Marked Graph Compression

Generated by Doxygen 1.8.13

Contents

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 colored_graph Class Reference

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

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

Public Member Functions

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

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

  initializes other variables

Public Attributes

- const marked_graph & G

  the marked graph from which this is created
- int h

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

  the maximum degree threshold
- graph_message M

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

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

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

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

- vector$<$ vector$<$ int $>$ $>$ ver_type

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

- map$<$ vector$<$ int $>$, int $>$ ver_type_dict

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

- vector$<$ vector$<$ int $>$ $>$ ver_type_list

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

- vector$<$ int $>$ ver_type_int

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

3.1.1 Detailed Description

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

quick member overview:

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

Sample Usage

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

3.1.2 Constructor & Destructor Documentation

3.1.2.1 colored_graph()

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

constructor from a graph, depth and maximum degree parameters

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

3.1.3 Member Function Documentation

3.1.3.1 init()

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

initializes other variables

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

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

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

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

```
    }
    ver_type_int[v] = ver_type_dict[ver_type[v]];
  }

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

3.1.4 Member Data Documentation

3.1.4.1 adj_list

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

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

3.1.4.2 adj_location

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

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

3.1.4.3 Delta

`int colored_graph::Delta`

the maximum degree threshold

3.1.4.4 G

`const marked_graph& colored_graph::G`

the marked graph from which this is created

3.1.4.5 h

```
int colored_graph::h
```

the depth up to which look at edge types

3.1.4.6 M

```
graph_message colored_graph::M
```

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

3.1.4.7 nu_vertices

```
int colored_graph::nu_vertices
```

the number of vertices in the graph.

3.1.4.8 ver_type

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

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

3.1.4.9 ver_type_dict

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

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

3.1.4.10 ver_type_int

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

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

3.1.4.11 ver_type_list

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

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

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

- graph_message.h
- graph_message.cpp

3.2 graph_message Class Reference

this class takes care of message passing on marked graphs.

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

Public Member Functions

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

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

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

    update message_dict and message_list

Public Attributes

- const marked_graph & G

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

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

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

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

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

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

3.2.1 Detailed Description

this class takes care of message passing on marked graphs.

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

Sample Usage

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

3.2.2 Constructor & Destructor Documentation

3.2.2.1 graph_message()

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

constructor, given reference to a graph

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

3.2.3 Member Function Documentation

3.2.3.1 update_message_dictionary()

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

update message_dict and message_list

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

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

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

3.2.3.2 update_messages()

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

performs the message passing algorithm and updates the messages array accordingly

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

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

```
{
 int nu_vertices = G.nu_vertices;

 messages.resize(nu_vertices);

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

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

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

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

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

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

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

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

3.2.4 Member Data Documentation

3.2.4.1 Delta

`int graph_message::Delta`

the maximum degree threshold

3.2.4.2 G

`const marked_graph& graph_message::G`

reference to the marked graph for which we do message passing

3.2.4.3 h

`int graph_message::h`

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

3.2.4.4 message_dict

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

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

3.2.4.5 message_list

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

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

3.2.4.6 messages

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

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

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

- graph_message.h
- graph_message.cpp

3.3 marked_graph Class Reference

simple marked graph

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

Public Member Functions

- marked_graph ()

    default constructor
- marked_graph (int n, vector< pair< pair< int, int >, pair< int, int > > > edges, vector< int > vertex_marks)

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

Public Attributes

- int nu_vertices

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

    adj_list[i] is the list of edges connected to vertex i, each of the format (other endpoint, mark towards i, mark towards other endpoint)
- vector< map< int, int > > adj_location

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

    ver_mark[i] is the mark of vertex i

3.3.1 Detailed Description

simple marked graph

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

3.3.2 Constructor & Destructor Documentation

3.3.2.1 marked_graph() [1/2]

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

default constructor

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

3.3.2.2 marked_graph() [2/2]

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

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

Parameters

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

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

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

3.3.3 Member Data Documentation

3.3.3.1 adj_list

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

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

3.3.3.2 adj_location

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

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

3.3.3.3 nu_vertices

```
int marked_graph::nu_vertices
```

number of vertices in the graph

3.3.3.4 ver_mark

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

ver_mark[i] is the mark of vertex i

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

- marked_graph.h
- marked_graph.cpp

Chapter 4

File Documentation

4.1 graph_message.cpp File Reference

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

Functions

- bool pair_compare (const pair< vector< int >, int > &a, const pair< vector< int >, int > &b)
  used for sorting messages

4.1.1 Function Documentation

4.1.1.1 pair_compare()

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

used for sorting messages

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

4.2 graph_message.h File Reference

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

Classes

- class graph_message

    this class takes care of message passing on marked graphs.
- class colored_graph

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

Functions

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

    used for sorting messages

4.2.1 Function Documentation

4.2.1.1 pair_compare()

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

used for sorting messages

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

4.3 marked_graph.cpp File Reference

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

Functions

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

    inputs a marked_graph

4.3.1 Function Documentation

4.3.1.1 operator>>()

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

inputs a marked_graph

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

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

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

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

  G = marked_graph(nu_vertices, edges, ver_marks);

  return inp;
}
```

4.4 marked_graph.h File Reference

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

Classes

- class marked_graph

  simple marked graph

Functions

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

  inputs a marked_graph

4.4.1 Function Documentation

4.4.1.1 operator>>()

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

inputs a marked_graph

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

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

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

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

  G = marked_graph(nu_vertices, edges, ver_marks);

  return inp;
}
```

4.5 test.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <vector>
#include "marked_graph.h"
#include "graph_message.h"
```

Functions

- int main ()

4.5.1 Function Documentation

4.5.1.1 main()

```
int main ( )
```

```
{
  marked_graph G;
  ifstream inp("star_graph.txt");
  inp >> G;
  graph_message M(G, 10, 2);
  M.update_messages();
}
```

Index