`int b_graph::n [private]`

the number of left vertices

6.1.5.4 np

`int b_graph::np [private]`

the number of right vertices

6.1.5.5 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

6.2 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 $\beta_v = b_v(i)$

6.2.1 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;
```

6.2.2 Constructor & Destructor Documentation

6.2.2.1 b_graph_decoder()

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

constructor
```
00188 {
00189   a = a_;
00190   b = b_;
00191   n = a.size();
00192   np = b.size();
00193   init();
00194 }
```

6.2.3 Member Function Documentation

6.2.3.1 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

```
00270 {
00271   mpz_class prod_b_factorial = prod_factorial(b, 0, np-1);
00272   mpz_class tN = f * prod_b_factorial;
00273   decode_interval(0,n-1,tN);
00274   return b_graph(x, b);
00275 }
```

6.2.3.2 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 $l_{i,j}(G)$

```
00241 {
00242   if (i==j)
00243     return decode_node(i,tN);
00244   int k = (i+j)/ 2; // midpoint to break
00245   int Wk = W.sum(k+1);
00246   int Wj = W.sum(j+1);
00247   mpz_class rkj = compute_product(Wk, Wk - Wj, 1) / prod_factorial(a, k+1, j); // r_{t+1, j}
00248   mpz_class tNik = tN / rkj; // \tilde{N}_{i,k}
00249   pair<mpz_class, mpz_class> ans; // to keep the return for each subinterval
00250
00251   // calling the left subinterval
00252   ans = decode_interval(i,k,tNik);
00253
00254   // preparing for the right subinterval
00255   mpz_class Nik = ans.first;
00256   mpz_class lik = ans.second;
00257   mpz_class tNkj = (tN - Nik * rkj) / lik; // \tilde{N}_{k+1, j}
00258
00259   // calling the right subinterval
00260   ans = decode_interval(k+1, j, tNkj);
00261   mpz_class Nkj = ans.first;
00262   mpz_class lkj = ans.second;
00263   mpz_class Nij = Nik * rkj + lik * Nkj;
00264   mpz_class lij = lik * lkj;
00265   return pair<mpz_class, mpz_class> (Nij, lij);
00266 }
```

6.2.3.3 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 $0 \le i < n$ 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 $l_i(G)$

```
00207 {
00208   mpz_class li = 1;
00209   mpz_class Ni = 0;
00210   int f, g; // endpoints of the interval for binary search
00211   int v;
00212   mpz_class y; // helper
00213   x[i].clear(); // make sure nothing is in the list to be decoded
00214   for (int k=0;k<a[i];k++){
00215     // finding x[i][k]
00216     if (k==0)
00217       f = 0;
00218     else
00219       f = 1 + x[i][k-1];
00220     g = np-1;
00221     while (g > f){
00222       v = (f+g)/2;
00223       if (binomial(U.sum(1+v) , a[i] - k) <= tN)
00224         g = v;
00225       else
00226         f = v + 1;
00227     }
00228     x[i].push_back(f); // decoded the kth connection of vertex i
```

```
00229     y = binomial(U.sum(1+x[i][k]), a[i] - k);
00230     tN = (tN - y) / beta[x[i][k]];
00231     Ni += li * y;
00232     li *= beta[x[i][k]];
00233     beta[x[i][k]] --;
00234     U.add(x[i][k], -1);
00235   }
00236   return pair<mpz_class, mpz_class>(Ni, li);
00237 }
```

6.2.3.4 init()

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

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

```
00197 {
00198   x.clear();
00199   x.resize(n);
00200   beta = b;
00201   U = reverse_fenwick_tree(b);
00202   W = reverse_fenwick_tree(a);
00203
00204 }
```

6.2.4 Member Data Documentation

6.2.4.1 a

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

left degree sequence

6.2.4.2 b

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

right degree sequence

6.2.4.3 beta

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

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

6.2.4.4 n

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

number of left vertices

6.2.4.5 np

`int b_graph_decoder::np [private]`

number of right vertices

6.2.4.6 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 $U_v = \sum_{k=v}^{n'-1} b_k(i)$

6.2.4.7 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 $W_v = \sum_{k=v}^{n-1} a_k$

6.2.4.8 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

6.3 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 (const b_graph &G)

    computes $N(G)$
- 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] $= \sum_{k=v}^{n} b_k(i)$.

6.3.1 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.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 b_graph_encoder()

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

constructor

```
00046 : a(a_), b(b_) {}
```

6.3.3 Member Function Documentation

6.3.3.1 compute_N()

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

computes $N(G)$

Parameters

G	reference to the bipartite graph for which we compute N

Returns

A pair, where the first component is $N(G)$, and the second component is $l(G)$

```
00026                                                                           {
00027    //logger::item_start("bip init");
00028    int n_l = G.nu_left_vertices(); // number of left vertices
00029    int n_bits = 0;
00030    int n_copy = n_l;
00031    while (n_copy > 0){
00032      n_bits ++;
00033      n_copy »= 1;
00034    }
00035    n_bits += 2;
00036
00037    vector<pair<int, int> > call_stack(2 * n_bits);
00038    vector<pair<mpz_class, mpz_class> > return_stack(2 * n_bits); // first = N, second = l
00039    vector<mpz_class> r_stack(2 * n_bits); // stack of r values
00040
00041    vector<int> status_stack(2 * n_bits);
00042    //vector<int> St_stack(2 * n_bits); // stack to store values of St
00043
00044    call_stack[0] = pair<int, int> (0,n_l-1); // i j
00045    status_stack[0] = 0; // newly added
00046
00047    int call_size = 1; // the size of the call stack
00048    int return_size = 0; // the size of the return stack
00049
00050    int i, j, t, Sj;
00051    int status;
00052
00053    vector<int> gamma; // forward list  of the graph
00054
00055    mpz_class rtj, prod_afac, Nit_rtj, lit_Ntj, bin;
00056    //logger::item_stop("bip init");
00057    while(call_size > 0){
00058      //cerr « " call_size " « call_size « endl;
00059      i = call_stack[call_size-1].first;
00060      j = call_stack[call_size-1].second;
00061      if (i==j){
00062        //logger::item_start("bip enc i = j");
00063        return_stack[return_size].first = 0; // N_{i,j} is initialized with 0
00064        return_stack[return_size].second = 1; // l_{i,j} is initialized with 1
00065        r_stack[return_size] = binomial(U.sum(0), a[i]); // r_i = \binom{S_i}{a_i}, s_i = U.sum(0)
00066        gamma = G.get_adj_list(i);
00067        for (int k=0;k<a[i];k++){
00068          //logger::item_start("bip enc i = j binomial");
00069          bin = binomial(U.sum(1+gamma[k]), a[i] - k);
00070          //logger::item_stop("bip enc i = j binomial");
00071
00072          //logger::item_start("bip enc i = j arithmetic");
00073          return_stack[return_size].first += return_stack[return_size].second * bin;
00074          return_stack[return_size].second *= beta[gamma[k]];
00075          //logger::item_start("bip enc i = j arithmetic");
00076          beta[gamma[k]] --;
00077          U.add(gamma[k],-1);
00078        }
00079        return_size ++;
00080        call_size --;
00081        //logger::item_stop("bip enc i = j");
00082      }else{
00083        //logger::item_start("bip enc i neq j");
00084        t = (i+j)/2;
00085        status = status_stack[call_size - 1];
00086        //logger::item_start("bip enc stacking 0 1");
00087        if (status == 0){
00088          // newly added, left node must be called
00089          call_stack[call_size].first = i;
00090          call_stack[call_size].second = t;
00091          status_stack[call_size-1] = 1; // left is called
00092          status_stack[call_size] = 0; // newly added
00093          call_size++;
00094        }
00095        if (status == 1){
00096          // left is returned
00097          //St_stack[call_size-1] = U.sum(0);
00098          // call the right child
00099          call_stack[call_size].first = t+1;
```

```
00100            call_stack[call_size].second = j;
00101            status_stack[call_size-1] = 2; //right is called
00102            status_stack[call_size] = 0;  // newly called
00103            call_size ++;
00104        }
00105        //logger::item_stop("bip enc stacking 0 1");
00106        if (status == 2){
00107            //Sj = U.sum(0);
00108            //logger::item_start("bip enc i neq j prod_factorial");
00109            //prod_afac = prod_factorial(a, t+1, j); // the product of a_k! for t + 1 <= k <= j
00110            //logger::item_stop("bip enc i neq j prod_factorial");
00111
00112            //logger::item_start("bip enc i neq j compute_product");
00113            //rtj = compute_product(St_stack[call_size-1], St_stack[call_size-1] - Sj, 1) / prod_afac;
00114            //logger::item_stop("bip enc i neq j compute_product");
00115
00116            //logger::item_start("bip enc i neq j arithmetic");
00117            Nit_rtj = return_stack[return_size-2].first * r_stack[return_size-1];
00118            lit_Ntj = return_stack[return_size-2].second * return_stack[return_size-1].first;
00119            return_stack[return_size-2].first = Nit_rtj + lit_Ntj; // Nij
00120            return_stack[return_size-2].second = return_stack[return_size-2].second *
          return_stack[return_size-1].second; // lij
00121            r_stack[return_size - 2] = r_stack[return_size-2] * r_stack[return_size-1];
00122            //logger::item_stop("bip enc i neq j arithmetic");
00123            return_size --; // pop 2 add 1
00124            call_size --;
00125        }
00126        //logger::item_stop("bip enc i neq j");
00127    }
00128
00129  }
00130  if (return_size != 1){
00131    cerr << " error: bip compute_N return_size is not 1 it is " << return_size << endl;
00132  }
00133  return return_stack[0];
00134 }
```

6.3.3.2 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

```
00138 {
00139  if (a != G.get_left_degree_sequence() or b != G.get_right_degree_sequence())
00140    cerr << " WARNING b_graph_encoder::encoder : vectors a and/or b do not match with the degree
      sequences of the given bipartite graph  " << endl;
00141
00142  //init(G); // initialize U and beta for G
00143  //pair<mpz_class, mpz_class> ans = compute_N(0,G.nu_left_vertices()-1, G);
00144  //init(G);
00145  //logger::item_start("bip enc compute N");
00146  //pair<mpz_class, mpz_class> ans = compute_N_new(G);
00147  //logger::item_stop("bip enc compute N");
00148
00149  init(G);
00150  //logger::item_start("bip enc compute N new r");
00151  pair<mpz_class, mpz_class> ans = compute_N(G);
00152  //logger::item_stop("bip enc compute N new r");
00153  // if (ans.first == ans2.first and ans.second == ans2.second){
00154  //    cout << " = " << endl;
00155  // }else{
00156  //    cout << " != " << endl;
00157  // }
00158  //if (ans.first!= ans_2.first or ans.second != ans_2.second){
00159  //  cerr << " bip ans != ans_2 ans = (" << ans.first << " , " << ans.second <<  " ) ans_2 = (" <<
      ans_2.first << " , " << ans_2.second << " ) "  << endl;
00160  //}//else{
00161  //cerr << " the same! ans = (" << ans.first << " , " << ans.second <<  " ) ans_2 = (" << ans_2.first << " ,
      " << ans_2.second << " ) "  << endl;
00162  //}
00163  //mpz_class prod_b_factorial = prod_factorial(b, 0, b.size()-1); // \prod_{i=0}^{n-1} b_i
00164
00165  //if (prod_b_factorial != ans.second)
00166  //  cerr << "EEEEEEEEEEEEEEEEEEEEEEEEEE prod_b_factorial != ans.second" << endl;
00167
00168  bool ceil = false;
00169  //logger::item_start("bip enc ceil");
00170  if (ans.first % ans.second != 0)
00171    ceil = true;
00172  //logger::item_stop("bip enc ceil");
```

```
00173
00174   //logger::item_start("bip enc final div");
00175   ans.first /= ans.second;
00176   //logger::item_stop("bip enc final div");
00177   if (ceil)
00178     ans.first ++;
00179   return ans.first;
00180 }
```

6.3.3.3 init()

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

initializes beta and U

```
00009 {
00010   // initializing beta
00011   beta = G.get_right_degree_sequence();
00012
00013   // initializing the Fenwick tree
00014   U = reverse_fenwick_tree(beta);
00015
00016   if (a != G.get_left_degree_sequence() or b != G.get_right_degree_sequence())
00017     cerr « " WARNING b_graph_encoder::init : vectors a and/or b do not match with the degree sequences
    of the given bipartite graph  " « endl;
00018
00019 }
```

6.3.4 Member Data Documentation

6.3.4.1 a

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

the degree sequence for the left nodes

6.3.4.2 b

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

the degree sequence for the right nodes

6.3.4.3 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] = $b_v(i)$

6.3.4.4 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

6.4 bit_pipe Class Reference

A sequence of arbitrary number of bits.

```
#include <bitstream.h>
```

Public Member Functions

- bit_pipe ()
- bit_pipe (const unsigned int &n)

    constructor given an integer
- bit_pipe (const mpz_class &n)

    constructor given an mpz_class object
- void shift_right (int n)

    shifts n bits to the right.
- void shift_left (int n)

    shift everything n bits to the left
- int size () const

    return the number of chunks
- int residue () const

    returns the number of residual bits in the last chunk
- const vector< unsigned int > & chunks () const

    returns const reference to the bit sequence (object bits)
- void append_left (const bit_pipe &B)

    append B to the left of me
- unsigned int & operator[ ] (int n)

    returns a reference to the nth chunk
- const unsigned int & operator[ ] (int n) const

    returns a (const) reference to the nth chunk (const)

Private Attributes

- vector< unsigned int > bits

    a vector of chunks, each of size 4 bytes. This represents an arbitrary sequence of bits
- int last_bits

    the number of bits in the last chunk (the last chunk starts from MSB, so the BIT_INT - last_bits many bits to the right (LSB) are empty and should be zero)

Friends

- class obitstream
- class ibitstream
- ostream & operator<< (ostream &o, const bit_pipe &B)

    write to the output
- bit_pipe operator<< (const bit_pipe &B, int n)

    shifts bits in B n bits to the left
- bit_pipe operator>> (const bit_pipe &B, int n)

    shifts bits in B n bits to the right

6.4.1 Detailed Description

A sequence of arbitrary number of bits.

The bit_pipe class implements an arbitrary sequence of bits. This is useful for example when we want to use Elias delta code to write some integer to the output. This can lead to storage efficiencies, since in such cases we will need to work with incomplete bytes.

The bits vector stores an array of chunks, each having 4 bytes (32 bits). For instance the sequence of bits $<11001100110011>$ is stored as a single chunk $<11001100110011|000000000000000000>$ of size 32 where the | sign shows that the remaining zeros are residuals (not part of data). This is stored as the last_bits variable. In this example, last_bits is 14 because there are 14 bits of data in the last chunk.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 bit_pipe() [1/3]

```
bit_pipe::bit_pipe ( )  [inline]
00028 {bits.resize(0); last_bits = 0;}
```

6.4.2.2 bit_pipe() [2/3]

```
bit_pipe::bit_pipe (
            const unsigned int & n )
```

constructor given an integer

Some examples: n = 1, bits = 1|0000000 (followed by 3 zero bytes) n = 12, bits = 1100|0000 (followed by 3 zero bytes) n = 255633, bits = 11111001 10100100 01|000000 (followed by a zero byte)

```
00010                                          {
00011   bits.resize(1);
00012   bits[0] = n;
00013   last_bits = nu_bits(n);
00014   bits[0] «= BIT_INT - last_bits; // so that n appears in the MSB place
00015 }
```

6.4.2.3 bit_pipe() [3/3]

```
bit_pipe::bit_pipe (
            const mpz_class & n )
```

constructor given an mpz_class object

```
00017                                          {
00018   size_t n_bits = mpz_sizeinbase(n.get_mpz_t(), 2);
00019   size_t size = n_bits / BIT_INT + 1; // how many unsigned int chunks we need
00020   bits.resize(size);
00021   mpz_export(&bits[0],
00022              &size,
00023              1, // order can be 1 for most significant word first or -1 for least significant first
00024              BYTE_INT, // size: each word will be size bytes and
00025              0, // Within each word endian can be 1 for most significant byte first, -1 for least
    significant first, or 0 for the native endianness of the host CPU.
00026              0, // The most significant nails bits of each word are unused and set to zero, this can
    be 0 to produce full words.
00027              n.get_mpz_t());
00028   bits.resize(size);
00029   last_bits = BIT_INT; // at the moment LSB of n is the LSB bit of the rightmost chunk
00030   // but we need the MSB of n to be the MSB of the leftmost chunk
00031   // in order to do this, we must shift left
00032   // but how much? it is related to the remainder of bit count in n with respect to BIT_INT
00033   int rem = n_bits % BIT_INT; // the remainder
00034   if (rem != 0){
00035     // if remainder is zero, nothing should be done
00036     // otherwise, shift left BIT_INT - rem bits
00037     shift_left(BIT_INT - rem);
00038   }
00039
00040 }
```

6.4.3 Member Function Documentation

6.4.3.1 append_left()

```
void bit_pipe::append_left (
             const bit_pipe & B )
```

append B to the left of me

Example: if this is $<1100|0000>$ and B is $<11110000\ 1111|0000>$ then this becomes $<11110000\ 11111100>$ (trailing zero bytes not shown in example)

```
00146                                         {
00147    if (B.size() == 0) // nothing should be done, B is empty
00148      return;
00149    int B_res = B.residue(); // number of incomplete bits in B
00150    if (B_res == BIT_INT){
00151      // B has complete chunks, so I just need to insert chunks of B at the beginning of my chunks
00152      bits.insert(bits.begin(), B.chunks().begin(), B.chunks().end());
00153      return; // all set!
00154    }
00155    // B has a residue
00156    // so I need to shift myself to the right and then append
00157    shift_right(B_res);
00158    // then, my leftmost chunk must be combined with the rightmost chunk of B:
00159    bits[0] |= B[B.size()-1];
00160    // then insert all but the rightmost chunk of B at my left
00161    if (B.chunks().size()>1)
00162      bits.insert(bits.begin(), B.chunks().begin(), B.chunks().end()-1);
00163 }
```

6.4.3.2 chunks()

```
const vector< unsigned int > & bit_pipe::chunks ( ) const  [inline]
```

returns const reference to the bit sequence (object bits)

```
00052 {return bits;}
```

6.4.3.3 operator[]() [1/2]

```
unsigned int & bit_pipe::operator[] (
             int n )
```

returns a reference to the nth chunk

```
00179                                         {
00180    if (n < 0 or n >= bits.size()){
00181      cerr << " ERROR: bit_pipe::operator [] called for value out of range " << n << " the range is [0, " <<
      bits.size()-1 << "]" << endl;
00182    }
00183    return bits[n];
00184 }
```

6.4.3.4 operator[]() [2/2]

```
const unsigned int & bit_pipe::operator[] (
             int n ) const
```

returns a (const) reference to the nth chunk (const)

```
00187                                                   {
00188    if (n < 0 or n >= bits.size()){
00189      cerr << " ERROR: bit_pipe::operator [] called for value out of range " << n << " the range is [0, " <<
      bits.size()-1 << "]" << endl;
00190    }
00191    return bits[n];
00192 }
```

6.4.3.5 residue()

```
int bit_pipe::residue ( ) const  [inline]
```

returns the number of residual bits in the last chunk

```
00049 {return last_bits;}
```

6.4.3.6 shift_left()

```
void bit_pipe::shift_left (
              int n )
```

shift everything n bits to the left

```
00076                                  {
00077   if (n < 0){
00078     cerr « " ERROR: bit_pipe::shift_left called for negative value " « n « endl;
00079     return;
00080   }
00081   if (n >= BIT_INT){
00082     // we need to remove a number of bytes
00083     int bytes_to_remove = n / BIT_INT; // these many bytes must be remove
00084     bits.erase(bits.begin(), bits.begin() + bytes_to_remove);
00085     n = n % BIT_INT;
00086   }
00087   if (n == 0)
00088     return;
00089
00090   // when we reach at this line, we have 1 <= n <= 7
00091   unsigned int mask = mask_gen(n) « (BIT_INT-n); // n bits in MSB for carryover masking
00092   unsigned int carry; // carryover to the left byte
00093   for (int i=0;i<bits.size(); i++){
00094     carry = (mask & bits[i]) » (BIT_INT-n); // bring it to the right
00095     if ( i> 0)
00096       bits[i-1] |= carry; // add carry to the left guy
00097     bits[i] «= n;
00098   }
00099
00100   // now, deal with last_bits
00101   last_bits -= n;
00102   if (last_bits <= 0){
00103     // means that the rightmost byte must vanish
00104     last_bits += BIT_INT;
00105     bits.pop_back(); // remove the last byte
00106   }
00107 }
```

6.4.3.7 shift_right()

```
void bit_pipe::shift_right (
              int n )
```

shifts n bits to the right.

Example: if bits is 11111111 11111000 and n = 5, then bits becomes 00000111 11111111 11000000 (trailing zero bytes are not shown in this example)

```
00045                                  {
00046   if (n == 0)
00047     return; // nothing to do
00048   if (n >= BIT_INT){
00049     bits.insert(bits.begin(), n / BIT_INT, 0); // n/BIT_INT bytes each zero will be added
00050     shift_right(n%BIT_INT);
00051     return;
00052   }
00053   // when we arrive at this line, n must be strictly less than BIT_INT and strictly bigger than zero,
       i.e. 0 < n < BIT_INT
00054   unsigned int mask = mask_gen(n); // mask is going to be n many ones (in LSB), e.g. if n = 3, mask is
       00000111, this is useful in carrying over LSB of left bytes to the right bytes
00055   unsigned int carry_current = 0; // carry over of left bytes to the right. For instance, if we want
       to shift 11111111 3 bits to the right, it becomes 00011111 but a carry over 111 must be added to the
       byte to the right. This is initially zero
00056   unsigned int carry_prev = 0; // the same concept, but for the previous byte (to the left of me).
```

```
00057    for (int i=0;i<bits.size();i++){
00058      carry_current = bits[i] & mask; // find carryover bits for current byte
00059      bits[i] »= n; // shift the current byte
00060      carry_prev «= (BIT_INT-n); // put the previous carryover bits in place to be added to the current
         byte
00061      bits[i] |= carry_prev; // add the carryover to the current byte
00062      carry_prev = carry_current; // the current byte is the previous byte for the next byte
00063    }
00064
00065    if (n > (BIT_INT - last_bits)){
00066      // the LSB bits of the last chunk must fall into a new chunk, so I should push_back a new chunk,
         which is zero for now
00067      carry_prev «= (BIT_INT-n);
00068      bits.push_back(carry_prev); // the last byte is the last carryover shifted to the left
00069    }
00070    last_bits += n;
00071    if (last_bits > BIT_INT)
00072      last_bits -= BIT_INT;
00073 }
```

6.4.3.8 size()

```
int bit_pipe::size ( ) const  [inline]
```

return the number of chunks
```
00046 {return bits.size();}
```

6.4.4 Friends And Related Symbol Documentation

6.4.4.1 ibitstream

```
friend class ibitstream  [friend]
```

6.4.4.2 obitstream

```
friend class obitstream  [friend]
```

6.4.4.3 operator$<<$ [1/2]

```
bit_pipe operator<< (
            const bit_pipe & B,
            int n )  [friend]
```

shifts bits in B n bits to the left
```
00166                                                    {
00167    bit_pipe ans = B;
00168    ans.shift_left(n);
00169    return ans;
00170 }
```

6.4.4.4 operator$<<$ **[2/2]**

```
ostream & operator<< (
            ostream & o,
            const bit_pipe & B )   [friend]
```

write to the output

```
00112                                                              {
00113   if (B.bits.size()==0){
00114     o << "<>";
00115     return o;
00116   }
00117   o << "<";
00118   for (int i=0;i<(B.bits.size()-1); i++){ // the last byte requires special handling
00119     bitset<BIT_INT> b(B.bits[i]);
00120     o << b << " ";
00121   }
00122   unsigned int last_byte = B.bits[B.bits.size()-1];
00123
00124   for (int k=BIT_INT;k>(BIT_INT-B.last_bits);k--){ // starting from MSB bit to LSB for existing bits
00125     if (last_byte & (1<<(k-1)))
00126       o << "1";
00127     else
00128       o << "0";
00129   }
00130   o << "|"; // to show the place of the last bit
00131   for (int k=BIT_INT-B.last_bits; k>=1; k--){
00132     if (last_byte &(1<<(k-1)))
00133       o << "1";
00134     else
00135       o << "0";
00136   }
00137   o << ">";
00138   return o;
00139 }
```

6.4.4.5 operator$>>$

```
bit_pipe operator>> (
            const bit_pipe & B,
            int n )   [friend]
```

shifts bits in B n bits to the right

```
00172                                                              {
00173   bit_pipe ans = B;
00174   ans.shift_right(n);
00175   return ans;
00176 }
```

6.4.5 Member Data Documentation

6.4.5.1 bits

```
vector<unsigned int> bit_pipe::bits   [private]
```

a vector of chunks, each of size 4 bytes. This represents an arbitrary sequence of bits

6.4.5.2 last_bits

```
int bit_pipe::last_bits   [private]
```

the number of bits in the last chunk (the last chunk starts from MSB, so the BIT_INT - last_bits many bits to the right (LSB) are empty and should be zero)

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

- bitstream.h
- bitstream.cpp

6.5 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
- colored_graph ()

  default constructor
- void init (const marked_graph &G)

  initializes other variables. Here, G is the reference to the marked graph based on which this object is being created

Public Attributes

- 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< vector< int > > index_in_neighbor

  index_in_neighbor[v][i] is the index of vertex v in the adjacency list of the ith neighbor of v
- vector< map< pair< int, int >, int > > deg

  deg[v] for a vertex v is a map, where deg[v][(m, m')] for a pair of non star types m, m' is the number of edges connected to v with type m towards v and type m' towards the other endpoint. Note that only non star types appear in this map.
- vector< vector< int > > ver_type

  for a vertex v, ver_type[v] is a vector<int> and encodes the mark of v and its colored degree in the following way: ver_type[v][0] is the ver_mark of v, ver_type[v][3k+1], ver_type[v][3k+2] and ver_type[3k+3] are m, m' and $n_{m,m'}$, where m and m' are edge types, and $n_{m,m'}$ denotes the number of edges connected to v with type (m, m'). The list of m, m' is sorted (lexicographically) to ensure unique representation. Since we only represent types with nonzero $n_{m,m'}$, we are effectively giving the nonzero entries of the colored degree matrix, resulting in an improvement over storing the whole degree matrix.
- 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.
- 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]]
- vector< bool > is_star_vertex

  for $0 \leq v < n$, is_star_vertex[v] is true if vertex v has at least one star typed edge connected to it
- vector< int > star_vertices

  the (sorted) list of star_vertices, where a star vertex is the one which has at least one star type vertex connected to it.

6.5.1 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:

- 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);
```

6.5.2 Constructor & Destructor Documentation

6.5.2.1 colored_graph() [1/2]

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

constructor from a graph, depth and maximum degree parameters
```
00148                                                         : M(graph, depth, max_degree),
     h(depth), Delta(max_degree)
00149   {
00150     init(graph); // initialize other variables
00151   }
```

6.5.2.2 colored_graph() [2/2]

```
colored_graph::colored_graph ( )  [inline]
```

default constructor
```
00154 {}
```

6.5.3 Member Function Documentation

6.5.3.1 init()

```
void colored_graph::init (
            const marked_graph & G )
```

initializes other variables. Here, G is the reference to the marked graph based on which this object is being created

- 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

```
00531 {
00532    logger::add_entry("colored_graph::init init", "");
00533    nu_vertices = G.nu_vertices;
00534    //adj_location = G.adj_location; // neighborhood structure is the same as the given graph
00535    index_in_neighbor = G.index_in_neighbor;
00536
00537    // assigning edge colors based on the messages given by M
00538    //M.update_messages();
00539    adj_list.resize(nu_vertices);
00540
00541    // updating adj_list
00542    logger::add_entry("updating adj_list", "");
00543    int w, my_location, color_v, color_w;
00544    for (int v=0;v<nu_vertices;v++){
00545        adj_list[v].resize(G.adj_list[v].size()); // the same number of neighbors here
00546        for (int i=0;i<G.adj_list[v].size();i++){
00547            w = G.adj_list[v][i].first; // the ith neighbor, the same as in G
00548            //my_location = G.adj_location[w].at(v); // where v stands among the neighbors of w
00549            my_location = index_in_neighbor[v][i];
00550            color_v = M.messages[v][i]; // the color towards v corresponds to the message v sends to w
00551            color_w = M.messages[w][my_location]; // the color towards w is the message w sends towards v
00552            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
00553        }
00554    }
00555
00556    // updating the vertex type sequence, dictionary and list, i.e. variables ver_type, ver_type_dict
      and ver_type_list
00557    // we also update ver_type_int
00558
00559    // implement and update deg and type_vertex_list
00560
00561    int m, mp; // pair of types
00562
00563    logger::add_entry("Find deg and ver_types", "");
00564    deg.resize(nu_vertices);
00565    is_star_vertex.resize(nu_vertices);
00566    ver_type.resize(nu_vertices);
00567    ver_type_int.resize(nu_vertices);
00568
00569    vector<int> vt; // type of v
00570
00571    for (int v=0;v<nu_vertices;v++){
00572        is_star_vertex[v] = false; // it is false unless we figure out otherwise, see below
00573        for (int i=0;i<adj_list[v].size(); i++){
00574            m = adj_list[v][i].second.first;
00575            mp = adj_list[v][i].second.second;
00576            if (M.is_star_message[m] == false and M.is_star_message[mp] == false){
00577                // this edge is not star type
00578                if (deg[v].find(pair<int, int>(m, mp)) == deg[v].end()){
00579                    // this does not exist, so create it, since this is the first edge, its value must be 1
00580                    deg[v][pair<int, int>(m, mp)] = 1;
00581                    //type_vertex_list[pair<int, int>(m, mp)].push_back(v); // this must be done when we see the
      type (m, mp) for the first time here, so as to avoid multiple placing of v in the list
00582                }else{
00583                    // the edge exists, we only need to increase it by one
00584                    deg[v][pair<int, int>(m, mp)] ++;
00585                }
00586            }else{
00587                // this is a star type vertex
00588                is_star_vertex[v] = true;
```

```
00589         }
00590       }
00591
00592     // check if it was star vertex
00593     if (is_star_vertex[v] == true)
00594       star_vertices.push_back(v);
00595
00596
00597     // now, we form the type of this vertex
00598     // the type of a vertex is a vector x as follows:
00599     // x[0] is the vertex mark of v
00600     // x[3k+1], x[3k+2], x[3k+3] = (m_k, mp_k, deg[v][(m_k, mp_k)]) where (m_k, mp_k) is the kt key
    present in the map deg[v]. Since deg[v] is a map, we read its elements in increasing order
    (lexicographic order for pairs (m, mp)), hence this list is on a 1-1 correspondence with the pair
    (\theta(v), D(v)) in the paper.
00601
00602     vt.resize(1+3 * deg[v].size()); // motivated by the above explanation
00603     vt[0] = G.ver_mark[v]; // mark of v
00604     int k = 0; // current index of vt
00605     for (map<pair<int, int>, int>::iterator it = deg[v].begin(); it != deg[v].end(); it++){
00606       vt[++k] = it->first.first; // m
00607       vt[++k] = it->first.second; // mp
00608       vt[++k] = it->second;
00609
00610     }
00611
00612     // cerr << " vt ";
00613     // for (int AA=0; AA<vt.size(); AA++)
00614     //   cerr << vt[AA] << " ";
00615     // cerr << endl;
00616     ver_type[v] = vt;
00617     // find ver_type_int[v]
00618     if (ver_type_dict.find(vt) == ver_type_dict.end()){
00619       // this is a new type, so add it to the dictionary and the list
00620       ver_type_dict[vt] = ver_type_list.size();
00621       ver_type_list.push_back(vt);
00622     }
00623     ver_type_int[v] = ver_type_dict[vt];
00624   }
00625
00626   // editing vertex types so that the vertex type list is sorted lexicographically
00627
00628   // cerr << " ver_type_list before sorting " << endl;
00629   // for (int K=0; K<ver_type_list.size(); K++){
00630   //   //for (int J=0; J<ver_type_list[K].size(); J++)
00631   //   //   cerr << ver_type_list[K][J] << " ";
00632   //   cerr << ver_type_list[K][0]<< ": ";
00633   //   for (int j=0; j< (ver_type_list[K].size()-1)/3; j++)
00634   //     cerr << "| " << ver_type_list[K][1+3*j] << " " << ver_type_list[K][2+3*j] << " " <<
    ver_type_list[K][3+3*j] << " ";
00635   //   cerr << endl;
00636   // }
00637   // cerr << endl << endl;
00638
00639   map<vector<int>, int >::iterator it;
00640   int counter = 0;
00641   for (it=ver_type_dict.begin(); it!=ver_type_dict.end(); it++){
00642     // sweeping over elements in the dictionary increasingly
00643     it->second = counter;
00644     ver_type_list[counter] = it->first; // edit the ver_type_list as well
00645     counter ++;
00646   }
00647
00648   // edit vertex types given the updated ver_type_dict
00649   for (int v=0; v<nu_vertices;v++)
00650     ver_type_int[v] = ver_type_dict[ver_type[v]];
00651 }
```

6.5.4 Member Data Documentation

6.5.4.1 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)

6.5.4.2 deg

```
vector<map<pair<int, int> , int> > colored_graph::deg
```

deg[v] for a vertex v is a map, where deg[v][(m, m')] for a pair of non star types m, m' is the number of edges connected to v with type m towards v and type m' towards the other endpoint. Note that only non star types appear in this map.

6.5.4.3 Delta

```
int colored_graph::Delta
```

the maximum degree threshold

6.5.4.4 h

```
int colored_graph::h
```

the depth up to which look at edge types

6.5.4.5 index_in_neighbor

```
vector<vector<int> > colored_graph::index_in_neighbor
```

index_in_neighbor[v][i] is the index of vertex v in the adjacency list of the ith neighbor of v

6.5.4.6 is_star_vertex

```
vector<bool> colored_graph::is_star_vertex
```

for $0 \le v < n$, is_star_vertex[v] is true if vertex v has at least one star typed edge connected to it

6.5.4.7 M

```
graph_message colored_graph::M
```

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

6.5.4.8 nu_vertices

```
int colored_graph::nu_vertices
```

the number of vertices in the graph.

6.5.4.9   star_vertices

```
vector<int> colored_graph::star_vertices
```

the (sorted) list of star_vertices, where a star vertex is the one which has at least one star type vertex connected to it.

6.5.4.10   ver_type

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

for a vertex v, ver_type[v] is a vector<int> and encodes the mark of v and its colored degree in the following way: ver_type[v][0] is the ver_mark of v, ver_type[v][3k+1], ver_type[v][3k+2] and ver_type[3k+3] are m, m' and $n_{m,m}$', where m and m' are edge types, and $n_{m,m}$' denotes the number of edges connected to v with type (m, m'). The list of m, m' is sorted (lexicographically) to ensure unique representation. Since we only represent types with nonzero $n_{m,m}$', we are effectively giving the nonzero entries of the colored degree matrix, resulting in an improvement over storing the whole degree matrix.

6.5.4.11   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

6.5.4.12   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]]

6.5.4.13   ver_type_list

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

the list of all distinct vertex types, obtained from the ver_type array.

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

- graph_message.h
- graph_message.cpp

6.6   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

6.6.1 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.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 fenwick_tree() [1/2]

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

default constructor
```
00023     {
00024        sums.resize(0);
00025     }
```

6.6.2.2 fenwick_tree() [2/2]

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

constructor, which takes a vector of values and initializes
```
00009 {
00010   int n = vals.size();
00011   sums.resize(n+1);
00012   // initializes at zero
00013   for (int i=1;i<=n;i++)
00014     sums[i] = 0;
00015   for (int i=0;i<n;i++)
00016     add(i,vals[i]); // add values one by one
00017 }
```

6.6.3 Member Function Documentation

6.6.3.1 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

```
00020 {
00021   k = k +1; // the sums vector is one based while the index k was zero based
00022   while (k < sums.size()){
00023     sums[k] += val;
00024     k += (k & -k);
00025   }
00026 }
```

6.6.3.2 size()

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

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

```
00032   {
00033     return sums.size() - 1;
00034   }
```

6.6.3.3 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

```
00030 {
00031   k = k +1; // the sums vector is one based while the index k was zero based
00032   int sum_computed = 0;
00033   while (k > 0){
00034     sum_computed += sums[k];
00035     k -= (k & -k); // reduce the lsb bit
00036   }
00037   return sum_computed;
00038 }
```

6.6.4 Member Data Documentation

6.6.4.1 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

6.7   graph Class Reference

simple unmarked graph

```
#include <simple_graph.h>
```

Public Member Functions

- graph ()

    default constructor
- graph (const vector< vector< int > > &list, const vector< int > &deg)

    a constructor
- graph (const vector< vector< int > > &list)

    constructor, given only the forward adjacency list
- 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

6.7.1   Detailed Description

simple unmarked graph

6.7.2 Constructor & Destructor Documentation

6.7.2.1 graph() [1/3]

```
graph::graph ( ) [inline]
```

default constructor
```
00016 : n(0) {}
```

6.7.2.2 graph() [2/3]

```
graph::graph (
              const vector< vector< int > > & list,
              const 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).

```
00009                                                                       {
00010   n = list.size();
00011   forward_adj_list = list;
00012   // sorting the list
00013   for (int v=0; v<n; v++)
00014     sort(forward_adj_list[v].begin(), forward_adj_list[v].end());
00015   degree_sequence = deg;
00016 }
```

6.7.2.3 graph() [3/3]

```
graph::graph (
              const vector< vector< int > > & list )
```

constructor, given only the forward adjacency list

This constructor only takes the forward adjacency list and computes the degree sequence itself
```
00022 {
00023   n = list.size();
00024   forward_adj_list = list;
00025
00026
00027   // sorting the list
00028   for (int v=0; v<n; v++)
00029     sort(forward_adj_list[v].begin(), forward_adj_list[v].end());
00030
00031   // finding the degree sequence
00032   // first, removing and resize it
00033   degree_sequence.clear();
00034   degree_sequence.resize(n);
00035
00036   for (int v=0; v<n; v++){
00037     degree_sequence[v] += forward_adj_list[v].size(); // degree to the right
00038     for (int i=0;i<forward_adj_list[v].size();i++) // modifying degree of vertices to the right of v
00039       degree_sequence[forward_adj_list[v][i]]++;
00040   }
00041 }
```

6.7.3 Member Function Documentation

6.7.3.1 get_degree()

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

returns the overall degree of a vertex
```
00055                                          {
00056   return degree_sequence[v];
00057 }
```

6.7.3.2 get_degree_sequence()

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

returns the whole degree sequence
```
00059                                          {
00060   return degree_sequence;
00061 }
```

6.7.3.3 get_forward_degree()

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

returns the forward degree of a vertex v
```
00049                                          {
00050   if (v < 0 or v >= n)
00051     cerr « "graph::get_forward_degree, index v out of range" « endl;
00052   return forward_adj_list[v].size();
00053 }
```

6.7.3.4 get_forward_list()

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

returns the forward adjacency list of a given vertex
```
00043                                          {
00044   if (v < 0 or v >= n)
00045     cerr « "graph::get_forward_list, index v out of range" « endl;
00046   return forward_adj_list[v];
00047 }
```

6.7.3.5 nu_vertices()

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

the number of vertices in the graph
```
00064                                          {
00065   return n;
00066 }
```

6.7.4 Friends And Related Symbol Documentation

6.7.4.1 operator"!=

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

comparing for inequality
```
00102 {
00103   return !(G1 == G2);
00104 }
```

6.7.4.2 operator<<

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

printing the graph to the output
```
00069 {
00070   int n = G.nu_vertices();
00071   vector<int> list;
00072   for (int i=0;i<n;i++){
00073     list = G.get_forward_list(i);
00074     o « i « " -> ";
00075     for (int j=0;j<list.size();j++){
00076       o « list[j];
00077       if (j < list.size()-1)
00078         o « ", ";
00079     }
00080     o « endl;
00081   }
00082   return o;
00083 }
```

6.7.4.3 operator==

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

comparing two graphs for equality
```
00086 {
00087   int n1 = G1.nu_vertices();
00088   int n2 = G2.nu_vertices();
00089   if (n1!= n2)
00090     return false;
00091   vector<int> list1, list2;
00092   for (int v=0; v<n1; v++){
00093     list1 = G1.get_forward_list(v);
00094     list2 = G2.get_forward_list(v);
00095     if (list1 != list2)
00096       return false;
00097   }
00098   return true;
00099 }
```

6.7.5 Member Data Documentation

6.7.5.1 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).

6.7.5.2 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.

6.7.5.3 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

6.8 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
- void init ()

    initializes x to be empty vector of size n, and U and beta by a
- 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

    $\lfloor \log_2 n \rfloor^2$
- 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 $\log^2 n$.

6.8.1 Detailed Description

Decodes a simple unmarked graph.

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

```
vector<int> a = {3,2,2,3};
b_graph_decoder D(a);
```

Then, if variable f of type `pair<mpz_class, vector<int> >` is obtained from a graph_encoder class, we can reconstruct the graph using f:

```
graph Ghat = D.decode(f.first, f.second);
```

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 = {3,2,2,3}; // degree sequence

graph G({1,2,3},{3},{3},{}); // defining the graph

graph_encoder E(a); // constructing the encoder object
pair<mpz_class, vector<int> > f = E.encode(G);

graph_decoder D(a);
graph Ghat = D.decode(f.first, f.second);

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

6.8.2 Constructor & Destructor Documentation

6.8.2.1 graph_decoder()

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

constructor given the degree sequence

```
00224 {
00225   a = a_;
00226   n = a.size();
00227
00228   int log2n = 0; // log of n in base 2
00229   int nn = n; // a copy of n
00230   while (nn>0){
00231     log2n ++;
00232     nn = nn » 1; // divide by 2
00233   } // eventually, we count the number of bits in n
00234   log2n --; // we count extra, e.g. when n = 1, we end up having 1, rather than 0
00235   logn2 = log2n * log2n;
00236
00237   init(); // init x, beta and U
00238 }
```

6.8.3 Member Function Documentation

6.8.3.1 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

```
00249 {
00250   init(); // make x, U and beta ready for decoding
00251   tS = tS_;
00252   //mpz_class prod_a_factorial = 1; // \prod_{i=1}^n a_i!
00253   //for (int i=0; i<a.size();i++)
00254   //   prod_a_factorial *= compute_product(a[i], a[i], 1);
00255
00256   mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=0}^{n-1} a_i!
00257   mpz_class tN = f * prod_a_factorial;
00258   decode_interval(0,n-1,1,tN,0);
00259   return graph(x, a);
00260 }
```

6.8.3.2 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 $l_{i,j} = l_{i,j}(G)$

```
00315 {
00316   //cerr « " decode interval " « i « " " « j « " tN " « tN « endl;
00317   if (i == j)
00318     return decode_node(i, tN);
00319
00320   // sweeping for zero nodes
00321
00322   int t; // place to break
00323   int St; // S_{t+1}
00324   if ((j-i) > logn2){
00325     //cerr « " long interval I = " « I « endl;
00326     t = (i+j) / 2; // break at middle, since we have \tilde{S}
00327     St = tS[I]; // looking at the \f$\tilde{S}\f$ vector
00328   }else{
00329     //cerr « " short interval " « endl;
00330     t = i;
00331     St = U.sum(i) - 2 * beta[i];
00332   }
00333
00334   //cerr « " decode interval " « i « " " « j « " t " « t « " St " « St « " Sj " « Sj « endl;
00335   mpz_class rtj; // \f$t_{t+1, j}\f$
00336   mpz_class tNit; // \f$\tilde{N}_{i,t}\f$ for the left decoder
00337   mpz_class tNtj; // \f$\tilde{N}_{t+1, j}\f$ for the right decoder
00338   mpz_class Nit; // the true N_{i,t} returned by the left decoder
00339   mpz_class lit; // the true l_{i,t} returned by the left decoder
00340   mpz_class Ntj; // the true N_{t+1, j} returned by the right decoder
00341   mpz_class ltj; // the true l_{t+1,j} returned by the right decoder
00342   mpz_class Nij; // the true N_{i,j} to return
00343   mpz_class lij; // the true l_{i,j} to return
00344
00345   pair<mpz_class, mpz_class> ans; // returned by subintervals
00346
00347
00348   rtj = compute_product(St - 1, (St - Sj)/2, 2);
00349   //cerr « " interval " « i « " " « j « " t " « t « " St " « St « " rtj " « rtj « endl;
00350   tNit = tN / rtj;
00351
00352   // calling the left decoder
00353   ans = decode_interval(i,t,2*I,tNit, St);
00354   Nit = ans.first;
00355   lit = ans.second;
00356
00357   // reducing the contribution of the left decoder to prepare for the right decoder
00358   tNtj = (tN - Nit * rtj) / lit;
00359
00360   // calling the right decoder
00361   ans = decode_interval(t+1, j, 2*I + 1, tNtj, Sj);
00362   Ntj = ans.first;
00363   ltj = ans.second;
00364
00365   // preparing Nij and lij to return
```

```
00366   Nij = Nit * rtj + lit * Ntj;
00367   lij = lit * ltj;
00368   return pair<mpz_class, mpz_class> (Nij, lij);
00369 }
```

6.8.3.3 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 $N_{i,i} = N_{i,i}(G)$

```
00263 {
00264   //cerr « " decode node " « i « " tN " « tN « endl;
00265   //cerr « " beta[i] " « beta[i] « endl;
00266   //cerr « " beta " « endl;
00267   //for (int k = i; k< n;k++)
00268   //  cerr « k « " " « beta[k] « endl;
00269   //cerr « " U " « endl;
00270   //for (int k=i;k<n;k++)
00271   //  cerr « k « " " « U.sum(k) « endl;
00272
00273   if (beta[i] == 0)
00274     return pair<mpz_class, mpz_class> (0,1);
00275
00276   mpz_class li = 1; // l_i(G)
00277   mpz_class Ni = 0; // N_{i,i}(G)
00278   int f, g; // endpoints for the binary search
00279   int t; // midpoint for the binary search
00280   mpz_class zik, lik;
00281   for (int k=0;k<beta[i];k++){
00282     if (k==0)
00283       f = i+1;
00284     else
00285       f = x[i][k-1]+1;
00286     g = n-1;
00287     while(g > f){
00288       //cerr « " f , g " « f « " " « g « endl;
00289       t = (f+g)/2;
00290       // binary search:
00291       if(compute_product(U.sum(t+1), beta[i] - k, 1) <= tN)
00292         g = t;
00293       else
00294         f = t+1;
00295     }
00296     x[i].push_back(f);
00297     zik = compute_product(U.sum(x[i][k]+1), beta[i] - k, 1);
00298     Ni += li * zik;
00299     lik = (beta[i] - k) * beta[x[i][k]];
00300     li *= lik;
00301     tN -= zik;
00302     tN /= lik;
00303     U.add(x[i][k],-1);
00304     beta[x[i][k]] --;
00305   }
00306   //cerr « " decoded for " « i « " x: " « endl;
00307   //for (int j=0;j<x[i].size(); j++)
00308   //  cerr « x[i][j] « " " ;
00309   //cerr « endl;
00310   return pair<mpz_class, mpz_class> (Ni, li);
00311 }
```

6.8.3.4 init()

`void` graph_decoder::init ( )

initializes x to be empty vector of size n, and U and beta by a

```
00241 {
00242   x.clear();
00243   x.resize(n);
00244   beta = a;
00245   U = reverse_fenwick_tree(a);
00246 }
```

6.8.4 Member Data Documentation

6.8.4.1 a

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

the degree sequence of the graph.

6.8.4.2 beta

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

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

6.8.4.3 logn2

`int` graph_decoder::logn2 [private]

$\lfloor \log_2 n \rfloor^2$

6.8.4.4 n

`int` graph_decoder::n [private]

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

6.8.4.5 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 $\log^2 n$.

6.8.4.6 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 $U_v = \sum_{k=v}^{n-1} d_k(i)$.

6.8.4.7 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

6.9 graph_encoder Class Reference

Encodes a simple unmarked graph.

`#include <simple_graph_compression.h>`

Public Member Functions

- graph_encoder (const vector< int > &a_)

  constructor
- void init (const graph &G)

  initializes beta and U, clears Stilde for a fresh use
- pair< mpz_class, mpz_class > compute_N (const graph &G)

  computes $N(G)$
- pair< mpz_class, vector< int > > encode (const graph &G)

  Encodes the graph and returns N together with Stilde.

Private Attributes

- int n

  the number of vertices
- vector< int > a

  the degree sequence
- 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

  $\lfloor \log_2 n \rfloor^2$ where n is the number of vertices

6.9.1 Detailed Description

Encodes a simple unmarked graph.

Encodes a simple graph in the set of graphs with a given degree sequence a. Therefore, to construct an encoder object, we need to specify this degree sequence as a vector of int. For instance (in c++11)

```
vector<int> a = {3,2,2,3};
graph_encoder E(a);
```

constructs an encode object E which is capable of encoding graphs having 4 nodes with degrees 3, 2, 2, 3 (in order). Hence, assume that we have defined such a graph by giving forward adjacency list:

```
graph G({1,2,3},{3},{3},{});
```

Note that G has a degree sequences which is equal to a. Then, we can use E to encode G as follows:

```
pair<mpz_class, vector<int> > f = E.encode(G);
```

In this way, the encode converts G to a pair stored in f, where its first part is an integer, and the second part is the array of integers $\tilde{S}$. Later on, we can use f to decode G.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 graph_encoder()

```
graph_encoder::graph_encoder (
            const vector< int > & a_ )
```

constructor

initializes the degree sequence to a_, sets n and logn2, and resizes the Stilde vector

```
00011                                                      {
00012    a = a_;
00013    n = a.size();
00014    int log2n = 0; // log of n in base 2
00015    int nn = n; // a copy of n
00016    while (nn>0){
00017      log2n ++;
00018      nn = nn » 1; // divide by 2
00019    } // eventually, we count the number of bits in n
00020    log2n --; // we count extra, e.g. when n = 1, we end up having 1, rather than 0
00021    logn2 = log2n * log2n;
00022
00023    Stilde.clear();
00024    Stilde.resize(4 * n / logn2); // look at the explanation of the algorithm, before deriving the bound
       16 n, that 4n / \lfloor \log_2 n \rfloor^2 is also an upper bound. After this point, we have used
       \lfloor \log_2 n \rfloor \geq \log n / 2 to derive the 16 n bound.
00025    Stilde[0] = 0;
00026 }
```

6.9.3 Member Function Documentation

6.9.3.1 compute_N()

```
pair< mpz_class, mpz_class > graph_encoder::compute_N (
            const graph & G )
```

computes $N(G)$

Parameters

G	the reference to the simple graph G

Returns

A pair, where the first component is $N(G)$ and the second component is $l(G)$.

```
00053 {
00054
00055    int n = G.nu_vertices();
00056    int n_bits = 0;
00057    int n_copy = n;
00058    while (n_copy > 0){
00059      n_bits ++;
00060      n_copy >>= 1;
00061    }
00062    n_bits += 2;
00063
00064    vector<pair<pair<int, int>, int > > call_stack(2 * n_bits);
00065    vector<pair<mpz_class, mpz_class> > return_stack(2 * n_bits); // first = N, second = l
00066    vector<mpz_class> r_stack(2 * n_bits); // stack of r values
00067    vector<int> status_stack(2 * n_bits);
00068    //vector<int> St_stack(2 * n_bits); // stack to store values of St
00069
00070    call_stack[0].first = pair<int, int> (0,n-1); // i j
00071    call_stack[0].second = 1; // I
00072    status_stack[0] = 0; // newly added
00073
00074    int call_size = 1; // the size of the call stack
00075    int return_size = 0; // the size of the return stack
00076
00077    int i, j, I, t, Sj;
00078    int status;
00079
00080    vector<int> gamma; // forward list  of the graph
00081    mpz_class zik, lik, rtj; // intermediate variables
00082    mpz_class Nit_rtj; // result of Nij * rtj
00083    mpz_class lit_Ntj; // result of lit * Ntj
00084    while (call_size > 0){
00085      // cerr << " printing the whole stack " << endl;
00086      // for (int k = 0; k<call_size; k++){
00087      //   cerr << k << " : " << call_stack[k].first.first << " " << call_stack[k].first.second << " I " <<
      call_stack[k].second << "s= " << status_stack[k] << endl;
00088      // }
00089      // cerr << " return stack " << endl;
00090      // for (int k=0;k<return_size;k++)
00091      //   cerr << k << ": " << return_stack[k].first << " " << return_stack[k].second << endl;
00092      i = call_stack[call_size-1].first.first;
00093      j = call_stack[call_size-1].first.second;
00094      I = call_stack[call_size-1].second;
00095      if (i==j){
00096        //logger::item_start("sim enc i = j");
00097        return_stack[return_size].first = 0; // z_i is initialized with 0
00098        return_stack[return_size].second = 1; // l_i is initialize with 1
00099        gamma = G.get_forward_list(i); // the forward adjacency list of vertex i
00100        r_stack[return_size] = compute_product(U.sum(i) - 1, beta[i], 2); // this is r_i
00101        // cerr << " i " << i << " j " << j << " gamma: " << endl;
00102        // for (int k=0;k<gamma.size();k++)
00103        //   cerr << gamma[k] << " ";
00104        // cerr << endl;
00105        // cerr << " beta " << endl;
00106        // for (int k=0; k<n; k++)
00107        //   cerr << beta[k] << " ";
00108        // cerr << endl;
00109
00110        for (int k=0;k<gamma.size();k++){
00111          zik = compute_product(U.sum(1+gamma[k]), beta[i] - k, 1); // we are zero based here, so
      instead of -k + 1, we have -k
00112          //zik = helper_vars::return_stack[0];
00113          //cerr << " zik " << zik << endl;
00114          return_stack[return_size].first += return_stack[return_size].second * zik;
00115          lik = (beta[i] - k) * beta[gamma[k]]; // we are zero based here, so instead of -k + 1, we have
      -k
00116          //cerr << " lik " << lik << endl;
00117          return_stack[return_size].second *= lik;
00118          beta[gamma[k]] -- ;
00119          U.add(gamma[k],-1);
00120        }
00121        return_size ++; // establish the return
00122        call_size --;
00123        //logger::item_stop("sim enc i = j");
00124      }else{
00125        status = status_stack[call_size-1];
00126        if (status == 0){
00127          // newly added node, we should call its left child
00128          t = (i+j) / 2;
00129          call_stack[call_size].first.first = i;
00130          call_stack[call_size].first.second = t;
00131          call_stack[call_size].second = 2*I;
00132          status_stack[call_size-1] = 1; // left is called
00133          status_stack[call_size] = 0; // newly added
```

```
00134          call_size++;
00135       }
00136     if (status == 1){
00137       // left is returned
00138       t = (i+j) / 2;
00139       //St_stack[call_size-1] = U.sum(t+1);
00140       if (j - i > logn2)
00141         Stilde[I] = U.sum(t+1);//St_stack[call_size-1];
00142       // prepare to call right
00143
00144       call_stack[call_size].first.first = t + 1;
00145       call_stack[call_size].first.second = j;
00146       call_stack[call_size].second = 2*I + 1;
00147       status_stack[call_size-1] = 2; // right is called
00148       status_stack[call_size] = 0; // newly called
00149       call_size ++;
00150     }
00151
00152     if (status == 2){
00153       // both are returned, and results can be accessed by the top two elements in return stack
00154       //Sj = U.sum(j+1);
00155       //logger::item_start("sim enc i neq j compute_product");
00156       //rtj = compute_product(St_stack[call_size-1]-1, (St_stack[call_size-1] - Sj)/2, 2);
00157       //logger::item_stop("sim enc i neq j compute_product");
00158       //rtj = helper_vars::return_stack[0];
00159       //cerr « " rtj " « rtj « endl;
00160       //Nij = Nit * rtj + lit * Ntj ;
00161       //logger::item_start("sim enc i neq j arithmetic");
00162       Nit_rtj = return_stack[return_size-2].first * r_stack[return_size-1];
00163       lit_Ntj = return_stack[return_size-2].second * return_stack[return_size-1].first;
00164       return_stack[return_size-2].first = Nit_rtj + lit_Ntj; // Nij
00165       return_stack[return_size-2].second = return_stack[return_size-2].second *
    return_stack[return_size-1].second; // lij
00166       r_stack[return_size-2] = r_stack[return_size-2] * r_stack[return_size-1];
00167       //logger::item_stop("sim enc i neq j arithmetic");
00168       return_size --; // pop 2 add 1
00169       call_size --;
00170     }
00171   }
00172
00173   }
00174
00175   if (return_size != 1){
00176     cerr « " error: return_size is not 1 it is " « return_size « endl;
00177   }
00178   return return_stack[0];
00179 }
```

6.9.3.2 encode()

```
pair< mpz_class, vector< int > > graph_encoder::encode (
              const graph & G )
```

Encodes the graph and returns N together with Stilde.

Parameters

G	reference to the graph to encode

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 $n/\log^2 n$

```
00181                                                                    {
00182   if (G.get_degree_sequence()!= a)
00183     cerr « " WARNING graph_encoder::encode : vector a does not match with the degree sequence of the
    given graph ";
00184   //init(G); // initialize U and beta
00185   //pair<mpz_class, mpz_class> N_ans  = compute_N(0,G.nu_vertices()-1,1, G);
00186   init(G); // re initializing U abd beta for the second test
00187   pair<mpz_class, mpz_class> N_ans = compute_N(G);
00188   // init(G);
```

```
00189   // pair<mpz_class, mpz_class> N_ans_2 = compute_N(G);
00190   // if (N_ans.first == N_ans_2.first and N_ans.second == N_ans_2.second){
00191   //   cerr « " = " « endl;
00192   // }else{
00193   //   cerr « " error N_ans and N_ans_2 are not the same, "  « endl « "N_ans = (" « N_ans.first « " ,
        " « N_ans.second « ") " « endl « "N_ans_2 = (" « N_ans_2.first « " , " « N_ans_2.second « ")" « endl;
00194   // }
00195   //if (N_ans.first != N_ans_2.first or N_ans.second != N_ans_2.second)
00196   //  cerr « " error N_ans and N_ans_2 are not the same, "  « endl « "N_ans = (" « N_ans.first « " , "
        « N_ans.second « ") " « endl « "N_ans_2 = (" « N_ans_2.first « " , " « N_ans_2.second « ")" « endl;
00197   //else
00198   //  cerr « " N_ans = N_ans_2 " « endl;
00199   //mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=1}^n a_i!
00200   //if (prod_a_factorial!= N_ans.second)
00201   //  cerr « " ERROR: not equal " « endl;
00202   // N_ans.second = \prod_{i=1}^n a_i!
00203   // we need the ceiling of the ratio of N_ans.first and prod_a_factorial
00204   bool ceil = false; // if true, we will add one to the integer division
00205   //logger::item_start("simple_ar");
00206   if (N_ans.first % N_ans.second != 0)
00207     ceil = true;
00208   N_ans.first /= N_ans.second;
00209   if (ceil)
00210     N_ans.first ++;
00211   //logger::item_stop("simple_ar");
00212   return pair<mpz_class, vector<int> > (N_ans.first, Stilde);
00213 }
```

6.9.3.3 init()

```
void graph_encoder::init (
             const graph & G )
```

initializes beta and U, clears Stilde for a fresh use

```
00029 {
00030   // initializing the beta sequence
00031   beta = a;
00032
00033   //beta.resize(G.nu_vertices());
00034   //for (int v=0;v<G.nu_vertices();v++)
00035   //  beta[v] = G.get_degree(v);
00036
00037   // initializing the Fenwick Tree
00038   U = reverse_fenwick_tree(beta);
00039
00040
00041   //initializing the partial sum vector Stilde
00042   Stilde.clear();
00043   Stilde.resize(4 * n / logn2); // TO CHECK,
      2018-10-18_self-compression_Stilde-size-required-2nlogn2.pdf
00044   Stilde[0] = 0;
00045 }
```

6.9.4 Member Data Documentation

6.9.4.1 a

vector<int> graph_encoder::a [private]

the degree sequence

6.9.4.2 beta

vector<int> graph_encoder::beta [private]

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

6.9.4.3 logn2

`int graph_encoder::logn2 [private]`

$\lfloor \log_2 n \rfloor^2$ where n is the number of vertices

6.9.4.4 n

`int graph_encoder::n [private]`

the number of vertices

6.9.4.5 Stilde

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

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

6.9.4.6 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 $i \le j$, for $i \le v$, 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

6.10 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
- graph_message ()
    default constructor

Public Attributes

- vector< vector< int > > messages

  messages[v][i] is the integer version of the message from vertex v towards its ith neighbor (in the order given by adj_list of vertex i in graph G). The message is at any given step that update_messages is running, so after finishing update_message, the messages are at step (depth) h-1.

- unordered_map< vector< int >, int, vint_hash > message_dict

  message_dict is the message dictionary at any step that update_messages is running, which maps each message to its corresponding index in the dictionary. When update_messages is over, this corresponds to step (depth) h-1

- vector< int > message_mark

  for an integer message m, message_mark[m] is the mark component associated to the message m at any step that update_messages is working. This is basically the last index in the vector message associate to m. When update_messages is over, this corresponds to step (depth) h-1.

- vector< bool > is_star_message

  for an integer message m, is_star_message[m] is true if m is a star message and false otherwise. Note that m is star type iff the first index in the vector message corresponding to m is -1. This is updated at step of update_messages, so when it is over, it corresponds to step (depth) h-1.

Private Member Functions

- void update_messages (const marked_graph &)

  performs the message passing algorithm and updates the messages array accordingly

- void send_message (const vector< int > &m, int v, int i)

  update message_dict and message_list

Private Attributes

- 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

6.10.1 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. Star type messages (which roughly speaking corresponds to places where there is a vertex in the h neighborhood has degree more than delta) are vectors of size 2, first coordinate being -1, and the second being the edge mark component (towards the 'me' vertex).

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

6.10.2 Constructor & Destructor Documentation

6.10.2.1 graph_message() [1/2]

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

constructor, given reference to a graph

```
00069                                                                          {
00070     h = depth;
00071     Delta = max_degree;
00072     update_messages(graph); // do message passing
00073   }
```

6.10.2.2 graph_message() [2/2]

```
graph_message::graph_message ( )  [inline]
```

default constructor

```
00076 {}
```

6.10.3 Member Function Documentation

6.10.3.1 send_message()

```
void graph_message::send_message (
            const vector< int > & m,
            int v,
            int i )  [inline], [private]
```

update message_dict and message_list

send the message m from vertex v towards its ith neighbor. Updates message_dict, message_mark and is_↩
message_star

sends a message by setting messages, and puts it in the message hash table message_dict. It also updates message_mark and is_star_message corresponding to step s and the input message.

Parameters

m	the message to be sent
v	the vertex from which the message is originated
i	the message is sent to the ith neighbor of v

```
00487                                                                          {
00488     unordered_map<vector<int>, int, vint_hash>::iterator it;
00489     /*
00490     cerr « " send message (";
00491     for (int k=0;k<m.size();k++){
00492       cerr « m[k];
00493       if (k<m.size()-1)
00494         cerr « ", ";
00495     }
00496     cerr « "): " « v « " -> " « i;
00497     */
```

```
00498
00499    it = message_dict.find(m);
00500    if (it == message_dict.end()){
00501      // this is a new message
00502      message_dict.insert(pair<vector<int>, int> (m, message_mark.size())); // insert the message into
      the hash table, message_mark[s].size() is in fact the number of registered marks at step s
00503      messages[v][i] = message_mark.size(); // set the message
00504      message_mark.push_back(m.back()); // register m by adding its mark component (which is m.back()
      the last element in m) to the list of marks at step s
00505      is_star_message.push_back(m[0]==-1); // check if m is star type, and add this information to the
      list
00506    }else{
00507      // the message already exists, just use the registered integer value corresponding to m and send
      the message
00508      messages[v][i] = it->second;
00509    }
00510    //cerr << " message = " << messages[v][i] << endl;
00511 }
```

6.10.3.2 update_messages()

```
void graph_message::update_messages (
            const marked_graph & G )   [private]
```

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 $M_k(v, w) = (\tau_G(v), \deg_G(v) - 1, s, \xi_G(w, v))$.

```
00024 {
00025    logger::current_depth++;
00026    logger::add_entry("graph_message::update_message init", "");
00027    int nu_vertices = G.nu_vertices;
00028    int w;
00029    int my_location;
00030    messages.resize(nu_vertices);
00031    //inward_message.resize(nu_vertices);
00032    //message_dict.resize(h);
00033    //message_list.resize(h);
00034    //message_mark.resize(h);
00035    //is_star_message.resize(h);
00036
00037
00038    // initialize the messages
00039
00040    logger::add_entry("resizing messages", "");
00041    for (int v=0;v<nu_vertices;v++){
00042      messages[v].resize(G.adj_list[v].size());
00043      //inward_messages[v].resize(G.adj_list[v].size());
00044      //for (int i=0;i<G.adj_list[v].size();i++){
00045      //messages[v][i].resize(h);
00046        //inward_messages[v][i].resize(h);
00047      //}
00048    }
00049
00050    logger::add_entry("initializing messages","");
00051    vector<int> m(3);
00052    unordered_map<vector<int>, int, vint_hash>::iterator it;
00053    //map<vector<int>, int>::iterator it;
00054
00055    for (int v=0;v<nu_vertices;v++){
```

```
00056
00057       for (int i=0;i<G.adj_list[v].size();i++){
00058         // 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.
00059
00060
00061         //vector<int> m;
00062         //m.clear();
00063         //m.push_back(G.ver_mark[v]);
00064         //m.push_back(0);
00065         //m.push_back(G.adj_list[v][i].second.first);
00066
00067         m[0] = G.ver_mark[v];
00068         m[1] = 0;
00069         m[2] = G.adj_list[v][i].second.first;
00070         send_message(m, v, i);
00071
00072         // adding this message to the message dictionary
00073         //it = message_dict[0].find(m);
00074         //w = G.adj_list[v][i].first;
00075
00076         /*
00077         if (it == message_dict[0].end()){
00078           message_dict[0][m] = message_list[0].size();
00079           messages[v][i][0] = message_list[0].size();
00080           message_list[0].push_back(m);
00081
00082         }else{
00083           messages[v][i][0] = it->second;
00084         }
00085         */
00086         //messages[v][i][0] = message_dict[0][m]; // the message at time 0
00087       }
00088   }
00089
00090   // these are copies of message_dict, message_mark and is_star_message at the previous step, which
      are used to update messages at the current step.
00091
00092   //unordered_map<vector<int>, int, vint_hash> message_dict_old;
00093   //vector<int> message_mark_old;
00094   vector<bool> is_star_message_old;
00095   vector<vector<int> > messages_old;
00096
00097   // updating messages
00098   logger::add_entry("updating messages", "");
00099   m.reserve(5+ 2 * Delta);
00100   vector<int> m2;
00101   m2.reserve(5 + 2*Delta); // an auxiliary message when we need to work with two types of messages
      simultaneously
00102   duration<float> diff;
00103   high_resolution_clock::time_point t1, t2;
00104   float agg_search = 0;
00105   float agg_insert = 0;
00106   float agg_m = 0;
00107   float agg_sort = 0;
00108   float agg_neigh_message = 0;
00109
00110   vector<pair<pair<int, int>, int> > neighbor_messages; // the first component is the message and the
      second is the name of the neighbor
00111   // the second component is stored so that after sorting, we know the owner of the message
00112   neighbor_messages.reserve(5+2*Delta);
00113
00114   int nu_star_neigh; // number of star neighbors, i.e. neighbors of a vertex v whose message towards v
      are star type
00115   int star_neigh_index; // the index of the star neighbor of v, this is only useful when there is one
      star neighbor, if there are more than one star neighbor, then the message sent from v towards all
      other neighbors are star typed
00116   int star_neigh; // the label of the star neighbor, i.e. star_neigh =
      G.adj_list[v][star_neigh_index].first;
00117   int previous_message; // the message from the previous step
00118   int mark_to_v; // mark towards the current vertex directed from its neighbor
00119   vector<int> neighbors_list; // the list of neighbors of a vertex in the order after sorting with
      respect to their corresponding messages
00120   neighbors_list.reserve(Delta + 3);
00121   int deg_v; // the degree of vertex v
00122
00123
00124
00125   for (int s=1; s<h;s++){ // s stands for step
00126     //cerr « endl « endl« " depth " « s « endl;
00127     // store variables corresponding to the previous step in their old version, and clearing the
      variables for this step:
00128     //message_dict_old = message_dict;
00129     messages_old = messages;
00130     //message_mark_old = message_mark;
```

```
00131     is_star_message_old = is_star_message;
00132     message_dict.clear();
00133     message_mark.clear();
00134     is_star_message.clear();
00135     // we do not clear messages since we need its size to be the same, and we only modify its content
00136
00137
00138     for (int v=0;v<nu_vertices;v++){
00139       deg_v = G.adj_list[v].size();
00140       if (deg_v==1){
00141         // no need to collect messages, there is only one message towards the one neighbor, which is
     known
00142         m.resize(3); // there is only one message which is of the form \f$(\theta, 0, x)\f$, where
     \f$\theta\f$ is the mark of v, and \f$x = \xi_G(w,v)\f$  where \f$w\f$ is the only neighbor of v
00143         m[0] = G.ver_mark[v];
00144         m[1] = 0;
00145         m[2] = G.adj_list[v][0].second.first;
00146         send_message(m, v, 0);
00147       }else{
00148         if (deg_v <= Delta){
00149           neighbor_messages.clear();
00150           nu_star_neigh = 0;
00151           for (int i=0;i<deg_v;i++){
00152             w = G.adj_list[v][i].first; // neighbor label
00153             my_location = G.index_in_neighbor[v][i];
00154             previous_message = messages_old[w][my_location]; // the message sent from this neighbor
     towards v at time t-1
00155             // check if previous message is star
00156             if (is_star_message_old[previous_message]){
00157               nu_star_neigh ++;
00158               star_neigh_index = i;
00159               star_neigh = w;
00160             }
00161             if (nu_star_neigh >= 2)
00162               break; // then message towards all neighbors will be star, no need to collect messages
00163             mark_to_v = G.adj_list[v][i].second.first;
00164             neighbor_messages.push_back(pair<pair<int, int> , int> (pair<int,int>(previous_message,
     mark_to_v), i));
00165           }
00166           if (nu_star_neigh == 2){
00167             // message towards all the neighbors will be star
00168             m.resize(2);
00169             m[0] = -1;
00170             for (int i=0;i<G.adj_list[v].size();i++){
00171               m[1] = G.adj_list[v][i].second.first;
00172               send_message(m, v, i); // send message m from v towards its ith neighbor at step s
00173             }
00174           }
00175           if (nu_star_neigh == 1){
00176             // the message towards all the neighbors except for that star neighbor is star
00177             // let m be the message towards that neighbor and m2 be the star messages
00178
00179             // sorting neighbor messages
00180             sort(neighbor_messages.begin(), neighbor_messages.end());
00181
00182             // preparing m
00183             m.resize(0);
00184             m.push_back(G.ver_mark[v]);
00185             m.push_back(G.adj_list[v].size()-1);
00186
00187
00188
00189             for (int i=0;i<neighbor_messages.size();i++){
00190               if (neighbor_messages[i].second != star_neigh_index){
00191                 // collect the messages of non star neighbors
00192                 m.push_back(neighbor_messages[i].first.first);
00193                 m.push_back(neighbor_messages[i].first.second);
00194               }
00195             }
00196             // finalize m by inserting its mark component
00197             m.push_back(G.adj_list[v][star_neigh_index].second.first);
00198
00199             // prepare star messages
00200             m2.resize(2);
00201             m2[0] = -1;
00202             for (int i=0;i<deg_v;i++){
00203               if (i==star_neigh_index){
00204                 // send the prepared message m
00205                 send_message(m, v, i);
00206               }else{
00207                 // prepare a star message and send it
00208                 m2[1] = G.adj_list[v][i].second.first;
00209                 send_message(m2, v, i);
00210               }
00211             }
00212           }
00213
```

```
00214               if (nu_star_neigh == 0){
00215                   // no star neighbor, so we can prepare messages to all neighbors comfortably as none of
     them are star type
00216                   // we do this by a masking technique
00217                   // sorting neighbor messages
00218                   sort(neighbor_messages.begin(), neighbor_messages.end());
00219                   if (neighbor_messages.size() != deg_v){
00220                       cerr « " Error: no star messages and yet neighbor_messages does not have a size equal to
     the deg of v, step " « s « " v= " « v «   " deg_v= " « deg_v « " neighbor_messages.size() " «
     neighbor_messages.size() « endl;
00221                   }
00222                   m.resize(1 + 1 + 2*(G.adj_list[v].size()-1) +1); // 1 for vertex mark, 1 for deg -1, for
     (deg-1) many neighbors, each we have 2 values, and finally 1 for the mark component
00223                   m[0] = G.ver_mark[v];
00224                   m[1] = G.adj_list[v].size()-1;
00225                   neighbors_list.resize(deg_v);
00226                   for (int i=0;i<neighbor_messages.size();i++){
00227                     m[2*(i+1)]   = neighbor_messages[i].first.first;
00228                     m[2*(i+1)+1] = neighbor_messages[i].first.second;
00229                     neighbors_list[i] = neighbor_messages[i].second;
00230                   }
00231                   // swapping the last message so that its mark component comes first, so that we can treat
     m as a valid message in our standard
00232                   swap(m[2*deg_v], m[2*deg_v+1]);
00233
00234                   for (int i=neighbor_messages.size()-1;i>=0;i--){
00235                     if (i < neighbor_messages.size()-1){
00236                       swap(m[2*deg_v], m[2*(i+1)+1]);
00237                       swap(m[2*deg_v+1], m[2*(i+1)]);
00238                       swap(neighbors_list[deg_v-1], neighbors_list[i]);
00239                     }
00240                     send_message(m,v,neighbors_list[deg_v-1]);
00241                   }
00242                 }
00243             }
00244           if (deg_v > Delta){ // the message towards all neighbors is star
00245             m.resize(2);
00246             m[0] = -1;
00247             for(int i=0;i<deg_v;i++){
00248               m[1] = G.adj_list[v][i].second.first;
00249               send_message(m,v,i);
00250             }
00251           }
00252         }
00253       }
00254   }
00255
00256
00257   logger::add_entry("* symmetrizing", "");
00258   bool star1, star2;
00259   m.resize(2); // prepare for star message
00260   m[0] = -1;
00261   for (int v=0;v<nu_vertices;v++){
00262     for (int i=0;i<G.adj_list[v].size();i++){
00263       w = G.adj_list[v][i].first;
00264       my_location = G.index_in_neighbor[v][i];
00265       if (w > v){ // to avoid going over edges twice
00266         star1 = is_star_message[messages[v][i]];
00267         star2 = is_star_message[messages[w][my_location]];
00268         if (star1 and !star2){
00269           // message[w][my_location] should be star
00270           m[1] = G.adj_list[v][i].second.second;
00271           send_message(m,w,my_location);
00272         }
00273         if (!star1 and star2){
00274           // messages[v][i] should also become star
00275           m[1] = G.adj_list[v][i].second.first;
00276           send_message(m,v,i);
00277         }
00278         if ((!star1 and !star2) and (G.adj_list[v].size() > Delta or G.adj_list[w].size()>Delta)){ //
     this activates only when h = 1, ensures truncation of degrees bigger than delta
00279           // message[w][my_location] should be star
00280           m[1] = G.adj_list[v][i].second.second;
00281           send_message(m,w,my_location);
00282           // messages[v][i] should also become star
00283           m[1] = G.adj_list[v][i].second.first;
00284           send_message(m,v,i);
00285         }
00286       }
00287     }
00288   }
00289   logger::current_depth--;
00290
00291
00292
00293
00294   // =================================================
```

```
00295   // ==================================================
00296   // ==================================================
00297   // ==================================================
00298   // ==================================================
00299   // ==================================================
00300   // ==================================================
00301   // ==================================================
00302
00303   /*
00304   for (int t=1;t<h;t++){
00305     for (int v=0;v<nu_vertices;v++){
00306       //cerr « " vertex " « v « endl;
00307       if (G.adj_list[v].size() <= Delta){
00308         // the degree of v is no more than Delta
00309         // do the standard message passing by aggregating messages from neighbors
00310         // stacking all the messages from neighbors of v towards v
00311         neighbor_messages.clear();
00312
00313         // 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
00314         //t1 = high_resolution_clock::now();
00315         for (int i=0;i<G.adj_list[v].size();i++){
00316           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
00317           //my_location = G.adj_location[w].at(v); <--- the inefficient way
00318           my_location = G.index_in_neighbor[v][i];
00319           // where is the place of node v among the list of neighbors of the ith neighbor of v
00320           int previous_message = messages[w][my_location][t-1]; // the message sent from this neighbor
      towards v at time t-1
00321           int mark_to_v = G.adj_list[v][i].second.first;
00322           neighbor_messages.push_back(pair<pair<int, int> , int> (pair<int,int>(previous_message,
      mark_to_v), w));
00323         }
00324         //t2 = high_resolution_clock::now();
00325         //diff = t2 - t1;
00326         //agg_neigh_message += diff.count();
00327
00328         //t1 = high_resolution_clock::now();
00329         sort(neighbor_messages.begin(), neighbor_messages.end()); // sorts lexicographically
00330         //t2 = high_resolution_clock::now();
00331         //diff = t2 - t1;
00332         //agg_sort += diff.count();
00333
00334         for (int i=0;i<G.adj_list[v].size();i++){
00335           // let w be the current ith neighbor of v
00336           int w = G.adj_list[v][i].first;
00337           // first, start with the mark of v and the number of offsprings in the subgraph component of
      the message
00338           //vector<int> m; // the message that v is going to send to w
00339           //t1 = high_resolution_clock::now();
00340           m.clear();
00341           m.push_back(G.ver_mark[v]); // mark of v
00342           m.push_back(G.adj_list[v].size()-1); // the number of offsprings in the subgraph component
      of the message
00343           //t2 = high_resolution_clock::now();
00344           //diff = t2 - t1;
00345           //agg_m += diff.count();
00346
00347           // stacking messages from all neighbors of v expect for w towards v at time t-1
00348           for (int j=0;j<G.adj_list[v].size();j++){
00349             if (neighbor_messages[j].second != w){
00350               if (message_list[t-1][neighbor_messages[j].first.first][0] == -1){
00351                 // this means that one of the messages that should be aggregated is * typed, therefore
      the outgoing messages should also be * typed
00352                 // 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
00353                 // 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
00354                 //t1 = high_resolution_clock::now();
00355                 m.resize(0);
00356                 m.push_back(-1);
00357                 //t2 = high_resolution_clock::now();
00358                 //diff = t2 - t1;
00359                 //agg_m += diff.count();
00360                 break; // the message is decided, we do not need to go over any of the other neighbor
      messages, hence break
00361               }
00362               // this message should be added to the list of messages
00363               //t1 = high_resolution_clock::now();
00364               m.push_back(neighbor_messages[j].first.first); // message part
00365               m.push_back(neighbor_messages[j].first.second); // mark part towards v
00366               //t2 = high_resolution_clock::now();
00367               //diff = t2 - t1;
00368               //agg_m += diff.count();
00369
00370             }
```

```
00371              }
00372              // 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.
00373              // finally, the mark of the edge between v and w towards v, \xi(w,v), should be added to
     this list
00374              //t1 = high_resolution_clock::now();
00375              m.push_back(G.adj_list[v][i].second.first);
00376              //t2 = high_resolution_clock::now();
00377              //diff = t2 - t1;
00378              //agg_m += diff.count();
00379
00380              // set the current message
00381              //t1 = high_resolution_clock::now();
00382              it = message_dict[t].find(m);
00383              //t2 = high_resolution_clock::now();
00384              //diff = t2 - t1;
00385              //agg_search += diff.count();
00386
00387              if (it == message_dict[t].end()){
00388                //t1 = high_resolution_clock::now();
00389                //message_dict[t][m] = message_list[t].size();
00390                message_dict[t].insert(pair<vector<int>, int> (m, message_list[t].size()));
00391                //t2 = high_resolution_clock::now();
00392                //diff = t2 - t1;
00393                //agg_insert += diff.count();
00394
00395                messages[v][i][t] = message_list[t].size();
00396                message_list[t].push_back(m);
00397              }else{
00398                messages[v][i][t] = it->second;
00399              }
00400           }
00401         }else{
00402           // if the degree of v is bigger than Delta, the message towards all neighbors is of the form *
00403           // 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
00404           for (int i=0;i<G.adj_list[v].size();i++){
00405             //vector<int> m; // the current message from v to ith neighbor
00406             //t1 = high_resolution_clock::now();
00407             m.clear();
00408             m.resize(2);
00409             m[0] = -1;
00410             m[1] = G.adj_list[v][i].second.first;
00411             //t2 = high_resolution_clock::now();
00412             //diff = t2 - t1;
00413             //agg_m += diff.count();
00414
00415             // set the current message
00416             //t1 = high_resolution_clock::now();
00417             it = message_dict[t].find(m);
00418             //t2 = high_resolution_clock::now();
00419             //diff = t2 - t1;
00420             //agg_search += diff.count();
00421
00422             if (it == message_dict[t].end()){
00423                //t1 = high_resolution_clock::now();
00424                //message_dict[t][m] = message_list[t].size();
00425                message_dict[t].insert(pair<vector<int>, int> (m, message_list[t].size()));
00426                //t2 = high_resolution_clock::now();
00427                //diff = t2 - t1;
00428                //agg_insert += diff.count();
00429                messages[v][i][t] = message_list[t].size();
00430                message_list[t].push_back(m);
00431             }else{
00432                messages[v][i][t] = it->second;
00433             }
00434           }
00435         }
00436       }
00437    }
00438    cerr « " total time to search in hash table: " « agg_search « endl;
00439    cerr « " total time to insert in hash table: " « agg_insert « endl;
00440    cerr « " total time to modify vector m  " « agg_m « endl;
00441    cerr « " total time to sort  " « agg_sort « endl;
00442    cerr « " total time to collect neighbor messages " «  agg_neigh_message « endl;
00443    // 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)
00444    logger::add_entry("* symmetrizing", "");
00445    for (int v=0;v<nu_vertices;v++){
00446      for (int i=0;i<G.adj_list[v].size();i++){
00447        if (message_list[h-1][messages[v][i][h-1]][0] == -1){
00448          // it is of the form *
00449          w = G.adj_list[v][i].first; // the other endpoint of the edge
00450          //my_location = G.adj_location[w].at(v); // so that adj_list[w][my_location].first = v
00451          my_location = G.index_in_neighbor[v][i];
```

```
00452
00453        //vector<int> m;
00454        m.clear();
00455        m.resize(2);
00456        m[0] = -1;
00457        m[1] = G.adj_list[v][i].second.second; // the mark towards w
00458        if (message_dict[h-1].find(m) == message_dict[h-1].end()){
00459          message_dict[h-1][m] = message_list[h-1].size();
00460          message_list[h-1].push_back(m);
00461        }
00462        messages[w][my_location][h-1] = message_dict[h-1][m];
00463      }
00464    }
00465  }
00466
00467  // setting message_mark and is_star_message
00468  logger::add_entry("setting message_mark and is_star_message", "");
00469  message_mark.resize(message_list[h-1].size());
00470  is_star_message.resize(message_list[h-1].size());
00471  for (int i=0;i<message_list[h-1].size();i++){
00472    message_mark[i] = message_list[h-1][i].back(); // the last element is the mark component
00473    is_star_message[i] = (message_list[h-1][i][0] == -1); // message is star type when the first
    element is -1
00474  }
00475  logger::current_depth--;
00476  */
00477 }
```

6.10.4 Member Data Documentation

6.10.4.1 Delta

```
int graph_message::Delta  [private]
```

the maximum degree threshold

6.10.4.2 h

```
int graph_message::h  [private]
```

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

6.10.4.3 is_star_message

```
vector<bool> graph_message::is_star_message
```

for an integer message m, is_star_message[m] is true if m is a star message and false otherwise. Note that m is star type iff the first index in the vector message corresponding to m is -1. This is updated at step of update_messages, so when it is over, it corresponds to step (depth) h-1.

6.10.4.4 message_dict

```
unordered_map<vector<int>, int, vint_hash> graph_message::message_dict
```

message_dict is the message dictionary at any step that update_messages is running, which maps each message to its corresponding index in the dictionary. When update_messages is over, this corresponds to step (depth) h-1

6.10.4.5 message_mark

`vector<int> graph_message::message_mark`

for an integer message m, message_mark[m] is the mark component associated to the message m at any step that update_messages is working. This is basically the last index in the vector message associate to m. When update_messages is over, this corresponds to step (depth) h-1.

6.10.4.6 messages

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

messages[v][i] is the integer version of the message from vertex v towards its ith neighbor (in the order given by adj_list of vertex i in graph G). The message is at any given step that update_messages is running, so after finishing update_message, the messages are at step (depth) h-1.

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

- graph_message.h
- graph_message.cpp

6.11 ibitstream Class Reference

deals with reading bit streams from binary files, this is the reverse of obitstream

`#include <bitstream.h>`

Public Member Functions

- void read_chunk ()

    reads one chunk (4 bytes) from the input file and stores it in buffer
- unsigned int read_bits (unsigned int k)

    read k bits from the input, interpret it as integer (first bit read is MSB) and return its value. Here, k must be in the range 1 <= k <= BIT_INT
- void read_bits (int k, bit_pipe &B)

    reads k bits from input and stores in the given bit_pipe. $k \geq 1$ is arbitrary. The bits are stored in the bit_pipe so that can be interpreted as integer (e.g. mpz_class) so the LSB is located in the rightmost bit of the rightmost chunk (unlike the usual bit_pipe situation). We assume that the B given here is an empty bit_pipe
- void read_bits_append (int k, bit_pipe &B)

    similar to read_bits, but B does not have to be empty and the result will be appended to B (this is used in order to recursively implement read_bits)
- bool read_bit ()

    read one bit from input and return true if its value is 1 and false otherwise.
- ibitstream (string file_name)
- ibitstream & operator>> (unsigned int &n)

    reads an unsigned int from the input using Elias delta decoding and saves it in the reference given
- ibitstream & operator>> (mpz_class &n)

    reads a nonnegative mpz_class integer using Elias delta decoding and stores in the reference given
- void bin_inter_decode (vector< int > &a, int b)

    uses binary interpolative coding algorithm to decode for an array of increasing nonnegative integers. Caution: we do not sort the decoding vector for efficiency purposes and return elements in the order they were encoded (mid point first left subinterval then subinterval)
- void bin_inter_decode (vector< int > &a, int i, int j, int low, int high)

    using bit interpolative coding algorithm to decode for a subinterval of an array
- void close ()

    closes the session by closing the input file

Private Attributes

- FILE ∗ f

    pointer to input binary file
- unsigned int buffer

    the last chunk read from the input
- unsigned int head_mask

    the place of the head bit in buffer, represented in terms of mask. So if we are in the LSB, head_mask is one, if we are in two bit left of LSB, this is 4 so on. When this is zero, it means the buffer is expired and we should probably read one more chunk from the input file
- unsigned int head_place

    the place of head represented in terms of integer, LSB is 1, left of LSB is 2, 2 left of LSB is 3 and so on. head_place is effectively the number of unread bits remaining in the buffer.

6.11.1 Detailed Description

deals with reading bit streams from binary files, this is the reverse of obitstream

6.11.2 Constructor & Destructor Documentation

6.11.2.1 ibitstream()

```
ibitstream::ibitstream (
            string file_name )  [inline]
00142                                                          {
00143       f = fopen(file_name.c_str(), "rb+");
00144       buffer =0;
00145       head_mask = 0;
00146       head_place = 0;
00147   }
```

6.11.3 Member Function Documentation

6.11.3.1 bin_inter_decode() [1/2]

```
void ibitstream::bin_inter_decode (
            vector< int > & a,
            int b )
```

uses binary interpolative coding algorithm to decode for an array of increasing nonnegative integers. Caution: we do not sort the decoding vector for efficiency purposes and return elements in the order they were encoded (mid point first left subinterval then subinterval)

Parameters

b	The number of bits used in the compression phase to encode size of array and lower and upper values (for graph compression, it is number of bits in the number of vertices).
a	reference to the array to add elements to. Here, we do not erase a, so you need to make sure a is empty.

```
00549                                                          {
00550    unsigned int a_size;
00551    if (b > BIT_INT)
```

```
00552     cerr « " 549 b = " « b « endl;
00553   a_size = read_bits(b); // size of the vector
00554   //cout « "a_size " « a_size « endl;
00555
00556   if (a_size == 0)
00557     return;
00558   if (a_size == 1){
00559     if (b > BIT_INT)
00560       cerr « " 557 b = " « b « endl;
00561     a.push_back(read_bits(b));
00562     return;
00563   }
00564
00565   // read low and high values
00566   unsigned int low, high;
00567   if (b > BIT_INT)
00568     cerr « " 565 b = " « b « endl;
00569   low = read_bits(b);
00570   high = read_bits(b);
00571   //cout « " low " « low « " high " « high « endl;
00572   bin_inter_decode(a, 0, a_size - 1, low, high);
00573   return;
00574 }
```

6.11.3.2 bin_inter_decode() [2/2]

```
void ibitstream::bin_inter_decode (
              vector< int > & a,
              int i,
              int j,
              int low,
              int high )
```

using bit interpolative coding algorithm to decode for a subinterval of an array

Parameters

a	reference to the array to add elements to
i,j	the endpoints of the interval to be decoded (with respect to the encoded array)
low	lower bound on the elements of the encoded array in the interval [i,j]n
high	lower bound on the elements of the encoded array in the interval [i,j]

```
00584                                                                          {
00585   // cout « " i " « i « " j " « j « " low " « low « " high " « high « endl;
00586   if (j < i)
00587     return;
00588   if (i==j){
00589     if(low == high){
00590       a.push_back(low); // the element must be low = high, no other change, nothing to read
00591     }else{
00592       if (nu_bits(high-low) > BIT_INT)
00593         cerr « " 590 nu_bits(high-low) = " « nu_bits(high-low) « endl;
00594       a.push_back(read_bits(nu_bits(high-low)) + low);
00595     }
00596     return;
00597   }
00598
00599   int m = (i+j)/2;
00600   unsigned int L = low + m - i; // lower bound on a[m]
00601   unsigned int H = high - (j - m); // upper bound on a[m]
00602   unsigned int a_m; // the value of the intermediate point
00603   if (L == H){
00604     a_m = L; // there will be no bits to read
00605   }else{
00606     if (nu_bits(H-L) > BIT_INT)
00607       cerr « " 604 nu_bits(H-L) = " « nu_bits(H-L) « endl;
00608     a_m = read_bits(nu_bits(H-L)) + L;
00609   }
00610
00611   a.push_back(a_m);
00612
00613   bin_inter_decode(a, i, m-1, low, a_m - 1);
00614   bin_inter_decode(a, m+1, j, a_m + 1, high);
```

```
00615 }
```

6.11.3.3 close()

```
void ibitstream::close ( )  [inline]
```

closes the session by closing the input file

```
00165 {fclose(f);}
```

6.11.3.4 operator>>() [1/2]

```
ibitstream & ibitstream::operator>> (
            mpz_class & n )
```

reads a nonnegative mpz_class integer using Elias delta decoding and stores in the reference given

```
00489                                                           {
00490   unsigned int L = 0;
00491   //cout << "head_place " << head_place << endl;
00492   //cout << "head_mask  " << head_mask << endl;
00493   while (!read_bit()){
00494     // read until reach one
00495     L++;
00496   }
00497
00498   //cout << " L = " << L << endl;
00499   unsigned int N;
00500   if (L == 0){// special case, avoid going over further calculations
00501     n = 0; // we had subtracted one when encoding
00502     return *this;
00503   }
00504
00505   if (L > BIT_INT)
00506     cerr << " 503 L = " << L << endl;
00507   N = read_bits(L); // read L digits
00508
00509   N += (1<<L); // we must add 2^L
00510   N --; // this was N + 1
00511
00512   // we must read N bits and form n based on that
00513
00514   bit_pipe B;
00515   //cout << " N " << N << endl;
00516   read_bits(N, B);
00517   //cout << " B first " << B << endl;
00518   // we should add a leading 1 to B
00519   // in order to do so, we should consider 2 cases:
00520   if (N % BIT_INT == 0){
00521     // this is the tricky case, since B now contains full chunks and there is no room to add the
     leading 1
00522     // so we need to insert a chunk at the beginning and place the leading bit there
00523     // since the leading bit will be in the rightmost bit in this case, the value of the initial chunk
     is 1 in this case
00524     B.bits.insert(B.bits.begin(), 1);
00525   }else{
00526     // in this case, the lading bit will be placed in the first chunk of B
00527     B.bits[0] |= (1 << (N % BIT_INT));
00528   }
00529   //cout << " B " << B << endl;
00530   //cout << B.bits[0] << endl;
00531
00532   // construct the mpz_clas
00533   mpz_import(n.get_mpz_t(),
00534               B.bits.size(), // the number of words
00535               1, // order: 1 means first significant word first
00536               sizeof(unsigned int), // each word is this many bytes
00537               0, // endian can be 1 for most significant byte first, -1 for least significant first, or
     0 for the native endianness of the host CPU.
00538               0, // nails
00539               &B.bits[0]); //&B.bits[0]);
00540
00541   n --; // when encoding, we added 1 to make sure it is positive
00542   return *this;
00543 }
```

6.11.3.5 operator>>() [2/2]

```
ibitstream & ibitstream::operator>> (
            unsigned int & n )
```

reads an unsigned int from the input using Elias delta decoding and saves it in the reference given

```
00455                                                          {
00456    // implement Elias delta decoding
00457    //bitset<32> B(buffer);
00458    //cerr « " buffer " « B « endl;
00459    //cerr « " head " « bitset<32>(head_mask) « endl;
00460    //cerr « " head position " « head_place « endl;
00461
00462    unsigned int L = 0;
00463    while (!read_bit()){
00464      // read until reach one
00465      L++;
00466    }
00467    //cerr « " L " « L « endl;
00468
00469    unsigned int N;
00470    if (L == 0){// special case, avoid going over further calculations
00471      n = 0; // we had subtracted one when encoding
00472      return *this;
00473    }
00474    if (L > BIT_INT)
00475      cerr « " 472 L = " « L « endl;
00476    N = read_bits(L); // read L digits
00477
00478    N += (1«L); // we must add 2^L
00479    N --; // this was N + 1
00480
00481    if (N > BIT_INT)
00482      cerr « " 479 N = " « N « " L = " « L « endl;
00483    n = read_bits(N); // read N digits
00484    n += (1 « N); // we must add 2^N
00485    n --; // when we encoded, in order to get a positive integer, we added one, now we subtract one
00486    return *this;
00487 }
```

6.11.3.6 read_bit()

```
bool ibitstream::read_bit ( )
```

read one bit from input and return true if its value is 1 and false otherwise.

```
00618                                                          {
00619    if (head_mask == 0){ // nothing is in buffer
00620      //cerr « " read a chunk " « endl;
00621      read_chunk();
00622    }
00623    bool ans = head_mask & buffer; // look at the value of buffer at the bit where the head_mask is
      pointing to
00624    head_mask »= 1; // go one bit to the right
00625    head_place --;
00626    //cerr « " read bit " « ans « endl;
00627    return ans;
00628 }
```

6.11.3.7 read_bits() [1/2]

```
void ibitstream::read_bits (
            int k,
            bit_pipe & B )
```

reads k bits from input and stores in the given bit_pipe. $k \geq 1$ is arbitrary. The bits are stored in the bit_pipe so that can be interpreted as integer (e.g. mpz_class) so the LSB is located in the rightmost bit of the rightmost chunk (unlike the usual bit_pipe situation). We assume that the B given here is an empty bit_pipe

```
00438                                                          {
00439    //cerr « " read_bits " « k « endl;
00440    // assumption: B is empty bit_pipe
00441    if (B.bits.size() != 0 or B.last_bits != 0){
```

```
00442      cerr « " ERROR: ibitstream::read_bits(int k, bit_pipe& B) must be called with an empty bit_pipe, a
      nonempty bitpipe is given with B.bits.size() = " « B.bits.size() « endl;
00443   }
00444   read_bits_append(k, B);
00445   // there might be a few zero chunks at the beginning of B which are redundant, we remove them here
00446   // the number of nonzero chunks is exactly the floor of k / BIT_INT
00447   int nonzero_chunks = k / BIT_INT;
00448   if (k % BIT_INT != 0)
00449     nonzero_chunks ++; // take the floor
00450   //cerr « " nonzero_chunks " « nonzero_chunks « endl;
00451   if (nonzero_chunks < B.bits.size())
00452     B.bits.erase(B.bits.begin(), B.bits.begin() + B.bits.size() - nonzero_chunks);
00453 }
```

6.11.3.8   read_bits() [2/2]

```
unsigned int ibitstream::read_bits (
            unsigned int k )
```

read k bits from the input, interpret it as integer (first bit read is MSB) and return its value. Here, k must be in the range 1 <= k <= BIT_INT

```
00336                                                          {
00337
00338   //cerr « " read bits with k = " « k « endl;
00339   if (k < 1 or k > BIT_INT){
00340     cerr « "ERROR: ibitstream::read_bits called with k out of range, k = " « k « endl;
00341   }
00342   if (head_place == 0)// no bits left
00343     read_chunk();
00344   if (head_place >= k){ // head_place is effectively the number of unread bits remaining in the buffer
00345     //cerr « " head_place >= k head_mask = " « head_mask « " head_place = " « head_place « endl;
00346     // there are enough number of bits in the current buffer to read
00347     // mask the input
00348     unsigned int mask = mask_gen(k); // k ones
00349     // now we should shift mask to start at head_place
00350     mask «= (head_place - k);
00351     unsigned int ans = buffer & mask; // mask out the corresponding bits
00352     ans »= (head_place - k); // bring it back to LSB
00353
00354     // we need to shift head k bits to the right
00355     // in some compilers, »= 32 does strange things, in fact it does nothing. To avoid that, I shift k
      - 1 bits and then an extra 1 bit
00356     head_mask »= k - 1;
00357     head_mask »= 1;
00358     head_place -= k;
00359     //cerr « " after head_mask " « head_mask « " head_place = " « head_place « endl;
00360     return ans;
00361   }else{
00362     // there is not enough bits in the current buffer.
00363     // So we should read head_place many bits from the current buffer
00364     // then read another chunk from input file
00365     // and then read k - head_place bits from the new buffer
00366     // we do these two steps recursively
00367     // but first need to store the number of bits we will have to read in the future, since these
      variables will be modified later:
00368     unsigned int future_bits = k - head_place;
00369     if (head_place > BIT_INT)
00370       cerr « " 367 head_place = " « head_place « endl;
00371     unsigned int a = read_bits(head_place); // the bits from the current buffer
00372     read_chunk();
00373     if (future_bits > BIT_INT)
00374       cerr « " 371 future_bits = " « future_bits « endl;
00375     unsigned int b = read_bits(future_bits); // bits from the next buffer
00376     // now we need to combine these
00377     // in order to do so, we need to shift a to the left and combine with b
00378     // but the number of bits we need to shift a is exactly future bits
00379     a «= future_bits;
00380     return a | b;
00381   }
00382 }
```

6.11.3.9   read_bits_append()

```
void ibitstream::read_bits_append (
            int k,
            bit_pipe & B )
```

similar to read_bits, but B does not have to be empty and the result will be appended to B (this is used in order to recursively implement read_bits)

```
00384                                        {
00385   //cout « " read_bits called k = " « k « " head_place = " « head_place « endl;
00386   // by assumption, when calling this function, B has full chunks (last_bits is either zero so
     BIT_INT)
00387   if (k == 0)
00388     return; // nothing remains to be done
00389   if (head_place == 0){
00390     // we are over with the current bits in the buffer
00391     // so we need to load a few chunks from the input
00392     // we should append k / BIT_INT full chunks to B and then k % BIT_INT bits from the next chunk
00393     unsigned int full_chunks = k / BIT_INT;
00394     if (full_chunks > 0){
00395       B.bits.resize(B.bits.size() + full_chunks);
00396       fread(&B.bits[B.bits.size() - full_chunks], sizeof(unsigned int), full_chunks, f); // read
     full_chunks many chunks
00397       B.last_bits = BIT_INT; // the last chunk contains full bits
00398     }
00399     unsigned int res_bits = k % BIT_INT; // the remaining bits to be read
00400     if (res_bits > 0){
00401       // we need to read an extra res bits
00402       if (res_bits > BIT_INT)
00403         cerr « " 400 res_bits = " « res_bits « endl;
00404       unsigned int res = read_bits (res_bits); // read res many bits
00405       // we should shift res_bits so that its MSB is the leftmost bits of the chunk
00406       res «= (BIT_INT - res_bits);
00407       B.bits.push_back(res);
00408       B.last_bits = res_bits;
00409     }
00410   }else{
00411     if (k <= head_place){
00412       // there are enough bits to read
00413       if (k > BIT_INT)
00414         cerr « " 411 k = " « k « endl;
00415       unsigned int a = read_bits(k);
00416       // no need to shift a since we need LSB of a to be in the rightmost bit
00417       B.bits.push_back(a);
00418       B.last_bits = BIT_INT;
00419     }else{
00420       // read head_place bits and call again
00421       unsigned int future_read; // number of bits to read in future after calling the read_bits
     function below
00422       future_read = k - head_place;
00423       if (head_place > BIT_INT)
00424         cerr « " 421 head_place = " « head_place « endl;
00425       unsigned int a = read_bits(head_place);
00426       B.bits.push_back(a);
00427       B.last_bits = BIT_INT;
00428       read_bits_append(future_read, B); // read the remaining bits
00429     }
00430   }
00431
00432   B.shift_right(BIT_INT - B.last_bits); // so that LSB of B is the rightmost bit of the lats chunk.
00433
00434   // this is important to make sure that B is correctly representing an integer and can be converted
     to mpz_class
00435   // TODO issue of 2^k - 1 correct
00436 }
```

6.11.3.10 read_chunk()

```
void ibitstream::read_chunk ( )
```

reads one chunk (4 bytes) from the input file and stores it in buffer

```
00328                                        {
00329   fread(&buffer, sizeof(unsigned int), 1, f);
00330   //cout « " in read chunk  buffer = " « bitset<32>(buffer) « endl;
00331   head_mask = 1 « (BIT_INT - 1); // pointing to the MSB which is the first bit to consider
00332   head_place = BIT_INT;
00333 }
```

6.11.4 Member Data Documentation

6.11.4.1 buffer

```
unsigned int ibitstream::buffer  [private]
```

the last chunk read from the input

6.11.4.2 f

```
FILE* ibitstream::f  [private]
```

pointer to input binary file

6.11.4.3 head_mask

```
unsigned int ibitstream::head_mask  [private]
```

the place of the head bit in buffer, represented in terms of mask. So if we are in the LSB, head_mask is one, if we are in two bit left of LSB, this is 4 so on. When this is zero, it means the buffer is expired and we should probably read one more chunk from the input file

6.11.4.4 head_place

```
unsigned int ibitstream::head_place  [private]
```

the place of head represented in terms of integer, LSB is 1, left of LSB is 2, 2 left of LSB is 3 and so on. head_place is effectively the number of unread bits remaining in the buffer.

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

- bitstream.h
- bitstream.cpp

6.12 log_entry Class Reference

```
#include <logger.h>
```

Public Member Functions

- log_entry (string name, string description, int depth)

Public Attributes

- string name
- string description
- int depth
- high_resolution_clock::time_point t
- system_clock::time_point sys_t

6.12.1 Constructor & Destructor Documentation

6.12.1.1 log_entry()

```
log_entry::log_entry (
            string name,
            string description,
            int depth )
00016                                                                    {
00017   name = name_;
00018   description = description_;
00019   depth = depth_;
00020   t = high_resolution_clock::now();
00021   sys_t = system_clock::now();
00022 }
```

6.12.2 Member Data Documentation

6.12.2.1 depth

```
int log_entry::depth
```

6.12.2.2 description

```
string log_entry::description
```

6.12.2.3 name

```
string log_entry::name
```

6.12.2.4 sys_t

```
system_clock::time_point log_entry::sys_t
```

6.12.2.5 t

```
high_resolution_clock::time_point log_entry::t
```

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

- logger.h
- logger.cpp

6.13 logger Class Reference

```
#include <logger.h>
```

Static Public Member Functions

- static void add_entry (string name, string description)
- static void start ()
- static void stop ()
- static void log ()
- static void item_start (string name)
- static void item_stop (string name)

Static Public Attributes

- static bool verbose = true

  if true, every entry is printed at the time of its report, default is false

- static bool stat = false

  if true, statistics will be displayed, e.g. number of star vertices / edges or the number of partition graphs etc

- static ostream ∗ verbose_stream = &cout

  the stream for printing entries at the time of arrival, default is cout

- static bool report = true

  if true, at the end of the program, a hierarchical report is printed

- static ostream ∗ report_stream = &cout

  the stream to report the final report. Default is cout

- static ostream ∗ stat_stream = &cout

  the stream to report the statistics. Default is cout

- static vector< log_entry > logs
- static int current_depth = 1
- static map< string, float > item_duration
- static map< string, high_resolution_clock::time_point > item_last_start

  the last time each item was started

6.13.1 Member Function Documentation

6.13.1.1 add_entry()

```
void logger::add_entry (
            string name,
            string description ) [static]
00026 {
00027   //cerr « " adding entry current_depth " « current_depth « endl;
00028   log_entry new_entry(name, description, current_depth);
00029   logger::logs.push_back(new_entry);
00030   if (logger::verbose){
00031     string s = "";
00032     time_t tt = system_clock::to_time_t(new_entry.sys_t);
00033     char buffer[80];
00034     strftime(buffer, 80, "%F %r", localtime(&tt));
00035     for (int i=1;i<(current_depth-1);i++)
00036       s += "|---";
00037     string buffer_str(buffer);
00038     s +=  name + " (" + description + ") ";
00039     s += buffer_str;
00040     *verbose_stream « s « endl;
00041   }
00042 }
```

6.13.1.2 item_start()

```
void logger::item_start (
              string name ) [static]
00128                                   {
00129   item_last_start[name] = high_resolution_clock::now();
00130 }
```

6.13.1.3 item_stop()

```
void logger::item_stop (
              string name ) [static]
00132                                  {
00133   high_resolution_clock::time_point t = high_resolution_clock::now();
00134   duration<float> diff = t - item_last_start[name];
00135   item_duration[name] += diff.count();
00136 }
```

6.13.1.4 log()

```
void logger::log ( ) [static]
00045                          {
00046   //cerr « " log started " « endl;
00047   int max_depth = 0;
00048   for (int i=0;i<logs.size(); i++){
00049     if (logs[i].depth > max_depth)
00050       max_depth = logs[i].depth;
00051   }
00052
00053   vector<int> parent(logs.size()); // for a log L, parent[L] is the max index i < L such that depth[i]
       < depth[L], this shows in what block we are in
00054   vector<int> next(logs.size()); // for a log L, next[L] is the min index i > L such that depth[i] >=
       depth[L], this is the index right after L in its block, or if L is the last entry in its block, it is
       the start of the next block. The diff between L and next[L] shows the duration of running L
00055   for (int i=1;i<(logs.size()-1);i++){
00056     // finding parent[i]
00057     //cerr « " finding parent " « i « endl;
00058     for (int j=(i-1); j>=0 ;j--){
00059       if (logs[j].depth < logs[i].depth){
00060         parent[i] = j;
00061         break;
00062       }
00063     }
00064     //cerr « " parent[" « i « "] = " « parent[i] « endl;
00065     //cerr « " finding next " « i « endl;
00066     for (int j=(i+1); j<logs.size();j++){
00067       if (logs[j].depth <= logs[i].depth){
00068         next[i] = j;
00069         break;
00070       }
00071     }
00072     //cerr « " next[" « i « "] = " « next[i] « endl;
00073   }
00074   next[0] = logs.size()-1; // next of start entry is the finish entry
00075
00076   vector<float> dur(logs.size()); // dur[L] is the duration that entry L takes, meaning the difference
       between L and next[L]
00077   vector<float> block_dur(logs.size()); // the duration of the whole block for each entry, which is
       the difference between
00078   vector<float> block_percent(logs.size()); // the percentage of time each entry takes inside a block
00079   duration<float> diff;
00080   string s;
00081   //cerr « " logs.size() " « logs.size() « endl;
00082   *report_stream « endl;
00083   for (int i=1;i<(logs.size()-1); i++){
00084     //cerr « " i " « i « endl;
00085     // finding duration[i]
00086     diff = logs[next[i]].t - logs[i].t;
00087     dur[i] = diff.count();
00088     //cerr « " dur[i] " « dur[i] « endl;
00089     // finding block_duration
00090     diff = logs[next[parent[i]]].t - logs[parent[i]].t;
00091     block_dur[i] = diff.count();
00092     //cerr « " block_dur[i] " « block_dur[i] « endl;
00093     block_percent[i] = dur[i] / block_dur[i] * 100;
```

```
00094      //cerr « " block_percent[i] " « block_percent[i] « endl;
00095      s = "";
00096      //cerr « " logs[i].depth " « logs[i].depth « endl;
00097      for (int j=1;j<(logs[i].depth-1); j++)
00098        s += "|---";
00099      s += logs[i].name + " (" + logs[i].description + "): " + to_string(dur[i]) + "s " + "[" +
    to_string(block_percent[i]) + "\%]" + " ";
00100      *report_stream « s « endl;
00101    }
00102
00103    *report_stream « endl « " itemized log " « endl;
00104    for (map<string, float>::iterator it = item_duration.begin(); it!= item_duration.end(); it++)
00105      *report_stream « it->first « " : " « it->second « endl;
00106 }
```

6.13.1.5 start()

```
void logger::start ( )  [static]
00109                  {
00110    log_entry new_log("Start", "", 0);
00111    //cerr « " started " « endl;
00112    logs.push_back(new_log);
00113    //cerr « logger::logs.size() « endl;
00114 }
```

6.13.1.6 stop()

```
void logger::stop ( )  [static]
00117                   {
00118    log_entry new_log("Finish", "", 0);
00119    //cerr « " finished " « endl;
00120    logs.push_back(new_log);
00121    //cerr « logs.size() « endl;
00122    if (report)
00123      log();
00124
00125 }
```

6.13.2 Member Data Documentation

6.13.2.1 current_depth

```
int logger::current_depth = 1  [static]
```

6.13.2.2 item_duration

```
map< string, float > logger::item_duration  [static]
```

6.13.2.3 item_last_start

```
map< string, high_resolution_clock::time_point > logger::item_last_start  [static]
```

the last time each item was started

6.13.2.4 logs

```
vector< log_entry > logger::logs  [static]
```

6.13.2.5 report

```
bool logger::report = true  [static]
```

if true, at the end of the program, a hierarchical report is printed

6.13.2.6 report_stream

```
ostream * logger::report_stream = &cout  [static]
```

the stream to report the final report. Default is cout

6.13.2.7 stat

```
bool logger::stat = false  [static]
```

if true, statistics will be displayed, e.g. number of star vertices / edges or the number of partition graphs etc

6.13.2.8 stat_stream

```
ostream * logger::stat_stream = &cout  [static]
```

the stream to report the statistics. Default is cout

6.13.2.9 verbose

```
bool logger::verbose = true  [static]
```

if true, every entry is printed at the time of its report, default is false

6.13.2.10 verbose_stream

```
ostream * logger::verbose_stream = &cout  [static]
```

the stream for printing entries at the time of arrival, default is cout

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

- logger.h
- logger.cpp

6.14 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< vector< int > > index_in_neighbor

  index_in_neighbor[v][i] is the index of vertex v in the adjacency list of the ith neighbor of v
- vector< int > ver_mark

  ver_mark[i] is the mark of vertex i

Friends

- bool operator== (const marked_graph &G1, const marked_graph &G2)

  checks whether two marked graphs are the same.
- bool operator!= (const marked_graph &G1, const marked_graph &G2)

  checks whether two marked graphs are not equal
- ostream & operator<< (ostream &o, const marked_graph &G)

  prints a marked graph to the output

6.14.1 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.

6.14.2 Constructor & Destructor Documentation

6.14.2.1 marked_graph() [1/2]

```
marked_graph::marked_graph ( )  [inline]
```

default constructor
```
00027     {
00028        nu_vertices = 0;
00029     }
```

6.14.2.2 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

```
00004 {
00005    nu_vertices = n;
00006    adj_list.resize(n);
00007    //adj_location.resize(n);
00008    index_in_neighbor.resize(n);
00009    // modify the edges if necessary so that in each element of the form ((i,j), (x, x')), we have i <
         j. This is important when forming adjacency lists so that the list of each vertex is sorted
00010    for (int i=0;i<edges.size();i++){
00011      if (edges[i].first.first > edges[i].first.second){
00012        // swap the edge endpoints and represent it in the other direction
00013        swap(edges[i].first.first, edges[i].first.second);
00014        // also, we should swap the mark components
00015        swap(edges[i].second.first, edges[i].second.second);
00016      }
00017    }
00018    sort(edges.begin(), edges.end()); // so that the adjacency list is sorted
00019    for (int k=0; k<edges.size(); k++){
00020      // (i,j) are endpoints if the edge
00021      // (x,y) are marks, x towards i and y towards j
00022      int i = edges[k].first.first;
00023      int j = edges[k].first.second;
00024      int x = edges[k].second.first;
00025      int y = edges[k].second.second;
00026      if (i <0 || i >= n || j < 0 || j >= n || i ==j)
00027        cerr << " ERROR: graph::graph(n, edges) received an invalid pair of edges with n = " << n << " : ("
         << i << " , " << j << " )" << endl;
00028      adj_list[i].push_back(pair<int, pair<int, int> > (j, pair<int, int> (x,y)));
00029      //adj_location[i][j] = adj_list[i].size() - 1;
00030      adj_list[j].push_back(pair<int, pair<int, int> > (i, pair<int, int> (y,x)));
00031      //adj_location[j][i] = adj_list[j].size() - 1;
00032      index_in_neighbor[i].push_back(adj_list[j].size()-1);
00033      index_in_neighbor[j].push_back(adj_list[i].size()-1);
00034    }
00035
00036
00037    ver_mark = vertex_marks;
00038 }
```

6.14.3 Friends And Related Symbol Documentation

6.14.3.1 operator"!=

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

checks whether two marked graphs are not equal

```
00088 {
00089    return !(G1 == G2);
00090 }
```

6.14.3.2 operator$<<$

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

prints a marked graph to the output

```
00099 {
00100    o << G.nu_vertices << endl;
00101    for (int v=0;v<G.nu_vertices;v++){
```

```
00102       o « G.ver_mark[v];
00103       if (v < G.nu_vertices-1)
00104         o « " ";
00105    }
00106    o « endl;
00107
00108    vector<pair<pair<int, int>, pair<int, int> > > edges;
00109    pair<pair<int, int> , pair<int, int> > edge; // the current edge to be added to the list
00110    for (int v=0;v<G.nu_vertices;v++){
00111      for (int i=0;i<G.adj_list[v].size();i++){
00112        if (G.adj_list[v][i].first > v){ // avoid duplicate in edge list, only add edges where the other
    endpoint has a greater index
00113          edge.first.first = v;
00114          edge.first.second = G.adj_list[v][i].first;
00115          edge.second = G.adj_list[v][i].second;
00116          edges.push_back(edge);
00117        }
00118      }
00119    }
00120    sort(edges.begin(), edges.end());
00121    o « edges.size() « endl;
00122    for(int i=0;i<edges.size();i++){
00123      o « edges[i].first.first « " " « edges[i].first.second « " " « edges[i].second.first « " " «
    edges[i].second.second « endl;
00124    }
00125    return o;
00126    /*
00127    o « " number of vertices " « G.nu_vertices « endl;
00128    vector<pair<int, pair<int, int> > > l; // the adjacency list of a vertex
00129    for (int v=0; v<G.nu_vertices; v++){
00130      o « " vertex " « v « " mark " « G.ver_mark[v] « endl;
00131      //o « " adj list (connections to vertices with greater index): format (j, (x,y))" « endl;
00132      o « " adj list " « endl;
00133      l = G.adj_list[v];
00134      sort(l.begin(), l.end(), edge_compare);
00135      for (int i=0;i<l.size();i++){
00136        if (l[i].first > v)
00137          o « " (" « l[i].first « ", (" « l[i].second.first « ", " « l[i].second.second « ")) ";
00138      }
00139      o « endl « endl;
00140    }
00141    return o;
00142    */
00143 }
```

6.14.3.3 operator==

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

checks whether two marked graphs are the same.

two marked graphs are said to be the same if: 1) they have the same number of vertices, 2) vertex marks match and 3) each vertex has the same set of neighbors with matching marks.

```
00066 {
00067    if (G1.nu_vertices != G2.nu_vertices)
00068      return false;
00069    return G1.adj_list == G2.adj_list;
00070    int n = G1.nu_vertices; // number of vertices of the two graphs
00071    vector< pair< int, pair< int, int > > > l1, l2; // the adjacency list of a vertex in two graphs for
    comparison.
00072    for (int v=0;v<n;v++){
00073      if (G1.ver_mark[v] != G2.ver_mark[v]) // mark of each vertex should be the same
00074        return false;
00075      if (G1.adj_list[v].size() != G2.adj_list[v].size()) // each vertex must have the same degree in
    two graphs
00076        return false;
00077      l1 = G1.adj_list[v];
00078      l2 = G2.adj_list[v];
00079      sort(l1.begin(), l1.end(), edge_compare); // sort with respect to the other endpoint
00080      sort(l2.begin(), l2.end(), edge_compare);
00081      if (l1 != l2) // after sorting, the lists must match
00082        return false;
00083    }
00084    return true;
00085 }
```

6.14.4 Member Data Documentation

6.14.4.1 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)

6.14.4.2 index_in_neighbor

```
vector<vector<int> > marked_graph::index_in_neighbor
```

index_in_neighbor[v][i] is the index of vertex v in the adjacency list of the ith neighbor of v

6.14.4.3 nu_vertices

```
int marked_graph::nu_vertices
```

number of vertices in the graph

6.14.4.4 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

6.15 marked_graph_compressed Class Reference

```
#include <marked_graph_compression.h>
```

Public Member Functions

- void clear ()
- void binary_write (FILE ∗f)

    writes the compressed data to a binary file
- void binary_write (string s)
- void binary_read (FILE ∗f)

    read the compressed data from a binary file
- void binary_read (string s)
- int vtype_max_match (int i, int j)

    finds the maximum number of (t,t',n) blocks that match between two entries of ver_type_list
- void vtype_list_write (obitstream &oup)

    writes the ver_type_list array to the output using difference coding, assuming that its entries are lexicographically sorted (as vectors)
- void vtype_list_read (ibitstream &inp)

    reads the ver_type_list array from the input, assuming the bit sequence was generated during compression using vtype_list_write
- void vtype_block_write (obitstream &oup, int i, int j)
- void vtype_block_write (obitstream &oup, int i, int j, int ir, int jr)
- void vtype_block_read (ibitstream &inp, int i, int j)
- void vtype_block_read (ibitstream &inp, int i, int j, int ir, int jr)

Public Attributes

- int n

    the number of vertices

- int h

    the depth up to which the compression was performed

- int delta

    the degree threshold used when compression was performed

- pair< vector< int >, mpz_class > star_vertices

    the compressed form of the star_vertices list

- map< pair< int, int >, vector< vector< int > > > star_edges

    for each pair of edge marks x,x', and integer k, star_edges[pair<int,int>(x,x')][k] is a list of neighbors w of the kth star vertex (say v) so that v shares a star edge with w so that the mark towards v is x and the mark towards w is xp.

- vector< int > type_mark

    for an edge type t, type_mark[t] denotes the mark component of t

- vector< vector< int > > ver_type_list

    the list of all vertex types that appear in the graph, where the type of a vertex is a vector of integers, where its index 0 is the mark of the vertex, and indices $3k+1, 3k+2, 3k+3$ are m, m' and $n_{m,m'}$, where (m, m') is a type pair, and $n_{m,m'}$ is the number of edges connected to the vertex with that type. The list is sorted lexicographically to ensure unique representation.

- pair< vector< int >, mpz_class > ver_types

    the compressed form of vertex types, where the type of a vertex is the index with respect to ver_type_list of the list of integers specifying the type of the vertex (mark of the vertex followed by the number of edges of each type connected to that vertex)

- map< pair< int, int >, mpz_class > part_bgraph

    compressed form of partition bipartite graphs corresponding to colors in $C_<$. For a pair $0 \le t < t' < L$ of half edge types, part_bgraph[pair<int, int>(t,t')] is the compressed form of the bipartite graph with n left and right nodes, where a left node i is connected to a right node j if there is an edge connecting i to j with type t towards i and type t' towards j

- map< int, pair< mpz_class, vector< int > > > part_graph

    compressed form of partition graphs corresponding to colors in $C_=$. For a half edge type t, part_graph[t] is the compressed form of the simple unmarked graph with n vertices, where a node i is connected to a node j where there is an edge between i and j in the original graph with color (t,t)

6.15.1 Member Function Documentation

6.15.1.1 binary_read() [1/2]

```
void marked_graph_compressed::binary_read (
            FILE * f )
```

read the compressed data from a binary file

Parameters

f	a FILE∗ object which is the address of the binary file to write

```
00628                                                    {
00629    clear(); // to make sure nothing is stored inside me before reading
00630
00631    // ==== read n, h, delta
00632    fread(&n, sizeof n, 1, f);
00633    fread(&h, sizeof h, 1, f);
00634    fread(&delta, sizeof delta, 1, f);
00635
00636    int int_in; // auxiliary input integer
00637    // ===== read type_mark
```

```
00638    // read number of types
00639    fread(&int_in, sizeof int_in, 1, f);
00640    type_mark.resize(int_in);
00641    for (int i=0;i<type_mark.size();i++){
00642      fread(&int_in, sizeof int_in, 1, f);
00643      type_mark[i] = int_in;
00644    }
00645
00646    // ==== read star_vertices
00647    // first, read the frequency.
00648    star_vertices.first = vector<int>(2); // frequency,
00649    // we read its first index which is number of zeros, and the second is n - the first.
00650    fread(&int_in, sizeof int_in, 1, f);
00651    star_vertices.first[0] = int_in;
00652    star_vertices.first[1] = n - int_in;
00653
00654    // the integer representation which is star_vertices.second
00655    mpz_inp_raw(star_vertices.second.get_mpz_t(), f);
00656
00657    // ==== read star_edges
00658
00659    int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n),
       which is the number of bits to encode vertices
00660    int n_copy = n;
00661    while(n_copy > 0){
00662      n_copy >>= 1;
00663      log2n ++;
00664    }
00665    bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index
00666
00667    string s;
00668    stringstream ss;
00669    int sp; // the index of the string s we are studying
00670
00671    // read the size of star_edges
00672
00673    int star_edges_size;
00674    fread(&star_edges_size, sizeof star_edges_size, 1, f);
00675
00676    int x, xp; // edge marks
00677    int nu_star_vertices = star_vertices.first[1];
00678
00679    vector<vector<int> > V; // the list of star edges corresponding to each mark pair
00680    V.resize(nu_star_vertices);
00681
00682    for (int i=0;i<star_edges_size;i++){
00683      fread(&x, sizeof x, 1, f);
00684      fread(&xp, sizeof xp, 1, f);
00685
00686      s = bit_string_read(f);
00687      //cerr << " read  x " << x << " xp " << xp << " s " << s << endl;
00688      sp = 0; // starting from zero
00689      for (int j=0; j<nu_star_vertices; j++){ //
00690        V[j].clear(); // make it fresh
00691        while(s[sp++] == '1'){ // there is still some edge connected to this vertex
00692          // read log2n many bits
00693          //cerr << " s subtr " << s.substr(sp, log2n);
00694          //ss << s.substr(sp, log2n);
00695          B = bitset<8*sizeof(int)>(s.substr(sp, log2n));
00696          //cerr << " ss " << ss.str() << endl;
00697          sp += log2n;
00698          //ss >> B;
00699
00700          V[j].push_back(B.to_ulong());
00701        }
00702        //for (int k=0;k<V[j].size();k++)
00703        //  cerr << " , " << V[j][k];
00704        //cerr << endl;
00705      }
00706
00707
00708      star_edges.insert(pair< pair<int, int> , vector<vector<int> > > (pair<int, int>(x, xp), V));
00709    }
00710
00711    // ==== read vertex_types
00712
00713    // read ver_type_list
00714    fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list
00715    ver_type_list.resize(int_in);
00716    for (int i=0; i<ver_type_list.size();i++){
00717      fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list[i]
00718      ver_type_list[i].resize(int_in);
00719      for (int j=0;j<ver_type_list[i].size();j++){
00720        fread(&int_in, sizeof int_in, 1, f);
00721        ver_type_list[i][j] = int_in;
00722      }
00723    }
```

```
00724
00725    // ver_types
00726    // ver_types.first
00727    // ver_types.first.size()
00728    fread(&int_in, sizeof int_in, 1, f);
00729    ver_types.first.resize(int_in);
00730    for (int i=0;i<ver_types.first.size();i++){
00731      fread(&int_in, sizeof int_in, 1, f);
00732      ver_types.first[i] = int_in;
00733    }
00734    // ver_types.second
00735    mpz_inp_raw(ver_types.second.get_mpz_t(), f);
00736
00737
00738    // === part bgraphs
00739    int part_bgraph_size;
00740    int t, tp;
00741    pair<int, int> type;
00742    mpz_class part_g;
00743    fread(&part_bgraph_size, sizeof part_bgraph_size, 1, f);
00744    for (int i=0;i<part_bgraph_size;i++){
00745      // read t, t'
00746      fread(&t, sizeof t, 1, f);
00747      fread(&tp, sizeof tp, 1, f);
00748      type = pair<int, int>(t, tp);
00749      mpz_inp_raw(part_g.get_mpz_t(), f);
00750      part_bgraph.insert(pair<pair<int, int>, mpz_class> (type, part_g));
00751    }
00752
00753    // === part graphs
00754
00755    // first, the size
00756    int part_graph_size;
00757    int v_size;
00758    vector<int> W;
00759    fread(&part_graph_size, sizeof part_graph_size, 1, f);
00760    for (int i=0;i<part_graph_size; i++){
00761      // first, the type
00762      fread(&t, sizeof t, 1, f);
00763      // then, the mpz part
00764      mpz_inp_raw(part_g.get_mpz_t(), f);
00765      // then, the vector size
00766      fread(&v_size, sizeof v_size, 1, f);
00767      W.resize(v_size);
00768      for (int j=0;j<v_size; j++){
00769        fread(&int_in, sizeof int_in, 1, f);
00770        W[j] = int_in;
00771      }
00772      part_graph.insert(pair<int, pair< mpz_class, vector< int > > >(t, pair<mpz_class, vector<int>
      >(part_g, W)));
00773    }
00774 }
```

6.15.1.2   binary_read() [2/2]

```
void marked_graph_compressed::binary_read (
            string s )
```

read the compressed data from a binary file

Parameters

s	string containing the name of the binary file

```
00777                                                             {
00778    logger::current_depth++;
00779    clear(); // to make sure nothing is stored inside me before reading
00780    ibitstream inp(s);
00781
00782    // ==== read n, h, delta
00783    unsigned int int_in; // auxiliary input integer
00784    logger::add_entry("n", "");
00785    inp » int_in;
00786    n = int_in; // I need to do this, since ibitstream::operator » gets unsigned int& and the compile
      can not cast int& to unsigned int&
00787    logger::add_entry("h", "");
00788    inp » int_in;
00789    h = int_in;
```

```
00790    logger::add_entry("delta", "");
00791    inp » int_in;
00792    delta = int_in;
00793
00794    //fread(&n, sizeof n, 1, f);
00795    //fread(&h, sizeof h, 1, f);
00796    //fread(&delta, sizeof delta, 1, f);
00797
00798    logger::add_entry("type_mark", "");
00799    // ===== read type_mark
00800    // read number of types
00801    inp » int_in;
00802    //fread(&int_in, sizeof int_in, 1, f);
00803    type_mark.resize(int_in);
00804    for (int i=0;i<type_mark.size();i++){
00805      inp » int_in;
00806      type_mark[i] = int_in;
00807      //fread(&int_in, sizeof int_in, 1, f);
00808      //type_mark[i] = int_in;
00809    }
00810
00811    logger::add_entry("star vertices", "");
00812    // ==== read star_vertices
00813    // first, read the frequency.
00814    star_vertices.first = vector<int>(2); // frequency,
00815    // we read its first index which is number of zeros, and the second is n - the first.
00816    inp » int_in;
00817    //fread(&int_in, sizeof int_in, 1, f);
00818    star_vertices.first[0] = int_in;
00819    star_vertices.first[1] = n - int_in;
00820
00821    // the integer representation which is star_vertices.second
00822    inp » star_vertices.second;
00823    //mpz_inp_raw(star_vertices.second.get_mpz_t(), f);
00824
00825    logger::add_entry("star edges", "");
00826
00827    // ==== read star_edges
00828
00829    //int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n),
        which is the number of bits to encode vertices
00830    //int n_copy = n;
00831    //while(n_copy > 0){
00832    //n_copy »= 1;
00833    //log2n ++;
00834    //}
00835    //bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index
00836
00837    //string s;
00838    //stringstream ss;
00839    //int sp; // the index of the string s we are studying
00840
00841    // read the size of star_edges
00842
00843    unsigned int star_edges_size;
00844    inp » star_edges_size;
00845    //cerr « " star edges size " « star_edges_size « endl;
00846    //fread(&star_edges_size, sizeof star_edges_size, 1, f);
00847
00848    unsigned int x, xp; // edge marks
00849    int nu_star_vertices = star_vertices.first[1];
00850    unsigned int nb_nsv = nu_bits(nu_star_vertices-1); // defined to chose compression method, see below
00851    unsigned int nb_nb_nsv = nu_bits(nb_nsv); // defined to chose compression method, see below
00852    unsigned int diff_threshold = nb_nsv - nb_nb_nsv;
00853    unsigned int nb_diff; // number of bits in diff
00854
00855
00856    vector<vector<int> > V; // the list of star edges corresponding to each mark pair
00857    V.resize(nu_star_vertices);
00858
00859    unsigned int n_bits = nu_bits(n); // the number of bits in n, i.e. \f$1 + \lfloor \log_2 n
        \rfloor\f$
00860    unsigned int diff; // to decode differences between star vertex indices in star edges
00861
00862    for (int i=0;i<star_edges_size;i++){
00863
00864      inp » x;
00865      inp » xp;
00866      //cerr « " x  " « x « " xp " « xp « " i = " « i « endl;
00867      //fread(&x, sizeof x, 1, f);
00868      //fread(&xp, sizeof xp, 1, f);
00869
00870      //s = bit_string_read(f);
00871      //cerr « " read  x " « x « " xp " « xp « " s " « s « endl;
00872      //sp = 0; // starting from zero
00873      for (int j=0; j<nu_star_vertices; j++){ //
00874        V[j].clear(); // make it fresh
```

```
00875        //inp.bin_inter_decode(V[j], n_bits); // use binary interpolative decoding
00876        //cerr « "  jth star vertex j = " « j« endl;
00877        inp » int_in; // the number of star edges connected to jth star vertex
00878        //cerr « " number of star edges " « int_in « endl;
00879        for (int k=0; k<int_in; k++){
00880          // read diff
00881          //cerr « " k " « k « endl;
00882          if (inp.read_bit()){ // the flag bit is one, we have used the first method
00883            nb_diff = inp.read_bits(nb_nb_nsv);
00884            diff = 0; // initialize
00885            if (nb_diff > 1) // otherwise, reading zero bits is as iff diff = 0 (before adding the
      leading one) this is important when real diff is 1
00886              diff = inp.read_bits(nb_diff-1);
00887            diff += (1«(nb_diff-1)); // bring the lading bit in diff back
00888          }else{ // use the second method to read diff
00889            diff = inp.read_bits(nb_nsv);
00890          }
00891          //if (int_in < 100)
00892          //  cerr « diff « " ";
00893          //cerr « " diff " « diff « endl;
00894          if(k==0)
00895            V[j].push_back(j+diff);
00896          else
00897            V[j].push_back(V[j][k-1]+diff);
00898        }
00899        //if (int_in < 100)
00900        //  cerr « endl;
00901      }
00902
00903
00904      star_edges.insert(pair< pair<int, int> , vector<vector<int> > > (pair<int, int>(x, xp), V));
00905    }
00906
00907    logger::add_entry("vertex types", "");
00908    // ==== read vertex_types
00909
00910    vtype_list_read(inp);
00911
00912    // // read ver_type_list
00913    // inp » int_in;
00914    // //fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list
00915    // ver_type_list.resize(int_in);
00916    // for (int i=0; i<ver_type_list.size();i++){
00917    //   inp » int_in;
00918    //   //fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list[i]
00919    //   ver_type_list[i].resize(int_in);
00920
00921    //   inp » int_in;
00922    //   ver_type_list[i][0] = int_in; // read the vertex mark part
00923
00924    //   for (int j=0;j<((ver_type_list[i].size()-1)/3); j++){
00925    //     // the triple is 3j+1, 3j+2, 3j+3
00926    //     if (j==0){
00927    //       // the first chunk is written of the form t,t',n-1. The reason we write n-1 is because we
      know n > 0, so it is better to save some bits!
00928    //       inp » int_in;
00929    //       ver_type_list[i][3*j+1] = int_in;
00930    //       inp » int_in;
00931    //       ver_type_list[i][3*j+2] = int_in;
00932    //       inp » int_in;
00933    //       ver_type_list[i][3*j+3] = int_in + 1; // since we had subtracted one during compression
00934    //     }else{
00935    //       // we know that the list is lexicographically ordered, so the t here is not less than the t
      is the previous chunk, so better to write their difference
00936    //       inp » int_in;
00937    //       ver_type_list[i][3*j+1] = int_in + ver_type_list[i][3*(j-1)+1]; // since we had encoded the
      difference during the compression phasen
00938
00939    //       // if t_here is equal to t_previous, then t'_here >= t'_previous, so better to encode their
      difference!
00940    //       if (ver_type_list[i][3*j+1] == ver_type_list[i][3*(j-1)+1]){
00941    //         inp » int_in;
00942    //         ver_type_list[i][3*j+2] = int_in + ver_type_list[i][3*(j-1)+2];
00943    //       }else{
00944    //         // otherwise, just write t'_here
00945    //         inp » int_in;
00946    //         ver_type_list[i][3*j+2] = int_in;
00947    //       }
00948    //       inp » int_in;
00949    //       ver_type_list[i][3*j+3]= int_in + 1; // since we had subtracted one during compression
00950    //     }
00951    //   }
00952
00953    //   /* the old way of reading the list
00954    //   for (int j=0;j<ver_type_list[i].size();j++){
00955    //     //fread(&int_in, sizeof int_in, 1, f);
00956    //     inp » int_in;
```

```
00957   //     if (j%3 == 0 and j > 0) // we know that the cont part is positive, so no need to add one
        during compression. So during compression, we subtract one to make it nonnegative
00958   //       int_in ++;
00959   //     ver_type_list[i][j] = int_in;
00960   //   }
00961   //   */
00962   // }
00963
00964   // ver_types
00965   // ver_types.first
00966   // ver_types.first.size()
00967   inp » int_in;
00968   //fread(&int_in, sizeof int_in, 1, f);
00969   ver_types.first.resize(int_in);
00970   for (int i=0;i<ver_types.first.size();i++){
00971     //fread(&int_in, sizeof int_in, 1, f);
00972     inp » int_in;
00973     ver_types.first[i] = int_in;// = int_in;
00974   }
00975   // ver_types.second
00976   inp » ver_types.second;
00977   //mpz_inp_raw(ver_types.second.get_mpz_t(), f);
00978
00979
00980   logger::add_entry("partition bipartite graphs", "");
00981   // === part bgraphs
00982   unsigned int part_bgraph_size;
00983   unsigned int t, tp;
00984   pair<int, int> type;
00985   mpz_class part_g;
00986   inp » part_bgraph_size;
00987   //fread(&part_bgraph_size, sizeof part_bgraph_size, 1, f);
00988   for (int i=0;i<part_bgraph_size;i++){
00989     // read t, t'
00990     inp » t;
00991     inp » tp;
00992     //fread(&t, sizeof t, 1, f);
00993     //fread(&tp, sizeof tp, 1, f);
00994     type = pair<int, int>(t, tp);
00995     inp » part_g;
00996     //mpz_inp_raw(part_g.get_mpz_t(), f);
00997     part_bgraph.insert(pair<pair<int, int>, mpz_class> (type, part_g));
00998   }
00999
01000   logger::add_entry("partition graphs", "");
01001   // === part graphs
01002
01003   // first, the size
01004   unsigned int part_graph_size;
01005   unsigned int v_size;
01006   vector<int> W;
01007   inp » part_graph_size;
01008   //fread(&part_graph_size, sizeof part_graph_size, 1, f);
01009   for (int i=0;i<part_graph_size; i++){
01010     // first, the type
01011     inp » t;
01012     //fread(&t, sizeof t, 1, f);
01013     // then, the mpz part
01014     inp » part_g;
01015     //mpz_inp_raw(part_g.get_mpz_t(), f);
01016     // then, the vector size
01017     inp » v_size;
01018     //fread(&v_size, sizeof v_size, 1, f);
01019     W.resize(v_size);
01020     for (int j=0;j<v_size; j++){
01021       //fread(&int_in, sizeof int_in, 1, f);
01022       inp » int_in;
01023       W[j] = int_in; //= int_in;
01024     }
01025     part_graph.insert(pair<int, pair< mpz_class, vector< int > > >(t, pair<mpz_class, vector<int>
      >(part_g, W)));
01026   }
01027   inp.close();
01028   logger::current_depth--;
01029 }
```

6.15.1.3 binary_write() [1/2]

```
void marked_graph_compressed::binary_write (
            FILE * f )
```

writes the compressed data to a binary file

Parameters

f	a `FILE*` object which is the address of the binary file to write

```
00020                                                                          {
00021
00022    vector<pair<string, int> > space_log; // stores the number of bits used to store each category. The
         string part is description of the category, and the int part is the number of bits of output used to
         express that part.
00023
00024    int output_bits; // the number of bits in the output corresponding to the current category under
         investigation, to be zeroed at each step.
00025
00026    logger::current_depth++;
00027    // ==== write n, h, delta
00028    output_bits = 0;
00029    logger::add_entry("n", "");
00030    fwrite(&n, sizeof n, 1, f);
00031    output_bits += sizeof n;
00032
00033    logger::add_entry("h", "");
00034    fwrite(&h, sizeof h, 1, f);
00035    output_bits += sizeof h;
00036
00037    logger::add_entry("delta", "");
00038    fwrite(&delta, sizeof delta, 1, f);
00039    output_bits += sizeof delta;
00040
00041    space_log.push_back(pair<string, int> ("n, h, delta", output_bits));
00042
00043    logger::add_entry("type_mark", "");
00044    output_bits = 0;
00045
00046    int int_out; // auxiliary variable, an integer value to be written to output
00047    // ==== write type_mark
00048    // first, the number of types
00049    int_out = type_mark.size();
00050    fwrite(&int_out, sizeof int_out, 1, f);
00051    output_bits += sizeof int_out;
00052    // then, marks one by one
00053    for (int i=0;i<type_mark.size();i++){
00054      int_out = type_mark[i];
00055      fwrite(&int_out, sizeof int_out, 1, f);
00056      output_bits += sizeof int_out;
00057    }
00058
00059    space_log.push_back(pair<string, int>("type mark", output_bits));
00060
00061    logger::add_entry("star_vertices", "");
00062    output_bits = 0;
00063    // ==== write star vertices
00064    // first, write the frequency, note that star_vertices.first is a vector of size 2 with the first
         entry being the number of zeros, and the second one the number of ones, so it enough to write only one
         of them
00065    int_out = star_vertices.first[0];
00066    fwrite(&int_out, sizeof int_out, 1, f);
00067    output_bits += sizeof int_out;
00068
00069    // then, we write the integer representation star_vertices.second
00070    output_bits +=  mpz_out_raw(f, star_vertices.second.get_mpz_t()); // mpz_out_raw returns the number
         of bytes written to the output
00071
00072    space_log.push_back(pair<string, int> ("star vertices", output_bits));
00073
00074    logger::add_entry("star_edges", "");
00075    // ==== write star edges
00076    output_bits = 0;
00077    int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n),
         which is the number of bits to encode vertices
00078    int n_copy = n;
00079    while(n_copy > 0){
00080      n_copy >>= 1;
00081      log2n ++;
00082    }
00083    //cerr << " log2n " << log2n << endl;
00084    bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index
00085
00086    map<pair<int, int>, vector<vector<int> > >::iterator it;
00087    int x, xp;
00088    string s; // the bit stream
00089
00090    // first, write the size of star_edges so that the decoder knows how many blocks are coming
00091    int_out = star_edges.size();
00092    fwrite(&int_out, sizeof int_out, 1, f);
00093    output_bits += sizeof int_out;
00094
```

```
00095    int nu_star_edges = 0; // number of star edges
00096    for (it = star_edges.begin(); it!= star_edges.end(); it++){
00097      x = it->first.first;
00098      xp = it->first.second;
00099      //write x and xp
00100      fwrite(&x, sizeof x, 1, f);
00101      fwrite(&xp, sizeof xp, 1, f);
00102      output_bits += sizeof x;
00103      output_bits += sizeof xp;
00104      s = "";
00105      for (int i=0;i<it->second.size();i++){
00106        for(int j=0;j<it->second[i].size();j++){
00107          s += "1";
00108          B = it->second[i][j]; // convert the index of the other endpoint to binary
00109          s += B.to_string().substr(8*sizeof(int) - log2n, log2n); // take only log2n many bits of the
       representation (and this should be taken from the least significant bits)
00110          nu_star_edges ++;
00111        }
00112        s += "0"; // to indicate that the neighbor list of this vertex is over now
00113      }
00114      //cerr << " write  x " << x << " xp " << xp << " s " << s << endl;
00115      //for (int i=0;i<it->second.size();i++){
00116      //   for (int j=0;j<it->second[i].size();j++){
00117      //     cerr << " , " << it->second[i][j];
00118      //   }
00119      //   cerr << endl;
00120      //}
00121      output_bits += bit_string_write(f, s); // write this bitstream to the output
00122    }
00123
00124    space_log.push_back(pair<string, int> ("star edges", output_bits));
00125
00126
00127    logger::add_entry("vertex types", "");
00128    output_bits = 0;
00129
00130    // ==== write vertex types
00131
00132    // first, we need vertex types list (ver_type_list)
00133    // size of ver_type_list
00134    int_out = ver_type_list.size();
00135    fwrite(&int_out, sizeof int_out, 1, f);
00136    output_bits += sizeof int_out;
00137
00138    for (int i=0;i<ver_type_list.size();i++){
00139      int_out = ver_type_list[i].size();
00140      fwrite(&int_out, sizeof int_out, 1, f);
00141      output_bits += sizeof int_out;
00142
00143      for (int j=0;j<ver_type_list[i].size();j++){
00144        int_out = ver_type_list[i][j];
00145        fwrite(&int_out, sizeof int_out, 1, f);
00146        output_bits += sizeof int_out;
00147      }
00148    }
00149    space_log.push_back(pair<string, int>("vertex type list", output_bits));
00150    output_bits = 0;
00151
00152    // then, write ver_types
00153
00154    // ver_types.first
00155    // ver_types.first.size():
00156    int_out = ver_types.first.size();
00157    fwrite(&int_out, sizeof int_out, 1, f);
00158    output_bits += sizeof int_out;
00159
00160    for (int i =0;i<ver_types.first.size(); i++){
00161      int_out = ver_types.first[i];
00162      fwrite(&int_out, sizeof int_out, 1, f);
00163      output_bits += sizeof int_out;
00164    }
00165    // ver_types.second
00166    output_bits += mpz_out_raw(f, ver_types.second.get_mpz_t());
00167
00168    space_log.push_back(pair<string, int> ("vertex types", output_bits));
00169
00170    logger::add_entry("partition bipartite graphs", "");
00171
00172
00173    // ==== part bgraphs
00174    output_bits = 0;
00175
00176    // part_bgraphs.size
00177    int_out = part_bgraph.size();
00178    fwrite(&int_out, sizeof int_out, 1, f);
00179    output_bits += sizeof int_out;
00180
```

```
00181   map<pair<int, int>, mpz_class>::iterator it2;
00182   if (logger::stat){
00183     *logger::stat_stream « " ==== statistics ==== " « endl;
00184     *logger::stat_stream « " n:                 " « n « endl;
00185     *logger::stat_stream « " h:                 " « h « endl;
00186     *logger::stat_stream « " delta:             " « delta « endl;
00187     *logger::stat_stream « " No. edge types     " « type_mark.size() « endl;
00188     *logger::stat_stream « " No. vertex types   " « ver_type_list.size() « endl;
00189     *logger::stat_stream « " No. * vertices     " « n - star_vertices.first[0] « endl;
00190     *logger::stat_stream « " No. * edges        " « nu_star_edges « endl;
00191     *logger::stat_stream « " No. part bgraphs   " « part_bgraph.size() « endl;
00192     *logger::stat_stream « " No. part graphs    " « part_graph.size() « endl;
00193   }
00194
00195   for (it2 = part_bgraph.begin(); it2 != part_bgraph.end(); it2++){
00196     // first, write t, t'
00197     int_out = it2->first.first;
00198     fwrite(&int_out, sizeof int_out, 1, f);
00199     output_bits += sizeof int_out;
00200     int_out = it2->first.second;
00201     fwrite(&int_out, sizeof int_out, 1, f);
00202     output_bits += sizeof int_out;
00203     // then, the compressed integer
00204     output_bits += mpz_out_raw(f, it2->second.get_mpz_t());
00205   }
00206
00207   space_log.push_back(pair<string, int> ("partition bipartite graphs", output_bits));
00208
00209   logger::add_entry("partition graphs", "");
00210   output_bits = 0;
00211   // === part graphs
00212
00213   // part_graph.size
00214   int_out = part_graph.size();
00215   fwrite(&int_out, sizeof int_out, 1, f);
00216   output_bits += sizeof int_out;
00217
00218   map< int, pair< mpz_class, vector< int > > >::iterator it3;
00219   for (it3 = part_graph.begin(); it3 != part_graph.end(); it3++){
00220     int_out = it3->first; // the type
00221     fwrite(&int_out, sizeof int_out, 1, f);
00222     output_bits += sizeof int_out;
00223
00224     // the mpz part
00225     output_bits += mpz_out_raw(f, it3->second.first.get_mpz_t());
00226     // the vector part
00227     // first its size
00228     int_out = it3->second.second.size();
00229     fwrite(&int_out, sizeof int_out, 1, f);
00230     output_bits += sizeof int_out;
00231     // then element by element
00232     for(int j=0;j<it3->second.second.size();j++){
00233       int_out = it3->second.second[j];
00234       fwrite(&int_out, sizeof int_out, 1, f);
00235       output_bits += sizeof int_out;
00236     }
00237   }
00238   space_log.push_back(pair<string, int>("partition graphs", output_bits));
00239
00240
00241   if (logger::stat){
00242     *logger::stat_stream « endl « endl;
00243     *logger::stat_stream « " Number of bytes used for each part " « endl;
00244     *logger::stat_stream « " -------------------------------- " « endl « endl;
00245
00246     int total_bytes = 0;
00247     for (int i=0; i < space_log.size(); i++)
00248       total_bytes += space_log[i].second;
00249
00250     for (int i=0; i < space_log.size(); i++){
00251       *logger::stat_stream « space_log[i].first « " -> "  « space_log[i].second « " ( " « float(100) *
    float(space_log[i].second) / float(total_bytes) « " % " « endl;
00252     }
00253
00254     *logger::stat_stream « " Total number of bytes wrote to the output = " « total_bytes « endl;
00255   }
00256   logger::current_depth--;
00257 }
```

6.15.1.4 binary_write() [2/2]

```
void marked_graph_compressed::binary_write (
              string s )
```

writes the compressed data to a binary file

Parameters

s	string containing the name of the binary file

```
00261                                                                     {
00262
00263    obitstream oup(s);
00264
00265    vector<pair<string, int> > space_log; // stores the number of bits used to store each category. The
         string part is description of the category, and the int part is the number of bits of output used to
         express that part.
00266
00267    //int output_bits; // the number of bits in the output corresponding to the current category under
         investigation, to be zeroed at each step.
00268    unsigned int chunks = 0; // number of chunks written to the output. Each chunk is sizeof(unsigned
         int) = 32 bits long
00269    unsigned int chunks_new =0; // to take the difference in each step
00270
00271    logger::current_depth++;
00272    // ==== write n, h, delta
00273    //output_bits = 0;
00274    logger::add_entry("n", "");
00275    oup « n; //fwrite(&n, sizeof n, 1, f);
00276    //output_bits += sizeof n;
00277
00278    logger::add_entry("h", "");
00279    oup « h; //fwrite(&h, sizeof h, 1, f);
00280    //output_bits += sizeof h;
00281
00282    logger::add_entry("delta", "");
00283    oup « delta; //fwrite(&delta, sizeof delta, 1, f);
00284    //output_bits += sizeof delta;
00285
00286    chunks_new = oup.chunks();
00287    space_log.push_back(pair<string, int> ("n, h, delta", chunks_new – chunks));
00288    chunks = chunks_new;
00289
00290    logger::add_entry("type_mark", "");
00291    //output_bits = 0;
00292
00293    int int_out; // auxiliary variable, an integer value to be written to output
00294    // ==== write type_mark
00295    // first, the number of types
00296    oup «  type_mark.size();
00297    //fwrite(&int_out, sizeof int_out, 1, f);
00298    //output_bits += sizeof int_out;
00299    // then, marks one by one
00300    for (int i=0;i<type_mark.size();i++){
00301      oup «  type_mark[i];
00302      //fwrite(&int_out, sizeof int_out, 1, f);
00303      //output_bits += sizeof int_out;
00304    }
00305
00306    chunks_new = oup.chunks();
00307    space_log.push_back(pair<string, int>("type mark", chunks_new – chunks));
00308    chunks = chunks_new;
00309
00310    logger::add_entry("star_vertices", "");
00311    //output_bits = 0;
00312    // ==== write star vertices
00313    // first, write the frequency, note that star_vertices.first is a vector of size 2 with the first
         entry being the number of zeros, and the second one the number of ones, so it enough to write only one
         of them
00314    oup « star_vertices.first[0];
00315    //fwrite(&int_out, sizeof int_out, 1, f);
00316    //output_bits += sizeof int_out;
00317
00318    // then, we write the integer representation star_vertices.second
00319    oup « star_vertices.second;
00320    //output_bits +=  mpz_out_raw(f, star_vertices.second.get_mpz_t()); // mpz_out_raw returns the
         number of bytes written to the output
00321
00322    chunks_new = oup.chunks();
00323    space_log.push_back(pair<string, int> ("star vertices", chunks_new – chunks));
00324    chunks = oup.chunks();
00325
00326    logger::add_entry("star_edges", "");
00327    // ==== write star edges
00328    //output_bits = 0;
00329    //int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n),
         which is the number of bits to encode vertices
00330    //int n_copy = n;
00331    //while(n_copy > 0){
```

```
00332    //n_copy »= 1;
00333    //log2n ++;
00334    //}
00335    //cerr « " log2n " « log2n « endl;
00336    //bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index
00337
00338    map<pair<int, int>, vector<vector<int> > >::iterator it;
00339    int x, xp;
00340    //string s; // the bit stream
00341
00342    // first, write the size of star_edges so that the decoder knows how many blocks are coming
00343    oup « star_edges.size();
00344    //cerr « " star edges size " « star_edges.size() « endl;
00345    //fwrite(&int_out, sizeof int_out, 1, f);
00346    //output_bits += sizeof int_out;
00347
00348    int nu_star_edges = 0; // number of star edges
00349    unsigned int n_bits = nu_bits(n); // the number of bits in n, i.e. \f$ 1 + \lfloor \log_2 n
         \rfloor\f$
00350    unsigned int diff; // the difference for differential coding
00351    unsigned int nu_star_vertices = n - star_vertices.first[0]; // number of star vertices
00352    unsigned int nb_nsv = nu_bits(nu_star_vertices-1); // defined to chose compression method, see below
00353    unsigned int nb_nb_nsv = nu_bits(nb_nsv); // defined to chose compression method, see below
00354    unsigned int diff_threshold = nb_nsv - nb_nb_nsv;
00355    unsigned int nb_diff; // number of bits in diff
00356
00357    map<int, int> deg_map;
00358
00359    if (logger::stat){
00360      *logger::stat_stream « " * degree stat: " « endl;
00361      *logger::stat_stream « " -------------- " « endl;
00362    }
00363
00364    for (it = star_edges.begin(); it!= star_edges.end(); it++){
00365
00366      x = it->first.first;
00367      xp = it->first.second;
00368      //cerr « " x " « x « " xp " « xp « endl;
00369      //write x and xp
00370
00371      oup « x;
00372      oup « xp;
00373
00374      deg_map.clear();
00375
00376      for (int i=0;i<it->second.size();i++){
00377        //cerr « " star vertex " « i « endl;
00378        //oup.bin_inter_code(it->second[i], n_bits);
00379        oup « it->second[i].size(); // how many star edges are going next
00380        deg_map[it->second[i].size()] ++;
00381        //cout « " i " « i « endl;
00382        for (int j=0;j<it->second[i].size();j++){
00383
00384          //cout « " j " « j « " -> " « it->second[i][j] « endl;
00385          if (j==0)
00386            diff = it->second[i][j] - i;
00387          else
00388            diff = it->second[i][j] - it->second[i][j-1];
00389          //if (it->second[i].size() < 100)
00390          //  cerr « diff « " ";
00391          //cerr « " diff " « diff « endl;
00392          // diff is bounded by the number of star vertices
00393          // we can either encode diff by a modification of Elias delta, using the extra assumption that
     1 <= diff <= number of star vertices - 1
00394          // number of star vertices = n - star_vertices.first[0] lets call it nu_star_vertices defined
     above before the loop
00395          // with this extra assumption in Elias delta, we do not use the unary part of the code and
     store nu_bits(diff)
00396          // using nu_bits(nu_bits(nu_star_vertices-1))
00397          // if we chose this method of compression, we use nu_bits(nu_bits(nu_star_vertices-1)) +
     nu_bits(diff)
00398          // if we only use diff <= nu_star_vertices - 1, we spend nu_bits(nu_star_vertices-1) bits
00399          // so we should spend the first method iff we have nu_bits(nu_bits(nu_star_vertices-1)) +
     nu_bits(diff) < nu_bits(nu_star_vertices-1)
00400          // or equivalently if nu_bits(diff) < nu_bits(nu_star_vertices-1) -
     nu_bits(nu_bits(nu_star_vertices-1))
00401          // we define  nu_bits(nu_star_vertices-1) =: nb_nsv and nu_bits(nu_bits(nu_star_vertices-1))
     =: nb_nb_nsv this is define above the loop
00402          // we also have defined the difference above as diff_threshold to simplify
00403          nb_diff = nu_bits(diff);
00404          if (nb_diff < diff_threshold){
00405            //if (it->second[i].size() < 100)
00406            //  cerr « " f ";
00407            // we choose the first method
00408            // we write a flag 1 upfront to tell decoder which method we use
00409            oup.write_bits(1,1); // write a single bit with value 1
```

```
00410              oup.write_bits(nb_diff, nb_nb_nsv); // write the number of bits in diff
00411              diff -= (1<<(nb_diff-1)); // remove the leading (MSB) bit from diff
00412              if (nb_diff > 1) // otherwise, we do not need to write anything (note we remove leading one,
       so if diff = 1, we should not write anything at this stage)
00413                  oup.write_bits(diff, nb_diff-1); // write diff to the output
00414            }else{
00415              //if (it->second[i].size() < 100)
00416              //  cerr << " s ";
00417              // we should choose the second method
00418              // write a flag 0 upfront
00419              oup.write_bits(0,1);
00420              oup.write_bits(diff, nb_nsv);
00421            }
00422          }
00423          //if (it->second[i].size() < 100)
00424          //  cerr << endl;
00425
00426          nu_star_edges += it->second[i].size();
00427
00428        }
00429      if (logger::stat){
00430        *logger::stat_stream << "mark pair: " <<  x << ", " << xp << endl;
00431        for (map<int, int>::iterator deg_it = deg_map.begin(); deg_it != deg_map.end(); deg_it++){
00432          *logger::stat_stream << " d " << deg_it->first << " # " << deg_it->second;
00433        }
00434        *logger::stat_stream << endl;
00435      }
00436    }
00437
00438    chunks_new = oup.chunks();
00439    space_log.push_back(pair<string, int> ("star edges", chunks_new - chunks));
00440    chunks = oup.chunks();
00441
00442    logger::add_entry("vertex types", "");
00443    //output_bits = 0;
00444
00445    // ==== write vertex types
00446
00447    // first, we need vertex types list (ver_type_list)
00448    // size of ver_type_list
00449
00450
00451    vtype_list_write(oup);
00452
00453
00454    // oup <<  ver_type_list.size();
00455    // //fwrite(&int_out, sizeof int_out, 1, f);
00456    // //output_bits += sizeof int_out;
00457
00458    // for (int i=0;i<ver_type_list.size();i++){
00459    //   oup << ver_type_list[i].size();
00460    //   //fwrite(&int_out, sizeof int_out, 1, f);
00461    //   //output_bits += sizeof int_out;
00462
00463    //   oup << ver_type_list[i][0]; // write the vertex mark
00464    //   // then, we know that the rest of the list is (t,t',n_{t,t'}) chunks, each of course with size
       3
00465    //   for (int j=0;j<((ver_type_list[i].size()-1)/3); j++){
00466    //     // the triple is 3j+1, 3j+2, 3j+3
00467    //     if (j==0){
00468    //       // the first chunk is written of the form t,t',n-1. The reason we write n-1 is because we
       know n > 0, so it is better to save some bits!
00469    //        oup << ver_type_list[i][3*j+1];
00470    //        oup << ver_type_list[i][3*j+2];
00471    //        oup << ver_type_list[i][3*j+3]-1;
00472    //     }else{
00473    //       // we know that the list is lexicographically ordered, so the t here is not less than the t
       is the previous chunk, so better to write their difference
00474    //        oup << ver_type_list[i][3*j+1] - ver_type_list[i][3*(j-1)+1];
00475    //       // if t_here is equal to t_previous, then t'_here >= t'_previous, so better to encode their
       difference!
00476    //       if (ver_type_list[i][3*j+1] == ver_type_list[i][3*(j-1)+1]){
00477    //         oup << ver_type_list[i][3*j+2] - ver_type_list[i][3*(j-1)+2];
00478    //       }else{
00479    //         // otherwise, just write t'_here
00480    //         oup << ver_type_list[i][3*j+2];
00481    //       }
00482    //        oup << ver_type_list[i][3*j+3]-1;
00483    //     }
00484    //   }
00485
00486    //   /* the old way of writing the list:
00487    //   for (int j=0;j<ver_type_list[i].size();j++){
00488    //     if (j%3 == 0 and j > 0) // we know that these indices are the count part (list is of the form
       \theta, t, t', n_{t,t'}, \dots but the n_{t,t'} \geq 1, so in Elias delta encoding of oup << we do not
       need to add one. So it would be more efficient to subtract one here, and add one during decompression)
00489    //        oup << ver_type_list[i][j]-1;
```

```
00490   //      else
00491   //        oup « ver_type_list[i][j];
00492   //      //fwrite(&int_out, sizeof int_out, 1, f);
00493   //      //output_bits += sizeof int_out;
00494   //   }
00495   //   */
00496   // }
00497
00498
00499
00500   chunks_new = oup.chunks();
00501   space_log.push_back(pair<string, int>("vertex type list", chunks_new - chunks));
00502   chunks = chunks_new;
00503
00504   //output_bits = 0;
00505
00506   // then, write ver_types
00507
00508   // ver_types.first
00509   // ver_types.first.size():
00510   oup « ver_types.first.size();
00511   //fwrite(&int_out, sizeof int_out, 1, f);
00512   //output_bits += sizeof int_out;
00513
00514   for (int i =0;i<ver_types.first.size(); i++){
00515     oup «  ver_types.first[i];
00516     //fwrite(&int_out, sizeof int_out, 1, f);
00517     //output_bits += sizeof int_out;
00518   }
00519   // ver_types.second
00520   oup « ver_types.second;
00521   //output_bits += mpz_out_raw(f, ver_types.second.get_mpz_t());
00522
00523   chunks_new = oup.chunks();
00524   space_log.push_back(pair<string, int> ("vertex types", chunks_new - chunks));
00525   chunks = chunks_new;
00526
00527   logger::add_entry("partition bipartite graphs", "");
00528
00529
00530   // ==== part bgraphs
00531   //output_bits = 0;
00532
00533   // part_bgraphs.size
00534   oup « part_bgraph.size();
00535   //fwrite(&int_out, sizeof int_out, 1, f);
00536   //output_bits += sizeof int_out;
00537
00538   map<pair<int, int>, mpz_class>::iterator it2;
00539   if (logger::stat){
00540     *logger::stat_stream « " ==== statistics ==== " « endl;
00541     *logger::stat_stream « " n:                " « n « endl;
00542     *logger::stat_stream « " h:                " « h « endl;
00543     *logger::stat_stream « " delta:            " « delta « endl;
00544     *logger::stat_stream « " No. types         " « type_mark.size() « endl;
00545     *logger::stat_stream « " No. * vertices    " « n - star_vertices.first[0] « endl;
00546     *logger::stat_stream « " No. * edges       " « nu_star_edges « endl;
00547     *logger::stat_stream « " No. part bgraphs  " « part_bgraph.size() « endl;
00548     *logger::stat_stream « " No. part graphs   " « part_graph.size() « endl;
00549   }
00550
00551   for (it2 = part_bgraph.begin(); it2 != part_bgraph.end(); it2++){
00552     // first, write t, t'
00553     oup « it2->first.first;
00554     //fwrite(&int_out, sizeof int_out, 1, f);
00555     //output_bits += sizeof int_out;
00556     oup «  it2->first.second;
00557     //fwrite(&int_out, sizeof int_out, 1, f);
00558     //output_bits += sizeof int_out;
00559     // then, the compressed integer
00560     oup « it2->second;
00561     //output_bits += mpz_out_raw(f, it2->second.get_mpz_t());
00562   }
00563
00564   chunks_new = oup.chunks();
00565   space_log.push_back(pair<string, int> ("partition bipartite graphs", chunks_new - chunks));
00566   chunks = chunks_new;
00567
00568   logger::add_entry("partition graphs", "");
00569   //output_bits = 0;
00570   // === part graphs
00571
00572   // part_graph.size
00573   oup «  part_graph.size();
00574   //fwrite(&int_out, sizeof int_out, 1, f);
00575   //output_bits += sizeof int_out;
00576
```

```
00577   map< int, pair< mpz_class, vector< int > > >::iterator it3;
00578   for (it3 = part_graph.begin(); it3 != part_graph.end(); it3++){
00579     oup << it3->first; // the type
00580     //fwrite(&int_out, sizeof int_out, 1, f);
00581     //output_bits += sizeof int_out;
00582
00583     // the mpz part
00584     oup << it3->second.first;
00585     //output_bits += mpz_out_raw(f, it3->second.first.get_mpz_t());
00586     // the vector part
00587     // first its size
00588     oup << it3->second.second.size();
00589     //fwrite(&int_out, sizeof int_out, 1, f);
00590     //output_bits += sizeof int_out;
00591     // then element by element
00592     for(int j=0;j<it3->second.second.size();j++){
00593       oup << it3->second.second[j];
00594       //fwrite(&int_out, sizeof int_out, 1, f);
00595       //output_bits += sizeof int_out;
00596     }
00597   }
00598
00599   chunks_new = oup.chunks();
00600   space_log.push_back(pair<string, int>("partition graphs", chunks_new - chunks));
00601   chunks = chunks_new;
00602
00603
00604   if (logger::stat){
00605     *logger::stat_stream << endl << endl;
00606     *logger::stat_stream << " Number of bytes used for each part " << endl;
00607     *logger::stat_stream << " -------------------------------- " << endl << endl;
00608
00609     int total_chunks = 0;
00610     for (int i=0; i < space_log.size(); i++)
00611       total_chunks += space_log[i].second;
00612
00613     for (int i=0; i < space_log.size(); i++){
00614       // each chunks is 4 bytes.
00615       *logger::stat_stream << space_log[i].first << " -> " << 4 * space_log[i].second << " ( " <<
    float(100) * float(space_log[i].second) / float(total_chunks) << " % " << endl;
00616     }
00617
00618     *logger::stat_stream << " Total number of bytes wrote to the output = " << 4 * total_chunks << endl;
00619   }
00620
00621   oup.close();
00622   logger::current_depth--;
00623 }
```

6.15.1.5 clear()

```
void marked_graph_compressed::clear ( )
00009 {
00010   star_edges.clear();
00011   type_mark.clear();
00012   ver_type_list.clear();
00013   part_bgraph.clear();
00014   part_graph.clear();
00015 }
```

6.15.1.6 vtype_block_read() [1/2]

```
void marked_graph_compressed::vtype_block_read (
           ibitstream & inp,
           int i,
           int j )
```

reads a (t,t',n) block of a vertex type list element from input.

Parameters

inp	the input bitstream used to read the encoded bit sequence
i	the index of ver_type_list member of this class which is going to be decoded
j	the index of the t,t',n block of ver_type_list[i] that is going to be decoded. Hence, the block is
	ver_type_list[i][1+3∗j], ver_type_list[i][2+3∗j], and ver_type_list[i][3+3∗j]

```
01280                                                                                    {
01281   unsigned int int_in;
01282   inp » int_in;
01283   ver_type_list[i][3*j+1] = int_in;
01284   inp » int_in;
01285   ver_type_list[i][3*j+2] = int_in;
01286   inp » int_in;
01287   ver_type_list[i][3*j+3] = int_in + 1; // since we had subtracted one during compression
01288 }
```

6.15.1.7 vtype_block_read() [2/2]

```
void marked_graph_compressed::vtype_block_read (
              ibitstream & inp,
              int i,
              int j,
              int ir,
              int jr )
```

reads a (t,t',n) block of a vertex type list element with reference to another reference block. We assume that this block is lexicographically greater than the reference block, and this fact was used in the compression to encode the difference. The reference block must be prior to the current block so that it is already decoded and ready to be used as a reference.

Parameters

inp	the input bitstream used to read the encoded bit sequence
i	the index of ver_type_list member of this class which is going to be decoded
j	the index of the t,t',n block of ver_type_list[i] that is going to be decoded. Hence, the block is ver_type_list[i][1+3∗j], ver_type_list[i][2+3∗j], and ver_type_list[i][3+3∗j]
ir	the index of the ver_type_list used as the reference
jr	the index of the block in ver_type_list[ir] used as the reference. Hence, the reference block is ver_type_list[ir][1+3∗jr], ver_type_list[ir][2+3∗jr], and ver_type_list[ir][3+3∗jr]

```
01290                                                                                    {
01291   // we use the following terminologies for comments:
01292   // t = ver_type_list[i][3*j+1]
01293   // t' = ver_type_int[i][3*j+2]
01294   // n = ver_type_list[i][3*j+3]
01295   // t_r = ver_type_list[ir][3*jr+1]
01296   // t'_r = ver_type_list[ir][3*jr+2]
01297   // n_r = ver_type_list[ir][3*jr+3]
01298   unsigned int int_in;
01299   inp » int_in;
01300   ver_type_list[i][3*j+1] = int_in + ver_type_list[ir][3*jr+1]; // since we had encoded the difference
      during the compression phasen
01301
01302   // if t = t_r, we have encoded t' - t'_r - 1
01303   if (ver_type_list[i][3*j+1] == ver_type_list[ir][3*jr+1]){
01304     inp » int_in;
01305     ver_type_list[i][3*j+2] = int_in + ver_type_list[ir][3*jr+2];
01306     if (ver_type_list[i][3*j+2] == ver_type_list[ir][3*jr+2]){
01307       inp » int_in;
01308       ver_type_list[i][3*j+3] = int_in +  ver_type_list[ir][3*jr+3];
01309     }else{
01310       inp » int_in;
01311       ver_type_list[i][3*j+3] = int_in +  1;
01312     }
01313   }else{
01314     // otherwise, we have just encoded t'
01315     inp » int_in;
01316     ver_type_list[i][3*j+2] = int_in;
01317     inp » int_in;
01318     ver_type_list[i][3*j+3]= int_in + 1; // since we had subtracted one during compression
01319   }
01320 }
```

6.15.1.8 vtype_block_write() [1/2]

```
void marked_graph_compressed::vtype_block_write (
            obitstream & oup,
            int i,
            int j )
```

writes a (t,t',n) block of a vertex type list element to the output.

Parameters

oup	the output bitstream used to output the encoded bit sequence
i	the index of ver_type_list member of this class which is going to be encoded
j	the index of the t,t',n block of ver_type_list[i] that is going to be encoded. Hence, the block is ver_type_list[i][1+3∗j], ver_type_list[i][2+3∗j], and ver_type_list[i][3+3∗j]

```
01238                                                                        {
01239   oup « ver_type_list[i][3*j+1];
01240   oup « ver_type_list[i][3*j+2];
01241   oup « ver_type_list[i][3*j+3]-1;
01242 }
```

6.15.1.9 vtype_block_write() [2/2]

```
void marked_graph_compressed::vtype_block_write (
            obitstream & oup,
            int i,
            int j,
            int ir,
            int jr )
```

writes a (t,t',n) block of a vertex type list element with reference to another reference block to the output. We assume that this block is lexicographically greater than the reference block, and use it to encode the difference to save space

Parameters

oup	the output bitstream used to output the encoded bit sequence
i	the index of ver_type_list member of this class which is going to be encoded
j	the index of the t,t',n block of ver_type_list[i] that is going to be encoded. Hence, the block is ver_type_list[i][1+3∗j], ver_type_list[i][2+3∗j], and ver_type_list[i][3+3∗j]
ir	the index of the ver_type_list used as the reference
jr	the index of the block in ver_type_list[ir] used as the reference. Hence, the reference block is ver_type_list[ir][1+3∗jr], ver_type_list[ir][2+3∗jr], and ver_type_list[ir][3+3∗jr]

```
01244                                                                        {
01245   // we use the following terminologies for comments:
01246   // t = ver_type_list[i][3*j+1]
01247   // t' = ver_type_int[i][3*j+2]
01248   // n = ver_type_list[i][3*j+3]
01249   // t_r = ver_type_list[ir][3*jr+1]
01250   // t'_r = ver_type_list[ir][3*jr+2]
01251   // n_r = ver_type_list[ir][3*jr+3]
01252
01253   //write the difference between the t parts, i.e. t - t_r
01254   oup « ver_type_list[i][3*j+1] - ver_type_list[ir][3*jr+1];
01255
01256   // check if the t part is the same, i.e. t = t_r
01257   if (ver_type_list[i][3*j+1] == ver_type_list[ir][3*jr+1]){
01258     // encode the difference of the t' part
01259     // but in this case, t' >= t'_r, so write the difference
01260     oup « ver_type_list[i][3*j+2] - ver_type_list[ir][3*jr+2];
```

```
01261    // for sanity check:
01262    // TO BE REMOVED LATER
01263    //if (ver_type_list[i][3*j+2] == ver_type_list[ir][3*jr+2])
01264    //cerr << " warning marked_graph_compressed::vtype_block_write : (t,t') = (t_r, t'_r), (i,j) = " <<
    i << ", " << j << ") and (ir, jr) = (" << ir << ", " << jr << ")" << endl;
01265    if (ver_type_list[i][3*j+2] == ver_type_list[ir][3*jr+2]){
01266       oup << ver_type_list[i][3*j+3] - ver_type_list[ir][3*jr+3];
01267    }else{
01268       oup << ver_type_list[i][3*j+3] - 1;
01269    }
01270    }else{
01271    // just write t'
01272    oup << ver_type_list[i][3*j+2];
01273    // finally, write n - 1, note that n > 0, so we encode n - 1 to save some space
01274    oup << ver_type_list[i][3*j+3] - 1;
01275    }
01276
01277
01278 }
```

6.15.1.10 vtype_list_read()

```
void marked_graph_compressed::vtype_list_read (
            ibitstream & inp )
```

reads the ver_type_list array from the input, assuming the bit sequence was generated during compression using vtype_list_write

```
01149                                                     {
01150    // read vertex mark block counts
01151    vector<pair<int, int> > ver_types_freq; // each element (m,k) means that the ver mark m appears in k
    many vertex types
01152    //vector<pair<int, int> > ver_types_blocks; // the ith entry is (a,b), where a and b denote the
    range of indices in ith block. Note that the range is of the form [a,b)
01153    unsigned int int_in;
01154
01155    inp >> int_in;
01156    ver_types_freq.resize(int_in);
01157    int total_types = 0; // size of ver_type_int
01158
01159    int block_start = 0;
01160    for (int i=0;i<ver_types_freq.size();i++){
01161       inp >> int_in;
01162       ver_types_freq[i].first = int_in;
01163       inp >> int_in;
01164       ver_types_freq[i].second = int_in + 1;
01165       total_types += ver_types_freq[i].second;
01166       //ver_types_blocks.push_back(pair<int,int>(block_start, block_start+ver_types_freq[i].second));
01167       //block_start += ver_types_freq[i].second;
01168    }
01169
01170    // create ver_types_blocks
01171
01172    ver_type_list.resize(total_types);
01173    int block = 0; // the block at which we are in terms of vertex mark
01174    int count_in_block = 0; // the index of me in the current blcok
01175    int max_match; // maximum match with the previous row
01176    int total_chunks;
01177    for (int i=0; i<ver_type_list.size(); i++){
01178       if (count_in_block == ver_types_freq[block].second){
01179          block++;
01180          count_in_block =0 ;
01181       }
01182       if (count_in_block==0){
01183          inp >> int_in; // the number of chunks
01184          ver_type_list[i].resize(1+3*int_in);
01185          ver_type_list[i][0] = ver_types_freq[block].first; // fix the vertex mark
01186          for (int j=0; j<int_in; j++){ // read blocks one by one
01187             if (j==0)
01188                vtype_block_read(inp, i, j);
01189             else
01190                vtype_block_read(inp, i, j, i, j-1); // use the previous block as reference
01191          }
01192       }else{
01193          inp >> int_in;
01194          max_match = int_in;
01195          inp >> int_in; // the remaining chunks
01196          total_chunks = max_match + int_in;
01197          ver_type_list[i].resize(1+3*total_chunks);
01198          // first, fill the matching chunks
01199          ver_type_list[i][0] = ver_types_freq[block].first;
01200          for (int j=0; j<max_match;j++){
```

```
01201              ver_type_list[i][1+3*j] = ver_type_list[i-1][1+3*j];
01202              ver_type_list[i][2+3*j] = ver_type_list[i-1][2+3*j];
01203              ver_type_list[i][3+3*j] = ver_type_list[i-1][3+3*j];
01204          }
01205        if (max_match < total_chunks){
01206          // there are still some blocks to be decoded
01207          // first, we need to encode the block max_match itself
01208          // if this block exists in row i-1, we use that as a reference,
01209          if (max_match <(ver_type_list[i-1].size()-1)/3){
01210            vtype_block_read(inp, i, max_match, i-1, max_match);
01211            // then, write the remaining blocks, if any
01212            for (int j = max_match+1; j<(ver_type_list[i].size()-1)/3; j++){
01213              vtype_block_read(inp, i, j, i, j-1);
01214            }
01215          }else{
01216            // if block max_match - 1 exists in row i, use it as a reference
01217            if (max_match > 0){
01218              for (int j = max_match; j<total_chunks; j++){
01219                vtype_block_read(inp, i, j, i, j-1);
01220              }
01221            }else{
01222              // otherwise, encode the first block standalone, and recursively go forward and use
      previous block as reference
01223              vtype_block_read(inp, i, max_match);
01224              for (int j = max_match+1; j<total_chunks; j++){
01225                vtype_block_read(inp, i, j, i, j-1);
01226              }
01227            }
01228          }
01229        }
01230      }
01231      count_in_block++;
01232    }
01233
01234 }
```

6.15.1.11 vtype_list_write()

```
void marked_graph_compressed::vtype_list_write (
              obitstream & oup )
```

writes the ver_type_list array to the output using difference coding, assuming that its entries are lexicographically sorted (as vectors)

```
01049                                                      {
01050    // first, we should extract information about vertex marks
01051    vector<pair<int, int> > ver_types_freq; // each element (m,k) means that the ver mark m appears in k
      many vertex types
01052    vector<pair<int, int> > ver_types_blocks; // the ith entry is (a,b), where a and b denote the range
      of indices in ith block. Note that the range is of the form [a,b)
01053    //int prev_mark = ver_type_list[0][0]-1; // define it this way, so that initially it is different
      from the first mark
01054    int current_mark; // the mark of the current blcok
01055    int block_start = 0; // the start index of the current vertex mark block, a block is subsequent
      entries in ver_type_list with the same vertex marks
01056    int i = 0;
01057    // cerr « " ver_type_list.size() " « ver_type_list.size() « endl;
01058    // cerr « " ver_type_list " « endl;
01059    // for (int k=0; k<ver_type_list.size(); k++){
01060    //   cerr « "(" « k « ") ";
01061    //   cerr « ver_type_list[k][0]« ": ";
01062    //   for (int j=0; j< (ver_type_list[k].size()-1)/3; j++)
01063    //     cerr « "| " « ver_type_list[k][1+3*j] « " " « ver_type_list[k][2+3*j] « " " «
      ver_type_list[k][3+3*j] « " ";
01064    //   cerr « endl;
01065    // }
01066    while(i<ver_type_list.size()){
01067      current_mark = ver_type_list[block_start][0];
01068      while (ver_type_list[i][0] == current_mark){
01069        //cerr « i « " ";
01070        i++;
01071        if (i >= ver_type_list.size())
01072          break;
01073      }
01074      ver_types_freq.push_back(pair<int, int>(current_mark, i - block_start));
01075      ver_types_blocks.push_back(pair<int, int>(block_start, i));
01076      block_start = i;
01077    }
01078
01079
01080    // writing ver_type_freq to the output
01081    oup « ver_types_freq.size();
```

```
01082   for (i=0;i<ver_types_freq.size();i++){
01083     //cerr « " mark " « ver_types_freq[i].first « " count " « ver_types_freq[i].second « endl;
01084     oup « ver_types_freq[i].first;
01085     oup « ver_types_freq[i].second - 1;// since it is at least one
01086   }
01087
01088   // now, we go over each block
01089   // let b denote the block index
01090   int max_match;
01091
01092   bool cond_1 = false;
01093   bool cond_2 = false;
01094   bool cond_3 = false;
01095   for (int b=0; b<ver_types_freq.size();b++){
01096     for (i=ver_types_blocks[b].first; i<ver_types_blocks[b].second; i++){
01097       //for (i =0; i<ver_type_list.size();i++){
01098       if (i==ver_types_blocks[b].first){
01099         // this is the first row in that block, so its compression is different
01100         oup « (ver_type_list[i].size()-1)/3; // number of blocks
01101         for (int j=0;j<(ver_type_list[i].size()-1)/3; j++){
01102           if (j==0)
01103             vtype_block_write(oup, i, j);
01104           else
01105             vtype_block_write(oup, i, j, i, j-1); // use the previous block as reference
01106         }
01107       }else{
01108         max_match = vtype_max_match(i, i-1); // compare with the previous one
01109         oup « max_match;
01110         oup « (ver_type_list[i].size()-1)/3 - max_match; // number of remaining blocks
01111         //cerr « " i " « i « " max_match " « max_match;
01112         if (max_match < (ver_type_list[i].size()-1)/3){
01113           // there are still some blocks to be encoded
01114           // first, we need to encode the block max_match itself
01115           // if this block exists in row i-1, we use that as a reference,
01116           if (max_match <(ver_type_list[i-1].size()-1)/3){
01117             //cerr « " 1" « endl;
01118             cond_1 = true;
01119             vtype_block_write(oup, i, max_match, i-1, max_match);
01120             // then, write the remaining blocks, if any
01121             for (int j = max_match+1; j<(ver_type_list[i].size()-1)/3; j++){
01122               vtype_block_write(oup, i, j, i, j-1);
01123             }
01124           }else{
01125             // if block max_match - 1 exists in row i, use it as a reference
01126             if (max_match > 0){
01127               //cerr « " 2" « endl;
01128               cond_2 = true;
01129               for (int j = max_match; j<(ver_type_list[i].size()-1)/3; j++){
01130                 vtype_block_write(oup, i, j, i, j-1);
01131               }
01132             }else{
01133               //cerr « " 3" « endl;
01134               // otherwise, encode the first block standalone, and recursively go forward and use
      previous block as reference
01135               cond_3 = true;
01136               vtype_block_write(oup, i, max_match);
01137               for (int j = max_match+1; j<(ver_type_list[i].size()-1)/3; j++){
01138                 vtype_block_write(oup, i, j, i, j-1);
01139               }
01140             }
01141           }
01142         }
01143       }
01144     }
01145   }
01146   //cerr « " cond_1 " « cond_1 « " cond_2 " « cond_2 « " cond_3 " « cond_3 « endl;
01147 }
```

6.15.1.12 vtype_max_match()

```
int marked_graph_compressed::vtype_max_match (
            int i,
            int j )
```

finds the maximum number of (t,t',n) blocks that match between two entries of ver_type_list

```
01031                                                                      {
01032   int max_match = 0;
01033   int k;
01034   bool flag;
01035   while(true){
```

```
01036      if (((3*max_match + 3) >= ver_type_list[i].size()) or ((3*max_match+3) >=
      ver_type_list[j].size())) // one of the lists is reached an end
01037        break;
01038      flag = true;
01039      for (k=1;k<=3;k++)
01040        if (ver_type_list[i][3*max_match+k] != ver_type_list[j][3*max_match+k])
01041          flag = false;
01042      if (flag == false)
01043        break;
01044      max_match++;
01045    }
01046    return max_match;
01047 }
```

6.15.2 Member Data Documentation

6.15.2.1 delta

int marked_graph_compressed::delta

the degree threshold used when compression was performed

6.15.2.2 h

int marked_graph_compressed::h

the depth up to which the compression was performed

6.15.2.3 n

int marked_graph_compressed::n

the number of vertices

6.15.2.4 part_bgraph

map<pair<int, int>, mpz_class> marked_graph_compressed::part_bgraph

compressed form of partition bipartite graphs corresponding to colors in $C_<$. For a pair $0 \leq t < t' < L$ of half edge types, part_bgraph[pair<int, int>(t,t')] is the compressed form of the bipartite graph with n left and right nodes, where a left node i is connected to a right node j if there is an edge connecting i to j with type t towards i and type t' towards j

6.15.2.5 part_graph

map<int, pair<mpz_class, vector<int> > > marked_graph_compressed::part_graph

compressed form of partition graphs corresponding to colors in $C_=$. For a half edge type t, part_graph[t] is the compressed form of the simple unmarked graph with n vertices, where a node i is connected to a node j where there is an edge between i and j in the original graph with color (t,t)

6.15.2.6 star_edges

map<pair<int, int> , vector<vector<int> > > marked_graph_compressed::star_edges

for each pair of edge marks x,x', and integer k, star_edges[pair<int,int>(x,x')][k] is a list of neighbors w of the kth star vertex (say v) so that v shares a star edge with w so that the mark towards v is x and the mark towards w is xp.

6.15.2.7 star_vertices

pair<vector<int>, mpz_class> marked_graph_compressed::star_vertices

the compressed form of the star_vertices list

6.15.2.8 type_mark

vector<int> marked_graph_compressed::type_mark

for an edge type t, type_mark[t] denotes the mark component of t

6.15.2.9 ver_type_list

vector<vector<int> > marked_graph_compressed::ver_type_list

the list of all vertex types that appear in the graph, where the type of a vertex is a vector of integers, where its index 0 is the mark of the vertex, and indices $3k+1$, $3k+2$, $3k+3$ are m, m' and $n_{m,m'}$, where (m, m') is a type pair, and $n_{m,m'}$ is the number of edges connected to the vertex with that type. The list is sorted lexicographically to ensure unique representation.

6.15.2.10 ver_types

pair<vector<int>, mpz_class> marked_graph_compressed::ver_types

the compressed form of vertex types, where the type of a vertex is the index with respect to ver_type_list of the list of integers specifying the type of the vertex (mark of the vertex followed by the number of edges of each type connected to that vertex)

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

- marked_graph_compression.h
- marked_graph_compression.cpp

6.16 marked_graph_decoder Class Reference

#include <marked_graph_compression.h>

Public Member Functions

- marked_graph_decoder ()

  constructor
- marked_graph decode (const marked_graph_compressed &)

Private Member Functions

- void decode_star_vertices (const marked_graph_compressed &)
- void decode_star_edges (const marked_graph_compressed &)
- void decode_vertex_types (const marked_graph_compressed &)
- void find_part_deg_orig_index ()
- void decode_partition_graphs (const marked_graph_compressed &)
- void decode_partition_bgraphs (const marked_graph_compressed &)

Private Attributes

- int h
- int delta
- int n

  number of vertices, this is set when a graph G is given to be encoded
- vector< int > is_star_vertex

  for $0 \leq v < n$, is_star_vertex[v] is 1 if there is at least one star type edge connected to v and 0 otherwise.
- vector< int > star_vertices

  the list of star vertices
- map< pair< int, int >, b_graph > part_bgraph

  for $0 \leq t < t' < L$, part_bgraph[pair<int, int> (t,t')] is a bipartite graph with n left vertex and n right vertex. In this bipartite graph, a left vertex i is connected to a right vertex j iff there is an edge in the graph between vertices i and j with a half edge type towards i equal to t and a half edge type towards j equal to t'.
- map< int, graph > part_graph

  for $0 \leq t < L$, part_graph[i] is a simple unmarked graph with n vertices. In this graph, vertices i and j are connected in the original graph with an edge with half edge types t in both directions i and j.
- vector< pair< pair< int, int >, pair< int, int > > > edges

  the list of edges in the decoded graph, each index of the form $((i, j), (x, y))$, where i and j are the endpoints and x and y are the marks (towards i and j, respectively).
- vector< int > vertex_marks

  the list of vertex marks of the marked graph to be decoded
- vector< map< pair< int, int >, int > > Deg

  for a vertex $0 \leq v < n$, Deg[v] is a map such that Deg[v][(t,t')] is the number of edges connected to v with type (t, t') (if such an edge exists)
- map< pair< int, int >, vector< int > > part_deg

  part_deg[(t,t')] is the degree sequence of the partition graph corresponding to pair of types t and t', if $t \neq t'$, this is the degree sequence of the side of the graph corresponding to t.
- map< pair< int, int >, vector< int > > origin_index

  origin_index[(t,t')][v] gives the original index in the marked graph corresponding to the vertex v in the (t, t') partition graph. Here, if $t \neq t'$, v is in the side of the bipartite graph corresponding to t

6.16.1 Constructor & Destructor Documentation

6.16.1.1 marked_graph_decoder()

```
marked_graph_decoder::marked_graph_decoder ( )  [inline]
```

constructor
```
00192 {}
```

6.16.2 Member Function Documentation

6.16.2.1 decode()

```
marked_graph marked_graph_decoder::decode (
                const marked_graph_compressed & compressed )
01597 {
01598   logger::current_depth++;
01599   logger::add_entry("Init", "");
01600   n = compressed.n;
01601   h = compressed.h;
01602   delta = compressed.delta;
01603
01604
01605   edges.clear(); // clear the edge list of the marked graph to be decoded
01606   vertex_marks.clear(); // clear the list of vertex marks of the marked graph to be decoded
01607
01608   logger::add_entry("Decode * vertices", "");
01609   decode_star_vertices(compressed);
01610   //cerr « " decoded star vertices " « endl;
01611
01612   logger::add_entry("Decode * edges", "");
01613   decode_star_edges(compressed);
01614   //cerr « " decoded star edges " « endl;
01615
01616   logger::add_entry("Decode vertex types", "");
01617   decode_vertex_types(compressed);
01618   //cerr « " decoded vertex types " « endl;
01619
01620   logger::add_entry("Decode partition graphs", "");
01621   decode_partition_graphs(compressed);
01622   //cerr « " decoded partition graphs " « endl;
01623
01624   logger::add_entry("Decode partition b graphs", "");
01625   decode_partition_bgraphs(compressed);
01626   //cerr « " decoded partition b graphs " « endl;
01627
01628   // now, reconstruct the original marked graphs by assembling the vertex marks and edge list
01629   logger::add_entry("Construct decoded graph", "");
01630   marked_graph G(n, edges, vertex_marks);
01631
01632   logger::current_depth--;
01633   return G;
01634 }
```

6.16.2.2 decode_partition_bgraphs()

```
void marked_graph_decoder::decode_partition_bgraphs (
                const marked_graph_compressed & compressed )  [private]
01762 {
01763   pair<int, int> c; // the pair of types
01764   int t, tp; // types
01765   int x, xp; // mark components of t and tp
01766
01767   b_graph G; // the decoded partition bipartite graph
01768   int nl_G; // the number of left nodes in the partition graph G
01769   vector<int> adj_list; // adj list of a vertex in a partition bipartite graph
01770   int w; // a right node
01771   int v_orig, w_orig; // the original index of vertices v and w in partition graphs
01772   for (map<pair<int, int>, mpz_class>::const_iterator it = compressed.part_bgraph.begin();
    it!=compressed.part_bgraph.end(); it++){
01773     c = it->first;
01774     t = c.first;
01775     tp = c.second;
01776     x = compressed.type_mark[t]; // the mark component of t
01777     xp = compressed.type_mark[tp]; // the mark component of tp
01778
01779     //cerr « " t " « t « " tp " « tp « endl;
01780
01781     b_graph_decoder D(part_deg.at(pair<int, int>(t,tp)), part_deg.at(pair<int, int>(tp,t))); // the
    degree sequence of left nodes is precisely part_deg.at(pair<int, int>(t,tp)), while that of the right
    nodes is precisely part_deg.at(pair<int, int>(tp,t))
01782     //cerr « " decoder constructed " « endl;
01783     //cerr « " part graph t = "  « t « " t' = " « tp « " nl " « part_deg.at(pair<int,
    int>(t,tp)).size() « " nr = " « part_deg.at(pair<int, int>(tp,t)).size() « endl;
01784     G = D.decode(it->second);
01785
01786     //cerr « " G decoded " « endl;
```

```
01787      nl_G = part_deg.at(pair<int, int>(t,tp)).size(); // the number of left nodes in G is obtained from
     the size of the degree sequence of left nodes
01788
01789      for (int v=0;v<nl_G;v++){
01790        v_orig = origin_index.at(pair<int, int>(t,tp))[v];
01791        //cerr « " v " « v « " v_orig " « v_orig « endl;
01792        adj_list = G.get_adj_list(v);
01793        for (int i=0;i<adj_list.size();i++){
01794          w = adj_list[i];
01795          w_orig = origin_index.at(pair<int, int>(tp,t))[w]; // since w is a right node, we should read
     its original index through origin_index[(tp,t)]
01796          //cerr « " w " « w « " w_orig " « w_orig « endl;
01797          edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v_orig,w_orig),
     pair<int, int>(x,xp)));
01798        }
01799      }
01800    }
01801 }
```

6.16.2.3 decode_partition_graphs()

```
void marked_graph_decoder::decode_partition_graphs (
                const marked_graph_compressed & compressed )  [private]
01725 {
01726   int t; // the type corresponding to the partition graph
01727   vector<int> t_message; // the actual message corresponding to t
01728   int x; // the mark component associated to t
01729   pair< mpz_class, vector< int > > G_compressed; // the compressed form of the partition graph
01730   graph G; // the decoded partition graph
01731   vector<int> flist; // the forward adjacency list of a vertex in a partition graph
01732   int w; // vertex in partition graph
01733   int v_orig, w_orig; // the original index of vertices v and w
01734   int n_G; // the number of vertices of the partitioned graph
01735
01736   for(map< int, pair< mpz_class, vector< int > > >::const_iterator it=compressed.part_graph.begin();
     it!=compressed.part_graph.end(); it++){
01737     t = it->first;
01738     x = compressed.type_mark[t]; // the mark component of t
01739
01740     G_compressed = it->second;
01741     // the degree sequence of the graph can be obtained from part_deg.at(pair<int,int>(t,t))
01742     graph_decoder D(part_deg.at(pair<int, int>(t,t)));
01743     //cerr « " part_graph t = " « t « " with " « part_deg.at(pair<int, int>(t,t)).size() « " vertices
     " « endl;
01744     n_G = part_deg.at(pair<int, int>(t,t)).size(); // the number of vertices in the partition graph is
     read from the size of its degree sequence
01745     G = D.decode(G_compressed.first, G_compressed.second);
01746     // for each edge in G, we should add an edge with mark pair (x,x) to the edge list of the marked
     graph
01747     for (int v=0;v<n_G;v++){
01748       flist = G.get_forward_list(v);
01749       v_orig = origin_index.at(pair<int, int>(t,t))[v]; // the index of v in the original graph
01750       for (int i=0;i<flist.size();i++){
01751         w = flist[i]; // the other endpoint in the partition graph
01752         w_orig = origin_index.at(pair<int, int>(t,t))[w]; // the index of w in the original graph
01753         edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v_orig,w_orig),
     pair<int, int> (x,x)));
01754       }
01755     }
01756   }
01757 }
```

6.16.2.4 decode_star_edges()

```
void marked_graph_decoder::decode_star_edges (
                const marked_graph_compressed & compressed )  [private]
01648 {
01649   pair<int, int> mark_pair; // the pair of marks
01650   vector<vector<int> > list; // list of edges with this pair of marks
01651   int v, w; //endpoints of the star edge
01652   // iterating through the star_edges map
01653   for (map<pair<int, int>, vector<vector<int> > >::const_iterator it = compressed.star_edges.begin();
     it!=compressed.star_edges.end(); it++){
01654     mark_pair = it->first;
01655     //cerr « " mark_pair " « mark_pair.first « " " « mark_pair.second « endl;
01656     list = it->second;
01657     for (int i=0;i<list.size();i++){
```

```
01658        v = star_vertices[i];
01659        for (int j=0;j<list[i].size();j++){
01660          //cerr << " list[i][j] " << list[i][j] << endl;
01661          w = star_vertices[list[i][j]]; // star edges are stored in compressed format using to the
       indexing with respect to star vertices
01662          //cerr << " w " << w << endl;
01663          edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v,w), mark_pair));
01664        }
01665      }
01666    }
01667 }
```

6.16.2.5 decode_star_vertices()

```
void marked_graph_decoder::decode_star_vertices (
              const marked_graph_compressed & compressed )  [private]
01637 {
01638   time_series_decoder D(n);
01639   is_star_vertex = D.decode(compressed.star_vertices);
01640
01641   star_vertices.clear();
01642   for (int i=0;i<n;i++)
01643     if (is_star_vertex[i] == 1)
01644       star_vertices.push_back(i);
01645 }
```

6.16.2.6 decode_vertex_types()

```
void marked_graph_decoder::decode_vertex_types (
              const marked_graph_compressed & compressed )  [private]
01670 {
01671   time_series_decoder D(n);
01672   vector<int> ver_type_int = D.decode(compressed.ver_types);
01673
01674   // converting the integer value vertex types to actual vectors using the `ver_type_list` attribute
       of compressed
01675
01676   vertex_marks.resize(n); // preparing for decoding vertex marks
01677   Deg.clear(); // refresh
01678   Deg.resize(n);
01679
01680   vector<int> x; // auxiliary vector
01681
01682   for (int v=0;v<n;v++){
01683     if (ver_type_int[v] >= compressed.ver_type_list.size())
01684       cerr << " Warning: marked_graph_decoder::decode_vertex_types ver_type_int[" << v << "] is out of
       range" << endl;
01685     x = compressed.ver_type_list[ver_type_int[v]];
01686     vertex_marks[v] =x[0]; // the mark of vertex v is the first element in the type list of this
       vertex
01687     // now, we extract Deg[v] by looking at batches of size 3 in x
01688     for (int i=1;i<x.size();i+=3){
01689       if (i+2 >= x.size())
01690         cerr << " Error: marked_graph_decoder::decode_vertex_types, the type of vertex " << v << " does
       not obey length constrains, i.e. it does not have length 1 + 3k " << endl;
01691         Deg[v][pair<int, int>(x[i],x[i+1])] = x[i+2]; // x[i] and x[i+1] are types, and x[i+2] is
       the count
01692     }
01693   }
01694
01695   find_part_deg_orig_index(); // find part_deg and orig_index maps
01696 }
```

6.16.2.7 find_part_deg_orig_index()

```
void marked_graph_decoder::find_part_deg_orig_index ( )  [private]
01699 {
01700   part_deg.clear();
01701   origin_index.clear();
01702   int t, tp; // types
01703
01704   //cerr << " decoded deg : " << endl;
```

```
01705   for (int v=0;v<n;v++){
01706     //cerr « " v " « v « endl;
01707     for (map<pair<int, int>, int>::iterator it=Deg[v].begin(); it!=Deg[v].end(); it++){
01708       t = it->first.first;
01709       tp = it->first.second;
01710       //cerr « " t " « t « " tp " « tp « " : " « it->second « endl;
01711       if (part_deg.find(it->first) == part_deg.end()){
01712         // this is our first encounter with this type pair
01713         origin_index[it->first] = vector<int>({v}); // v is the first node in the t side of the (t,t')
    partition graph
01714         part_deg[it->first] = vector<int>({it->second}); // the degree in the partition graph is read
    from it->second
01715       }else{
01716         origin_index.at(it->first).push_back(v); // v is the next vertex observed with type t,t', so
    the vertex in the partition graph with index origin_index[it->first] has original index v
01717         // append degree of v, which is it->second
01718         part_deg.at(it->first).push_back(it->second);
01719       }
01720     }
01721   }
01722 }
```

6.16.3 Member Data Documentation

6.16.3.1 Deg

vector<map<pair<int, int>, int> > marked_graph_decoder::Deg [private]

for a vertex $0 \leq v < n$, Deg[v] is a map such that Deg[v][(t,t')] is the number of edges connected to v with type (t, t') (if such an edge exists)

6.16.3.2 delta

int marked_graph_decoder::delta [private]

6.16.3.3 edges

vector<pair<pair<int, int>, pair<int, int> > > marked_graph_decoder::edges [private]

the list of edges in the decoded graph, each index of the form $((i, j), (x, y))$, where i and j are the endpoints and x and y are the marks (towards i and j, respectively).

6.16.3.4 h

int marked_graph_decoder::h [private]

6.16.3.5 is_star_vertex

vector<int> marked_graph_decoder::is_star_vertex [private]

for $0 \leq v < n$, is_star_vertex[v] is 1 if there is at least one star type edge connected to v and 0 otherwise.

6.16.3.6 n

`int marked_graph_decoder::n [private]`

number of vertices, this is set when a graph G is given to be encoded

6.16.3.7 origin_index

`map<pair<int, int>, vector<int> > marked_graph_decoder::origin_index [private]`

origin_index[(t,t')][v] gives the original index in the marked graph corresponding to the vertex v in the (t, t') partition graph. Here, if $t \neq t'$, v is in the side of the bipartite graph corresponding to t

6.16.3.8 part_bgraph

`map<pair<int, int>, b_graph> marked_graph_decoder::part_bgraph [private]`

for $0 \leq t < t' < L$, part_bgraph[pair<int, int> (t,t')] is a bipartite graph with n left vertex and n right vertex. In this bipartite graph, a left vertex i is connected to a right vertex j iff there is an edge in the graph between vertices i and j with a half edge type towards i equal to t and a half edge type towards j equal to t'.

6.16.3.9 part_deg

`map<pair<int, int>, vector<int> > marked_graph_decoder::part_deg [private]`

part_deg[(t,t')] is the degree sequence of the partition graph corresponding to pair of types t and t', if $t \neq t'$, this is the degree sequence of the side of the graph corresponding to t.

6.16.3.10 part_graph

`map<int, graph> marked_graph_decoder::part_graph [private]`

for $0 \leq t < L$, part_graph[i] is a simple unmarked graph with n vertices. In this graph, vertices i and j are connected in the original graph with an edge with half edge types t in both directions i and j.

6.16.3.11 star_vertices

`vector<int> marked_graph_decoder::star_vertices [private]`

the list of star vertices

6.16.3.12 vertex_marks

`vector<int> marked_graph_decoder::vertex_marks [private]`

the list of vertex marks of the marked graph to be decoded

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

- marked_graph_compression.h
- marked_graph_compression.cpp

6.17   marked_graph_encoder Class Reference

`#include <marked_graph_compression.h>`

Public Member Functions

- marked_graph_encoder (int h_, int delta_)
- marked_graph_compressed encode (const marked_graph &G)

  compresses a simple marked graph G, and returns the compressed form as an object of type `marked_graph_compressed`
- void encode (const marked_graph &G, FILE *f)

  compresses a simple marked graph G, and writes the compressed form in a binary file f

Private Member Functions

- void encode_star_vertices ()

  encodes the star vertices (those vertices with at least one star edge connected to them)
- void extract_edge_types (const marked_graph &)

  Given a marked graph, extracts edge types by updating the colored_graph member C.
- void encode_star_edges ()

  Encodes star edges to the star_edges attribute of compressed.
- void encode_vertex_types ()

  encodes the type of vertices, where the type of a vertex denotes its mark as well as its degree matrix
- void find_part_index_deg ()

  update part_index and part_deg members
- void extract_partition_graphs ()

  by looking at the colored graph C, extract partition graphs (simple and bipartite)
- void encode_partition_bgraphs ()

  encode partition bipartite graphs
- void encode_partition_graphs ()

  encodes partition simple graphs

Private Attributes

- int h
- int delta
- int n

  number of vertices, this is set when a graph G is given to be encoded
- colored_graph C

  the auxiliary object to extract edge types
- vector< bool > is_star_vertex

  for $0 \leq v < n$, is_star_vertex[v] is true if there is at least one star type edge connected to v and false otherwise.
- vector< int > index_in_star

  for $0 \leq v < n$, if v is a star vertex, index_in_star[v] is the index of v among star vertices (this is to encode star edges)
- vector< int > star_vertices

  the list of star vertices
- map< pair< int, int >, b_graph > part_bgraph

  for $0 \leq t < t' < L$, part_bgraph[pair<int, int> (t,t')] is a bipartite graph with n left vertex and n right vertex. In this bipartite graph, a left vertex i is connected to a right vertex j iff there is an edge in the graph between vertices i and j with a half edge type towards i equal to t and a half edge type towards j equal to t'.

- vector< map< pair< int, int >, int > > part_index

  for a vertex $0 \leq v < n$, if v has a (t,t') edge connected to it. part_index[v][(t,t')] is the index of vertex v in the partition graph (or bipartite graph) corresponding to the pair (t,t'). If $t < t'$, this is the index of the left vertex corresponding to v in the partition bipartite graph, and if $t > t'$, this is the index of the right node corresponding to v in the bipartite partition graph.

- map< pair< int, int >, vector< int > > part_deg

  for a pair of types (t,t'), part_deg[(t,t')] is the degree sequence of the nodes in the partition graph corresponding to the pair t,t'. If $t < t'$, this is the degree sequence of the left nodes in the (t,t') partition bipartite graph, while if $t > t'$, this is the degree sequence of the right nodes in the (t', t) partition bipartite graph. Moreover, if t = t', this is the degree sequence of the (t,t) partition graph.

- map< int, graph > part_graph

  for $0 \leq t < L$, part_graph[i] is a simple unmarked graph with n vertices. In this graph, vertices i and j are connected in the original graph with an edge with half edge types t in both directions i and j.

- marked_graph_compressed compressed

  the compressed version of the given graph in encode function

6.17.1 Constructor & Destructor Documentation

6.17.1.1 marked_graph_encoder()

```
marked_graph_encoder::marked_graph_encoder (
            int h_,
            int delta_ )  [inline]
00148 : h(h_), delta(delta_) {}
```

6.17.2 Member Function Documentation

6.17.2.1 encode() [1/2]

```
marked_graph_compressed marked_graph_encoder::encode (
            const marked_graph & G )
```

compresses a simple marked graph G, and returns the compressed form as an object of type marked_graph_compressed

```
01331 {
01332   logger::current_depth++;
01333   logger::add_entry("Init compressed", "");
01334
01335   compressed.clear(); // reset the compressed variable before starting
01336
01337   n = G.nu_vertices;
01338   compressed.n = n;
01339   compressed.h = h;
01340   compressed.delta = delta;
01341
01342   logger::add_entry("Extact edge types", "");
01343   extract_edge_types(G);
01344   //cout << " edge types extracted " << endl;
01345
01346
01347   compressed.ver_type_list = C.ver_type_list; //
01348   compressed.type_mark = C.M.message_mark;
01349
01350   /*
01351   cout << " message list " << endl;
01352   for (int i=0;i<C.M.message_list.size();i++){
01353     cout << i << " : ";
01354     for (int j=0;j<C.M.message_list[i].size();j++)
01355       cout << C.M.message_list[i][j] << " ";
01356     cout << endl;
01357   }
01358   */
01359
01360   logger::add_entry("Encode * vertices", "");
```

```
01361    encode_star_vertices(); // encode the list of vertices with at least one star edge connected to them
01362    //cout << " encoded star vertices " << endl;
01363
01364    logger::add_entry("Encode * edges", "");
01365    encode_star_edges(); // encode edges with star types, i.e. those with half edge type L or larger
01366    //cout << " encoded star edges " << endl;
01367
01368    logger::add_entry("Encode vertex types", "");
01369    encode_vertex_types(); // encode the sequences \f$\vec{\beta}, \vec{D}\f$, which is encoded in
      C.ver_type
01370    //cout << " encoded vertex types " << endl;
01371
01372    logger::add_entry("Extract partition graphs", "");
01373    extract_partition_graphs(); // for equality types, we form simple unmarked graphs, and for
      inequality types, we form a bipartite graph
01374    //cout << " extracted partition graphs " << endl;
01375    /*
01376      cout << " partition bipartite graphs " << endl;
01377    for (map<pair<int, int>, b_graph>::iterator it = part_bgraph.begin(); it!=part_bgraph.end(); it++){
01378      cout << " c = " << it->first.first << " , " << it->first.second << endl;
01379      cout << it->second << endl;
01380    }
01381
01382    cout << " partition simple graphs " << endl;
01383    for (map<int, graph>::iterator it = part_graph.begin();it!=part_graph.end();it++){
01384      cout << " t = " << it->first << endl;
01385      cout << it->second << endl;
01386    }
01387    */
01388
01389    logger::add_entry("Encode partition b graphs", "");
01390    encode_partition_bgraphs();
01391    //cout << " encoded partition bgraphs " << endl;
01392
01393    logger::add_entry("Encode partition graphs", "");
01394    encode_partition_graphs();
01395    //cout << " encoded partition graphs " << endl;
01396
01397    logger::current_depth--;
01398    return compressed;
01399 }
```

6.17.2.2 encode() [2/2]

```
void marked_graph_encoder::encode (
            const marked_graph & G,
            FILE * f )
```

compresses a simple marked graph G, and writes the compressed form in a binary file f

```
01401                                                                        {
01402    logger::add_entry("Encode","");
01403    marked_graph_compressed comp = encode(G);
01404    logger::add_entry("Write to binary file", "");
01405    comp.binary_write(f);
01406 }
```

6.17.2.3 encode_partition_bgraphs()

```
void marked_graph_encoder::encode_partition_bgraphs ( )    [private]
```

encode partition bipartite graphs

```
01555 {
01556    int t, tp;
01557
01558    // compressing bipartite graphs
01559    for (map<pair<int, int> , b_graph>::iterator it = part_bgraph.begin(); it!=part_bgraph.end(); it++){
01560      // the color components are t, tp
01561      t = it->first.first;
01562      tp = it->first.second;
01563      b_graph_encoder E(part_deg.at(pair<int, int>(t,tp)),part_deg.at(pair<int, int>(tp, t)));
01564      compressed.part_bgraph[pair<int, int>(t,tp)] = E.encode(it->second);
01565    }
01566
01567 }
```

6.17.2.4 encode_partition_graphs()

```
void marked_graph_encoder::encode_partition_graphs ( )  [private]
```

encodes partition simple graphs

```
01570 {
01571   int t;
01572
01573   // compressing graphs
01574   for (map<int, graph>::iterator it=part_graph.begin(); it!=part_graph.end(); it++){
01575     t = it->first; // the color is t,t
01576     graph_encoder E(part_deg.at(pair<int, int>(t,t)));
01577     compressed.part_graph[t] = E.encode(it->second);
01578   }
01579 }
```

6.17.2.5 encode_star_edges()

```
void marked_graph_encoder::encode_star_edges ( )  [private]
```

Encodes star edges to the star_edges attribute of compressed.

```
01451 {
01452   int x, xp; // auxiliary mark variables
01453   int w; // auxiliary vertex variable
01454   int v; // auxiliary vertex variable
01455   for (int k=0; k<star_vertices.size(); k++){ // iterating over star vertices
01456     v = star_vertices[k];
01457     for (int i=0;i<C.adj_list[v].size();i++){
01458       if (C.M.is_star_message[C.adj_list[v][i].second.first] or
    C.M.is_star_message[C.adj_list[v][i].second.second]){ // this is a star edge
01459         x = C.M.message_mark[C.adj_list[v][i].second.first]; // mark towards v
01460         xp = C.M.message_mark[C.adj_list[v][i].second.second]; // mark towards other endpoint
01461         w = C.adj_list[v][i].first; // the other endpoint of the edge
01462         if (v < w){ // if v > w, we only store this edge when visiting the other endpoint (w), since
    we do not want to express an edge twice
01463           if (compressed.star_edges.find(pair<int, int>(x,xp)) == compressed.star_edges.end()) // this
    pair does not exist
01464             compressed.star_edges[pair<int, int>(x,xp)].resize(star_vertices.size()); // open space
    for all star vertices
01465           compressed.star_edges.at(pair<int, int>(x,xp))[k].push_back(index_in_star[w]); // add the
    index of w among star vertices to the position of v (which is k)
01466         }
01467       }
01468     }
01469   }
01470 }
```

6.17.2.6 encode_star_vertices()

```
void marked_graph_encoder::encode_star_vertices ( )  [private]
```

encodes the star vertices (those vertices with at least one star edge connected to them)

uses time_series_encode to encode the 0-1 sequence of star vertices stored in is_star_vertex to the star_vertices attribute of compressed

```
01431 {
01432   // compress the is_star_vertex list
01433   time_series_encoder star_encoder(n);
01434   vector<int> is_star_vertex_int(is_star_vertex.size());
01435   index_in_star.resize(n);
01436   int star_count = 0; // the number of star vertices
01437   for (int i=0;i<is_star_vertex.size();i++){
01438     if(is_star_vertex[i] == true){
01439       is_star_vertex_int[i] = 1;
01440       index_in_star[i] = star_count ++;
01441     }else{
01442       is_star_vertex_int[i] = 0;
01443     }
01444   }
01445   //for (int i=0;i<n;i++)
01446   //cout << is_star_vertex_int[i] << " : " << index_in_star[i] << endl;
01447   compressed.star_vertices = star_encoder.encode(is_star_vertex_int);
01448 }
```

6.17.2.7 encode_vertex_types()

```
void marked_graph_encoder::encode_vertex_types ( )  [private]
```

encodes the type of vertices, where the type of a vertex denotes its mark as well as its degree matrix

```
01409 {
01410   time_series_encoder vtype_encoder(n);
01411   //cerr « " C.ver_type_int " « endl;
01412   //for (int i=0;i<C.ver_type_int.size();i++)
01413   //  cerr « C.ver_type_int[i] « " ";
01414   //cerr « endl;
01415   compressed.ver_types = vtype_encoder.encode(C.ver_type_int);
01416 }
```

6.17.2.8 extract_edge_types()

```
void marked_graph_encoder::extract_edge_types (
              const marked_graph & G )  [private]
```

Given a marked graph, extracts edge types by updating the colored_graph member C.

```
01419 {
01420   // extracting edges types (aka colors)
01421   logger::current_depth++;
01422   logger::add_entry("Extract messages", "");
01423   C = colored_graph(G, h, delta);
01424   //cerr « " number of types " « C.M.message_mark.size() « endl;
01425   is_star_vertex = C.is_star_vertex;
01426   star_vertices = C.star_vertices;
01427   logger::current_depth--;
01428 }
```

6.17.2.9 extract_partition_graphs()

```
void marked_graph_encoder::extract_partition_graphs ( )  [private]
```

by looking at the colored graph C, extract partition graphs (simple and bipartite)

```
01495 {
01496   find_part_index_deg();
01497
01498   // for t \leq t', part_adj_list[(t,t')] is the adjacency list of the partition graph t,t'. If t <
       t', this is the adjacency list of the left nodes, if t = t', this is the forward adjacency list of the
       partition graph.
01499
01500   map<pair<int, int>, vector<vector<int> > > part_adj_list;
01501   int t, tp; // types
01502   for (map<pair<int, int>, vector<int> >::iterator it = part_deg.begin(); it!= part_deg.end(); it++){
01503     // search over all type pairs in part_deg
01504     t = it->first.first;
01505     tp = it->first.second;
01506     // t < t': bipartite, t = t': simple. In both cases,
01507     if (t <= tp)
01508       part_adj_list[it->first] = vector<vector<int> >(it->second.size());
01509   }
01510
01511   // going over the edges in the graph and forming partition_adj_list
01512   int w, p, q; // auxiliary variables
01513   for (int v =0; v<n; v++){
01514     for (int i=0;i<C.adj_list[v].size();i++){
01515       w = C.adj_list[v][i].first; // the other endpoint
01516       t = C.adj_list[v][i].second.first; // color towards v
01517       tp = C.adj_list[v][i].second.second; // color towards w
01518       if (C.M.is_star_message[t] == false and C.M.is_star_message[tp] == false){
01519         p = part_index[v].at(pair<int, int>(t,tp)); // the index of v in the t part of the t,tp
       partition graph
01520         //cerr « " p " « p « endl;
01521         q = part_index[w].at(pair<int, int>(tp, t)); // the index of w in the tp part of the t,tp
       partition graph
01522         //cerr « " q " « q « endl;
01523         if (t < tp)
01524           part_adj_list.at(pair<int, int>(t,tp))[p].push_back(q);
01525         if ((t == tp) and (q > p))
01526           part_adj_list.at(pair<int, int>(t,t))[p].push_back(q);
01527       }
```

```
01528      }
01529    }
01530
01531    // using partition_adj_list in order to construct partition graphs
01532    //if(logger::stat){
01533    //*logger::stat_stream « " partition graphs size: " « endl;
01534    //*logger::stat_stream « " ====================== " « endl;
01535    //}
01536    for (map<pair<int, int>, vector<vector<int> > >::iterator it=part_adj_list.begin();
     it!=part_adj_list.end();it++){
01537      t = it->first.first;
01538      tp = it->first.second;
01539      if (t<tp){
01540        part_bgraph[it->first] = b_graph(it->second, part_deg.at(pair<int, int>(t,tp)),
     part_deg.at(pair<int, int>(tp, t))); // left and right degree sequences are read from the part_deg map
01541        //if (logger::stat){
01542        //*logger::stat_stream « " bipartite: (" « part_deg.at(pair<int, int> (t,tp)).size() « " , " «
     part_deg.at(pair<int, int>(tp,t)).size() « ")" « endl;
01543        //}
01544      }
01545      if (t == tp){
01546        part_graph[t] = graph(it->second, part_deg.at(pair<int, int>(t,t)));
01547        //if (logger::stat){
01548        //*logger::stat_stream « " simple: "  « part_deg.at(pair<int, int>(t,t)).size() « endl;
01549        //}
01550      }
01551    }
01552 }
```

6.17.2.10   find_part_index_deg()

void marked_graph_encoder::find_part_index_deg ( )   [private]

update part_index and part_deg members

```
01473 {
01474    // extracting part_index and part_deg
01475    part_index.resize(n);
01476    for (int v =0; v<n; v++){
01477      for (map< pair< int, int >, int >::iterator it = C.deg[v].begin(); it != C.deg[v].end(); it++){
01478        if (part_deg.find(it->first) == part_deg.end()){
01479          // this pair has not been observed yet in the graph
01480          // so v is the first index node
01481          part_index[v][it->first] = 0;
01482          // the degree of v in the partition graph is indeed it->second
01483          part_deg[it->first] = vector<int>({it->second});
01484        }else{
01485          // there are currently part_deg[it->first].size() many elements there, and v is the last
     arrival one, so its index is equal to the number of existing nodes
01486          part_index[v][it->first] = part_deg.at(it->first).size();
01487          // append degree of v, which is it->second
01488          part_deg.at(it->first).push_back(it->second);
01489        }
01490      }
01491    }
01492 }
```

6.17.3   Member Data Documentation

6.17.3.1   C

colored_graph marked_graph_encoder::C [private]

the auxiliary object to extract edge types

6.17.3.2   compressed

marked_graph_compressed marked_graph_encoder::compressed [private]

the compressed version of the given graph in encode function

6.17.3.3 delta

int marked_graph_encoder::delta [private]

6.17.3.4 h

int marked_graph_encoder::h [private]

6.17.3.5 index_in_star

vector<int> marked_graph_encoder::index_in_star [private]

for $0 \le v < n$, if v is a star vertex, index_in_star[v] is the index of v among star vertices (this is to encode star edges)

6.17.3.6 is_star_vertex

vector<bool> marked_graph_encoder::is_star_vertex [private]

for $0 \le v < n$, is_star_vertex[v] is true if there is at least one star type edge connected to v and false otherwise.

6.17.3.7 n

int marked_graph_encoder::n [private]

number of vertices, this is set when a graph G is given to be encoded

6.17.3.8 part_bgraph

map<pair<int, int>, b_graph> marked_graph_encoder::part_bgraph [private]

for $0 \le t < t' < L$, part_bgraph[pair<int, int> (t,t')] is a bipartite graph with n left vertex and n right vertex. In this bipartite graph, a left vertex i is connected to a right vertex j iff there is an edge in the graph between vertices i and j with a half edge type towards i equal to t and a half edge type towards j equal to t'.

6.17.3.9 part_deg

map<pair<int, int>, vector<int> > marked_graph_encoder::part_deg [private]

for a pair of types (t,t'), part_deg[(t,t')] is the degree sequence of the nodes in the partition graph corresponding to the pair t,t'. If t < t', this is the degree sequence of the left nodes in the (t,t') partition bipartite graph, while if t > t', this is the degree sequence of the right nodes in the (t', t) partition bipartite graph. Moreover, if t = t', this is the degree sequence of the (t,t) partition graph.

6.17.3.10 part_graph

`map<int, graph> marked_graph_encoder::part_graph [private]`

for $0 \leq t < L$, part_graph[i] is a simple unmarked graph with n vertices. In this graph, vertices i and j are connected in the original graph with an edge with half edge types t in both directions i and j.

6.17.3.11 part_index

`vector<map<pair<int, int>, int> > marked_graph_encoder::part_index [private]`

for a vertex $0 \leq v < n$, if v has a (t,t') edge connected to it. part_index[v][(t,t')] is the index of vertex v in the partition graph (or bipartite graph) corresponding to the pair (t,t'). If t < t', this is the index of the left vertex corresponding to v in the partition bipartite graph, and if t > t', this is the index of the right node corresponding to v in the bipartite partition graph.

6.17.3.12 star_vertices

`vector<int> marked_graph_encoder::star_vertices [private]`

the list of star vertices

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

- marked_graph_compression.h
- marked_graph_compression.cpp

6.18 obitstream Class Reference

handles writing bitstreams to binary files

`#include <bitstream.h>`

Public Member Functions

- obitstream (string file_name)

    constructor
- void write_bits (unsigned int n, unsigned int nu_bits)

    write the bits given as unsigned int to the output
- obitstream & operator<< (const unsigned int &n)

    uses Elias delta code to write a nonnegative integer to the output. In order to make sure that n >= 1, we effectively encode n + 1 instead
- obitstream & operator<< (const mpz_class &n)

    uses Elias delta code to write a nonnegative mpz_class integer to the output. In order to make sure that n >= 1, we effectively encode n + 1 instead
- void bin_inter_code (const vector< int > &a, int b)

    uses binary interpolative coding to encode an increasing sequence of integers
- void bin_inter_code (const vector< int > &a, int i, int j, int low, int high)

    binary interpolative coding for array a, interval [i,j], where values are in the range [low, high]
- unsigned int chunks ()

    returns the number of chunks (each is BIT_INT = 32 bits) to the output.
- void close ()

    closes the session by writing the remaining chunk to the output (if any) and closing the file pointer f

Private Member Functions

- void write ()

  writes complete chunks to the output

Private Attributes

- bit_pipe buffer

  a bit_pipe carrying the buffered data
- FILE ∗ f
- unsigned int chunks_written

  the number of chunks written to the output so far

6.18.1 Detailed Description

handles writing bitstreams to binary files

When trying to write to binary files, we sometimes need to write less than a byte, or a few bytes followed by say 2 bits. This is not possible unless we turn those 2 bits to 8 bits by basically adding 6 zeros. . However, if we want to do a lot of such operations, this can result in space inefficiencies. To avoid this, we can concatenate the bitstreams together and perhaps gain a lot in space. This class also handles Elias delta encoding of unsigned int and mpz↩ _class. The way it is done is to buffer the data, write complete bytes to the output, and keeping the residuals for future operations.

In order to make sure that the carry over from the last operation is also written to the output, we should call the close() function.

6.18.2 Constructor & Destructor Documentation

6.18.2.1 obitstream()

```
obitstream::obitstream (
            string file_name )  [inline]
```

constructor
```
00089                                    {
00090     f = fopen(file_name.c_str(), "wb+");
00091     chunks_written = 0;
00092   }
```

6.18.3 Member Function Documentation

6.18.3.1 bin_inter_code() [1/2]

```
void obitstream::bin_inter_code (
            const vector< int > & a,
            int b )
```

uses binary interpolative coding to encode an increasing sequence of integers

We use the binary interpolative coding algorithm introduced by Moffat and Stuiver, reference:

Moffat, Alistair, and Lang Stuiver. "Binary interpolative coding for effective index compression." Information Retrieval 3.1 (2000): 25-47.

Parameters

a	array of nonnegative increasing integers (this function assumes a contains nonnegative increasing integers, and does not check it)
b	an upper bound on the number of bits necessary to encode values in a and the size of a

```
00267                                                                              {
00268   // write a.size
00269
00270   write_bits(a.size(),b);
00271   if (a.size() == 0)
00272     return;
00273   if (a.size()==1){
00274     write_bits(a[0],b);
00275     return;
00276   }
00277   // write low and high values in a
00278   write_bits(a[0], b);
00279   write_bits(a[a.size()-1], b);
00280
00281   // then, encode recursively
00282   bin_inter_code(a, 0, a.size()-1, a[0], a[a.size()-1]);
00283 }
```

6.18.3.2 bin_inter_code() [2/2]

```
void obitstream::bin_inter_code (
              const vector< int > & a,
              int i,
              int j,
              int low,
              int high )
```

binary interpolative coding for array a, interval [i,j], where values are in the range [low, high]

Parameters

a	array of increasing nonnegative integers
i,j	endpoints of the interval to be encoded
low	lower bound for the integers in the interval [i,j]
high	upper bound for the integers in the interval [i,j]

```
00291                                                                              {
00292   if (j < i)
00293     return;
00294   if (i==j){
00295     // we should encode a[i] using the assumption that it is bounded by high - low
00296     // therefore low <= a[i] <= high
00297     // so 0 <= a[i]-low <= high - low
00298     // so we can encode a[i]-low using nu_bits(high - low) bits
00299     if (high > low) // otherwise, there will be nothing to be printed
00300       write_bits(a[i] - low, nu_bits(high-low));
00301     return;
00302   }
00303   // find the intermediate value
00304   int m = (i+j)/ 2;
00305   unsigned int L = low + m - i; // lower bound on a[m]
00306   unsigned int H = high - (j - m); // upper bound on a[m]
00307   // so L <= a[m] <= H
00308   // and we can encode a[m] - L using nu_bits(H-L) bits
00309   if (H > L) // otherwise, a[m] is clearly H = L and nothing need to be written
00310     write_bits(a[m] - L, nu_bits(H-L));
00311
00312   // then, we should recursively encode intervals [i,m-1] and [m+1, j]
00313   bin_inter_code(a, i, m-1, low, a[m]-1);
00314   bin_inter_code(a, m+1, j, a[m]+1, high);
00315 }
```

6.18.3.3 chunks()

```
unsigned int obitstream::chunks ( )  [inline]
```

returns the number of chunks (each is BIT_INT = 32 bits) to the output.

```
00109                              {
00110      return chunks_written;
00111    }
```

6.18.3.4 close()

```
void obitstream::close ( )
```

closes the session by writing the remaining chunk to the output (if any) and closing the file pointer f

```
00318                              {
00319    if (buffer.bits.size() > 0){
00320      fwrite(&buffer.bits[0], sizeof(unsigned int), buffer.bits.size(), f);
00321      buffer.bits.clear();
00322      buffer.last_bits = 0;
00323    }
00324    fclose(f);
00325 }
```

6.18.3.5 operator<<() [1/2]

```
obitstream & obitstream::operator<< (
            const mpz_class & n )
```

uses Elias delta code to write a nonnegative mpz_class integer to the output. In order to make sure that n >= 1, we effectively encode n + 1 instead

```
00242                                                  {
00243    if (buffer.bits.size() > 1){
00244      cerr « " ERROR: buffer has more than 1 chunk! " « endl;
00245    }
00246    unsigned int buffer_backup = 0; // the backup of the remaining chunk in
00247    int buffer_res = 0;
00248    if (buffer.bits.size() != 0){
00249      buffer_backup = buffer.bits[0];
00250      buffer_res = buffer.last_bits;
00251    }
00252    elias_delta_encode(n+1, buffer); // find the delta encoded version of n + 1
00253    buffer.shift_right(buffer_res); // open up space for the residual of the previous operation
00254    buffer.bits[0] |= buffer_backup; // add the residual
00255    write();
00256    return *this;
00257 }
```

6.18.3.6 operator<<() [2/2]

```
obitstream & obitstream::operator<< (
            const unsigned int & n )
```

uses Elias delta code to write a nonnegative integer to the output. In order to make sure that n >= 1, we effectively encode n + 1 instead

```
00225                                                  {
00226    if (buffer.bits.size() > 1){
00227      cerr « " ERROR: buffer has more than 1 chunk! " « endl;
00228    }
00229    unsigned int buffer_backup = 0; // the backup of the remaining chunk in
00230    int buffer_res = 0;
00231    if (buffer.bits.size() != 0){
00232      buffer_backup = buffer.bits[0];
00233      buffer_res = buffer.last_bits;
00234    }
00235    elias_delta_encode(n+1, buffer); // find the delta encoded version of n + 1
00236    buffer.shift_right(buffer_res); // open up space for the residual of the previous operation
00237    buffer.bits[0] |= buffer_backup; // add the residual
00238    write();
00239    return *this;
00240 }
```

6.18.3.7 write()

```
void obitstream::write ( ) [private]
```

writes complete chunks to the output

```
00194                          {
00195   if (buffer.bits.size() > 1){
00196     // write the first chunks to the output
00197     fwrite(&buffer.bits[0], sizeof(unsigned int), buffer.bits.size()-1, f);
00198     // add the number of chunks written to chunks_written
00199     chunks_written += buffer.bits.size() -1;
00200     // then, remove the first buffer.bits.size()-1 chunks which were written to the output
00201     buffer.bits.erase(buffer.bits.begin(), buffer.bits.begin() + buffer.bits.size()-1);
00202   }
00203 }
```

6.18.3.8 write_bits()

```
void obitstream::write_bits (
            unsigned int n,
            unsigned int nu_bits )
```

write the bits given as unsigned int to the output

Parameters

n	bits to be written to the output in the form of an unsigned int (4 bytes of data)
nu_bits	number of bits, counted from LSB of n, to write to the output. For instance if n = 1 and nu_bits = 1, a single bit with value 1 is written

```
00209                                                              {
00210   unsigned int buffer_backup = 0; // the backup of the remaining chunk in
00211   int buffer_res = 0; // number of bits remaining in the buffer
00212   if (buffer.bits.size() != 0){
00213     buffer_backup = buffer.bits[0];
00214     buffer_res = buffer.last_bits;
00215   }
00216   buffer.bits.resize(1);
00217   buffer.bits[0] = n << (BIT_INT - nu_bits); // shift left so that exactly nu_bits many bits are in the
    buffer
00218   buffer.last_bits = nu_bits;
00219   buffer.shift_right(buffer_res); // open up space for the residual of the previous operation
00220   buffer.bits[0] |= buffer_backup; // add the residual
00221   write(); // write the buffer to the output
00222 }
```

6.18.4 Member Data Documentation

6.18.4.1 buffer

```
bit_pipe obitstream::buffer [private]
```

a bit_pipe carrying the buffered data

6.18.4.2 chunks_written

```
unsigned int obitstream::chunks_written [private]
```

the number of chunks written to the output so far

6.18.4.3 f

```
FILE* obitstream::f  [private]
```

pointer to the binary output file

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

- bitstream.h
- bitstream.cpp

6.19   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.

6.19.1   Detailed Description

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

6.19.2   Constructor & Destructor Documentation

6.19.2.1   reverse_fenwick_tree() [1/2]

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

default constructor
```
00058 {}
```

6.19.2.2 reverse_fenwick_tree() `[2/2]`

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

constructor which receives values and initializes

```
00047 {
00048   reverse(vals.begin(),vals.end()); // reverse the array and then use the previously defined
        fenwick_tree class
00049   FT = fenwick_tree(vals);
00050 }
```

6.19.3 Member Function Documentation

6.19.3.1 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

```
00053 {
00054   FT.add(FT.size() - 1 - k, val);
00055 }
```

6.19.3.2 size()

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

the number of elements in the original array

```
00070             {
00071     return FT.size();
00072   }
```

6.19.3.3 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

```
00059 {
00060   if (k >= size())
00061     return 0;
00062   return FT.sum(FT.size() - 1 - k);
```

```
00063 }
```

6.19.4   Member Data Documentation

6.19.4.1   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

6.20   time_series_decoder Class Reference

decodes a time series which is basically an array of arbitrary nonnegative integers

```
#include <time_series_compression.h>
```

Public Member Functions

- time_series_decoder (int n_)

    constructor
- vector< int > decode (pair< vector< int >, mpz_class >)

    inputs an object of type pair<vector<int>, mpz_class> *generated by* time_series_encoder *and returns the decoded array.*

Private Attributes

- int n

    the length
- int alph_size

    the number of distinct integers showing up in the sequence after decoding. Therefore, the sequence would consists of integers in the range [0,alph_size-1].
- vector< int > freq

    the frequency of symbols after decoding, so has size alph_size
- b_graph G

    the decoded bipartite graph as in time_series_encoder

6.20.1 Detailed Description

decodes a time series which is basically an array of arbitrary nonnegative integers

This class is capable of decompressing arrays of nonnegative integers with size n. Upon construction, n must be given. But later, the object is capable of decompressing any sequence with this size, universally. The input would be the output of `time_series_decoder` class.

Usage Example
```
vector<int> a = {0,2,3,1,2,1,0,1,0,2,1,0,0,2,1,3,4,5,0};
int n = a.size();
time_series_encoder E(n);
pair<vector<int>, mpz_class > ans = E.encode(a);

time_series_decoder D(n);
vector<int> ahat = D.decode(ans);
if (ahat == a)
    cout « " successfully decoded the original time series! " « endl;
```

6.20.2 Constructor & Destructor Documentation

6.20.2.1 time_series_decoder()

```
time_series_decoder::time_series_decoder (
            int n_ )  [inline]
```

constructor
```
00090 : n(n_) {}
```

6.20.3 Member Function Documentation

6.20.3.1 decode()

```
vector< int > time_series_decoder::decode (
            pair< vector< int >, mpz_class > E )
```

inputs an object of type `pair<vector<int>, mpz_class>` generated by `time_series_encoder` and returns the decoded array.
```
00077 {
00078    freq = E.first;
00079    mpz_class f = E.second;
00080    vector<int> left_deg(n,1); // the left degree sequence
00081    b_graph_decoder D(left_deg, freq); // the bipartite graph decoder to convert f to G
00082    G = D.decode(f);
00083
00084    // reconstructing the original sequence given G
00085    vector<int> x(n);
00086    vector<int> adj_list;
00087    for (int i=0;i<n;i++){
00088      adj_list = G.get_adj_list(i);
00089      x[i] = adj_list[0];
00090    }
00091    return x;
00092 }
```

6.20.4 Member Data Documentation

6.20.4.1 alph_size

```
int time_series_decoder::alph_size  [private]
```

the number of distinct integers showing up in the sequence after decoding. Therefore, the sequence would consists of integers in the range [0,alph_size-1].

6.20.4.2 freq

`vector<int> time_series_decoder::freq [private]`

the frequency of symbols after decoding, so has size alph_size

6.20.4.3 G

`b_graph time_series_decoder::G [private]`

the decoded bipartite graph as in `time_series_encoder`

6.20.4.4 n

`int time_series_decoder::n [private]`

the length

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

- time_series_compression.h
- time_series_compression.cpp

6.21 time_series_encoder Class Reference

encodes a time series which is basically an array of arbitrary nonnegative integers

`#include <time_series_compression.h>`

Public Member Functions

- time_series_encoder (int n_)
    - *constructor*
- pair< vector< int >, mpz_class > encode (const vector< int > &x)
    - *Encodes a `vector<int>` with size n.*

Private Member Functions

- void init_alph_size (const vector< int > &x)
    - *initializes the alphabet size, i.e. the variable init_alph_size*
- void init_freq (const vector< int > &x)
    - *initializes the freq vector*
- void init_G (const vector< int > &x)
    - *initializes the auxiliary bipartite graph G*

Private Attributes

- int n

    length of the series is assumed to be known

- int alph_size

    the number of distinct integers showing up in the sequence. Therefore, the sequence would consists of integers in the range [0,alph_size-1].

- vector< int > freq

    the frequency of symbols, so has size alph_size

- b_graph G

    the bipartite graph version of the sequence, which has n left vertices and alph_size right sequence. Left vertex v is connected to right vertex w if the value of the time series in coordinate v is w. This way, each vertex on the left has degree 1, and the degree of a right vertex w is the frequency of w.

6.21.1 Detailed Description

encodes a time series which is basically an array of arbitrary nonnegative integers

This class is capable of compressing arrays of nonnegative integers with size n. Upon construction, n must be given. But later, the object is capable of compressing any sequence with this size, universally. The output of the compression is an object of type `pair<vector<int>, mpz_class>`.

Usage Example
```
vector<int> a = {0,2,3,1,2,1,0,1,0,2,1,0,0,2,1,3,4,5,0};
int n = a.size();
time_series_encoder E(n);
pair<vector<int>, mpz_class > ans = E.encode(a);
```

See time_series_decoder for decoding.

6.21.2 Constructor & Destructor Documentation

6.21.2.1 time_series_encoder()

```
time_series_encoder::time_series_encoder (
            int n_ )  [inline]
```

constructor
```
00044 : n(n_) {}
```

6.21.3 Member Function Documentation

6.21.3.1 encode()

```
pair< vector< int >, mpz_class > time_series_encoder::encode (
            const vector< int > & x )
```

Encodes a `vector<int>` with size n.

Parameters

x	const reference to the array to be compressed.

Returns

> an object of type `pair<vector<int>, mpz_class>`. The first part is the corresponding frequency array (`freq` member), and the second is the compressed form of the bipartite graph G

```
00035 {
00036   //check whether x is a compatible sequence
00037   if (x.size()!= n)
00038     cerr « " WARNING: time_series_encoder::encode, called for a vector with size different from n,
    x.size() = " « x.size() « endl;
00039
00040   // initialize alph_size, freq and G
00041   //logger::item_start("time series init alph size");
00042   init_alph_size(x);
00043   //logger::item_stop("time series init alph size");
00044   //logger::item_start("time series init freq");
00045   init_freq(x);
00046   //logger::item_stop("time series init freq");
00047
00048   //logger::item_start("time series init G");
00049   init_G(x);
00050   //logger::item_stop("time series init G");
00051
00052   // initializing a b_graph_encoder
00053   vector<int> left_deg(n, 1); // the left degree sequence
00054   b_graph_encoder E(left_deg, freq); // the right degree sequence is freq
00055   //logger::item_start("time series encode");
00056   mpz_class f = E.encode(G);
00057   //logger::item_stop("time series encode");
00058   pair<vector<int>, mpz_class> ans;
00059   ans.first = freq;
00060   ans.second  = f;
00061   return ans;
00062 }
```

6.21.3.2 init_alph_size()

```
void time_series_encoder::init_alph_size (
            const vector< int > & x )  [private]
```

initializes the alphabet size, i.e. the variable init_alph_size

```
00004 {
00005   alph_size = 0;
00006   for (int i=0;i<x.size();i++){
00007     if (x[i] > alph_size)
00008       alph_size = x[i];
00009     if (x[i] < 0)
00010       cerr « " WARNING: time_series_encoder::encode called for a vector with negative entries " «
    endl;
00011   }
00012
00013   alph_size ++; // the array is zero based
00014 }
```

6.21.3.3 init_freq()

```
void time_series_encoder::init_freq (
            const vector< int > & x )  [private]
```

initializes the freq vector

```
00017 {
00018   freq.clear();
00019   freq.resize(alph_size); //assuming that alph_size is already set
00020   for (int i=0;i<x.size();i++)
00021     freq[x[i]] ++;
00022 }
```

6.21.3.4 init_G()

```
void time_series_encoder::init_G (
            const vector< int > & x ) [private]
```

initializes the auxiliary bipartite graph G

```
00025 {
00026   //initializing the adjacency list
00027   vector<vector<int> > list;
00028   list.resize(n);
00029   for (int i=0;i<x.size();i++)
00030     list[i] = vector<int>({x[i]}); // list[i] has a single member, which is x[i]. In other words, the
      left vertex i has only one right neighbor, which is precisely x[i]
00031   G = b_graph(list, freq); // construct G based on its adjacency list, and the right degree sequence
      which is freq
00032 }
```

6.21.4 Member Data Documentation

6.21.4.1 alph_size

```
int time_series_encoder::alph_size [private]
```

the number of distinct integers showing up in the sequence. Therefore, the sequence would consists of integers in the range [0,alph_size-1].

6.21.4.2 freq

```
vector<int> time_series_encoder::freq [private]
```

the frequency of symbols, so has size alph_size

6.21.4.3 G

```
b_graph time_series_encoder::G [private]
```

the bipartite graph version of the sequence, which has n left vertices and alph_size right sequence. Left vertex v is connected to right vertex w if the value of the time series in coordinate v is w. This way, each vertex on the left has degree 1, and the degree of a right vertex w is the frequency of w.

6.21.4.4 n

```
int time_series_encoder::n [private]
```

length of the series is assumed to be known

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

- time_series_compression.h
- time_series_compression.cpp

6.22   vint_hash Struct Reference

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

Public Member Functions

- size_t operator() (vector< int > const &v) const

6.22.1   Member Function Documentation

6.22.1.1   operator()()

```
size_t vint_hash::operator() (
            vector< int > const & v ) const
00003                                                                {
00004   return boost::hash_range(v.begin(),v.end());
00005 }
```

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

- graph_message.h
- graph_message.cpp

Chapter 7

File Documentation

7.1 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)

7.1.1 Function Documentation

7.1.1.1 operator"!=()

```
bool operator!= (
            const b_graph & G1,
            const b_graph & G2 )
00131 {
00132   return !(G1 == G2);
00133 }
```

7.1.1.2 operator<<()

```
ostream & operator<< (
            ostream & o,
            const b_graph & G )
00095 {
00096   int n = G.nu_left_vertices();
00097   vector<int> list;
00098   for (int i=0;i<n;i++){
00099     list = G.get_adj_list(i);
00100     o « i « " -> ";
00101     for (int j=0;j<list.size();j++){
00102       o « list[j];
00103       if (j < list.size()-1)
00104         o « ", ";
00105     }
00106     o « endl;
00107   }
00108   return o;
00109 }
```

7.1.1.3 operator==()

```
bool operator== (
            const b_graph & G1,
            const b_graph & G2 )
00112 {
00113   int n1 = G1.nu_left_vertices();
00114   int n2 = G2.nu_left_vertices();
00115
00116   int np1 = G1.nu_right_vertices();
00117   int np2 = G2.nu_right_vertices();
00118   if (n1!= n2 or np1 != np2)
00119     return false;
00120   vector<int> list1, list2;
00121   for (int v=0; v<n1; v++){
00122     list1 = G1.get_adj_list(v);
00123     list2 = G2.get_adj_list(v);
00124     if (list1 != list2)
00125       return false;
00126   }
00127   return true;
00128 }
```

7.2 bipartite_graph.h File Reference

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

Classes

- class b_graph

    simple unmarked bipartite graph

7.3 bipartite_graph.h

Go to the documentation of this file.
```
00001 #ifndef __BIPARTITE_GRAPH__
00002 #define __BIPARTITE_GRAPH__
00003
00004 #include <iostream>
00005 #include <vector>
00006 using namespace std;
00007
00009
00024 class b_graph{
00025   int n;
00026   int np;
00027   vector<vector<int> > adj_list;
00028   vector<int> left_deg_seq;
00029   vector<int> right_deg_seq;
00030 public:
00031
00033  b_graph(): n(0), np(0) {}
00034
00036
00042   b_graph(const vector<vector<int> > &list, const vector<int> &left_deg, const vector<int>
      &right_deg);
00043
00045
00050   b_graph(const vector<vector<int> > &list, const vector<int> &right_deg);
00051
00053
00057   b_graph(const vector<vector<int> > &list);
00058
00059
00061   vector<int> get_adj_list(int v) const;
```

```
00062
00064    int get_right_degree(int v) const;
00065
00067    int get_left_degree(int v) const;
00068
00070    vector<int> get_right_degree_sequence() const;
00071
00073    vector<int> get_left_degree_sequence() const;
00074
00076    int nu_left_vertices() const;
00077
00079    int nu_right_vertices() const;
00080
00082    friend ostream& operator << (ostream& o, const b_graph& G);
00083
00085    friend bool operator == (const b_graph& G1, const b_graph& G2);
00086
00087
00089    friend bool operator != (const b_graph& G1, const b_graph& G2);
00090 };
00091
00092
00093 #endif
```

7.4 bipartite_graph_compression.cpp File Reference

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

7.5 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.

7.6 bipartite_graph_compression.h

Go to the documentation of this file.
```
00001 #ifndef __BIPARTITE_GRAPH_COMPRESSION__
00002 #define __BIPARTITE_GRAPH_COMPRESSION__
00003
00004 #include <iostream>
00005 #include <vector>
00006 #include "compression_helper.h"
00007 #include "bipartite_graph.h"
00008 #include "fenwick.h"
00009
00010 using namespace std;
00011
00013
```

```
00036 class b_graph_encoder
00037 {
00038   //const b_graph& G; //!< the simple unmarked bipartite graph to be encoded
00039   vector<int> beta;
00040   vector<int> a;
00041   vector<int> b;
00042   reverse_fenwick_tree U;
00043 public:
00044
00046  b_graph_encoder(vector<int> a_, vector<int> b_): a(a_), b(b_) {}
00047
00049   void init(const b_graph& G);
00050
00051
00053
00058   pair<mpz_class, mpz_class> compute_N(const b_graph& G);
00059
00061   mpz_class encode(const b_graph& G);
00062
00063 };
00064
00066
00100 class b_graph_decoder
00101 {
00102   int n;
00103   int np;
00104   vector<int> a;
00105   vector<int> b;
00106   vector<vector<int> > x;
00107   reverse_fenwick_tree U;
00108   reverse_fenwick_tree W;
00109   vector<int> beta;
00110
00111 public:
00113   b_graph_decoder(vector<int> a_, vector<int> b_);
00114
00116   void init();
00117
00119
00124   pair<mpz_class, mpz_class> decode_node(int i, mpz_class tN);
00125
00127
00132   pair<mpz_class, mpz_class> decode_interval(int i, int j, mpz_class tN);
00133
00135
00139   b_graph decode(mpz_class f);
00140 };
00141
00142
00143 #endif
```

7.7 bitstream.cpp File Reference

```
#include "bitstream.h"
```

Functions

- ostream & operator<< (ostream &o, const bit_pipe &B)
- bit_pipe operator<< (const bit_pipe &B, int n)
- bit_pipe operator>> (const bit_pipe &B, int n)
- unsigned int nu_bits (unsigned int n)

  returns number of bits in a positive integer n, e.g. 3 has 3 bits, 12 has 4 bits, and 0 has 0 bits.
- unsigned int mask_gen (int n)

  generates a binary mask with n consecutive ones in LSB
- bit_pipe elias_delta_encode (const unsigned int &n)

  returns the Elias delta representation of an integer in bit_pipe format
- void elias_delta_encode (const unsigned int &n, bit_pipe &B)

  performs Elias delta encode for an integer, and stores the results in the given reference to bit_pipe objects

- bit_pipe elias_delta_encode (const mpz_class &n)

    returns the Elias delta representation of an mpz_class in bit_pipe format
- void elias_delta_encode (const mpz_class &n, bit_pipe &B)

    performs Elias delta encoding on n, and stores the results in the given reference to bit_pipe objects

7.7.1 Function Documentation

7.7.1.1 elias_delta_encode() [1/4]

```
bit_pipe elias_delta_encode (
              const mpz_class & n )
```

returns the Elias delta representation of an mpz_class in bit_pipe format

```
00694                                                        {
00695    if (n == 0){
00696      cerr « " ERROR: elias delta called for 0, input must be a positive integer" « endl;
00697    }
00698    // first, find number of bits in n
00699    int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
00700    int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00701
00702    bit_pipe N(n_bits); // binary representation
00703    N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00704    bit_pipe n_pipe(n); // binary representation of n
00705    n_pipe.shift_left(1); // remove the leading 1
00706    n_pipe.append_left(N);
00707    return n_pipe;
00708 }
```

7.7.1.2 elias_delta_encode() [2/4]

```
void elias_delta_encode (
              const mpz_class & n,
              bit_pipe & B )
```

performs Elias delta encoding on n, and stores the results in the given reference to bit_pipe objects

```
00710                                                        {
00711    if (n == 0){
00712      cerr « " ERROR: elias delta called for 0, input must be a positive integer" « endl;
00713    }
00714    // first, find number of bits in n
00715    int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
00716    int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00717
00718    bit_pipe N(n_bits); // binary representation
00719    N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00720    B = bit_pipe(n); // binary representation of n
00721    B.shift_left(1); // remove the leading 1
00722    B.append_left(N);
00723 }
```

7.7.1.3 elias_delta_encode() [3/4]

```
bit_pipe elias_delta_encode (
              const unsigned int & n )
```

returns the Elias delta representation of an integer in bit_pipe format

```
00662                                                        {
00663    if (n == 0){
00664      cerr « " ERROR: elias delta called for 0, input must be a positive integer" « endl;
00665    }
00666    // first, find number of bits in n
```

```
00667   int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
00668   int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00669
00670   bit_pipe N(n_bits); // binary representation
00671   N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00672   bit_pipe n_pipe(n); // binary representation of n
00673   n_pipe.shift_left(1); // remove the leading 1
00674   n_pipe.append_left(N);
00675   return n_pipe;
00676 }
```

7.7.1.4 elias_delta_encode() [4/4]

```
void elias_delta_encode (
            const unsigned int & n,
            bit_pipe & B )
```

performs Elias delta encode for an integer, and stores the results in the given reference to bit_pipe objects

```
00678                                                    {
00679   if (n == 0){
00680     cerr « " ERROR: elias delta called for 0, input must be a positive integer" « endl;
00681   }
00682   // first, find number of bits in n
00683   int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
00684   int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00685
00686   bit_pipe N(n_bits); // binary representation
00687   N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00688   B = bit_pipe(n); // binary representation of n
00689   B.shift_left(1); // remove the leading 1
00690   B.append_left(N);
00691 }
```

7.7.1.5 mask_gen()

```
unsigned int mask_gen (
            int n )
```

generates a binary mask with n consecutive ones in LSB

Example: n = 1 -> 00000001, n = 7 -> 01111111

```
00649                                          {
00650   if (n < 1 or n > BIT_INT){
00651     cerr « " ERROR: mask_gen called for n outside the range [1,BIT_INT] " « endl;
00652     return 0;
00653   }
00654   unsigned int mask = 1;
00655   for (int i=1; i<n; i++){
00656     mask «= 1;
00657     mask += 1;
00658   }
00659   return mask;
00660 }
```

7.7.1.6 nu_bits()

```
unsigned int nu_bits (
            unsigned int n )
```

returns number of bits in a positive integer n, e.g. 3 has 3 bits, 12 has 4 bits, and 0 has 0 bits.

This is in fact nothing but $\lfloor \log_2 n \rfloor + 1$

```
00635                                          {
00636   int nu_bits = 0;
00637   unsigned int n_copy = n;
00638   while (n_copy > 0){
00639     nu_bits ++;
00640     n_copy »= 1;
00641   }
00642   return nu_bits;
00643 }
```

7.7.1.7 operator<<() [1/2]

```
bit_pipe operator<< (
                const bit_pipe & B,
                int n )
00166                                                    {
00167   bit_pipe ans = B;
00168   ans.shift_left(n);
00169   return ans;
00170 }
```

7.7.1.8 operator<<() [2/2]

```
ostream & operator<< (
                ostream & o,
                const bit_pipe & B )
00112                                                    {
00113   if (B.bits.size()==0){
00114     o << "<>";
00115     return o;
00116   }
00117   o << "<";
00118   for (int i=0;i<(B.bits.size()-1); i++){ // the last byte requires special handling
00119     bitset<BIT_INT> b(B.bits[i]);
00120     o << b << " ";
00121   }
00122   unsigned int last_byte = B.bits[B.bits.size()-1];
00123
00124   for (int k=BIT_INT;k>(BIT_INT-B.last_bits);k--){ // starting from MSB bit to LSB for existing bits
00125     if (last_byte & (1<<(k-1)))
00126       o << "1";
00127     else
00128       o << "0";
00129   }
00130   o << "|"; // to show the place of the last bit
00131   for (int k=BIT_INT-B.last_bits; k>=1; k--){
00132     if (last_byte &(1<<(k-1)))
00133       o << "1";
00134     else
00135       o << "0";
00136   }
00137   o << ">";
00138   return o;
00139 }
```

7.7.1.9 operator>>()

```
bit_pipe operator>> (
                const bit_pipe & B,
                int n )
00172                                                    {
00173   bit_pipe ans = B;
00174   ans.shift_right(n);
00175   return ans;
00176 }
```

7.8 bitstream.h File Reference

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

Classes

- class bit_pipe

    A sequence of arbitrary number of bits.
- class obitstream

    handles writing bitstreams to binary files
- class ibitstream

    deals with reading bit streams from binary files, this is the reverse of obitstream

Functions

- unsigned int nu_bits (unsigned int n)

    returns number of bits in a positive integer n, e.g. 3 has 3 bits, 12 has 4 bits, and 0 has 0 bits.
- unsigned int mask_gen (int n)

    generates a binary mask with n consecutive ones in LSB
- bit_pipe elias_delta_encode (const unsigned int &n)

    returns the Elias delta representation of an integer in bit_pipe format
- void elias_delta_encode (const unsigned int &n, bit_pipe &B)

    performs Elias delta encode for an integer, and stores the results in the given reference to bit_pipe objects
- bit_pipe elias_delta_encode (const mpz_class &n)

    returns the Elias delta representation of an mpz_class in bit_pipe format
- void elias_delta_encode (const mpz_class &n, bit_pipe &B)

    performs Elias delta encoding on n, and stores the results in the given reference to bit_pipe objects

Variables

- const unsigned int BYTE_INT = sizeof(unsigned int)
- const unsigned int BIT_INT = 8 ∗ sizeof(unsigned int)

7.8.1 Function Documentation

7.8.1.1 elias_delta_encode() [1/4]

```
bit_pipe elias_delta_encode (
            const mpz_class & n )
```

returns the Elias delta representation of an mpz_class in bit_pipe format

```
00694                                               {
00695   if (n == 0){
00696     cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
00697   }
00698   // first, find number of bits in n
00699   int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
00700   int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00701
00702   bit_pipe N(n_bits); // binary representation
00703   N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00704   bit_pipe n_pipe(n); // binary representation of n
00705   n_pipe.shift_left(1); // remove the leading 1
00706   n_pipe.append_left(N);
00707   return n_pipe;
00708 }
```

7.8.1.2 elias_delta_encode() [2/4]

```
void elias_delta_encode (
            const mpz_class & n,
            bit_pipe & B )
```

performs Elias delta encoding on n, and stores the results in the given reference to bit_pipe objects

```
00710                                                                              {
00711   if (n == 0){
00712     cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
00713   }
00714   // first, find number of bits in n
00715   int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
00716   int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00717
00718   bit_pipe N(n_bits); // binary representation
00719   N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00720   B = bit_pipe(n); // binary representation of n
00721   B.shift_left(1); // remove the leading 1
00722   B.append_left(N);
00723 }
```

7.8.1.3 elias_delta_encode() [3/4]

```
bit_pipe elias_delta_encode (
            const unsigned int & n )
```

returns the Elias delta representation of an integer in bit_pipe format

```
00662                                                                              {
00663   if (n == 0){
00664     cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
00665   }
00666   // first, find number of bits in n
00667   int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
00668   int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00669
00670   bit_pipe N(n_bits); // binary representation
00671   N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00672   bit_pipe n_pipe(n); // binary representation of n
00673   n_pipe.shift_left(1); // remove the leading 1
00674   n_pipe.append_left(N);
00675   return n_pipe;
00676 }
```

7.8.1.4 elias_delta_encode() [4/4]

```
void elias_delta_encode (
            const unsigned int & n,
            bit_pipe & B )
```

performs Elias delta encode for an integer, and stores the results in the given reference to bit_pipe objects

```
00678                                                                              {
00679   if (n == 0){
00680     cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
00681   }
00682   // first, find number of bits in n
00683   int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
00684   int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2
    n \rfloor\f$
00685
00686   bit_pipe N(n_bits); // binary representation
00687   N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
00688   B = bit_pipe(n); // binary representation of n
00689   B.shift_left(1); // remove the leading 1
00690   B.append_left(N);
00691 }
```

7.8.1.5 mask_gen()

```
unsigned int mask_gen (
            int n )
```

generates a binary mask with n consecutive ones in LSB

Example: n = 1 -> 00000001, n = 7 -> 01111111

```
00649                                      {
00650    if (n < 1 or n > BIT_INT){
00651      cerr « " ERROR: mask_gen called for n outside the range [1,BIT_INT] " « endl;
00652      return 0;
00653    }
00654    unsigned int mask = 1;
00655    for (int i=1; i<n; i++){
00656      mask «= 1;
00657      mask += 1;
00658    }
00659    return mask;
00660 }
```

7.8.1.6 nu_bits()

```
unsigned int nu_bits (
            unsigned int n )
```

returns number of bits in a positive integer n, e.g. 3 has 3 bits, 12 has 4 bits, and 0 has 0 bits.

This is in fact nothing but $\lfloor \log_2 n \rfloor + 1$

```
00635                                         {
00636    int nu_bits = 0;
00637    unsigned int n_copy = n;
00638    while (n_copy > 0){
00639      nu_bits ++;
00640      n_copy »= 1;
00641    }
00642    return nu_bits;
00643 }
```

7.8.2 Variable Documentation

7.8.2.1 BIT_INT

```
const unsigned int BIT_INT = 8 * sizeof(unsigned int)
```

7.8.2.2 BYTE_INT

```
const unsigned int BYTE_INT = sizeof(unsigned int)
```

7.9 bitstream.h

```
00001 #ifndef __BITSTREAM__
00002 #define __BITSTREAM__
00003
00004 #include <iostream>
00005 #include <gmpxx.h>
00006 #include <vector>
00007
00008
00009
00010 using namespace std;
00011
00012 const unsigned int BYTE_INT = sizeof(unsigned int); // number of bytes in an unsigned int data type
00013 const unsigned int BIT_INT = 8 * sizeof(unsigned int); // the number of bits in an unsigned int data
     type
00014
00015
00016
00018
00023 class bit_pipe{
00024  private:
00025   vector<unsigned int> bits;
00026   int last_bits;
00027  public:
00028   bit_pipe(){bits.resize(0); last_bits = 0;}
00029
00031   bit_pipe(const unsigned int &n);
00032
00034   bit_pipe(const mpz_class& n);
00035
00037   void shift_right(int n);
00038
00040   void shift_left(int n);
00041
00043   friend ostream& operator << (ostream& o, const bit_pipe& B);
00044
00046   int size()const{return bits.size();}
00047
00049   int residue()const{return last_bits;}
00050
00052   const vector<unsigned int>& chunks() const{return bits;}
00053
00055   void append_left(const bit_pipe& B);
00056
00058   friend bit_pipe operator << (const bit_pipe& B, int n);
00059
00061   friend bit_pipe operator >> (const bit_pipe& B, int n);
00062
00064   unsigned int& operator [](int n);
00065
00067   const unsigned int& operator [](int n)const;
00068
00069   friend class obitstream;
00070   friend class ibitstream;
00071 };
00072
00073
00075
00080 class obitstream{
00081  private:
00082   bit_pipe buffer;
00083   FILE* f;
00085   void write();
00086   unsigned int chunks_written;
00087  public:
00089   obitstream(string file_name){
00090     f = fopen(file_name.c_str(), "wb+");
00091     chunks_written = 0;
00092   }
00093
00095   void write_bits(unsigned int n, unsigned int nu_bits);
00096
00098   obitstream& operator << (const unsigned int& n);
00099
00101   obitstream& operator << (const mpz_class& n);
00102
00104   void bin_inter_code(const vector<int>& a, int b);
00105
00107   void bin_inter_code(const vector<int>& a, int i, int j, int low, int high);
00109   unsigned int chunks(){
00110     return chunks_written;
00111   }
00112
```

```
00114   void close();
00115 };
00116
00118 class ibitstream{
00119  private:
00120   FILE* f;
00121   unsigned int buffer;
00122   unsigned int head_mask;
00123   unsigned int head_place;
00124
00125  public:
00126
00128   void read_chunk();
00129
00131   unsigned int read_bits(unsigned int k);
00132
00134   void read_bits(int k, bit_pipe& B);
00135
00137   void read_bits_append(int k, bit_pipe& B);
00138
00140   bool read_bit();
00141
00142   ibitstream(string file_name){
00143     f = fopen(file_name.c_str(), "rb+");
00144     buffer =0;
00145     head_mask = 0;
00146     head_place = 0;
00147   }
00148
00150   ibitstream& operator >> (unsigned int& n);
00151
00153   ibitstream& operator >> (mpz_class& n);
00154
00156   void bin_inter_decode(vector<int>& a, int b);
00157
00158
00160   void bin_inter_decode(vector<int>& a, int i, int j, int low, int high);
00161
00162
00163
00165   void close(){fclose(f);}
00166 };
00167
00168
00170 unsigned int nu_bits(unsigned int n);
00171
00173 unsigned int mask_gen(int n);
00174
00176 bit_pipe elias_delta_encode(const unsigned int &n);
00177
00179
00180 void elias_delta_encode(const unsigned int &n, bit_pipe& B);
00181
00183 bit_pipe elias_delta_encode(const mpz_class &n);
00184
00186 void elias_delta_encode(const mpz_class &n, bit_pipe& B);
00187
00188
00189 #endif
00190
00191
```

7.10   compression_helper.cpp File Reference

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

Functions

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

    This function computes the product of consecutive integers separated by a given iteration. This is the old version which uses standard recursion.

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

This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.

- void compute_product_void (int N, int k, int s)
- void compute_array_product (vector< mpz_class > &a)

  computes the product of elements in vector a by inline multiplication of adjacent elements recursively. The results will be in a[0].

- mpz_class compute_product (int N, int k, int s)
- mpz_class binomial (const int n, const int m)

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

- mpz_class prod_factorial_old (const vector< int > &a, int i, int j)

  computes the product of factorials in a vector given a range

- mpz_class prod_factorial (const vector< int > &a, int i, int j)
- int bit_string_write (FILE ∗f, const string &s)

  Write a string containing 0 and 1 to a binary file, treating the string as a bit sequence. Returns the number of bytes written to the output.

- string bit_string_read (FILE ∗f)

  Reads a bit sequence from a binary file, assuming the bit sequence was generated by the `bit_string_write` *function.*

7.10.1 Function Documentation

7.10.1.1 binomial()

```
mpz_class binomial (
            const int n,
            const int 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

```
00319 {
00320   if (n <= 0 or m > n or m <= 0)
00321     return 0;
00322   return compute_product(n, m, 1) / compute_product(m, m, 1);
00323 }
```

7.10.1.2 bit_string_read()

```
string bit_string_read (
            FILE * f )
```

Reads a bit sequence from a binary file, assuming the bit sequence was generated by the `bit_string_write` function.

Parameters

f	a file pointer

Returns

a string of zeros and ones.

```
00375                                        {
00376    int nu_bytes;
00377    int ssize;
00378    // read the number of bytes to read
00379    fread(&ssize, sizeof(ssize), 1, f);
00380    //cerr << " ssize " << ssize << endl;
00381    nu_bytes = ssize / 8;
00382    if (ssize % 8 != 0)
00383      nu_bytes ++;
00384
00385    int last_byte_size = ssize % 8;
00386    if (last_byte_size == 0)
00387      last_byte_size  = 8;
00388
00389    unsigned char c;
00390    bitset<8> B;
00391    string s;
00392    for (int i=0;i<nu_bytes;i++){
00393      fread(&c, sizeof(c), 1, f);
00394      B = c;
00395      //cout << B << endl;
00396      if (i < nu_bytes -1){
00397        s += B.to_string();
00398      }else{
00399        s += B.to_string().substr(8-last_byte_size, last_byte_size);
00400      }
00401    }
00402    return s;
00403 }
```

7.10.1.3 bit_string_write()

```
int bit_string_write (
            FILE * f,
            const string & s )
```

Write a string containing 0 and 1 to a binary file, treating the string as a bit sequence. Returns the number of bytes written to the output.

First, the size of the bit sequence is written to the output, then the input is split into 8 bit chunks, perhaps with some leftover, which are written to the output file as bytes.

Parameters

f	a file pointer
s	a string where each character is either 0 or 1

Returns

the number of bytes written to the output

```
00350                                        {
00351    // find out the number of bytes
00352    int ssize = s.size();
00353    int nu_bytes; // number of bytes wrote to the output
00354
00355    //if (ssize % 8 != 0) // an incomplete byte is required
00356    //nu_bytes++;
```

```
00357
00358    fwrite(&ssize, sizeof(ssize), 1, f); // first, write down how many bytes are coming.
00359    nu_bytes += sizeof(ssize);
00360
00361    stringstream ss;
00362    ss << s;
00363
00364    bitset<8> B;
00365    unsigned char c;
00366    while (ss >> B){
00367       c = B.to_ulong();
00368       fwrite(&c, sizeof(c), 1, f);
00369       nu_bytes += sizeof(c);
00370    }
00371    return nu_bytes;
00372 }
```

7.10.1.4 compute_array_product()

```
void compute_array_product (
              vector< mpz_class > & a )
```

computes the product of elements in vector a by inline multiplication of adjacent elements recursively. The results will be in a[0].

```
00237                                              {
00238    //logger::item_start("Compute Array Product");
00239    int step_size, to_mul;
00240    int k = a.size();
00241    for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
00242       for (int i=0; i<k; i+=step_size){
00243          if (i+to_mul < k)
00244             a[i] *= a[i+to_mul];
00245       }
00246    }
00247    //logger::item_stop("Compute Array Product");
00248 }
```

7.10.1.5 compute_product()

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

```
00251                                                   {
00252    if (k==1)
00253       return N;
00254    if (k == 0) // TO CHECK because there are no terms to compute product
00255       return 1;
00256
00257    if (k < 0){
00258       cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
         << endl;
00259       return 1;
00260    }
00261    if (N - (k-1) * s <= 0){ // the terms go negative
00262       //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00263       return 0;
00264    }
00265
00266    if (k == 2){
00267       helper_vars::mul_1 = N;
00268       helper_vars::mul_2 = N - s;
00269       return helper_vars::mul_1 * helper_vars::mul_2;
00270    }
00271
00272    helper_vars::mpz_vec.resize(k);
00273    for (int i=0;i<k;i++)
00274       helper_vars::mpz_vec[i] = N - i * s;
00275
00276    compute_array_product(helper_vars::mpz_vec);
00277
00278    // int step_size, to_mul;
00279
00280
```

```
00281   // for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
00282   //   for (int i=0; i<k; i+=step_size){
00283   //      if (i+to_mul < k)
00284   //        helper_vars::mpz_vec[i] *= helper_vars::mpz_vec[i+to_mul];
00285   //   }
00286   // }
00287   return helper_vars::mpz_vec[0];
00288 }
```

7.10.1.6 compute_product_old()

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

This function computes the product of consecutive integers separated by a given iteration. This is the old version which uses standard recursion.

Parameters

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

Returns

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

```
00009                                                {
00010   //cerr << " compute_product  N " << N << " k " << k << " s " << s << endl;
00011
00012   if (k==1)
00013     return N;
00014   if (k == 0) // TO CHECK because there are no terms to compute product
00015     return 1;
00016
00017   if (k < 0){
00018     cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
00019 << endl;
00019     return 1;
00020   }
00021   if (N - (k-1) * s <= 0){ // the terms go negative
00022     //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00023     return 0;
00024   }
00025
00026   if (k == 2)
00027     return mpz_class(N) * mpz_class(N-s);
00028   // we do this by dividing the terms into two parts
00029   int m = k / 2; // the middle point
00030   mpz_class left, right; // each of the half products
00031   left = compute_product(N, m, s);
00032   right = compute_product(N-m * s, k-m, s);
00033   //logger::item_start("cp_mul");
00034   mpz_class ans = left*right;
00035   //logger::item_stop("cp_mul");
00036   return ans;
00037 }
```

7.10.1.7 compute_product_stack()

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

This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.

Parameters

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

Returns

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

```
00039                                                                    {
00040
00041     if (k==1)
00042       return N;
00043     if (k == 0) // TO CHECK because there are no terms to compute product
00044       return 1;
00045
00046     if (k < 0){
00047       cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
      << endl;
00048       return 1;
00049     }
00050     if (N - (k-1) * s <= 0){ // the terms go negative
00051       //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00052       return 0;
00053     }
00054
00055     if (k == 2){
00056       helper_vars::mul_1 = N;
00057       helper_vars::mul_2 = N - s;
00058       return helper_vars::mul_1 * helper_vars::mul_2;
00059     }
00060
00061     logger::item_start("CP body");
00062
00063     int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
00064     int k_copy = k;
00065     while (k_copy > 0){
00066       k_bits ++;
00067       k_copy >>= 1;
00068     }
00069     k_bits += 2;
00070     vector<pair<int, int> > call_stack(2 * k_bits);
00071     //cout << " 2 * k_bits " << 2 * k_bits << endl;
00072     int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
00073     vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
00074     //vector<mpz_class> return_stack(2 * k_bits);
00075     helper_vars::return_stack.resize(2*k_bits);
00076     int return_pointer = 0;
00077
00078     call_stack[call_pointer] = pair<int, int> (N, k);
00079     status_stack[call_pointer] = 0;
00080     call_pointer ++;
00081
00082     int m;
00083     int N_now, k_now; // N and k for the current stack element
00084
00085     while (call_pointer > 0){
00086       N_now = call_stack[call_pointer-1].first;
00087       k_now = call_stack[call_pointer-1].second;
00088       //cout << "call_pointer = " << call_pointer << " N = " << N_now << " k = " << k_now << " stat = " <<
      status_stack[call_pointer-1] << endl;
00089       //cout << " the whole stack " << endl;
00090       //for (int i=0;i<call_pointer; i++){
00091       //   cout << call_stack[i].first << " , " << call_stack[i].second << " " << status_stack[i] << endl;
00092       //}
00093       if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
00094         // to collect two top elements in return stack and multiply them
00095         logger::item_start("CP arithmetic");
00096         helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] *
      helper_vars::return_stack[return_pointer-1];
00097         logger::item_stop("CP arithmetic");
00098         return_pointer--; // remove two items, add one item
00099         call_pointer --;
00100       }else{
00101         //cout << " else " << endl;
00102         if(k_now == 1){
00103           // to return the corresponding N
00104           helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
00105           call_pointer --; // pop this element
00106         }
```

```
00107          if (k_now == 2){
00108            helper_vars::mul_1 = N_now;
00109            helper_vars::mul_2 = N_now - s;
00110            logger::item_start("CP arithmetic");
00111            helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
00112            logger::item_stop("CP arithmetic");
00113            call_pointer --;
00114          }
00115          if (k_now > 2){
00116            m = k_now / 2;
00117            status_stack[call_pointer-1] = 1; // when return to this state, we know that we should
     aggregate
00118            call_stack[call_pointer] = pair<int, int>(N_now, m);
00119            status_stack[call_pointer] = 0; // just added
00120            call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
00121            status_stack[call_pointer+1]  = 0;
00122            call_pointer += 2;
00123          }
00124       }
00125    }
00126    // make sure there is exactly one element in return stack
00127    if (return_pointer != 1){
00128      cerr << " return pointer is not zero";
00129    }
00130    logger::item_stop("CP body");
00131    return helper_vars::return_stack[0]; // the top element remaining in the return stack
00132 }
```

7.10.1.8 compute_product_void()

```
void compute_product_void (
               int N,
               int k,
               int s )
00136                                            {
00137    //cerr << " void called N " << N << " k " << k << " s " << s << endl;
00138    if (k==1){
00139      helper_vars::return_stack.resize(1);
00140      helper_vars::return_stack[0] = N;//return N;
00141      return;
00142    }
00143    if (k == 0){ // TO CHECK because there are no terms to compute product
00144      helper_vars::return_stack.resize(1);
00145      helper_vars::return_stack[0] = 1;//return 1;
00146      return;
00147    }
00148
00149    if (k < 0){
00150      cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
     << endl;
00151      helper_vars::return_stack.resize(1);
00152      helper_vars::return_stack[0] = 1;//return 1;
00153      return;
00154    }
00155    if (N - (k-1) * s <= 0){ // the terms go negative
00156      //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00157      helper_vars::return_stack.resize(1);
00158      helper_vars::return_stack[0] = 0; //return 0;
00159      return;
00160    }
00161
00162    if (k == 2){
00163      helper_vars::mul_1 = N;
00164      helper_vars::mul_2 = N - s;
00165      helper_vars::return_stack.resize(1);
00166      helper_vars::return_stack[0] = helper_vars::mul_1 * helper_vars::mul_2;
00167      return;
00168    }
00169
00170    int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
00171    int k_copy = k;
00172    while (k_copy > 0){
00173      k_bits ++;
00174      k_copy >>= 1;
00175    }
00176    k_bits += 2;
00177    vector<pair<int, int> > call_stack(2 * k_bits);
00178    //cout << " 2 * k_bits " << 2 * k_bits << endl;
00179    int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
00180    vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
00181    //vector<mpz_class> return_stack(2 * k_bits);
```

```
00182   helper_vars::return_stack.resize(2*k_bits);
00183   int return_pointer = 0;
00184
00185   call_stack[call_pointer] = pair<int, int> (N, k);
00186   status_stack[call_pointer] = 0;
00187   call_pointer ++;
00188
00189   int m;
00190   int N_now, k_now; // N and k for the current stack element
00191
00192   while (call_pointer > 0){
00193     N_now = call_stack[call_pointer-1].first;
00194     k_now = call_stack[call_pointer-1].second;
00195     //cout « "call_pointer = " « call_pointer « " N = " « N_now « " k = " « k_now « " stat = " «
       status_stack[call_pointer-1] « endl;
00196     //cout « " the whole stack " « endl;
00197     //for (int i=0;i<call_pointer; i++){
00198     //   cout « call_stack[i].first « " , " « call_stack[i].second « " " « status_stack[i] « endl;
00199     //}
00200     if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
00201       // to collect two top elements in return stack and multiply them
00202       helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] *
       helper_vars::return_stack[return_pointer-1];
00203       return_pointer--; // remove two items, add one item
00204       call_pointer --;
00205     }else{
00206       //cout « " else " « endl;
00207       if(k_now == 1){
00208         // to return the corresponding N
00209         helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
00210         call_pointer --; // pop this element
00211       }
00212       if (k_now == 2){
00213         helper_vars::mul_1 = N_now;
00214         helper_vars::mul_2 = N_now - s;
00215         helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
00216         call_pointer --;
00217       }
00218       if (k_now > 2){
00219         m = k_now / 2;
00220         status_stack[call_pointer-1] = 1; // when return to this state, we know that we should
       aggregate
00221         call_stack[call_pointer] = pair<int, int>(N_now, m);
00222         status_stack[call_pointer] = 0; // just added
00223         call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
00224         status_stack[call_pointer+1]  = 0;
00225         call_pointer += 2;
00226       }
00227     }
00228   }
00229   // make sure there is exactly one element in return stack
00230   if (return_pointer != 1){
00231     cerr « " return pointer is not zero";
00232   }
00233   //return helper_vars::return_stack[0]; // the top element remaining in the return stack
00234 }
```

7.10.1.9   prod_factorial()

```
mpz_class prod_factorial (
          const vector< int > & a,
          int i,
          int j )
00338                                                                      {
00339   if (i==j){
00340     return compute_product(a[i], a[i], 1);
00341   }else{
00342     helper_vars::mpz_vec2.resize(j-i+1);
00343     for (int k = i; k<=j;k++)
00344       helper_vars::mpz_vec2[k- i] = compute_product(a[k], a[k],1);
00345     compute_array_product(helper_vars::mpz_vec2);
00346     return helper_vars::mpz_vec2[0];
00347   }
00348 }
```

7.10.1.10   prod_factorial_old()

```
mpz_class prod_factorial_old (
          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

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

```
00327 {
00328   if (i == j){
00329     return compute_product(a[i], a[i], 1);
00330   }else{
00331     int k = (i+j)/2;
00332     mpz_class x = prod_factorial(a, i, k);
00333     mpz_class y = prod_factorial(a, k+1, j);
00334     return x * y;
00335   }
00336 }
```

7.11 compression_helper.h File Reference

```
#include <iostream>
#include <gmpxx.h>
#include <vector>
#include <stdio.h>
#include <bitset>
#include <sstream>
#include "logger.h"
```

Namespaces

- namespace helper_vars

Functions

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

  This function computes the product of consecutive integers separated by a given iteration. This is the old version which uses standard recursion.

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

  This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.

- mpz_class compute_product (int N, int k, int s)
- void compute_product_void (int N, int k, int s)
- void compute_array_product (vector< mpz_class > &a)

  computes the product of elements in vector a by inline multiplication of adjacent elements recursively. The results will be in a[0].

- mpz_class binomial (const int n, const int m)

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

- mpz_class prod_factorial_old (const vector< int > &a, int i, int j)

    computes the product of factorials in a vector given a range

- mpz_class prod_factorial (const vector< int > &a, int i, int j)
- int bit_string_write (FILE ∗f, const string &s)

    Write a string containing 0 and 1 to a binary file, treating the string as a bit sequence. Returns the number of bytes written to the output.

- string bit_string_read (FILE ∗f)

    Reads a bit sequence from a binary file, assuming the bit sequence was generated by the `bit_string_write` *function.*

Variables

- mpz_class helper_vars::mul_1
- mpz_class helper_vars::mul_2

    helper variables in order to avoid initialization

- vector< mpz_class > helper_vars::return_stack
- vector< mpz_class > helper_vars::mpz_vec
- vector< mpz_class > helper_vars::mpz_vec2

7.11.1 Function Documentation

7.11.1.1 binomial()

```
mpz_class binomial (
            const int n,
            const int 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

```
00319 {
00320   if (n <= 0 or m > n or m <= 0)
00321     return 0;
00322   return compute_product(n, m, 1) / compute_product(m, m, 1);
00323 }
```

7.11.1.2 bit_string_read()

```
string bit_string_read (
            FILE ∗ f )
```

Reads a bit sequence from a binary file, assuming the bit sequence was generated by the `bit_string_write` function.

Parameters

f	a file pointer

Returns

a string of zeros and ones.

```
00375                                      {
00376    int nu_bytes;
00377    int ssize;
00378    // read the number of bytes to read
00379    fread(&ssize, sizeof(ssize), 1, f);
00380    //cerr « " ssize " « ssize « endl;
00381    nu_bytes = ssize / 8;
00382    if (ssize % 8 != 0)
00383      nu_bytes ++;
00384
00385    int last_byte_size = ssize % 8;
00386    if (last_byte_size == 0)
00387      last_byte_size  = 8;
00388
00389    unsigned char c;
00390    bitset<8> B;
00391    string s;
00392    for (int i=0;i<nu_bytes;i++){
00393      fread(&c, sizeof(c), 1, f);
00394      B = c;
00395      //cout « B « endl;
00396      if (i < nu_bytes -1){
00397        s += B.to_string();
00398      }else{
00399        s += B.to_string().substr(8-last_byte_size, last_byte_size);
00400      }
00401    }
00402    return s;
00403 }
```

7.11.1.3 bit_string_write()

```
int bit_string_write (
            FILE * f,
            const string & s )
```

Write a string containing 0 and 1 to a binary file, treating the string as a bit sequence. Returns the number of bytes written to the output.

First, the size of the bit sequence is written to the output, then the input is split into 8 bit chunks, perhaps with some leftover, which are written to the output file as bytes.

Parameters

f	a file pointer
s	a string where each character is either 0 or 1

Returns

the number of bytes written to the output

```
00350                                      {
00351    // find out the number of bytes
00352    int ssize = s.size();
00353    int nu_bytes; // number of bytes wrote to the output
00354
00355    //if (ssize % 8 != 0) // an incomplete byte is required
00356    //nu_bytes++;
```

```
00357
00358    fwrite(&ssize, sizeof(ssize), 1, f); // first, write down how many bytes are coming.
00359    nu_bytes += sizeof(ssize);
00360
00361    stringstream ss;
00362    ss << s;
00363
00364    bitset<8> B;
00365    unsigned char c;
00366    while (ss >> B){
00367      c = B.to_ulong();
00368      fwrite(&c, sizeof(c), 1, f);
00369      nu_bytes += sizeof(c);
00370    }
00371    return nu_bytes;
00372 }
```

7.11.1.4 compute_array_product()

```
void compute_array_product (
            vector< mpz_class > & a )
```

computes the product of elements in vector a by inline multiplication of adjacent elements recursively. The results will be in a[0].

```
00237                                          {
00238    //logger::item_start("Compute Array Product");
00239    int step_size, to_mul;
00240    int k = a.size();
00241    for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
00242      for (int i=0; i<k; i+=step_size){
00243        if (i+to_mul < k)
00244          a[i] *= a[i+to_mul];
00245      }
00246    }
00247    //logger::item_stop("Compute Array Product");
00248 }
```

7.11.1.5 compute_product()

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

```
00251                                                {
00252    if (k==1)
00253      return N;
00254    if (k == 0) // TO CHECK because there are no terms to compute product
00255      return 1;
00256
00257    if (k < 0){
00258      cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
      << endl;
00259      return 1;
00260    }
00261    if (N - (k-1) * s <= 0){ // the terms go negative
00262      //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00263      return 0;
00264    }
00265
00266    if (k == 2){
00267      helper_vars::mul_1 = N;
00268      helper_vars::mul_2 = N - s;
00269      return helper_vars::mul_1 * helper_vars::mul_2;
00270    }
00271
00272    helper_vars::mpz_vec.resize(k);
00273    for (int i=0;i<k;i++)
00274      helper_vars::mpz_vec[i] = N - i * s;
00275
00276    compute_array_product(helper_vars::mpz_vec);
00277
00278    // int step_size, to_mul;
00279
00280
```

```
00281    // for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
00282    //    for (int i=0; i<k; i+=step_size){
00283    //       if (i+to_mul < k)
00284    //          helper_vars::mpz_vec[i] *= helper_vars::mpz_vec[i+to_mul];
00285    //    }
00286    // }
00287    return helper_vars::mpz_vec[0];
00288 }
```

7.11.1.6   compute_product_old()

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

This function computes the product of consecutive integers separated by a given iteration. This is the old version which uses standard recursion.

Parameters

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

Returns

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

```
00009                                                              {
00010    //cerr << " compute_product  N " << N << " k " << k << " s " << s << endl;
00011
00012    if (k==1)
00013      return N;
00014    if (k == 0) // TO CHECK because there are no terms to compute product
00015      return 1;
00016
00017    if (k < 0){
00018      cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
00019 << endl;
00019      return 1;
00020    }
00021    if (N - (k-1) * s <= 0){ // the terms go negative
00022      //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00023      return 0;
00024    }
00025
00026    if (k == 2)
00027      return mpz_class(N) * mpz_class(N-s);
00028    // we do this by dividing the terms into two parts
00029    int m = k / 2; // the middle point
00030    mpz_class left, right; // each of the half products
00031    left = compute_product(N, m, s);
00032    right = compute_product(N-m * s, k-m, s);
00033    //logger::item_start("cp_mul");
00034    mpz_class ans = left*right;
00035    //logger::item_stop("cp_mul");
00036    return ans;
00037 }
```

7.11.1.7   compute_product_stack()

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

This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.

Parameters

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

Returns

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

```
00039                                                                      {
00040
00041    if (k==1)
00042      return N;
00043    if (k == 0) // TO CHECK because there are no terms to compute product
00044      return 1;
00045
00046    if (k < 0){
00047      cerr « " WARNING: compute_product called for k < 0, returning 1, N  " « N « " k " « k « " s " « s
       « endl;
00048      return 1;
00049    }
00050    if (N - (k-1) * s <= 0){ // the terms go negative
00051      //cerr « " WARNING: compute_product called for N - (k-1) * s <= 0 " « endl;
00052      return 0;
00053    }
00054
00055    if (k == 2){
00056      helper_vars::mul_1 = N;
00057      helper_vars::mul_2 = N - s;
00058      return helper_vars::mul_1 * helper_vars::mul_2;
00059    }
00060
00061    logger::item_start("CP body");
00062
00063    int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
00064    int k_copy = k;
00065    while (k_copy > 0){
00066      k_bits ++;
00067      k_copy »= 1;
00068    }
00069    k_bits += 2;
00070    vector<pair<int, int> > call_stack(2 * k_bits);
00071    //cout « " 2 * k_bits " « 2 * k_bits « endl;
00072    int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
00073    vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
00074    //vector<mpz_class> return_stack(2 * k_bits);
00075    helper_vars::return_stack.resize(2*k_bits);
00076    int return_pointer = 0;
00077
00078    call_stack[call_pointer] = pair<int, int> (N, k);
00079    status_stack[call_pointer] = 0;
00080    call_pointer ++;
00081
00082    int m;
00083    int N_now, k_now; // N and k for the current stack element
00084
00085    while (call_pointer > 0){
00086      N_now = call_stack[call_pointer-1].first;
00087      k_now = call_stack[call_pointer-1].second;
00088      //cout « "call_pointer = " « call_pointer « " N = " « N_now « " k = " « k_now « " stat = " «
       status_stack[call_pointer-1] « endl;
00089      //cout « " the whole stack " « endl;
00090      //for (int i=0;i<call_pointer; i++){
00091      //  cout « call_stack[i].first « " , " « call_stack[i].second « " " « status_stack[i] « endl;
00092      //}
00093      if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
00094        // to collect two top elements in return stack and multiply them
00095        logger::item_start("CP arithmetic");
00096        helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] *
       helper_vars::return_stack[return_pointer-1];
00097        logger::item_stop("CP arithmetic");
00098        return_pointer--; // remove two items, add one item
00099        call_pointer --;
00100      }else{
00101        //cout « " else " « endl;
00102        if(k_now == 1){
00103          // to return the corresponding N
00104          helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
00105          call_pointer --; // pop this element
00106        }
```

```
00107        if (k_now == 2){
00108            helper_vars::mul_1 = N_now;
00109            helper_vars::mul_2 = N_now - s;
00110            logger::item_start("CP arithmetic");
00111            helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
00112            logger::item_stop("CP arithmetic");
00113            call_pointer --;
00114        }
00115        if (k_now > 2){
00116            m = k_now / 2;
00117            status_stack[call_pointer-1] = 1; // when return to this state, we know that we should
     aggregate
00118            call_stack[call_pointer] = pair<int, int>(N_now, m);
00119            status_stack[call_pointer] = 0; // just added
00120            call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
00121            status_stack[call_pointer+1]  = 0;
00122            call_pointer += 2;
00123        }
00124    }
00125    }
00126    // make sure there is exactly one element in return stack
00127    if (return_pointer != 1){
00128      cerr << " return pointer is not zero";
00129    }
00130    logger::item_stop("CP body");
00131    return helper_vars::return_stack[0]; // the top element remaining in the return stack
00132 }
```

7.11.1.8 compute_product_void()

```
void compute_product_void (
            int N,
            int k,
            int s )
00136                                              {
00137    //cerr << " void called N " << N << " k " << k << " s " << s << endl;
00138    if (k==1){
00139      helper_vars::return_stack.resize(1);
00140      helper_vars::return_stack[0] = N;//return N;
00141      return;
00142    }
00143    if (k == 0){ // TO CHECK because there are no terms to compute product
00144      helper_vars::return_stack.resize(1);
00145      helper_vars::return_stack[0] = 1;//return 1;
00146      return;
00147    }
00148
00149    if (k < 0){
00150      cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
     << endl;
00151      helper_vars::return_stack.resize(1);
00152      helper_vars::return_stack[0] = 1;//return 1;
00153      return;
00154    }
00155    if (N - (k-1) * s <= 0){ // the terms go negative
00156      //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
00157      helper_vars::return_stack.resize(1);
00158      helper_vars::return_stack[0] = 0; //return 0;
00159      return;
00160    }
00161
00162    if (k == 2){
00163      helper_vars::mul_1 = N;
00164      helper_vars::mul_2 = N - s;
00165      helper_vars::return_stack.resize(1);
00166      helper_vars::return_stack[0] = helper_vars::mul_1 * helper_vars::mul_2;
00167      return;
00168    }
00169
00170    int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
00171    int k_copy = k;
00172    while (k_copy > 0){
00173      k_bits ++;
00174      k_copy >>= 1;
00175    }
00176    k_bits += 2;
00177    vector<pair<int, int> > call_stack(2 * k_bits);
00178    //cout << " 2 * k_bits " << 2 * k_bits << endl;
00179    int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
00180    vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
00181    //vector<mpz_class> return_stack(2 * k_bits);
```

```
00182    helper_vars::return_stack.resize(2*k_bits);
00183    int return_pointer = 0;
00184
00185    call_stack[call_pointer] = pair<int, int> (N, k);
00186    status_stack[call_pointer] = 0;
00187    call_pointer ++;
00188
00189    int m;
00190    int N_now, k_now; // N and k for the current stack element
00191
00192    while (call_pointer > 0){
00193      N_now = call_stack[call_pointer-1].first;
00194      k_now = call_stack[call_pointer-1].second;
00195      //cout << "call_pointer = " << call_pointer << " N = " << N_now << " k = " << k_now << " stat = " <<
         status_stack[call_pointer-1] << endl;
00196      //cout << " the whole stack " << endl;
00197      //for (int i=0;i<call_pointer; i++){
00198      //  cout << call_stack[i].first << " , " << call_stack[i].second << " " << status_stack[i] << endl;
00199      //}
00200      if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
00201        // to collect two top elements in return stack and multiply them
00202        helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] *
         helper_vars::return_stack[return_pointer-1];
00203        return_pointer--; // remove two items, add one item
00204        call_pointer --;
00205      }else{
00206        //cout << " else " << endl;
00207        if(k_now == 1){
00208          // to return the corresponding N
00209          helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
00210          call_pointer --; // pop this element
00211        }
00212        if (k_now == 2){
00213          helper_vars::mul_1 = N_now;
00214          helper_vars::mul_2 = N_now - s;
00215          helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
00216          call_pointer --;
00217        }
00218        if (k_now > 2){
00219          m = k_now / 2;
00220          status_stack[call_pointer-1] = 1; // when return to this state, we know that we should
     aggregate
00221          call_stack[call_pointer] = pair<int, int>(N_now, m);
00222          status_stack[call_pointer] = 0; // just added
00223          call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
00224          status_stack[call_pointer+1]  = 0;
00225          call_pointer += 2;
00226        }
00227      }
00228    }
00229    // make sure there is exactly one element in return stack
00230    if (return_pointer != 1){
00231      cerr << " return pointer is not zero";
00232    }
00233    //return helper_vars::return_stack[0]; // the top element remaining in the return stack
00234 }
```

7.11.1.9 prod_factorial()

```
mpz_class prod_factorial (
            const vector< int > & a,
            int i,
            int j )
00338                                                                        {
00339    if (i==j){
00340      return compute_product(a[i], a[i], 1);
00341    }else{
00342      helper_vars::mpz_vec2.resize(j-i+1);
00343      for (int k = i; k<=j;k++)
00344        helper_vars::mpz_vec2[k- i] = compute_product(a[k], a[k],1);
00345      compute_array_product(helper_vars::mpz_vec2);
00346      return helper_vars::mpz_vec2[0];
00347    }
00348 }
```

7.11.1.10 prod_factorial_old()

```
mpz_class prod_factorial_old (
            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

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

```
00327 {
00328   if (i == j){
00329     return compute_product(a[i], a[i], 1);
00330   }else{
00331     int k = (i+j)/2;
00332     mpz_class x = prod_factorial(a, i, k);
00333     mpz_class y = prod_factorial(a, k+1, j);
00334     return x * y;
00335   }
00336 }
```

7.12   compression_helper.h

Go to the documentation of this file.
```
00001 #ifndef __COMPRESSION_HELPER__
00002 #define __COMPRESSION_HELPER__
00003
00004 #include <iostream>
00005 #include <gmpxx.h>
00006 #include <vector>
00007 #include <stdio.h>
00008 #include <bitset>
00009 #include <sstream>
00010 #include "logger.h"
00011
00012 using namespace std;
00013
00014 namespace helper_vars{
00015   extern mpz_class mul_1, mul_2;
00016   extern vector<mpz_class> return_stack;
00017   extern vector<mpz_class> mpz_vec;
00018   extern vector<mpz_class> mpz_vec2;
00019 };
00020
00021
00023
00029 //mpz_class compute_product(mpz_class N, mpz_class k, int s);
00030 mpz_class compute_product_old(int N, int k, int s);
00031
00033
00039 mpz_class compute_product_stack(int N, int k, int s);
00040 mpz_class compute_product(int N, int k, int s);
00041 void compute_product_void(int N, int k, int s);
00042
00044 void compute_array_product(vector<mpz_class>& a);
00045
00047
00052 mpz_class binomial(const int n, const int m);
00053
00054
00056
00061 mpz_class prod_factorial_old(const vector<int>& a, int i, int j);
00062
00063 mpz_class prod_factorial(const vector<int>& a, int i, int j);
00064
00066
00072 int bit_string_write(FILE* f, const string& s);
00073
00075
00079 string bit_string_read(FILE* f);
00080
00081
00082 #endif
```

7.13 fenwick.cpp File Reference

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

7.14 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.

7.15 fenwick.h

Go to the documentation of this file.
```
00001
00002 #ifndef __FENWICK__
00003 #define __FENWICK__
00004
00005 #include <vector>
00006
00007
00008 using namespace std;
00009
00011
00014 class fenwick_tree{
00016
00019   vector<int> sums;
00020  public:
00022    fenwick_tree()
00023      {
00024        sums.resize(0);
00025      }
00026
00028    fenwick_tree(vector<int>);
00029
00031    int size()
00032    {
00033      return sums.size() - 1;
00034    }
00035
00040    void add(int k, int val);
00041
00046    int sum(int k);
00047 };
00048
00050 /*
00051   Similar to the Fenwick tree class, but returns sum of values from a given index until the end of the
     array.
00052   This is implemented based on the previous fenwick_tree class.
00053 */
00054 class reverse_fenwick_tree{
00055    fenwick_tree FT;
00056 public:
00058    reverse_fenwick_tree(){}
00059
00061    reverse_fenwick_tree(vector<int>);
00062
00067    void add(int k, int val);
00068
00070    int size(){
```

```
00071      return FT.size();
00072   }
00073
00078   int sum(int k);
00079
00080
00081 };
00082
00083
00084
00085
00086 #endif
```

7.16   gcomp.cpp File Reference

To compress / decompress simple marked graphs.

```
#include <iostream>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>
#include "marked_graph_compression.h"
```

Functions

- int main (int argc, char ∗∗argv)

7.16.1   Detailed Description

To compress / decompress simple marked graphs.

This code has two functionalities: 1) gets a simple marked graph and compresses it, 2) gets a simple marked graph in its compressed form and decompresses it.

In order to compress a graph, the hyperparameters h and delta should be given using -h and -d, respectively. The input graph should be given using -i option, followed by the name of the file containing the graph. Also, the compressed graph will be stored in the file specified by -o option. A graph must be specified using its edge list in the following format: first, the number of vertices comes, then the mark of vertices in order, then each line contains the information on an edge, which is of the form i j x y, meaning there is an edge between vertices i and j, with mark x and y towards i and j, respectively.

In order to decompress, the compressed file should be given after -i, the file to store the decompressed graph should be given using -o, and an argument -u for uncompress should be given.

7.16.2 Function Documentation

7.16.2.1 main()

```
int main (
            int argc,
            char ** argv )
00022                                          {
00023    int h, delta;
00024    string infile, outfile;
00025    bool uncompress = false; // becomes true if -u option is given (to decompress)
00026    bool quiet = true; // becomes false if -v option is given (verbose)
00027    bool stat = false; // if true, statistics on the properties of the compressed graph, e.g. number of
        star vertices / edges or the number of partition graphs will be given
00028    char opt;
00029
00030    string report_file, stat_file;
00031    ofstream report_stream, stat_stream;
00032
00033    while ((opt = getopt(argc, argv, "h:d:i:o:uvsV:S:")) != EOF){
00034      switch(opt){
00035      case 'h':
00036        h = atoi(optarg);
00037        break;
00038      case 'd':
00039        delta = atoi(optarg);
00040        break;
00041      case 'i':
00042        infile = string(optarg);
00043        break;
00044      case 'o':
00045        outfile = string(optarg);
00046        break;
00047      case 'u':
00048        uncompress = true; // in the decompression phase
00049        break;
00050      case 'v':
00051        quiet = false;
00052        break;
00053      case 'V':
00054        report_file = string(optarg);
00055        if (report_file != ""){
00056          report_stream.open(report_file);
00057          logger::report_stream = &report_stream;
00058        }
00059        break;
00060      case 's':
00061        stat = true;
00062        break;
00063      case 'S':
00064        stat_file = string(optarg);
00065        if (stat_file != ""){
00066          stat_stream.open(stat_file);
00067          logger::stat_stream = &stat_stream;
00068        }
00069        break;
00070      case '?':
00071        cerr « "Error: option -" « char(optopt) « " requires an argument" « endl;
00072        return 1;
00073      }
00074    }
00075    if (uncompress == false and h <= 0){
00076      cerr « "Error: parameter h must be a positive integer. Instead, the value " « h « " was given." «
        endl;
00077      return 1;
00078    }
00079    if (uncompress == false and delta <= 0){
00080      cerr « "Error: parameter d (delta) must be a positive integer. Instead, the value " « delta « "
        was given." « endl;
00081      return 1;
00082    }
00083
00084    ifstream inp(infile.c_str());
00085    ofstream oup(outfile.c_str());
00086
00087    if (!inp.good()){
00088      cerr « "Error: invalid input file <" « infile « "> given " « endl;
00089      return 1;
00090    }
00091
00092    if (!oup.good()){
00093      cerr « "Error: invalid output file <" « outfile « "> given " « endl;
00094      return 1;
```

```
00095   }
00096
00097
00098   if (quiet == true){
00099     // do not log
00100     logger::verbose = false; // no run time log
00101     logger::report = false; // no final report
00102   }
00103
00104   if (stat == true){
00105     logger::stat = true;
00106   }
00107
00108   //cout « " h = " « h « " delta = " « delta « " infile = " « infile « " outfile = " « outfile « endl;
00109
00110   logger::start();
00111   if (uncompress == false){
00112     // goal is compression
00113     logger::current_depth++;
00114     logger::add_entry("Read Graph", "");
00115     marked_graph_encoder E(h, delta);
00116     marked_graph G; // the input graph to be compressed
00117     inp » G;
00118     logger::current_depth--;
00119     logger::current_depth++;
00120     logger::add_entry("Encode", "");
00121     marked_graph_compressed C = E.encode(G);
00122     logger::current_depth--;
00123     //FILE* f;
00124     //f = fopen(outfile.c_str(), "wb+");
00125     logger::current_depth++;
00126     logger::add_entry("Write to binary", "");
00127     //C.binary_write(f);
00128     C.binary_write(outfile);
00129     //fclose(f);
00130     logger::current_depth--;
00131   }else{
00132     // goal is to decompess
00133     //FILE* f;
00134     //f = fopen(infile.c_str(), "rb+");
00135     marked_graph_compressed C;
00136     logger::current_depth++;
00137     logger::add_entry("Read from binary", "");
00138     //C.binary_read(f);
00139     C.binary_read(infile);
00140     //fclose(f);
00141     logger::current_depth--;
00142
00143     logger::current_depth++;
00144     logger::add_entry("Decode", "");
00145     marked_graph_decoder D;
00146     marked_graph G = D.decode(C);
00147     logger::current_depth--;
00148
00149     logger::current_depth++;
00150     logger::add_entry("Write decoded graph to output file","");
00151     oup « G;
00152     logger::current_depth--;
00153   }
00154   logger::stop();
00155   return 0;
00156 }
```

7.17   **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

7.17.1 Function Documentation

7.17.1.1 pair_compare()

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

used for sorting messages
```
00515                                                                                      {
00516    return a.first < b.first;
00517 }
```

7.18 graph_message.h File Reference

```
#include <vector>
#include <map>
#include <unordered_map>
#include <boost/functional/hash/hash.hpp>
#include "marked_graph.h"
#include "logger.h"
```

Classes

- struct vint_hash
- 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

7.18.1 Function Documentation

7.18.1.1 pair_compare()

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

used for sorting messages
```
00515                                                                                      {
00516    return a.first < b.first;
00517 }
```

7.19 graph_message.h

Go to the documentation of this file.
```
00001 #ifndef __GRPAH_MESSAGE__
00002 #define __GRPAH_MESSAGE__
00003
00004 #include <vector>
00005 #include <map>
00006 #include <unordered_map>
00007 #include <boost/functional/hash/hash.hpp>
00008 #include "marked_graph.h"
00009 #include "logger.h"
00010
00011 using namespace std;
00012
00013 struct vint_hash{
00014   size_t operator()(vector<int> const& v) const;
00015 };
00016
00018
00032 class graph_message{
00033   int h;
00034   int Delta;
00035
00037   void update_messages(const marked_graph&);
00038
00040   //void update_message_dictionary(const marked_graph&);
00041
00043   inline void send_message(const vector<int>& m, int v, int i);
00044
00045 public:
00046   //const marked_graph & G; //!< reference to the marked graph for which we do message passing
00047   //vector<vector<vector<int > > > messages; //!< messages[v][i][t] is the integer version of 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
00048   vector<vector<int > > messages;
00049
00050   //vector<vector<vector<int > > > inward_messages; //!< inward_messages[v][i][t] is the integer
     version of the  message at time t from the ith neighbor of v towards v (in the order given by adj_list
     in graph G).
00051
00052   //vector<unordered_map<vector<int>, int, vint_hash> > message_dict; //!< message_dict[t] for \f$0
     \leq t \leq h-1\f$ is the message dictionary at depth t, which maps each message to its corresponding
     index in the dictionary
00053
00054   unordered_map<vector<int>, int, vint_hash> message_dict;
00055
00056   //vector<map<vector<int>, int> > message_dict; //!< message_dict[t] for \f$0 \leq t \leq h-1\f$ is
     the message dictionary at depth t, which maps each message to its corresponding index in the
     dictionary
00057
00058   //vector<vector<vector<int> > > message_list; //!< message_list[t] is the list of messages present
     in the graph at depth t, stored in an order consistent with message_dict[t], i.e. for a message m, if
     messsage_dict[t][m] = i, then message_list[t][i] = m. This is constructed in such a way that
     message_list[t][message_dict[t][x]] = x. message_list[h-1] is sorted in reverse order so that all *
     messages (those messages starting with -1) go to the end of the list. Star type messages (which
     roughly speaking corresponds to places where there is a vertex in the h neighborhood has degree more
     than delta) are vectors of size 2, first coordinate being -1, and the second being the edge mark
     component (towards the 'me' vertex).
00059
00060   vector<int> message_mark;
00061
00062   vector<bool> is_star_message;
00063
00064   //vector<int> message_mark; //!< for an integer message such as m at depth h-1, message_mark[m]
     denotes the mark component of the message that corresponds to m. The message corresponds to m is
     basically message_list[h-1][m] which is of type vector<int> and the last component in this array is
     the mark component.
00065
00066   //vector<bool> is_star_message; //!< for a message m (integer type) at depth h-1, is_star_message[m]
     is true if the corresponding message starts with -1, ans is false otherwise.
00067
00069   graph_message(const marked_graph& graph, int depth, int max_degree){
00070     h = depth;
00071     Delta = max_degree;
00072     update_messages(graph); // do message passing
00073   }
00074
00076   graph_message() {}
00077 };
00078
00079
00080
00081
00082
```

```
00083
00084
00086 bool pair_compare(const pair<vector<int> , int>& , const pair<vector<int>, int>& );
00087
00088
00089
00090
00091
00093
00117 class colored_graph{
00118 public:
00119   //const marked_graph & G; //!< the marked graph from which this is created
00120   int h;
00121   int Delta;
00122   graph_message M;
00123   int nu_vertices;
00124   vector<vector<pair<int, pair<int, int> > > > adj_list;
00125
00126   vector<vector<int> > index_in_neighbor;
00127
00128   //vector<map<int,int> > adj_location;  //!< adj_location[v] for \f$0 \leq v < n\f$, 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
00129
00130   vector<map<pair<int, int> , int> > deg;
00131
00132   //map<pair<int, int> , vector<int> > type_vertex_list; //!< type_vertex_list[(m,m')] for a pair of
      types m and m' is the list of vertices v in the graph with at least one edge adjacent to v with type m
      towards v and m' towards the other endpoint. type_vertex_list[(m, m')] is guaranteed to be an
      increasing list.
00133
00134   vector<vector<int> >  ver_type;
00135
00136   map<vector<int>, int > ver_type_dict;
00137
00138   vector<vector<int> > ver_type_list;
00139
00140
00141
00142   vector<int> ver_type_int;
00143
00144   vector<bool> is_star_vertex;
00145   vector<int> star_vertices;
00146
00148  colored_graph(const marked_graph& graph, int depth, int max_degree): M(graph, depth, max_degree),
    h(depth), Delta(max_degree)
00149   {
00150     init(graph); // initialize other variables
00151   }
00152
00154   colored_graph() {}
00155
00157   void init(const marked_graph& G);
00158 };
00159
00160
00161 #endif //__GRPAH_MESSAGE__
```

7.20 logger.cpp File Reference

```
#include "logger.h"
```

7.21 logger.h File Reference

```
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
#include <ctime>
#include <map>
```

Classes

- class log_entry
- class logger

Functions

- void __attribute__ ((constructor)) prog_start()
- void __attribute__ ((destructor)) prog_finish()

7.21.1 Function Documentation

7.21.1.1 __attribute__() [1/2]

```
void __attribute__ (
            (constructor)  )
```

7.21.1.2 __attribute__() [2/2]

```
void __attribute__ (
            (destructor)  )
```

7.22 logger.h

Go to the documentation of this file.
```
00001 #ifndef __LOGGER__
00002 #define __LOGGER__
00003
00004 #include <iostream>
00005 #include <string>
00006 #include <vector>
00007 #include <chrono>
00008 #include <ctime>
00009 #include <map>
00010
00011 using namespace std;
00012 using namespace std::chrono;
00013
00014 class log_entry{
00015  public:
00016    string name;
00017    string description;
00018    int depth;
00019    high_resolution_clock::time_point t;
00020    system_clock::time_point sys_t;
00021    log_entry(string name, string description, int depth);
00022 };
00023
00024 class logger{
00025  public:
00026    static bool verbose;
00027    static bool stat;
00028    static ostream* verbose_stream;
00029    static bool report;
00030    static ostream* report_stream;
00031    static ostream* stat_stream;
00032    static vector<log_entry> logs;
00033    static int current_depth;
00034    static void add_entry(string name, string description);
00035    static void start();
00036    static void stop();
00037    static void log();
00038    static void item_start(string name);
```

```
00039    static void item_stop(string name);
00040    static map<string, float> item_duration; //<! maps the title of each item to its duration
00041    static map<string, high_resolution_clock::time_point> item_last_start;
00042 };
00043
00044
00045 void __attribute__ ((constructor)) prog_start();
00046 void __attribute__ ((destructor))  prog_finish();
00047
00048 #endif
```

7.23 marked_graph.cpp File Reference

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

Functions

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

  inputs a marked_graph
- bool operator== (const marked_graph &G1, const marked_graph &G2)
- bool operator!= (const marked_graph &G1, const marked_graph &G2)
- bool edge_compare (const pair< int, pair< int, int > > &a, pair< int, pair< int, int > > &b)

  this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.
- ostream & operator<< (ostream &o, const marked_graph &G)

7.23.1 Function Documentation

7.23.1.1 edge_compare()

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

this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.

```
00093 {
00094    return a.first < b.first;
00095 }
```

7.23.1.2 operator"!=()

```
bool operator!= (
            const marked_graph & G1,
            const marked_graph & G2 )
00088 {
00089    return !(G1 == G2);
00090 }
```

7.23.1.3 operator$<<$()

```
ostream & operator<< (
            ostream & o,
            const marked_graph & G )
00099 {
00100   o << G.nu_vertices << endl;
00101   for (int v=0;v<G.nu_vertices;v++){
00102     o << G.ver_mark[v];
00103     if (v < G.nu_vertices-1)
00104       o << " ";
00105   }
00106   o << endl;
00107
00108   vector<pair<pair<int, int>, pair<int, int> > > edges;
00109   pair<pair<int, int> , pair<int, int> > edge; // the current edge to be added to the list
00110   for (int v=0;v<G.nu_vertices;v++){
00111     for (int i=0;i<G.adj_list[v].size();i++){
00112       if (G.adj_list[v][i].first > v){ // avoid duplicate in edge list, only add edges where the other
    endpoint has a greater index
00113         edge.first.first = v;
00114         edge.first.second = G.adj_list[v][i].first;
00115         edge.second = G.adj_list[v][i].second;
00116         edges.push_back(edge);
00117       }
00118     }
00119   }
00120   sort(edges.begin(), edges.end());
00121   o << edges.size() << endl;
00122   for(int i=0;i<edges.size();i++){
00123     o << edges[i].first.first << " " << edges[i].first.second << " " << edges[i].second.first << " " <<
    edges[i].second.second << endl;
00124   }
00125   return o;
00126   /*
00127   o << " number of vertices " << G.nu_vertices << endl;
00128   vector<pair<int, pair<int, int> > > l; // the adjacency list of a vertex
00129   for (int v=0; v<G.nu_vertices; v++){
00130     o << " vertex " << v << " mark " << G.ver_mark[v] << endl;
00131     //o << " adj list (connections to vertices with greater index): format (j, (x,y))" << endl;
00132     o << " adj list " << endl;
00133     l = G.adj_list[v];
00134     sort(l.begin(), l.end(), edge_compare);
00135     for (int i=0;i<l.size();i++){
00136       if (l[i].first > v)
00137         o << " (" << l[i].first << ", (" << l[i].second.first << ", " << l[i].second.second << ")) ";
00138     }
00139     o << endl << endl;
00140   }
00141   return o;
00142   */
00143 }
```

7.23.1.4 operator==()

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

two marked graphs are said to be the same if: 1) they have the same number of vertices, 2) vertex marks match and 3) each vertex has the same set of neighbors with matching marks.

```
00066 {
00067   if (G1.nu_vertices != G2.nu_vertices)
00068     return false;
00069   return G1.adj_list == G2.adj_list;
00070   int n = G1.nu_vertices; // number of vertices of the two graphs
00071   vector< pair< int, pair< int, int > > > l1, l2; // the adjacency list of a vertex in two graphs for
    comparison.
00072   for (int v=0;v<n;v++){
00073     if (G1.ver_mark[v] != G2.ver_mark[v]) // mark of each vertex should be the same
00074       return false;
00075     if (G1.adj_list[v].size() != G2.adj_list[v].size()) // each vertex must have the same degree in
    two graphs
00076       return false;
00077     l1 = G1.adj_list[v];
00078     l2 = G2.adj_list[v];
00079     sort(l1.begin(), l1.end(), edge_compare); // sort with respect to the other endpoint
```

```
00080     sort(l2.begin(), l2.end(), edge_compare);
00081     if (l1 != l2) // after sorting, the lists must match
00082       return false;
00083   }
00084   return true;
00085 }
```

7.23.1.5 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

```
00041 {
00042   logger::current_depth++;
00043   logger::add_entry("Read vertex marks and edges","");
00044   int nu_vertices;
00045   inp >> nu_vertices;
00046
00047   vector<int> ver_marks;
00048   ver_marks.resize(nu_vertices);
00049   for (int i=0;i<nu_vertices;i++)
00050     inp >> ver_marks[i];
00051
00052   int nu_edges;
00053   inp >> nu_edges;
00054   vector<pair< pair<int, int> , pair<int, int> > > edges;
00055   edges.resize(nu_edges);
00056   for (int i=0;i<nu_edges;i++)
00057     inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >>
      edges[i].second.second;
00058
00059   logger::add_entry("Constructing marked graph", "");
00060   G = marked_graph(nu_vertices, edges, ver_marks);
00061   logger::current_depth--;
00062   return inp;
00063 }
```

7.24 marked_graph.h File Reference

```
#include <iostream>
#include <vector>
#include <map>
#include <fstream>
#include "logger.h"
```

Classes

- class marked_graph

  simple marked graph

Functions

- istream & [operator>>](#) (istream &inp, [marked_graph](#) &G)

  inputs a [marked_graph](#)

- bool [edge_compare](#) (const pair< int, pair< int, int > > &a, pair< int, pair< int, int > > &b)

  this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.

7.24.1 Function Documentation

7.24.1.1 edge_compare()

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

this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.

```
00093 {
00094    return a.first < b.first;
00095 }
```

7.24.1.2 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

```
00041 {
00042    logger::current_depth++;
00043    logger::add_entry("Read vertex marks and edges","");
00044    int nu_vertices;
00045    inp » nu_vertices;
00046
00047    vector<int> ver_marks;
00048    ver_marks.resize(nu_vertices);
00049    for (int i=0;i<nu_vertices;i++)
00050      inp » ver_marks[i];
00051
00052    int nu_edges;
00053    inp » nu_edges;
00054    vector<pair< pair<int, int> , pair<int, int> > > edges;
00055    edges.resize(nu_edges);
00056    for (int i=0;i<nu_edges;i++)
00057      inp » edges[i].first.first » edges[i].first.second » edges[i].second.first »
   edges[i].second.second;
00058
00059    logger::add_entry("Constructing marked graph", "");
00060    G = marked_graph(nu_vertices, edges, ver_marks);
00061    logger::current_depth--;
00062    return inp;
00063 }
```

7.25 marked_graph.h

```
00001 #ifndef __MARKED_GRAPH__
00002 #define __MARKED_GRAPH__
00003
00004 #include<iostream>
00005 #include<vector>
00006 #include<map>
00007 #include<fstream>
00008 #include "logger.h"
00009 using namespace std;
00010
00012
00015 class marked_graph{
00016 public:
00017   int nu_vertices;
00018   vector<vector<pair<int, pair<int, int> > > > adj_list;
00019   //vector<map<int,int> > adj_location;  //!< adj_location[v] for \f$0 \leq v < n\f$, 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
00020
00021   vector<vector<int> > index_in_neighbor;
00022
00023   vector<int> ver_mark;
00024
00026   marked_graph()
00027     {
00028       nu_vertices = 0;
00029     }
00030
00032
00037   marked_graph(int n, vector<pair< pair<int, int> , pair<int, int> > > edges, vector<int>
      vertex_marks);
00038
00040
00043   friend bool operator== (const marked_graph& G1, const marked_graph& G2);
00044
00046   friend bool operator!= (const marked_graph& G1, const marked_graph& G2);
00047
00049   friend ostream& operator« (ostream& o, const marked_graph& G);
00050 };
00051
00053
00066 istream& operator»(istream& inp, marked_graph& G);
00067
00068
00070 bool edge_compare(const pair<int, pair<int, int> >& a, pair<int, pair<int, int> >& b);
00071
00072
00073 #endif // __MARKED_GRAPH__
```

7.26 marked_graph_compression.cpp File Reference

```
#include "marked_graph_compression.h"
```

Functions

- void vtype_list_read (ibitstream &inp)

7.26.1 Function Documentation

7.26.1.1 vtype_list_read()

```
void vtype_list_read (
            ibitstream & inp )
```

7.27 marked_graph_compression.h File Reference

```
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
#include "simple_graph.h"
#include "bipartite_graph.h"
#include "simple_graph_compression.h"
#include "bipartite_graph_compression.h"
#include "time_series_compression.h"
#include "logger.h"
#include "bitstream.h"
```

Classes

- class marked_graph_compressed
- class marked_graph_encoder
- class marked_graph_decoder

7.28 marked_graph_compression.h

Go to the documentation of this file.
```
00001 #ifndef __MARKED_GRAPH_COMPRESSION__
00002 #define __MARKED_GRAPH_COMPRESSION__
00003
00004
00005 #include <vector>
00006 #include "marked_graph.h"
00007 #include "graph_message.h"
00008 #include "simple_graph.h"
00009 #include "bipartite_graph.h"
00010 #include "simple_graph_compression.h"
00011 #include "bipartite_graph_compression.h"
00012 #include "time_series_compression.h"
00013 #include "logger.h"
00014 #include "bitstream.h"
00015
00016
00017 using namespace std;
00018
00019 class marked_graph_compressed
00020 {
00021  public:
00022   int n;
00023   int h;
00024   int delta;
00025   pair<vector<int>, mpz_class> star_vertices;
00026   map<pair<int, int> , vector<vector<int> > > star_edges;
00027
00028   vector<int> type_mark;
00029
00030   vector<vector<int> > ver_type_list;
00031
00032
00033   pair<vector<int>, mpz_class> ver_types;
00034
00035   map<pair<int, int>, mpz_class> part_bgraph;
00036
00037   map<int, pair<mpz_class, vector<int> > > part_graph;
00038
00039   void clear();
00040
00042   void binary_write(FILE* f);
00043
00046   void binary_write(string s);
00047
00049   void binary_read(FILE* f);
00050
```

```
00053    void binary_read(string s);
00054
00056    // \param i, j entries of ver_type_list to be compared, i.e. we compare ver_type_list[i] and
      ver_type_list[j]
00057    int vtype_max_match(int i, int j);
00058
00060    // \param oup output bitstream used to write the bit stream
00061    void vtype_list_write(obitstream& oup);
00062
00064    // \param inp input bitstream
00065    void vtype_list_read(ibitstream& inp);
00066
00071    void vtype_block_write(obitstream& oup, int i, int j);
00072
00079    void vtype_block_write(obitstream& oup, int i, int j, int ir, int jr);
00080
00081
00086    void vtype_block_read(ibitstream& inp, int i, int j);
00087
00094    void vtype_block_read(ibitstream& inp, int i, int j, int ir, int jr);
00095
00096 };
00097
00098 class marked_graph_encoder
00099 {
00100    int h;
00101    int delta;
00102    int n;
00103    colored_graph C;
00104    //int L; //!< the number of edge colors, excluding star types
00105    vector<bool> is_star_vertex;
00106    vector<int> index_in_star;
00107
00108    vector<int> star_vertices;
00109
00110    map<pair<int, int>, b_graph> part_bgraph;
00111
00112    vector<map<pair<int, int>, int> > part_index;
00113
00114    map<pair<int, int>, vector<int> > part_deg;
00115
00116    map<int, graph> part_graph;
00117
00118    marked_graph_compressed compressed;
00119
00121
00124    void encode_star_vertices();
00125
00127    void extract_edge_types(const marked_graph&);
00128
00130    void encode_star_edges();
00131
00133    void encode_vertex_types();
00134
00136    void find_part_index_deg();
00137
00139    void extract_partition_graphs();
00140
00142    void encode_partition_bgraphs();
00143
00145    void encode_partition_graphs();
00146  public:
00147
00148  marked_graph_encoder(int h_, int delta_): h(h_), delta(delta_) {}
00150    marked_graph_compressed encode(const marked_graph& G);
00151
00153    void encode(const marked_graph& G, FILE* f);
00154
00155 };
00156
00157
00158 class marked_graph_decoder
00159 {
00160    int h;
00161    int delta;
00162    int n;
00163    //int L; //!< the number of edge colors, excluding star types
00164    vector<int> is_star_vertex;
00165    vector<int> star_vertices;
00166
00167    //vector<vector<int> > ver_type; //!< the list of vertex types, where the type of a vertex is an
      array of size \f$1+L\times L\f$, using the same convention as in the `colored_graph` class.
00168
00169    map<pair<int, int>, b_graph> part_bgraph;
00170    map<int, graph> part_graph;
00171
00172    vector<pair<pair<int, int>, pair<int, int> > > edges;
```

```
00173
00174   vector<int> vertex_marks;
00175
00176   vector<map<pair<int, int>, int> > Deg;
00177
00178   map<pair<int, int>, vector<int> > part_deg;
00179
00180   map<pair<int, int>, vector<int> > origin_index;
00181
00182   void decode_star_vertices(const marked_graph_compressed&);
00183   void decode_star_edges(const marked_graph_compressed&);
00184   void decode_vertex_types(const marked_graph_compressed&);
00185   void find_part_deg_orig_index();
00186   void decode_partition_graphs(const marked_graph_compressed&);
00187   void decode_partition_bgraphs(const marked_graph_compressed&);
00188
00189  public:
00190
00192   marked_graph_decoder() {}
00193
00194
00195   marked_graph decode(const marked_graph_compressed&);
00196 };
00197
00198 #endif
```

7.29 random_graph.cpp File Reference

```
#include "random_graph.h"
```

Functions

- [marked_graph marked_ER](int n, double p, int ver_mark, int edge_mark)

  generates a marked Erdos Renyi graph

- [marked_graph poisson_graph](int n, double deg_mean, int ver_mark, int edge_mark)

  generates a random graph where roughly speaking, the degree of a vertex is Poisson

- [marked_graph near_regular_graph](int n, int half_deg, int ver_mark, int edge_mark)

  generates a random graph which is nearly regular, and the degree of each vertex is close to $2 * half\_deg$. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \text{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \text{edge\_mark} - 1\}$.

7.29.1 Function Documentation

7.29.1.1 marked_ER()

```
marked_graph marked_ER (
            int n,
            double p,
            int ver_mark,
            int edge_mark )
```

generates a marked Erdos Renyi graph

Parameters

n	the number of vertices
p	probability of an edge being present
ver_mark	the number of possible vertex marks
edge_mark	the number of possible edge marks

Returns

a random marked graph, where each edge is independently present with probability p, each vertex has a random integer mark in the range [0,ver_mark), and each edge has two random integers marks in the range [0,edge_mark)

```
00004 {
00005   unsigned seed = chrono::system_clock::now().time_since_epoch().count();
00006   default_random_engine generator(seed);
00007   uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
00008   uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
00009   uniform_real_distribution<double> unif_dist(0.0,1.0);
00010   double unif;
00011   int x, xp; // generated marks
00012
00013   vector<int> ver_marks(n); // vector of size n
00014   vector<pair< pair<int, int>, pair<int, int> > > edges; // the edge list of the graph
00015
00016   for (int v=0; v<n; v++){
00017     ver_marks[v] = ver_mark_dist(generator);
00018     for (int w=v+1; w<n;w++){
00019       unif = unif_dist(generator);
00020       if (unif < p){ // we put an edge between v and w
00021         x = edge_mark_dist(generator);
00022         xp = edge_mark_dist(generator);
00023         edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v,w), pair<int,
     int>(x,xp)));
00024       }
00025     }
00026   }
00027   cout << " marked_ER number of edges " << edges.size() << endl;
00028   return marked_graph(n, edges, ver_marks);
00029 }
```

7.29.1.2 near_regular_graph()

```
marked_graph near_regular_graph (
              int n,
              int half_deg,
              int ver_mark,
              int edge_mark )
```

generates a random graph which is nearly regular, and the degree of each vertex is close to $2 * $ half_deg. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \text{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \text{edge\_mark} - 1\}$.

Parameters

n	the number of vertices
half_deg	the number of edges each vertex tries to connect to, so the final average degree is 2∗half_deg
ver_mark	the number of possible vertex marks
edge_mark	the number of possible edge_marks

Returns

a random marked graph as described above.

```
00076                                                                                              {
00077   unsigned seed = chrono::system_clock::now().time_since_epoch().count();
00078   default_random_engine generator(seed);
00079
00080   uniform_int_distribution<int> neighbor_dist(0,n-1);
00081   uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
00082   uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
00083
00084   int w;
00085   pair<int, int> edge;
```

```
00086
00087    set<pair<int, int> > umarked_edges;
00088    vector< pair< pair< int, int >, pair< int, int > > > edges;
00089
00090    int x, xp; // edge marks
00091    vector<int> ver_marks(n);
00092
00093    for (int i=0;i<n;i++){
00094      ver_marks[i] = ver_mark_dist(generator);
00095      for (int j=0;j<half_deg;j++){
00096        w = neighbor_dist(generator);
00097        if (w!= i){
00098          edge = pair<int,int>(i,w);
00099          if (edge.first > edge.second)
00100            swap (edge.first, edge.second); // to make sure pairs are ordered
00101          umarked_edges.insert(edge);
00102        }
00103      }
00104    }
00105    for (set<pair<int, int»::iterator it = umarked_edges.begin(); it!=umarked_edges.end(); it++){
00106      x = edge_mark_dist(generator);
00107      xp = edge_mark_dist(generator);
00108      edges.push_back(pair<pair<int, int>, pair<int, int> >(*it, pair<int, int>(x, xp)));
00109    }
00110
00111
00112    return marked_graph(n, edges, ver_marks);
00113 }
```

7.29.1.3 poisson_graph()

```
marked_graph poisson_graph (
            int n,
            double deg_mean,
            int ver_mark,
            int edge_mark )
```

generates a random graph where roughly speaking, the degree of a vertex is Poisson

Parameters

n	the number of vertices
deg_mean	mean of Poisson
ver_mark	the number of possible vertex marks
edge_mark	the number of possible edge marks

Returns

A random graph, where each vertex chooses its degree according to Poisson(deg_mean), then picks neighbors uniformly at random, and connects to them (if the neighbors have not already connected to them, if some of the neighbors I pick are already connected to me, I just don't do anything). Vertex and edge marks are picked independently and uniformly.

```
00032                                                                                        {
00033    unsigned seed = chrono::system_clock::now().time_since_epoch().count();
00034    default_random_engine generator(seed);
00035    poisson_distribution<int> deg_dist(deg_mean);
00036    uniform_int_distribution<int> neighbor_dist(0,n-1); // distribution for the other endpoint
00037
00038    vector< pair< pair< int, int >, pair< int, int > > > edges;
00039    vector<int> ver_marks(n);
00040
00041    uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
00042    uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
00043    int x, xp; // edge mark values
00044
00045    pair< pair< int, int >, pair< int, int > > edge; // the current edge to be added
00046    vector<set<int> > neighbors(n); // the list of neighbors of vertices
00047    int deg; // the degree of a vertex
```

```
00048   int w; // the neighbor
00049   for (int v=0;v<n;v++){
00050     ver_marks[v] = ver_mark_dist(generator);
00051     deg = deg_dist(generator);
00052     for (int i=0; i<deg; i++){
00053       do{
00054         w = neighbor_dist(generator);
00055       }while(w == v or neighbors[v].find(w) != neighbors[v].end()); // not myself and not already
       connected
00056       // now, w is a possible neighbor
00057       // see if w has picked v as a neighbor
00058       if (neighbors[w].find(v) == neighbors[w].end()){
00059         // add w as a neighbors to v
00060         neighbors[v].insert(w);
00061         // marks for the edge
00062         x = edge_mark_dist(generator);
00063         xp = edge_mark_dist(generator);
00064         edge.first.first = v;
00065         edge.first.second = w;
00066         edge.second.first = x;
00067         edge.second.second = xp;
00068         edges.push_back(edge);
00069       }
00070     }
00071   }
00072   cerr « " edges size " « edges.size() « endl;
00073   return marked_graph(n, edges, ver_marks);
00074 }
```

7.30 random_graph.h File Reference

```
#include "marked_graph.h"
#include <random>
#include <chrono>
#include <vector>
#include <set>
```

Functions

- marked_graph marked_ER (int n, double p, int ver_mark, int edge_mark)

  generates a marked Erdos Renyi graph
- marked_graph poisson_graph (int n, double deg_mean, int ver_mark, int edge_mark)

  generates a random graph where roughly speaking, the degree of a vertex is Poisson
- marked_graph near_regular_graph (int n, int half_deg, int ver_mark, int edge_mark)

  generates a random graph which is nearly regular, and the degree of each vertex is close to $2 * half\_deg$. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \text{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \text{edge\_mark} - 1\}$.

7.30.1 Function Documentation

7.30.1.1 marked_ER()

```
marked_graph marked_ER (
            int n,
            double p,
            int ver_mark,
            int edge_mark )
```

generates a marked Erdos Renyi graph

Parameters

n	the number of vertices
p	probability of an edge being present
ver_mark	the number of possible vertex marks
edge_mark	the number of possible edge marks

Returns

a random marked graph, where each edge is independently present with probability p, each vertex has a random integer mark in the range [0,ver_mark), and each edge has two random integers marks in the range [0,edge_mark)

```
00004 {
00005   unsigned seed = chrono::system_clock::now().time_since_epoch().count();
00006   default_random_engine generator(seed);
00007   uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
00008   uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
00009   uniform_real_distribution<double> unif_dist(0.0,1.0);
00010   double unif;
00011   int x, xp; // generated marks
00012
00013   vector<int> ver_marks(n); // vector of size n
00014   vector<pair< pair<int, int>, pair<int, int> > > edges; // the edge list of the graph
00015
00016   for (int v=0; v<n; v++){
00017     ver_marks[v] = ver_mark_dist(generator);
00018     for (int w=v+1; w<n;w++){
00019       unif = unif_dist(generator);
00020       if (unif < p){ // we put an edge between v and w
00021         x = edge_mark_dist(generator);
00022         xp = edge_mark_dist(generator);
00023         edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v,w), pair<int,
    int>(x,xp)));
00024       }
00025     }
00026   }
00027   cout << " marked_ER number of edges " << edges.size() << endl;
00028   return marked_graph(n, edges, ver_marks);
00029 }
```

7.30.1.2 near_regular_graph()

```
marked_graph near_regular_graph (
            int n,
            int half_deg,
            int ver_mark,
            int edge_mark )
```

generates a random graph which is nearly regular, and the degree of each vertex is close to $2 * $ half_deg. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \texttt{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \texttt{edge\_mark} - 1\}$.

Parameters

n	the number of vertices
half_deg	the number of edges each vertex tries to connect to, so the final average degree is 2∗half_deg
ver_mark	the number of possible vertex marks
edge_mark	the number of possible edge_marks

Returns

a random marked graph as described above.

```
00076                                                                        {
00077    unsigned seed = chrono::system_clock::now().time_since_epoch().count();
00078    default_random_engine generator(seed);
00079
00080    uniform_int_distribution<int> neighbor_dist(0,n-1);
00081    uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
00082    uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
00083
00084    int w;
00085    pair<int, int> edge;
00086
00087    set<pair<int, int> > umarked_edges;
00088    vector< pair< pair< int, int >, pair< int, int > > > edges;
00089
00090    int x, xp; // edge marks
00091    vector<int> ver_marks(n);
00092
00093    for (int i=0;i<n;i++){
00094      ver_marks[i] = ver_mark_dist(generator);
00095      for (int j=0;j<half_deg;j++){
00096        w = neighbor_dist(generator);
00097        if (w!= i){
00098          edge = pair<int,int>(i,w);
00099          if (edge.first > edge.second)
00100            swap (edge.first, edge.second); // to make sure pairs are ordered
00101          umarked_edges.insert(edge);
00102        }
00103      }
00104    }
00105    for (set<pair<int, int»::iterator it = umarked_edges.begin(); it!=umarked_edges.end(); it++){
00106      x = edge_mark_dist(generator);
00107      xp = edge_mark_dist(generator);
00108      edges.push_back(pair<pair<int, int>, pair<int, int> >(*it, pair<int, int>(x, xp)));
00109    }
00110
00111
00112    return marked_graph(n, edges, ver_marks);
00113 }
```

7.30.1.3 poisson_graph()

```
marked_graph poisson_graph (
            int n,
            double deg_mean,
            int ver_mark,
            int edge_mark )
```

generates a random graph where roughly speaking, the degree of a vertex is Poisson

Parameters

n	the number of vertices
deg_mean	mean of Poisson
ver_mark	the number of possible vertex marks
edge_mark	the number of possible edge marks

Returns

A random graph, where each vertex chooses its degree according to Poisson(deg_mean), then picks neighbors uniformly at random, and connects to them (if the neighbors have not already connected to them, if some of the neighbors I pick are already connected to me, I just don't do anything). Vertex and edge marks are picked independently and uniformly.

```
00032                                                                        {
00033    unsigned seed = chrono::system_clock::now().time_since_epoch().count();
```

```
00034    default_random_engine generator(seed);
00035    poisson_distribution<int> deg_dist(deg_mean);
00036    uniform_int_distribution<int> neighbor_dist(0,n-1); // distribution for the other endpoint
00037
00038    vector< pair< pair< int, int >, pair< int, int > > > edges;
00039    vector<int> ver_marks(n);
00040
00041    uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
00042    uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
00043    int x, xp; // edge mark values
00044
00045    pair< pair< int, int >, pair< int, int > > edge; // the current edge to be added
00046    vector<set<int> > neighbors(n); // the list of neighbors of vertices
00047    int deg; // the degree of a vertex
00048    int w; // the neighbor
00049    for (int v=0;v<n;v++){
00050      ver_marks[v] = ver_mark_dist(generator);
00051      deg = deg_dist(generator);
00052      for (int i=0; i<deg; i++){
00053        do{
00054          w = neighbor_dist(generator);
00055        }while(w == v or neighbors[v].find(w) != neighbors[v].end()); // not myself and not already
    connected
00056        // now, w is a possible neighbor
00057        // see if w has picked v as a neighbor
00058        if (neighbors[w].find(v) == neighbors[w].end()){
00059          // add w as a neighbors to v
00060          neighbors[v].insert(w);
00061          // marks for the edge
00062          x = edge_mark_dist(generator);
00063          xp = edge_mark_dist(generator);
00064          edge.first.first = v;
00065          edge.first.second = w;
00066          edge.second.first = x;
00067          edge.second.second = xp;
00068          edges.push_back(edge);
00069        }
00070      }
00071    }
00072    cerr << " edges size " << edges.size() << endl;
00073    return marked_graph(n, edges, ver_marks);
00074 }
```

7.31 random_graph.h

Go to the documentation of this file.
```
00001 #ifndef __RANDOM_GRAPH__
00002 #define __RANDOM_GRAPH__
00003
00004 #include "marked_graph.h"
00005 #include <random>
00006 #include <chrono>
00007 #include <vector>
00008 #include <set>
00009 using namespace std;
00010
00012
00020 marked_graph marked_ER(int n, double p, int ver_mark, int edge_mark);
00021
00023
00031 marked_graph poisson_graph(int n, double deg_mean, int ver_mark, int edge_mark);
00032
00033
00035
00043 marked_graph near_regular_graph(int n, int half_deg, int ver_mark, int edge_mark);
00044
00045
00046 #endif
```

7.32 README.md File Reference

7.33 rnd_graph.cpp File Reference

```
#include <iostream>
#include <stdio.h>
```

```
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>
#include "random_graph.h"
```

Functions

- void print_usage ()
- int main (int argc, char ∗∗argv)

7.33.1 Function Documentation

7.33.1.1 main()

```
int main (
            int argc,
            char ** argv )
00043                                                  {
00044   int n, edge_mark, ver_mark;
00045   double p, deg;
00046   string type, outfile;
00047   char opt;
00048
00049   while ((opt = getopt(argc, argv, "t:n:p:d:e:v:o:h")) != EOF){
00050     switch(opt){
00051     case 't':
00052       type = string(optarg);
00053       break;
00054     case 'n':
00055       n = atoi(optarg);
00056       break;
00057     case 'p':
00058       p = atof(optarg);
00059       break;
00060     case 'd':
00061       deg = atof(optarg);
00062       break;
00063     case 'e':
00064       edge_mark = atoi(optarg);
00065       break;
00066     case 'v':
00067       ver_mark = atoi(optarg);
00068       break;
00069     case 'o':
00070       outfile = string(optarg);
00071       break;
00072     case 'h':
00073       print_usage();
00074       break;
00075     case '?':
00076       cerr << "Error: option -" << char(optopt) << " requires an argument" << endl;
00077       print_usage();
00078       return 1;
00079     }
00080   }
00081   ofstream oup(outfile.c_str());
00082   if (!oup.good()){
00083     cerr << "Error: invalid output file <" << outfile << "> given " << endl;
00084     return 1;
00085   }
00086
00087   marked_graph G;
00088   if (type == "er"){
00089     G = marked_ER(n, p, ver_mark, edge_mark);
00090     oup << G;
00091   }else if(type == "reg"){
00092     G = near_regular_graph(n, deg, ver_mark, edge_mark);
00093     oup << G;
00094   }else if(type == "poi"){
00095     G = poisson_graph(n, deg, ver_mark, edge_mark);
00096     oup << G;
00097   }else{
00098     cerr << " ERROR: unknown type " << type << ", type must be either <er> or <reg> or <poi> " << endl;
00099     return 1;
00100   }
00101   return 0;
00102 }
```

7.33.1.2 print_usage()

```
void print_usage ( )
00010                       {
00011    cout « " Usage: " « endl;
00012    cout « " rnd_graph -t [random graph type] -n [number of vertices] [[options with -p / -d]] -e
     [number of edge marks] -v [number of vertex marks] -o [output file]" « endl;
00013    cout « endl;
00014    cout « " OPTIONS " « endl;
00015    cout « " -n : number of vertices (positive integer) " « endl;
00016    cout « " -e : size of edge mark set " « endl;
00017    cout « " -v : size of vertex mark set " « endl;
00018    cout « " -o : output file to store graph " « endl;
00019    cout « " -t : type of random graph: " « endl;
00020    cout « "              \"er\" for Erdos Renyi " « endl;
00021    cout « "              \"reg\" for near regular graph " « endl;
00022    cout « "              \"poi\" for Poisson graph " « endl;
00023    cout « endl;
00024    cout « " TYPE SPECIFIC OPTIONS " « endl;
00025    cout « endl;
00026    cout « " Erdos Renyi Graph " « endl;
00027    cout « " ---------------- " « endl;
00028    cout « " -p : edge probability " « endl;
00029    cout « endl;
00030    cout « " Near Regular Graph " « endl;
00031    cout « " ------------------ " « endl;
00032    cout « " -d : half degree, generates a random graph which is nearly regular, and the degree of each
     vertex is close to 2 * half_deg. " « endl;
00033    cout « endl;
00034    cout « " Poisson Graph " « endl;
00035    cout « " ------------- " « endl;
00036    cout « " -d : mean degree, each vertex chooses Poisson neighbors with this mean " « endl;
00037    cout « endl;
00038
00039 }
```

7.34 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)

7.34.1 Function Documentation

7.34.1.1 operator"!=()

```
bool operator!= (
            const graph & G1,
            const graph & G2 )
00102 {
00103    return !(G1 == G2);
00104 }
```

7.34.1.2 operator<<()

```
ostream & operator<< (
              ostream & o,
              const graph & G )
00069 {
00070   int n = G.nu_vertices();
00071   vector<int> list;
00072   for (int i=0;i<n;i++){
00073     list = G.get_forward_list(i);
00074     o « i « " -> ";
00075     for (int j=0;j<list.size();j++){
00076       o « list[j];
00077       if (j < list.size()-1)
00078         o « ", ";
00079     }
00080     o « endl;
00081   }
00082   return o;
00083 }
```

7.34.1.3 operator==()

```
bool operator== (
              const graph & G1,
              const graph & G2 )
00086 {
00087   int n1 = G1.nu_vertices();
00088   int n2 = G2.nu_vertices();
00089   if (n1!= n2)
00090     return false;
00091   vector<int> list1, list2;
00092   for (int v=0; v<n1; v++){
00093     list1 = G1.get_forward_list(v);
00094     list2 = G2.get_forward_list(v);
00095     if (list1 != list2)
00096       return false;
00097   }
00098   return true;
00099 }
```

7.35 simple_graph.h File Reference

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

Classes

- class graph

    simple unmarked graph

7.36 simple_graph.h

Go to the documentation of this file.
```
00001 #ifndef __SIMPLE_GRAPH__
00002 #define __SIMPLE_GRAPH__
00003
00004 #include <iostream>
00005 #include <vector>
00006 using namespace std;
```

```
00007
00009 class graph{
00010   int n;
00011   vector<vector<int> > forward_adj_list;
00012   vector<int> degree_sequence;
00013 public:
00014
00016  graph(): n(0) {}
00017
00018
00020   graph(const vector<vector<int> > &list, const vector<int> &deg);
00021
00023   graph(const vector<vector<int> > &list);
00024
00026   vector<int> get_forward_list(int v) const;
00027
00029   int get_forward_degree(int v) const;
00030
00032   int get_degree(int v) const;
00033
00035   vector<int> get_degree_sequence() const;
00036
00038   int nu_vertices() const;
00039
00041   friend ostream& operator << (ostream& o, const graph& G);
00042
00044   friend bool operator == (const graph& G1, const graph& G2);
00045
00047   friend bool operator != (const graph& G1, const graph& G2);
00048
00049 };
00050
00051
00052 #endif
```

7.37   simple_graph_compression.cpp File Reference

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

7.38   simple_graph_compression.h File Reference

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

Classes

- class graph_encoder

    Encodes a simple unmarked graph.

- class graph_decoder

    Decodes a simple unmarked graph.

7.39 simple_graph_compression.h

Go to the documentation of this file.
```
00001 #ifndef __SIMPLE_GRAPH_COMPRESSION__
00002 #define __SIMPLE_GRAPH_COMPRESSION__
00003
00004 #include <vector>
00005 #include <math.h>
00006 #include "simple_graph.h"
00007 #include "compression_helper.h"
00008 #include "fenwick.h"
00009 #include "logger.h"
00010
00012
00034 class graph_encoder{
00035   //const graph& G; //!< the simple unmarked graph which is going to be encoded
00036   int n;
00037   vector<int> a;
00038   vector<int> beta;
00039   reverse_fenwick_tree U;
00040   vector<int> Stilde;
00041   int logn2;
00042
00043  public:
00045   graph_encoder(const vector<int>& a_);
00046
00047
00049   void init(const graph& G);
00050
00052
00057   pair<mpz_class, mpz_class> compute_N(const graph& G);
00058
00060
00064   pair<mpz_class, vector<int> > encode(const graph& G);
00065
00066 };
00067
00069
00101 class graph_decoder{
00102   vector<int> a;
00103   int n;
00104   int logn2;
00105   vector<vector<int> > x;
00106   vector<int> beta;
00107   reverse_fenwick_tree U;
00108   vector<int> tS;
00109 public:
00111   graph_decoder(vector<int> a_);
00112
00114   void init();
00115
00117   graph decode(mpz_class f, vector<int> tS_);
00118
00120
00125   pair<mpz_class, mpz_class> decode_node (int i, mpz_class tN);
00126
00127
00129
00136   pair<mpz_class, mpz_class> decode_interval(int i, int j, int I, mpz_class tN, int Sj);
00137
00138 };
00139
00140
00141 #endif
```

7.40 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"
#include "time_series_compression.h"
#include "marked_graph_compression.h"
#include "random_graph.h"
#include "logger.h"
```

Functions

- ostream & operator<< (ostream &o, const vector< int > &v)
- void b_graph_test ()
- void graph_test ()
- void time_series_compression_test ()
- void marked_graph_encoder_test ()
- void random_graph_test ()
- int main ()

7.40.1 Function Documentation

7.40.1.1 b_graph_test()

```
void b_graph_test ( )
00028                       {
00029   vector<int> a = {1,1,2}; // left degree sequence
00030   vector<int> b = {2,2}; // right degree sequence
00031
00032   b_graph G({{0},{1},{0,1}}); // defining the graph
00033
00034   b_graph_encoder E(a,b); // constructing the encoder object
00035   mpz_class f = E.encode(G);
00036
00037   b_graph_decoder D(a, b);
00038   b_graph Ghat = D.decode(f);
00039
00040   if (Ghat == G)
00041     cout « " successfully decoded the graph! " « endl;
00042 }
```

7.40.1.2 graph_test()

```
void graph_test ( )
00044                         {
00045   vector<int> a = {3,2,2,3};
00046   vector<vector<int> > list = {{1,2,3},{3},{3},{}};
00047
00048   graph G(list);
00049
00050   graph_encoder E(a);
00051   pair<mpz_class, vector<int> > f = E.encode(G);
00052
00053   graph_decoder D(a);
00054   graph Ghat = D.decode(f.first, f.second);
00055   Ghat = D.decode(f.first, f.second);
00056
00057   if (Ghat == G)
00058     cout « " successfully decoded the graph! " « endl;
00059 }
```

7.40.1.3 main()

```
int main ( )
00142            {
00143   cout « compute_product(100,100,1) « endl;
00144   cout « compute_product_old(100,100,1) « endl;
00145   //logger::start();
00146   //marked_graph G;
00147   //ifstream inp("star_graph.txt");
00148   //inp » G;
00149   //graph_message M(G, 10, 2);
00150   //M.update_messages();
00151   //graph_test();
00152   //time_series_compression_test();
00153   //marked_graph_encoder_test();
00154   //random_graph_test();
00155   //logger::stop();
00156   return 0;
00157   // vector<vector<int> > list = {{}, {}, {}};
00158
00159   // b_graph G({{0},{1},{0,1}});
00160   // // cout « G « endl;
00161   // // cout « G.nu_left_vertices() « endl;
00162   // // cout « G.nu_right_vertices() « endl;
00163   // // cout « G.get_left_degree_sequence() « endl;
00164   // // cout « G.get_right_degree_sequence() « endl;
00165   // vector<int> a = G.get_left_degree_sequence();
00166   // vector<int> b = G.get_right_degree_sequence();
00167
00168
00169   // b_graph_encoder  E(a,b);
00170   // mpz_class m = E.encode(G);
00171   // cout « "encoded: " « m « endl;
00172
00173   // b_graph_decoder D(a, b);
00174   // b_graph Ghat = D.decode(m);
00175   // cout « "decoded graph: " « endl « Ghat « endl;
00176
00177   // if (Ghat == G)
00178   //    cout « " equal " « endl;
00179   // return 0;
00180
00181 }
```

7.40.1.4 marked_graph_encoder_test()

```
void marked_graph_encoder_test ( )
00075 {
00076   logger::current_depth++;
00077
00078   logger::add_entry("Construct G", "");
00079   marked_graph G;
00080   //ifstream inp("test_graphs/ten_node.txt"); //("test_graphs/hexagon_diagonal_marked.txt");
00081   //ifstream inp("test_graphs/problem_4.txt");
00082   //inp » G;
00083   //G = marked_graph(5, {{{0,1}, {0,0}}, {{1,2}, {0,0}}, {{0,3},{0,0}}}, {0,0,0,0,0});
00084   //int h, delta;
00085   //cout « " h " « endl;
00086   //cin » h;
00087   //cout « " delta " « endl;
00088   //cin » delta;
00089   G = poisson_graph(100000,3, 10, 10);//
00090   //G = near_regular_graph(100000,3,1,1);
00091   //G = marked_ER(100,0.05,1 ,1);
00092   cout « " graph constructed " « endl;
00093   //cout « G « endl;
00094
00095   logger::add_entry("Encode","");
00096
00097   marked_graph_encoder E(3,20);
00098   //marked_graph_encoder E(1,20);
00099   marked_graph_compressed C = E.encode(G);
00100   FILE* f;
00101   logger::add_entry("write to binary file", "");
00102   f = fopen("test.dat", "wb+");
00103   C.binary_write(f);
00104   fclose(f);
00105   //cerr « " graph encoded " « endl;
00106
00107   FILE* g;
00108   g = fopen("test.dat", "rb+");
```

```
00109   marked_graph_compressed Chat;
00110   logger::add_entry("read from binary file", "");
00111   Chat.binary_read(g);
00112   fclose(g);
00113
00114   if (Chat.star_edges != C.star_edges)
00115     cerr « " star edges do not match " « endl;
00116
00117   logger::add_entry("Decode", "");
00118   marked_graph_decoder D;
00119   marked_graph Ghat = D.decode(Chat);
00120
00121   //cout « " Ghat " « endl;
00122   //cout « Ghat « endl;
00123
00124   logger::add_entry("compare", "");
00125   if (Ghat == G)
00126     cout « " successfully decoded the marked graph :D " « endl;
00127   else
00128     cout « " they do not match :(" « endl;
00129
00130   logger::current_depth--;
00131   //cout « " G " « endl;
00132   //cout « G « endl;
00133   //cout « " Ghat " « endl;
00134   //cout « Ghat « endl;
00135 }
```

7.40.1.5 operator$<<$()

```
ostream & operator<< (
            ostream & o,
            const vector< int > & v )
00019                                                   {
00020   for (int i=0;i<v.size();i++){
00021     o « v[i];
00022     if (i < v.size()-1)
00023       o « ", ";
00024   }
00025   return o;
00026 }
```

7.40.1.6 random_graph_test()

```
void random_graph_test ( )
00138 {
00139   marked_graph G = marked_ER(100,1,3,4);
00140   //cout « G « endl;
00141 }
```

7.40.1.7 time_series_compression_test()

```
void time_series_compression_test ( )
00062 {
00063   vector<int> a = {0,2,3,1,2,1,0,1,0,2,1,0,0,2,1,3,4,5,0};
00064   int n = a.size();
00065   time_series_encoder E(n);
00066   pair<vector<int>, mpz_class > ans = E.encode(a);
00067
00068   time_series_decoder D(n);
00069   vector<int> ahat = D.decode(ans);
00070   if (ahat == a)
00071     cout « " successfully decoded the original time series! " « endl;
00072 }
```

7.41 test_mp.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
#include "random_graph.h"
#include "logger.h"
```

Functions

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

7.41.1 Function Documentation

7.41.1.1 main()

```
int main ( )
00058                {
00059   logger::start();
00060   //mp_test();
00061   random_graph_test();
00062   logger::stop();
00063   //random_graph_test ();
00064   return 0;
00065 }
```

7.41.1.2 mp_test()

```
void mp_test ( )
00028                   {
00029   marked_graph G;
00030   ifstream inp("test_graphs/hexagon_diagonal_marked2.txt");
00031   inp >> G;
00032   //G = marked_ER(10000,0.0003,2 ,2);
00033   //G = poisson_graph(100000,3, 10, 10);
00034   //cerr << " graph generated " << endl;
00035   //cerr << " graph generated " << endl;
00036   //cout << " G " << endl << G << endl;
00037   colored_graph C(G, 3, 2);
00038
00039   /*
00040   int n = C.nu_vertices;
00041   for (int v=0; v<n; v++){
00042     cout << v << " : ";
00043     for (int i=0;i<C.adj_list[v].size();i++){
00044       cout << C.adj_list[v][i].first << " ( " << C.adj_list[v][i].second.first << " , " <<
     C.adj_list[v][i].second.second << " ) ";
00045     }
00046     cout << endl;
00047   }
00048
00049   cout << " message marks " << endl;
00050   for (int m=0; m<C.M.message_mark.size(); m++){
00051     cout << m << " mark = " << C.M.message_mark[m] << " star " << C.M.is_star_message[m] << endl;
00052   }
00053   */
00054
00055 }
```

7.41.1.3 operator$<<$()

```
ostream & operator<< (
            ostream & o,
            const vector< int > & v )                           {
00012
00013   for (int i=0;i<v.size();i++){
00014     o « v[i];
00015     if (i < v.size()-1)
00016       o « ", ";
00017   }
00018   return o;
00019 }
```

7.41.1.4 random_graph_test()

```
void random_graph_test ( )
00022                                 {
00023   //marked_graph G = poisson_graph(10, 2, 2, 2);
00024   marked_graph G = near_regular_graph(10,2,1,1);
00025   cout « G;
00026 }
```

7.42 time_series_compression.cpp File Reference

```
#include "time_series_compression.h"
```

7.43 time_series_compression.h File Reference

```
#include <vector>
#include "bipartite_graph.h"
#include "bipartite_graph_compression.h"
```

Classes

- class time_series_encoder

    encodes a time series which is basically an array of arbitrary nonnegative integers

- class time_series_decoder

    decodes a time series which is basically an array of arbitrary nonnegative integers

7.44 time_series_compression.h

Go to the documentation of this file.
```
00001 #ifndef __TIME_SERIES_COPMRESSION__
00002 #define __TIME_SERIES_COPMRESSION__
00003
00004 #include <vector>
00005 #include "bipartite_graph.h"
00006 #include "bipartite_graph_compression.h"
00007 using namespace std;
00008
00010
00024 class time_series_encoder{
00025    int n;
00026    int alph_size;
00027    vector<int> freq;
00028    b_graph G;
00029
00030
00032    void init_alph_size(const vector<int>& x);
00033
00035    void init_freq(const vector<int>& x);
00036
00038    void init_G(const vector<int>& x);
00039
00040
00041  public:
00042
00044  time_series_encoder(int n_): n(n_) {}
00045
00047
00051    pair<vector<int>, mpz_class> encode(const vector<int>& x);
00052 };
00053
00054
00055
00056 //=================================================
00057 //          time_series_decoder
00058 //=================================================
00059
00060
00061
00062
00064
00081 class time_series_decoder
00082 {
00083    int n;
00084    int alph_size;
00085    vector<int> freq;
00086    b_graph G;
00087
00088  public:
00090  time_series_decoder(int n_): n(n_) {}
00091
00093    vector<int> decode(pair<vector<int>, mpz_class>);
00094 };
00095
00096
00097
00098 #endif
```

Index