# Marked Graph Compression

Generated by Doxygen 1.8.13

# Contents


**1 Main Page** **1**

**2 Namespace Index** **3**

2.1 Namespace List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**3 Class Index** **5**

3.1 Class List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**4 File Index** **7**

4.1 File List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**5 Namespace Documentation** **9**

5.1 helper_vars Namespace Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

5.1.1 Variable Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

5.1.1.1 mpz_vec . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

5.1.1.2 mpz_vec2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

5.1.1.3 mul_1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

5.1.1.4 mul_2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

5.1.1.5 return_stack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

6 Class Documentation 11
6.1 b_graph Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
6.1.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
6.1.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . 12
6.1.2.1 b_graph() [1/4] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6.1.2.2 b_graph() [2/4] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6.1.2.3 b_graph() [3/4] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6.1.2.4 b_graph() [4/4] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
6.1.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 14
6.1.3.1 get_adj_list() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
6.1.3.2 get_left_degree() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
6.1.3.3 get_left_degree_sequence() . . . . . . . . . . . . . . . . . . . . . . . . 15
6.1.3.4 get_right_degree() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
6.1.3.5 get_right_degree_sequence() . . . . . . . . . . . . . . . . . . . . . . . 16
6.1.3.6 nu_left_vertices() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
6.1.3.7 nu_right_vertices() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
6.1.4 Friends And Related Function Documentation . . . . . . . . . . . . . . . . . . . . 16
6.1.4.1 operator"!= . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
6.1.4.2 operator<< . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
6.1.4.3 operator== . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
6.1.5 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
6.1.5.1 adj_list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
6.1.5.2 left_deg_seq . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.1.5.3 n . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.1.5.4 np . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.1.5.5 right_deg_seq . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.2 b_graph_decoder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.2.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
6.2.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . 20
6.2.2.1 b_graph_decoder() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

6.2.3 Member Function Documentation . . . . . 20
6.2.3.1 decode() . . . . . 20
6.2.3.2 decode_interval() . . . . . 21
6.2.3.3 decode_node() . . . . . 22
6.2.3.4 init() . . . . . 22
6.2.4 Member Data Documentation . . . . . 23
6.2.4.1 a . . . . . 23
6.2.4.2 b . . . . . 23
6.2.4.3 beta . . . . . 23
6.2.4.4 n . . . . . 23
6.2.4.5 np . . . . . 23
6.2.4.6 U . . . . . 24
6.2.4.7 W . . . . . 24
6.2.4.8 x . . . . . 24
6.3 b_graph_encoder Class Reference . . . . . 24
6.3.1 Detailed Description . . . . . 25
6.3.2 Constructor & Destructor Documentation . . . . . 25
6.3.2.1 b_graph_encoder() . . . . . 25
6.3.3 Member Function Documentation . . . . . 26
6.3.3.1 compute_N() . . . . . 26
6.3.3.2 encode() . . . . . 28
6.3.3.3 init() . . . . . 29
6.3.4 Member Data Documentation . . . . . 29
6.3.4.1 a . . . . . 29
6.3.4.2 b . . . . . 29
6.3.4.3 beta . . . . . 30
6.3.4.4 U . . . . . 30
6.4 bit_pipe Class Reference . . . . . 30
6.4.1 Detailed Description . . . . . 31
6.4.2 Constructor & Destructor Documentation . . . . . 31

6.4.2.1 bit_pipe() [1/3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
6.4.2.2 bit_pipe() [2/3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
6.4.2.3 bit_pipe() [3/3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
6.4.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 32
6.4.3.1 append_left() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.4.3.2 chunks() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.4.3.3 operator[]() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.4.3.4 operator[]() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4.3.5 residue() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4.3.6 shift_left() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4.3.7 shift_right() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.4.3.8 size() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.4.4 Friends And Related Function Documentation . . . . . . . . . . . . . . . . . . 35
6.4.4.1 ibitstream . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.4.4.2 obitstream . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.4.4.3 operator<< [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.4.4.4 operator<< [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.4.4.5 operator>> . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.4.5 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.4.5.1 bits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.4.5.2 last_bits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.5 colored_graph Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.5.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
6.5.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . 39
6.5.2.1 colored_graph() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
6.5.2.2 colored_graph() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.5.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 40
6.5.3.1 init() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.5.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.5.4.1 adj_list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

6.5.4.2 deg . . . . . 42
6.5.4.3 Delta . . . . . 42
6.5.4.4 h . . . . . 42
6.5.4.5 index_in_neighbor . . . . . 42
6.5.4.6 is_star_vertex . . . . . 42
6.5.4.7 M . . . . . 42
6.5.4.8 nu_vertices . . . . . 43
6.5.4.9 star_vertices . . . . . 43
6.5.4.10 ver_type . . . . . 43
6.5.4.11 ver_type_dict . . . . . 43
6.5.4.12 ver_type_int . . . . . 43
6.5.4.13 ver_type_list . . . . . 44
6.6 fenwick_tree Class Reference . . . . . 44
6.6.1 Detailed Description . . . . . 44
6.6.2 Constructor & Destructor Documentation . . . . . 44
6.6.2.1 fenwick_tree() [1/2] . . . . . 45
6.6.2.2 fenwick_tree() [2/2] . . . . . 45
6.6.3 Member Function Documentation . . . . . 45
6.6.3.1 add() . . . . . 45
6.6.3.2 size() . . . . . 46
6.6.3.3 sum() . . . . . 46
6.6.4 Member Data Documentation . . . . . 46
6.6.4.1 sums . . . . . 46
6.7 graph Class Reference . . . . . 47
6.7.1 Detailed Description . . . . . 47
6.7.2 Constructor & Destructor Documentation . . . . . 48
6.7.2.1 graph() [1/3] . . . . . 48
6.7.2.2 graph() [2/3] . . . . . 48
6.7.2.3 graph() [3/3] . . . . . 48
6.7.3 Member Function Documentation . . . . . 49

6.7.3.1 get_degree() . . . . . 49
6.7.3.2 get_degree_sequence() . . . . . 49
6.7.3.3 get_forward_degree() . . . . . 50
6.7.3.4 get_forward_list() . . . . . 50
6.7.3.5 nu_vertices() . . . . . 50
6.7.4 Friends And Related Function Documentation . . . . . 50
6.7.4.1 operator"!= . . . . . 50
6.7.4.2 operator<< . . . . . 51
6.7.4.3 operator== . . . . . 51
6.7.5 Member Data Documentation . . . . . 51
6.7.5.1 degree_sequence . . . . . 51
6.7.5.2 forward_adj_list . . . . . 52
6.7.5.3 n . . . . . 52
6.8 graph_decoder Class Reference . . . . . 52
6.8.1 Detailed Description . . . . . 53
6.8.2 Constructor & Destructor Documentation . . . . . 53
6.8.2.1 graph_decoder() . . . . . 53
6.8.3 Member Function Documentation . . . . . 54
6.8.3.1 decode() . . . . . 54
6.8.3.2 decode_interval() . . . . . 54
6.8.3.3 decode_node() . . . . . 55
6.8.3.4 init() . . . . . 56
6.8.4 Member Data Documentation . . . . . 56
6.8.4.1 a . . . . . 57
6.8.4.2 beta . . . . . 57
6.8.4.3 logn2 . . . . . 57
6.8.4.4 n . . . . . 57
6.8.4.5 tS . . . . . 57
6.8.4.6 U . . . . . 57
6.8.4.7 x . . . . . 58

6.9 graph_encoder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.9.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.9.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . 59
6.9.2.1 graph_encoder() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.9.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 59
6.9.3.1 compute_N() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.9.3.2 encode() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.9.3.3 init() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.9.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9.4.1 a . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9.4.2 beta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9.4.3 logn2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9.4.4 n . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9.4.5 Stilde . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9.4.6 U . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.10 graph_message Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.10.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.10.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . 65
6.10.2.1 graph_message() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.10.2.2 graph_message() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.10.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 66
6.10.3.1 send_message() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.10.3.2 update_messages() . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.10.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.10.4.1 Delta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.10.4.2 h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.10.4.3 is_star_message . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.10.4.4 message_dict . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.10.4.5 message_mark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.10.4.6 messages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

6.11 ibitstream Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.11.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.11.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.11.2.1 ibitstream() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.11.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.11.3.1 bin_inter_decode() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . 75
6.11.3.2 bin_inter_decode() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . 76
6.11.3.3 close() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.11.3.4 operator>>() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.11.3.5 operator>>() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . 78
6.11.3.6 read_bit() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
6.11.3.7 read_bits() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . 79
6.11.3.8 read_bits() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.11.3.9 read_bits_append() . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.11.3.10 read_chunk() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.11.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.11.4.1 buffer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.11.4.2 f . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.11.4.3 head_mask . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.11.4.4 head_place . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.12 log_entry Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.12.1 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.12.1.1 log_entry() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.12.2 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.12.2.1 depth . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.12.2.2 description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.12.2.3 name . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.12.2.4 sys_t . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.12.2.5 t . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.13 logger Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

6.13.1 Member Function Documentation . . . . . 84
6.13.1.1 add_entry() . . . . . 84
6.13.1.2 item_start() . . . . . 85
6.13.1.3 item_stop() . . . . . 85
6.13.1.4 log() . . . . . 85
6.13.1.5 start() . . . . . 86
6.13.1.6 stop() . . . . . 86
6.13.2 Member Data Documentation . . . . . 86
6.13.2.1 current_depth . . . . . 87
6.13.2.2 item_duration . . . . . 87
6.13.2.3 item_last_start . . . . . 87
6.13.2.4 logs . . . . . 87
6.13.2.5 report . . . . . 87
6.13.2.6 report_stream . . . . . 87
6.13.2.7 stat . . . . . 87
6.13.2.8 stat_stream . . . . . 88
6.13.2.9 verbose . . . . . 88
6.13.2.10 verbose_stream . . . . . 88
6.14 marked_graph Class Reference . . . . . 88
6.14.1 Detailed Description . . . . . 89
6.14.2 Constructor & Destructor Documentation . . . . . 89
6.14.2.1 marked_graph() [1/2] . . . . . 89
6.14.2.2 marked_graph() [2/2] . . . . . 89
6.14.3 Friends And Related Function Documentation . . . . . 90
6.14.3.1 operator"!= . . . . . 90
6.14.3.2 operator<< . . . . . 90
6.14.3.3 operator== . . . . . 91
6.14.4 Member Data Documentation . . . . . 92
6.14.4.1 adj_list . . . . . 92
6.14.4.2 index_in_neighbor . . . . . 92

6.14.4.3 nu_vertices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
6.14.4.4 ver_mark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
6.15 marked_graph_compressed Class Reference . . . . . . . . . . . . . . . . . . . . . . . . 93
6.15.1 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 93
6.15.1.1 binary_read() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
6.15.1.2 binary_read() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
6.15.1.3 binary_write() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
6.15.1.4 binary_write() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.15.1.5 clear() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.15.2 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.15.2.1 delta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.15.2.2 h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.15.2.3 n . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.15.2.4 part_bgraph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.15.2.5 part_graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.15.2.6 star_edges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.15.2.7 star_vertices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.15.2.8 type_mark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.15.2.9 ver_type_list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.15.2.10 ver_types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
6.16 marked_graph_decoder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . 107
6.16.1 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . 108
6.16.1.1 marked_graph_decoder() . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.16.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 108
6.16.2.1 decode() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.16.2.2 decode_partition_bgraphs() . . . . . . . . . . . . . . . . . . . . . . . . 109
6.16.2.3 decode_partition_graphs() . . . . . . . . . . . . . . . . . . . . . . . . 110
6.16.2.4 decode_star_edges() . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
6.16.2.5 decode_star_vertices() . . . . . . . . . . . . . . . . . . . . . . . . . . 111
6.16.2.6 decode_vertex_types() . . . . . . . . . . . . . . . . . . . . . . . . . . 111

6.16.2.7 find_part_deg_orig_index() . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
6.16.3 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
6.16.3.1 Deg . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
6.16.3.2 delta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
6.16.3.3 edges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
6.16.3.4 h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.16.3.5 is_star_vertex . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.16.3.6 n . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.16.3.7 origin_index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.16.3.8 part_bgraph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.16.3.9 part_deg . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.16.3.10 part_graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.16.3.11 star_vertices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.16.3.12 vertex_marks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.17 marked_graph_encoder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.17.1 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 116
6.17.1.1 marked_graph_encoder() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
6.17.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
6.17.2.1 encode() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
6.17.2.2 encode() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
6.17.2.3 encode_partition_bgraphs() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
6.17.2.4 encode_partition_graphs() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
6.17.2.5 encode_star_edges() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
6.17.2.6 encode_star_vertices() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.17.2.7 encode_vertex_types() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.17.2.8 extract_edge_types() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.17.2.9 extract_partition_graphs() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
6.17.2.10 find_part_index_deg() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.17.3 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.17.3.1 C . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

6.17.3.2 compressed . . . 121
6.17.3.3 delta . . . 121
6.17.3.4 h . . . 122
6.17.3.5 is_star_vertex . . . 122
6.17.3.6 n . . . 122
6.17.3.7 part_bgraph . . . 122
6.17.3.8 part_deg . . . 122
6.17.3.9 part_graph . . . 122
6.17.3.10 part_index . . . 123
6.17.3.11 star_vertices . . . 123
6.18 obitstream Class Reference . . . 123
6.18.1 Detailed Description . . . 124
6.18.2 Constructor & Destructor Documentation . . . 124
6.18.2.1 obitstream() . . . 124
6.18.3 Member Function Documentation . . . 124
6.18.3.1 bin_inter_code() [1/2] . . . 124
6.18.3.2 bin_inter_code() [2/2] . . . 125
6.18.3.3 chunks() . . . 126
6.18.3.4 close() . . . 126
6.18.3.5 operator<<() [1/2] . . . 126
6.18.3.6 operator<<() [2/2] . . . 127
6.18.3.7 write() . . . 127
6.18.3.8 write_bits() . . . 127
6.18.4 Member Data Documentation . . . 128
6.18.4.1 buffer . . . 128
6.18.4.2 chunks_written . . . 128
6.18.4.3 f . . . 128
6.19 reverse_fenwick_tree Class Reference . . . 128
6.19.1 Detailed Description . . . 129
6.19.2 Constructor & Destructor Documentation . . . 129

6.19.2.1 reverse_fenwick_tree() [1/2] . . . . . . . . . . . . . . . . . . . . . . . . 129
6.19.2.2 reverse_fenwick_tree() [2/2] . . . . . . . . . . . . . . . . . . . . . . . . 129
6.19.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 129
6.19.3.1 add() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.19.3.2 size() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.19.3.3 sum() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.19.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.19.4.1 FT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.20 time_series_decoder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . 131
6.20.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
6.20.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . 132
6.20.2.1 time_series_decoder() . . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.20.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 132
6.20.3.1 decode() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.20.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.20.4.1 alph_size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.20.4.2 freq . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.20.4.3 G . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.20.4.4 n . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.21 time_series_encoder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . 133
6.21.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
6.21.2 Constructor & Destructor Documentation . . . . . . . . . . . . . . . . . . . . . 134
6.21.2.1 time_series_encoder() . . . . . . . . . . . . . . . . . . . . . . . . . . 134
6.21.3 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 134
6.21.3.1 encode() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
6.21.3.2 init_alph_size() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
6.21.3.3 init_freq() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.21.3.4 init_G() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.21.4 Member Data Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.21.4.1 alph_size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.21.4.2 freq . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.21.4.3 G . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
6.21.4.4 n . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
6.22 vint_hash Struct Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
6.22.1 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . 137
6.22.1.1 operator()() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

7 File Documentation 139
7.1 bipartite_graph.cpp File Reference . . . . . 139
7.1.1 Function Documentation . . . . . 139
7.1.1.1 operator"!=() . . . . . 139
7.1.1.2 operator<<() . . . . . 140
7.1.1.3 operator==() . . . . . 140
7.2 bipartite_graph.h File Reference . . . . . 140
7.3 bipartite_graph_compression.cpp File Reference . . . . . 141
7.4 bipartite_graph_compression.h File Reference . . . . . 141
7.5 bitstream.cpp File Reference . . . . . 141
7.5.1 Function Documentation . . . . . 142
7.5.1.1 elias_delta_encode() [1/4] . . . . . 142
7.5.1.2 elias_delta_encode() [2/4] . . . . . 142
7.5.1.3 elias_delta_encode() [3/4] . . . . . 143
7.5.1.4 elias_delta_encode() [4/4] . . . . . 143
7.5.1.5 mask_gen() . . . . . 143
7.5.1.6 nu_bits() . . . . . 144
7.5.1.7 operator<<() [1/2] . . . . . 144
7.5.1.8 operator<<() [2/2] . . . . . 144
7.5.1.9 operator>>() . . . . . 145
7.6 bitstream.h File Reference . . . . . 145
7.6.1 Function Documentation . . . . . 146
7.6.1.1 elias_delta_encode() [1/4] . . . . . 146
7.6.1.2 elias_delta_encode() [2/4] . . . . . 146
7.6.1.3 elias_delta_encode() [3/4] . . . . . 147
7.6.1.4 elias_delta_encode() [4/4] . . . . . 147
7.6.1.5 mask_gen() . . . . . 147
7.6.1.6 nu_bits() . . . . . 148
7.6.2 Variable Documentation . . . . . 148
7.6.2.1 BIT_INT . . . . . 148

7.6.2.2 BYTE_INT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
7.7 compression_helper.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
7.7.1 Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
7.7.1.1 binomial() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
7.7.1.2 bit_string_read() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150
7.7.1.3 bit_string_write() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150
7.7.1.4 compute_array_product() . . . . . . . . . . . . . . . . . . . . . . . . 151
7.7.1.5 compute_product() . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
7.7.1.6 compute_product_old() . . . . . . . . . . . . . . . . . . . . . . . . . 152
7.7.1.7 compute_product_stack() . . . . . . . . . . . . . . . . . . . . . . . . 153
7.7.1.8 compute_product_void() . . . . . . . . . . . . . . . . . . . . . . . . . 155
7.7.1.9 prod_factorial() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
7.7.1.10 prod_factorial_old() . . . . . . . . . . . . . . . . . . . . . . . . . . 156
7.8 compression_helper.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
7.8.1 Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
7.8.1.1 binomial() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
7.8.1.2 bit_string_read() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
7.8.1.3 bit_string_write() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
7.8.1.4 compute_array_product() . . . . . . . . . . . . . . . . . . . . . . . . 160
7.8.1.5 compute_product() . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
7.8.1.6 compute_product_old() . . . . . . . . . . . . . . . . . . . . . . . . . 161
7.8.1.7 compute_product_stack() . . . . . . . . . . . . . . . . . . . . . . . . 162
7.8.1.8 compute_product_void() . . . . . . . . . . . . . . . . . . . . . . . . . 163
7.8.1.9 prod_factorial() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
7.8.1.10 prod_factorial_old() . . . . . . . . . . . . . . . . . . . . . . . . . . 165
7.9 fenwick.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
7.10 fenwick.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
7.11 gcomp.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
7.11.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167
7.11.2 Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167

7.11.2.1 main() . . . . . . . . . . 167
7.12 graph_message.cpp File Reference . . . . . . . . . . 169
7.12.1 Function Documentation . . . . . . . . . . 169
7.12.1.1 pair_compare() . . . . . . . . . . 169
7.13 graph_message.h File Reference . . . . . . . . . . 169
7.13.1 Function Documentation . . . . . . . . . . 170
7.13.1.1 pair_compare() . . . . . . . . . . 170
7.14 logger.cpp File Reference . . . . . . . . . . 170
7.15 logger.h File Reference . . . . . . . . . . 170
7.15.1 Function Documentation . . . . . . . . . . 171
7.15.1.1 __attribute__() [1/2] . . . . . . . . . . 171
7.15.1.2 __attribute__() [2/2] . . . . . . . . . . 171
7.16 marked_graph.cpp File Reference . . . . . . . . . . 171
7.16.1 Function Documentation . . . . . . . . . . 171
7.16.1.1 edge_compare() . . . . . . . . . . 172
7.16.1.2 operator"!=() . . . . . . . . . . 172
7.16.1.3 operator<<() . . . . . . . . . . 172
7.16.1.4 operator==() . . . . . . . . . . 173
7.16.1.5 operator>>() . . . . . . . . . . 173
7.17 marked_graph.h File Reference . . . . . . . . . . 174
7.17.1 Function Documentation . . . . . . . . . . 174
7.17.1.1 edge_compare() . . . . . . . . . . 175
7.17.1.2 operator>>() . . . . . . . . . . 175
7.18 marked_graph_compression.cpp File Reference . . . . . . . . . . 175
7.19 marked_graph_compression.h File Reference . . . . . . . . . . 176
7.20 random_graph.cpp File Reference . . . . . . . . . . 176
7.20.1 Function Documentation . . . . . . . . . . 176
7.20.1.1 marked_ER() . . . . . . . . . . 176
7.20.1.2 near_regular_graph() . . . . . . . . . . 177
7.20.1.3 poisson_graph() . . . . . . . . . . 178

7.21 random_graph.h File Reference . . . . . 179
7.21.1 Function Documentation . . . . . 180
7.21.1.1 marked_ER() . . . . . 180
7.21.1.2 near_regular_graph() . . . . . 180
7.21.1.3 poisson_graph() . . . . . 181
7.22 README.md File Reference . . . . . 182
7.23 rnd_graph.cpp File Reference . . . . . 182
7.23.1 Function Documentation . . . . . 183
7.23.1.1 main() . . . . . 183
7.23.1.2 print_usage() . . . . . 184
7.24 simple_graph.cpp File Reference . . . . . 184
7.24.1 Function Documentation . . . . . 184
7.24.1.1 operator"!=() . . . . . 184
7.24.1.2 operator<<() . . . . . 185
7.24.1.3 operator==() . . . . . 185
7.25 simple_graph.h File Reference . . . . . 185
7.26 simple_graph_compression.cpp File Reference . . . . . 186
7.27 simple_graph_compression.h File Reference . . . . . 186
7.28 test.cpp File Reference . . . . . 186
7.28.1 Function Documentation . . . . . 187
7.28.1.1 b_graph_test() . . . . . 187
7.28.1.2 graph_test() . . . . . 187
7.28.1.3 main() . . . . . 187
7.28.1.4 marked_graph_encoder_test() . . . . . 188
7.28.1.5 operator<<() . . . . . 189
7.28.1.6 random_graph_test() . . . . . 189
7.28.1.7 time_series_compression_test() . . . . . 190
7.29 test_mp.cpp File Reference . . . . . 190
7.29.1 Function Documentation . . . . . 190
7.29.1.1 main() . . . . . 190
7.29.1.2 mp_test() . . . . . 191
7.29.1.3 operator<<() . . . . . 191
7.29.1.4 random_graph_test() . . . . . 191
7.30 time_series_compression.cpp File Reference . . . . . 192
7.31 time_series_compression.h File Reference . . . . . 192

**Index 193**

# Chapter 1

# Main Page

This is a test

# Chapter 2

# Namespace Index

## 2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

helper_vars . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

# Chapter 3

# Class Index

## 3.1 Class List

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


b_graph
Simple unmarked bipartite graph . . . 11
b_graph_decoder
Decodes a simple unmarked bipartite graph . . . 18
b_graph_encoder
Encodes a simple unmarked bipartite graph . . . 24
bit_pipe
A sequence of arbitrary number of bits . . . 30
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 . . . 37
fenwick_tree
Fenwick tree class . . . 44
graph
Simple unmarked graph . . . 47
graph_decoder
Decodes a simple unmarked graph . . . 52
graph_encoder
Encodes a simple unmarked graph . . . 58
graph_message
This class takes care of message passing on marked graphs . . . 64
ibitstream
Deals with reading bit streams from binary files, this is the reverse of obitstream . . . 74
log_entry . . . 82
logger . . . 84
marked_graph
Simple marked graph . . . 88
marked_graph_compressed . . . 93
marked_graph_decoder . . . 107
marked_graph_encoder . . . 114
obitstream
Handles writing bitstreams to binary files . . . 123
reverse_fenwick_tree
Similar to the fenwick_tree class, but instead of prefix sums, this class computes suffix sums . . . 128
time_series_decoder
Decodes a time series which is basically an array of arbitrary nonnegative integers . . . 131

time_series_encoder
Encodes a time series which is basically an array of arbitrary nonnegative integers . . . . . . . . 133
vint_hash . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

# Chapter 4

# File Index

## 4.1 File List

Here is a list of all files with brief descriptions:

bipartite_graph.cpp . . . 139
bipartite_graph.h . . . 140
bipartite_graph_compression.cpp . . . 141
bipartite_graph_compression.h . . . 141
bitstream.cpp . . . 141
bitstream.h . . . 145
compression_helper.cpp . . . 148
compression_helper.h . . . 157
fenwick.cpp . . . 166
fenwick.h . . . 166
gcomp.cpp
To compress / decompress simple marked graphs . . . 166
graph_message.cpp . . . 169
graph_message.h . . . 169
logger.cpp . . . 170
logger.h . . . 170
marked_graph.cpp . . . 171
marked_graph.h . . . 174
marked_graph_compression.cpp . . . 175
marked_graph_compression.h . . . 176
random_graph.cpp . . . 176
random_graph.h . . . 179
rnd_graph.cpp . . . 182
simple_graph.cpp . . . 184
simple_graph.h . . . 185
simple_graph_compression.cpp . . . 186
simple_graph_compression.h . . . 186
test.cpp . . . 186
test_mp.cpp . . . 190
time_series_compression.cpp . . . 192
time_series_compression.h . . . 192

# Chapter 5

# Namespace Documentation

## 5.1 helper_vars Namespace Reference

### Variables

- mpz_class mul_1
- mpz_class mul_2

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

### 5.1.1 Variable Documentation

#### 5.1.1.1 mpz_vec

```
vector< mpz_class > helper_vars::mpz_vec
```

#### 5.1.1.2 mpz_vec2

```
vector< mpz_class > helper_vars::mpz_vec2
```

#### 5.1.1.3 mul_1

```
mpz_class helper_vars::mul_1
```

#### 5.1.1.4 mul_2

```
mpz_class helper_vars::mul_2
```

helper variables in order to avoid initialization

#### 5.1.1.5 return_stack

```
vector< mpz_class > helper_vars::return_stack
```

# Chapter 6

# Class Documentation

## 6.1 b_graph Class Reference

simple unmarked bipartite graph

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

### Public Member Functions

- b_graph ()

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

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

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

  *a constructor*
- vector< int > get_adj_list (int v) const

  *returns the adjacency list of a given left vertex*
- int get_right_degree (int v) const

  *returns the degree of a right vertex v*
- int get_left_degree (int v) const

  *returns the degree of a right vertex v*
- vector< int > get_right_degree_sequence () const

  *return the right degree sequence*
- vector< int > get_left_degree_sequence () const

  *return the left degree sequence*
- int nu_left_vertices () const

  *returns the number of left vertices*
- int nu_right_vertices () const

  *returns the number of right vertices*

### Private Attributes

- int n

    *the number of left vertices*
- int np

    *the number of right vertices*
- vector< vector< int > > adj_list

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

    *degree sequence for left vertices, where left_deg_seq[v] is the degree of the left node v*
- vector< int > right_deg_seq

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

### Friends

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

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

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

    *comparing for inequality*

## 6.1.1 Detailed Description

simple unmarked bipartite graph

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

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

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

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

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

## 6.1.2 Constructor & Destructor Documentation

#### 6.1.2.1 b_graph() [1/4]

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

default constructor

```
: n(0), np(0) {}
```

#### 6.1.2.2 b_graph() [2/4]

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

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

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

**Parameters**

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

```
{
  n = left_deg.size();
  np = right_deg.size();
  adj_list = list;
  left_deg_seq = left_deg;
  right_deg_seq = right_deg;
}
```

#### 6.1.2.3 b_graph() [3/4]

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

a constructor

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

**Parameters**

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

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

#### 6.1.2.4 b_graph() [4/4]

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

a constructor

This constructor takes the list of adjacent vertices

**Parameters**

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

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

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

### 6.1.3 Member Function Documentation

#### 6.1.3.1 get_adj_list()

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

returns the adjacency list of a given left vertex

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

#### 6.1.3.2 get_left_degree()

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

returns the degree of a right vertex v

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

#### 6.1.3.3 get_left_degree_sequence()

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

return the left degree sequence

```
{
  return left_deg_seq;
}
```

#### 6.1.3.4 get_right_degree()

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

returns the degree of a right vertex v

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

#### 6.1.3.5 get_right_degree_sequence()

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

return the right degree sequence

```
{
  return right_deg_seq;
}
```

#### 6.1.3.6 nu_left_vertices()

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

returns the number of left vertices

```
{
  return n;
}
```

#### 6.1.3.7 nu_right_vertices()

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

returns the number of right vertices

```
{
  return np;
}
```

### 6.1.4 Friends And Related Function Documentation

#### 6.1.4.1 operator"!=

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

comparing for inequality

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

#### 6.1.4.2 operator<<

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

printing the graph to the output

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

#### 6.1.4.3 operator==

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

comparing two graphs for equality

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

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

### 6.1.5 Member Data Documentation

#### 6.1.5.1 adj_list

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

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

#### 6.1.5.2 left_deg_seq

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

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

#### 6.1.5.3 n

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

the number of left vertices

#### 6.1.5.4 np

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

the number of right vertices

#### 6.1.5.5 right_deg_seq

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

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

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

- bipartite_graph.h
- bipartite_graph.cpp

## 6.2 b_graph_decoder Class Reference

Decodes a simple unmarked bipartite graph.

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

### Public Member Functions

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

  *constructor*
- void init ()

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

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

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

  *decodes the bipartite graph given the encoded integer*

### Private Attributes

- int n

  *number of left vertices*
- int np

  *number of right vertices*
- vector< int > a

  *left degree sequence*
- vector< int > b

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

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

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

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

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

### 6.2.1 Detailed Description

Decodes a simple unmarked bipartite graph.

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

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

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

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

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

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

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

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

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

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

### 6.2.2 Constructor & Destructor Documentation

#### 6.2.2.1 b_graph_decoder()

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

constructor

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

### 6.2.3 Member Function Documentation

#### 6.2.3.1 decode()

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

decodes the bipartite graph given the encoded integer

**Parameters**

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

**Returns**

the decoded bipartite graph $G$

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

#### 6.2.3.2 decode_interval()

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

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

**Parameters**

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

**Returns**

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

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

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

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

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

#### 6.2.3.3 decode_node()

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

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

**Parameters**

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

**Returns**

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

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

#### 6.2.3.4 init()

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

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

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

}
```

### 6.2.4 Member Data Documentation

#### 6.2.4.1 a

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

left degree sequence

#### 6.2.4.2 b

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

right degree sequence

#### 6.2.4.3 beta

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

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

#### 6.2.4.4 n

`int b_graph_decoder::n [private]`

number of left vertices

#### 6.2.4.5 np

`int b_graph_decoder::np [private]`

number of right vertices

#### 6.2.4.6 U

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

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

#### 6.2.4.7 W

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

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

#### 6.2.4.8 x

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

the adjacency list of left nodes for the decoded graph

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

- bipartite_graph_compression.h
- bipartite_graph_compression.cpp

## 6.3 b_graph_encoder Class Reference

Encodes a simple unmarked bipartite graph.

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

### Public Member Functions

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

  *constructor*
- void init (const b_graph &G)

  *initializes beta and U*
- pair< mpz_class, mpz_class > compute_N (const b_graph &G)

  *computes $N(G)$*
- mpz_class encode (const b_graph &G)

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

## Private Attributes

- vector< int > beta

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

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

  *the degree sequence for the right nodes*
- reverse_fenwick_tree U

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

### 6.3.1 Detailed Description

Encodes a simple unmarked bipartite graph.

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

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

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

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

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

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

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

### 6.3.2 Constructor & Destructor Documentation

#### 6.3.2.1 b_graph_encoder()

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

constructor

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

### 6.3.3 Member Function Documentation

#### 6.3.3.1 compute_N()

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

computes $N(G)$

#### Parameters

| | |
|---|---|
| *G* | reference to the bipartite graph for which we compute N |

#### Returns

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

```
                                                                        {
//logger::item_start("bip init");
int n_l = G.nu_left_vertices(); // number of left vertices
int n_bits = 0;
int n_copy = n_l;
while (n_copy > 0){
  n_bits ++;
  n_copy >>= 1;
}
n_bits += 2;

vector<pair<int, int> > call_stack(2 * n_bits);
vector<pair<mpz_class, mpz_class> > return_stack(2 * n_bits); // first = N, second = l
vector<mpz_class> r_stack(2 * n_bits); // stack of r values

vector<int> status_stack(2 * n_bits);
//vector<int> St_stack(2 * n_bits); // stack to store values of St

call_stack[0] = pair<int, int> (0,n_l-1); // i j
status_stack[0] = 0; // newly added

int call_size = 1; // the size of the call stack
int return_size = 0; // the size of the return stack

int i, j, t, Sj;
int status;

vector<int> gamma; // forward list  of the graph

mpz_class rtj, prod_afac, Nit_rtj, lit_Ntj, bin;
//logger::item_stop("bip init");
while(call_size > 0){
  //cerr << " call_size " << call_size << endl;
  i = call_stack[call_size-1].first;
  j = call_stack[call_size-1].second;
  if (i==j){
    //logger::item_start("bip enc i = j");
    return_stack[return_size].first = 0; // N_{i,j} is initialized with 0
    return_stack[return_size].second = 1; // l_{i,j} is initialized with 1
    r_stack[return_size] = binomial(U.sum(0), a[i]); // r_i = \binom{S_i}{a_i}, s_i =
  U.sum(0)
    gamma = G.get_adj_list(i);
    for (int k=0;k<a[i];k++){
      //logger::item_start("bip enc i = j binomial");
      bin = binomial(U.sum(1+gamma[k]), a[i] - k);
      //logger::item_stop("bip enc i = j binomial");

      //logger::item_start("bip enc i = j arithmetic");
      return_stack[return_size].first += return_stack[return_size].second * bin;
      return_stack[return_size].second *= beta[gamma[k]];
      //logger::item_start("bip enc i = j arithmetic");
      beta[gamma[k]] --;
      U.add(gamma[k],-1);
    }
    return_size ++;
    call_size --;
    //logger::item_stop("bip enc i = j");
  }else{
    //logger::item_start("bip enc i neq j");
    t = (i+j)/2;
    status = status_stack[call_size - 1];
    //logger::item_start("bip enc stacking 0 1");
    if (status == 0){
      // newly added, left node must be called
      call_stack[call_size].first = i;
      call_stack[call_size].second = t;
      status_stack[call_size-1] = 1; // left is called
      status_stack[call_size] = 0; // newly added
      call_size++;
    }
    if (status == 1){
      // left is returned
      //St_stack[call_size-1] = U.sum(0);
```

```
       // call the right child
       call_stack[call_size].first = t+1;
        call_stack[call_size].second = j;
        status_stack[call_size-1] = 2; //right is called
        status_stack[call_size] = 0;  // newly called
        call_size ++;
      }
      //logger::item_stop("bip enc stacking 0 1");
      if (status == 2){
        //Sj = U.sum(0);
        //logger::item_start("bip enc i neq j prod_factorial");
        //prod_afac = prod_factorial(a, t+1, j); // the product of a_k! for t + 1 <= k <= j
        //logger::item_stop("bip enc i neq j prod_factorial");

        //logger::item_start("bip enc i neq j compute_product");
        //rtj = compute_product(St_stack[call_size-1], St_stack[call_size-1] - Sj, 1) / prod_afac;
        //logger::item_stop("bip enc i neq j compute_product");

        //logger::item_start("bip enc i neq j arithmetic");
        Nit_rtj = return_stack[return_size-2].first * r_stack[return_size-1];
        lit_Ntj = return_stack[return_size-2].second * return_stack[return_size-1].first;
        return_stack[return_size-2].first = Nit_rtj + lit_Ntj; // Nij
        return_stack[return_size-2].second = return_stack[return_size-2].second * return_stack[return_size-1].second; // lij
        r_stack[return_size - 2] = r_stack[return_size-2] * r_stack[return_size-1];
        //logger::item_stop("bip enc i neq j arithmetic");
        return_size --; // pop 2 add 1
        call_size --;
      }
      //logger::item_stop("bip enc i neq j");
    }

  }
  if (return_size != 1){
    cerr << " error: bip compute_N return_size is not 1 it is " << return_size << endl;
  }
  return return_stack[0];
}
```

#### 6.3.3.2 encode()

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

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

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

  //init(G); // initialize U and beta for G
  //pair<mpz_class, mpz_class> ans = compute_N(0,G.nu_left_vertices()-1, G);
  //init(G);
  //logger::item_start("bip enc compute N");
  //pair<mpz_class, mpz_class> ans = compute_N_new(G);
  //logger::item_stop("bip enc compute N");

  init(G);
  //logger::item_start("bip enc compute N new r");
  pair<mpz_class, mpz_class> ans = compute_N(G);
  //logger::item_stop("bip enc compute N new r");
  // if (ans.first == ans2.first and ans.second == ans2.second){
  //   cout << " = " << endl;
  // }else{
  //   cout << " != " << endl;
  // }
  //if (ans.first!= ans_2.first or ans.second != ans_2.second){
  //  cerr << " bip ans != ans_2 ans = (" << ans.first << " , " << ans.second <<  " ) ans_2 = (" << ans_2.first << " , " << ans_2.second << " ) "  << endl;
  //}//else{
  //cerr << " the same! ans = (" << ans.first << " , " << ans.second <<  " ) ans_2 = (" << ans_2.first << " , " << ans_2.second << " ) "  << endl;
```

```
  //}
  //mpz_class prod_b_factorial = prod_factorial(b, 0, b.size()-1); // \prod_{i=0}^{n-1} b_i

  //if (prod_b_factorial != ans.second)
  //  cerr << "EEEEEEEEEEEEEEEEEEEEEEEEE prod_b_factorial != ans.second" << endl;

  bool ceil = false;
  //logger::item_start("bip enc ceil");
  if (ans.first % ans.second != 0)
    ceil = true;
  //logger::item_stop("bip enc ceil");

  //logger::item_start("bip enc final div");
  ans.first /= ans.second;
  //logger::item_stop("bip enc final div");
  if (ceil)
    ans.first ++;
  return ans.first;
}
```

#### 6.3.3.3 init()

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

initializes beta and U

```
{
  // initializing beta
  beta = G.get_right_degree_sequence();

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

  if (a != G.get_left_degree_sequence() or b != G.get_right_degree_sequence())
    cerr << " WARNING b_graph_encoder::init : vectors a and/or b do not match with the degree sequences of the given bipartite graph  " << endl;

}
```

### 6.3.4 Member Data Documentation

#### 6.3.4.1 a

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

the degree sequence for the left nodes

#### 6.3.4.2 b

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

the degree sequence for the right nodes

#### 6.3.4.3 beta

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

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

#### 6.3.4.4 U

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

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

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

- bipartite_graph_compression.h
- bipartite_graph_compression.cpp

## 6.4 bit_pipe Class Reference

A sequence of arbitrary number of bits.

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

### Public Member Functions

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

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

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

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

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

  *return the number of chunks*
- int residue () const

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

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

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

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

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

## Private Attributes

- vector< unsigned int > bits

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

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

## Friends

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

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

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

    *shifts bits in B n bits to the right*

### 6.4.1 Detailed Description

A sequence of arbitrary number of bits.

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

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

### 6.4.2 Constructor & Destructor Documentation

#### 6.4.2.1 bit_pipe() [1/3]

```
bit_pipe::bit_pipe ( )  [inline]
```

```
26 {bits.resize(0); last_bits = 0;}
```

#### 6.4.2.2 bit_pipe() [2/3]

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

constructor given an integer

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

```
{
    bits.resize(1);
    bits[0] = n;
    last_bits = nu_bits(n);
    bits[0] <<= BIT_INT - last_bits; // so that n appears in the MSB place
}
```

#### 6.4.2.3 bit_pipe() [3/3]

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

constructor given an mpz_class object

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

}
```

### 6.4.3 Member Function Documentation

#### 6.4.3.1 append_left()

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

append B to the left of me

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

```
{
  if (B.size() == 0) // nothing should be done, B is empty
    return;
  int B_res = B.residue(); // number of incomplete bits in B
  if (B_res == BIT_INT){
    // B has complete chunks, so I just need to insert chunks of B at the beginning of my chunks
    bits.insert(bits.begin(), B.chunks().begin(), B.chunks().end());
    return; // all set!
  }
  // B has a residue
  // so I need to shift myself to the right and then append
  shift_right(B_res);
  // then, my leftmost chunk must be combined with the rightmost chunk of B:
  bits[0] |= B[B.size()-1];
  // then insert all but the rightmost chunk of B at my left
  bits.insert(bits.begin(), B.chunks().begin(), B.chunks().end()-1);
}
```

#### 6.4.3.2 chunks()

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

returns const reference to the bit sequence (object bits)

```
{return bits;}
```

#### 6.4.3.3 operator[]() [1/2]

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

returns a reference to the nth chunk

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

#### 6.4.3.4 operator[]() [2/2]

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

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

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

#### 6.4.3.5 residue()

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

returns the number of residual bits in the last chunk

```
{return last_bits;}
```

#### 6.4.3.6 shift_left()

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

shift everything n bits to the left

```
                              {
  if (n < 0){
    cerr << " ERROR: bit_pipe::shift_left called for negative value " << n << endl;
    return;
  }
  if (n >= BIT_INT){
    // we need to remove a number of bytes
    int bytes_to_remove = n / BIT_INT; // these many bytes must be remove
    bits.erase(bits.begin(), bits.begin() + bytes_to_remove);
    n = n % BIT_INT;
  }
  if (n == 0)
    return;

  // when we reach at this line, we have 1 <= n <= 7
  unsigned int mask = mask_gen(n) << (BIT_INT-n); // n bits in MSB for carryover masking
  unsigned int carry; // carryover to the left byte
  for (int i=0;i<bits.size(); i++){
    carry = (mask & bits[i]) >> (BIT_INT-n); // bring it to the right
    if ( i> 0)
      bits[i-1] |= carry; // add carry to the left guy
    bits[i] <<= n;
  }

  // now, deal with last_bits
  last_bits -= n;
  if (last_bits <= 0){
    // means that the rightmost byte must vanish
    last_bits += BIT_INT;
    bits.pop_back(); // remove the last byte
  }
}
```

#### 6.4.3.7 shift_right()

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

shifts n bits to the right.

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

```
{
  if (n == 0)
    return; // nothing to do
  if (n >= BIT_INT){
    bits.insert(bits.begin(), n / BIT_INT, 0); // n/BIT_INT bytes each zero will be added
    shift_right(n%BIT_INT);
    return;
  }
  // when we arrive at this line, n must be strictly less than BIT_INT and strictly bigger than zero, i.e. 0 < n < BIT_INT
  unsigned int mask = mask_gen(n); // mask is going to be n many ones (in LSB), e.g. if n = 3, mask is 00000111, this is useful in carrying over LSB of left bytes to the right bytes
  unsigned int carry_current = 0; // carry over of left bytes to the right. For instance, if we want to shift 11111111 3 bits to the right, it becomes 00011111 but a carry over 111 must be added to the byte to the right. This is initially zero
  unsigned int carry_prev = 0; // the same concept, but for the previous byte (to the left of me).
  for (int i=0;i<bits.size();i++){
    carry_current = bits[i] & mask; // find carryover bits for current byte
    bits[i] >>= n; // shift the current byte
    carry_prev <<= (BIT_INT-n); // put the previous carryover bits in place to be added to the current byte
    bits[i] |= carry_prev; // add the carryover to the current byte
    carry_prev = carry_current; // the current byte is the previous byte for the next byte
  }

  if (n > (BIT_INT - last_bits)){
    // the LSB bits of the last chunk must fall into a new chunk, so I should push_back a new chunk, which is zero for now
    carry_prev <<= (BIT_INT-n);
    bits.push_back(carry_prev); // the last byte is the last carryover shifted to the left
  }
  last_bits += n;
  if (last_bits > BIT_INT)
    last_bits -= BIT_INT;
}
```

#### 6.4.3.8 size()

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

return the number of chunks

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

### 6.4.4 Friends And Related Function Documentation

#### 6.4.4.1 ibitstream

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

#### 6.4.4.2 obitstream

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

#### 6.4.4.3 operator<< [1/2]

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

write to the output

```
{
  if (B.bits.size()==0){
    o << "<>";
    return o;
  }
  o << "<";
  for (int i=0;i<(B.bits.size()-1); i++){ // the last byte requires special handling
    bitset<BIT_INT> b(B.bits[i]);
    o << b << " ";
  }
  unsigned int last_byte = B.bits[B.bits.size()-1];

  for (int k=BIT_INT;k>(BIT_INT-B.last_bits);k--){ // starting from MSB bit to LSB for existing bits
    if (last_byte & (1<<(k-1)))
      o << "1";
    else
      o << "0";
  }
  o << "|"; // to show the place of the last bit
  for (int k=BIT_INT-B.last_bits; k>=1; k--){
    if (last_byte &(1<<(k-1)))
      o << "1";
    else
      o << "0";
  }
  o << ">";
  return o;
}
```

#### 6.4.4.4 operator<< [2/2]

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

shifts bits in B n bits to the left

```
{
  bit_pipe ans = B;
  ans.shift_left(n);
  return ans;
}
```

#### 6.4.4.5 operator>>

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

shifts bits in B n bits to the right

```
{
  bit_pipe ans = B;
  ans.shift_right(n);
  return ans;
}
```

### 6.4.5 Member Data Documentation

#### 6.4.5.1 bits

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

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

#### 6.4.5.2 last_bits

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

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

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

- bitstream.h
- bitstream.cpp

## 6.5 colored_graph Class Reference

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

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

## Public Member Functions

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

    *constructor from a graph, depth and maximum degree parameters*
- colored_graph ()

    *default constructor*
- void init (const marked_graph &G)

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

## Public Attributes

- int h

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

    *the maximum degree threshold*
- graph_message M

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

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

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

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

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

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

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

    *the list of all distinct vertex types, obtained from the ver_type array.*
- vector$<$ int $>$ ver_type_int

    *vertex type converted to integers, using the ver_type_dict map, i.e. ver_type_int[v] = ver_type_dict[ver_type[v]]*
- vector$<$ bool $>$ is_star_vertex

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

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

### 6.5.1 Detailed Description

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

quick member overview:

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

**Sample Usage**

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

### 6.5.2 Constructor & Destructor Documentation

#### 6.5.2.1 colored_graph() [1/2]

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

constructor from a graph, depth and maximum degree parameters

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

#### 6.5.2.2 colored_graph() [2/2]

colored_graph::colored_graph ( ) [inline]

default constructor

```
{}
```

### 6.5.3 Member Function Documentation

#### 6.5.3.1 init()

void colored_graph::init (
const marked_graph & G )

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

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

```
{
  logger::add_entry("colored_graph::init init", "");
  nu_vertices = G.nu_vertices;
  //adj_location = G.adj_location; // neighborhood structure is the same as the given graph
  index_in_neighbor = G.index_in_neighbor;

  // assigning edge colors based on the messages given by M
  //M.update_messages();
  adj_list.resize(nu_vertices);

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

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

  // implement and update deg and type_vertex_list

  int m, mp; // pair of types

  logger::add_entry("Find deg and ver_types", "");
  deg.resize(nu_vertices);
  is_star_vertex.resize(nu_vertices);
  ver_type.resize(nu_vertices);
  ver_type_int.resize(nu_vertices);
```

```
  for (int v=0;v<nu_vertices;v++){
    is_star_vertex[v] = false; // it is false unless we figure out otherwise, see below
    for (int i=0;i<adj_list[v].size(); i++){
      m = adj_list[v][i].second.first;
      mp = adj_list[v][i].second.second;
      if (M.is_star_message[m] == false and M.is_star_message[mp] == false) {
        // this edge is not star type
        if (deg[v].find(pair<int, int>(m, mp)) == deg[v].end()){
          // this does not exist, so create it, since this is the first edge, its value must be 1
          deg[v][pair<int, int>(m, mp)] = 1;
          //type_vertex_list[pair<int, int>(m, mp)].push_back(v); // this must be done when we see the type (m, mp) for the first time here, so as to avoid multiple placing of v in the list
        }else{
          // the edge exists, we only need to increase it by one
          deg[v][pair<int, int>(m, mp)] ++;
        }
      }else{
        // this is a star type vertex
        is_star_vertex[v] = true;
      }
    }

    // check if it was star vertex
    if (is_star_vertex[v] == true)
      star_vertices.push_back(v);


    // now, we form the type of this vertex
    // the type of a vertex is a vector x as follows:
    // x[0] is the vertex mark of v
    // x[3k+1], x[3k+2], x[3k+3] = (m_k, mp_k, deg[v][(m_k, mp_k)]) where (m_k, mp_k) is the kt key present in the map deg[v]. Since deg[v] is a map, we read its elements in increasing order (lexicographic order for pairs (m, mp)), hence this list is on a 1-1 correspondence with the pair (\theta(v), D(v)) in the paper.
    vector<int> vt; // type of v
    vt.resize(1+3 * deg[v].size()); // motivated by the above explanation
    vt[0] = G.ver_mark[v]; // mark of v
    int k = 0; // current index of vt
    for (map<pair<int, int>, int>::iterator it = deg[v].begin(); it != deg[v].end(); it++){
      vt[++k] = it->first.first; // m
      vt[++k] = it->first.second; // mp
      vt[++k] = it->second;
    }

    ver_type[v] = vt;
    // find ver_type_int[v]
    if (ver_type_dict.find(vt) == ver_type_dict.end()){
      // this is a new type, so add it to the dictionary and the list
      ver_type_dict[vt] = ver_type_list.size();
      ver_type_list.push_back(vt);
    }
    ver_type_int[v] = ver_type_dict[vt];
  }
}
```

### 6.5.4 Member Data Documentation

#### 6.5.4.1 adj_list

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

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

#### 6.5.4.2 deg

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

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

#### 6.5.4.3 Delta

```
int colored_graph::Delta
```

the maximum degree threshold

#### 6.5.4.4 h

```
int colored_graph::h
```

the depth up to which look at edge types

#### 6.5.4.5 index_in_neighbor

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

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

#### 6.5.4.6 is_star_vertex

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

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

#### 6.5.4.7 M

```
graph_message colored_graph::M
```

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

#### 6.5.4.8 nu_vertices

```
int colored_graph::nu_vertices
```

the number of vertices in the graph.

#### 6.5.4.9 star_vertices

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

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

#### 6.5.4.10 ver_type

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

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

#### 6.5.4.11 ver_type_dict

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

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

#### 6.5.4.12 ver_type_int

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

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

#### 6.5.4.13 ver_type_list

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

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

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

- graph_message.h
- graph_message.cpp

## 6.6 fenwick_tree Class Reference

Fenwick tree class.

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

### Public Member Functions

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

### Private Attributes

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

### 6.6.1 Detailed Description

Fenwick tree class.

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

### 6.6.2 Constructor & Destructor Documentation

#### 6.6.2.1 fenwick_tree() [1/2]

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

default constructor

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

#### 6.6.2.2 fenwick_tree() [2/2]

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

constructor, which takes a vector of values and initializes

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

### 6.6.3 Member Function Documentation

#### 6.6.3.1 add()

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

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

**Parameters**

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

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

#### 6.6.3.2 size()

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

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

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

#### 6.6.3.3 sum()

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

returns the sum of values from 0 to k

**Parameters**

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

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

### 6.6.4 Member Data Documentation

#### 6.6.4.1 sums

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

a one based vector containing sum of values

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

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

- fenwick.h
- fenwick.cpp

## 6.7 graph Class Reference

simple unmarked graph

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

### Public Member Functions

- graph ()

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

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

  *constructor, given only the forward adjacency list*
- vector< int > get_forward_list (int v) const

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

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

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

  *returns the whole degree sequence*
- int nu_vertices () const

  *the number of vertices in the graph*

### Private Attributes

- int n

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

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

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

### Friends

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

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

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

  *comparing for inequality*

### 6.7.1 Detailed Description

simple unmarked graph

## 6.7.2 Constructor & Destructor Documentation

### 6.7.2.1 graph() [1/3]

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

default constructor

```
: n(0) {}
```

### 6.7.2.2 graph() [2/3]

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

a constructor

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

**Parameters**

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

```
                                                                        {
  n = list.size();
  forward_adj_list = list;
  // sorting the list
  for (int v=0; v<n; v++)
    sort(forward_adj_list[v].begin(), forward_adj_list[v].end());
  degree_sequence = deg;
}
```

### 6.7.2.3 graph() [3/3]

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

constructor, given only the forward adjacency list

This constructor only takes the forward adjacency list and computes the degree sequence itself

```
{
  n = list.size();
  forward_adj_list = list;


  // sorting the list
  for (int v=0; v<n; v++)
    sort(forward_adj_list[v].begin(), forward_adj_list[v].end());

  // finding the degree sequence
  // first, removing and resize it
  degree_sequence.clear();
  degree_sequence.resize(n);

  for (int v=0; v<n; v++){
    degree_sequence[v] += forward_adj_list[v].size(); // degree to the right
    for (int i=0;i<forward_adj_list[v].size();i++) // modifying degree of vertices to the
    right of v
      degree_sequence[forward_adj_list[v][i]]++;
  }
}
```

## 6.7.3 Member Function Documentation

### 6.7.3.1 get_degree()

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

returns the overall degree of a vertex

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

### 6.7.3.2 get_degree_sequence()

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

returns the whole degree sequence

```
{
  return degree_sequence;
}
```

#### 6.7.3.3 get_forward_degree()

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

returns the forward degree of a vertex v

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

#### 6.7.3.4 get_forward_list()

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

returns the forward adjacency list of a given vertex

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

#### 6.7.3.5 nu_vertices()

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

the number of vertices in the graph

```
{
  return n;
}
```

### 6.7.4 Friends And Related Function Documentation

#### 6.7.4.1 operator"!=

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

comparing for inequality

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

#### 6.7.4.2 operator<<

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

printing the graph to the output

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

#### 6.7.4.3 operator==

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

comparing two graphs for equality

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

### 6.7.5 Member Data Documentation

#### 6.7.5.1 degree_sequence

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

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

#### 6.7.5.2 forward_adj_list

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

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

#### 6.7.5.3 n

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

the number of vertices in the graph

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

- simple_graph.h
- simple_graph.cpp

## 6.8 graph_decoder Class Reference

Decodes a simple unmarked graph.

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

### Public Member Functions

- graph_decoder (vector< int > a_)

  *constructor given the degree sequence*
- void init ()

  *initializes x to be empty vector of size n, and U and beta by a*
- graph decode (mpz_class f, vector< int > tS_)

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

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

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

### Private Attributes

- vector< int > a

  *the degree sequence of the graph.*
- int n

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

  $\lfloor \log_2 n \rfloor^2$
- vector< vector< int > > x

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

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

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

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

### 6.8.1 Detailed Description

Decodes a simple unmarked graph.

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

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

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

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

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

```
vector<int> a = {3,2,2,3}; // degree sequence

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

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

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

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

### 6.8.2 Constructor & Destructor Documentation

#### 6.8.2.1 graph_decoder()

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

constructor given the degree sequence

```
{
  a = a_;
  n = a.size();

  int log2n = 0; // log of n in base 2
  int nn = n; // a copy of n
  while (nn>0){
    log2n ++;
    nn = nn >> 1; // divide by 2
  } // eventually, we count the number of bits in n
  log2n --; // we count extra, e.g. when n = 1, we end up having 1, rather than 0
  logn2 = log2n * log2n;

  init(); // init x, beta and U
}
```

### 6.8.3 Member Function Documentation

#### 6.8.3.1 decode()

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

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

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

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

#### 6.8.3.2 decode_interval()

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

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

**Parameters**

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

**Returns**

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

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

  // sweeping for zero nodes
```

```

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

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

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


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

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

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

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

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

#### 6.8.3.3 decode_node()

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

decode the node i

**Parameters**

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

**Returns**

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

```
{
```

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

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

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

#### 6.8.3.4 init()

```
void graph_decoder::init ( )
```

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

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

### 6.8.4 Member Data Documentation

#### 6.8.4.1 a

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

the degree sequence of the graph.

#### 6.8.4.2 beta

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

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

#### 6.8.4.3 logn2

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

$\lfloor \log_2 n \rfloor^2$

#### 6.8.4.4 n

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

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

#### 6.8.4.5 tS

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

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

#### 6.8.4.6 U

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

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

#### 6.8.4.7 x

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

the forward adjacency list of the decoded graph

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

- simple_graph_compression.h
- simple_graph_compression.cpp

## 6.9 graph_encoder Class Reference

Encodes a simple unmarked graph.

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

### Public Member Functions

- graph_encoder (const vector< int > &a_)
  *constructor*
- void init (const graph &G)
  *initializes beta and U, clears Stilde for a fresh use*
- pair< mpz_class, mpz_class > compute_N (const graph &G)
  *computes* $N(G)$
- pair< mpz_class, vector< int > > encode (const graph &G)
  *Encodes the graph and returns N together with Stilde.*

### Private Attributes

- int n
  *the number of vertices*
- vector< int > a
  *the degree sequence*
- vector< int > beta
  *When compute_N is called for* $i \leq j$*, for* $i \leq v \leq n$*, we have* $\beta_v = d_v(i)$*.*
- reverse_fenwick_tree U
  *a Fenwick tree which encodes the forward degrees to the right. When compute_N is called for* $i \leq j$*, for* $i \leq v$*, we have* $U_v = \sum_{k=v}^{n} d_k(i)$*.*
- vector< int > Stilde
  *Summation of forward degrees at* $n / \log^2 n$ *many points.*
- int logn2
  $\lfloor \log_2 n \rfloor^2$ *where* $n$ *is the number of vertices*

### 6.9.1 Detailed Description

Encodes a simple unmarked graph.

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

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

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

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

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

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

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

### 6.9.2 Constructor & Destructor Documentation

#### 6.9.2.1 graph_encoder()

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

constructor

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

```
                                                   {
  a = a_;
  n = a.size();
  int log2n = 0; // log of n in base 2
  int nn = n; // a copy of n
  while (nn>0){
    log2n ++;
    nn = nn >> 1; // divide by 2
  } // eventually, we count the number of bits in n
  log2n --; // we count extra, e.g. when n = 1, we end up having 1, rather than 0
  logn2 = log2n * log2n;

  Stilde.clear();
  Stilde.resize(4 * n / logn2); // look at the explanation of the algorithm, before deriving
    the bound 16 n, that 4n / \lfloor \log_2 n \rfloor^2 is also an upper bound. After this point, we have used
    \lfloor \log_2 n \rfloor \geq \log n / 2 to derive the 16 n bound.
  Stilde[0] = 0;
}
```

### 6.9.3 Member Function Documentation

#### 6.9.3.1 compute_N()

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

computes $N(G)$

#### Parameters

| | |
|---|---|
| *G* | the reference to the simple graph G |

#### Returns

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

```
{

  int n = G.nu_vertices();
  int n_bits = 0;
  int n_copy = n;
  while (n_copy > 0){
    n_bits ++;
    n_copy >>= 1;
  }
  n_bits += 2;

  vector<pair<pair<int, int>, int > > call_stack(2 * n_bits);
  vector<pair<mpz_class, mpz_class> > return_stack(2 * n_bits); // first = N, second = l
  vector<mpz_class> r_stack(2 * n_bits); // stack of r values
  vector<int> status_stack(2 * n_bits);
  //vector<int> St_stack(2 * n_bits); // stack to store values of St

  call_stack[0].first = pair<int, int> (0,n-1); // i j
  call_stack[0].second = 1; // I
  status_stack[0] = 0; // newly added

  int call_size = 1; // the size of the call stack
  int return_size = 0; // the size of the return stack

  int i, j, I, t, Sj;
  int status;

  vector<int> gamma; // forward list  of the graph
  mpz_class zik, lik, rtj; // intermediate variables
  mpz_class Nit_rtj; // result of Nij * rtj
  mpz_class lit_Ntj; // result of lit * Ntj
  while (call_size > 0){
    // cerr << " printing the whole stack " << endl;
    // for (int k = 0; k<call_size; k++){
    //   cerr << k << " : " << call_stack[k].first.first << " " << call_stack[k].first.second << " I " << call_stack[k].second << "s= " << status_stack[k] << endl;
    // }
    // cerr << " return stack " << endl;
    // for (int k=0;k<return_size;k++)
    //   cerr << k << ": " << return_stack[k].first << " " << return_stack[k].second << endl;
    i = call_stack[call_size-1].first.first;
    j = call_stack[call_size-1].first.second;
    I = call_stack[call_size-1].second;
    if (i==j){
      //logger::item_start("sim enc i = j");
      return_stack[return_size].first = 0; // z_i is initialized with 0
      return_stack[return_size].second = 1; // l_i is initialize with 1
      gamma = G.get_forward_list(i); // the forward adjacency list of vertex i
       r_stack[return_size] = compute_product(U.sum(i) - 1, beta[i], 2); // this is r_i
       // cerr << " i " << i << " j " << j << " gamma: " << endl;
       // for (int k=0;k<gamma.size();k++)
       //   cerr << gamma[k] << " ";
       // cerr << endl;
       // cerr << " beta " << endl;
       // for (int k=0; k<n; k++)
       //   cerr << beta[k] << " ";
       // cerr << endl;

       for (int k=0;k<gamma.size();k++){
         zik = compute_product(U.sum(1+gamma[k]), beta[i] - k, 1); // we are zero based here, so instead of -k + 1, we have -k
         //zik = helper_vars::return_stack[0];
         //cerr << " zik " << zik << endl;
         return_stack[return_size].first += return_stack[return_size].second * zik;
         lik = (beta[i] - k) * beta[gamma[k]]; // we are zero based here, so instead of -k + 1, we have -k
         //cerr << " lik " << lik << endl;
         return_stack[return_size].second *= lik;
         beta[gamma[k]] -- ;
         U.add(gamma[k],-1);
       }
       return_size ++; // establish the return
```

```
      call_size --;
      //logger::item_stop("sim enc i = j");
    }else{
      status = status_stack[call_size-1];
      if (status == 0){
        // newly added node, we should call its left child
        t = (i+j) / 2;
        call_stack[call_size].first.first = i;
        call_stack[call_size].first.second = t;
        call_stack[call_size].second = 2*I;
        status_stack[call_size-1] = 1; // left is called
        status_stack[call_size] = 0; // newly added
        call_size++;
      }
      if (status == 1){
        // left is returned
        t = (i+j) / 2;
        //St_stack[call_size-1] = U.sum(t+1);
        if (j - i > logn2)
          Stilde[I] = U.sum(t+1);//St_stack[call_size-1];
        // prepare to call right

        call_stack[call_size].first.first = t + 1;
        call_stack[call_size].first.second = j;
        call_stack[call_size].second = 2*I + 1;
        status_stack[call_size-1] = 2; // right is called
        status_stack[call_size] = 0; // newly called
        call_size ++;
      }

      if (status == 2){
        // both are returned, and results can be accessed by the top two elements in return stack
        //Sj = U.sum(j+1);
        //logger::item_start("sim enc i neq j compute_product");
        //rtj = compute_product(St_stack[call_size-1]-1, (St_stack[call_size-1] - Sj)/2, 2);
        //logger::item_stop("sim enc i neq j compute_product");
        //rtj = helper_vars::return_stack[0];
        //cerr << " rtj " << rtj << endl;
        //Nij = Nit * rtj + lit * Ntj ;
        //logger::item_start("sim enc i neq j arithmetic");
        Nit_rtj = return_stack[return_size-2].first * r_stack[return_size-1];
        lit_Ntj = return_stack[return_size-2].second * return_stack[return_size-1].first;
        return_stack[return_size-2].first = Nit_rtj + lit_Ntj; // Nij
        return_stack[return_size-2].second = return_stack[return_size-2].second * return_stack[return_size-1].second; // lij
        r_stack[return_size-2] = r_stack[return_size-2] * r_stack[return_size-1];
        //logger::item_stop("sim enc i neq j arithmetic");
        return_size --; // pop 2 add 1
        call_size --;
      }
    }

  }

  if (return_size != 1){
    cerr << " error: return_size is not 1 it is " << return_size << endl;
  }
  return return_stack[0];
}
```

#### 6.9.3.2 encode()

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

Encodes the graph and returns N together with Stilde.

**Parameters**

| | |
|---|---|
| *G* | reference to the graph to encode |

**Returns**

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

```
{
  if (G.get_degree_sequence()!= a)
    cerr << " WARNING graph_encoder::encode : vector a does not match with the degree sequence of the given graph ";
  //init(G); // initialize U and beta
  //pair<mpz_class, mpz_class> N_ans  = compute_N(0,G.nu_vertices()-1,1, G);
  init(G); // re initializing U abd beta for the second test
  pair<mpz_class, mpz_class> N_ans = compute_N(G);
  // init(G);
  // pair<mpz_class, mpz_class> N_ans_2 = compute_N(G);
  // if (N_ans.first == N_ans_2.first and N_ans.second == N_ans_2.second){
  //   cerr << " = " << endl;
  // }else{
  //   cerr << " error N_ans and N_ans_2 are not the same, "  << endl << "N_ans = (" << N_ans.first << " , " << N_ans.second << ") " << endl << "N_ans_2 = (" << N_ans_2.first << " , " << N_ans_2.second << ")" << endl;
  // }
  //if (N_ans.first != N_ans_2.first or N_ans.second != N_ans_2.second)
  //  cerr << " error N_ans and N_ans_2 are not the same, "  << endl << "N_ans = (" << N_ans.first << " , " << N_ans.second << ") " << endl << "N_ans_2 = (" << N_ans_2.first << " , " << N_ans_2.second << ")" << endl;
  //else
  //  cerr << " N_ans = N_ans_2 " << endl;
  //mpz_class prod_a_factorial = prod_factorial(a, 0,a.size()-1); // \prod_{i=1}^n a_i!
  //if (prod_a_factorial!= N_ans.second)
  //  cerr << " ERROR: not equal " << endl;
  // N_ans.second = \prod_{i=1}^n a_i!
  // we need the ceiling of the ratio of N_ans.first and prod_a_factorial
  bool ceil = false; // if true, we will add one to the integer division
  //logger::item_start("simple_ar");
  if (N_ans.first % N_ans.second != 0)
    ceil = true;
  N_ans.first /= N_ans.second;
  if (ceil)
    N_ans.first ++;
  //logger::item_stop("simple_ar");
  return pair<mpz_class, vector<int> > (N_ans.first, Stilde);
}
```

#### 6.9.3.3 init()

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

initializes beta and U, clears Stilde for a fresh use

```
{
  // initializing the beta sequence
  beta = a;

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

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


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

### 6.9.4 Member Data Documentation

#### 6.9.4.1 a

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

the degree sequence

#### 6.9.4.2 beta

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

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

#### 6.9.4.3 logn2

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

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

#### 6.9.4.4 n

```
int graph_encoder::n [private]
```

the number of vertices

#### 6.9.4.5 Stilde

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

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

#### 6.9.4.6 U

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

a Fenwick tree which encodes the forward degrees to the right. When compute_N is called for $i \leq j$, 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

## 6.10 graph_message Class Reference

this class takes care of message passing on marked graphs.

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

### Public Member Functions

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

  *constructor, given reference to a graph*
- graph_message ()

  *default constructor*

### Public Attributes

- vector< vector< int > > messages

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

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

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

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

### Private Member Functions

- void update_messages (const marked_graph &)

  *performs the message passing algorithm and updates the messages array accordingly*
- void send_message (const vector< int > &m, int v, int i)

  *update message_dict and message_list*

## Private Attributes

- int h

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

  *the maximum degree threshold*

### 6.10.1 Detailed Description

this class takes care of message passing on marked graphs.

This graph has a reference to a marked_graph object for which we perform message passing to find edge types. The edge types are discovered up to depth h-1, and with degree parameter Delta, where h and Delta are member objects. Star type messages (which roughly speaking corresponds to places where there is a vertex in the h neighborhood has degree more than delta) are vectors of size 2, first coordinate being -1, and the second being the edge mark component (towards the 'me' vertex).

*Sample Usage*

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

### 6.10.2 Constructor & Destructor Documentation

#### 6.10.2.1 graph_message() [1/2]

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

constructor, given reference to a graph

```
                                                                        {
    h = depth;
    Delta = max_degree;
    update_messages(graph); // do message passing
  }
```

#### 6.10.2.2 graph_message() [2/2]

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

default constructor

```
{}
```

### 6.10.3 Member Function Documentation

#### 6.10.3.1 send_message()

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

update message_dict and message_list

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

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

**Parameters**

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

```
{
  unordered_map<vector<int>, int, vint_hash>::iterator it;
  /*
  cerr << " send message (";
  for (int k=0;k<m.size();k++){
    cerr << m[k];
    if (k<m.size()-1)
      cerr << ", ";
   }
  cerr << "): " << v << " -> " << i;
  */

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

#### 6.10.3.2 update_messages()

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

performs the message passing algorithm and updates the messages array accordingly

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

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

```
{
  logger::current_depth++;
  logger::add_entry("graph_message::update_message init", "");
  int nu_vertices = G.nu_vertices;
  int w;
  int my_location;
  messages.resize(nu_vertices);
  //inward_message.resize(nu_vertices);
  //message_dict.resize(h);
  //message_list.resize(h);
  //message_mark.resize(h);
  //is_star_message.resize(h);


  // initialize the messages

  logger::add_entry("resizing messages", "");
  for (int v=0;v<nu_vertices;v++){
    messages[v].resize(G.adj_list[v].size());
    //inward_messages[v].resize(G.adj_list[v].size());
    //for (int i=0;i<G.adj_list[v].size();i++){
    //messages[v][i].resize(h);
      //inward_messages[v][i].resize(h);
    //}
  }

  logger::add_entry("initializing messages","");
  vector<int> m(3);
  unordered_map<vector<int>, int, vint_hash>::iterator it;
  //map<vector<int>, int>::iterator it;

  for (int v=0;v<nu_vertices;v++){

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


      //vector<int> m;
      //m.clear();
      //m.push_back(G.ver_mark[v]);
      //m.push_back(0);
      //m.push_back(G.adj_list[v][i].second.first);

      m[0] = G.ver_mark[v];
      m[1] = 0;
      m[2] = G.adj_list[v][i].second.first;
      send_message(m, v, i);

      // adding this message to the message dictionary
      //it = message_dict[0].find(m);
      //w = G.adj_list[v][i].first;

      /*
      if (it == message_dict[0].end()){
        message_dict[0][m] = message_list[0].size();
        messages[v][i][0] = message_list[0].size();
        message_list[0].push_back(m);

      }else{
```

```
      messages[v][i][0] = it->second;
    }
    */
    //messages[v][i][0] = message_dict[0][m]; // the message at time 0
  }
}

// these are copies of message_dict, message_mark and is_star_message at the previous step, which are used to update messages at the current step.

//unordered_map<vector<int>, int, vint_hash> message_dict_old;
//vector<int> message_mark_old;
vector<bool> is_star_message_old;
vector<vector<int> > messages_old;

// updating messages
logger::add_entry("updating messages", "");
m.reserve(5+ 2 * Delta);
 vector<int> m2;
 m2.reserve(5 + 2*Delta); // an auxiliary message when we need to work with two types of messages simultaneously
 duration<float> diff;
 high_resolution_clock::time_point t1, t2;
 float agg_search = 0;
 float agg_insert = 0;
 float agg_m = 0;
 float agg_sort = 0;
 float agg_neigh_message = 0;

 vector<pair<pair<int, int>, int> > neighbor_messages; // the first component is the message and the second is the name of the neighbor
 // the second component is stored so that after sorting, we know the owner of the message
 neighbor_messages.reserve(5+2*Delta);

 int nu_star_neigh; // number of star neighbors, i.e. neighbors of a vertex v whose message towards v are star type
 int star_neigh_index; // the index of the star neighbor of v, this is only useful when there is one star neighbor, if there are more than one star neighbor, then the message sent from v towards all other neighbors are star typed
 int star_neigh; // the label of the star neighbor, i.e. star_neigh = G.adj_list[v][star_neigh_index].first;
 int previous_message; // the message from the previous step
 int mark_to_v; // mark towards the current vertex directed from its neighbor
 vector<int> neighbors_list; // the list of neighbors of a vertex in the order after sorting with respect to their corresponding messages
 neighbors_list.reserve(Delta + 3);
 int deg_v; // the degree of vertex v



 for (int s=1; s<h;s++){ // s stands for step
   //cerr << endl << endl<< " depth " << s << endl;
   // store variables corresponding to the previous step in their old version, and clearing the variables for this step:
   //message_dict_old = message_dict;
   messages_old = messages;
   //message_mark_old = message_mark;
   is_star_message_old = is_star_message;
   message_dict.clear();
   message_mark.clear();
   is_star_message.clear();
   // we do not clear messages since we need its size to be the same, and we only modify its content


   for (int v=0;v<nu_vertices;v++){
     deg_v = G.adj_list[v].size();
     if (deg_v==1){
       // no need to collect messages, there is only one message towards the one neighbor, which is known
       m.resize(3); // there is only one message which is of the form \f$(\theta, 0, x)\f$, where \f$\theta\f$ is the mark of v, and \f$x = \xi_G(w,v)\f$  where \f$w\f$ is the only neighbor of v
       m[0] = G.ver_mark[v];
       m[1] = 0;
       m[2] = G.adj_list[v][0].second.first;
       send_message(m, v, 0);
     }else{
       if (deg_v <= Delta){
         neighbor_messages.clear();
         nu_star_neigh = 0;
         for (int i=0;i<deg_v;i++){
           w = G.adj_list[v][i].first; // neighbor label
           my_location = G.index_in_neighbor[v][i];
           previous_message = messages_old[w][my_location]; // the message sent from this neighbor towards v at time t-1
           // check if previous message is star
           if (is_star_message_old[previous_message]){
             nu_star_neigh ++;
             star_neigh_index = i;
```

```
        star_neigh = w;
      }
      if (nu_star_neigh >= 2)
        break; // then message towards all neighbors will be star, no need to collect messages
      mark_to_v = G.adj_list[v][i].second.first;
      neighbor_messages.push_back(pair<pair<int, int> , int> (pair<int,int>(previous_message,
mark_to_v), i));
    }
    if (nu_star_neigh == 2){
      // message towards all the neighbors will be star
      m.resize(2);
      m[0] = -1;
      for (int i=0;i<G.adj_list[v].size();i++){
        m[1] = G.adj_list[v][i].second.first;
        send_message(m, v, i); // send message m from v towards its ith neighbor at step
 s
      }
    }
    if (nu_star_neigh == 1){
      // the message towards all the neighbors except for that star neighbor is star
      // let m be the message towards that neighbor and m2 be the star messages

      // sorting neighbor messages
      sort(neighbor_messages.begin(), neighbor_messages.end());

      // preparing m
      m.resize(0);
      m.push_back(G.ver_mark[v]);
      m.push_back(G.adj_list[v].size()-1);



      for (int i=0;i<neighbor_messages.size();i++){
        if (neighbor_messages[i].second != star_neigh_index){
          // collect the messages of non star neighbors
          m.push_back(neighbor_messages[i].first.first);
          m.push_back(neighbor_messages[i].first.second);
        }
      }
      // finalize m by inserting its mark component
      m.push_back(G.adj_list[v][star_neigh_index].second.first);

      // prepare star messages
      m2.resize(2);
      m2[0] = -1;
      for (int i=0;i<deg_v;i++){
        if (i==star_neigh_index){
          // send the prepared message m
          send_message(m, v, i);
        }else{
          // prepare a star message and send it
          m2[1] = G.adj_list[v][i].second.first;
          send_message(m2, v, i);
        }
      }
    }

    if (nu_star_neigh == 0){
      // no star neighbor, so we can prepare messages to all neighbors comfortably as none of them
 are star type
      // we do this by a masking technique
      // sorting neighbor messages
      sort(neighbor_messages.begin(), neighbor_messages.end());
      if (neighbor_messages.size() != deg_v){
        cerr << " Error: no star messages and yet neighbor_messages does not have a size equal to the
 deg of v, step " << s << " v= " << v <<  " deg_v= " << deg_v << " neighbor_messages.size() " <<
neighbor_messages.size() << endl;
      }
      m.resize(1 + 1 + 2*(G.adj_list[v].size()-1) +1); // 1 for vertex mark, 1 for deg -1,
 for (deg-1) many neighbors, each we have 2 values, and finally 1 for the mark component
      m[0] = G.ver_mark[v];
      m[1] = G.adj_list[v].size()-1;
      neighbors_list.resize(deg_v);
      for (int i=0;i<neighbor_messages.size();i++){
        m[2*(i+1)]   = neighbor_messages[i].first.first;
        m[2*(i+1)+1] = neighbor_messages[i].first.second;
        neighbors_list[i] = neighbor_messages[i].second;
      }
      // swapping the last message so that its mark component comes first, so that we can treat m as
 a valid message in our standard
      swap(m[2*deg_v], m[2*deg_v+1]);

      for (int i=neighbor_messages.size()-1;i>=0;i--){
        if (i < neighbor_messages.size()-1){
          swap(m[2*deg_v], m[2*(i+1)+1]);
          swap(m[2*deg_v+1], m[2*(i+1)]);
          swap(neighbors_list[deg_v-1], neighbors_list[i]);
```

```
            }
            send_message(m,v,neighbors_list[deg_v-1]);
          }
        }
      }
      if (deg_v > Delta){ // the message towards all neighbors is star
        m.resize(2);
        m[0] = -1;
        for(int i=0;i<deg_v;i++){
          m[1] = G.adj_list[v][i].second.first;
          send_message(m,v,i);
        }
      }
    }
  }
}


logger::add_entry("* symmetrizing", "");
bool star1, star2;
m.resize(2); // prepare for star message
m[0] = -1;
for (int v=0;v<nu_vertices;v++){
  for (int i=0;i<G.adj_list[v].size();i++){
    w = G.adj_list[v][i].first;
    my_location = G.index_in_neighbor[v][i];
    if (w > v){ // to avoid going over edges twice
      star1 = is_star_message[messages[v][i]];
      star2 = is_star_message[messages[w][my_location]];
      if (star1 and !star2){
        // message[w][my_location] should be star
        m[1] = G.adj_list[v][i].second.second;
        send_message(m,w,my_location);
      }
      if (!star1 and star2){
        // messages[v][i] should also become star
        m[1] = G.adj_list[v][i].second.first;
        send_message(m,v,i);
      }
      if ((!star1 and !star2) and (G.adj_list[v].size() > Delta or G.adj_list[w].size()>Delta)){ // this activates only when h = 1, ensures truncation of degrees bigger than delta
        // message[w][my_location] should be star
        m[1] = G.adj_list[v][i].second.second;
        send_message(m,w,my_location);
        // messages[v][i] should also become star
        m[1] = G.adj_list[v][i].second.first;
        send_message(m,v,i);
      }
    }
  }
}
logger::current_depth--;




// ==================================================
// ==================================================
// ==================================================
// ==================================================
// ==================================================
// ==================================================
// ==================================================
// ==================================================

/*
for (int t=1;t<h;t++){
  for (int v=0;v<nu_vertices;v++){
    //cerr << " vertex " << v << endl;
    if (G.adj_list[v].size() <= Delta){
      // the degree of v is no more than Delta
      // do the standard message passing by aggregating messages from neighbors
      // stacking all the messages from neighbors of v towards v
      neighbor_messages.clear();

      // 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
      //t1 = high_resolution_clock::now();
      for (int i=0;i<G.adj_list[v].size();i++){
        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
        //my_location = G.adj_location[w].at(v); <--- the inefficient way
        my_location = G.index_in_neighbor[v][i];
        // where is the place of node v among the list of neighbors of the ith neighbor of v
        int previous_message = messages[w][my_location][t-1]; // the message sent from this neighbor
```

```
towards v at time t-1
        int mark_to_v = G.adj_list[v][i].second.first;
        neighbor_messages.push_back(pair<pair<int, int> , int> (pair<int,int>(previous_message, mark_to_v), w));
      }
      //t2 = high_resolution_clock::now();
      //diff = t2 - t1;
      //agg_neigh_message += diff.count();

      //t1 = high_resolution_clock::now();
      sort(neighbor_messages.begin(), neighbor_messages.end()); // sorts lexicographically
      //t2 = high_resolution_clock::now();
      //diff = t2 - t1;
      //agg_sort += diff.count();

      for (int i=0;i<G.adj_list[v].size();i++){
        // let w be the current ith neighbor of v
        int w = G.adj_list[v][i].first;
        // first, start with the mark of v and the number of offsprings in the subgraph component of the message
        //vector<int> m; // the message that v is going to send to w
        //t1 = high_resolution_clock::now();
        m.clear();
        m.push_back(G.ver_mark[v]); // mark of v
        m.push_back(G.adj_list[v].size()-1); // the number of offsprings in the subgraph component of the message
        //t2 = high_resolution_clock::now();
        //diff = t2 - t1;
        //agg_m += diff.count();

        // stacking messages from all neighbors of v expect for w towards v at time t-1
        for (int j=0;j<G.adj_list[v].size();j++){
          if (neighbor_messages[j].second != w){
            if (message_list[t-1][neighbor_messages[j].first.first][0] == -1){
              // this means that one of the messages that should be aggregated is * typed, therefore the outgoing messages should also be * typed
              // i.e. the message has only two entries: (-1, \xi(w,v)) where \xi(w,v) is the mark of the edge between v and w towards v
              // since after this loop, the mark \xi(w,v) is added to the message (after the comment starting with 'finally'), we only add the initial -1 part
              //t1 = high_resolution_clock::now();
              m.resize(0);
              m.push_back(-1);
              //t2 = high_resolution_clock::now();
              //diff = t2 - t1;
              //agg_m += diff.count();
              break; // the message is decided, we do not need to go over any of the other neighbor messages, hence break
            }
            // this message should be added to the list of messages
            //t1 = high_resolution_clock::now();
            m.push_back(neighbor_messages[j].first.first); // message part
            m.push_back(neighbor_messages[j].first.second); // mark part towards v
            //t2 = high_resolution_clock::now();
            //diff = t2 - t1;
            //agg_m += diff.count();

          }
        }
        // if we break, we reach at this point and message is (-1), otherwise the message is of the form (\tau(v), \deg(v) - 1, ...) where ... is the list of all neighbor messages towards v except for w.
        // finally, the mark of the edge between v and w towards v, \xi(w,v), should be added to this list
        //t1 = high_resolution_clock::now();
        m.push_back(G.adj_list[v][i].second.first);
        //t2 = high_resolution_clock::now();
        //diff = t2 - t1;
        //agg_m += diff.count();

        // set the current message
        //t1 = high_resolution_clock::now();
        it = message_dict[t].find(m);
        //t2 = high_resolution_clock::now();
        //diff = t2 - t1;
        //agg_search += diff.count();

        if (it == message_dict[t].end()){
          //t1 = high_resolution_clock::now();
          //message_dict[t][m] = message_list[t].size();
          message_dict[t].insert(pair<vector<int>, int> (m, message_list[t].size()));
          //t2 = high_resolution_clock::now();
          //diff = t2 - t1;
          //agg_insert += diff.count();

          messages[v][i][t] = message_list[t].size();
          message_list[t].push_back(m);
        }else{
```

```
            messages[v][i][t] = it->second;
          }
        }
      }else{
        // if the degree of v is bigger than Delta, the message towards all neighbors is of the form *
        // i.e. message of v towards a neighbor w is of the form (-1, \xi(w,v)) where \xi(w,v) is the mark of the edge between v and w towards v
        for (int i=0;i<G.adj_list[v].size();i++){
          //vector<int> m; // the current message from v to ith neighbor
          //t1 = high_resolution_clock::now();
          m.clear();
          m.resize(2);
          m[0] = -1;
          m[1] = G.adj_list[v][i].second.first;
          //t2 = high_resolution_clock::now();
          //diff = t2 - t1;
          //agg_m += diff.count();

          // set the current message
          //t1 = high_resolution_clock::now();
          it = message_dict[t].find(m);
          //t2 = high_resolution_clock::now();
          //diff = t2 - t1;
          //agg_search += diff.count();

          if (it == message_dict[t].end()){
            //t1 = high_resolution_clock::now();
            //message_dict[t][m] = message_list[t].size();
            message_dict[t].insert(pair<vector<int>, int> (m, message_list[t].size()));
            //t2 = high_resolution_clock::now();
            //diff = t2 - t1;
            //agg_insert += diff.count();
            messages[v][i][t] = message_list[t].size();
            message_list[t].push_back(m);
          }else{
            messages[v][i][t] = it->second;
          }
        }
      }
    }
  }
  cerr << " total time to search in hash table: " << agg_search << endl;
  cerr << " total time to insert in hash table: " << agg_insert << endl;
  cerr << " total time to modify vector m  " << agg_m << endl;
  cerr << " total time to sort  " << agg_sort << endl;
  cerr << " total time to collect neighbor messages " <<  agg_neigh_message << endl;
  // 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)
  logger::add_entry("* symmetrizing", "");
  for (int v=0;v<nu_vertices;v++){
    for (int i=0;i<G.adj_list[v].size();i++){
      if (message_list[h-1][messages[v][i][h-1]][0] == -1){
        // it is of the form *
        w = G.adj_list[v][i].first; // the other endpoint of the edge
        //my_location = G.adj_location[w].at(v); // so that adj_list[w][my_location].first = v
        my_location = G.index_in_neighbor[v][i];

        //vector<int> m;
        m.clear();
        m.resize(2);
        m[0] = -1;
        m[1] = G.adj_list[v][i].second.second; // the mark towards w
        if (message_dict[h-1].find(m) == message_dict[h-1].end()){
          message_dict[h-1][m] = message_list[h-1].size();
          message_list[h-1].push_back(m);
        }
        messages[w][my_location][h-1] = message_dict[h-1][m];
      }
    }
  }

  // setting message_mark and is_star_message
  logger::add_entry("setting message_mark and is_star_message", "");
  message_mark.resize(message_list[h-1].size());
  is_star_message.resize(message_list[h-1].size());
  for (int i=0;i<message_list[h-1].size();i++){
    message_mark[i] = message_list[h-1][i].back(); // the last element is the mark component
    is_star_message[i] = (message_list[h-1][i][0] == -1); // message is star type when the first element is -1
  }
  logger::current_depth--;
  */
}
```

### 6.10.4 Member Data Documentation

#### 6.10.4.1 Delta

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

the maximum degree threshold

#### 6.10.4.2 h

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

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

#### 6.10.4.3 is_star_message

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

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

#### 6.10.4.4 message_dict

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

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

#### 6.10.4.5 message_mark

```
vector<int> graph_message::message_mark
```

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

#### 6.10.4.6 messages

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

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

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

- graph_message.h
- graph_message.cpp

## 6.11 ibitstream Class Reference

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

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

### Public Member Functions

- void read_chunk ()

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

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

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

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

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

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

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

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

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

    *closes the session by closing the input file*

## Private Attributes

- FILE ∗ f

  *pointer to input binary file*
- unsigned int buffer

  *the last chunk read from the input*
- unsigned int head_mask

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

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

### 6.11.1 Detailed Description

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

### 6.11.2 Constructor & Destructor Documentation

#### 6.11.2.1 ibitstream()

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

```
                            {
    f = fopen(file_name.c_str(), "rb+");
    buffer =0;
    head_mask = 0;
    head_place = 0;
  }
```

### 6.11.3 Member Function Documentation

#### 6.11.3.1 bin_inter_decode() [1/2]

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

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

**Parameters**

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

```
                                                    {
  unsigned int a_size;
  a_size = read_bits(b); // size of the vector
  //cout << "a_size " << a_size << endl;

  if (a_size == 0)
    return;
  if (a_size == 1){
    a.push_back(read_bits(b));
    return;
  }

  // read low and high values
  unsigned int low, high;
  low = read_bits(b);
  high = read_bits(b);
  //cout << " low " << low << " high " << high << endl;
  bin_inter_decode(a, 0, a_size - 1, low, high);
  return;
}
```

#### 6.11.3.2 bin_inter_decode() [2/2]

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

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

**Parameters**

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

```
                                                                                {
  // cout << " i " << i << " j " << j << " low " << low << " high " << high << endl;
  if (j < i)
    return;
  if (i==j){
    if(low == high)
      a.push_back(low); // the element must be low = high, no other change, nothing to read
    else
      a.push_back(read_bits(nu_bits(high-low)) + low);
    return;
  }

  int m = (i+j)/2;
  unsigned int L = low + m - i; // lower bound on a[m]
  unsigned int H = high - (j - m); // upper bound on a[m]
```

```
  unsigned int a_m; // the value of the intermediate point
  if (L == H)
    a_m = L; // there will be no bits to read
  else
    a_m = read_bits(nu_bits(H-L)) + L;

  a.push_back(a_m);

  bin_inter_decode(a, i, m-1, low, a_m - 1);
  bin_inter_decode(a, m+1, j, a_m + 1, high);
}
```

#### 6.11.3.3 close()

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

closes the session by closing the input file

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

#### 6.11.3.4 operator>>() [1/2]

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

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

```
                                                {
  // implement Elias delta decoding
  //bitset<32> B(buffer);
  //cerr << " buffer " << B << endl;
  //cerr << " head " << bitset<32>(head_mask) << endl;
  //cerr << " head position " << head_place << endl;

  unsigned int L = 0;
  while (!read_bit()){
    // read until reach one
    L++;
  }
  //cerr << " L " << L << endl;

  unsigned int N;
  if (L == 0){// special case, avoid going over further calculations
    n = 0; // we had subtracted one when encoding
    return *this;
  }
  N = read_bits(L); // read L digits

  N += (1<<L); // we must add 2^L
  N --; // this was N + 1

  n = read_bits(N); // read N digits
  n += (1 << N); // we must add 2^N
  n --; // when we encoded, in order to get a positive integer, we added one, now we subtract one
  return *this;
}
```

#### 6.11.3.5 operator>>() [2/2]

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

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

```
{
  unsigned int L = 0;
  //cout << "head_place " << head_place << endl;
  //cout << "head_mask  " << head_mask << endl;
  while (!read_bit()){
    // read until reach one
    L++;
  }

  //cout << " L = " << L << endl;
  unsigned int N;
  if (L == 0){// special case, avoid going over further calculations
    n = 0; // we had subtracted one when encoding
    return *this;
  }

  N = read_bits(L); // read L digits

  N += (1<<L); // we must add 2^L
  N --; // this was N + 1

  // we must read N bits and form n based on that

  bit_pipe B;
  //cout << " N " << N << endl;
  read_bits(N, B);
  //cout << " B first " << B << endl;
  // we should add a leading 1 to B
  // in order to do so, we should consider 2 cases:
  if (N % BIT_INT == 0){
    // this is the tricky case, since B now contains full chunks and there is no room to add the leading 1
    // so we need to insert a chunk at the beginning and place the leading bit there
    // since the leading bit will be in the rightmost bit in this case, the value of the initial chunk is 1 in this case
    B.bits.insert(B.bits.begin(), 1);
  }else{
    // in this case, the lading bit will be placed in the first chunk of B
    B.bits[0] |= (1 << (N % BIT_INT));
  }
  //cout << " B " << B << endl;
  //cout << B.bits[0] << endl;

  // construct the mpz_clas
  mpz_import(n.get_mpz_t(),
             B.bits.size(), // the number of words
             1, // order: 1 means first significant word first
             sizeof(unsigned int), // each word is this many bytes
             0, // endian can be 1 for most significant byte first, -1 for least significant first, or 0 for the native endianness of the host CPU.
             0, // nails
             &B.bits[0]); //&B.bits[0]);

  n --; // when encoding, we added 1 to make sure it is positive
  return *this;
}
```

#### 6.11.3.6 read_bit()

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

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

```
                        {
  if (head_mask == 0){ // nothing is in buffer
    //cerr << " read a chunk " << endl;
    read_chunk();
  }
  bool ans = head_mask & buffer; // look at the value of buffer at the bit where the head_mask is pointing to
  head_mask >>= 1; // go one bit to the right
  head_place --;
  //cerr << " read bit " << ans << endl;
  return ans;
}
```

#### 6.11.3.7 read_bits() [1/2]

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

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

```
                                               {

  //cerr << " read bits with k = " << k << endl;
  if (k < 1 or k > BIT_INT){
    cerr << "ERROR: ibitstream::read_bits called with k out of range, k = " << k << endl;
  }
  if (head_place == 0)// no bits left
    read_chunk();
  if (head_place >= k){ // head_place is effectively the number of unread bits remaining in the buffer
    //cerr << " head_place >= k head_mask = " << head_mask << " head_place = " << head_place << endl;
    // there are enough number of bits in the current buffer to read
    // mask the input
    unsigned int mask = mask_gen(k); // k ones
    // now we should shift mask to start at head_place
    mask <<= (head_place - k);
    unsigned int ans = buffer & mask; // mask out the corresponding bits
    ans >>= (head_place - k); // bring it back to LSB

    // we need to shift head k bits to the right
    // in some compilers, >>= 32 does strange things, in fact it does nothing. To avoid that, I shift k - 1 bits and then an extra 1 bit
    head_mask >>= k - 1;
    head_mask >>= 1;
    head_place -= k;
    //cerr << " after head_mask " << head_mask << " head_place = " << head_place << endl;
    return ans;
  }else{
    // there is not enough bits in the current buffer.
    // So we should read head_place many bits from the current buffer
    // then read another chunk from input file
    // and then read k - head_place bits from the new buffer
    // we do these two steps recursively
    // but first need to store the number of bits we will have to read in the future, since these variables will be modified later:
    unsigned int future_bits = k - head_place;
    unsigned int a = read_bits(head_place); // the bits from the current buffer
    read_chunk();
    unsigned int b = read_bits(future_bits); // bits from the next buffer
    // now we need to combine these
    // in order to do so, we need to shift a to the left and combine with b
    // but the number of bits we need to shift a is exactly future bits
    a <<= future_bits;
    return a | b;
  }
}
```

#### 6.11.3.8 read_bits() [2/2]

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

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

```
{
  //cerr << " read_bits " << k << endl;
  // assumption: B is empty bit_pipe
  if (B.bits.size() != 0 or B.last_bits != 0){
    cerr << " ERROR: ibitstream::read_bits(int k, bit_pipe& B) must be called with an empty bit_pipe, a
 nonempty bitpipe is given with B.bits.size() = " << B.bits.size() << endl;
  }
  read_bits_append(k, B);
  // there might be a few zero chunks at the beginning of B which are redundant, we remove them here
  // the number of nonzero chunks is exactly the floor of k / BIT_INT
  int nonzero_chunks = k / BIT_INT;
  if (k % BIT_INT != 0)
    nonzero_chunks ++; // take the floor
  //cerr << " nonzero_chunks " << nonzero_chunks << endl;
  if (nonzero_chunks < B.bits.size())
    B.bits.erase(B.bits.begin(), B.bits.begin() + B.bits.size() - nonzero_chunks);
}
```

#### 6.11.3.9 read_bits_append()

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

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

```
{
  //cout << " read_bits called k = " << k << " head_place = " << head_place << endl;
  // by assumption, when calling this function, B has full chunks (last_bits is either zero so BIT_INT)
  if (k == 0)
    return; // nothing remains to be done
  if (head_place == 0){
    // we are over with the current bits in the buffer
    // so we need to load a few chunks from the input
    // we should append k / BIT_INT full chunks to B and then k % BIT_INT bits from the next chunk
    unsigned int full_chunks = k / BIT_INT;
    if (full_chunks > 0){
      B.bits.resize(B.bits.size() + full_chunks);
      fread(&B.bits[B.bits.size() - full_chunks], sizeof(unsigned int), full_chunks,
  f); // read full_chunks many chunks
      B.last_bits = BIT_INT; // the last chunk contains full bits
    }
    unsigned int res_bits = k % BIT_INT; // the remaining bits to be read
    if (res_bits > 0){
      // we need to read an extra res bits
      unsigned int res = read_bits (res_bits); // read res many bits
      // we should shift res_bits so that its MSB is the leftmost bits of the chunk
      res <<= (BIT_INT - res_bits);
      B.bits.push_back(res);
      B.last_bits = res_bits;
    }
  }else{
    if (k <= head_place){
      // there are enough bits to read
      unsigned int a = read_bits(k);
      // no need to shift a since we need LSB of a to be in the rightmost bit
      B.bits.push_back(a);
```

```
      B.last_bits = BIT_INT;
    }else{
      // read head_place bits and call again
      unsigned int future_read; // number of bits to read in future after calling the read_bits function below
      future_read = k - head_place;
      unsigned int a = read_bits(head_place);
      B.bits.push_back(a);
      B.last_bits = BIT_INT;
      read_bits_append(future_read, B); // read the remaining bits
    }
  }

  B.shift_right(BIT_INT - B.last_bits); // so that LSB of B is the rightmost bit of the lats chunk.

  // this is important to make sure that B is correctly representing an integer and can be converted to mpz_class
  // TODO issue of 2^k - 1 correct
}
```

#### 6.11.3.10 read_chunk()

`void ibitstream::read_chunk ( )`

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

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

### 6.11.4 Member Data Documentation

#### 6.11.4.1 buffer

`unsigned int ibitstream::buffer  [private]`

the last chunk read from the input

#### 6.11.4.2 f

`FILE* ibitstream::f  [private]`

pointer to input binary file

#### 6.11.4.3 head_mask

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

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

#### 6.11.4.4 head_place

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

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

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

- bitstream.h
- bitstream.cpp

## 6.12 log_entry Class Reference

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

### Public Member Functions

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

### Public Attributes

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

### 6.12.1 Constructor & Destructor Documentation

#### 6.12.1.1 log_entry()

```
log_entry::log_entry (
            string name,
            string description,
            int depth )
```

```
{
  name = name_;
  description = description_;
  depth = depth_;
  t = high_resolution_clock::now();
  sys_t = system_clock::now();
}
```

### 6.12.2 Member Data Documentation

#### 6.12.2.1 depth

```
int log_entry::depth
```

#### 6.12.2.2 description

```
string log_entry::description
```

#### 6.12.2.3 name

```
string log_entry::name
```

#### 6.12.2.4 sys_t

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

#### 6.12.2.5 t

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

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

- logger.h
- logger.cpp

## 6.13 logger Class Reference

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

### Static Public Member Functions

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

### Static Public Attributes

- static bool verbose = true

  *if true, every entry is printed at the time of its report, default is false*
- static bool stat = false

  *if true, statistics will be displayed, e.g. number of star vertices / edges or the number of partition graphs etc*
- static ostream ∗ verbose_stream = &cout

  *the stream for printing entries at the time of arrival, default is cout*
- static bool report = true

  *if true, at the end of the program, a hierarchical report is printed*
- static ostream ∗ report_stream = &cout

  *the stream to report the final report. Default is cout*
- static ostream ∗ stat_stream = &cout

  *the stream to report the statistics. Default is cout*
- static vector< log_entry > logs
- static int current_depth = 1
- static map< string, float > item_duration
- static map< string, high_resolution_clock::time_point > item_last_start

  *the last time each item was started*

### 6.13.1 Member Function Documentation

#### 6.13.1.1 add_entry()

```
void logger::add_entry (
            string name,
            string description ) [static]
```

```
{
  //cerr << " adding entry current_depth " << current_depth << endl;
  log_entry new_entry(name, description, current_depth);
  logger::logs.push_back(new_entry);
  if (logger::verbose){
    string s = "";
    time_t tt = system_clock::to_time_t(new_entry.sys_t);
    char buffer[80];
    strftime(buffer, 80, "%F %r", localtime(&tt));
    for (int i=1;i<(current_depth-1);i++)
      s += "|---";
    string buffer_str(buffer);
    s +=  name + " (" + description + ") ";
    s += buffer_str;
    *verbose_stream << s << endl;
  }
}
```

#### 6.13.1.2 item_start()

```
void logger::item_start (
            string name )  [static]
```

```
                               {
  item_last_start[name] = high_resolution_clock::now();
}
```

#### 6.13.1.3 item_stop()

```
void logger::item_stop (
            string name )  [static]
```

```
                              {
  high_resolution_clock::time_point t = high_resolution_clock::now();
  duration<float> diff = t - item_last_start[name];
  item_duration[name] += diff.count();
}
```

#### 6.13.1.4 log()

```
void logger::log ( )  [static]
```

```
               {
 //cerr << " log started " << endl;
 int max_depth = 0;
 for (int i=0;i<logs.size(); i++){
   if (logs[i].depth > max_depth)
     max_depth = logs[i].depth;
 }

 vector<int> parent(logs.size()); // for a log L, parent[L] is the max index i < L such that depth[i] < depth[L], this shows in what block we are in
 vector<int> next(logs.size()); // for a log L, next[L] is the min index i > L such that depth[i] >= depth[L], this is the index right after L in its block, or if L is the last entry in its block, it is the start of the next block. The diff between L and next[L] shows the duration of running L
 for (int i=1;i<(logs.size()-1);i++){
   // finding parent[i]
   //cerr << " finding parent " << i << endl;
   for (int j=(i-1); j>=0 ;j--){
     if (logs[j].depth < logs[i].depth){
       parent[i] = j;
       break;
     }
   }
   //cerr << " parent[" << i << "] = " << parent[i] << endl;
   //cerr << " finding next " << i << endl;
   for (int j=(i+1); j<logs.size();j++){
     if (logs[j].depth <= logs[i].depth){
       next[i] = j;
       break;
     }
   }
   //cerr << " next[" << i << "] = " << next[i] << endl;
 }
 next[0] = logs.size()-1; // next of start entry is the finish entry

 vector<float> dur(logs.size()); // dur[L] is the duration that entry L takes, meaning the difference between L and next[L]
 vector<float> block_dur(logs.size()); // the duration of the whole block for each entry, which is the difference between
```

```
  vector<float> block_percent(logs.size()); // the percentage of time each entry takes inside a block
  duration<float> diff;
  string s;
  //cerr << " logs.size() " << logs.size() << endl;
  *report_stream << endl;
  for (int i=1;i<(logs.size()-1); i++){
    //cerr << " i " << i << endl;
    // finding duration[i]
    diff = logs[next[i]].t - logs[i].t;
    dur[i] = diff.count();
    //cerr << " dur[i] " << dur[i] << endl;
    // finding block_duration
    diff = logs[next[parent[i]]].t - logs[parent[i]].t;
    block_dur[i] = diff.count();
    //cerr << " block_dur[i] " << block_dur[i] << endl;
    block_percent[i] = dur[i] / block_dur[i] * 100;
    //cerr << " block_percent[i] " << block_percent[i] << endl;
    s = "";
    //cerr << " logs[i].depth " << logs[i].depth << endl;
    for (int j=1;j<(logs[i].depth-1); j++)
      s += "|---";
    s +=  logs[i].name + " (" + logs[i].description + "): " + to_string(dur[i]) + "s " + "[" +
   to_string(block_percent[i]) + "\%]" + " ";
    *report_stream << s << endl;
  }

  *report_stream << endl << " itemized log " << endl;
  for (map<string, float>::iterator it = item_duration.begin(); it!=
   item_duration.end(); it++)
    *report_stream << it->first << " : " << it->second << endl;
}
```

#### 6.13.1.5 start()

`void logger::start ( )  [static]`

```
                {
  log_entry new_log("Start", "", 0);
  //cerr << " started " << endl;
  logs.push_back(new_log);
  //cerr << logger::logs.size() << endl;
}
```

#### 6.13.1.6 stop()

`void logger::stop ( )  [static]`

```
               {
  log_entry new_log("Finish", "", 0);
  //cerr << " finished " << endl;
  logs.push_back(new_log);
  //cerr << logs.size() << endl;
  if (report)
    log();

}
```

### 6.13.2 Member Data Documentation

#### 6.13.2.1 current_depth

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

#### 6.13.2.2 item_duration

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

#### 6.13.2.3 item_last_start

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

the last time each item was started

#### 6.13.2.4 logs

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

#### 6.13.2.5 report

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

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

#### 6.13.2.6 report_stream

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

the stream to report the final report. Default is cout

#### 6.13.2.7 stat

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

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

#### 6.13.2.8 stat_stream

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

the stream to report the statistics. Default is cout

#### 6.13.2.9 verbose

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

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

#### 6.13.2.10 verbose_stream

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

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

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

- logger.h
- logger.cpp

## 6.14 marked_graph Class Reference

simple marked graph

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

### Public Member Functions

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

### Public Attributes

- int nu_vertices
  *number of vertices in the graph*
- vector< vector< pair< int, pair< int, int > > > > adj_list
  *adj_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, mark towards i, mark towards other endpoint)*
- vector< vector< int > > index_in_neighbor
  *index_in_neighbor[v][i] is the index of vertex v in the adjacency list of the ith neighbor of v*
- vector< int > ver_mark
  *ver_mark[i] is the mark of vertex i*

### Friends

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

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

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

  *prints a marked graph to the output*

## 6.14.1 Detailed Description

simple marked graph

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

## 6.14.2 Constructor & Destructor Documentation

### 6.14.2.1 marked_graph() [1/2]

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

default constructor

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

### 6.14.2.2 marked_graph() [2/2]

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

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

**Parameters**

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

```
{
  nu_vertices = n;
  adj_list.resize(n);
  //adj_location.resize(n);
  index_in_neighbor.resize(n);
  // modify the edges if necessary so that in each element of the form ((i,j), (x, x')), we have i < j. This is important when forming adjacency lists so that the list of each vertex is sorted
   for (int i=0;i<edges.size();i++){
     if (edges[i].first.first > edges[i].first.second){
       // swap the edge endpoints and represent it in the other direction
       swap(edges[i].first.first, edges[i].first.second);
       // also, we should swap the mark components
       swap(edges[i].second.first, edges[i].second.second);
     }
   }
   sort(edges.begin(), edges.end()); // so that the adjacency list is sorted
   for (int k=0; k<edges.size(); k++){
     // (i,j) are endpoints if the edge
     // (x,y) are marks, x towards i and y towards j
     int i = edges[k].first.first;
     int j = edges[k].first.second;
     int x = edges[k].second.first;
     int y = edges[k].second.second;
     if (i <0 || i >= n || j < 0 || j >= n || i ==j)
       cerr << " ERROR: graph::graph(n, edges) received an invalid pair of edges with n = " << n << " : (" << i << " , " << j << " )" << endl;
     adj_list[i].push_back(pair<int, pair<int, int> > (j, pair<int, int> (x,y)));
     //adj_location[i][j] = adj_list[i].size() - 1;
     adj_list[j].push_back(pair<int, pair<int, int> > (i, pair<int, int> (y,x)));
     //adj_location[j][i] = adj_list[j].size() - 1;
     index_in_neighbor[i].push_back(adj_list[j].size()-1);
     index_in_neighbor[j].push_back(adj_list[i].size()-1);
   }


   ver_mark = vertex_marks;
 }
```

### 6.14.3 Friends And Related Function Documentation

#### 6.14.3.1 operator"!=

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

checks whether two marked graphs are not equal

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

#### 6.14.3.2 operator<<

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

prints a marked graph to the output

```
{
  o << G.nu_vertices << endl;
  for (int v=0;v<G.nu_vertices;v++){
    o << G.ver_mark[v];
    if (v < G.nu_vertices-1)
      o << " ";
  }
  o << endl;

  vector<pair<pair<int, int>, pair<int, int> > > edges;
  pair<pair<int, int> , pair<int, int> > edge; // the current edge to be added to the list
  for (int v=0;v<G.nu_vertices;v++){
    for (int i=0;i<G.adj_list[v].size();i++){
      if (G.adj_list[v][i].first > v){ // avoid duplicate in edge list, only add edges where the other endpoint has a greater index
        edge.first.first = v;
        edge.first.second = G.adj_list[v][i].first;
        edge.second = G.adj_list[v][i].second;
        edges.push_back(edge);
      }
    }
  }
  sort(edges.begin(), edges.end());
  o << edges.size() << endl;
  for(int i=0;i<edges.size();i++){
    o << edges[i].first.first << " " << edges[i].first.second << " " << edges[i].second.first << " " << edges[i].second.second << endl;
  }
  return o;
  /*
  o << " number of vertices " << G.nu_vertices << endl;
  vector<pair<int, pair<int, int> > > l; // the adjacency list of a vertex
  for (int v=0; v<G.nu_vertices; v++){
    o << " vertex " << v << " mark " << G.ver_mark[v] << endl;
    //o << " adj list (connections to vertices with greater index): format (j, (x,y))" << endl;
    o << " adj list " << endl;
    l = G.adj_list[v];
    sort(l.begin(), l.end(), edge_compare);
    for (int i=0;i<l.size();i++){
      if (l[i].first > v)
        o << " (" << l[i].first << ", (" << l[i].second.first << ", " << l[i].second.second << ")) ";
    }
    o << endl << endl;
  }
  return o;
  */
}
```

#### 6.14.3.3 operator==

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

checks whether two marked graphs are the same.

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

```
{
  if (G1.nu_vertices != G2.nu_vertices)
    return false;
  return G1.adj_list == G2.adj_list;
  int n = G1.nu_vertices; // number of vertices of the two graphs
  vector< pair< int, pair< int, int > > > l1, l2; // the adjacency list of a vertex in two graphs for comparison.
  for (int v=0;v<n;v++){
    if (G1.ver_mark[v] != G2.ver_mark[v]) // mark of each vertex should be the same
      return false;
    if (G1.adj_list[v].size() != G2.adj_list[v].size()) // each vertex must have the same degree in two graphs
      return false;
    l1 = G1.adj_list[v];
```

```
    l2 = G2.adj_list[v];
    sort(l1.begin(), l1.end(), edge_compare); // sort with respect to the other endpoint
    sort(l2.begin(), l2.end(), edge_compare);
    if (l1 != l2) // after sorting, the lists must match
      return false;
  }
  return true;
}
```

## 6.14.4 Member Data Documentation

### 6.14.4.1 adj_list

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

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

### 6.14.4.2 index_in_neighbor

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

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

### 6.14.4.3 nu_vertices

```
int marked_graph::nu_vertices
```

number of vertices in the graph

### 6.14.4.4 ver_mark

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

ver_mark[i] is the mark of vertex i

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

- marked_graph.h
- marked_graph.cpp

## 6.15 marked_graph_compressed Class Reference

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

### Public Member Functions

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

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

    *read the compressed data from a binary file*
- void binary_read (string s)

### Public Attributes

- int n

    *the number of vertices*
- int h

    *the depth up to which the compression was performed*
- int delta

    *the degree threshold used when compression was performed*
- pair< vector< int >, mpz_class > star_vertices

    *the compressed form of the star_vertices list*
- map< pair< int, int >, vector< vector< int > > > star_edges

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

    *for an edge type t, type_mark[t] denotes the mark component of t*
- vector< vector< int > > ver_type_list

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

    *the compressed form of vertex types, where the type of a vertex is the index with respect to ver_type_list of the list of integers specifying the type of the vertex (mark of the vertex followed by the number of edges of each type connected to that vertex)*
- map< pair< int, int >, mpz_class > part_bgraph

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

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

#### 6.15.1 Member Function Documentation

#### 6.15.1.1 binary_read() [1/2]

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

read the compressed data from a binary file

**Parameters**

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

```
                                              {
clear(); // to make sure nothing is stored inside me before reading

// ==== read n, h, delta
fread(&n, sizeof n, 1, f);
fread(&h, sizeof h, 1, f);
fread(&delta, sizeof delta, 1, f);

int int_in; // auxiliary input integer
// ===== read type_mark
// read number of types
fread(&int_in, sizeof int_in, 1, f);
type_mark.resize(int_in);
for (int i=0;i<type_mark.size();i++){
  fread(&int_in, sizeof int_in, 1, f);
  type_mark[i] = int_in;
}

// ==== read star_vertices
// first, read the frequency.
star_vertices.first = vector<int>(2); // frequency,
// we read its first index which is number of zeros, and the second is n - the first.
fread(&int_in, sizeof int_in, 1, f);
star_vertices.first[0] = int_in;
star_vertices.first[1] = n - int_in;

// the integer representation which is star_vertices.second
mpz_inp_raw(star_vertices.second.get_mpz_t(), f);

// ==== read star_edges

int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n), which
 is the number of bits to encode vertices
int n_copy = n;
while(n_copy > 0){
  n_copy >>= 1;
  log2n ++;
}
bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index

string s;
stringstream ss;
int sp; // the index of the string s we are studying

// read the size of star_edges

int star_edges_size;
fread(&star_edges_size, sizeof star_edges_size, 1, f);

int x, xp; // edge marks
int nu_star_vertices = star_vertices.first[1];

vector<vector<int> > V; // the list of star edges corresponding to each mark pair
V.resize(nu_star_vertices);

for (int i=0;i<star_edges_size;i++){
  fread(&x, sizeof x, 1, f);
  fread(&xp, sizeof xp, 1, f);

  s = bit_string_read(f);
  //cerr << " read  x " << x << " xp " << xp << " s " << s << endl;
  sp = 0; // starting from zero
  for (int j=0; j<nu_star_vertices; j++){ //
    V[j].clear(); // make it fresh
    while(s[sp++] == '1'){ // there is still some edge connected to this vertex
      // read log2n many bits
      //cerr << " s subtr " << s.substr(sp, log2n);
```

```
        //ss << s.substr(sp, log2n);
        B = bitset<8*sizeof(int)>(s.substr(sp, log2n));
        //cerr << " ss " << ss.str() << endl;
        sp += log2n;
        //ss >> B;

        V[j].push_back(B.to_ulong());
      }
      //for (int k=0;k<V[j].size();k++)
      //  cerr << " , " << V[j][k];
      //cerr << endl;
    }


    star_edges.insert(pair< pair<int, int> , vector<vector<int> > > (pair<int, int>(x, xp), V));
  }

  // ==== read vertex_types

  // read ver_type_list
  fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list
  ver_type_list.resize(int_in);
  for (int i=0; i<ver_type_list.size();i++){
    fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list[i]
    ver_type_list[i].resize(int_in);
    for (int j=0;j<ver_type_list[i].size();j++){
      fread(&int_in, sizeof int_in, 1, f);
      ver_type_list[i][j] = int_in;
    }
  }

  // ver_types
  // ver_types.first
  // ver_types.first.size()
  fread(&int_in, sizeof int_in, 1, f);
  ver_types.first.resize(int_in);
  for (int i=0;i<ver_types.first.size();i++){
    fread(&int_in, sizeof int_in, 1, f);
    ver_types.first[i] = int_in;
  }
  // ver_types.second
  mpz_inp_raw(ver_types.second.get_mpz_t(), f);


  // === part bgraphs
  int part_bgraph_size;
  int t, tp;
  pair<int, int> type;
  mpz_class part_g;
  fread(&part_bgraph_size, sizeof part_bgraph_size, 1, f);
  for (int i=0;i<part_bgraph_size;i++){
    // read t, t'
    fread(&t, sizeof t, 1, f);
    fread(&tp, sizeof tp, 1, f);
    type = pair<int, int>(t, tp);
    mpz_inp_raw(part_g.get_mpz_t(), f);
    part_bgraph.insert(pair<pair<int, int>, mpz_class> (type, part_g));
  }

  // === part graphs

  // first, the size
  int part_graph_size;
  int v_size;
  vector<int> W;
  fread(&part_graph_size, sizeof part_graph_size, 1, f);
  for (int i=0;i<part_graph_size; i++){
    // first, the type
    fread(&t, sizeof t, 1, f);
    // then, the mpz part
    mpz_inp_raw(part_g.get_mpz_t(), f);
    // then, the vector size
    fread(&v_size, sizeof v_size, 1, f);
    W.resize(v_size);
    for (int j=0;j<v_size; j++){
      fread(&int_in, sizeof int_in, 1, f);
      W[j] = int_in;
    }
    part_graph.insert(pair<int, pair< mpz_class, vector< int > > >(t, pair<mpz_class, vector<int>
   >(part_g, W)));
  }
}
```

#### 6.15.1.2 binary_read() [2/2]

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

read the compressed data from a binary file

**Parameters**

| | |
|---|---|
| *s* | string containing the name of the binary file |

```
                                                  {
clear(); // to make sure nothing is stored inside me before reading
ibitstream inp(s);

// ==== read n, h, delta
unsigned int int_in; // auxiliary input integer
inp >> int_in;
n = int_in; // I need to do this, since ibitstream::operator >> gets unsigned int& and the compile can not cast int& to unsigned int&
inp >> int_in;
h = int_in;
inp >> int_in;
delta = int_in;

//fread(&n, sizeof n, 1, f);
//fread(&h, sizeof h, 1, f);
//fread(&delta, sizeof delta, 1, f);


// ===== read type_mark
// read number of types
inp >> int_in;
//fread(&int_in, sizeof int_in, 1, f);
type_mark.resize(int_in);
for (int i=0;i<type_mark.size();i++){
  inp >> int_in;
  type_mark[i] = int_in;
  //fread(&int_in, sizeof int_in, 1, f);
  //type_mark[i] = int_in;
}

// ==== read star_vertices
// first, read the frequency.
star_vertices.first = vector<int>(2); // frequency,
// we read its first index which is number of zeros, and the second is n - the first.
inp >> int_in;
//fread(&int_in, sizeof int_in, 1, f);
star_vertices.first[0] = int_in;
star_vertices.first[1] = n - int_in;

// the integer representation which is star_vertices.second
inp >> star_vertices.second;
//mpz_inp_raw(star_vertices.second.get_mpz_t(), f);

// ==== read star_edges

//int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n), which is the number of bits to encode vertices
//int n_copy = n;
//while(n_copy > 0){
//n_copy >>= 1;
//log2n ++;
//}
//bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index

//string s;
//stringstream ss;
//int sp; // the index of the string s we are studying

// read the size of star_edges

unsigned int star_edges_size;
inp >> star_edges_size;
//fread(&star_edges_size, sizeof star_edges_size, 1, f);

unsigned int x, xp; // edge marks
int nu_star_vertices = star_vertices.first[1];
```

```
vector<vector<int> > V; // the list of star edges corresponding to each mark pair
V.resize(nu_star_vertices);

unsigned int n_bits = nu_bits(n); // the number of bits in n, i.e. \f$1 + \lfloor \log_2 n \rfloor\f$

for (int i=0;i<star_edges_size;i++){
  inp >> x;
  inp >> xp;
  //fread(&x, sizeof x, 1, f);
  //fread(&xp, sizeof xp, 1, f);

  //s = bit_string_read(f);
  //cerr << " read  x " << x << " xp " << xp << " s " << s << endl;
  //sp = 0; // starting from zero
  for (int j=0; j<nu_star_vertices; j++){ //
    V[j].clear(); // make it fresh
    inp.bin_inter_decode(V[j], n_bits); // use binary interpolative decoding
    // while(inp.read_bit()){ // there is still some edge connected to this vertex
    //   // read log2n many bits
    //   //cerr << " s subtr " << s.substr(sp, log2n);
    //   //ss << s.substr(sp, log2n);
    //   //B = bitset<8*sizeof(int)>(s.substr(sp, log2n));
    //   //cerr << " ss " << ss.str() << endl;
    //   //sp += log2n;
    //   //ss >> B;
    //   //inp >> int_in;
    //   int_in = inp.read_bits(n_bits);
    //   V[j].push_back(int_in);
    // }
    //for (int k=0;k<V[j].size();k++)
    //  cerr << " , " << V[j][k];
    //cerr << endl;
  }


  star_edges.insert(pair< pair<int, int> , vector<vector<int> > > (pair<int, int>(x, xp), V));
}

// ==== read vertex_types

// read ver_type_list
inp >> int_in;
//fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list
ver_type_list.resize(int_in);
for (int i=0; i<ver_type_list.size();i++){
  inp >> int_in;
  //fread(&int_in, sizeof int_in, 1, f); // size of ver_type_list[i]
  ver_type_list[i].resize(int_in);
  for (int j=0;j<ver_type_list[i].size();j++){
    //fread(&int_in, sizeof int_in, 1, f);
    inp >> int_in;
    ver_type_list[i][j] = int_in;
  }
}

// ver_types
// ver_types.first
// ver_types.first.size()
inp >> int_in;
//fread(&int_in, sizeof int_in, 1, f);
ver_types.first.resize(int_in);
for (int i=0;i<ver_types.first.size();i++){
  //fread(&int_in, sizeof int_in, 1, f);
  inp >> int_in;
  ver_types.first[i] = int_in;// = int_in;
}
// ver_types.second
inp >> ver_types.second;
//mpz_inp_raw(ver_types.second.get_mpz_t(), f);


// === part bgraphs
unsigned int part_bgraph_size;
unsigned int t, tp;
pair<int, int> type;
mpz_class part_g;
inp >> part_bgraph_size;
//fread(&part_bgraph_size, sizeof part_bgraph_size, 1, f);
for (int i=0;i<part_bgraph_size;i++){
  // read t, t'
  inp >> t;
  inp >> tp;
  //fread(&t, sizeof t, 1, f);
  //fread(&tp, sizeof tp, 1, f);
  type = pair<int, int>(t, tp);
```

```
    inp >> part_g;
    //mpz_inp_raw(part_g.get_mpz_t(), f);
    part_bgraph.insert(pair<pair<int, int>, mpz_class> (type, part_g));
  }

  // === part graphs

  // first, the size
  unsigned int part_graph_size;
  unsigned int v_size;
  vector<int> W;
  inp >> part_graph_size;
  //fread(&part_graph_size, sizeof part_graph_size, 1, f);
  for (int i=0;i<part_graph_size; i++){
    // first, the type
    inp >> t;
    //fread(&t, sizeof t, 1, f);
    // then, the mpz part
    inp >> part_g;
    //mpz_inp_raw(part_g.get_mpz_t(), f);
    // then, the vector size
    inp >> v_size;
    //fread(&v_size, sizeof v_size, 1, f);
    W.resize(v_size);
    for (int j=0;j<v_size; j++){
      //fread(&int_in, sizeof int_in, 1, f);
      inp >> int_in;
      W[j] = int_in; //= int_in;
    }
    part_graph.insert(pair<int, pair< mpz_class, vector< int > > >(t, pair<mpz_class, vector<int> >(part_g, W)));
  }
  inp.close();
}
```

#### 6.15.1.3 binary_write() [1/2]

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

writes the compressed data to a binary file

**Parameters**

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

```
                                               {

  vector<pair<string, int> > space_log; // stores the number of bits used to store each category. The string part is description of the category, and the int part is the number of bits of output used to express that part.

  int output_bits; // the number of bits in the output corresponding to the current category under investigation, to be zeroed at each step.

  logger::current_depth++;
  // ==== write n, h, delta
  output_bits = 0;
  logger::add_entry("n", "");
  fwrite(&n, sizeof n, 1, f);
  output_bits += sizeof n;

  logger::add_entry("h", "");
  fwrite(&h, sizeof h, 1, f);
  output_bits += sizeof h;

  logger::add_entry("delta", "");
  fwrite(&delta, sizeof delta, 1, f);
  output_bits += sizeof delta;

  space_log.push_back(pair<string, int> ("n, h, delta", output_bits));
```

```

logger::add_entry("type_mark", "");
output_bits = 0;

int int_out; // auxiliary variable, an integer value to be written to output
// ==== write type_mark
// first, the number of types
int_out = type_mark.size();
fwrite(&int_out, sizeof int_out, 1, f);
output_bits += sizeof int_out;
// then, marks one by one
for (int i=0;i<type_mark.size();i++){
  int_out = type_mark[i];
  fwrite(&int_out, sizeof int_out, 1, f);
  output_bits += sizeof int_out;
}

space_log.push_back(pair<string, int>("type mark", output_bits));

logger::add_entry("star_vertices", "");
output_bits = 0;
// ==== write star vertices
// first, write the frequency, note that star_vertices.first is a vector of size 2 with the first entry being the number of zeros, and the second one the number of ones, so it enough to write only one of them
int_out = star_vertices.first[0];
fwrite(&int_out, sizeof int_out, 1, f);
output_bits += sizeof int_out;

// then, we write the integer representation star_vertices.second
output_bits +=  mpz_out_raw(f, star_vertices.second.get_mpz_t()); // mpz_out_raw returns the number of bytes written to the output

space_log.push_back(pair<string, int> ("star vertices", output_bits));

logger::add_entry("star_edges", "");
// ==== write star edges
output_bits = 0;
int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n), which is the number of bits to encode vertices
int n_copy = n;
while(n_copy > 0){
  n_copy >>= 1;
  log2n ++;
}
//cerr << " log2n " << log2n << endl;
bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index

map<pair<int, int>, vector<vector<int> > >::iterator it;
int x, xp;
string s; // the bit stream

// first, write the size of star_edges so that the decoder knows how many blocks are coming
int_out = star_edges.size();
fwrite(&int_out, sizeof int_out, 1, f);
output_bits += sizeof int_out;

int nu_star_edges = 0; // number of star edges
for (it = star_edges.begin(); it!= star_edges.end(); it++){
  x = it->first.first;
  xp = it->first.second;
  //write x and xp
  fwrite(&x, sizeof x, 1, f);
  fwrite(&xp, sizeof xp, 1, f);
  output_bits += sizeof x;
  output_bits += sizeof xp;
  s = "";
  for (int i=0;i<it->second.size();i++){
    for(int j=0;j<it->second[i].size();j++){
      s += "1";
      B = it->second[i][j]; // convert the index of the other endpoint to binary
      s += B.to_string().substr(8*sizeof(int) - log2n, log2n); // take only log2n many bits of the representation (and this should be taken from the least significant bits)
      nu_star_edges ++;
    }
    s += "0"; // to indicate that the neighbor list of this vertex is over now
  }
  //cerr << " write  x " << x << " xp " << xp << " s " << s << endl;
  //for (int i=0;i<it->second.size();i++){
  //  for (int j=0;j<it->second[i].size();j++){
  //    cerr << " , " << it->second[i][j];
  //  }
  //  cerr << endl;
  //}
  output_bits += bit_string_write(f, s); // write this bitstream to the output
}

space_log.push_back(pair<string, int> ("star edges", output_bits));
```

```


logger::add_entry("vertex types", "");
output_bits = 0;

// ==== write vertex types

// first, we need vertex types list (ver_type_list)
// size of ver_type_list
int_out = ver_type_list.size();
fwrite(&int_out, sizeof int_out, 1, f);
output_bits += sizeof int_out;

for (int i=0;i<ver_type_list.size();i++){
  int_out = ver_type_list[i].size();
  fwrite(&int_out, sizeof int_out, 1, f);
  output_bits += sizeof int_out;

  for (int j=0;j<ver_type_list[i].size();j++){
    int_out = ver_type_list[i][j];
    fwrite(&int_out, sizeof int_out, 1, f);
    output_bits += sizeof int_out;
  }
}
space_log.push_back(pair<string, int>("vertex type list", output_bits));
output_bits = 0;

// then, write ver_types

// ver_types.first
// ver_types.first.size():
int_out = ver_types.first.size();
fwrite(&int_out, sizeof int_out, 1, f);
output_bits += sizeof int_out;

for (int i =0;i<ver_types.first.size(); i++){
  int_out = ver_types.first[i];
  fwrite(&int_out, sizeof int_out, 1, f);
  output_bits += sizeof int_out;
}
// ver_types.second
output_bits += mpz_out_raw(f, ver_types.second.get_mpz_t());

space_log.push_back(pair<string, int> ("vertex types", output_bits));

logger::add_entry("partition bipartite graphs", "");


// ==== part bgraphs
output_bits = 0;

// part_bgraphs.size
int_out = part_bgraph.size();
fwrite(&int_out, sizeof int_out, 1, f);
output_bits += sizeof int_out;

map<pair<int, int>, mpz_class>::iterator it2;
if (logger::stat){
  *logger::stat_stream << " ==== statistics ==== " << endl;
  *logger::stat_stream << " n:                 " << n << endl;
  *logger::stat_stream << " h:                 " << h << endl;
  *logger::stat_stream << " delta:             " << delta << endl;
  *logger::stat_stream << " No. types          " <<
type_mark.size() << endl;
  *logger::stat_stream << " No. * vertices     " << n -
star_vertices.first[0] << endl;
  *logger::stat_stream << " No. * edges        " << nu_star_edges << endl;
  *logger::stat_stream << " No. part bgraphs   " <<
part_bgraph.size() << endl;
  *logger::stat_stream << " No. part graphs    " <<
part_graph.size() << endl;
}

for (it2 = part_bgraph.begin(); it2 != part_bgraph.end(); it2++){
  // first, write t, t'
  int_out = it2->first.first;
  fwrite(&int_out, sizeof int_out, 1, f);
  output_bits += sizeof int_out;
  int_out = it2->first.second;
  fwrite(&int_out, sizeof int_out, 1, f);
  output_bits += sizeof int_out;
  // then, the compressed integer
  output_bits += mpz_out_raw(f, it2->second.get_mpz_t());
}

space_log.push_back(pair<string, int> ("partition bipartite graphs", output_bits));

```

```
  logger::add_entry("partition graphs", "");
  output_bits = 0;
  // === part graphs

  // part_graph.size
  int_out = part_graph.size();
  fwrite(&int_out, sizeof int_out, 1, f);
  output_bits += sizeof int_out;

  map< int, pair< mpz_class, vector< int > > >::iterator it3;
  for (it3 = part_graph.begin(); it3 != part_graph.end(); it3++){
    int_out = it3->first; // the type
    fwrite(&int_out, sizeof int_out, 1, f);
    output_bits += sizeof int_out;

    // the mpz part
    output_bits += mpz_out_raw(f, it3->second.first.get_mpz_t());
    // the vector part
    // first its size
    int_out = it3->second.second.size();
    fwrite(&int_out, sizeof int_out, 1, f);
    output_bits += sizeof int_out;
    // then element by element
    for(int j=0;j<it3->second.second.size();j++){
      int_out = it3->second.second[j];
      fwrite(&int_out, sizeof int_out, 1, f);
      output_bits += sizeof int_out;
    }
  }
  space_log.push_back(pair<string, int>("partition graphs", output_bits));


  if (logger::stat){
    *logger::stat_stream << endl << endl;
    *logger::stat_stream << " Number of bytes used for each part " << endl;
    *logger::stat_stream << " ---------------------------------- " << endl << endl;

    int total_bytes = 0;
    for (int i=0; i < space_log.size(); i++)
      total_bytes += space_log[i].second;

    for (int i=0; i < space_log.size(); i++){
      *logger::stat_stream << space_log[i].first << " -> "  << space_log[i].second << " ( " << float(100) * float(space_log[i].second) / float(total_bytes) << " % " << endl;
    }

    *logger::stat_stream << " Total number of bytes wrote to the output = " << total_bytes << endl;
  }
  logger::current_depth--;
}
```

#### 6.15.1.4 binary_write() [2/2]

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

writes the compressed data to a binary file

**Parameters**

| | |
|---|---|
| *s* | string containing the name of the binary file |

```
{

  obitstream oup(s);

  vector<pair<string, int> > space_log; // stores the number of bits used to store each category. The string part is description of the category, and the int part is the number of bits of output used to express that part.

```

```
//int output_bits; // the number of bits in the output corresponding to the current category under
 investigation, to be zeroed at each step.
unsigned int chunks = 0; // number of chunks written to the output. Each chunk is sizeof(unsigned int) =
 32 bits long
unsigned int chunks_new =0; // to take the difference in each step

logger::current_depth++;
// ==== write n, h, delta
//output_bits = 0;
logger::add_entry("n", "");
oup << n; //fwrite(&n, sizeof n, 1, f);
//output_bits += sizeof n;

logger::add_entry("h", "");
oup << h; //fwrite(&h, sizeof h, 1, f);
//output_bits += sizeof h;

logger::add_entry("delta", "");
oup << delta; //fwrite(&delta, sizeof delta, 1, f);
//output_bits += sizeof delta;

chunks_new = oup.chunks();
space_log.push_back(pair<string, int> ("n, h, delta", chunks_new - chunks));
chunks = chunks_new;

logger::add_entry("type_mark", "");
//output_bits = 0;

int int_out; // auxiliary variable, an integer value to be written to output
// ==== write type_mark
// first, the number of types
oup <<  type_mark.size();
//fwrite(&int_out, sizeof int_out, 1, f);
//output_bits += sizeof int_out;
// then, marks one by one
for (int i=0;i<type_mark.size();i++){
  oup <<  type_mark[i];
  //fwrite(&int_out, sizeof int_out, 1, f);
  //output_bits += sizeof int_out;
}

chunks_new = oup.chunks();
space_log.push_back(pair<string, int>("type mark", chunks_new - chunks));
chunks = chunks_new;

logger::add_entry("star_vertices", "");
//output_bits = 0;
// ==== write star vertices
// first, write the frequency, note that star_vertices.first is a vector of size 2 with the first entry
 being the number of zeros, and the second one the number of ones, so it enough to write only one of them
oup << star_vertices.first[0];
//fwrite(&int_out, sizeof int_out, 1, f);
//output_bits += sizeof int_out;

// then, we write the integer representation star_vertices.second
oup << star_vertices.second;
//output_bits +=  mpz_out_raw(f, star_vertices.second.get_mpz_t()); // mpz_out_raw returns the number of
 bytes written to the output

chunks_new = oup.chunks();
space_log.push_back(pair<string, int> ("star vertices", chunks_new - chunks));
chunks = oup.chunks();

logger::add_entry("star_edges", "");
// ==== write star edges
//output_bits = 0;
//int log2n = 0; // the ceiling of log (n+1) in base 2 (which is equal to 1 + the floor of log_2 n),
 which is the number of bits to encode vertices
//int n_copy = n;
//while(n_copy > 0){
//n_copy >>= 1;
//log2n ++;
//}
//cerr << " log2n " << log2n << endl;
//bitset<8*sizeof(int)> B; // a bit stream with maximum length of int to store a vertex index

map<pair<int, int>, vector<vector<int> > >::iterator it;
int x, xp;
//string s; // the bit stream

// first, write the size of star_edges so that the decoder knows how many blocks are coming
oup << star_edges.size();
//fwrite(&int_out, sizeof int_out, 1, f);
//output_bits += sizeof int_out;

int nu_star_edges = 0; // number of star edges
unsigned int n_bits = nu_bits(n); // the number of bits in n, i.e. \f$1 + \lfloor \log_2 n
```

```
    \rfloor\f$
   for (it = star_edges.begin(); it!= star_edges.end(); it++){
     x = it->first.first;
     xp = it->first.second;
     //write x and xp
     oup << x;
     oup << xp;
     for (int i=0;i<it->second.size();i++){
       oup.bin_inter_code(it->second[i], n_bits);
       nu_star_edges += it->second[i].size();
     }
   }

   chunks_new = oup.chunks();
   space_log.push_back(pair<string, int> ("star edges", chunks_new - chunks));
   chunks = oup.chunks();

   logger::add_entry("vertex types", "");
   //output_bits = 0;

   // ==== write vertex types

   // first, we need vertex types list (ver_type_list)
   // size of ver_type_list
   oup <<  ver_type_list.size();
   //fwrite(&int_out, sizeof int_out, 1, f);
   //output_bits += sizeof int_out;

   for (int i=0;i<ver_type_list.size();i++){
     oup << ver_type_list[i].size();
     //fwrite(&int_out, sizeof int_out, 1, f);
     //output_bits += sizeof int_out;

     for (int j=0;j<ver_type_list[i].size();j++){
       oup << ver_type_list[i][j];
       //fwrite(&int_out, sizeof int_out, 1, f);
       //output_bits += sizeof int_out;
     }
   }

   chunks_new = oup.chunks();
   space_log.push_back(pair<string, int>("vertex type list", chunks_new - chunks));
   chunks = chunks_new;

   //output_bits = 0;

   // then, write ver_types

   // ver_types.first
   // ver_types.first.size():
   oup << ver_types.first.size();
   //fwrite(&int_out, sizeof int_out, 1, f);
   //output_bits += sizeof int_out;

   for (int i =0;i<ver_types.first.size(); i++){
     oup <<  ver_types.first[i];
     //fwrite(&int_out, sizeof int_out, 1, f);
     //output_bits += sizeof int_out;
   }
   // ver_types.second
   oup << ver_types.second;
   //output_bits += mpz_out_raw(f, ver_types.second.get_mpz_t());

   chunks_new = oup.chunks();
   space_log.push_back(pair<string, int> ("vertex types", chunks_new - chunks));
   chunks = chunks_new;

   logger::add_entry("partition bipartite graphs", "");


   // ==== part bgraphs
   //output_bits = 0;

   // part_bgraphs.size
   oup << part_bgraph.size();
   //fwrite(&int_out, sizeof int_out, 1, f);
   //output_bits += sizeof int_out;

   map<pair<int, int>, mpz_class>::iterator it2;
   if (logger::stat){
     *logger::stat_stream << " ==== statistics ==== " << endl;
     *logger::stat_stream << " n:                   " << n << endl;
     *logger::stat_stream << " h:                   " << h << endl;
     *logger::stat_stream << " delta:               " << delta << endl;
     *logger::stat_stream << " No. types            " <<
   type_mark.size() << endl;
     *logger::stat_stream << " No. * vertices       " << n -
```

```
  star_vertices.first[0] << endl;
    *logger::stat_stream << " No. * edges        " << nu_star_edges << endl;
    *logger::stat_stream << " No. part bgraphs   " << part_bgraph.size() << endl;
    *logger::stat_stream << " No. part graphs    " << part_graph.size() << endl;
  }

  for (it2 = part_bgraph.begin(); it2 != part_bgraph.end(); it2++){
    // first, write t, t'
    oup << it2->first.first;
    //fwrite(&int_out, sizeof int_out, 1, f);
    //output_bits += sizeof int_out;
    oup <<  it2->first.second;
    //fwrite(&int_out, sizeof int_out, 1, f);
    //output_bits += sizeof int_out;
    // then, the compressed integer
    oup << it2->second;
    //output_bits += mpz_out_raw(f, it2->second.get_mpz_t());
  }

  chunks_new = oup.chunks();
  space_log.push_back(pair<string, int> ("partition bipartite graphs", chunks_new - chunks));
  chunks = chunks_new;

  logger::add_entry("partition graphs", "");
  //output_bits = 0;
  // === part graphs

  // part_graph.size
  oup <<  part_graph.size();
  //fwrite(&int_out, sizeof int_out, 1, f);
  //output_bits += sizeof int_out;

  map< int, pair< mpz_class, vector< int > > >::iterator it3;
  for (it3 = part_graph.begin(); it3 != part_graph.end(); it3++){
    oup <<  it3->first; // the type
    //fwrite(&int_out, sizeof int_out, 1, f);
    //output_bits += sizeof int_out;

    // the mpz part
    oup << it3->second.first;
    //output_bits += mpz_out_raw(f, it3->second.first.get_mpz_t());
    // the vector part
    // first its size
    oup <<  it3->second.second.size();
    //fwrite(&int_out, sizeof int_out, 1, f);
    //output_bits += sizeof int_out;
    // then element by element
    for(int j=0;j<it3->second.second.size();j++){
      oup <<  it3->second.second[j];
      //fwrite(&int_out, sizeof int_out, 1, f);
      //output_bits += sizeof int_out;
    }
  }

  chunks_new = oup.chunks();
  space_log.push_back(pair<string, int>("partition graphs", chunks_new - chunks));
  chunks = chunks_new;


  if (logger::stat){
    *logger::stat_stream << endl << endl;
    *logger::stat_stream << " Number of bytes used for each part " << endl;
    *logger::stat_stream << " ---------------------------------- " << endl << endl;

    int total_chunks = 0;
    for (int i=0; i < space_log.size(); i++)
      total_chunks += space_log[i].second;

    for (int i=0; i < space_log.size(); i++){
      // each chunks is 4 bytes.
      *logger::stat_stream << space_log[i].first << " -> "  << 4 * space_log[i].second << " ( " << float(100) * float(space_log[i].second) / float(total_chunks) << " % " << endl;
    }

    *logger::stat_stream << " Total number of bytes wrote to the output = " << 4 * total_chunks << endl;
  }

  oup.close();
  logger::current_depth--;
}
```

#### 6.15.1.5 clear()

```
void marked_graph_compressed::clear ( )
```

```
{
  star_edges.clear();
  type_mark.clear();
  ver_type_list.clear();
  part_bgraph.clear();
  part_graph.clear();
}
```

### 6.15.2 Member Data Documentation

#### 6.15.2.1 delta

```
int marked_graph_compressed::delta
```

the degree threshold used when compression was performed

#### 6.15.2.2 h

```
int marked_graph_compressed::h
```

the depth up to which the compression was performed

#### 6.15.2.3 n

```
int marked_graph_compressed::n
```

the number of vertices

#### 6.15.2.4 part_bgraph

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

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

#### 6.15.2.5 part_graph

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

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

#### 6.15.2.6 star_edges

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

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

#### 6.15.2.7 star_vertices

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

the compressed form of the star_vertices list

#### 6.15.2.8 type_mark

```
vector<int> marked_graph_compressed::type_mark
```

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

#### 6.15.2.9 ver_type_list

```
vector<vector<int> > marked_graph_compressed::ver_type_list
```

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

#### 6.15.2.10 ver_types

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

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

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

- marked_graph_compression.h
- marked_graph_compression.cpp

## 6.16 marked_graph_decoder Class Reference

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

### Public Member Functions

- marked_graph_decoder ()
  *constructor*
- marked_graph decode (const marked_graph_compressed &)

### Private Member Functions

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

### Private Attributes

- int h
- int delta
- int n
  *number of vertices, this is set when a graph G is given to be encoded*
- vector$<$ int $>$ is_star_vertex
  *for $0 \leq v < n$, is_star_vertex[v] is 1 if there is at least one star type edge connected to v and 0 otherwise.*
- vector$<$ int $>$ star_vertices
  *the list of star vertices*
- map$<$ pair$<$ int, int $>$, b_graph $>$ part_bgraph
  *for $0 \leq t < t' < L$, part_bgraph[pair<int, int> (t,t')] is a bipartite graph with n left vertex and n right vertex. In this bipartite graph, a left vertex i is connected to a right vertex j iff there is an edge in the graph between vertices i and j with a half edge type towards i equal to t and a half edge type towards j equal to t'.*
- map$<$ int, graph $>$ part_graph

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

- vector< pair< pair< int, int >, pair< int, int > > > edges

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

- vector< int > vertex_marks

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

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

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

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

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

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

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

## 6.16.1 Constructor & Destructor Documentation

### 6.16.1.1 marked_graph_decoder()

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

constructor

```
{}
```

## 6.16.2 Member Function Documentation

### 6.16.2.1 decode()

```
marked_graph marked_graph_decoder::decode (
            const marked_graph_compressed & compressed )
```

```
{
  logger::current_depth++;
  logger::add_entry("Init", "");
  n = compressed.n;
  h = compressed.h;
  delta = compressed.delta;


  edges.clear(); // clear the edge list of the marked graph to be decoded
  vertex_marks.clear(); // clear the list of vertex marks of the marked graph to be decoded

  logger::add_entry("Decode * vertices", "");
  decode_star_vertices(compressed);
  //cerr << " decoded star vertices " << endl;

  logger::add_entry("Decode * edges", "");
```

```
  decode_star_edges(compressed);
  //cerr << " decoded star edges " << endl;

  logger::add_entry("Decode vertex types", "");
  decode_vertex_types(compressed);
  //cerr << " decoded vertex types " << endl;

  logger::add_entry("Decode partition graphs", "");
  decode_partition_graphs(compressed);
  //cerr << " decoded partition graphs " << endl;

  logger::add_entry("Decode partition b graphs", "");
  decode_partition_bgraphs(compressed);
  //cerr << " decoded partition b graphs " << endl;

  // now, reconstruct the original marked graphs by assembling the vertex marks and edge list
  logger::add_entry("Construct decoded graph", "");
  marked_graph G(n, edges, vertex_marks);

  logger::current_depth--;
  return G;
}
```

#### 6.16.2.2 decode_partition_bgraphs()

```
void marked_graph_decoder::decode_partition_bgraphs (
            const marked_graph_compressed & compressed )  [private]
```

```
{
  pair<int, int> c; // the pair of types
  int t, tp; // types
  int x, xp; // mark components of t and tp

  b_graph G; // the decoded partition bipartite graph
  int nl_G; // the number of left nodes in the partition graph G
  vector<int> adj_list; // adj list of a vertex in a partition bipartite graph
  int w; // a right node
  int v_orig, w_orig; // the original index of vertices v and w in partition graphs
  for (map<pair<int, int>, mpz_class>::const_iterator it = compressed.part_bgraph.begin(); it!=compressed.part_bgraph.end(); it++){
    c = it->first;
    t = c.first;
    tp = c.second;
    x = compressed.type_mark[t]; // the mark component of t
    xp = compressed.type_mark[tp]; // the mark component of tp

    //cerr << " t " << t << " tp " << tp << endl;

    b_graph_decoder D(part_deg.at(pair<int, int>(t,tp)), part_deg.at(pair<int, int>(tp,t))); // the degree sequence of left nodes is precisely part_deg.at(pair<int, int>(t,tp)), while that of the right nodes is precisely part_deg.at(pair<int, int>(tp,t))
    //cerr << " decoder constructed " << endl;
    //cerr << " part graph t = "  << t << " t' = " << tp << " nl " << part_deg.at(pair<int, int>(t,tp)).size() << " nr = " << part_deg.at(pair<int, int>(tp,t)).size() << endl;
    G = D.decode(it->second);

    //cerr << " G decoded " << endl;
    nl_G = part_deg.at(pair<int, int>(t,tp)).size(); // the number of left nodes in G is obtained from the size of the degree sequence of left nodes

    for (int v=0;v<nl_G;v++){
      v_orig = origin_index.at(pair<int, int>(t,tp))[v];
      //cerr << " v " << v << " v_orig " << v_orig << endl;
      adj_list = G.get_adj_list(v);
      for (int i=0;i<adj_list.size();i++){
        w = adj_list[i];
        w_orig = origin_index.at(pair<int, int>(tp,t))[w]; // since w is a right node, we should read its original index through origin_index[(tp,t)]
        //cerr << " w " << w << " w_orig " << w_orig << endl;
        edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v_orig,w_orig), pair<int, int>(x,xp)));
      }
    }
  }
}
```

### 6.16.2.3 decode_partition_graphs()

```
void marked_graph_decoder::decode_partition_graphs (
            const marked_graph_compressed & compressed )  [private]
```

```
{
  int t; // the type corresponding to the partition graph
  vector<int> t_message; // the actual message corresponding to t
  int x; // the mark component associated to t
  pair< mpz_class, vector< int > > G_compressed; // the compressed form of the partition graph
  graph G; // the decoded partition graph
  vector<int> flist; // the forward adjacency list of a vertex in a partition graph
  int w; // vertex in partition graph
  int v_orig, w_orig; // the original index of vertices v and w
  int n_G; // the number of vertices of the partitioned graph

  for(map< int, pair< mpz_class, vector< int > > >::const_iterator it=compressed.part_graph.begin(); it!=compressed.part_graph.end(); it++){
    t = it->first;
    x = compressed.type_mark[t]; // the mark component of t

    G_compressed = it->second;
    // the degree sequence of the graph can be obtained from part_deg.at(pair<int,int>(t,t))
    graph_decoder D(part_deg.at(pair<int, int>(t,t)));
    //cerr << " part_graph t = " << t << " with " << part_deg.at(pair<int, int>(t,t)).size() << " vertices " << endl;
    n_G = part_deg.at(pair<int, int>(t,t)).size(); // the number of vertices in the partition graph is read from the size of its degree sequence
    G = D.decode(G_compressed.first, G_compressed.second);
    // for each edge in G, we should add an edge with mark pair (x,x) to the edge list of the marked graph
    for (int v=0;v<n_G;v++){
      flist = G.get_forward_list(v);
      v_orig = origin_index.at(pair<int, int>(t,t))[v]; // the index of v in the original graph

      for (int i=0;i<flist.size();i++){
        w = flist[i]; // the other endpoint in the partition graph
        w_orig = origin_index.at(pair<int, int>(t,t))[w]; // the index of w in the original graph
        edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v_orig,w_orig), pair<int, int> (x,x)));
      }
    }
  }
}
```

### 6.16.2.4 decode_star_edges()

```
void marked_graph_decoder::decode_star_edges (
            const marked_graph_compressed & compressed )  [private]
```

```
{
  pair<int, int> mark_pair; // the pair of marks
  vector<vector<int> > list; // list of edges with this pair of marks
  int v; // one endpoint of the star edge
  // iterating through the star_edges map
  for (map<pair<int, int>, vector<vector<int> > >::const_iterator it = compressed.star_edges.begin(); it!=compressed.star_edges.end(); it++){
    mark_pair = it->first;
    //cerr << " mark_pair " << mark_pair.first << " " << mark_pair.second << endl;
    list = it->second;
    for (int i=0;i<list.size();i++){
      v = star_vertices[i];
      for (int j=0;j<list[i].size();j++){
        //cerr << " list[i][j] " << list[i][j] << endl;
        edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v,list[i][j]), mark_pair));
      }
    }
  }
}
```

#### 6.16.2.5 decode_star_vertices()

```
void marked_graph_decoder::decode_star_vertices (
            const marked_graph_compressed & compressed )  [private]
```

```
{
  time_series_decoder D(n);
  is_star_vertex = D.decode(compressed.star_vertices);

  star_vertices.clear();
  for (int i=0;i<n;i++)
    if (is_star_vertex[i] == 1)
      star_vertices.push_back(i);
}
```

#### 6.16.2.6 decode_vertex_types()

```
void marked_graph_decoder::decode_vertex_types (
            const marked_graph_compressed & compressed )  [private]
```

```
{
  time_series_decoder D(n);
  vector<int> ver_type_int = D.decode(compressed.ver_types);

  // converting the integer value vertex types to actual vectors using the 'ver_type_list' attribute of compressed

  vertex_marks.resize(n); // preparing for decoding vertex marks
  Deg.clear(); // refresh
  Deg.resize(n);

  vector<int> x; // auxiliary vector

  for (int v=0;v<n;v++){
    if (ver_type_int[v] >= compressed.ver_type_list.size())
      cerr << " Warning: marked_graph_decoder::decode_vertex_types ver_type_int[" << v << "] is out of range" << endl;
    x = compressed.ver_type_list[ver_type_int[v]];
    vertex_marks[v] =x[0]; // the mark of vertex v is the first element in the type list of this vertex
    // now, we extract Deg[v] by looking at batches of size 3 in x
    for (int i=1;i<x.size();i+=3){
      if (i+2 >= x.size())
        cerr << " Error: marked_graph_decoder::decode_vertex_types, the type of vertex " << v << " does not obey length constrains, i.e. it does not have length 1 + 3k " << endl;
        Deg[v][pair<int, int>(x[i],x[i+1])] = x[i+2]; // x[i] and x[i+1] are types, and x[i+2] is the count
    }
  }

  find_part_deg_orig_index(); // find part_deg and orig_index maps
}
```

#### 6.16.2.7 find_part_deg_orig_index()

```
void marked_graph_decoder::find_part_deg_orig_index ( )  [private]
```

```
{
  part_deg.clear();
  origin_index.clear();
  int t, tp; // types

  //cerr << " decoded deg : " << endl;
  for (int v=0;v<n;v++){
    //cerr << " v " << v << endl;
    for (map<pair<int, int>, int>::iterator it=Deg[v].begin(); it!=Deg[v].end(); it++){
      t = it->first.first;
      tp = it->first.second;
      //cerr << " t " << t << " tp " << tp << " : " << it->second << endl;
      if (part_deg.find(it->first) == part_deg.end()){
        // this is our first encounter with this type pair
        origin_index[it->first] = vector<int>({v}); // v is the first node in the t side of the (t,t') partition graph
        part_deg[it->first] = vector<int>({it->second}); // the degree in the partition graph is read from it->second
      }else{
        origin_index.at(it->first).push_back(v); // v is the next vertex observed with type t,t', so the vertex in the partition graph with index origin_index[it->first] has original index v
        // append degree of v, which is it->second
        part_deg.at(it->first).push_back(it->second);
      }
    }
  }
}
```

## 6.16.3 Member Data Documentation

### 6.16.3.1 Deg

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

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

### 6.16.3.2 delta

`int marked_graph_decoder::delta [private]`

### 6.16.3.3 edges

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

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

#### 6.16.3.4 h

```
int marked_graph_decoder::h [private]
```

#### 6.16.3.5 is_star_vertex

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

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

#### 6.16.3.6 n

```
int marked_graph_decoder::n [private]
```

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

#### 6.16.3.7 origin_index

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

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

#### 6.16.3.8 part_bgraph

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

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

#### 6.16.3.9 part_deg

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

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

#### 6.16.3.10 part_graph

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

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

#### 6.16.3.11 star_vertices

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

the list of star vertices

#### 6.16.3.12 vertex_marks

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

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

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

- marked_graph_compression.h
- marked_graph_compression.cpp

## 6.17 marked_graph_encoder Class Reference

`#include <marked_graph_compression.h>`

### Public Member Functions

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

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

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

## Private Member Functions

- void encode_star_vertices ()

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

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

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

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

    *update part_index and part_deg members*
- void extract_partition_graphs ()

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

    *encode partition bipartite graphs*
- void encode_partition_graphs ()

    *encodes partition simple graphs*

## Private Attributes

- int h
- int delta
- int n

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

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

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

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

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

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

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

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

    *the compressed version of the given graph in encode function*

## 6.17.1 Constructor & Destructor Documentation

### 6.17.1.1 marked_graph_encoder()

```
marked_graph_encoder::marked_graph_encoder (
        int h_,
        int delta_ )  [inline]
```

```
: h(h_), delta(delta_) {}
```

## 6.17.2 Member Function Documentation

### 6.17.2.1 encode() [1/2]

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

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

```
{
  logger::current_depth++;
  logger::add_entry("Init compressed", "");

  compressed.clear(); // reset the compressed variable before starting

  n = G.nu_vertices;
  compressed.n = n;
  compressed.h = h;
  compressed.delta = delta;

  logger::add_entry("Extact edge types", "");
  extract_edge_types(G);
  //cout << " edge types extracted " << endl;


  compressed.ver_type_list = C.ver_type_list; //
  compressed.type_mark = C.M.message_mark;

  /*
  cout << " message list " << endl;
  for (int i=0;i<C.M.message_list.size();i++){
    cout << i << " : ";
    for (int j=0;j<C.M.message_list[i].size();j++)
      cout << C.M.message_list[i][j] << " ";
    cout << endl;
  }
  */

  logger::add_entry("Encode * vertices", "");
  encode_star_vertices(); // encode the list of vertices with at least one star edge connected to them
  //cout << " encoded star vertices " << endl;

  logger::add_entry("Encode * edges", "");
  encode_star_edges(); // encode edges with star types, i.e. those with half edge type L or larger
  //cout << " encoded star edges " << endl;

  logger::add_entry("Encode vertex types", "");
  encode_vertex_types(); // encode the sequences \f$\vec{\beta}, \vec{D}\f$, which is
```

```
 encoded in C.ver_type
//cout << " encoded vertex types " << endl;

logger::add_entry("Extract partition graphs", "");
extract_partition_graphs(); // for equality types, we form simple unmarked
 graphs, and for inequality types, we form a bipartite graph
//cout << " extracted partition graphs " << endl;
/*
  cout << " partition bipartite graphs " << endl;
for (map<pair<int, int>, b_graph>::iterator it = part_bgraph.begin(); it!=part_bgraph.end(); it++){
  cout << " c = " << it->first.first << " , " << it->first.second << endl;
  cout << it->second << endl;
}

cout << " partition simple graphs " << endl;
for (map<int, graph>::iterator it = part_graph.begin();it!=part_graph.end();it++){
  cout << " t = " << it->first << endl;
  cout << it->second << endl;
}
*/

logger::add_entry("Encode partition b graphs", "");
encode_partition_bgraphs();
//cout << " encoded partition bgraphs " << endl;

logger::add_entry("Encode partition graphs", "");
encode_partition_graphs();
//cout << " encoded partition graphs " << endl;

logger::current_depth--;
return compressed;
}
```

#### 6.17.2.2 encode() [2/2]

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

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

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

#### 6.17.2.3 encode_partition_bgraphs()

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

encode partition bipartite graphs

```
{
 int t, tp;

 // compressing bipartite graphs
 for (map<pair<int, int> , b_graph>::iterator it = part_bgraph.begin(); it!=
part_bgraph.end(); it++){
   // the color components are t, tp
   t = it->first.first;
   tp = it->first.second;
   b_graph_encoder E(part_deg.at(pair<int, int>(t,tp)),
part_deg.at(pair<int, int>(tp, t)));
   compressed.part_bgraph[pair<int, int>(t,tp)] = E.encode(it->second);
 }

}
```

#### 6.17.2.4 encode_partition_graphs()

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

encodes partition simple graphs

```
{
  int t;

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

#### 6.17.2.5 encode_star_edges()

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

Encodes star edges to the star_edges attribute of compressed.

```
{
  int x, xp; // auxiliary mark variables
  int w; // auxiliary vertex variable
  int v; // auxiliary vertex variable
  for (int k=0; k<star_vertices.size(); k++){ // iterating over star vertices
    v = star_vertices[k];
    for (int i=0;i<C.adj_list[v].size();i++){
      if (C.M.is_star_message[C.adj_list[v][i].second.first] or C.M.is_star_message[C.adj_list[v][i].second.second]){ // this is a star edge
        x = C.M.message_mark[C.adj_list[v][i].second.first]; // mark towards v
        xp = C.M.message_mark[C.adj_list[v][i].second.second]; // mark towards other endpoint
        w = C.adj_list[v][i].first; // the other endpoint of the edge
        if (x < xp){ // if x > xp, we only store this edge when visiting the other endpoint (w), since we do not want to express an edge twice
          if (compressed.star_edges.find(pair<int, int>(x,xp)) == compressed.star_edges.end()) // this pair does not exist
            compressed.star_edges[pair<int, int>(x,xp)].resize(star_vertices.size()); // open space for all star vertices
          compressed.star_edges.at(pair<int, int>(x,xp))[k].push_back(w); // add w to the position of v (which is k)
        }
        if (x == xp and w > v){ // if w < v, we store this edge when visiting the other endpoint (w) to avoid storing and edge twice
          if (compressed.star_edges.find(pair<int, int>(x,xp)) == compressed.star_edges.end()) // not yet exist
            compressed.star_edges[pair<int, int>(x,xp)].resize(star_vertices.size()); // open space
          compressed.star_edges.at(pair<int, int>(x,xp))[k].push_back(w);
        }
      }
    }
  }
}
```

#### 6.17.2.6 encode_star_vertices()

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

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

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

```
{
  // compress the is_star_vertex list
  time_series_encoder star_encoder(n);
  vector<int> is_star_vertex_int(is_star_vertex.size());
  for (int i=0;i<is_star_vertex.size();i++){
    if(is_star_vertex[i] == true)
      is_star_vertex_int[i] = 1;
    else
      is_star_vertex_int[i] = 0;
  }
  compressed.star_vertices = star_encoder.encode(is_star_vertex_int);
}
```

#### 6.17.2.7 encode_vertex_types()

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

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

```
{
  time_series_encoder vtype_encoder(n);
  //cerr << " C.ver_type_int " << endl;
  //for (int i=0;i<C.ver_type_int.size();i++)
  //  cerr << C.ver_type_int[i] << " ";
  //cerr << endl;
  compressed.ver_types = vtype_encoder.encode(C.ver_type_int);
}
```

#### 6.17.2.8 extract_edge_types()

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

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

```
{
  // extracting edges types (aka colors)
  logger::current_depth++;
  logger::add_entry("Extract messages", "");
  C = colored_graph(G, h, delta);
  //cerr << " number of types " << C.M.message_mark.size() << endl;
  is_star_vertex = C.is_star_vertex;
  star_vertices = C.star_vertices;
  logger::current_depth--;
}
```

#### 6.17.2.9 extract_partition_graphs()

void marked_graph_encoder::extract_partition_graphs ( ) [private]

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

```
{
  find_part_index_deg();

  // for t \leq t', part_adj_list[(t,t')] is the adjacency list of the partition graph t,t'. If t < t', this is the adjacency list of the left nodes, if t = t', this is the forward adjacency list of the partition graph.

  map<pair<int, int>, vector<vector<int> > > part_adj_list;
  int t, tp; // types
  for (map<pair<int, int>, vector<int> >::iterator it = part_deg.begin(); it!= part_deg.end(); it++){
    // search over all type pairs in part_deg
    t = it->first.first;
    tp = it->first.second;
    // t < t': bipartite, t = t': simple. In both cases,
    if (t <= tp)
      part_adj_list[it->first] = vector<vector<int> >(it->second.size());
  }

  // going over the edges in the graph and forming partition_adj_list
  int w, p, q; // auxiliary variables
  for (int v =0; v<n; v++){
    for (int i=0;i<C.adj_list[v].size();i++){
      w = C.adj_list[v][i].first; // the other endpoint
      t = C.adj_list[v][i].second.first; // color towards v
      tp = C.adj_list[v][i].second.second; // color towards w
      if (C.M.is_star_message[t] == false and C.M.is_star_message[tp] == false){
        p = part_index[v].at(pair<int, int>(t,tp)); // the index of v in the t part of the t,tp partition graph
        //cerr << " p " << p << endl;
        q = part_index[w].at(pair<int, int>(tp, t)); // the index of w in the tp part of the t,tp partition graph
        //cerr << " q " << q << endl;
        if (t < tp)
          part_adj_list.at(pair<int, int>(t,tp))[p].push_back(q);
        if ((t == tp) and (q > p))
          part_adj_list.at(pair<int, int>(t,t))[p].push_back(q);
      }
    }
  }

  // using partition_adj_list in order to construct partition graphs
  //if(logger::stat){
  //*logger::stat_stream << " partition graphs size: " << endl;
  //*logger::stat_stream << " ====================== " << endl;
  //}
  for (map<pair<int, int>, vector<vector<int> > >::iterator it=part_adj_list.begin(); it!=part_adj_list.end();it++){
    t = it->first.first;
    tp = it->first.second;
    if (t<tp){
      part_bgraph[it->first] = b_graph(it->second, part_deg.at(pair<int, int>(t, tp)), part_deg.at(pair<int, int>(tp, t))); // left and right degree sequences are read from the part_deg map
      //if (logger::stat){
      //*logger::stat_stream << " bipartite: (" << part_deg.at(pair<int, int> (t,tp)).size() << " , " << part_deg.at(pair<int, int>(tp,t)).size() << ")" << endl;
      //}
    }
    if (t == tp){
      part_graph[t] = graph(it->second, part_deg.at(pair<int, int>(t,t)));
      //if (logger::stat){
      //*logger::stat_stream << " simple: "  << part_deg.at(pair<int, int>(t,t)).size() << endl;
      //}
    }
  }
}
```

#### 6.17.2.10 find_part_index_deg()

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

update part_index and part_deg members

```
{
  // extracting part_index and part_deg
  part_index.resize(n);
  for (int v =0; v<n; v++){
    for (map< pair< int, int >, int >::iterator it = C.deg[v].begin(); it != C.deg[v].end(); it++){
      if (part_deg.find(it->first) == part_deg.end()){
        // this pair has not been observed yet in the graph
        // so v is the first index node
        part_index[v][it->first] = 0;
        // the degree of v in the partition graph is indeed it->second
        part_deg[it->first] = vector<int>({it->second});
      }else{
        // there are currently part_deg[it->first].size() many elements there, and v is the last arrival one, so its index is equal to the number of existing nodes
        part_index[v][it->first] = part_deg.at(it->first).size();
        // append degree of v, which is it->second
        part_deg.at(it->first).push_back(it->second);
      }
    }
  }
}
```

### 6.17.3 Member Data Documentation

#### 6.17.3.1 C

`colored_graph marked_graph_encoder::C [private]`

the auxiliary object to extract edge types

#### 6.17.3.2 compressed

`marked_graph_compressed marked_graph_encoder::compressed [private]`

the compressed version of the given graph in encode function

#### 6.17.3.3 delta

`int marked_graph_encoder::delta [private]`

#### 6.17.3.4 h

```
int marked_graph_encoder::h [private]
```

#### 6.17.3.5 is_star_vertex

```
vector<bool> marked_graph_encoder::is_star_vertex [private]
```

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

#### 6.17.3.6 n

```
int marked_graph_encoder::n [private]
```

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

#### 6.17.3.7 part_bgraph

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

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

#### 6.17.3.8 part_deg

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

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

#### 6.17.3.9 part_graph

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

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

#### 6.17.3.10 part_index

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

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

#### 6.17.3.11 star_vertices

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

the list of star vertices

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

- marked_graph_compression.h
- marked_graph_compression.cpp

## 6.18 obitstream Class Reference

handles writing bitstreams to binary files

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

### Public Member Functions

- obitstream (string file_name)

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

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

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

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

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

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

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

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

### Private Member Functions

- void write ()

    *writes complete chunks to the output*

### Private Attributes

- bit_pipe buffer

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

    *the number of chunks written to the output so far*

## 6.18.1 Detailed Description

handles writing bitstreams to binary files

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

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

## 6.18.2 Constructor & Destructor Documentation

### 6.18.2.1 obitstream()

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

constructor

```
                             {
    f = fopen(file_name.c_str(), "wb+");
    chunks_written = 0;
 }
```

## 6.18.3 Member Function Documentation

### 6.18.3.1 bin_inter_code() [1/2]

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

uses binary interpolative coding to encode an increasing sequence of integers

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

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

**Parameters**

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

```
{
  // write a.size

  write_bits(a.size(),b);
  if (a.size() == 0)
    return;
  if (a.size()==1){
    write_bits(a[0],b);
    return;
  }
  // write low and high values in a
  write_bits(a[0], b);
  write_bits(a[a.size()-1], b);

  // then, encode recursively
  bin_inter_code(a, 0, a.size()-1, a[0], a[a.size()-1]);
}
```

#### 6.18.3.2 bin_inter_code() [2/2]

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

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

**Parameters**

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

```
{
  if (j < i)
    return;
  if (i==j){
    // we should encode a[i] using the assumption that it is bounded by high - low
    // therefore low <= a[i] <= high
    // so 0 <= a[i]-low <= high - low
    // so we can encode a[i]-low using nu_bits(high - low) bits
    if (high > low) // otherwise, there will be nothing to be printed
      write_bits(a[i] - low, nu_bits(high-low));
    return;
  }
  // find the intermediate value
  int m = (i+j)/ 2;
  unsigned int L = low + m - i; // lower bound on a[m]
  unsigned int H = high - (j - m); // upper bound on a[m]
  // so L <= a[m] <= H
  // and we can encode a[m] - L using nu_bits(H-L) bits
```

```
  if (H > L) // otherwise, a[m] is clearly H = L and nothing need to be written
    write_bits(a[m] - L, nu_bits(H-L));

  // then, we should recursively encode intervals [i,m-1] and [m+1, j]
  bin_inter_code(a, i, m-1, low, a[m]-1);
  bin_inter_code(a, m+1, j, a[m]+1, high);
}
```

#### 6.18.3.3 chunks()

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

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

```
                    {
    return chunks_written;
  }
```

#### 6.18.3.4 close()

`void obitstream::close ( )`

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

```
                   {
  if (buffer.bits.size() > 0){
    fwrite(&buffer.bits[0], sizeof(unsigned int), buffer.bits.size(), f);
    buffer.bits.clear();
    buffer.last_bits = 0;
  }
  fclose(f);
}
```

#### 6.18.3.5 operator<<() [1/2]

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

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

```
                                                     {
  if (buffer.bits.size() > 1){
    cerr << " ERROR: buffer has more than 1 chunk! " << endl;
  }
  unsigned int buffer_backup = 0; // the backup of the remaining chunk in
  int buffer_res = 0;
  if (buffer.bits.size() != 0){
    buffer_backup = buffer.bits[0];
    buffer_res = buffer.last_bits;
  }
  elias_delta_encode(n+1, buffer); // find the delta encoded version of n + 1
  buffer.shift_right(buffer_res); // open up space for the residual of the previous operation
  buffer.bits[0] |= buffer_backup; // add the residual
  write();
  return *this;
}
```

#### 6.18.3.6 operator<<() [2/2]

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

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

```
                                                     {
  if (buffer.bits.size() > 1){
    cerr << " ERROR: buffer has more than 1 chunk! " << endl;
  }
  unsigned int buffer_backup = 0; // the backup of the remaining chunk in
  int buffer_res = 0;
  if (buffer.bits.size() != 0){
    buffer_backup = buffer.bits[0];
    buffer_res = buffer.last_bits;
  }
  elias_delta_encode(n+1, buffer); // find the delta encoded version of n + 1
  buffer.shift_right(buffer_res); // open up space for the residual of the previous operation
  buffer.bits[0] |= buffer_backup; // add the residual
  write();
  return *this;
}
```

#### 6.18.3.7 write()

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

writes complete chunks to the output

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

#### 6.18.3.8 write_bits()

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

write the bits given as unsigned int to the output

**Parameters**

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

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

### 6.18.4 Member Data Documentation

#### 6.18.4.1 buffer

`bit_pipe obitstream::buffer [private]`

a bit_pipe carrying the buffered data

#### 6.18.4.2 chunks_written

`unsigned int obitstream::chunks_written [private]`

the number of chunks written to the output so far

#### 6.18.4.3 f

`FILE* obitstream::f [private]`

pointer to the binary output file

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

- bitstream.h
- bitstream.cpp

## 6.19 reverse_fenwick_tree Class Reference

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

`#include <fenwick.h>`

## Public Member Functions

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

## Private Attributes

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

### 6.19.1 Detailed Description

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

### 6.19.2 Constructor & Destructor Documentation

#### 6.19.2.1 reverse_fenwick_tree() [1/2]

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

default constructor

```
{}
```

#### 6.19.2.2 reverse_fenwick_tree() [2/2]

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

constructor which receives values and initializes

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

### 6.19.3 Member Function Documentation

#### 6.19.3.1 add()

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

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

**Parameters**

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

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

#### 6.19.3.2 size()

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

the number of elements in the original array

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

#### 6.19.3.3 sum()

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

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

**Parameters**

| | |
|---|---|
| *k* | the index from which (including) the sum is computed |

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

### 6.19.4 Member Data Documentation

#### 6.19.4.1 FT

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

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

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

- fenwick.h
- fenwick.cpp

## 6.20 time_series_decoder Class Reference

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

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

### Public Member Functions

- time_series_decoder (int n_)

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

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

### Private Attributes

- int n

  *the length*
- int alph_size

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

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

  *the decoded bipartite graph as in* `time_series_encoder`

### 6.20.1 Detailed Description

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

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

**Usage Example**

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

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

## 6.20.2 Constructor & Destructor Documentation

### 6.20.2.1 time_series_decoder()

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

constructor

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

## 6.20.3 Member Function Documentation

### 6.20.3.1 decode()

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

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

```
{
  freq = E.first;
  mpz_class f = E.second;
  vector<int> left_deg(n,1); // the left degree sequence
  b_graph_decoder D(left_deg, freq); // the bipartite graph decoder to convert f to G
  G = D.decode(f);

  // reconstructing the original sequence given G
  vector<int> x(n);
  vector<int> adj_list;
  for (int i=0;i<n;i++){
    adj_list = G.get_adj_list(i);
    x[i] = adj_list[0];
  }
  return x;
}
```

## 6.20.4 Member Data Documentation

### 6.20.4.1 alph_size

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

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

#### 6.20.4.2 freq

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

the frequency of symbols after decoding, so has size alph_size

#### 6.20.4.3 G

```
b_graph time_series_decoder::G  [private]
```

the decoded bipartite graph as in `time_series_encoder`

#### 6.20.4.4 n

```
int time_series_decoder::n  [private]
```

the length

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

- time_series_compression.h
- time_series_compression.cpp

## 6.21 time_series_encoder Class Reference

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

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

### Public Member Functions

- time_series_encoder (int n_)

  *constructor*
- pair< vector< int >, mpz_class > encode (const vector< int > &x)

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

### Private Member Functions

- void init_alph_size (const vector< int > &x)

  *initializes the alphabet size, i.e. the variable init_alph_size*
- void init_freq (const vector< int > &x)

  *initializes the freq vector*
- void init_G (const vector< int > &x)

  *initializes the auxiliary bipartite graph G*

### Private Attributes

- int n
  *length of the series is assumed to be known*
- int alph_size
  *the number of distinct integers showing up in the sequence. Therefore, the sequence would consists of integers in the range [0,alph_size-1].*
- vector< int > freq
  *the frequency of symbols, so has size alph_size*
- b_graph G
  *the bipartite graph version of the sequence, which has n left vertices and alph_size right sequence. Left vertex v is connected to right vertex w if the value of the time series in coordinate v is w. This way, each vertex on the left has degree 1, and the degree of a right vertex w is the frequency of w.*

## 6.21.1 Detailed Description

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

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

**Usage Example**

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

See `time_series_decoder` for decoding.

## 6.21.2 Constructor & Destructor Documentation

### 6.21.2.1 time_series_encoder()

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

constructor

```
44 : n(n_) {}
```

## 6.21.3 Member Function Documentation

### 6.21.3.1 encode()

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

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

#### Parameters

| | |
|---|---|
| *x* | const reference to the array to be compressed. |

#### Returns

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

```
{
  //check whether x is a compatible sequence
  if (x.size()!= n)
    cerr << " WARNING: time_series_encoder::encode, called for a vector with size different from n,
    x.size() = " << x.size() << endl;

  // initialize alph_size, freq and G
  //logger::item_start("time series init alph size");
  init_alph_size(x);
  //logger::item_stop("time series init alph size");
  //logger::item_start("time series init freq");
  init_freq(x);
  //logger::item_stop("time series init freq");

  //logger::item_start("time series init G");
  init_G(x);
  //logger::item_stop("time series init G");

  // initializing a b_graph_encoder
  vector<int> left_deg(n, 1); // the left degree sequence
  b_graph_encoder E(left_deg, freq); // the right degree sequence is freq
  //logger::item_start("time series encode");
  mpz_class f = E.encode(G);
  //logger::item_stop("time series encode");
  pair<vector<int>, mpz_class> ans;
  ans.first = freq;
  ans.second  = f;
  return ans;
}
```

### 6.21.3.2 init_alph_size()

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

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

```
{
  alph_size = 0;
  for (int i=0;i<x.size();i++){
    if (x[i] > alph_size)
      alph_size = x[i];
    if (x[i] < 0)
      cerr << " WARNING: time_series_encoder::encode called for a vector with negative entries " << endl;
  }

  alph_size ++; // the array is zero based
}
```

#### 6.21.3.3 init_freq()

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

initializes the freq vector

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

#### 6.21.3.4 init_G()

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

initializes the auxiliary bipartite graph G

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

### 6.21.4 Member Data Documentation

#### 6.21.4.1 alph_size

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

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

#### 6.21.4.2 freq

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

the frequency of symbols, so has size alph_size

#### 6.21.4.3 G

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

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

#### 6.21.4.4 n

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

length of the series is assumed to be known

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

- time_series_compression.h
- time_series_compression.cpp

## 6.22 vint_hash Struct Reference

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

### Public Member Functions

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

### 6.22.1 Member Function Documentation

#### 6.22.1.1 operator()()

```
size_t vint_hash::operator() (
            vector< int > const & v ) const
```

```
{
  return boost::hash_range(v.begin(),v.end());
}
```

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

- graph_message.h
- graph_message.cpp

# Chapter 7

# File Documentation

## 7.1 bipartite_graph.cpp File Reference

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

### Functions

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

### 7.1.1 Function Documentation

#### 7.1.1.1 operator"!=()

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

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

#### 7.1.1.2 operator<<()

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

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

#### 7.1.1.3 operator==()

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

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

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

## 7.2 bipartite_graph.h File Reference

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

### Classes

- class b_graph

  *simple unmarked bipartite graph*

## 7.3 bipartite_graph_compression.cpp File Reference

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

## 7.4 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.*

## 7.5 bitstream.cpp File Reference

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

### Functions

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

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

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

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

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

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

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

### 7.5.1 Function Documentation

#### 7.5.1.1 elias_delta_encode() [1/4]

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

returns the Elias delta representation of an integer in bit_pipe format

```
                                                    {
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  bit_pipe n_pipe(n); // binary representation of n
  n_pipe.shift_left(1); // remove the leading 1
  n_pipe.append_left(N);
  return n_pipe;
}
```

#### 7.5.1.2 elias_delta_encode() [2/4]

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

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

```
                                                             {
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  B = bit_pipe(n); // binary representation of n
  B.shift_left(1); // remove the leading 1
  B.append_left(N);
}
```

### 7.5.1.3 elias_delta_encode() [3/4]

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

returns the Elias delta representation of an mpz_class in bit_pipe format

```
                                           {
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  bit_pipe n_pipe(n); // binary representation of n
  n_pipe.shift_left(1); // remove the leading 1
  n_pipe.append_left(N);
  return n_pipe;
}
```

### 7.5.1.4 elias_delta_encode() [4/4]

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

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

```
                                                   {
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  B = bit_pipe(n); // binary representation of n
  B.shift_left(1); // remove the leading 1
  B.append_left(N);
}
```

### 7.5.1.5 mask_gen()

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

generates a binary mask with n consecutive ones in LSB

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

```
                           {
  if (n < 1 or n > BIT_INT){
    cerr << " ERROR: mask_gen called for n outside the range [1,BIT_INT] " << endl;
    return 0;
  }
  unsigned int mask = 1;
  for (int i=1; i<n; i++){
    mask <<= 1;
    mask += 1;
  }
  return mask;
}
```

#### 7.5.1.6 nu_bits()

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

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

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

```
{
  int nu_bits = 0;
  unsigned int n_copy = n;
  while (n_copy > 0){
    nu_bits ++;
    n_copy >>= 1;
  }
  return nu_bits;
}
```

#### 7.5.1.7 operator<<() [1/2]

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

```
{
  if (B.bits.size()==0){
    o << "<>";
    return o;
  }
  o << "<";
  for (int i=0;i<(B.bits.size()-1); i++){ // the last byte requires special handling
    bitset<BIT_INT> b(B.bits[i]);
    o << b << " ";
  }
  unsigned int last_byte = B.bits[B.bits.size()-1];

  for (int k=BIT_INT;k>(BIT_INT-B.last_bits);k--){ // starting from MSB bit to LSB for existing bits
    if (last_byte & (1<<(k-1)))
      o << "1";
    else
      o << "0";
  }
  o << "|"; // to show the place of the last bit
  for (int k=BIT_INT-B.last_bits; k>=1; k--){
    if (last_byte &(1<<(k-1)))
      o << "1";
    else
      o << "0";
  }
  o << ">";
  return o;
}
```

#### 7.5.1.8 operator<<() [2/2]

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

```
{
  bit_pipe ans = B;
  ans.shift_left(n);
  return ans;
}
```

#### 7.5.1.9 operator>>()

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

```
{
    bit_pipe ans = B;
    ans.shift_right(n);
    return ans;
}
```

## 7.6 bitstream.h File Reference

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

### Classes

- class bit_pipe

    *A sequence of arbitrary number of bits.*
- class obitstream

    *handles writing bitstreams to binary files*
- class ibitstream

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

### Functions

- unsigned int nu_bits (unsigned int n)

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

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

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

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

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

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

### Variables

- const unsigned int BYTE_INT = sizeof(unsigned int)
- const unsigned int BIT_INT = 8 $*$ sizeof(unsigned int)

### 7.6.1 Function Documentation

#### 7.6.1.1 elias_delta_encode() [1/4]

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

returns the Elias delta representation of an integer in bit_pipe format

```
{
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor
   \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  bit_pipe n_pipe(n); // binary representation of n
  n_pipe.shift_left(1); // remove the leading 1
  n_pipe.append_left(N);
  return n_pipe;
}
```

#### 7.6.1.2 elias_delta_encode() [2/4]

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

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

```
{
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = nu_bits(n); // or equivalently \f$\lfloor \log_2 n \rfloor + 1\f$
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor
   \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  B = bit_pipe(n); // binary representation of n
  B.shift_left(1); // remove the leading 1
  B.append_left(N);
}
```

#### 7.6.1.3 elias_delta_encode() [3/4]

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

returns the Elias delta representation of an mpz_class in bit_pipe format

```
                                        {
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  bit_pipe n_pipe(n); // binary representation of n
  n_pipe.shift_left(1); // remove the leading 1
  n_pipe.append_left(N);
  return n_pipe;
}
```

#### 7.6.1.4 elias_delta_encode() [4/4]

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

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

```
                                                {
  if (n == 0){
    cerr << " ERROR: elias delta called for 0, input must be a positive integer" << endl;
  }
  // first, find number of bits in n
  int n_bits = mpz_sizeinbase(n.get_mpz_t(), 2); // number of bits in n
  int L = nu_bits(n_bits) - 1; // this is \f$\floor \og_2 (N+1) \rfloor\f$ where \f$N = \lfloor \log_2 n \rfloor\f$

  bit_pipe N(n_bits); // binary representation
  N.shift_right(L); // it is as if I write L zeros followed by binary representation of N
  B = bit_pipe(n); // binary representation of n
  B.shift_left(1); // remove the leading 1
  B.append_left(N);
}
```

#### 7.6.1.5 mask_gen()

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

generates a binary mask with n consecutive ones in LSB

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

```
                           {
  if (n < 1 or n > BIT_INT){
    cerr << " ERROR: mask_gen called for n outside the range [1,BIT_INT] " << endl;
    return 0;
  }
  unsigned int mask = 1;
  for (int i=1; i<n; i++){
    mask <<= 1;
    mask += 1;
  }
  return mask;
}
```

#### 7.6.1.6 nu_bits()

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

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

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

```
                                {
  int nu_bits = 0;
  unsigned int n_copy = n;
  while (n_copy > 0){
    nu_bits ++;
    n_copy >>= 1;
  }
  return nu_bits;
}
```

### 7.6.2 Variable Documentation

#### 7.6.2.1 BIT_INT

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

#### 7.6.2.2 BYTE_INT

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

## 7.7 compression_helper.cpp File Reference

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

### Functions

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

  *This function computes the product of consecutive integers separated by a given iteration. This is the old version which uses standard recursion.*
- mpz_class compute_product_stack (int N, int k, int s)

  *This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.*
- void compute_product_void (int N, int k, int s)
- void compute_array_product (vector< mpz_class > &a)

  *computes the product of elements in vector a by inline multiplication of adjacent elements recursively. The results will be in a[0].*
- mpz_class compute_product (int N, int k, int s)
- mpz_class binomial (const int n, const int m)

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

  *computes the product of factorials in a vector given a range*
- mpz_class prod_factorial (const vector< int > &a, int i, int j)
- int bit_string_write (FILE ∗f, const string &s)

  *Write a string containing 0 and 1 to a binary file, treating the string as a bit sequence. Returns the number of bytes written to the output.*
- string bit_string_read (FILE ∗f)

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

## 7.7.1 Function Documentation

#### 7.7.1.1 binomial()

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

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

**Parameters**

| | |
|---|---|
| *n* | integer |
| *m* | integer |

**Returns**

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

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

#### 7.7.1.2 bit_string_read()

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

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

**Parameters**

| | |
|---|---|
| *f* | a file pointer |

**Returns**

a string of zeros and ones.

```
{
  int nu_bytes;
  int ssize;
  // read the number of bytes to read
  fread(&ssize, sizeof(ssize), 1, f);
  //cerr << " ssize " << ssize << endl;
  nu_bytes = ssize / 8;
  if (ssize % 8 != 0)
    nu_bytes ++;

  int last_byte_size = ssize % 8;
  if (last_byte_size == 0)
    last_byte_size  = 8;

  unsigned char c;
  bitset<8> B;
  string s;
  for (int i=0;i<nu_bytes;i++){
    fread(&c, sizeof(c), 1, f);
    B = c;
    //cout << B << endl;
    if (i < nu_bytes -1){
      s += B.to_string();
    }else{
      s += B.to_string().substr(8-last_byte_size, last_byte_size);
    }
  }
  return s;
}
```

#### 7.7.1.3 bit_string_write()

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

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

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

**Parameters**

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

**Returns**

the number of bytes written to the output

```
                                              {
  // find out the number of bytes
  int ssize = s.size();
  int nu_bytes; // number of bytes wrote to the output

  //if (ssize % 8 != 0) // an incomplete byte is required
  //nu_bytes++;

  fwrite(&ssize, sizeof(ssize), 1, f); // first, write down how many bytes are coming.
  nu_bytes += sizeof(ssize);

  stringstream ss;
  ss << s;

  bitset<8> B;
  unsigned char c;
  while (ss >> B){
    c = B.to_ulong();
    fwrite(&c, sizeof(c), 1, f);
    nu_bytes += sizeof(c);
  }
  return nu_bytes;
}
```

#### 7.7.1.4 compute_array_product()

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

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

```
                                                {
  //logger::item_start("Compute Array Product");
  int step_size, to_mul;
  int k = a.size();
  for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
    for (int i=0; i<k; i+=step_size){
      if (i+to_mul < k)
        a[i] *= a[i+to_mul];
    }
  }
  //logger::item_stop("Compute Array Product");
}
```

#### 7.7.1.5 compute_product()

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

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

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

  if (k == 2){
    helper_vars::mul_1 = N;
    helper_vars::mul_2 = N - s;
    return helper_vars::mul_1 * helper_vars::mul_2;
  }

  helper_vars::mpz_vec.resize(k);
  for (int i=0;i<k;i++)
    helper_vars::mpz_vec[i] = N - i * s;

  compute_array_product(helper_vars::mpz_vec);

  // int step_size, to_mul;


  // for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
  //   for (int i=0; i<k; i+=step_size){
  //     if (i+to_mul < k)
  //       helper_vars::mpz_vec[i] *= helper_vars::mpz_vec[i+to_mul];
  //   }
  // }
  return helper_vars::mpz_vec[0];
}
```

#### 7.7.1.6 compute_product_old()

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

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

**Parameters**

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

**Returns**

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

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

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

```
    return 1;

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

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

#### 7.7.1.7 compute_product_stack()

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

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

**Parameters**

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

**Returns**

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

```
                                                 {

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

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

  if (k == 2){
    helper_vars::mul_1 = N;
    helper_vars::mul_2 = N - s;
    return helper_vars::mul_1 * helper_vars::mul_2;
```

```
  }

  logger::item_start("CP body");

  int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
  int k_copy = k;
  while (k_copy > 0){
    k_bits ++;
    k_copy >>= 1;
  }
  k_bits += 2;
  vector<pair<int, int> > call_stack(2 * k_bits);
  //cout << " 2 * k_bits " << 2 * k_bits << endl;
  int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
  vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
  //vector<mpz_class> return_stack(2 * k_bits);
  helper_vars::return_stack.resize(2*k_bits);
  int return_pointer = 0;

  call_stack[call_pointer] = pair<int, int> (N, k);
  status_stack[call_pointer] = 0;
  call_pointer ++;

  int m;
  int N_now, k_now; // N and k for the current stack element

  while (call_pointer > 0){
    N_now = call_stack[call_pointer-1].first;
    k_now = call_stack[call_pointer-1].second;
    //cout << "call_pointer = " << call_pointer << " N = " << N_now << " k = " << k_now << " stat = " << status_stack[call_pointer-1] << endl;
    //cout << " the whole stack " << endl;
    //for (int i=0;i<call_pointer; i++){
    //  cout << call_stack[i].first << " , " << call_stack[i].second << " " << status_stack[i] << endl;
    //}
    if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
      // to collect two top elements in return stack and multiply them
      logger::item_start("CP arithmetic");
      helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] * helper_vars::return_stack[return_pointer-1];
      logger::item_stop("CP arithmetic");
      return_pointer--; // remove two items, add one item
      call_pointer --;
     }else{
       //cout << " else " << endl;
       if(k_now == 1){
         // to return the corresponding N
         helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
         call_pointer --; // pop this element
       }
       if (k_now == 2){
         helper_vars::mul_1 = N_now;
         helper_vars::mul_2 = N_now - s;
         logger::item_start("CP arithmetic");
         helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
         logger::item_stop("CP arithmetic");
         call_pointer --;
       }
       if (k_now > 2){
         m = k_now / 2;
         status_stack[call_pointer-1] = 1; // when return to this state, we know that we should aggregate
         call_stack[call_pointer] = pair<int, int>(N_now, m);
         status_stack[call_pointer] = 0; // just added
         call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
         status_stack[call_pointer+1]  = 0;
         call_pointer += 2;
       }
     }
   }
   // make sure there is exactly one element in return stack
   if (return_pointer != 1){
     cerr << " return pointer is not zero";
   }
   logger::item_stop("CP body");
   return helper_vars::return_stack[0]; // the top element remaining in the return stack
 }
```

#### 7.7.1.8 compute_product_void()

```
void compute_product_void (
            int N,
            int k,
            int s )
```

```
{
  //cerr << " void called N " << N << " k " << k << " s " << s << endl;
  if (k==1){
    helper_vars::return_stack.resize(1);
    helper_vars::return_stack[0] = N;//return N;
    return;
  }
  if (k == 0){ // TO CHECK because there are no terms to compute product
    helper_vars::return_stack.resize(1);
    helper_vars::return_stack[0] = 1;//return 1;
    return;
  }

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

  if (k == 2){
    helper_vars::mul_1 = N;
    helper_vars::mul_2 = N - s;
    helper_vars::return_stack.resize(1);
    helper_vars::return_stack[0] = helper_vars::mul_1 * helper_vars::mul_2;
    return;
  }

  int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
  int k_copy = k;
  while (k_copy > 0){
    k_bits ++;
    k_copy >>= 1;
  }
  k_bits += 2;
  vector<pair<int, int> > call_stack(2 * k_bits);
  //cout << " 2 * k_bits " << 2 * k_bits << endl;
  int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
  vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
  //vector<mpz_class> return_stack(2 * k_bits);
  helper_vars::return_stack.resize(2*k_bits);
  int return_pointer = 0;

  call_stack[call_pointer] = pair<int, int> (N, k);
  status_stack[call_pointer] = 0;
  call_pointer ++;

  int m;
  int N_now, k_now; // N and k for the current stack element

  while (call_pointer > 0){
    N_now = call_stack[call_pointer-1].first;
    k_now = call_stack[call_pointer-1].second;
    //cout << "call_pointer = " << call_pointer << " N = " << N_now << " k = " << k_now << " stat = " << status_stack[call_pointer-1] << endl;
    //cout << " the whole stack " << endl;
    //for (int i=0;i<call_pointer; i++){
    //  cout << call_stack[i].first << " , " << call_stack[i].second << " " << status_stack[i] << endl;
    //}
    if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
      // to collect two top elements in return stack and multiply them
      helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] * helper_vars::return_stack[return_pointer-1];
      return_pointer--; // remove two items, add one item
      call_pointer --;
    }else{
```

```
      //cout << " else " << endl;
      if(k_now == 1){
        // to return the corresponding N
        helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
        call_pointer --; // pop this element
      }
      if (k_now == 2){
        helper_vars::mul_1 = N_now;
        helper_vars::mul_2 = N_now - s;
        helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
        call_pointer --;
      }
      if (k_now > 2){
        m = k_now / 2;
        status_stack[call_pointer-1] = 1; // when return to this state, we know that we should aggregate
        call_stack[call_pointer] = pair<int, int>(N_now, m);
        status_stack[call_pointer] = 0; // just added
        call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
        status_stack[call_pointer+1]  = 0;
        call_pointer += 2;
      }
    }
  }
  // make sure there is exactly one element in return stack
  if (return_pointer != 1){
    cerr << " return pointer is not zero";
  }
  //return helper_vars::return_stack[0]; // the top element remaining in the return stack
}
```

#### 7.7.1.9 prod_factorial()

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

```
                                                            {
  if (i==j){
    return compute_product(a[i], a[i], 1);
  }else{
    helper_vars::mpz_vec2.resize(j-i+1);
    for (int k = i; k<=j;k++)
      helper_vars::mpz_vec2[k- i] = compute_product(a[k], a[k],1);
    compute_array_product(helper_vars::mpz_vec2);
    return helper_vars::mpz_vec2[0];
  }
}
```

#### 7.7.1.10 prod_factorial_old()

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

computes the product of factorials in a vector given a range

**Parameters**

| | |
|---|---|
| *a* | vector of integers |
| *i,j* | endpoints of the interval |

**Returns**

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

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

## 7.8 compression_helper.h File Reference

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

### Namespaces

- helper_vars

### Functions

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

  *This function computes the product of consecutive integers separated by a given iteration. This is the old version which uses standard recursion.*
- mpz_class compute_product_stack (int N, int k, int s)

  *This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.*
- mpz_class compute_product (int N, int k, int s)
- void compute_product_void (int N, int k, int s)
- void compute_array_product (vector< mpz_class > &a)

  *computes the product of elements in vector a by inline multiplication of adjacent elements recursively. The results will be in a[0].*
- mpz_class binomial (const int n, const int m)

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

  *computes the product of factorials in a vector given a range*
- mpz_class prod_factorial (const vector< int > &a, int i, int j)
- int bit_string_write (FILE ∗f, const string &s)

  *Write a string containing 0 and 1 to a binary file, treating the string as a bit sequence. Returns the number of bytes written to the output.*
- string bit_string_read (FILE ∗f)

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

## Variables

- mpz_class helper_vars::mul_1
- mpz_class helper_vars::mul_2

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

## 7.8.1 Function Documentation

### 7.8.1.1 binomial()

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

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

**Parameters**

| | |
|---|---|
| *n* | integer |
| *m* | integer |

**Returns**

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

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

### 7.8.1.2 bit_string_read()

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

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

**Parameters**

| | |
|---|---|
| *f* | a file pointer |

**Returns**

a string of zeros and ones.

```
                                    {
  int nu_bytes;
  int ssize;
  // read the number of bytes to read
  fread(&ssize, sizeof(ssize), 1, f);
  //cerr << " ssize " << ssize << endl;
  nu_bytes = ssize / 8;
  if (ssize % 8 != 0)
    nu_bytes ++;

  int last_byte_size = ssize % 8;
  if (last_byte_size == 0)
    last_byte_size  = 8;

  unsigned char c;
  bitset<8> B;
  string s;
  for (int i=0;i<nu_bytes;i++){
    fread(&c, sizeof(c), 1, f);
    B = c;
    //cout << B << endl;
    if (i < nu_bytes -1){
      s += B.to_string();
    }else{
      s += B.to_string().substr(8-last_byte_size, last_byte_size);
    }
  }
  return s;
}
```

#### 7.8.1.3 bit_string_write()

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

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

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

**Parameters**

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

**Returns**

the number of bytes written to the output

```
                                                   {
  // find out the number of bytes
  int ssize = s.size();
  int nu_bytes; // number of bytes wrote to the output

  //if (ssize % 8 != 0) // an incomplete byte is required
  //nu_bytes++;

  fwrite(&ssize, sizeof(ssize), 1, f); // first, write down how many bytes are coming.
```

```
  nu_bytes += sizeof(ssize);

  stringstream ss;
  ss << s;

  bitset<8> B;
  unsigned char c;
  while (ss >> B){
    c = B.to_ulong();
    fwrite(&c, sizeof(c), 1, f);
    nu_bytes += sizeof(c);
  }
  return nu_bytes;
}
```

#### 7.8.1.4 compute_array_product()

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

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

```
{
  //logger::item_start("Compute Array Product");
  int step_size, to_mul;
  int k = a.size();
  for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
    for (int i=0; i<k; i+=step_size){
      if (i+to_mul < k)
        a[i] *= a[i+to_mul];
    }
  }
  //logger::item_stop("Compute Array Product");
}
```

#### 7.8.1.5 compute_product()

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

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

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

  if (k == 2){
    helper_vars::mul_1 = N;
    helper_vars::mul_2 = N - s;
    return helper_vars::mul_1 * helper_vars::mul_2;
```

```
  }

  helper_vars::mpz_vec.resize(k);
  for (int i=0;i<k;i++)
    helper_vars::mpz_vec[i] = N - i * s;

  compute_array_product(helper_vars::mpz_vec);

  // int step_size, to_mul;


  // for (step_size = 2, to_mul = 1; to_mul < k; step_size <<=1, to_mul <<=1){
  //   for (int i=0; i<k; i+=step_size){
  //     if (i+to_mul < k)
  //       helper_vars::mpz_vec[i] *= helper_vars::mpz_vec[i+to_mul];
  //   }
  // }
  return helper_vars::mpz_vec[0];
}
```

#### 7.8.1.6 compute_product_old()

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

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

**Parameters**

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

**Returns**

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

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

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

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

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

```
  mpz_class ans = left*right;
  //logger::item_stop("cp_mul");
  return ans;
}
```

#### 7.8.1.7 compute_product_stack()

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

This function computes the product of consecutive integers separated by a given iteration. This is the new version which implements recursion via stack.

**Parameters**

| | |
|---|---|
| *N* | The first term in the product |
| *k* | the number of terms in the product |
| *s* | the iteration |

**Returns**

the product N $N \times (N-s) \times (N-2s) \times \ldots \times (N-(k-1)s)$

```
{

  if (k==1)
    return N;
  if (k == 0) // TO CHECK because there are no terms to compute product
    return 1;

  if (k < 0){
    cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s << endl;
    return 1;
  }
  if (N - (k-1) * s <= 0){ // the terms go negative
    //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
    return 0;
  }

  if (k == 2){
    helper_vars::mul_1 = N;
    helper_vars::mul_2 = N - s;
    return helper_vars::mul_1 * helper_vars::mul_2;
  }

  logger::item_start("CP body");

  int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
  int k_copy = k;
  while (k_copy > 0){
    k_bits ++;
    k_copy >>= 1;
  }
  k_bits += 2;
  vector<pair<int, int> > call_stack(2 * k_bits);
  //cout << " 2 * k_bits " << 2 * k_bits << endl;
  int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
  vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
  //vector<mpz_class> return_stack(2 * k_bits);
  helper_vars::return_stack.resize(2*k_bits);
  int return_pointer = 0;

  call_stack[call_pointer] = pair<int, int> (N, k);
```

```
    status_stack[call_pointer] = 0;
    call_pointer ++;

    int m;
    int N_now, k_now; // N and k for the current stack element

    while (call_pointer > 0){
      N_now = call_stack[call_pointer-1].first;
      k_now = call_stack[call_pointer-1].second;
      //cout << "call_pointer = " << call_pointer << " N = " << N_now << " k = " << k_now << " stat = " << status_stack[call_pointer-1] << endl;
      //cout << " the whole stack " << endl;
      //for (int i=0;i<call_pointer; i++){
      //  cout << call_stack[i].first << " , " << call_stack[i].second << " " << status_stack[i] << endl;
      //}
      if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
        // to collect two top elements in return stack and multiply them
        logger::item_start("CP arithmetic");
        helper_vars::return_stack[return_pointer-2] = helper_vars::return_stack[return_pointer-2] * helper_vars::return_stack[return_pointer-1];
        logger::item_stop("CP arithmetic");
        return_pointer--; // remove two items, add one item
        call_pointer --;
      }else{
        //cout << " else " << endl;
        if(k_now == 1){
          // to return the corresponding N
          helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].first;
          call_pointer --; // pop this element
        }
        if (k_now == 2){
          helper_vars::mul_1 = N_now;
          helper_vars::mul_2 = N_now - s;
          logger::item_start("CP arithmetic");
          helper_vars::return_stack[return_pointer++] = helper_vars::mul_1 * helper_vars::mul_2;
          logger::item_stop("CP arithmetic");
          call_pointer --;
        }
        if (k_now > 2){
          m = k_now / 2;
          status_stack[call_pointer-1] = 1; // when return to this state, we know that we should aggregate
          call_stack[call_pointer] = pair<int, int>(N_now, m);
          status_stack[call_pointer] = 0; // just added
          call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
          status_stack[call_pointer+1]  = 0;
          call_pointer += 2;
        }
      }
    }
    // make sure there is exactly one element in return stack
    if (return_pointer != 1){
      cerr << " return pointer is not zero";
    }
    logger::item_stop("CP body");
    return helper_vars::return_stack[0]; // the top element remaining in the return stack
}
```

#### 7.8.1.8 compute_product_void()

```
void compute_product_void (
            int N,
            int k,
            int s )
```

```
{
    //cerr << " void called N " << N << " k " << k << " s " << s << endl;
    if (k==1){
      helper_vars::return_stack.resize(1);
      helper_vars::return_stack[0] = N;//return N;
      return;
    }
```

```
    if (k == 0){ // TO CHECK because there are no terms to compute product
      helper_vars::return_stack.resize(1);
      helper_vars::return_stack[0] = 1;//return 1;
      return;
    }

    if (k < 0){
      cerr << " WARNING: compute_product called for k < 0, returning 1, N  " << N << " k " << k << " s " << s
     << endl;
      helper_vars::return_stack.resize(1);
      helper_vars::return_stack[0] = 1;//return 1;
      return;
    }
    if (N - (k-1) * s <= 0){ // the terms go negative
      //cerr << " WARNING: compute_product called for N - (k-1) * s <= 0 " << endl;
      helper_vars::return_stack.resize(1);
      helper_vars::return_stack[0] = 0; //return 0;
      return;
    }

    if (k == 2){
      helper_vars::mul_1 = N;
      helper_vars::mul_2 = N - s;
      helper_vars::return_stack.resize(1);
      helper_vars::return_stack[0] = helper_vars::mul_1 *
    helper_vars::mul_2;
      return;
    }

    int k_bits = 0; // roughly , the number of bits in k, the depth of the stack during run time
    int k_copy = k;
    while (k_copy > 0){
      k_bits ++;
      k_copy >>= 1;
    }
    k_bits += 2;
    vector<pair<int, int> > call_stack(2 * k_bits);
    //cout << " 2 * k_bits " << 2 * k_bits << endl;
    int call_pointer = 0; // size of the call pointer, so the top index is call_pointer - 1
    vector<int> status_stack(2 * k_bits); // 0: first meet, 1: to return
    //vector<mpz_class> return_stack(2 * k_bits);
    helper_vars::return_stack.resize(2*k_bits);
    int return_pointer = 0;

    call_stack[call_pointer] = pair<int, int> (N, k);
    status_stack[call_pointer] = 0;
    call_pointer ++;

    int m;
    int N_now, k_now; // N and k for the current stack element

    while (call_pointer > 0){
      N_now = call_stack[call_pointer-1].first;
      k_now = call_stack[call_pointer-1].second;
      //cout << "call_pointer = " << call_pointer << " N = " << N_now << " k = " << k_now << " stat = " <<
     status_stack[call_pointer-1] << endl;
      //cout << " the whole stack " << endl;
      //for (int i=0;i<call_pointer; i++){
      //  cout << call_stack[i].first << " , " << call_stack[i].second << " " << status_stack[i] << endl;
      //}
      if (status_stack[call_pointer-1] == 1){ // we should multiply two top elements in the return stack
        // to collect two top elements in return stack and multiply them
        helper_vars::return_stack[return_pointer-2] =
    helper_vars::return_stack[return_pointer-2] *
    helper_vars::return_stack[return_pointer-1];
        return_pointer--; // remove two items, add one item
        call_pointer --;
      }else{
        //cout << " else " << endl;
        if(k_now == 1){
          // to return the corresponding N
          helper_vars::return_stack[return_pointer++] = call_stack[call_pointer-1].
    first;
          call_pointer --; // pop this element
        }
        if (k_now == 2){
          helper_vars::mul_1 = N_now;
          helper_vars::mul_2 = N_now - s;
          helper_vars::return_stack[return_pointer++] =
    helper_vars::mul_1 * helper_vars::mul_2;
          call_pointer --;
        }
        if (k_now > 2){
          m = k_now / 2;
          status_stack[call_pointer-1] = 1; // when return to this state, we know that we should aggregate
          call_stack[call_pointer] = pair<int, int>(N_now, m);
          status_stack[call_pointer] = 0; // just added
```

```
          call_stack[call_pointer+1] = pair<int, int>(N_now - m*s, k_now - m );
          status_stack[call_pointer+1]  = 0;
          call_pointer += 2;
        }
      }
    }
    // make sure there is exactly one element in return stack
    if (return_pointer != 1){
      cerr << " return pointer is not zero";
    }
    //return helper_vars::return_stack[0]; // the top element remaining in the return stack
}
```

#### 7.8.1.9 prod_factorial()

```
mpz_class prod_factorial (
              const vector< int > & a,
              int i,
              int j )
```

```
                                                                            {
    if (i==j){
      return compute_product(a[i], a[i], 1);
    }else{
      helper_vars::mpz_vec2.resize(j-i+1);
      for (int k = i; k<=j;k++)
        helper_vars::mpz_vec2[k- i] = compute_product(a[k], a[k],1);
      compute_array_product(helper_vars::mpz_vec2);
      return helper_vars::mpz_vec2[0];
    }
}
```

#### 7.8.1.10 prod_factorial_old()

```
mpz_class prod_factorial_old (
              const vector< int > & a,
              int i,
              int j )
```

computes the product of factorials in a vector given a range

**Parameters**

| | |
|---|---|
| *a* | vector of integers |
| *i,j* | endpoints of the interval |

**Returns**

$\prod_{v=i}^{j} a_v!$

```
{
    if (i == j){
      return compute_product(a[i], a[i], 1);
    }else{
```

```
    int k = (i+j)/2;
    mpz_class x = prod_factorial(a, i, k);
    mpz_class y = prod_factorial(a, k+1, j);
    return x * y;
  }
}
```

## 7.9 fenwick.cpp File Reference

```
#include "fenwick.h"
```

## 7.10 fenwick.h File Reference

```
#include <vector>
```

### Classes

- class fenwick_tree

  *Fenwick tree class.*
- class reverse_fenwick_tree

  *similar to the fenwick_tree class, but instead of prefix sums, this class computes suffix sums.*

## 7.11 gcomp.cpp File Reference

To compress / decompress simple marked graphs.

```
#include <iostream>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>
#include "marked_graph_compression.h"
```

### Functions

- int main (int argc, char **argv)

### 7.11.1 Detailed Description

To compress / decompress simple marked graphs.

This code has two functionalities: 1) gets a simple marked graph and compresses it, 2) gets a simple marked graph in its compressed form and decompresses it.

In order to compress a graph, the hyperparameters h and delta should be given using -h and -d, respectively. The input graph should be given using -i option, followed by the name of the file containing the graph. Also, the compressed graph will be stored in the file specified by -o option. A graph must be specified using its edge list in the following format: first, the number of vertices comes, then the mark of vertices in order, then each line contains the information on an edge, which is of the form i j x y, meaning there is an edge between vertices i and j, with mark x and y towards i and j, respectively.

In order to decompress, the compressed file should be given after -i, the file to store the decompressed graph should be given using -o, and an argument -u for uncompress should be given.

### 7.11.2 Function Documentation

#### 7.11.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
  int h, delta;
  string infile, outfile;
  bool uncompress = false; // becomes true if -u option is given (to decompress)
  bool quiet = true; // becomes false if -v option is given (verbose)
  bool stat = false; // if true, statistics on the properties of the compressed graph, e.g. number of star
    vertices / edges or the number of partition graphs will be given
  char opt;

  string report_file, stat_file;
  ofstream report_stream, stat_stream;

  while ((opt = getopt(argc, argv, "h:d:i:o:uvsV:S:")) != EOF){
    switch(opt){
    case 'h':
      h = atoi(optarg);
      break;
    case 'd':
      delta = atoi(optarg);
      break;
    case 'i':
      infile = string(optarg);
      break;
    case 'o':
      outfile = string(optarg);
      break;
    case 'u':
      uncompress = true; // in the decompression phase
      break;
    case 'v':
      quiet = false;
      break;
    case 'V':
      report_file = string(optarg);
      if (report_file != ""){
        report_stream.open(report_file);
        logger::report_stream = &report_stream;
      }
      break;
    case 's':
```

```
      stat = true;
      break;
    case 'S':
      stat_file = string(optarg);
      if (stat_file != ""){
        stat_stream.open(stat_file);
        logger::stat_stream = &stat_stream;
      }
      break;
    case '?':
      cerr << "Error: option -" << char(optopt) << " requires an argument" << endl;
      return 1;
    }
  }
  if (uncompress == false and h <= 0){
    cerr << "Error: parameter h must be a positive integer. Instead, the value " << h << " was given." << endl;
    return 1;
  }
  if (uncompress == false and delta <= 0){
    cerr << "Error: parameter d (delta) must be a positive integer. Instead, the value " << delta << " was given." << endl;
    return 1;
  }

  ifstream inp(infile.c_str());
  ofstream oup(outfile.c_str());

  if (!inp.good()){
    cerr << "Error: invalid input file <" << infile << "> given " << endl;
    return 1;
  }

  if (!oup.good()){
    cerr << "Error: invalid output file <" << outfile << "> given " << endl;
    return 1;
  }


  if (quiet == true){
    // do not log
    logger::verbose = false; // no run time log
    logger::report = false; // no final report
  }

  if (stat == true){
    logger::stat = true;
  }

  //cout << " h = " << h << " delta = " << delta << " infile = " << infile << " outfile = " << outfile << endl;

  logger::start();
  if (uncompress == false){
    // goal is compression
    logger::current_depth++;
    logger::add_entry("Read Graph", "");
    marked_graph_encoder E(h, delta);
    marked_graph G; // the input graph to be compressed
    inp >> G;
    logger::current_depth--;
    logger::current_depth++;
    logger::add_entry("Encode", "");
    marked_graph_compressed C = E.encode(G);
    logger::current_depth--;
    //FILE* f;
    //f = fopen(outfile.c_str(), "wb+");
    logger::current_depth++;
    logger::add_entry("Write to binary", "");
    //C.binary_write(f);
    C.binary_write(outfile);
    //fclose(f);
    logger::current_depth--;
  }else{
    // goal is to decompess
    //FILE* f;
    //f = fopen(infile.c_str(), "rb+");
    marked_graph_compressed C;
    logger::current_depth++;
    logger::add_entry("Read from binary", "");
    //C.binary_read(f);
    C.binary_read(infile);
    //fclose(f);
    logger::current_depth--;

    logger::current_depth++;
    logger::add_entry("Decode", "");
```

```
    marked_graph_decoder D;
    marked_graph G = D.decode(C);
    logger::current_depth--;

    logger::current_depth++;
    logger::add_entry("Write decoded graph to output file","");
    oup << G;
    logger::current_depth--;
  }
  logger::stop();
  return 0;
}
```

## 7.12 graph_message.cpp File Reference

```
#include "graph_message.h"
```

### Functions

- bool pair_compare (const pair< vector< int >, int > &a, const pair< vector< int >, int > &b)

    *used for sorting messages*

### 7.12.1 Function Documentation

#### 7.12.1.1 pair_compare()

```
bool pair_compare (
            const pair< vector< int >, int > & a,
            const pair< vector< int >, int > & b )
```

used for sorting messages

```
                                                                        {
  return a.first < b.first;
}
```

## 7.13 graph_message.h File Reference

```
#include <vector>
#include <map>
#include <unordered_map>
#include <boost/functional/hash/hash.hpp>
#include "marked_graph.h"
#include "logger.h"
```

### Classes

- struct vint_hash
- class graph_message

  *this class takes care of message passing on marked graphs.*
- class colored_graph

  *this class defines a colored graph, which is obtained from a simple marked graph and the color of edges come from the type of edges*

### Functions

- bool pair_compare (const pair< vector< int >, int > &, const pair< vector< int >, int > &)

  *used for sorting messages*

## 7.13.1 Function Documentation

#### 7.13.1.1 pair_compare()

```
bool pair_compare (
            const pair< vector< int >, int > & ,
            const pair< vector< int >, int > &  )
```

used for sorting messages

```
{
  return a.first < b.first;
}
```

## 7.14 logger.cpp File Reference

```
#include "logger.h"
```

## 7.15 logger.h File Reference

```
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
#include <ctime>
#include <map>
```

## Classes

- class log_entry
- class logger

## Functions

- void __attribute__ ((constructor)) prog_start()
- void __attribute__ ((destructor)) prog_finish()

### 7.15.1 Function Documentation

#### 7.15.1.1 __attribute__() [1/2]

```
void __attribute__ (
            (constructor)  )
```

#### 7.15.1.2 __attribute__() [2/2]

```
void __attribute__ (
            (destructor)  )
```

## 7.16 marked_graph.cpp File Reference

```
#include "marked_graph.h"
```

## Functions

- istream & operator>> (istream &inp, marked_graph &G)

    *inputs a marked_graph*
- bool operator== (const marked_graph &G1, const marked_graph &G2)
- bool operator!= (const marked_graph &G1, const marked_graph &G2)
- bool edge_compare (const pair< int, pair< int, int > > &a, pair< int, pair< int, int > > &b)

    *this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.*
- ostream & operator<< (ostream &o, const marked_graph &G)

### 7.16.1 Function Documentation

#### 7.16.1.1 edge_compare()

```
bool edge_compare (
            const pair< int, pair< int, int > > & a,
            pair< int, pair< int, int > > & b )
```

this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.

```
{
  return a.first < b.first;
}
```

#### 7.16.1.2 operator"!=()

```
bool operator!= (
            const marked_graph & G1,
            const marked_graph & G2 )
```

```
{
  return !(G1 == G2);
}
```

#### 7.16.1.3 operator<<()

```
ostream& operator<< (
            ostream & o,
            const marked_graph & G )
```

```
{
  o << G.nu_vertices << endl;
  for (int v=0;v<G.nu_vertices;v++){
    o << G.ver_mark[v];
    if (v < G.nu_vertices-1)
      o << " ";
  }
  o << endl;

  vector<pair<pair<int, int>, pair<int, int> > > edges;
  pair<pair<int, int> , pair<int, int> > edge; // the current edge to be added to the list
  for (int v=0;v<G.nu_vertices;v++){
    for (int i=0;i<G.adj_list[v].size();i++){
      if (G.adj_list[v][i].first > v){ // avoid duplicate in edge list, only add edges where the other endpoint has a greater index
        edge.first.first = v;
        edge.first.second = G.adj_list[v][i].first;
        edge.second = G.adj_list[v][i].second;
        edges.push_back(edge);
      }
    }
  }
  sort(edges.begin(), edges.end());
  o << edges.size() << endl;
  for(int i=0;i<edges.size();i++){
    o << edges[i].first.first << " " << edges[i].first.second << " " << edges[i].second.first << " " << edges[i].second.second << endl;
  }
  return o;
  /*
```

```
  o << " number of vertices " << G.nu_vertices << endl;
  vector<pair<int, pair<int, int> > > l; // the adjacency list of a vertex
  for (int v=0; v<G.nu_vertices; v++){
    o << " vertex " << v << " mark " << G.ver_mark[v] << endl;
    //o << " adj list (connections to vertices with greater index): format (j, (x,y))" << endl;
    o << " adj list " << endl;
    l = G.adj_list[v];
    sort(l.begin(), l.end(), edge_compare);
    for (int i=0;i<l.size();i++){
      if (l[i].first > v)
        o << " (" << l[i].first << ", (" << l[i].second.first << ", " << l[i].second.second << ")) ";
    }
    o << endl << endl;
  }
  return o;
  */
}
```

#### 7.16.1.4 operator==()

```
bool operator== (
            const marked_graph & G1,
            const marked_graph & G2 )
```

two marked graphs are said to be the same if: 1) they have the same number of vertices, 2) vertex marks match and 3) each vertex has the same set of neighbors with matching marks.

```
{
  if (G1.nu_vertices != G2.nu_vertices)
    return false;
  return G1.adj_list == G2.adj_list;
  int n = G1.nu_vertices; // number of vertices of the two graphs
  vector< pair< int, pair< int, int > > > l1, l2; // the adjacency list of a vertex in two graphs for comparison.
  for (int v=0;v<n;v++){
    if (G1.ver_mark[v] != G2.ver_mark[v]) // mark of each vertex should be the same
      return false;
    if (G1.adj_list[v].size() != G2.adj_list[v].size()) // each vertex must have the same degree in two graphs
      return false;
    l1 = G1.adj_list[v];
    l2 = G2.adj_list[v];
    sort(l1.begin(), l1.end(), edge_compare); // sort with respect to the other endpoint
    sort(l2.begin(), l2.end(), edge_compare);
    if (l1 != l2) // after sorting, the lists must match
      return false;
  }
  return true;
}
```

#### 7.16.1.5 operator>>()

```
istream& operator>> (
            istream & inp,
            marked_graph & G )
```

inputs a marked_graph

The input format is as follows: 1) number of vertices 2) a list of vertex marks as nonnegative integers 3) number of edges 4) for each edge: write $ijxy$, where $i$ and $j$ are the endpoints (here, $0 \leq i, j \leq n-1$ with $n$ being the number of vertices), $x$ is the mark towards $i$ and $y$ is the mark towards $j$ (both nonnegative integers) Example: 2 1 2 1 0 1 1 2 which is a graph with 2 vertices, the mark of vertex 0 is 1 and the mark of vertex 1 is 2, there is one edge between these two vertices with mark 1 towards 0 and mark 2 toward s 1

```
{
  logger::current_depth++;
  logger::add_entry("Read vertex marks and edges","");
  int nu_vertices;
  inp >> nu_vertices;

  vector<int> ver_marks;
  ver_marks.resize(nu_vertices);
  for (int i=0;i<nu_vertices;i++)
    inp >> ver_marks[i];

  int nu_edges;
  inp >> nu_edges;
  vector<pair< pair<int, int> , pair<int, int> > > edges;
  edges.resize(nu_edges);
  for (int i=0;i<nu_edges;i++)
    inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >> edges[i].second.second
   ;

  logger::add_entry("Constructing marked graph", "");
  G = marked_graph(nu_vertices, edges, ver_marks);
  logger::current_depth--;
  return inp;
}
```

## 7.17 marked_graph.h File Reference

```
#include <iostream>
#include <vector>
#include <map>
#include <fstream>
#include "logger.h"
```

### Classes

- class marked_graph

  *simple marked graph*

### Functions

- istream & operator>> (istream &inp, marked_graph &G)

  *inputs a marked_graph*
- bool edge_compare (const pair< int, pair< int, int > > &a, pair< int, pair< int, int > > &b)

  *this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.*

### 7.17.1 Function Documentation

#### 7.17.1.1 edge_compare()

```
bool edge_compare (
            const pair< int, pair< int, int > > & a,
            pair< int, pair< int, int > > & b )
```

this is to help comparing two marked graphs. The inputs would resemble two edge information, of the form (j, (x, y)), where j is the other endpoint, and x and y are marks. We want to sort them with respect to the neighbor index j.

```
{
  return a.first < b.first;
}
```

#### 7.17.1.2 operator>>()

```
istream& operator>> (
            istream & inp,
            marked_graph & G )
```

inputs a marked_graph

The input format is as follows: 1) number of vertices 2) a list of vertex marks as nonnegative integers 3) number of edges 4) for each edge: write $ijxy$, where $i$ and $j$ are the endpoints (here, $0 \leq i, j \leq n-1$ with $n$ being the number of vertices), $x$ is the mark towards $i$ and $y$ is the mark towards $j$ (both nonnegative integers) Example: 2 1 2 1 0 1 1 2 which is a graph with 2 vertices, the mark of vertex 0 is 1 and the mark of vertex 1 is 2, there is one edge between these two vertices with mark 1 towards 0 and mark 2 toward s 1

```
{
  logger::current_depth++;
  logger::add_entry("Read vertex marks and edges","");
  int nu_vertices;
  inp >> nu_vertices;

  vector<int> ver_marks;
  ver_marks.resize(nu_vertices);
  for (int i=0;i<nu_vertices;i++)
    inp >> ver_marks[i];

  int nu_edges;
  inp >> nu_edges;
  vector<pair< pair<int, int> , pair<int, int> > > edges;
  edges.resize(nu_edges);
  for (int i=0;i<nu_edges;i++)
    inp >> edges[i].first.first >> edges[i].first.second >> edges[i].second.first >> edges[i].second.second
   ;

  logger::add_entry("Constructing marked graph", "");
  G = marked_graph(nu_vertices, edges, ver_marks);
  logger::current_depth--;
  return inp;
}
```

## 7.18 marked_graph_compression.cpp File Reference

```
#include "marked_graph_compression.h"
```

## 7.19 marked_graph_compression.h File Reference

```
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
#include "simple_graph.h"
#include "bipartite_graph.h"
#include "simple_graph_compression.h"
#include "bipartite_graph_compression.h"
#include "time_series_compression.h"
#include "logger.h"
#include "bitstream.h"
```

### Classes

- class marked_graph_compressed
- class marked_graph_encoder
- class marked_graph_decoder

## 7.20 random_graph.cpp File Reference

```
#include "random_graph.h"
```

### Functions

- marked_graph marked_ER (int n, double p, int ver_mark, int edge_mark)

    *generates a marked Erdos Renyi graph*
- marked_graph poisson_graph (int n, double deg_mean, int ver_mark, int edge_mark)

    *generates a random graph where roughly speaking, the degree of a vertex is Poisson*
- marked_graph near_regular_graph (int n, int half_deg, int ver_mark, int edge_mark)

    *generates a random graph which is nearly regular, and the degree of each vertex is close to 2 $*$ half_deg. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \mathtt{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \mathtt{edge\_mark} - 1\}$.*

### 7.20.1 Function Documentation

#### 7.20.1.1 marked_ER()

```
marked_graph marked_ER (
        int n,
        double p,
        int ver_mark,
        int edge_mark )
```

generates a marked Erdos Renyi graph

**Parameters**

| | |
|---|---|
| *n* | the number of vertices |
| *p* | probability of an edge being present |
| *ver_mark* | the number of possible vertex marks |
| *edge_mark* | the number of possible edge marks |

**Returns**

a random marked graph, where each edge is independently present with probability p, each vertex has a random integer mark in the range [0,ver_mark), and each edge has two random integers marks in the range [0,edge_mark)

```
{
  unsigned seed = chrono::system_clock::now().time_since_epoch().count();
  default_random_engine generator(seed);
  uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
  uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
  uniform_real_distribution<double> unif_dist(0.0,1.0);
  double unif;
  int x, xp; // generated marks

  vector<int> ver_marks(n); // vector of size n
  vector<pair< pair<int, int>, pair<int, int> > > edges; // the edge list of the graph

  for (int v=0; v<n; v++){
    ver_marks[v] = ver_mark_dist(generator);
    for (int w=v+1; w<n;w++){
      unif = unif_dist(generator);
      if (unif < p){ // we put an edge between v and w
        x = edge_mark_dist(generator);
        xp = edge_mark_dist(generator);
        edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v,w), pair<int, int>(x,xp)));
      }
    }
  }
  cout << " marked_ER number of edges " << edges.size() << endl;
  return marked_graph(n, edges, ver_marks);
}
```

#### 7.20.1.2 near_regular_graph()

```
marked_graph near_regular_graph (
              int n,
              int half_deg,
              int ver_mark,
              int edge_mark )
```

generates a random graph which is nearly regular, and the degree of each vertex is close to $2 * \text{half\_deg}$. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \mathtt{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \mathtt{edge\_mark} - 1\}$.

**Parameters**

| | |
|---|---|
| *n* | the number of vertices |
| *half_deg* | the number of edges each vertex tries to connect to, so the final average degree is 2∗half_deg |
| *ver_mark* | the number of possible vertex marks |
| *edge_mark* | the number of possible edge_marks |

**Returns**

a random marked graph as described above.

```
                                                                                {
unsigned seed = chrono::system_clock::now().time_since_epoch().count();
default_random_engine generator(seed);

uniform_int_distribution<int> neighbor_dist(0,n-1);
uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark

int w;
pair<int, int> edge;

set<pair<int, int> > umarked_edges;
vector< pair< pair< int, int >, pair< int, int > > > edges;

int x, xp; // edge marks
vector<int> ver_marks(n);

for (int i=0;i<n;i++){
  ver_marks[i] = ver_mark_dist(generator);
  for (int j=0;j<half_deg;j++){
    w = neighbor_dist(generator);
    if (w!= i){
      edge = pair<int,int>(i,w);
      if (edge.first > edge.second)
         swap (edge.first, edge.second); // to make sure pairs are ordered
       umarked_edges.insert(edge);
     }
   }
 }
 for (set<pair<int, int>>::iterator it = umarked_edges.begin(); it!=umarked_edges.end(); it++){
   x = edge_mark_dist(generator);
   xp = edge_mark_dist(generator);
   edges.push_back(pair<pair<int, int>, pair<int, int> >(*it, pair<int, int>(x, xp)));
 }


 return marked_graph(n, edges, ver_marks);
}
```

#### 7.20.1.3 poisson_graph()

```
marked_graph poisson_graph (
               int n,
               double deg_mean,
               int ver_mark,
               int edge_mark )
```

generates a random graph where roughly speaking, the degree of a vertex is Poisson

**Parameters**

| | |
|---|---|
| *n* | the number of vertices |
| *deg_mean* | mean of Poisson |
| *ver_mark* | the number of possible vertex marks |
| *edge_mark* | the number of possible edge marks |

**Returns**

A random graph, where each vertex chooses its degree according to Poisson(deg_mean), then picks neighbors uniformly at random, and connects to them (if the neighbors have not already connected to them, if some of the neighbors I pick are already connected to me, I just don't do anything). Vertex and edge marks are picked independently and uniformly.

```
                                                                                   {
  unsigned seed = chrono::system_clock::now().time_since_epoch().count();
  default_random_engine generator(seed);
  poisson_distribution<int> deg_dist(deg_mean);
  uniform_int_distribution<int> neighbor_dist(0,n-1); // distribution for the other endpoint

  vector< pair< pair< int, int >, pair< int, int > > > edges;
  vector<int> ver_marks(n);

  uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
  uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
  int x, xp; // edge mark values

  pair< pair< int, int >, pair< int, int > > edge; // the current edge to be added
  vector<set<int> > neighbors(n); // the list of neighbors of vertices
  int deg; // the degree of a vertex
  int w; // the neighbor
  for (int v=0;v<n;v++){
    ver_marks[v] = ver_mark_dist(generator);
    deg = deg_dist(generator);
    for (int i=0; i<deg; i++){
      do{
        w = neighbor_dist(generator);
      }while(w == v or neighbors[v].find(w) != neighbors[v].end()); // not myself and not already connected
      // now, w is a possible neighbor
      // see if w has picked v as a neighbor
      if (neighbors[w].find(v) == neighbors[w].end()){
        // add w as a neighbors to v
        neighbors[v].insert(w);
        // marks for the edge
        x = edge_mark_dist(generator);
        xp = edge_mark_dist(generator);
        edge.first.first = v;
        edge.first.second = w;
        edge.second.first = x;
        edge.second.second = xp;
        edges.push_back(edge);
      }
    }
  }
  cerr << " edges size " << edges.size() << endl;
  return marked_graph(n, edges, ver_marks);
}
```

## 7.21 random_graph.h File Reference

```
#include "marked_graph.h"
#include <random>
#include <chrono>
#include <vector>
#include <set>
```

### Functions

- marked_graph marked_ER (int n, double p, int ver_mark, int edge_mark)

  *generates a marked Erdos Renyi graph*
- marked_graph poisson_graph (int n, double deg_mean, int ver_mark, int edge_mark)

  *generates a random graph where roughly speaking, the degree of a vertex is Poisson*
- marked_graph near_regular_graph (int n, int half_deg, int ver_mark, int edge_mark)

  *generates a random graph which is nearly regular, and the degree of each vertex is close to 2 $*$ half_deg. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \texttt{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \texttt{edge\_mark} - 1\}$.*

### 7.21.1 Function Documentation

#### 7.21.1.1 marked_ER()

```
marked_graph marked_ER (
             int n,
             double p,
             int ver_mark,
             int edge_mark )
```

generates a marked Erdos Renyi graph

**Parameters**

| | |
|---|---|
| *n* | the number of vertices |
| *p* | probability of an edge being present |
| *ver_mark* | the number of possible vertex marks |
| *edge_mark* | the number of possible edge marks |

**Returns**

a random marked graph, where each edge is independently present with probability p, each vertex has a random integer mark in the range [0,ver_mark), and each edge has two random integers marks in the range [0,edge_mark)

```
{
  unsigned seed = chrono::system_clock::now().time_since_epoch().count();
  default_random_engine generator(seed);
  uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
  uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
  uniform_real_distribution<double> unif_dist(0.0,1.0);
   double unif;
   int x, xp; // generated marks

   vector<int> ver_marks(n); // vector of size n
   vector<pair< pair<int, int>, pair<int, int> > > edges; // the edge list of the graph

   for (int v=0; v<n; v++){
     ver_marks[v] = ver_mark_dist(generator);
     for (int w=v+1; w<n;w++){
       unif = unif_dist(generator);
       if (unif < p){ // we put an edge between v and w
         x = edge_mark_dist(generator);
         xp = edge_mark_dist(generator);
         edges.push_back(pair<pair<int, int>, pair<int, int> >(pair<int, int>(v,w), pair<int, int>(x,xp)));
       }
     }
   }
   cout << " marked_ER number of edges " << edges.size() << endl;
   return marked_graph(n, edges, ver_marks);
 }
```

#### 7.21.1.2 near_regular_graph()

```
marked_graph near_regular_graph (
             int n,
```

```
            int half_deg,
            int ver_mark,
            int edge_mark )
```

generates a random graph which is nearly regular, and the degree of each vertex is close to 2 $*$ half_deg. Each vertex is uniformly connected to half_deg many other vertices, and multiple edges are dropped. Furthermore, each vertex and edge is randomly assigned marks, where the vertex mark set is $\{0, 1, \ldots, \mathtt{ver\_mark} - 1\}$ and $\{0, 1, \ldots, \mathtt{edge\_mark} - 1\}$.

**Parameters**

| | |
|---|---|
| *n* | the number of vertices |
| *half_deg* | the number of edges each vertex tries to connect to, so the final average degree is 2$*$half_deg |
| *ver_mark* | the number of possible vertex marks |
| *edge_mark* | the number of possible edge_marks |

**Returns**

a random marked graph as described above.

```
                                                                             {
  unsigned seed = chrono::system_clock::now().time_since_epoch().count();
  default_random_engine generator(seed);

  uniform_int_distribution<int> neighbor_dist(0,n-1);
  uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
  uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark

  int w;
  pair<int, int> edge;

  set<pair<int, int> > umarked_edges;
  vector< pair< pair< int, int >, pair< int, int > > > edges;

  int x, xp; // edge marks
  vector<int> ver_marks(n);

  for (int i=0;i<n;i++){
    ver_marks[i] = ver_mark_dist(generator);
    for (int j=0;j<half_deg;j++){
      w = neighbor_dist(generator);
      if (w!= i){
        edge = pair<int,int>(i,w);
        if (edge.first > edge.second)
           swap (edge.first, edge.second); // to make sure pairs are ordered
         umarked_edges.insert(edge);
      }
    }
  }
   for (set<pair<int, int>>::iterator it = umarked_edges.begin(); it!=umarked_edges.end(); it++){
     x = edge_mark_dist(generator);
     xp = edge_mark_dist(generator);
     edges.push_back(pair<pair<int, int>, pair<int, int> >(*it, pair<int, int>(x, xp)));
   }


   return marked_graph(n, edges, ver_marks);
}
```

#### 7.21.1.3 poisson_graph()

```
marked_graph poisson_graph (
            int n,
            double deg_mean,
            int ver_mark,
            int edge_mark )
```

generates a random graph where roughly speaking, the degree of a vertex is Poisson

#### Parameters

| | |
|---|---|
| *n* | the number of vertices |
| *deg_mean* | mean of Poisson |
| *ver_mark* | the number of possible vertex marks |
| *edge_mark* | the number of possible edge marks |

#### Returns

A random graph, where each vertex chooses its degree according to Poisson(deg_mean), then picks neighbors uniformly at random, and connects to them (if the neighbors have not already connected to them, if some of the neighbors I pick are already connected to me, I just don't do anything). Vertex and edge marks are picked independently and uniformly.

```
                                                                                   {
  unsigned seed = chrono::system_clock::now().time_since_epoch().count();
  default_random_engine generator(seed);
  poisson_distribution<int> deg_dist(deg_mean);
  uniform_int_distribution<int> neighbor_dist(0,n-1); // distribution for the other endpoint

  vector< pair< pair< int, int >, pair< int, int > > > edges;
  vector<int> ver_marks(n);

  uniform_int_distribution<int> ver_mark_dist(0,ver_mark-1); // distribution a vertex mark
  uniform_int_distribution<int> edge_mark_dist(0,edge_mark-1); // distribution an edge mark
  int x, xp; // edge mark values

  pair< pair< int, int >, pair< int, int > > edge; // the current edge to be added
  vector<set<int> > neighbors(n); // the list of neighbors of vertices
  int deg; // the degree of a vertex
  int w; // the neighbor
  for (int v=0;v<n;v++){
    ver_marks[v] = ver_mark_dist(generator);
    deg = deg_dist(generator);
    for (int i=0; i<deg; i++){
      do{
        w = neighbor_dist(generator);
      }while(w == v or neighbors[v].find(w) != neighbors[v].end()); // not myself and not already connected
      // now, w is a possible neighbor
      // see if w has picked v as a neighbor
      if (neighbors[w].find(v) == neighbors[w].end()){
        // add w as a neighbors to v
        neighbors[v].insert(w);
        // marks for the edge
        x = edge_mark_dist(generator);
        xp = edge_mark_dist(generator);
        edge.first.first = v;
        edge.first.second = w;
        edge.second.first = x;
        edge.second.second = xp;
        edges.push_back(edge);
      }
    }
  }
  cerr << " edges size " << edges.size() << endl;
  return marked_graph(n, edges, ver_marks);
}
```

## 7.22 README.md File Reference

## 7.23 rnd_graph.cpp File Reference

```
#include <iostream>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>
#include "random_graph.h"
```

## Functions

- void print_usage ()
- int main (int argc, char **argv)

### 7.23.1 Function Documentation

#### 7.23.1.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
  int n, edge_mark, ver_mark;
  double p, deg;
  string type, outfile;
  char opt;

  while ((opt = getopt(argc, argv, "t:n:p:d:e:v:o:h")) != EOF){
    switch(opt){
    case 't':
      type = string(optarg);
      break;
    case 'n':
      n = atoi(optarg);
      break;
    case 'p':
      p = atof(optarg);
      break;
    case 'd':
      deg = atof(optarg);
      break;
    case 'e':
      edge_mark = atoi(optarg);
      break;
    case 'v':
      ver_mark = atoi(optarg);
      break;
    case 'o':
      outfile = string(optarg);
      break;
    case 'h':
      print_usage();
      break;
    case '?':
      cerr << "Error: option -" << char(optopt) << " requires an argument" << endl;
      print_usage();
      return 1;
    }
  }
  ofstream oup(outfile.c_str());
  if (!oup.good()){
    cerr << "Error: invalid output file <" << outfile << "> given " << endl;
    return 1;
  }

  marked_graph G;
  if (type == "er"){
    G = marked_ER(n, p, ver_mark, edge_mark);
    oup << G;
  }else if(type == "reg"){
    G = near_regular_graph(n, deg, ver_mark, edge_mark);
    oup << G;
  }else if(type == "poi"){
    G = poisson_graph(n, deg, ver_mark, edge_mark);
    oup << G;
  }else{
    cerr << " ERROR: unknown type " << type << ", type must be either <er> or <reg> or <poi> " << endl;
    return 1;
  }
  return 0;
}
```

#### 7.23.1.2 print_usage()

```
void print_usage ( )
```

```
                     {
  cout << " Usage: " << endl;
  cout << " rnd_graph -t [random graph type] -n [number of vertices] [[options with -p / -d]] -e [number of
    edge marks] -v [number of vertex marks] -o [output file]" << endl;
  cout << endl;
  cout << " OPTIONS " << endl;
  cout << " -n : number of vertices (positive integer) " << endl;
  cout << " -e : size of edge mark set " << endl;
  cout << " -v : size of vertex mark set " << endl;
  cout << " -o : output file to store graph " << endl;
  cout << " -t : type of random graph: " << endl;
  cout << "            \"er\" for Erdos Renyi " << endl;
  cout << "            \"reg\" for near regular graph " << endl;
  cout << "            \"poi\" for Poisson graph " << endl;
  cout << endl;
  cout << " TYPE SPECIFIC OPTIONS " << endl;
  cout << endl;
  cout << " Erdos Renyi Graph " << endl;
  cout << " ----------------- " << endl;
  cout << " -p : edge probability " << endl;
  cout << endl;
  cout << " Near Regular Graph " << endl;
  cout << " ------------------ " << endl;
  cout << " -d : half degree, generates a random graph which is nearly regular, and the degree of each
    vertex is close to 2 * half_deg. " << endl;
  cout << endl;
  cout << " Poisson Graph " << endl;
  cout << " ------------- " << endl;
  cout << " -d : mean degree, each vertex chooses Poisson neighbors with this mean " << endl;
  cout << endl;

}
```

## 7.24 simple_graph.cpp File Reference

```
#include "simple_graph.h"
```

### Functions

- ostream & operator<< (ostream &o, const graph &G)
- bool operator== (const graph &G1, const graph &G2)
- bool operator!= (const graph &G1, const graph &G2)

### 7.24.1 Function Documentation

#### 7.24.1.1 operator"!=()

```
bool operator!= (
            const graph & G1,
            const graph & G2 )
```

```
{
  return !(G1 == G2);
}
```

#### 7.24.1.2 operator<<()

```
ostream& operator<< (
        ostream & o,
        const graph & G )
```

```
{
  int n = G.nu_vertices();
  vector<int> list;
  for (int i=0;i<n;i++){
    list = G.get_forward_list(i);
    o << i << " -> ";
    for (int j=0;j<list.size();j++){
      o << list[j];
      if (j < list.size()-1)
        o << ", ";
    }
    o << endl;
  }
  return o;
}
```

#### 7.24.1.3 operator==()

```
bool operator== (
        const graph & G1,
        const graph & G2 )
```

```
{
  int n1 = G1.nu_vertices();
  int n2 = G2.nu_vertices();
  if (n1!= n2)
    return false;
  vector<int> list1, list2;
  for (int v=0; v<n1; v++){
    list1 = G1.get_forward_list(v);
    list2 = G2.get_forward_list(v);
    if (list1 != list2)
      return false;
  }
  return true;
}
```

## 7.25 simple_graph.h File Reference

```
#include <iostream>
#include <vector>
```

### Classes

- class graph

  *simple unmarked graph*

## 7.26 simple_graph_compression.cpp File Reference

```
#include "simple_graph_compression.h"
```

## 7.27 simple_graph_compression.h File Reference

```
#include <vector>
#include <math.h>
#include "simple_graph.h"
#include "compression_helper.h"
#include "fenwick.h"
#include "logger.h"
```

### Classes

- class graph_encoder

    *Encodes a simple unmarked graph.*
- class graph_decoder

    *Decodes a simple unmarked graph.*

## 7.28 test.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
#include "fenwick.h"
#include "simple_graph.h"
#include "simple_graph_compression.h"
#include "bipartite_graph.h"
#include "bipartite_graph_compression.h"
#include "time_series_compression.h"
#include "marked_graph_compression.h"
#include "random_graph.h"
#include "logger.h"
```

### Functions

- ostream & operator<< (ostream &o, const vector< int > &v)
- void b_graph_test ()
- void graph_test ()
- void time_series_compression_test ()
- void marked_graph_encoder_test ()
- void random_graph_test ()
- int main ()

## 7.28.1 Function Documentation

### 7.28.1.1 b_graph_test()

void b_graph_test ( )

```
{
  vector<int> a = {1,1,2}; // left degree sequence
  vector<int> b = {2,2}; // right degree sequence

  b_graph G({{0},{1},{0,1}}); // defining the graph

  b_graph_encoder E(a,b); // constructing the encoder object
  mpz_class f = E.encode(G);

  b_graph_decoder D(a, b);
  b_graph Ghat = D.decode(f);

  if (Ghat == G)
    cout << " successfully decoded the graph! " << endl;
}
```

### 7.28.1.2 graph_test()

void graph_test ( )

```
{
  vector<int> a = {3,2,2,3};
  vector<vector<int> > list = {{1,2,3},{3},{3},{}};

  graph G(list);

  graph_encoder E(a);
  pair<mpz_class, vector<int> > f = E.encode(G);

  graph_decoder D(a);
  graph Ghat = D.decode(f.first, f.second);
  Ghat = D.decode(f.first, f.second);

  if (Ghat == G)
    cout << " successfully decoded the graph! " << endl;
}
```

### 7.28.1.3 main()

int main ( )

```
        {
    cout << compute_product(100,100,1) << endl;
    cout << compute_product_old(100,100,1) << endl;
    //logger::start();
    //marked_graph G;
    //ifstream inp("star_graph.txt");
    //inp >> G;
    //graph_message M(G, 10, 2);
    //M.update_messages();
    //graph_test();
    //time_series_compression_test();
    //marked_graph_encoder_test();
    //random_graph_test();
    //logger::stop();
    return 0;
    // vector<vector<int> > list = {{}, {}, {}};

    // b_graph G({{0},{1},{0,1}});
    // // cout << G << endl;
    // // cout << G.nu_left_vertices() << endl;
    // // cout << G.nu_right_vertices() << endl;
    // // cout << G.get_left_degree_sequence() << endl;
    // // cout << G.get_right_degree_sequence() << endl;
    // vector<int> a = G.get_left_degree_sequence();
    // vector<int> b = G.get_right_degree_sequence();


    // b_graph_encoder  E(a,b);
    // mpz_class m = E.encode(G);
    // cout << "encoded: " << m << endl;

    // b_graph_decoder D(a, b);
    // b_graph Ghat = D.decode(m);
    // cout << "decoded graph: " << endl << Ghat << endl;

    // if (Ghat == G)
    //   cout << " equal " << endl;
    // return 0;

}
```

#### 7.28.1.4 marked_graph_encoder_test()

```
void marked_graph_encoder_test ( )
```

```
{
  logger::current_depth++;

  logger::add_entry("Construct G", "");
  marked_graph G;
  //ifstream inp("test_graphs/ten_node.txt"); //("test_graphs/hexagon_diagonal_marked.txt");
  //ifstream inp("test_graphs/problem_4.txt");
  //inp >> G;
  //G = marked_graph(5, {{{0,1}, {0,0}}, {{1,2}, {0,0}}, {{0,3},{0,0}}}, {0,0,0,0,0});
  //int h, delta;
  //cout << " h " << endl;
  //cin >> h;
  //cout << " delta " << endl;
  //cin >> delta;
  G = poisson_graph(100000,3, 10, 10);//
  //G = near_regular_graph(100000,3,1,1);
  //G = marked_ER(100,0.05,1 ,1);
  cout << " graph constructed " << endl;
  //cout << G << endl;

  logger::add_entry("Encode","");

  marked_graph_encoder E(3,20);
  //marked_graph_encoder E(1,20);
  marked_graph_compressed C = E.encode(G);
   FILE* f;
   logger::add_entry("write to binary file", "");
   f = fopen("test.dat", "wb+");
   C.binary_write(f);
   fclose(f);
   //cerr << " graph encoded " << endl;

```

```
  FILE* g;
  g = fopen("test.dat", "rb+");
  marked_graph_compressed Chat;
  logger::add_entry("read from binary file", "");
  Chat.binary_read(g);
  fclose(g);

  if (Chat.star_edges != C.star_edges)
    cerr << " star edges do not match " << endl;

  logger::add_entry("Decode", "");
  marked_graph_decoder D;
  marked_graph Ghat = D.decode(Chat);

  //cout << " Ghat " << endl;
  //cout << Ghat << endl;

  logger::add_entry("compare", "");
  if (Ghat == G)
    cout << " successfully decoded the marked graph :D " << endl;
  else
    cout << " they do not match :(" << endl;

  logger::current_depth--;
  //cout << " G " << endl;
  //cout << G << endl;
  //cout << " Ghat " << endl;
  //cout << Ghat << endl;
}
```

#### 7.28.1.5 operator<<()

```
ostream& operator<< (
            ostream & o,
            const vector< int > & v )
```

```
{
  for (int i=0;i<v.size();i++){
    o << v[i];
    if (i < v.size()-1)
      o << ", ";
  }
  return o;
}
```

#### 7.28.1.6 random_graph_test()

```
void random_graph_test ( )
```

```
{
  marked_graph G = marked_ER(100,1,3,4);
  //cout << G << endl;
}
```

#### 7.28.1.7 time_series_compression_test()

```
void time_series_compression_test ( )
```

```
{
  vector<int> a = {0,2,3,1,2,1,0,1,0,2,1,0,0,2,1,3,4,5,0};
  int n = a.size();
  time_series_encoder E(n);
  pair<vector<int>, mpz_class > ans = E.encode(a);

  time_series_decoder D(n);
  vector<int> ahat = D.decode(ans);
  if (ahat == a)
    cout << " successfully decoded the original time series! " << endl;
}
```

## 7.29 test_mp.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
#include "random_graph.h"
#include "logger.h"
```

### Functions

- ostream & operator<< (ostream &o, const vector< int > &v)
- void random_graph_test ()
- void mp_test ()
- int main ()

### 7.29.1 Function Documentation

#### 7.29.1.1 main()

```
int main ( )
```

```
        {
  logger::start();
  //mp_test();
  random_graph_test();
  logger::stop();
  //random_graph_test ();
  return 0;
}
```

#### 7.29.1.2 mp_test()

void mp_test ( )

```
{
marked_graph G;
ifstream inp("test_graphs/hexagon_diagonal_marked2.txt");
inp >> G;
//G = marked_ER(10000,0.0003,2 ,2);
//G = poisson_graph(100000,3, 10, 10);
//cerr << " graph generated " << endl;
//cerr << " graph generated " << endl;
//cout << " G " << endl << G << endl;
colored_graph C(G, 3, 2);

/*
int n = C.nu_vertices;
for (int v=0; v<n; v++){
  cout << v << " : ";
  for (int i=0;i<C.adj_list[v].size();i++){
    cout << C.adj_list[v][i].first << " ( " << C.adj_list[v][i].second.first << " , " << C.adj_list[v][i].second.second << " ) ";
  }
  cout << endl;
}

cout << " message marks " << endl;
for (int m=0; m<C.M.message_mark.size(); m++){
  cout << m << " mark = " << C.M.message_mark[m] << " star " << C.M.is_star_message[m] << endl;
}
*/

}
```

#### 7.29.1.3 operator<<()

ostream& operator<< (
    ostream & *o*,
    const vector< int > & *v* )

```
{
for (int i=0;i<v.size();i++){
  o << v[i];
  if (i < v.size()-1)
    o << ", ";
}
return o;
}
```

#### 7.29.1.4 random_graph_test()

void random_graph_test ( )

```
{
//marked_graph G = poisson_graph(10, 2, 2, 2);
marked_graph G = near_regular_graph(10,2,1,1);
cout << G;
}
```

## 7.30 time_series_compression.cpp File Reference

```
#include "time_series_compression.h"
```

## 7.31 time_series_compression.h File Reference

```
#include <vector>
#include "bipartite_graph.h"
#include "bipartite_graph_compression.h"
```

### Classes

- class time_series_encoder

    *encodes a time series which is basically an array of arbitrary nonnegative integers*
- class time_series_decoder

    *decodes a time series which is basically an array of arbitrary nonnegative integers*

# Index


__attribute__
    logger.h, 171

a
    b_graph_decoder, 23
    b_graph_encoder, 29
    graph_decoder, 56
    graph_encoder, 63
add
    fenwick_tree, 45
    reverse_fenwick_tree, 129
add_entry
    logger, 84
adj_list
    b_graph, 17
    colored_graph, 41
    marked_graph, 92
alph_size
    time_series_decoder, 132
    time_series_encoder, 136
append_left
    bit_pipe, 32

b
    b_graph_decoder, 23
    b_graph_encoder, 29
b_graph, 11
    adj_list, 17
    b_graph, 12–14
    get_adj_list, 14
    get_left_degree, 15
    get_left_degree_sequence, 15
    get_right_degree, 15
    get_right_degree_sequence, 15
    left_deg_seq, 17
    n, 18
    np, 18
    nu_left_vertices, 16
    nu_right_vertices, 16
    operator!=, 16
    operator<<, 16
    operator==, 17
    right_deg_seq, 18
b_graph_decoder, 18
    a, 23
    b, 23
    b_graph_decoder, 20
    beta, 23
    decode, 20
    decode_interval, 21
    decode_node, 21
    init, 22
    n, 23
    np, 23
    U, 23
    W, 24
    x, 24
b_graph_encoder, 24
    a, 29
    b, 29
    b_graph_encoder, 25
    beta, 29
    compute_N, 26
    encode, 28
    init, 29
    U, 30
b_graph_test
    test.cpp, 187
BIT_INT
    bitstream.h, 148
BYTE_INT
    bitstream.h, 148
beta
    b_graph_decoder, 23
    b_graph_encoder, 29
    graph_decoder, 57
    graph_encoder, 63
bin_inter_code
    obitstream, 124, 125
bin_inter_decode
    ibitstream, 75, 76
binary_read
    marked_graph_compressed, 93, 95
binary_write
    marked_graph_compressed, 98, 101
binomial
    compression_helper.cpp, 149
    compression_helper.h, 158
bipartite_graph.cpp, 139
    operator!=, 139
    operator<<, 139
    operator==, 140
bipartite_graph.h, 140
bipartite_graph_compression.cpp, 141
bipartite_graph_compression.h, 141
bit_pipe, 30
    append_left, 32
    bit_pipe, 31, 32
    bits, 37

chunks, 33
ibitstream, 35
last_bits, 37
obitstream, 35
operator<<, 36
operator>>, 36
operator[], 33
residue, 34
shift_left, 34
shift_right, 34
size, 35
bit_string_read
compression_helper.cpp, 149
compression_helper.h, 158
bit_string_write
compression_helper.cpp, 150
compression_helper.h, 159
bits
bit_pipe, 37
bitstream.cpp, 141
elias_delta_encode, 142, 143
mask_gen, 143
nu_bits, 143
operator<<, 144
operator>>, 144
bitstream.h, 145
BIT_INT, 148
BYTE_INT, 148
elias_delta_encode, 146, 147
mask_gen, 147
nu_bits, 147
buffer
ibitstream, 81
obitstream, 128

C
marked_graph_encoder, 121
chunks
bit_pipe, 33
obitstream, 126
chunks_written
obitstream, 128
clear
marked_graph_compressed, 104
close
ibitstream, 77
obitstream, 126
colored_graph, 37
adj_list, 41
colored_graph, 39
deg, 41
Delta, 42
h, 42
index_in_neighbor, 42
init, 40
is_star_vertex, 42
M, 42
nu_vertices, 42
star_vertices, 43
ver_type, 43
ver_type_dict, 43
ver_type_int, 43
ver_type_list, 43
compressed
marked_graph_encoder, 121
compression_helper.cpp, 148
binomial, 149
bit_string_read, 149
bit_string_write, 150
compute_array_product, 151
compute_product, 151
compute_product_old, 152
compute_product_stack, 153
compute_product_void, 154
prod_factorial, 156
prod_factorial_old, 156
compression_helper.h, 157
binomial, 158
bit_string_read, 158
bit_string_write, 159
compute_array_product, 160
compute_product, 160
compute_product_old, 161
compute_product_stack, 162
compute_product_void, 163
prod_factorial, 165
prod_factorial_old, 165
compute_array_product
compression_helper.cpp, 151
compression_helper.h, 160
compute_N
b_graph_encoder, 26
graph_encoder, 59
compute_product
compression_helper.cpp, 151
compression_helper.h, 160
compute_product_old
compression_helper.cpp, 152
compression_helper.h, 161
compute_product_stack
compression_helper.cpp, 153
compression_helper.h, 162
compute_product_void
compression_helper.cpp, 154
compression_helper.h, 163
current_depth
logger, 86

decode
b_graph_decoder, 20
graph_decoder, 54
marked_graph_decoder, 108
time_series_decoder, 132
decode_interval
b_graph_decoder, 21
graph_decoder, 54
decode_node
b_graph_decoder, 21

graph_decoder, 55
decode_partition_bgraphs
marked_graph_decoder, 109
decode_partition_graphs
marked_graph_decoder, 109
decode_star_edges
marked_graph_decoder, 110
decode_star_vertices
marked_graph_decoder, 110
decode_vertex_types
marked_graph_decoder, 111
Deg
marked_graph_decoder, 112
deg
colored_graph, 41
degree_sequence
graph, 51
Delta
colored_graph, 42
graph_message, 73
delta
marked_graph_compressed, 105
marked_graph_decoder, 112
marked_graph_encoder, 121
depth
log_entry, 83
description
log_entry, 83

edge_compare
marked_graph.cpp, 171
marked_graph.h, 174
edges
marked_graph_decoder, 112
elias_delta_encode
bitstream.cpp, 142, 143
bitstream.h, 146, 147
encode
b_graph_encoder, 28
graph_encoder, 61
marked_graph_encoder, 116, 117
time_series_encoder, 134
encode_partition_bgraphs
marked_graph_encoder, 117
encode_partition_graphs
marked_graph_encoder, 117
encode_star_edges
marked_graph_encoder, 118
encode_star_vertices
marked_graph_encoder, 118
encode_vertex_types
marked_graph_encoder, 119
extract_edge_types
marked_graph_encoder, 119
extract_partition_graphs
marked_graph_encoder, 119

f
ibitstream, 81
obitstream, 128
fenwick.cpp, 166
fenwick.h, 166
fenwick_tree, 44
add, 45
fenwick_tree, 44, 45
size, 46
sum, 46
sums, 46
find_part_deg_orig_index
marked_graph_decoder, 111
find_part_index_deg
marked_graph_encoder, 120
forward_adj_list
graph, 51
freq
time_series_decoder, 132
time_series_encoder, 136
FT
reverse_fenwick_tree, 130

G
time_series_decoder, 133
time_series_encoder, 136
gcomp.cpp, 166
main, 167
get_adj_list
b_graph, 14
get_degree
graph, 49
get_degree_sequence
graph, 49
get_forward_degree
graph, 49
get_forward_list
graph, 50
get_left_degree
b_graph, 15
get_left_degree_sequence
b_graph, 15
get_right_degree
b_graph, 15
get_right_degree_sequence
b_graph, 15
graph, 47
degree_sequence, 51
forward_adj_list, 51
get_degree, 49
get_degree_sequence, 49
get_forward_degree, 49
get_forward_list, 50
graph, 48
n, 52
nu_vertices, 50
operator!=, 50
operator<<, 50
operator==, 51
graph_decoder, 52
a, 56

beta, 57
decode, 54
decode_interval, 54
decode_node, 55
graph_decoder, 53
init, 56
logn2, 57
n, 57
tS, 57
U, 57
x, 57
graph_encoder, 58
a, 63
beta, 63
compute_N, 59
encode, 61
graph_encoder, 59
init, 62
logn2, 63
n, 63
Stilde, 63
U, 63
graph_message, 64
Delta, 73
graph_message, 65
h, 73
is_star_message, 73
message_dict, 73
message_mark, 73
messages, 73
send_message, 66
update_messages, 66
graph_message.cpp, 169
pair_compare, 169
graph_message.h, 169
pair_compare, 170
graph_test
test.cpp, 187

h
colored_graph, 42
graph_message, 73
marked_graph_compressed, 105
marked_graph_decoder, 112
marked_graph_encoder, 121
head_mask
ibitstream, 81
head_place
ibitstream, 82
helper_vars, 9
mpz_vec, 9
mpz_vec2, 9
mul_1, 9
mul_2, 9
return_stack, 10

ibitstream, 74
bin_inter_decode, 75, 76
bit_pipe, 35
buffer, 81
close, 77
f, 81
head_mask, 81
head_place, 82
ibitstream, 75
operator>>, 77
read_bit, 78
read_bits, 79
read_bits_append, 80
read_chunk, 81
index_in_neighbor
colored_graph, 42
marked_graph, 92
init
b_graph_decoder, 22
b_graph_encoder, 29
colored_graph, 40
graph_decoder, 56
graph_encoder, 62
init_alph_size
time_series_encoder, 135
init_freq
time_series_encoder, 135
init_G
time_series_encoder, 136
is_star_message
graph_message, 73
is_star_vertex
colored_graph, 42
marked_graph_decoder, 113
marked_graph_encoder, 122
item_duration
logger, 87
item_last_start
logger, 87
item_start
logger, 84
item_stop
logger, 85

last_bits
bit_pipe, 37
left_deg_seq
b_graph, 17
log
logger, 85
log_entry, 82
depth, 83
description, 83
log_entry, 82
name, 83
sys_t, 83
t, 83
logger, 84
add_entry, 84
current_depth, 86
item_duration, 87
item_last_start, 87

item_start, 84
item_stop, 85
log, 85
logs, 87
report, 87
report_stream, 87
start, 86
stat, 87
stat_stream, 87
stop, 86
verbose, 88
verbose_stream, 88
logger.cpp, 170
logger.h, 170
__attribute__, 171
logn2
graph_decoder, 57
graph_encoder, 63
logs
logger, 87

M
colored_graph, 42
main
gcomp.cpp, 167
rnd_graph.cpp, 183
test.cpp, 187
test_mp.cpp, 190
marked_ER
random_graph.cpp, 176
random_graph.h, 180
marked_graph, 88
adj_list, 92
index_in_neighbor, 92
marked_graph, 89
nu_vertices, 92
operator!=, 90
operator<<, 90
operator==, 91
ver_mark, 92
marked_graph.cpp, 171
edge_compare, 171
operator!=, 172
operator<<, 172
operator>>, 173
operator==, 173
marked_graph.h, 174
edge_compare, 174
operator>>, 175
marked_graph_compressed, 93
binary_read, 93, 95
binary_write, 98, 101
clear, 104
delta, 105
h, 105
n, 105
part_bgraph, 105
part_graph, 105
star_edges, 106
star_vertices, 106
type_mark, 106
ver_type_list, 106
ver_types, 106
marked_graph_compression.cpp, 175
marked_graph_compression.h, 176
marked_graph_decoder, 107
decode, 108
decode_partition_bgraphs, 109
decode_partition_graphs, 109
decode_star_edges, 110
decode_star_vertices, 110
decode_vertex_types, 111
Deg, 112
delta, 112
edges, 112
find_part_deg_orig_index, 111
h, 112
is_star_vertex, 113
marked_graph_decoder, 108
n, 113
origin_index, 113
part_bgraph, 113
part_deg, 113
part_graph, 113
star_vertices, 114
vertex_marks, 114
marked_graph_encoder, 114
C, 121
compressed, 121
delta, 121
encode, 116, 117
encode_partition_bgraphs, 117
encode_partition_graphs, 117
encode_star_edges, 118
encode_star_vertices, 118
encode_vertex_types, 119
extract_edge_types, 119
extract_partition_graphs, 119
find_part_index_deg, 120
h, 121
is_star_vertex, 122
marked_graph_encoder, 116
n, 122
part_bgraph, 122
part_deg, 122
part_graph, 122
part_index, 122
star_vertices, 123
marked_graph_encoder_test
test.cpp, 188
mask_gen
bitstream.cpp, 143
bitstream.h, 147
message_dict
graph_message, 73
message_mark
graph_message, 73

messages
graph_message, 73
mp_test
test_mp.cpp, 190
mpz_vec
helper_vars, 9
mpz_vec2
helper_vars, 9
mul_1
helper_vars, 9
mul_2
helper_vars, 9

n
b_graph, 18
b_graph_decoder, 23
graph, 52
graph_decoder, 57
graph_encoder, 63
marked_graph_compressed, 105
marked_graph_decoder, 113
marked_graph_encoder, 122
time_series_decoder, 133
time_series_encoder, 137
name
log_entry, 83
near_regular_graph
random_graph.cpp, 177
random_graph.h, 180
np
b_graph, 18
b_graph_decoder, 23
nu_bits
bitstream.cpp, 143
bitstream.h, 147
nu_left_vertices
b_graph, 16
nu_right_vertices
b_graph, 16
nu_vertices
colored_graph, 42
graph, 50
marked_graph, 92

obitstream, 123
bin_inter_code, 124, 125
bit_pipe, 35
buffer, 128
chunks, 126
chunks_written, 128
close, 126
f, 128
obitstream, 124
operator<<, 126
write, 127
write_bits, 127
operator!=
b_graph, 16
bipartite_graph.cpp, 139
graph, 50
marked_graph, 90
marked_graph.cpp, 172
simple_graph.cpp, 184
operator<<
b_graph, 16
bipartite_graph.cpp, 139
bit_pipe, 36
bitstream.cpp, 144
graph, 50
marked_graph, 90
marked_graph.cpp, 172
obitstream, 126
simple_graph.cpp, 184
test.cpp, 189
test_mp.cpp, 191
operator>>
bit_pipe, 36
bitstream.cpp, 144
ibitstream, 77
marked_graph.cpp, 173
marked_graph.h, 175
operator()
vint_hash, 137
operator==
b_graph, 17
bipartite_graph.cpp, 140
graph, 51
marked_graph, 91
marked_graph.cpp, 173
simple_graph.cpp, 185
operator[]
bit_pipe, 33
origin_index
marked_graph_decoder, 113

pair_compare
graph_message.cpp, 169
graph_message.h, 170
part_bgraph
marked_graph_compressed, 105
marked_graph_decoder, 113
marked_graph_encoder, 122
part_deg
marked_graph_decoder, 113
marked_graph_encoder, 122
part_graph
marked_graph_compressed, 105
marked_graph_decoder, 113
marked_graph_encoder, 122
part_index
marked_graph_encoder, 122
poisson_graph
random_graph.cpp, 178
random_graph.h, 181
print_usage
rnd_graph.cpp, 183
prod_factorial
compression_helper.cpp, 156

compression_helper.h, 165
prod_factorial_old
compression_helper.cpp, 156
compression_helper.h, 165

README.md, 182
random_graph.cpp, 176
marked_ER, 176
near_regular_graph, 177
poisson_graph, 178
random_graph.h, 179
marked_ER, 180
near_regular_graph, 180
poisson_graph, 181
random_graph_test
test.cpp, 189
test_mp.cpp, 191
read_bit
ibitstream, 78
read_bits
ibitstream, 79
read_bits_append
ibitstream, 80
read_chunk
ibitstream, 81
report
logger, 87
report_stream
logger, 87
residue
bit_pipe, 34
return_stack
helper_vars, 10
reverse_fenwick_tree, 128
add, 129
FT, 130
reverse_fenwick_tree, 129
size, 130
sum, 130
right_deg_seq
b_graph, 18
rnd_graph.cpp, 182
main, 183
print_usage, 183

send_message
graph_message, 66
shift_left
bit_pipe, 34
shift_right
bit_pipe, 34
simple_graph.cpp, 184
operator!=, 184
operator<<, 184
operator==, 185
simple_graph.h, 185
simple_graph_compression.cpp, 186
simple_graph_compression.h, 186
size
bit_pipe, 35
fenwick_tree, 46
reverse_fenwick_tree, 130
star_edges
marked_graph_compressed, 106
star_vertices
colored_graph, 43
marked_graph_compressed, 106
marked_graph_decoder, 114
marked_graph_encoder, 123
start
logger, 86
stat
logger, 87
stat_stream
logger, 87
Stilde
graph_encoder, 63
stop
logger, 86
sum
fenwick_tree, 46
reverse_fenwick_tree, 130
sums
fenwick_tree, 46
sys_t
log_entry, 83

t
log_entry, 83
test.cpp, 186
b_graph_test, 187
graph_test, 187
main, 187
marked_graph_encoder_test, 188
operator<<, 189
random_graph_test, 189
time_series_compression_test, 189
test_mp.cpp, 190
main, 190
mp_test, 190
operator<<, 191
random_graph_test, 191
time_series_compression.cpp, 192
time_series_compression.h, 192
time_series_compression_test
test.cpp, 189
time_series_decoder, 131
alph_size, 132
decode, 132
freq, 132
G, 133
n, 133
time_series_decoder, 132
time_series_encoder, 133
alph_size, 136
encode, 134
freq, 136
G, 136

init_alph_size, 135
init_freq, 135
init_G, 136
n, 137
time_series_encoder, 134
tS
graph_decoder, 57
type_mark
marked_graph_compressed, 106

U
b_graph_decoder, 23
b_graph_encoder, 30
graph_decoder, 57
graph_encoder, 63
update_messages
graph_message, 66

ver_mark
marked_graph, 92
ver_type
colored_graph, 43
ver_type_dict
colored_graph, 43
ver_type_int
colored_graph, 43
ver_type_list
colored_graph, 43
marked_graph_compressed, 106
ver_types
marked_graph_compressed, 106
verbose
logger, 88
verbose_stream
logger, 88
vertex_marks
marked_graph_decoder, 114
vint_hash, 137
operator(), 137

W
b_graph_decoder, 24
write
obitstream, 127
write_bits
obitstream, 127

x
b_graph_decoder, 24
graph_decoder, 57