

## Marked Graph Compression

Generated by Doxygen 1.8.13



# Contents

<b>1</b>	<b>Class Index</b>	<b>1</b>
1.1	Class List . . . . .	1
<b>2</b>	<b>File Index</b>	<b>3</b>
2.1	File List . . . . .	3
<b>3</b>	<b>Class Documentation</b>	<b>5</b>
3.1	b_graph Class Reference . . . . .	5
3.1.1	Detailed Description . . . . .	6
3.1.2	Constructor & Destructor Documentation . . . . .	6
3.1.2.1	b_graph() [1/2] . . . . .	6
3.1.2.2	b_graph() [2/2] . . . . .	7
3.1.3	Member Function Documentation . . . . .	7
3.1.3.1	get_adj_list() . . . . .	7
3.1.3.2	get_left_degree() . . . . .	8
3.1.3.3	get_left_degree_sequence() . . . . .	8
3.1.3.4	get_right_degree() . . . . .	8
3.1.3.5	get_right_degree_sequence() . . . . .	9
3.1.3.6	nu_left_vertices() . . . . .	9
3.1.3.7	nu_right_vertices() . . . . .	9
3.1.4	Friends And Related Function Documentation . . . . .	9
3.1.4.1	operator!= . . . . .	9
3.1.4.2	operator<< . . . . .	10
3.1.4.3	operator== . . . . .	10

3.1.5	Member Data Documentation . . . . .	10
3.1.5.1	adj_list . . . . .	10
3.1.5.2	left_deg_seq . . . . .	11
3.1.5.3	n . . . . .	11
3.1.5.4	np . . . . .	11
3.1.5.5	right_deg_seq . . . . .	11
3.2	b_graph_decoder Class Reference . . . . .	11
3.2.1	Detailed Description . . . . .	12
3.2.2	Constructor & Destructor Documentation . . . . .	13
3.2.2.1	b_graph_decoder() . . . . .	13
3.2.3	Member Function Documentation . . . . .	13
3.2.3.1	decode() . . . . .	13
3.2.3.2	decode_interval() . . . . .	14
3.2.3.3	decode_node() . . . . .	15
3.2.3.4	init() . . . . .	15
3.2.4	Member Data Documentation . . . . .	16
3.2.4.1	a . . . . .	16
3.2.4.2	b . . . . .	16
3.2.4.3	beta . . . . .	16
3.2.4.4	n . . . . .	16
3.2.4.5	np . . . . .	16
3.2.4.6	U . . . . .	17
3.2.4.7	W . . . . .	17
3.2.4.8	x . . . . .	17
3.3	b_graph_encoder Class Reference . . . . .	17
3.3.1	Detailed Description . . . . .	18
3.3.2	Constructor & Destructor Documentation . . . . .	18
3.3.2.1	b_graph_encoder() . . . . .	18
3.3.3	Member Function Documentation . . . . .	19
3.3.3.1	compute_N() . . . . .	19

3.3.3.2	<a href="#">encode()</a>	20
3.3.3.3	<a href="#">init()</a>	20
3.3.4	<a href="#">Member Data Documentation</a>	20
3.3.4.1	<a href="#">a</a>	20
3.3.4.2	<a href="#">b</a>	21
3.3.4.3	<a href="#">beta</a>	21
3.3.4.4	<a href="#">U</a>	21
3.4	<a href="#">colored_graph Class Reference</a>	21
3.4.1	<a href="#">Detailed Description</a>	22
3.4.2	<a href="#">Constructor &amp; Destructor Documentation</a>	23
3.4.2.1	<a href="#">colored_graph()</a>	23
3.4.3	<a href="#">Member Function Documentation</a>	23
3.4.3.1	<a href="#">init()</a>	23
3.4.4	<a href="#">Member Data Documentation</a>	24
3.4.4.1	<a href="#">adj_list</a>	24
3.4.4.2	<a href="#">adj_location</a>	25
3.4.4.3	<a href="#">Delta</a>	25
3.4.4.4	<a href="#">G</a>	25
3.4.4.5	<a href="#">h</a>	25
3.4.4.6	<a href="#">M</a>	25
3.4.4.7	<a href="#">nu_vertices</a>	25
3.4.4.8	<a href="#">ver_type</a>	26
3.4.4.9	<a href="#">ver_type_dict</a>	26
3.4.4.10	<a href="#">ver_type_int</a>	26
3.4.4.11	<a href="#">ver_type_list</a>	26
3.5	<a href="#">fenwick_tree Class Reference</a>	26
3.5.1	<a href="#">Detailed Description</a>	27
3.5.2	<a href="#">Constructor &amp; Destructor Documentation</a>	27
3.5.2.1	<a href="#">fenwick_tree()</a> [1/2]	27
3.5.2.2	<a href="#">fenwick_tree()</a> [2/2]	27

3.5.3	Member Function Documentation . . . . .	28
3.5.3.1	add() . . . . .	28
3.5.3.2	size() . . . . .	28
3.5.3.3	sum() . . . . .	28
3.5.4	Member Data Documentation . . . . .	29
3.5.4.1	sums . . . . .	29
3.6	graph Class Reference . . . . .	29
3.6.1	Detailed Description . . . . .	30
3.6.2	Constructor & Destructor Documentation . . . . .	30
3.6.2.1	graph() . . . . .	30
3.6.3	Member Function Documentation . . . . .	31
3.6.3.1	get_degree() . . . . .	31
3.6.3.2	get_degree_sequence() . . . . .	31
3.6.3.3	get_forward_degree() . . . . .	31
3.6.3.4	get_forward_list() . . . . .	32
3.6.3.5	nu_vertices() . . . . .	32
3.6.4	Friends And Related Function Documentation . . . . .	32
3.6.4.1	operator!= . . . . .	32
3.6.4.2	operator<< . . . . .	33
3.6.4.3	operator== . . . . .	33
3.6.5	Member Data Documentation . . . . .	33
3.6.5.1	degree_sequence . . . . .	33
3.6.5.2	forward_adj_list . . . . .	34
3.6.5.3	n . . . . .	34
3.7	graph_decoder Class Reference . . . . .	34
3.7.1	Detailed Description . . . . .	35
3.7.2	Constructor & Destructor Documentation . . . . .	35
3.7.2.1	graph_decoder() . . . . .	35
3.7.3	Member Function Documentation . . . . .	35
3.7.3.1	decode() . . . . .	35

3.7.3.2	<a href="#">decode_interval()</a>	35
3.7.3.3	<a href="#">decode_node()</a>	36
3.7.4	<a href="#">Member Data Documentation</a>	37
3.7.4.1	<a href="#">a</a>	37
3.7.4.2	<a href="#">beta</a>	38
3.7.4.3	<a href="#">logn2</a>	38
3.7.4.4	<a href="#">n</a>	38
3.7.4.5	<a href="#">tS</a>	38
3.7.4.6	<a href="#">U</a>	38
3.7.4.7	<a href="#">x</a>	38
3.8	<a href="#">graph_encoder Class Reference</a>	39
3.8.1	<a href="#">Detailed Description</a>	39
3.8.2	<a href="#">Constructor &amp; Destructor Documentation</a>	39
3.8.2.1	<a href="#">graph_encoder()</a>	40
3.8.3	<a href="#">Member Function Documentation</a>	40
3.8.3.1	<a href="#">compute_N()</a>	40
3.8.3.2	<a href="#">encode()</a>	41
3.8.3.3	<a href="#">init()</a>	42
3.8.4	<a href="#">Member Data Documentation</a>	42
3.8.4.1	<a href="#">beta</a>	42
3.8.4.2	<a href="#">G</a>	42
3.8.4.3	<a href="#">logn2</a>	42
3.8.4.4	<a href="#">Stilde</a>	43
3.8.4.5	<a href="#">U</a>	43
3.9	<a href="#">graph_message Class Reference</a>	43
3.9.1	<a href="#">Detailed Description</a>	44
3.9.2	<a href="#">Constructor &amp; Destructor Documentation</a>	44
3.9.2.1	<a href="#">graph_message()</a>	44
3.9.3	<a href="#">Member Function Documentation</a>	44
3.9.3.1	<a href="#">update_message_dictionary()</a>	44

3.9.3.2	<a href="#">update_messages()</a> . . . . .	45
3.9.4	<a href="#">Member Data Documentation</a> . . . . .	47
3.9.4.1	<a href="#">Delta</a> . . . . .	47
3.9.4.2	<a href="#">G</a> . . . . .	47
3.9.4.3	<a href="#">h</a> . . . . .	47
3.9.4.4	<a href="#">message_dict</a> . . . . .	47
3.9.4.5	<a href="#">message_list</a> . . . . .	48
3.9.4.6	<a href="#">messages</a> . . . . .	48
3.10	<a href="#">marked_graph Class Reference</a> . . . . .	48
3.10.1	<a href="#">Detailed Description</a> . . . . .	49
3.10.2	<a href="#">Constructor &amp; Destructor Documentation</a> . . . . .	49
3.10.2.1	<a href="#">marked_graph()</a> [1/2] . . . . .	49
3.10.2.2	<a href="#">marked_graph()</a> [2/2] . . . . .	49
3.10.3	<a href="#">Member Data Documentation</a> . . . . .	50
3.10.3.1	<a href="#">adj_list</a> . . . . .	50
3.10.3.2	<a href="#">adj_location</a> . . . . .	50
3.10.3.3	<a href="#">nu_vertices</a> . . . . .	50
3.10.3.4	<a href="#">ver_mark</a> . . . . .	50
3.11	<a href="#">reverse_fenwick_tree Class Reference</a> . . . . .	51
3.11.1	<a href="#">Detailed Description</a> . . . . .	51
3.11.2	<a href="#">Constructor &amp; Destructor Documentation</a> . . . . .	51
3.11.2.1	<a href="#">reverse_fenwick_tree()</a> [1/2] . . . . .	51
3.11.2.2	<a href="#">reverse_fenwick_tree()</a> [2/2] . . . . .	52
3.11.3	<a href="#">Member Function Documentation</a> . . . . .	52
3.11.3.1	<a href="#">add()</a> . . . . .	52
3.11.3.2	<a href="#">size()</a> . . . . .	52
3.11.3.3	<a href="#">sum()</a> . . . . .	52
3.11.4	<a href="#">Member Data Documentation</a> . . . . .	53
3.11.4.1	<a href="#">FT</a> . . . . .	53



<b>4 File Documentation</b>	<b>55</b>
4.1 bipartite_graph.cpp File Reference	55
4.1.1 Function Documentation	55
4.1.1.1 operator!=()	55
4.1.1.2 operator<<()	56
4.1.1.3 operator==()	56
4.2 bipartite_graph.h File Reference	56
4.3 bipartite_graph_compression.cpp File Reference	57
4.4 bipartite_graph_compression.h File Reference	57
4.5 compression_helper.cpp File Reference	57
4.5.1 Function Documentation	57
4.5.1.1 binomial()	57
4.5.1.2 compute_product()	58
4.5.1.3 prod_factorial()	59
4.6 compression_helper.h File Reference	59
4.6.1 Function Documentation	59
4.6.1.1 binomial()	59
4.6.1.2 compute_product()	60
4.6.1.3 prod_factorial()	61
4.7 fenwick.cpp File Reference	61
4.8 fenwick.h File Reference	61
4.9 graph_message.cpp File Reference	61
4.9.1 Function Documentation	62
4.9.1.1 pair_compare()	62
4.10 graph_message.h File Reference	62
4.10.1 Function Documentation	62
4.10.1.1 pair_compare()	63
4.11 marked_graph.cpp File Reference	63
4.11.1 Function Documentation	63
4.11.1.1 operator>>()	63

4.12	marked_graph.h File Reference . . . . .	64
4.12.1	Function Documentation . . . . .	64
4.12.1.1	operator>>() . . . . .	64
4.13	simple_graph.cpp File Reference . . . . .	65
4.13.1	Function Documentation . . . . .	65
4.13.1.1	operator"!=( ) . . . . .	65
4.13.1.2	operator<<() . . . . .	65
4.13.1.3	operator==( ) . . . . .	66
4.14	simple_graph.h File Reference . . . . .	66
4.15	simple_graph_compression.cpp File Reference . . . . .	66
4.16	simple_graph_compression.h File Reference . . . . .	66
4.17	test.cpp File Reference . . . . .	67
4.17.1	Function Documentation . . . . .	67
4.17.1.1	main() . . . . .	67
4.17.1.2	operator<<() . . . . .	68
	<b>Index</b>	<b>69</b>

# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">b_graph</a>	Simple unmarked bipartite graph . . . . .	5
<a href="#">b_graph_decoder</a>	Decodes a simple unmarked bipartite graph . . . . .	11
<a href="#">b_graph_encoder</a>	Encodes a simple unmarked bipartite graph . . . . .	17
<a href="#">colored_graph</a>	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 . . . . .	21
<a href="#">fenwick_tree</a>	Fenwick tree class . . . . .	26
<a href="#">graph</a>	Simple unmarked graph . . . . .	29
<a href="#">graph_decoder</a>	Decodes a simple unmarked graph . . . . .	34
<a href="#">graph_encoder</a>	Encodes a simple unmarked graph . . . . .	39
<a href="#">graph_message</a>	This class takes care of message passing on marked graphs . . . . .	43
<a href="#">marked_graph</a>	Simple marked graph . . . . .	48
<a href="#">reverse_fenwick_tree</a>	Similar to the <a href="#">fenwick_tree</a> class, but instead of prefix sums, this class computes suffix sums . . . . .	51



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

<a href="#">bipartite_graph.cpp</a>	55
<a href="#">bipartite_graph.h</a>	56
<a href="#">bipartite_graph_compression.cpp</a>	57
<a href="#">bipartite_graph_compression.h</a>	57
<a href="#">compression_helper.cpp</a>	57
<a href="#">compression_helper.h</a>	59
<a href="#">fenwick.cpp</a>	61
<a href="#">fenwick.h</a>	61
<a href="#">graph_message.cpp</a>	61
<a href="#">graph_message.h</a>	62
<a href="#">marked_graph.cpp</a>	63
<a href="#">marked_graph.h</a>	64
<a href="#">simple_graph.cpp</a>	65
<a href="#">simple_graph.h</a>	66
<a href="#">simple_graph_compression.cpp</a>	66
<a href="#">simple_graph_compression.h</a>	66
<a href="#">test.cpp</a>	67



## Chapter 3

# Class Documentation

### 3.1 b\_graph Class Reference

simple unmarked bipartite graph

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

#### Public Member Functions

- [b\\_graph](#) (vector< vector< int > > list, vector< int > right\_deg)  
*a constructor*
- [b\\_graph](#) (vector< vector< int > > list)  
*a constructor*
- vector< int > [get\\_adj\\_list](#) (int v) const  
*returns the adjacency list of a given left vertex*
- int [get\\_right\\_degree](#) (int v) const  
*returns the degree of a right vertex v*
- int [get\\_left\\_degree](#) (int v) const  
*returns the degree of a right vertex v*
- vector< int > [get\\_right\\_degree\\_sequence](#) () const  
*return the right degree sequence*
- vector< int > [get\\_left\\_degree\\_sequence](#) () const  
*return the left degree sequence*
- int [nu\\_left\\_vertices](#) () const  
*returns the number of left vertices*
- int [nu\\_right\\_vertices](#) () const  
*returns the number of right vertices*

#### Private Attributes

- int [n](#)  
*the number of left vertices*
- int [np](#)  
*the number of right vertices*
- vector< vector< int > > [adj\\_list](#)  
*adjacency list for left vertices, where for  $0 \leq v < n$ ,  $adj\_list[v]$  is a sorted list of right vertices connected to v.*
- vector< int > [left\\_deg\\_seq](#)  
*degree sequence for left vertices, where  $left\_deg\_seq[v]$  is the degree of the left node v*
- vector< int > [right\\_deg\\_seq](#)  
*degree sequence for right vertices, where  $left\_deg\_seq[v]$  is the degree of the right node v*

## Friends

- ostream & `operator<<` (ostream &o, const `b_graph` &G)  
*printing the graph to the output*
- bool `operator==` (const `b_graph` &G1, const `b_graph` &G2)  
*comparing two graphs for equality*
- bool `operator!=` (const `b_graph` &G1, const `b_graph` &G2)  
*comparing for inequality*

### 3.1.1 Detailed Description

simple unmarked bipartite graph

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

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

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

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

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

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 `b_graph()` [1/2]

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

a constructor

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

#### Parameters

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



```

4 {
5     n = list.size();
6     np = right_deg.size(); // the number of right nodes
7     adj_list = list;
8     left_deg_seq.resize(n);
9     // sorting the list
10    for (int v=0; v<n; v++){
11        sort(adj_list[v].begin(), adj_list[v].end());
12        left_deg_seq[v] = adj_list[v].size();
13    }
14    right_deg_seq = right_deg;
15 }

```

### 3.1.2.2 b\_graph() [2/2]

```

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

```

a constructor

This constructor takes the list of adjacent vertices

#### Parameters

<i>list</i>	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
-------------	---

```

18 {
19     // goal: finding right degrees and calling the above constructor
20     // first, we find the number of right nodes
21     np = 0; // the number of right nodes
22     n = list.size();
23     adj_list = list;
24     left_deg_seq.resize(n);
25     for (int v=0; v<adj_list.size(); v++){
26         sort(adj_list[v].begin(), adj_list[v].end());
27         if (adj_list[v][adj_list[v].size()-1] > np)
28             np = list[v][adj_list[v].size()-1];
29         left_deg_seq[v] = adj_list[v].size();
30     }
31     np++; // node indexing is zero based
32
33     right_deg_seq.resize(np);
34     fill(right_deg_seq.begin(), right_deg_seq.end(), 0); // make all elements 0
35     for (int v=0; v<list.size(); v++)
36         for (int i=0; i<list[v].size(); i++)
37             right_deg_seq[list[v][i]]++;
38 }

```

## 3.1.3 Member Function Documentation

### 3.1.3.1 get\_adj\_list()

```

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

```

returns the adjacency list of a given left vertex

```
41 {  
42     if (v < 0 or v >= n)  
43         cerr << "b_graph::get_adj_list, index v out of range" << endl;  
44     return adj_list[v];  
45 }
```

### 3.1.3.2 get\_left\_degree()

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

returns the degree of a right vertex v

```
56 {  
57     if (v < 0 or v >= n)  
58         cerr << "b_graph::get_left_degree, index v out of range" << endl;  
59     return left_deg_seq[v];  
60 }
```

### 3.1.3.3 get\_left\_degree\_sequence()

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

return the left degree sequence

```
68 {  
69     return left_deg_seq;  
70 }
```

### 3.1.3.4 get\_right\_degree()

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

returns the degree of a right vertex v

```
49 {  
50     if (v < 0 or v >= n)  
51         cerr << "b_graph::get_right_degree, index v out of range" << endl;  
52     return right_deg_seq[v];  
53 }
```

### 3.1.3.5 get\_right\_degree\_sequence()

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

return the right degree sequence

```
63 {  
64     return right_deg_seq;  
65 }
```

### 3.1.3.6 nu\_left\_vertices()

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

returns the number of left vertices

```
74 {  
75     return n;  
76 }
```

### 3.1.3.7 nu\_right\_vertices()

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

returns the number of right vertices

```
79 {  
80     return np;  
81 }
```

## 3.1.4 Friends And Related Function Documentation

### 3.1.4.1 operator!=

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

comparing for inequality

```
121 {  
122     return !(G1 == G2);  
123 }
```

### 3.1.4.2 operator<<

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

printing the graph to the output

```
85 {
86     int n = G.nu_left_vertices();
87     vector<int> list;
88     for (int i=0;i<n;i++){
89         list = G.get_adj_list(i);
90         o << i << " -> ";
91         for (int j=0;j<list.size();j++){
92             o << list[j];
93             if (j < list.size()-1)
94                 o << ", ";
95         }
96         o << endl;
97     }
98     return o;
99 }
```

### 3.1.4.3 operator==

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

comparing two graphs for equality

```
102 {
103     int n1 = G1.nu_left_vertices();
104     int n2 = G2.nu_left_vertices();
105
106     int np1 = G1.nu_right_vertices();
107     int np2 = G2.nu_right_vertices();
108     if (n1!= n2 or np1 != np2)
109         return false;
110     vector<int> list1, list2;
111     for (int v=0; v<n1; v++){
112         list1 = G1.get_adj_list(v);
113         list2 = G2.get_adj_list(v);
114         if (list1 != list2)
115             return false;
116     }
117     return true;
118 }
```

## 3.1.5 Member Data Documentation

### 3.1.5.1 adj\_list

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

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

### 3.1.5.2 left\_deg\_seq

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

degree sequence for left vertices, where left\_deg\_seq[v] is the degree of the left node v

### 3.1.5.3 n

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

the number of left vertices

### 3.1.5.4 np

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

the number of right vertices

### 3.1.5.5 right\_deg\_seq

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

degree sequence for right vertices, where left\_deg\_seq[v] is the degree of the right node v

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

- [bipartite\\_graph.h](#)
- [bipartite\\_graph.cpp](#)

## 3.2 b\_graph\_decoder Class Reference

Decodes a simple unmarked bipartite graph.

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

## Public Member Functions

- [b\\_graph\\_decoder](#) (vector< int > a\_, vector< int > b\_)  
*constructor*
- void [init](#) ()  
*initializes x as empty list of size n, beta as b, U with b and W with a*
- pair< mpz\_class, mpz\_class > [decode\\_node](#) (int i, mpz\_class tN)  
*decodes the connectivity list of a left node  $0 \leq i < n$  given  $\tilde{N}_{i,i}$*
- pair< mpz\_class, mpz\_class > [decode\\_interval](#) (int i, int j, mpz\_class tN)  
*decodes the connectivity list of left vertices  $i \leq v \leq j$  given  $\tilde{N}_{i,j}$*
- [b\\_graph\\_decode](#) (mpz\_class f)  
*decodes the bipartite graph given the encoded integer*

## Private Attributes

- int [n](#)  
*number of left vertices*
- int [np](#)  
*number of right vertices*
- vector< int > [a](#)  
*left degree sequence*
- vector< int > [b](#)  
*right degree sequence*
- vector< vector< int > > [x](#)  
*the adjacency list of left nodes for the decoded graph*
- [reverse\\_fenwick\\_tree](#) U  
*reverse Fenwick tree initialized with the right degree sequence b, and after decoding vertex i, for  $0 \leq v < n'$ , we have  $U_v = \sum_{k=v}^{n'-1} b_k(i)$*
- [reverse\\_fenwick\\_tree](#) W  
*keeping partial sums for the degree sequence a. More precisely, for  $0 \leq v < n$ , we have  $W_v = \sum_{k=v}^{n-1} a_k$*
- vector< int > [beta](#)  
*the sequence  $\vec{\beta}$ , where before decoding vertex i, for  $0 \leq v < n'$ , we have  $\beta_v = b_v(i)$*

### 3.2.1 Detailed Description

Decodes a simple unmarked bipartite graph.

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

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

Then, if variable f of type mpz\_class is obtained from a [b\\_graph\\_encoder](#) class, we can reconstruct the graph using f:

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

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

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

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

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

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

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

## 3.2.2 Constructor & Destructor Documentation

### 3.2.2.1 b\_graph\_decoder()

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

constructor

```
97 {
98     a = a_;
99     b = b_;
100     n = a.size();
101     np = b.size();
102     init();
103 }
```

## 3.2.3 Member Function Documentation

### 3.2.3.1 decode()

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

decodes the bipartite graph given the encoded integer

Parameters

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

**Returns**

the decoded bipartite graph  $G$

```

179 {
180     mpz_class prod_b_factorial = prod_factorial(b, 0, np-1);
181     mpz_class tN = f * prod_b_factorial;
182     decode_interval(0,n-1,tN);
183     return b_graph(x, b);
184 }
```

**3.2.3.2 decode\_interval()**

```

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

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

**Parameters**

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

**Returns**

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

```

150 {
151     if (i==j)
152         return decode_node(i,tN);
153     int k = (i+j)/ 2; // midpoint to break
154     int Wk = W.sum(k+1);
155     int Wj = W.sum(j+1);
156     mpz_class rkj = compute_product(Wk, Wk - Wj, 1) /
157         prod_factorial(a, k+1, j); // r_{t+1, j}
158     mpz_class tNik = tN / rkj; // \tilde{N}_{i,k}
159     pair<mpz_class, mpz_class> ans; // to keep the return for each subinterval
160     // calling the left subinterval
161     ans = decode_interval(i,k,tNik);
162     // preparing for the right subinterval
163     mpz_class Nik = ans.first;
164     mpz_class lik = ans.second;
165     mpz_class tNkj = (tN - Nik * rkj) / lik; // \tilde{N}_{k+1, j}
166     // calling the right subinterval
167     ans = decode_interval(k+1, j, tNkj);
168     mpz_class Nkj = ans.first;
169     mpz_class lkj = ans.second;
170     mpz_class Nij = Nik * rkj + lik * Nkj;
171     mpz_class lij = lik * lkj;
172     return pair<mpz_class, mpz_class> (Nij, lij);
173 }
```



## 3.2.3.3 decode\_node()

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

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

## Parameters

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

## Returns

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

```
116 {
117     mpz_class li = 1;
118     mpz_class Ni = 0;
119     int f, g; // endpoints of the interval for binary search
120     int v;
121     mpz_class y; // helper
122     x[i].clear(); // make sure nothing is in the list to be decoded
123     for (int k=0; k<a[i]; k++){
124         // finding x[i][k]
125         if (k==0)
126             f = 0;
127         else
128             f = 1 + x[i][k-1];
129         g = np-1;
130         while (g > f){
131             v = (f+g)/2;
132             if (binomial(U.sum(1+v), a[i] - k) <= tN)
133                 g = v;
134             else
135                 f = v + 1;
136         }
137         x[i].push_back(f); // decoded the kth connection of vertex i
138         y = binomial(U.sum(1+x[i][k]), a[i] - k);
139         tN = (tN - y) / beta[x[i][k]];
140         Ni += li * y;
141         li *= beta[x[i][k]];
142         beta[x[i][k]]--;
143         U.add(x[i][k], -1);
144     }
145     return pair<mpz_class, mpz_class>(Ni, li);
146 }
```

## 3.2.3.4 init()

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

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

```
106 {
107     x.clear();
108     x.resize(n);
109     beta = b;
110     U = reverse_fenwick_tree(b);
111     W = reverse_fenwick_tree(a);
112 }
113 }
```

### 3.2.4 Member Data Documentation

#### 3.2.4.1 a

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

left degree sequence

#### 3.2.4.2 b

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

right degree sequence

#### 3.2.4.3 beta

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

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

#### 3.2.4.4 n

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

number of left vertices

#### 3.2.4.5 np

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

number of right vertices

## 3.2.4.6 U

```
reverse_fenwick_tree b_graph_decoder::U [private]
```

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

## 3.2.4.7 W

```
reverse_fenwick_tree b_graph_decoder::W [private]
```

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

## 3.2.4.8 x

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

the adjacency list of left nodes for the decoded graph

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

- [bipartite\\_graph\\_compression.h](#)
- [bipartite\\_graph\\_compression.cpp](#)

## 3.3 b\_graph\_encoder Class Reference

Encodes a simple unmarked bipartite graph.

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

## Public Member Functions

- [b\\_graph\\_encoder](#) (vector< int > a\_, vector< int > b\_)  
*constructor*
- void [init](#) (const [b\\_graph](#) &G)  
*initializes beta and U*
- pair< mpz\_class, mpz\_class > [compute\\_N](#) (int i, int j, const [b\\_graph](#) &G)  
*computes  $N_{i,j}(G)$*
- mpz\_class [encode](#) (const [b\\_graph](#) &G)  
*encodes the given bipartite graph  $G$  and returns an integer in the specified range*

## Private Attributes

- `vector< int > beta`  
*when compute\_N is called for  $i \leq j$ , for  $i \leq v \leq n$ , we have  $\text{beta}[v] = b_v(i)$*
- `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  $U.\text{sum}[v] = \sum_{k=v}^n b_k(i)$ .*

### 3.3.1 Detailed Description

Encodes a simple unmarked bipartite graph.

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

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

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

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

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

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

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

### 3.3.2 Constructor & Destructor Documentation

#### 3.3.2.1 b\_graph\_encoder()

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

constructor

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

### 3.3.3 Member Function Documentation

#### 3.3.3.1 compute\_N()

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

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

##### Parameters

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

##### Returns

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

```
24 {
25     mpz_class Nij = 0;
26     mpz_class lij = 1;
27     if (i == j){
28         vector<int> x = G.get_adj_list(i); // the adjacency list of the vertex
29         for (int k=0;k<a[i];k++){
30             Nij += lij * binomial(U.sum(1+x[k]), a[i] - k);
31             lij *= beta[x[k]];
32             beta[x[k]] --;
33             U.add(x[k],-1);
34         }
35         return pair<mpz_class, mpz_class> (Nij, lij);
36     }else{
37         int t = (i+j)/ 2;
38         mpz_class Nit, lit; // for the left subinterval i, j
39         mpz_class Ntj, ltj; // for the right subinterval t+1, j
40         mpz_class Nij, lij; // to return
41         int St; // S_{t+1}
42         int Sj; // S_{j+1}
43
44         pair<mpz_class, mpz_class> ans; // for collecting the results from subintervals
45
46         // left subinterval
47         ans = compute_N(i,t, G);
48         Nit = ans.first;
49         lit = ans.second;
50         St = U.sum(0);
51
52         // right subinterval
53         ans = compute_N(t+1, j, G);
54         Ntj = ans.first;
55         ltj = ans.second;
56         Sj = U.sum(0);
57
58         mpz_class rtj; // r_{t+1, j}
59         mpz_class prod_afac = prod_factorial(a, t+1, j); // the product of a_k! for t + 1 <= k
60         //<= j
61
62         rtj = compute_product(St, St - Sj, 1) / prod_afac;
63
64         Nij = Nit * rtj + lit * Ntj;
65         lij = lit * ltj;
66         return pair<mpz_class, mpz_class>(Nij, lij);
67     }
68 }
```

### 3.3.3.2 encode()

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

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

```
73 {
74     if (a != G.get_left_degree_sequence() or b != G.
75         get_right_degree_sequence())
76         cerr << " WARNING b_graph_encoder::encoder : vectors a and/or b do not match with the degree sequences
77             of the given bipartite graph " << endl;
78     init(G); // initialize U and beta for G
79     pair<mpz_class, mpz_class> ans = compute_N(0,G.nu_left_vertices()-1, G);
80     mpz_class prod_b_factorial = prod_factorial(b, 0, b.size()-1); // \prod_{i=0}^{n-1} b_i
81
82     bool ceil = false;
83     if (ans.first % prod_b_factorial != 0)
84         ceil = true;
85     ans.first /= prod_b_factorial;
86     if (ceil)
87         ans.first ++;
88     return ans.first;
89 }
```

### 3.3.3.3 init()

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

initializes beta and U

```
9 {
10     // initializing beta
11     beta = G.get_right_degree_sequence();
12
13     // initializing the Fenwick tree
14     U = reverse_fenwick_tree(beta);
15
16     if (a != G.get_left_degree_sequence() or b != G.
17         get_right_degree_sequence())
18         cerr << " WARNING b_graph_encoder::init : vectors a and/or b do not match with the degree sequences of
19             the given bipartite graph " << endl;
19 }
```

## 3.3.4 Member Data Documentation

### 3.3.4.1 a

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

the degree sequence for the left nodes

## 3.3.4.2 b

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

the degree sequence for the right nodes

## 3.3.4.3 beta

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

when compute\_N is called for  $i \leq j$ , for  $i \leq v \leq n$ , we have  $\text{beta}[v] = b_v(i)$

## 3.3.4.4 U

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

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

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

- [bipartite\\_graph\\_compression.h](#)
- [bipartite\\_graph\\_compression.cpp](#)

## 3.4 colored\_graph Class Reference

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

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

## Public Member Functions

- [colored\\_graph](#) (const [marked\\_graph](#) &graph, int depth, int max\_degree)  
*constructor from a graph, depth and maximum degree parameters*
- void [init](#) ()  
*initializes other variables*

## Public Attributes

- const [marked\\_graph](#) & G  
*the marked graph from which this is created*
- int h  
*the depth up to which look at edge types*
- int Delta  
*the maximum degree threshold*
- [graph\\_message](#) M  
*we use the message passing algorithm of class [graph\\_message](#) to find out edge types*
- int nu\_vertices  
*the number of vertices in the graph.*
- vector< vector< pair< int, pair< int, int > > > > [adj\\_list](#)  
*adj\_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, color component towards i, color component towards other endpoint). Therefore, the color of an edge between v and its ith neighbor is of the form (adj\_list[v][i].second.first, adj\_list[v][i].second.second)*
- vector< map< int, int > > [adj\\_location](#)  
*adj\_location[v] for  $0 \leq v < n$ , is a map, where adj\_location[v][w] denotes the index in adj\_list[v] where the information regarding the edge between v and w is stored. Hence, adj\_location[v][w] does not exist if w is not adjacent to v, and adj\_list[v][adj\_location[v][w]] is the edge between v and w*
- vector< vector< int > > [ver\\_type](#)  
*vertex mark and the colored degree matrix of each vertex. For a vertex v, D[v] is a vector of size  $1 + L \times L$ , where the first entry is the vertex mark, and the rest is the colored degree matrix row by row. Here, L denotes the number of colors.*
- map< vector< int >, int > [ver\\_type\\_dict](#)  
*the dictionary mapping vertex types to integers, obtained from the ver\_type array defined above*
- vector< vector< int > > [ver\\_type\\_list](#)  
*the list of all distinct vertex types, obtained from the ver\_type array. This is constructed in such a way that ver\_type↔\_list[ver\_type\_dict[x]] = x*
- vector< int > [ver\\_type\\_int](#)  
*vertex type converted to integers, using the ver\_type\_dict map, i.e. ver\_type\_int[v] = ver\_type\_dict[ver\_type[v]]*

### 3.4.1 Detailed Description

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

quick member overview:

- There is a reference to a [marked\\_graph](#) object G,
- h and Delta are parameters that determine depth and maximum degree to form edge types,
- M is a member with type [graph\\_message](#) that is used to form edge types,
- nu\_vertices: number of vertices in the graph
- adj\_list: the adjacency list of vertices, which also includes edge colors
- adj\_location: map for finding where neighbors of vertices are in the adjacency list
- ver\_type: a vector for each vertex, containing mark + vectorized degree matrix
- ver\_type\_dict: dictionary mapping vertex mark + degree matrix to integer
- ver\_type\_list: list of "distinct" vertex types



- `ver_type_int`: vertex types converted to integers

### Sample Usage

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

## 3.4.2 Constructor & Destructor Documentation

### 3.4.2.1 colored\_graph()

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

constructor from a graph, depth and maximum degree parameters

```
104         M(graph, depth, max_degree), h(depth), Delta(max_degree) : G(graph),
105     {
106         init(); // initialize other variables
107     }
```

## 3.4.3 Member Function Documentation

### 3.4.3.1 init()

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

initializes other variables

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

```

169 {
170     nu_vertices = G.nu_vertices;
171     adj_location = G.adj_location; // neighborhood structure is the same as the
        given graph
172     // assigning edge colors based on the messages given by M
173     M.update_messages();
174     adj_list.resize(nu_vertices);
175
176     // updating adj_list
177     int w, my_location, color_v, color_w;
178     for (int v=0;v<nu_vertices;v++){
179         adj_list[v].resize(G.adj_list[v].size()); // the same number of neighbors here
180         for (int i=0;i<G.adj_list[v].size();i++){
181             w = G.adj_list[v][i].first; // the ith neighbor, the same as in G
182             my_location = G.adj_location[w].at(v); // where v stands among the neighbors of w
183             color_v = M.message_dict[M.messages[v][i][h-1]]; // the color towards v
        corresponds to the message v sends to w
184             color_w = M.message_dict[M.messages[w][my_location][
        h-1]]; // the color towards w is the message w sends towards v
185             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
186         }
187     }
188
189     // updating the vertex type sequence, dictionary and list, i.e. variables ver_type, ver_type_dict and
        ver_type_list
190     // we also update ver_type_int
191
192     int L = M.message_list.size(); // the number of messages
193     ver_type.resize(nu_vertices);
194     ver_type_int.resize(nu_vertices);
195     for (int v=0;v<nu_vertices;v++){
196         ver_type[v].resize(1 + L * L);
197         ver_type[v][0] = G.ver_mark[v];
198         for (int i=0;i<adj_list[v].size();i++){
199             //if (adj_list[v][i].second.first < M.message_list.size()){ // equivalently, the edge is not * typed,
        since all * typed messages are after L by sorting
200             ver_type[v][1 + adj_list[v][i].second.first * L +
        adj_list[v][i].second.second] ++;
201             //}
202         }
203         if (ver_type_dict.find(ver_type[v]) == ver_type_dict.end()){
204             ver_type_list.push_back(ver_type[v]);
205             ver_type_dict[ver_type[v]] = ver_type_list.size() -1;
206         }
207         ver_type_int[v] = ver_type_dict[ver_type[v]];
208     }
209
210     // checking whether the sum of degrees is symmetric
211     vector<int> sum;
212     sum.resize(1 + L * L);
213     for (int v=0;v<nu_vertices;v++){
214         for (int i=0;i<1 + L * L;i++){
215             sum[i] += ver_type[v][i];
216         }
217         for (int j=0;j<L;j++){
218             cout << sum[1+i*L + j] << " ";
219             if (sum[1+i*L + j] != sum[1+j*L+i])
220                 cout << " DANGER! the sum matrix is not symmetric" << endl;
221         }
222         cout << endl;
223     }
224 }

```

### 3.4.4 Member Data Documentation

#### 3.4.4.1 adj\_list

vector<vector<pair<int, pair<int, int>>>> colored\_graph::adj\_list

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

#### 3.4.4.2 adj\_location

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

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

#### 3.4.4.3 Delta

```
int colored_graph::Delta
```

the maximum degree threshold

#### 3.4.4.4 G

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

the marked graph from which this is created

#### 3.4.4.5 h

```
int colored_graph::h
```

the depth up to which look at edge types

#### 3.4.4.6 M

```
graph_message colored_graph::M
```

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

#### 3.4.4.7 nu\_vertices

```
int colored_graph::nu_vertices
```

the number of vertices in the graph.

#### 3.4.4.8 ver\_type

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

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

#### 3.4.4.9 ver\_type\_dict

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

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

#### 3.4.4.10 ver\_type\_int

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

vertex type converted to integers, using the `ver_type_dict` map, i.e.  $\text{ver\_type\_int}[v] = \text{ver\_type\_dict}[\text{ver\_type}[v]]$

#### 3.4.4.11 ver\_type\_list

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

the list of all distinct vertex types, obtained from the `ver_type` array. This is constructed in such a way that  $\text{ver\_type\_list}[\text{ver\_type\_dict}[x]] = x$

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

- [graph\\_message.h](#)
- [graph\\_message.cpp](#)

## 3.5 fenwick\_tree Class Reference

Fenwick tree class.

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

## Public Member Functions

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

## Private Attributes

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

### 3.5.1 Detailed Description

Fenwick tree class.

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

### 3.5.2 Constructor & Destructor Documentation

#### 3.5.2.1 fenwick\_tree() [1/2]

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

default constructor

```
23     {
24         sums.resize(0);
25     }
```

#### 3.5.2.2 fenwick\_tree() [2/2]

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

constructor, which takes a vector of values and initializes

```
9 {
10     int n = vals.size();
11     sums.resize(n+1);
12     // initializes at zero
13     for (int i=1;i<=n;i++)
14         sums[i] = 0;
15     for (int i=0;i<n;i++)
16         add(i,vals[i]); // add values one by one
17 }
```

### 3.5.3 Member Function Documentation

#### 3.5.3.1 add()

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

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

##### Parameters

<i>k</i>	the index to be modified, this is zero based
<i>val</i>	the value to be added to the above index

```
20 {
21     k = k + 1; // the sums vector is one based while the index k was zero based
22     while (k < sums.size()){
23         sums[k] += val;
24         k += (k & -k);
25     }
26 }
```

#### 3.5.3.2 size()

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

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

```
32 {
33     return sums.size() - 1;
34 }
```

#### 3.5.3.3 sum()

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

returns the sum of values from 0 to k

##### Parameters

<i>k</i>	the index up to which (including) the sum is computed
----------	---

```

30 {
31     k = k + 1; // the sums vector is one based while the index k was zero based
32     int sum_computed = 0;
33     while (k > 0){
34         sum_computed += sums[k];
35         k -= (k & -k); // reduce the lsb bit
36     }
37     return sum_computed;
38 }

```

### 3.5.4 Member Data Documentation

#### 3.5.4.1 sums

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

a one based vector containing sum of values

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

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

- [fenwick.h](#)
- [fenwick.cpp](#)

## 3.6 graph Class Reference

simple unmarked graph

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

### Public Member Functions

- [graph](#) (vector< vector< int > > list, vector< int > deg)  
*a constructor*
- vector< int > [get\\_forward\\_list](#) (int v) const  
*returns the forward adjacency list of a given vertex*
- int [get\\_forward\\_degree](#) (int v) const  
*returns the forward degree of a vertex v*
- int [get\\_degree](#) (int v) const  
*returns the overall degree of a vertex*
- vector< int > [get\\_degree\\_sequence](#) () const  
*returns the whole degree sequence*
- int [nu\\_vertices](#) () const  
*the number of vertices in the graph*

## Private Attributes

- `int n`  
*the number of vertices in the graph*
- `vector< vector< int > > forward_adj_list`  
*for a vertex  $0 \leq v < n$ , `forward_adj_list[v]` is a vector containing vertices  $w$  such that are adjacent to  $v$  and also  $w > v$ , i.e. the adjacent vertices in the forward direction. For such  $v$ , `forward_adj_list[v]` is sorted increasing.*
- `vector< int > degree_sequence`  
*the degree sequence of the graph, where the degree of a vertex is the number of all edges connected to it (not just the ones with greater index).*

## Friends

- `ostream & operator<< (ostream &o, const graph &G)`  
*printing the graph to the output*
- `bool operator== (const graph &G1, const graph &G2)`  
*comparing two graphs for equality*
- `bool operator!= (const graph &G1, const graph &G2)`  
*comparing for inequality*

### 3.6.1 Detailed Description

simple unmarked graph

### 3.6.2 Constructor & Destructor Documentation

#### 3.6.2.1 graph()

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

a constructor

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

#### Parameters

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

```
9
10         {
11     n = list.size();
12     forward_adj_list = list;
13     // sorting the list
14     for (int v=0; v<n; v++)
```



```
14     sort(forward_adj_list[v].begin(), forward_adj_list[v].end());
15     degree_sequence = deg;
16 }
```

### 3.6.3 Member Function Documentation

#### 3.6.3.1 get\_degree()

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

returns the overall degree of a vertex

```
30                                     {
31     return degree_sequence[v];
32 }
```

#### 3.6.3.2 get\_degree\_sequence()

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

returns the whole degree sequence

```
34                                     {
35     return degree_sequence;
36 }
```

#### 3.6.3.3 get\_forward\_degree()

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

returns the forward degree of a vertex v

```
24                                     {
25     if (v < 0 or v >= n)
26         cerr << "graph::get_forward_degree, index v out of range" << endl;
27     return forward_adj_list[v].size();
28 }
```

### 3.6.3.4 get\_forward\_list()

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

returns the forward adjacency list of a given vertex

```
18                                     {
19     if (v < 0 or v >= n)
20         cerr << "graph::get_forward_list, index v out of range" << endl;
21     return forward_adj_list[v];
22 }
```

### 3.6.3.5 nu\_vertices()

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

the number of vertices in the graph

```
39                                     {
40     return n;
41 }
```

## 3.6.4 Friends And Related Function Documentation

### 3.6.4.1 operator!=

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

comparing for inequality

```
77 {
78     return !(G1 == G2);
79 }
```

## 3.6.4.2 operator&lt;&lt;

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

printing the graph to the output

```
44 {
45     int n = G.nu_vertices();
46     vector<int> list;
47     for (int i=0;i<n;i++){
48         list = G.get_forward_list(i);
49         o << i << " -> ";
50         for (int j=0;j<list.size();j++){
51             o << list[j];
52             if (j < list.size()-1)
53                 o << ", ";
54         }
55         o << endl;
56     }
57     return o;
58 }
```

## 3.6.4.3 operator==

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

comparing two graphs for equality

```
61 {
62     int n1 = G1.nu_vertices();
63     int n2 = G2.nu_vertices();
64     if (n1!= n2)
65         return false;
66     vector<int> list1, list2;
67     for (int v=0; v<n1; v++){
68         list1 = G1.get_forward_list(v);
69         list2 = G2.get_forward_list(v);
70         if (list1 != list2)
71             return false;
72     }
73     return true;
74 }
```

## 3.6.5 Member Data Documentation

## 3.6.5.1 degree\_sequence

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

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

### 3.6.5.2 forward\_adj\_list

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

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

### 3.6.5.3 n

```
int graph::n [private]
```

the number of vertices in the graph

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

- [simple\\_graph.h](#)
- [simple\\_graph.cpp](#)

## 3.7 graph\_decoder Class Reference

Decodes a simple unmarked graph.

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

### Public Member Functions

- [graph\\_decoder](#) (vector< int > a\_)  
*constructor given the degree sequence*
- [graph decode](#) (mpz\_class f, vector< int > tS\_)  
*given  $\tilde{N}$  and a vector  $\tilde{S}$ , decodes the graph and returns an object of type graph*
- pair< mpz\_class, mpz\_class > [decode\\_node](#) (int i, mpz\_class tN)  
*decode the node  $i$*
- pair< mpz\_class, mpz\_class > [decode\\_interval](#) (int i, int j, int l, mpz\_class tN, int Sj)  
*decodes the interval  $[i, j]$  with interval index  $I$ .*

### Private Attributes

- vector< int > [a](#)  
*the degree sequence of the graph.*
- int [n](#)  
*the number of vertices, which is  $a.size()$*
- int [logn2](#)  
*the integer part of  $\log^2 n$*
- vector< vector< int > > [x](#)  
*the forward adjacency list of the decoded graph*
- vector< int > [beta](#)  
*the sequence  $\vec{\beta}$ , where after decoding vertex  $i$ , for  $i \leq v \leq n$  we have  $\beta_v = d_v(i)$ .*
- [reverse\\_fenwick\\_tree](#) [U](#)  
*a Fenwick tree initialized with the degree sequence  $a$ , and after decoding vertex  $i$ , for  $i \leq v$ , we have  $U_v = \sum_{k=v}^{n-1} d_k(i)$ .*
- vector< int > [tS](#)  
*the  $\tilde{S}$  vector, which stores the partial sums for the midpoints of intervals with length more than  $\log^2 n$ .*

### 3.7.1 Detailed Description

Decodes a simple unmarked graph.

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

### 3.7.2 Constructor & Destructor Documentation

#### 3.7.2.1 graph\_decoder()

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

constructor given the degree sequence

```
113 {
114     a = a_;
115     n = a.size();
116     double logn = log(n);
117     logn2 = int(logn * logn);
118     x.resize(n);
119     beta = a;
120     U = reverse_fenwick_tree(a);
121 }
```

### 3.7.3 Member Function Documentation

#### 3.7.3.1 decode()

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

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

```
125 {
126     tS = tS_;
127     //mpz_class prod_a_factorial = 1; // \prod_{i=1}^n a_i!
128     //for (int i=0; i<a.size();i++)
129     // prod_a_factorial *= compute_product(a[i], a[i], 1);
130
131     mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=0}^{n-1} a_i!
132     mpz_class tN = f * prod_a_factorial;
133     decode_interval(0,n-1,1,tN,0);
134     return graph(x, a);
135 }
```

#### 3.7.3.2 decode\_interval()

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

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

## Parameters

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

## Returns

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

```

190 {
191     //cerr << " decode interval " << i << " " << j << " tN " << tN << endl;
192     if (i == j)
193         return decode_node(i, tN);
194
195     // sweeping for zero nodes
196
197     int t; // place to break
198     int St; // S_{t+1}
199     if ((j-i) > logn2){
200         //cerr << " long interval I = " << I << endl;
201         t = (i+j) / 2; // break at middle, since we have \tilde{S}
202         St = tS[I]; // looking at the \f$\tilde{S}$\f$ vector
203     }else{
204         //cerr << " short interval " << endl;
205         t = i;
206         St = U.sum(i) - 2 * beta[i];
207     }
208
209     //cerr << " decode interval " << i << " " << j << " t " << t << " St " << St << " Sj " << Sj << endl;
210     mpz_class rtj; // \f$S_{t+1, j}\f$
211     mpz_class tNit; // \f$\tilde{N}_{i,t}\f$ for the left decoder
212     mpz_class tNtj; // \f$\tilde{N}_{t+1, j}\f$ for the right decoder
213     mpz_class Nit; // the true  $N_{i,t}$  returned by the left decoder
214     mpz_class lit; // the true  $l_{i,t}$  returned by the left decoder
215     mpz_class Ntj; // the true  $N_{t+1, j}$  returned by the right decoder
216     mpz_class ltj; // the true  $l_{t+1, j}$  returned by the right decoder
217     mpz_class Nij; // the true  $N_{i,j}$  to return
218     mpz_class lij; // the true  $l_{i,j}$  to return
219
220     pair<mpz_class, mpz_class> ans; // returned by subintervals
221
222
223     rtj = compute_product(St - 1, (St - Sj)/2, 2);
224     //cerr << " interval " << i << " " << j << " t " << t << " St " << St << " rtj " << rtj << endl;
225     tNit = tN / rtj;
226
227     // calling the left decoder
228     ans = decode_interval(i, t, 2*I, tNit, St);
229     Nit = ans.first;
230     lit = ans.second;
231
232     // reducing the contribution of the left decoder to prepare for the right decoder
233     tNtj = (tN - Nit * rtj) / lit;
234
235     // calling the right decoder
236     ans = decode_interval(t+1, j, 2*I + 1, tNtj, Sj);
237     Ntj = ans.first;
238     ltj = ans.second;
239
240     // preparing Nij and lij to return
241     Nij = Nit * rtj + lit * Ntj;
242     lij = lit * ltj;
243     return pair<mpz_class, mpz_class> (Nij, lij);
244 }

```

## 3.7.3.3 decode\_node()

```

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

```

decode the node i

## Parameters

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

## Returns

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

```

138 {
139     //cerr << " decode node " << i << " tN " << tN << endl;
140     //cerr << " beta[i] " << beta[i] << endl;
141     //cerr << " beta " << endl;
142     //for (int k = i; k < n; k++)
143     // cerr << k << " " << beta[k] << endl;
144     //cerr << " U " << endl;
145     //for (int k=i; k<n; k++)
146     // cerr << k << " " << U.sum(k) << endl;
147
148     if (beta[i] == 0)
149         return pair<mpz_class, mpz_class> (0,1);
150
151     mpz_class li = 1; // l_i(G)
152     mpz_class Ni = 0; // N_{i,i}(G)
153     int f, g; // endpoints for the binary search
154     int t; // midpoint for the binary search
155     mpz_class zik, lik;
156     for (int k=0; k<beta[i]; k++){
157         if (k==0)
158             f = i+1;
159         else
160             f = x[i][k-1]+1;
161         g = n-1;
162         while(g > f){
163             //cerr << " f , g " << f << " " << g << endl;
164             t = (f+g)/2;
165             // binary search:
166             if(compute_product(U.sum(t+1), beta[i] - k, 1) <= tN)
167                 g = t;
168             else
169                 f = t+1;
170         }
171         x[i].push_back(f);
172         zik = compute_product(U.sum(x[i][k]+1), beta[i] - k, 1);
173         Ni += li * zik;
174         lik = (beta[i] - k) * beta[x[i][k]];
175         li *= lik;
176         tN -= zik;
177         tN /= lik;
178         U.add(x[i][k], -1);
179         beta[x[i][k]] --;
180     }
181     //cerr << " decoded for " << i << " x: " << endl;
182     //for (int j=0; j<x[i].size(); j++)
183     // cerr << x[i][j] << " " ;
184     //cerr << endl;
185     return pair<mpz_class, mpz_class> (Ni, li);
186 }

```

## 3.7.4 Member Data Documentation

## 3.7.4.1 a

vector<int> graph\_decoder::a [private]

the degree sequence of the graph.

**3.7.4.2 beta**

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

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

**3.7.4.3 logn2**

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

the integer part of  $\log^2 n$

**3.7.4.4 n**

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

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

**3.7.4.5 tS**

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

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

**3.7.4.6 U**

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

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

**3.7.4.7 x**

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

the forward adjacency list of the decoded graph

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

- [simple\\_graph\\_compression.h](#)
- [simple\\_graph\\_compression.cpp](#)



## 3.8 graph\_encoder Class Reference

Encodes a simple unmarked graph.

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

### Public Member Functions

- [graph\\_encoder](#) (const [graph](#) &Gin)  
*constructor*
- void [init](#) ()  
*initializes beta and U, logn2, and resizes Stilde.*
- pair< mpz\_class, mpz\_class > [compute\\_N](#) (int i, int j, int l)  
*computes  $N_{i,j}(G)$*
- pair< mpz\_class, vector< int > > [encode](#) ()  
*Encodes the graph and returns N together with Stilde.*

### Private Attributes

- const [graph](#) & [G](#)  
*the simple unmarked graph which is going to be encoded*
- vector< int > [beta](#)  
*When compute\_N is called for  $i \leq j$ , for  $i \leq v \leq n$ , we have  $\beta_v = d_v(i)$ .*
- [reverse\\_fenwick\\_tree](#) [U](#)  
*a Fenwick tree which encodes the forward degrees to the right. When compute\_N is called for  $i \leq j$ , for  $i \leq v$ , we have  $U_v = \sum_{k=v}^n d_k(i)$ .*
- vector< int > [Stilde](#)  
*Summation of forward degrees at  $n / \log^2 n$  many points.*
- int [logn2](#)  
*the integer part of  $\log^2 n$  where  $n$  is the number of vertices in  $G$ .*

#### 3.8.1 Detailed Description

Encodes a simple unmarked graph.

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

#### 3.8.2 Constructor & Destructor Documentation

### 3.8.2.1 graph\_encoder()

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

constructor

```
22                                     : G(Gin)
23 {
24     init();
25 }
```

## 3.8.3 Member Function Documentation

### 3.8.3.1 compute\_N()

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

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

Parameters

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

Returns

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

```
33 {
34     //cerr << " i, j " << i << " , " << j << endl;
35     if (i==j){
36         mpz_class zi, li, zik, lik;
37         zi = 0;
38         li = 1;
39         vector<int> x = G.get_forward_list(i); // the forward adjacency list of vertex i
40         if (beta[i] != x.size())
41             cerr << " DANGER! beta[i] is not the same as x.size()!!" << endl;
42         for (int k=0;k<x.size();k++){
43             zik = compute_product(U.sum(1+x[k]), beta[i] - k, 1); // we are zero based
44             here, so instead of -k + 1, we have -k
45             zi += li * zik;
46             lik = (beta[i] - k) * beta[x[k]]; // we are zero based here, so instead of -k + 1, we have -k
47             li *= lik;
48             beta[x[k]] --;
49             U.add(x[k], -1);
50         }
51         //cerr << " returning ( " << i << " , " << j << " ) N " << zi << " 1 " << li << endl;
52         return pair<mpz_class, mpz_class>(zi, li);
53     }else{
54         int t = (i+j) / 2;
55         mpz_class Nit, lit; // \f$N_{i,t}\f$ and \f$1_{[i:t]}\f$
56         mpz_class Ntj, ltj; // \f$N_{t+1,j}\f$ and \f$1_{[t+1:j]}\f$
57         mpz_class Nij, lij; // \f$N_{i,j}\f$ and \f$1_{[i:j]}\f$
58         int St, Sj; // \f$S_{t+1}\f$ and \f$S_{j+1}\f$
```

```

58     mpz_class rtj; // \f$__{[t+1:j]}\f$
59
60     pair<mpz_class, mpz_class> return_left = compute_N(i,t, 2*I); // calling for the interval i,t
61     Nit = return_left.first;
62     lit = return_left.second;
63     St = U.sum(t+1);
64
65     if (j - i > logn2){// we should save the midpoint sum St
66         //cerr << " i " << i << " j " << j << " I " << I << " storing St " << St << endl;
67         if (I >= Stilde.size()){
68             cerr << " BAD: I out of range I " << I << " Stilde.size() " << Stilde.size() << endl;
69             cerr << " i " << i << " j " << j << " logn2 " << logn2 << endl;
70         }
71         Stilde[I] = St;
72     }
73
74     pair<mpz_class, mpz_class> return_right = compute_N(t+1, j, 2*I + 1); // calling for the
75     interval t+1, j
76     Ntj = return_right.first;
77     ltj = return_right.second;
78     Sj = U.sum(j+1);
79     rtj = compute_product(St-1, (St - Sj)/2, 2);
80     //cerr << "( " << i << " , " << j << " ) St " << St << " Sj " << Sj << " rtj " << rtj << endl;
81     Nij = Nit * rtj + lit * Ntj ;
82     lij = lit * ltj;
83     //cerr << " returning ( " << i << " , " << j << " ) N " << Nij << " l " << lij << endl;
84     return pair<mpz_class, mpz_class> (Nij, lij);
85 }

```

### 3.8.3.2 encode()

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

Encodes the graph and returns N together with Stilde.

#### Returns

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

```

87     {
88     pair<mpz_class, mpz_class> ans = compute_N(0,G.nu_vertices()-1,1);
89     vector<int> a = G.get_degree_sequence(); // the graph degree sequence
90     //mpz_class prod_a_factorial = 1; // \prod_{i=1}^n a_i!
91     //for (int i=0; i<a.size();i++)
92     // prod_a_factorial *= compute_product(a[i], a[i], 1);
93     mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=1}^n a_i!
94     // we need the ceiling of the ratio of ans.first and prod_a_factorial
95     bool ceil = false; // if true, we will add one to the integer division
96     if (ans.first % prod_a_factorial != 0)
97         ceil = true;
98     ans.first /= prod_a_factorial;
99     if (ceil)
100         ans.first ++;
101     return pair<mpz_class, vector<int> > (ans.first, Stilde);
102 }

```

### 3.8.3.3 init()

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

initializes beta and U, logn2, and resizes Stilde.

```
10 {
11     // initializing the beta sequence
12     beta = G.get_degree_sequence();
13
14     //beta.resize(G.nu_vertices());
15     //for (int v=0;v<G.nu_vertices();v++)
16     //    beta[v] = G.get_degree(v);
17
18     // initializing the Fenwick Tree
19     U = reverse_fenwick_tree(beta);
20
21     // initializing logn2
22     int n = G.nu_vertices();
23     double logn = log(n);
24     logn2 = int(logn * logn);
25
26     //initializing the partial sum vector Stilde
27     Stilde.resize(4 * n / logn2); // TO CHECK,
28     // 2018-10-18_self-compression_Stilde-size-required-2nlogn2.pdf
29     Stilde[0] = 0;
30 }
```

## 3.8.4 Member Data Documentation

### 3.8.4.1 beta

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

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

### 3.8.4.2 G

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

the simple unmarked graph which is going to be encoded

### 3.8.4.3 logn2

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

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

## 3.8.4.4 Stilde

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

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

## 3.8.4.5 U

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

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

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

- [simple\\_graph\\_compression.h](#)
- [simple\\_graph\\_compression.cpp](#)

## 3.9 graph\_message Class Reference

this class takes care of message passing on marked graphs.

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

## Public Member Functions

- [graph\\_message](#) (const [marked\\_graph](#) &graph, int depth, int max\_degree)  
*constructor, given reference to a graph*
- void [update\\_messages](#) ()  
*performs the message passing algorithm and updates the messages array accordingly*
- void [update\\_message\\_dictionary](#) ()  
*update message\_dict and message\_list*

## Public Attributes

- const [marked\\_graph](#) & G  
*reference to the marked graph for which we do message passing*
- int h  
*the depth up to which we do message passing (the type of edges go through depth h-1)*
- int [Delta](#)  
*the maximum degree threshold*
- vector< vector< vector< vector< int > > > > [messages](#)  
*messages[v][i][t] is the message at time t from vertex v towards its ith neighbor (in the order given by adj\_list of vertex i in graph G). Messages will be useful to find edge types*
- map< vector< int >, int > [message\\_dict](#)  
*the message dictionary (at depth t=h-1), which maps each message to its corresponding index in the dictionary*
- vector< vector< int > > [message\\_list](#)  
*the list of messages present in the graph, stored in an order consistent with message\_dict, i.e. for a message m, if message\_dict[m] = i, then message\_list[i] = m.*

### 3.9.1 Detailed Description

this class takes care of message passing on marked graphs.

This graph has a reference to a [marked\\_graph](#) object for which we perform message passing to find edge types. The edge types are discovered up to depth  $h-1$ , and with degree parameter  $\Delta$ , where  $h$  and  $\Delta$  are member objects.

#### *Sample Usage*

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

### 3.9.2 Constructor & Destructor Documentation

#### 3.9.2.1 graph\_message()

```
graph_message::graph_message (
    const marked\_graph & graph,
    int depth,
    int max_degree ) [inline]
```

constructor, given reference to a graph

```
34                                     : G(graph) {
35     h = depth;
36     Delta = max_degree;
37 }
```

### 3.9.3 Member Function Documentation

#### 3.9.3.1 update\_message\_dictionary()

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

update message\_dict and message\_list

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

```

120 {
121     vector<int> message;
122     for (int v=0;v<G.nu_vertices;v++){
123         for (int i=0;i<G.adj_list[v].size();i++){
124             message = messages[v][i][h-1];
125             if(message_dict.find(message) == message_dict.end()){
126                 // the message does not exist in the dictionary, hence add it
127                 message_dict[message] = message_list.size(); // so that it points to the
128                 last element in message_list which is going to be added in the next line, this assures that if message_dict[m]
129                 = i, then message_list[i] = m
130                 message_list.push_back(message); // add the message to the list
131             }
132         }
133     }
134     // we want all the * messages to be together so that later we can easily distinguish between * messages
135     // and normal messages.
136     // in order to do this, we simply sort the message list
137     sort (message_list.begin(), message_list.end());
138     // but, since we want the * messages which start by -1 to go to the end of the list, after sorting, we
139     // reverse the vector as well
140     reverse(message_list.begin(), message_list.end());
141     // then, we update message_dict accordingly
142     // at the same time, we count the number of non * messages, i.e. L
143     //L = 0;
144     for (int i=0;i<message_list.size();i++){
145         message_dict[message_list[i]] = i;
146         //if (message_list[i][0] != -1)
147         //L++;
148     }
149 }

```

### 3.9.3.2 update\_messages()

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

performs the message passing algorithm and updates the messages array accordingly

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

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

```

19 {
20     int nu_vertices = G.nu_vertices;
21     messages.resize(nu_vertices);
22     // initialize the messages
23     for (int v=0;v<nu_vertices;v++){
24         messages[v].resize(G.adj_list[v].size());
25         for (int i=0;i<G.adj_list[v].size();i++){
26             // the message from v towards the ith neighbor (lets call is w) at time 0 has a mark component which
27             // is \xi(v,w) and a subtree component which is a single root with mark \tau(v). This is encoded as a message
28             // vector with size 3 of the form (\tau(v), 0, \xi(v,w)) where the last 0 indicates that there is no offspring.
29             messages[v][i].resize(h);
30             // initialize messages to be empty
31             for (int t=0;t<h;t++)

```

```

32     messages[v][i][t].resize(0);
33
34     vector<int> m;
35     m.push_back(G.ver_mark[v]);
36     m.push_back(0);
37     m.push_back(G.adj_list[v][i].second.first);
38     messages[v][i][0] = m; // the message at time 0
39 }
40 }
41
42 // updating messages
43 for (int t=1;t<h;t++){
44     for (int v=0;v<nu_vertices;v++){
45         if (G.adj_list[v].size() <= Delta){
46             // the degree of v is no more than Delta
47             // do the standard message passing by aggregating messages from neighbors
48             // stacking all the messages from neighbors of v towards v
49             vector<pair<vector<int>, int> > neighbor_messages; // the first component is the message and the
second is the name of the neighbor
50             // the second component is stored so that after sorting, we know the owner of the message
51
52             // 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
53             for (int i=0;i<G.adj_list[v].size();i++){
54                 int w = G.adj_list[v][i].first; // what is the name of the neighbor I am looking at now,
which is the ith neighbor of vertex v
55                 int my_location = G.adj_location[w].at(v); // where is the place of node v among the
list of neighbors of the ith neighbor of v
56                 vector<int> previous_message = messages[w][my_location][t-1]; // the message sent from
this neighbor towards v at time t-1
57                 previous_message.push_back(G.adj_list[v][i].second.first); // adding the mark towards v
to the list
58                 neighbor_messages.push_back(pair<vector<int>, int> (previous_message, w));
59             }
60
61             sort(neighbor_messages.begin(), neighbor_messages.end(), pair_compare);
62             for (int i=0;i<G.adj_list[v].size();i++){
63                 // let w be the current ith neighbor of v
64                 int w = G.adj_list[v][i].first;
65                 // first, start with the mark of v and the number of offsprings in the subgraph component of the
message
66                 messages[v][i][t].push_back(G.ver_mark[v]); // mark of v
67                 messages[v][i][t].push_back(G.adj_list[v].size()-1); // the number of offsprings
in the subgraph component of the message
68                 // stacking messages from all neighbors of v except for w towards v at time t-1
69                 for (int j=0;j<G.adj_list[v].size();j++){
70                     if (neighbor_messages[j].second != w){
71                         if (neighbor_messages[j].first[0] == -1){
72                             // this means that one of the messages that should be aggregated is * typed, therefore the
outgoing messages should also be * typed
73                             // 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
74                             // 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
75                             messages[v][i][t].resize(0);
76                             messages[v][i][t].push_back(-1);
77                             break; // the message is decided, we do not need to go over any of the other neighbor
messages, hence break
78                         }
79                         // this message should be added to the list of messages
80                         messages[v][i][t].insert(messages[v][i][t].end(), neighbor_messages[j].first.
begin(), neighbor_messages[j].first.end());
81                     }
82                 }
83                 // 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.
84                 // finally, the mark of the edge between v and w towards v, \xi(w,v), should be added to this
list
85                 messages[v][i][t].push_back(G.adj_list[v][i].second.first);
86             }
87         }else{
88             // if the degree of v is bigger than Delta, the message towards all neighbors is of the form *
89             // 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
90             for (int i=0;i<G.adj_list[v].size();i++){
91                 messages[v][i][t].resize(2);
92                 messages[v][i][t][0] = -1;
93                 messages[v][i][t][1] = G.adj_list[v][i].second.first;
94             }
95         }
96     }
97 }
98
99 // 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)
100 for (int v=0;v<nu_vertices;v++){

```



```

101     for (int i=0;i<G.adj_list[v].size();i++){
102         if (messages[v][i][h-1][0] == -1){
103             // it is of the form *
104             int w = G.adj_list[v][i].first; // the other endpoint of the edge
105             int my_location = G.adj_location[w].at(v); // so that adj_list[w][my_location].first =
106             v
107             messages[w][my_location][h-1].resize(2);
108             messages[w][my_location][h-1][0] = -1;
109             messages[w][my_location][h-1][1] = G.adj_list[v][i].second.second; // the mark
110             towards w
111         }
112     }
113     update_message_dictionary(); // update the variables message_dict and
114     message_list
115 }

```

### 3.9.4 Member Data Documentation

#### 3.9.4.1 Delta

```
int graph_message::Delta
```

the maximum degree threshold

#### 3.9.4.2 G

```
const marked_graph& graph_message::G
```

reference to the marked graph for which we do message passing

#### 3.9.4.3 h

```
int graph_message::h
```

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

#### 3.9.4.4 message\_dict

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

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

### 3.9.4.5 message\_list

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

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

### 3.9.4.6 messages

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

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

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

- [graph\\_message.h](#)
- [graph\\_message.cpp](#)

## 3.10 marked\_graph Class Reference

simple marked graph

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

### Public Member Functions

- [marked\\_graph](#) ()  
*default constructor*
- [marked\\_graph](#) (int n, vector< pair< pair< int, int >, pair< int, int > > > edges, vector< int > vertex\_marks)  
*constructs a marked graph based on edges lists and vertex marks.*

### Public Attributes

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

### 3.10.1 Detailed Description

simple marked graph

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

### 3.10.2 Constructor & Destructor Documentation

#### 3.10.2.1 marked\_graph() [1/2]

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

default constructor

```
25     {
26         nu_vertices = 0;
27     }
```

#### 3.10.2.2 marked\_graph() [2/2]

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

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

Parameters

<i>n</i>	the number of vertices in the graph
<i>edges</i>	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 $j$
<i>vertex_marks</i>	is a vector of size $n$ , where <code>vertex_marks[i]</code> is the mark of vertex $i$

```
4 {
5     nu_vertices = n;
6     adj_list.resize(n);
7     adj_location.resize(n);
8     for (int k=0; k<edges.size(); k++){
9         // (i,j) are endpoints if the edge
10        // (x,y) are marks, x towards i and y towards j
11        int i = edges[k].first.first;
12        int j = edges[k].first.second;
13        int x = edges[k].second.first;
14        int y = edges[k].second.second;
15        if (i < 0 || i >= n || j < 0 || j >= n || i == j)
16            cerr << " ERROR: graph::graph(n, edges) received an invalid pair of edges with n = " << n << " : (" <
            < i << " , " << j << " )" << endl;
```

```

17     adj_list[i].push_back(pair<int, pair<int, int> > (j, pair<int, int> (x,y)));
18     adj_location[i][j] = adj_list[i].size() - 1;
19     adj_list[j].push_back(pair<int, pair<int, int> > (i, pair<int, int> (y,x)));
20     adj_location[j][i] = adj_list[j].size() - 1;
21 }
22 ver_mark = vertex_marks;
23 }

```

### 3.10.3 Member Data Documentation

#### 3.10.3.1 adj\_list

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

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

#### 3.10.3.2 adj\_location

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

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

#### 3.10.3.3 nu\_vertices

```
int marked_graph::nu_vertices
```

number of vertices in the graph

#### 3.10.3.4 ver\_mark

```
vector<int> marked_graph::ver_mark
```

ver\_mark[i] is the mark of vertex i

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

- [marked\\_graph.h](#)
- [marked\\_graph.cpp](#)

## 3.11 reverse\_fenwick\_tree Class Reference

similar to the [fenwick\\_tree](#) class, but instead of prefix sums, this class computes suffix sums.

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

### Public Member Functions

- [reverse\\_fenwick\\_tree](#) ()  
*default constructor*
- [reverse\\_fenwick\\_tree](#) (vector< int >)  
*constructor which receives values and initializes*
- void [add](#) (int k, int val)
- int [size](#) ()  
*the number of elements in the original array*
- int [sum](#) (int k)

### Private Attributes

- [fenwick\\_tree](#) FT  
*member of type [fenwick\\_tree](#), which saves the partial sums for the reversed array.*

#### 3.11.1 Detailed Description

similar to the [fenwick\\_tree](#) class, but instead of prefix sums, this class computes suffix sums.

#### 3.11.2 Constructor & Destructor Documentation

##### 3.11.2.1 reverse\_fenwick\_tree() [1/2]

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

default constructor

```
58 {}
```

### 3.11.2.2 reverse\_fenwick\_tree() [2/2]

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

constructor which receives values and initializes

```
47 {
48     reverse(vals.begin(),vals.end()); // reverse the array and then use the previously defined fenwick_tree
        class
49     FT = fenwick_tree(vals);
50 }
```

## 3.11.3 Member Function Documentation

### 3.11.3.1 add()

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

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

#### Parameters

<i>k</i>	the index to be modified, this is zero based
<i>val</i>	the value to be added to the above index

```
53 {
54     FT.add(FT.size() - 1 - k, val);
55 }
```

### 3.11.3.2 size()

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

the number of elements in the original array

```
70     {
71         return FT.size();
72     }
```

### 3.11.3.3 sum()

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

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

## Parameters

<i>k</i>	the index from which (including) the sum is computed
----------	--

```
59 {  
60     if (k >= size())  
61         return 0;  
62     return FT.sum(FT.size() - 1 - k);  
63 }
```

### 3.11.4 Member Data Documentation

#### 3.11.4.1 FT

`fenwick_tree` reverse\_fenwick\_tree::FT [private]

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

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

- [fenwick.h](#)
- [fenwick.cpp](#)





## Chapter 4

# File Documentation

### 4.1 bipartite\_graph.cpp File Reference

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

#### Functions

- ostream & [operator<<](#) (ostream &o, const [b\\_graph](#) &G)
- bool [operator==](#) (const [b\\_graph](#) &G1, const [b\\_graph](#) &G2)
- bool [operator!=](#) (const [b\\_graph](#) &G1, const [b\\_graph](#) &G2)

#### 4.1.1 Function Documentation

##### 4.1.1.1 [operator!=\(\)](#)

```
bool operator!= (
    const b\_graph & G1,
    const b\_graph & G2 )
```

```
121 {
122     return !(G1 == G2);
123 }
```

#### 4.1.1.2 operator<<()

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

85 {
86     int n = G.nu_left_vertices();
87     vector<int> list;
88     for (int i=0; i<n; i++){
89         list = G.get_adj_list(i);
90         o << i << " -> ";
91         for (int j=0; j<list.size(); j++){
92             o << list[j];
93             if (j < list.size()-1)
94                 o << ", ";
95         }
96         o << endl;
97     }
98     return o;
99 }
```

#### 4.1.1.3 operator==( )

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

102 {
103     int n1 = G1.nu_left_vertices();
104     int n2 = G2.nu_left_vertices();
105
106     int np1 = G1.nu_right_vertices();
107     int np2 = G2.nu_right_vertices();
108     if (n1!= n2 or np1 != np2)
109         return false;
110     vector<int> list1, list2;
111     for (int v=0; v<n1; v++){
112         list1 = G1.get_adj_list(v);
113         list2 = G2.get_adj_list(v);
114         if (list1 != list2)
115             return false;
116     }
117     return true;
118 }
```

## 4.2 bipartite\_graph.h File Reference

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

### Classes

- class [b\\_graph](#)  
*simple unmarked bipartite graph*

## 4.3 bipartite\_graph\_compression.cpp File Reference

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

## 4.4 bipartite\_graph\_compression.h File Reference

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

### Classes

- class [b\\_graph\\_encoder](#)  
*Encodes a simple unmarked bipartite graph.*
- class [b\\_graph\\_decoder](#)  
*Decodes a simple unmarked bipartite graph.*

## 4.5 compression\_helper.cpp File Reference

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

### Functions

- `mpz_class` [compute\\_product](#) (`mpz_class` N, `mpz_class` k, `int` s)  
*This function computes the product of consecutive integers separated by a given iteration.*
- `mpz_class` [binomial](#) (`const` `mpz_class` n, `const` `mpz_class` m)  
*computes the binomial coefficient  $n$  choose  $m = n! / m! (n-m)!$*
- `mpz_class` [prod\\_factorial](#) (`const` `vector`< `int` > &a, `int` i, `int` j)  
*computes the product of factorials in a vector given a range*

### 4.5.1 Function Documentation

#### 4.5.1.1 binomial()

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

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

**Parameters**

$n$	integer
$m$	integer

**Returns**

the binomial coefficient  $n! / m! (n-m)!$ . If  $n \leq 0$ , or  $m > n$ , or  $m \leq 0$ , returns 0

```

29 {
30     if (n <= 0 or m > n or m <= 0)
31         return 0;
32     return compute_product(n, m, 1) / compute_product(m, m, 1);
33 }
```

**4.5.1.2 compute\_product()**

```

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

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

**Parameters**

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

**Returns**

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

```

4
5 //cerr << " compute_product  N " << N << " k " << k << " s " << s << endl;
6
7 if (k==1)
8     return N;
9 if (k == 0) // TO CHECK because there are no terms to compute product
10     return 1;
11
12 if (k < 0){
13     cerr << " WARNING: compute_product called for k < 0, returning 1, N " << N << " k " << k << " s " << s
14     << endl;
15     return 1;
16 }
17 if (N - (k-1) * s <= 0) // the terms go negative
18     return 0;
19
20 // we do this by dividing the terms into two parts
21 mpz_class m = k / 2; // the middle point
22 mpz_class left, right; // each of the half products
23 left = compute_product(N, m, s);
24 right = compute_product(N-m * s, k-m, s);
25 return left * right;
26 }
```

## 4.5.1.3 prod\_factorial()

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

computes the product of factorials in a vector given a range

## Parameters

<i>a</i>	vector of integers
<i>i,j</i>	endpoints of the interval

## Returns

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

```
37 {
38     if (i == j){
39         return compute_product(a[i], a[i], 1);
40     }else{
41         int k = (i+j)/2;
42         mpz_class x = prod_factorial(a, i, k);
43         mpz_class y = prod_factorial(a, k+1, j);
44         return x * y;
45     }
46 }
```

## 4.6 compression\_helper.h File Reference

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

## Functions

- `mpz_class compute_product` (`mpz_class N`, `mpz_class k`, `int s`)  
*This function computes the product of consecutive integers separated by a given iteration.*
- `mpz_class binomial` (`const mpz_class n`, `const mpz_class m`)  
*computes the binomial coefficient  $n$  choose  $m = n! / m! (n-m)!$*
- `mpz_class prod_factorial` (`const vector< int > &a`, `int i`, `int j`)  
*computes the product of factorials in a vector given a range*

## 4.6.1 Function Documentation

## 4.6.1.1 binomial()

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

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

**Parameters**

$n$	integer
$m$	integer

**Returns**

the binomial coefficient  $n! / m! (n-m)!$ . If  $n \leq 0$ , or  $m > n$ , or  $m \leq 0$ , returns 0

```

29 {
30     if (n <= 0 or m > n or m <= 0)
31         return 0;
32     return compute_product(n, m, 1) / compute_product(m, m, 1);
33 }
```

**4.6.1.2 compute\_product()**

```

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

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

**Parameters**

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

**Returns**

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

```

4
5 //cerr << " compute_product  N " << N << " k " << k << " s " << s << endl;
6
7 if (k==1)
8     return N;
9 if (k == 0) // TO CHECK because there are no terms to compute product
10     return 1;
11
12 if (k < 0){
13     cerr << " WARNING: compute_product called for k < 0, returning 1, N " << N << " k " << k << " s " << s
14     << endl;
15     return 1;
16 }
17 if (N - (k-1) * s <= 0) // the terms go negative
18     return 0;
19
20 // we do this by dividing the terms into two parts
21 mpz_class m = k / 2; // the middle point
22 mpz_class left, right; // each of the half products
23 left = compute_product(N, m, s);
24 right = compute_product(N-m * s, k-m, s);
25 return left * right;
26 }
```

## 4.6.1.3 prod\_factorial()

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

computes the product of factorials in a vector given a range

## Parameters

<i>a</i>	vector of integers
<i>i,j</i>	endpoints of the interval

## Returns

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

```
37 {
38     if (i == j){
39         return compute_product(a[i], a[i], 1);
40     }else{
41         int k = (i+j)/2;
42         mpz_class x = prod_factorial(a, i, k);
43         mpz_class y = prod_factorial(a, k+1, j);
44         return x * y;
45     }
46 }
```

## 4.7 fenwick.cpp File Reference

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

## 4.8 fenwick.h File Reference

```
#include <vector>
```

## Classes

- class [fenwick\\_tree](#)  
*Fenwick tree class.*
- class [reverse\\_fenwick\\_tree](#)  
*similar to the [fenwick\\_tree](#) class, but instead of prefix sums, this class computes suffix sums.*

## 4.9 graph\_message.cpp File Reference

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

## Functions

- bool [pair\\_compare](#) (const pair< vector< int >, int > &a, const pair< vector< int >, int > &b)  
*used for sorting messages*

### 4.9.1 Function Documentation

#### 4.9.1.1 pair\_compare()

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

used for sorting messages

```
153                                     {
154     return a.first < b.first;
155 }
```

## 4.10 graph\_message.h File Reference

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

## Classes

- class [graph\\_message](#)  
*this class takes care of message passing on marked graphs.*
- class [colored\\_graph](#)  
*this class defines a colored graph, which is obtained from a simple marked graph and the color of edges come from the type of edges*

## Functions

- bool [pair\\_compare](#) (const pair< vector< int >, int > &, const pair< vector< int >, int > &)  
*used for sorting messages*

### 4.10.1 Function Documentation



## 4.10.1.1 pair\_compare()

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

used for sorting messages

```
153                                     {
154     return a.first < b.first;
155 }
```

## 4.11 marked\_graph.cpp File Reference

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

## Functions

- `istream & operator>>` (`istream &inp`, `marked_graph &G`)  
inputs a `marked_graph`

## 4.11.1 Function Documentation

## 4.11.1.1 operator&gt;&gt;()

```
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 towards 1

```
26 {
27     int nu_vertices;
28     inp >> nu_vertices;
29
30     vector<int> ver_marks;
31     ver_marks.resize(nu_vertices);
32     for (int i=0;i<nu_vertices;i++)
33         inp >> ver_marks[i];
34
35     int nu_edges;
36     inp >> nu_edges;
37     vector<pair< pair<int, int> , pair<int, int> > > edges;
38     edges.resize(nu_edges);
39     for (int i=0;i<nu_edges;i++)
40         inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >> edges[i].second.second
41         ;
42     G = marked_graph(nu_vertices, edges, ver_marks);
43
44     return inp;
45 }
```

## 4.12 marked\_graph.h File Reference

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

### Classes

- class [marked\\_graph](#)  
*simple marked graph*

### Functions

- [istream & operator>>](#) (istream &inp, [marked\\_graph](#) &G)  
*inputs a [marked\\_graph](#)*

#### 4.12.1 Function Documentation

##### 4.12.1.1 [operator>>\(\)](#)

```
istream& operator>> (
    istream & inp,
    marked\_graph & G )
```

inputs a [marked\\_graph](#)

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

```
26 {
27     int nu_vertices;
28     inp >> nu_vertices;
29
30     vector<int> ver_marks;
31     ver_marks.resize(nu_vertices);
32     for (int i=0;i<nu_vertices;i++)
33         inp >> ver_marks[i];
34
35     int nu_edges;
36     inp >> nu_edges;
37     vector<pair< pair<int, int> , pair<int, int> > > edges;
38     edges.resize(nu_edges);
39     for (int i=0;i<nu_edges;i++)
40         inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >> edges[i].second.second
41         ;
42     G = marked\_graph(nu_vertices, edges, ver_marks);
43
44     return inp;
45 }
```

## 4.13 simple\_graph.cpp File Reference

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

### Functions

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

#### 4.13.1 Function Documentation

##### 4.13.1.1 operator!=()

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

```
77 {
78     return !(G1 == G2);
79 }
```

##### 4.13.1.2 operator<<()

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

```
44 {
45     int n = G.nu_vertices();
46     vector<int> list;
47     for (int i=0; i<n; i++){
48         list = G.get_forward_list(i);
49         o << i << " -> ";
50         for (int j=0; j<list.size(); j++){
51             o << list[j];
52             if (j < list.size()-1)
53                 o << ", ";
54         }
55         o << endl;
56     }
57     return o;
58 }
```

#### 4.13.1.3 operator==( )

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

61 {
62     int n1 = G1.nu_vertices();
63     int n2 = G2.nu_vertices();
64     if (n1!= n2)
65         return false;
66     vector<int> list1, list2;
67     for (int v=0; v<n1; v++){
68         list1 = G1.get_forward_list(v);
69         list2 = G2.get_forward_list(v);
70         if (list1 != list2)
71             return false;
72     }
73     return true;
74 }
```

### 4.14 simple\_graph.h File Reference

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

#### Classes

- class [graph](#)  
*simple unmarked graph*

### 4.15 simple\_graph\_compression.cpp File Reference

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

### 4.16 simple\_graph\_compression.h File Reference

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

#### Classes

- class [graph\\_encoder](#)  
*Encodes a simple unmarked graph.*
- class [graph\\_decoder](#)  
*Decodes a simple unmarked graph.*

## 4.17 test.cpp File Reference

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

### Functions

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

#### 4.17.1 Function Documentation

##### 4.17.1.1 main()

```
int main ( )

24     {
25     //marked_graph G;
26     //ifstream inp("star_graph.txt");
27     //inp >> G;
28     //graph_message M(G, 10, 2);
29     //M.update_messages();
30     vector<int> a = {1,1,2}; // left degree sequence
31     vector<int> b = {2,2}; // right degree sequence
32
33     b_graph G({{0},{1},{0,1}}); // defining the graph
34
35     b_graph_encoder E(a,b); // constructing the encoder object
36     mpz_class f = E.encode(G);
37
38     b_graph_decoder D(a, b);
39     b_graph Ghat = D.decode(f);
40
41     if (Ghat == G)
42         cout << " successfully decoded the graph! " << endl;
43
44
45     return 0;
46     // vector<vector<int> > list = {{}, {}, {} };
47
48     // b_graph G({{0},{1},{0,1}});
49     // // cout << G << endl;
50     // // cout << G.nu_left_vertices() << endl;
51     // // cout << G.nu_right_vertices() << endl;
52     // // cout << G.get_left_degree_sequence() << endl;
53     // // cout << G.get_right_degree_sequence() << endl;
54     // vector<int> a = G.get_left_degree_sequence();
55     // vector<int> b = G.get_right_degree_sequence();
56
57
58     // b_graph_encoder E(a,b);
59     // mpz_class m = E.encode(G);
60     // cout << "encoded: " << m << endl;
```

```
61
62 // b_graph_decoder D(a, b);
63 // b_graph Ghat = D.decode(m);
64 // cout << "decoded graph: " << endl << Ghat << endl;
65
66 // if (Ghat == G)
67 //   cout << " equal " << endl;
68 // return 0;
69
70 }
```

#### 4.17.1.2 operator<<()

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

```
15                                     {
16   for (int i=0;i<v.size();i++){
17     o << v[i];
18     if (i < v.size()-1)
19       o << ", ";
20   }
21   return o;
22 }
```

# Index

a

- b\_graph\_decoder, 16
- b\_graph\_encoder, 20
- graph\_decoder, 37

add

- fenwick\_tree, 28
- reverse\_fenwick\_tree, 52

adj\_list

- b\_graph, 10
- colored\_graph, 24
- marked\_graph, 50

adj\_location

- colored\_graph, 24
- marked\_graph, 50

b

- b\_graph\_decoder, 16
- b\_graph\_encoder, 20

b\_graph, 5

- adj\_list, 10
- b\_graph, 6, 7
- get\_adj\_list, 7
- get\_left\_degree, 8
- get\_left\_degree\_sequence, 8
- get\_right\_degree, 8
- get\_right\_degree\_sequence, 8
- left\_deg\_seq, 10
- n, 11
- np, 11
- nu\_left\_vertices, 9
- nu\_right\_vertices, 9
- operator!=, 9
- operator<<, 9
- operator==, 10
- right\_deg\_seq, 11

b\_graph\_decoder, 11

- a, 16
- b, 16
- b\_graph\_decoder, 13
- beta, 16
- decode, 13
- decode\_interval, 14
- decode\_node, 14
- init, 15
- n, 16
- np, 16
- U, 16
- W, 17
- x, 17

b\_graph\_encoder, 17

a, 20

b, 20

b\_graph\_encoder, 18

beta, 21

compute\_N, 19

encode, 19

init, 20

U, 21

beta

- b\_graph\_decoder, 16

- b\_graph\_encoder, 21

- graph\_decoder, 37

- graph\_encoder, 42

binomial

- compression\_helper.cpp, 57

- compression\_helper.h, 59

bipartite\_graph.cpp, 55

- operator!=, 55

- operator<<, 55

- operator==, 56

bipartite\_graph.h, 56

bipartite\_graph\_compression.cpp, 57

bipartite\_graph\_compression.h, 57

colored\_graph, 21

- adj\_list, 24

- adj\_location, 24

- colored\_graph, 23

- Delta, 25

- G, 25

- h, 25

- init, 23

- M, 25

- nu\_vertices, 25

- ver\_type, 25

- ver\_type\_dict, 26

- ver\_type\_int, 26

- ver\_type\_list, 26

compression\_helper.cpp, 57

- binomial, 57

- compute\_product, 58

- prod\_factorial, 58

compression\_helper.h, 59

- binomial, 59

- compute\_product, 60

- prod\_factorial, 60

compute\_N

- b\_graph\_encoder, 19

- graph\_encoder, 40

compute\_product

- compression\_helper.cpp, 58
- compression\_helper.h, 60
- decode
  - b\_graph\_decoder, 13
  - graph\_decoder, 35
- decode\_interval
  - b\_graph\_decoder, 14
  - graph\_decoder, 35
- decode\_node
  - b\_graph\_decoder, 14
  - graph\_decoder, 36
- degree\_sequence
  - graph, 33
- Delta
  - colored\_graph, 25
  - graph\_message, 47
- encode
  - b\_graph\_encoder, 19
  - graph\_encoder, 41
- fenwick.cpp, 61
- fenwick.h, 61
- fenwick\_tree, 26
  - add, 28
  - fenwick\_tree, 27
  - size, 28
  - sum, 28
  - sums, 29
- forward\_adj\_list
  - graph, 33
- FT
  - reverse\_fenwick\_tree, 53
- G
  - colored\_graph, 25
  - graph\_encoder, 42
  - graph\_message, 47
- get\_adj\_list
  - b\_graph, 7
- get\_degree
  - graph, 31
- get\_degree\_sequence
  - graph, 31
- get\_forward\_degree
  - graph, 31
- get\_forward\_list
  - graph, 31
- get\_left\_degree
  - b\_graph, 8
- get\_left\_degree\_sequence
  - b\_graph, 8
- get\_right\_degree
  - b\_graph, 8
- get\_right\_degree\_sequence
  - b\_graph, 8
- graph, 29
  - degree\_sequence, 33
  - forward\_adj\_list, 33
  - get\_degree, 31
  - get\_degree\_sequence, 31
  - get\_forward\_degree, 31
  - get\_forward\_list, 31
  - n, 34
  - nu\_vertices, 32
  - operator!=, 32
  - operator<<, 32
  - operator==, 33
- graph\_decoder, 34
  - a, 37
  - beta, 37
  - decode, 35
  - decode\_interval, 35
  - decode\_node, 36
  - graph\_decoder, 35
  - logn2, 38
  - n, 38
  - tS, 38
  - U, 38
  - x, 38
- graph\_encoder, 39
  - beta, 42
  - compute\_N, 40
  - encode, 41
  - G, 42
  - graph\_encoder, 39
  - init, 41
  - logn2, 42
  - Stilde, 42
  - U, 43
- graph\_message, 43
  - Delta, 47
  - G, 47
  - graph\_message, 44
  - h, 47
  - message\_dict, 47
  - message\_list, 47
  - messages, 48
  - update\_message\_dictionary, 44
  - update\_messages, 45
- graph\_message.cpp, 61
  - pair\_compare, 62
- graph\_message.h, 62
  - pair\_compare, 62
- h
  - colored\_graph, 25
  - graph\_message, 47
- init
  - b\_graph\_decoder, 15
  - b\_graph\_encoder, 20
  - colored\_graph, 23
  - graph\_encoder, 41
- left\_deg\_seq



- b\_graph, 10
- logn2
  - graph\_decoder, 38
  - graph\_encoder, 42
- M
  - colored\_graph, 25
- main
  - test.cpp, 67
- marked\_graph, 48
  - adj\_list, 50
  - adj\_location, 50
  - marked\_graph, 49
  - nu\_vertices, 50
  - ver\_mark, 50
- marked\_graph.cpp, 63
  - operator>>, 63
- marked\_graph.h, 64
  - operator>>, 64
- message\_dict
  - graph\_message, 47
- message\_list
  - graph\_message, 47
- messages
  - graph\_message, 48
- n
  - b\_graph, 11
  - b\_graph\_decoder, 16
  - graph, 34
  - graph\_decoder, 38
- np
  - b\_graph, 11
  - b\_graph\_decoder, 16
- nu\_left\_vertices
  - b\_graph, 9
- nu\_right\_vertices
  - b\_graph, 9
- nu\_vertices
  - colored\_graph, 25
  - graph, 32
  - marked\_graph, 50
- operator!=
  - b\_graph, 9
  - bipartite\_graph.cpp, 55
  - graph, 32
  - simple\_graph.cpp, 65
- operator<<
  - b\_graph, 9
  - bipartite\_graph.cpp, 55
  - graph, 32
  - simple\_graph.cpp, 65
  - test.cpp, 68
- operator>>
  - marked\_graph.cpp, 63
  - marked\_graph.h, 64
- operator==
  - b\_graph, 10
- bipartite\_graph.cpp, 56
- graph, 33
- simple\_graph.cpp, 65
- pair\_compare
  - graph\_message.cpp, 62
  - graph\_message.h, 62
- prod\_factorial
  - compression\_helper.cpp, 58
  - compression\_helper.h, 60
- reverse\_fenwick\_tree, 51
  - add, 52
  - FT, 53
  - reverse\_fenwick\_tree, 51
  - size, 52
  - sum, 52
- right\_deg\_seq
  - b\_graph, 11
- simple\_graph.cpp, 65
  - operator!=, 65
  - operator<<, 65
  - operator==, 65
- simple\_graph.h, 66
- simple\_graph\_compression.cpp, 66
- simple\_graph\_compression.h, 66
- size
  - fenwick\_tree, 28
  - reverse\_fenwick\_tree, 52
- Stilde
  - graph\_encoder, 42
- sum
  - fenwick\_tree, 28
  - reverse\_fenwick\_tree, 52
- sums
  - fenwick\_tree, 29
- test.cpp, 67
  - main, 67
  - operator<<, 68
- tS
  - graph\_decoder, 38
- U
  - b\_graph\_decoder, 16
  - b\_graph\_encoder, 21
  - graph\_decoder, 38
  - graph\_encoder, 43
- update\_message\_dictionary
  - graph\_message, 44
- update\_messages
  - graph\_message, 45
- ver\_mark
  - marked\_graph, 50
- ver\_type
  - colored\_graph, 25
- ver\_type\_dict
  - colored\_graph, 26

ver\_type\_int  
    [colored\\_graph, 26](#)  
ver\_type\_list  
    [colored\\_graph, 26](#)

## W

[b\\_graph\\_decoder, 17](#)

## x

[b\\_graph\\_decoder, 17](#)  
    [graph\\_decoder, 38](#)